Inter-OS Communication on Highly Parallel Multi-Core Architectures

Lamia Youseff, Dmitrii Zagorodnov, and Rich Wolski

Department of Computer Science
University of California, Santa Barbara
{lyouseff, dmitrii, rich}@cs.ucsb.edu

Abstract. The next generation of large-scale machines for scientific computing will consist of a large number of nodes, each supporting many-core and multi-core processor configurations. On one hand, to extract performance from such machines, the operating system will have to be streamlined. On the other hand, the heterogeneity of these machines – i.e., the variety of communication channels and computing elements – will demand a larger set of OS services to achieve scalability. We believe that high-performance OS paravirtualization techniques, where different light-weight OS instances cooperate to provide the different OS services, can resolve this conflict. Current research in paravirtualization aims to isolate OS instances for performance guarantees and security; in a high-performance computing setting, however, thread synchronization and inter-core communication within an application require that the memory is shared and coordinated to meet the application needs. In this paper, we present an approach to enabling efficient memory sharing and synchronization across OS instances running on a multi-core machine. Our performance evaluation shows that this approach allows native execution speeds in a paravirtualized setting, along with communication latencies that are lower than under any existing inter-OS communication mechanism.

1 Introduction

The next generation of large-scale machines for scientific computing will consist of a large number of nodes, each supporting many-core and multi-core processor configurations with heterogeneous communication channels (memory and I/O) and computing elements (GPUs, vector units, etc.). This heterogeneity implies the possibility of different computational features that must be successfully amalgamated by applications that aim to use the full capability of an individual machine. At the same time, new research in the area of operating systems for large-scale architectures indicates that to achieve performance, and to remove "OS noise," operating systems must be made very lean and specifically focused [1, 2, 3]. Thus, at a time when machine heterogeneity is demanding a *broader* set of OS services to achieve scalable performance, the current research indicates that operating systems must become *narrower* and more specialized lest they retard this performance.

We believe that recent developments in high-performance operating system virtualization techniques offer a way out of this quandary as they allow multiple OS instances to share one machine. Current virtualization systems, such as Xen [4] and VMware [5], are specifically designed to *isolate* processes or threads running in different operating systems from each other. With the possibility of 10^7 cores within a single HPC machine, however, both flexibility and scalability requirements make it necessary to be able to support – within the same machine – multiple operating systems that can be used simultaneously by a *single* large-scale application. Such application will consist of "tasks" that exchange messages, but within which different forms of parallelism based on shared memory (e.g., MIMD or SIMD/vector) will be combined.

The goal of our work is to allow different lightweight OS instances, as well as heavier-weight utility operating systems coexisting on the same physical multi-core machine, to cooperate simultaneously, under the control of one application. To that end, we present an approach that enables efficient memory sharing and synchronization across light-weight OS instances, based on paravirtualized OS kernels running on top of the Xen hypervisor [4]. Our performance evaluation shows that it is possible to enable native execution speeds with intelligent memory sharing mechanism. Furthermore, we confirmed experimentally that the latency of communication – which is of paramount importance to high-performance applications – is considerably lower with shared-memory than with any type of socket communication.

The remainder of this paper is organized as follows. The next section describes related work, both to provide context for this paper and to further motivate our approach. In Section 3, we describe our technique, including the implementation details of shared memory across OS instances. We present and analyze the performance of memory sharing in Section 4. In Section 5, we draw conclusions and summarize our future plans.

2 Related Work

Generally speaking, there are three classes of approaches for deploying multiple operating systems on a single machine in a high-end, scalable, HPC setting. They are transparent virtualization, paravirtualization, and microkernels. The approaches differ by the nature of the interface between the operating systems and the underlying *coordination layer*, which ensures safe sharing of hardware resources by the systems.

2.1 Coordinating multiple operating systems

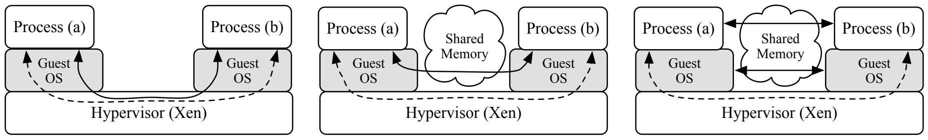
Under *transparent virtualization*, the coordination-layer interface is the Instruction Set Architecture (ISA) – i.e., the hardware-software interface – of some architecture. The instructions from this virtualized ISA may be interpreted purely in software, as done by BOCHS [6] and QEMU [7] emulators, or – when the virtualized ISA matches the ISA of machine's hardware – unprivileged instructions can execute on the hardware directly, through a technique called *full virtualization* (with which only privileged instructions are handled in software). VMware [5, 8, 9] is the best-known example of this approach. The ability to run existing systems out-of-the-box is the key advantage of transparent virtualization; the key disadvantage is the loss of performance due to the interception of privileged operations by the coordination layer.

With *paravirtualization* – a term introduced by one of the first projects to implement this technique [10] – the coordination-layer interface changes somewhat, typically by losing the privileged hardware instructions and gaining the operations for communicating with the coordination layer, which is called a *hypervisor* in this setting. To date, paravirtualized systems have outperformed systems running under transparent virtualization by paying the price of the additional development effort required to port a standard system to a hypervisor. Xen [4, 11] is the most prevalent paravirtualization system in use, at present. Many Linux versions are supported by it and ports for OS X and Solaris are under way. Our own work has investigated the performance of Xen in High Performance Computing (HPC) settings [12, 13]. We found that for many scientific applications, the performance impact associated with using Xen is statistically insignificant. Consequently, paravirtualization is our method of choice.

Under the *microkernel* approach, the interface between the systems and the coordination layer in microkernels moves away from low-level hardware-specific operations to message-passing primitives through which all system components communicate, including the multiple operating systems that could be sharing it. Although a traditional operating system can be ported to run on top of a microkernel, obtaining best performance typically requires significant re-engineering of system's architecture. Mach [14, 15] is perhaps the most successful example of a microkernel system that is capable of supporting a full-featured OS. Generally speaking, the additional flexibility provided by a microkernel implies a performance penalty that is difficult to overcome; nevertheless, careful implementation has been able to address some of these challenges.

2.2 Inter-OS communication in Xen

Based on our previous investigation [12, 13] into paravirtualization in HPC setting, we chose Xen as our starting point. Communication between processes in different guest operating systems under Xen can be achieved in several ways.



(i) Inter-process communication through TCP connection

(ii) Communication through kernel-level shared buffer (e.g., XenSocket and Xway)

(iii) Inter-OS and inter-process communications through memory shared by OS instances

Fig. 1. *This figure shows three types of approaches to establishing communication between processes executing in different OS instances running over Xen. (The solid lines represent the data paths, while the dashed lines represent the control paths.)* ***(i)****Communication between two processes in two OS instances can use a standard TCP/IP connection. All data and control communication go through the hypervisor, hence the high performance overhead. Alternatively* ***(ii)****, a TCP/IP connection can be used for the control channel, while the data passes through a kernel-level shared memory buffer, as done by XenSocket and Xway. In our approach* ***(iii)****, control channel is established through Xen (using one hypercall) but all data communications proceed through kernel-level or user-level buffers. No copies are needed.*

Perhaps the most intuitive communication method is a standard TCP connection. In that case, both control and data pass through Xen hypervisor, as illustrated in Fig. 1 (i), incurring the overhead of data copying, TCP state management, and the necessary negotiation with the hypervisor via *hypercalls* (Xen's equivalent of a system call, allowing the guest OS to invoke a privileged method in the hypervisor). Menon et al. [16] measured this performance penalty using *netperf* [17] TCP streaming benchmark over a single connection. Their study found that Xen guests lag behind by 80% and 67% in the measured maximum transmit and receive TCP streaming throughput, respectively, relative to native Linux performance.

XenSocket [18] offers a socket-like API for communication using shared memory. Bypassing the hypervisor for data copying allows XenSocket to achieve up to 72 times the throughput experienced by regular Xen-based TCP connections. This approach is illustrated in Fig. 1 (ii). XenSocket reduced the number of hypercalls invoked for inter-OS communication to just a few, mainly to discover the other guest OS and establish the shared buffer. Despite the significant reduction in the number of hypercalls, *XenSocket* still lags behind the standard Linux socket communication throughput by 33%. This overhead is caused by the extra memory copies in and out of the shared buffer, which are needed to ensure isolation between the communicating OSs. *Xway* [19] is a project similar to XenSocket. It achieves up to 100 times the throughput of a standard TCP stream by replacing the TCP protocol with a simpler "Xway" protocol. Unfortunately, no performance measurements comparing Xway to native Linux are currently available.

Our research is different from XenSocket and Xway in several ways. The two projects strive to hide the shared memory buffer using the POSIX socket interface; We, on the other hand, expose the shared memory to the programmer. Furthermore, XenSocket and Xway assume that the communicating OS instances can not trust each other. In our model, the different OS instances cooperate to efficiently share and synchronize the memory within an application. Fig. 1 (iii) illustrates our approach, where we limit the interaction with Xen hypervisor to a single hypercall (during shared memory setup). Hence, we minimize the performance penalty. In other words, the isolation restriction between OS instances is not as strict in our model, and relaxing this constraint allows us to reach native performance.

Memory in large-scale machines is certain to be a valuable resource – perhaps more valuable than additional processors. Thus, the current approach to hosting multiple operating systems, which is to partition statically the available memory among the guests, implies too great a fragmentation penalty. Moreover, as recent developments in the Xen community indicate [20], efficient memory sharing is essential for some cross-OS communications. Our work provides one efficient solution to inter-OS memory sharing without imposing significant performance overhead.

3 Methodology

In this section, we describe our approach to enabling communication across OS instances running on a Xen hypervisor. We describe how memory isolation and memory sharing are currently implemented in Xen (version 3.0.4) and what changes we had to make to enable inter-OS instances communication.

3.1 Memory management in Xen

The current approach to implementing memory isolation in Xen relies on the "controller" intercepting all updates to critical virtual memory state in the operating system. In Xen parlance, there is a "master" operating system (referred to as device driver domain, domain 0, or *dom0*), guest operating systems (each referred to as *domU*), and the Xen hypervisor itself (which sits between the hardware and OS domains) [4]. We will adopt this nomenclature for the remainder of this paper. However, while the Xen literature often draws a distinction between dom0 and the hypervisor, it is often convenient to think of them as being co-mingled. That is, dom0 and the hypervisor, together, implement the native virtualization functionality necessary to host one or more domUs.

In this parlance, then, the basic Xen memory isolation mechanism works as follows. Xen is responsible for partitioning the available physical memory among hosted domUs and mapping it into their respective address spaces. To do so, the memory each domU uses to implement its own page tables is marked as *read-only* to that domU. That is, each domU can only read the memory used to implement its own page tables, but Xen has read and write access to this memory. When a domU needs to update its page tables, it must explicitly call Xen through a hypercall so that Xen can check to ensure the memory being updated belongs to that domU (and not some other domU or dom0). The paravirtualization occurs because domU must be explicitly modified to make this call in any place where it would normally just update its page tables by writing memory. (In practice, the Linux kernel uses a a set of C pre-processor macros for all page table entry accesses, which simplifies modifications.)

This approach allows each domU to run at "native" speeds as long as page table entries do not need to be written. However, when page tables are written in a domU (page faults, memory extensions via `sbrk()`, page permission changes, or process creation), the performance penalty is substantial since the domU must make a hypercall to Xen so that the change can be checked. Moreover, since thread creation in Linux uses many of the same kernel mechanisms as process creation, the penalty for this approach in an HPC context can be high.

3.2 Memory sharing in Xen

Baseline implementation of Xen cannot implement memory sharing across domUs easily. Because the standard memory management hardware only supports read-write and read-only permissions, Xen must be interposed between *all* page table entry changes in any domU and it is not possible to share memory regions selectively. To address this difficulty, *grant tables* were recently added to Xen. Grant tables allow different domUs to "grant" access to regions of their own memory to dom0 for the purpose of delivering message data from device drivers. Two data delivery techniques are currently supported by grant tables: shared pages and page flipping.

In using grant tables to grant access to shared pages, the granter domU allocates a new grant reference and fills out its access permissions. Once the grantee domain receives this grant reference, it uses it along with the granter domU domain ID to map the granted frame to its local memory. It does the mapping by calling the *"GNTTABOP_map_grant_ref"* hypercall. Once the memory mapping is established, the granter and grantee domUs can read and write to the shared memory without the intervention of the Xen hypervisor. The domUs can later terminate the page sharing using the hypervisor's hypercall *"GNTTABOP_unmap_grant_ref"*. This simple protocol ensures minimal intervention from the hypervisor, and thus minimal performance overhead since only one hypercall is needed for sharing memory.

Page flipping, on the other hand, transfers ownership of a page frame from the granter domain to the grantee. The ownership transfer is done by having the grantee call the hypervisor through the *"GNTTABOP_transfer"* hypercall after the granter domU has authorized the transfer. Page flipping is efficient for transferring large amount of data between domains, where the overhead of the hypercall is amortized by the throughput of data transfer. Its overhead, however, retards the performance of TCP/IP connections between domains running on the same physical machine, since the package size transferred is too small to amortize that cost. Some projects worked on optimizing page-flipping overhead in Xen. Menon et al [16] experimented with using memory-copying instead of page-flipping and found that it provides lower performance overhead. Menon et al's optimizations are not merged into mainstream Xen, yet.

We modified the implementation of page sharing so that memory can be shared between domUs to enable fast memory sharing in different guest operating systems. The reader familiar with the Xen reference documentation [21] may find our Xen modifications unnecessary, since the reference manual states that grant tables are a generic memory sharing interface between domUs. However, using Xen version 3.0.4 we tried using them to share data outside of the split driver implementation (i.e., between two domUs rather than between a domU and dom0) and found that the current implementation does not allow that. We created a patch for our system to allow arbitrary sharing between domUs and implemented the sharing functionality in two kernels modules: one that grants the shared page, and the other that maps the granted page to the grantee's guest OS virtual memory.

4 Performance Evaluation

Using our modified version of Xen, we are able to compare the performance of memory sharing between threads running under native Linux on separate cores, and between threads running in separate domUs hosted by Xen hypervisor. The purpose of this investigation is to determine if shared memory communication performance is impacted adversely by virtualization.

In each of the experiments we use a two-core, 2.8-GHz Pentium D with an 800-MHz processor bus and 2 MB of L2 cache. The machine memory system uses a 533-MHz bus with 1 GB of dual interleaved DDR2 SDRAM cores. We use our modified Xen v3.0.4 and the 2.6.19 Linux kernel.

4.1 Memory sharing micro-benchmarks

We crafted three simple micro-benchmarks to measure the efficiency of memory sharing. Two of the benchmarks pass control back and forth *(ping-pong)* between two threads of execution for a set number of turns by modifying data in a memory region shared by the threads. The benchmarks were written in C, neither relying on existing

(a)

```
for (i = 0; i < iter; i++) {
        while ( my_id == turn ) { }
        turn = my_id;
}
```

(b)

```
sem[0] = sem_init(0);
sem[1] = sem_init(1);
for (i = 0; i < iter; i++) {
        sem_P(sem[my_id]);
        turn = my_id;
        sem_V(sem[!my_id]);
}
```

Fig. 2. *Simplified C code for* Raw *(a) and* Sync *(b) micro-benchmarks. The same code runs in two threads, with the value of* `my_id` *being the only difference (it is 0 in one thread and 1 in the other). Variable* `turn` *and the semaphore structures in (b) are in shared memory.*

synchronization libraries or hardware support for synchronization, which makes them usable without modifications both in user-space and inside the kernel. Both use busy waiting (instead of proper sleep/wakeup) to minimize context switches.

In the first benchmark – termed *Raw* – control is passed using a single shared binary variable, as shown in Fig. 2 (a)[1]. This experiment measures the raw speed of memory sharing. The second benchmark – denoted *Sync* – uses classic P/V semaphores implemented using *bakery algorithm* [22] to enforce synchronized (and necessarily alternating) access to a shared integer variable, which is modified at each turn, as shown in Fig. 2 (b). *Sync* benchmark demonstrates the performance of a more realistic fine-grained synchronization scenario.

In addition to the two shared-memory benchmarks, we wrote a simple program to ping-pong an integer through a TCP socket by alternating `send()` and `recv()` invocations in each thread. Our intent was to measure the overhead of TCP connection management and Xen protection mechanisms.

4.2 Performance results

We used our micro-benchmarks to measure the latency of communication between two threads as follows. For each benchmark run, the threads were ping-ponging for 10^6 times. We then computed the mean duration of an iteration (along with its standard deviation).

For each benchmark, we studied three cases. The first is memory sharing between processes executing under native Linux, the second is memory sharing between dom0 and a domU, and the third is memory sharing between two domUs. In all cases we enable Linux processor affinity to minimize cache and TLB pollution effects. Finally, we were concerned about the possibility that enabling support for SMP threading in the dom0 kernel might perturb the results (all domUs in this study were not SMP-enabled). Thus we conducted the experiments both for the native dom0 without SMP support and with it enabled. Table 1 summarizes the results.

In the table, each row corresponds to a benchmark, with Sync appearing twice (Sync$_{1k}$ involved passing one thousand bytes from one thread to the other instead of a single integer) and the socket-based ping-pong results listed as *TCP*. The columns are divided into three sections: one for native Linux with SMP enabled (left-most section),

[1] A diligent reader may notice that the code does not force strict alternation since it is possible for one of the threads to loop around twice as the other only loops around once; by keeping track of iterations made we know that in practice alternation is the norm.

	Native	**dom0 w/o SMP**		**dom0 w/ SMP**		
	Linux	dom0-domU	domU-domU	dom0-dom0	dom0-domU	domU-domU
Raw	0.49 (0.03)	0.36 (0.007)	0.36 (0.005)	0.36 (0.009)	0.35 (0.005)	0.39 (0.03)
Sync	1.4 (0.02)	0.8 (0.01)	0.8 (0.02)	0.8 (0.01)	0.78 (0.03)	0.8 (0.02)
Sync$_{1k}$	3.4 (0.07)	3.3 (0.05)	3.3 (1.14)	2.8 (0.15)	3.1 (0.2)	2.9 (0.1)
TCP	52.6 (0.4)	56.7 (0.2)	80.9 (0.3)	38.4 (0.3)	70.3 (0.4)	104.0 (0.5)

Table 1. *Latency of communication under* Raw*,* Sync*, and* TCP *micro-benchmarks. The units are microseconds, each number not in parentheses is the average over* 10^6 *iterations of* 25 *runs and the parenthesized number is the standard deviation of the* 25 *runs.*

one for Xen dom0 with SMP disabled (middle section) and one with Xen dom0 with SMP enabled (right-most section). In the middle section, reading from left to right, the values correspond to dom0-domU sharing and domU-domU sharing, respectively. In the right section, also reading from left to right, the values correspond to dom0-dom0 sharing, dom0-domU sharing, and domU-domU sharing, respectively. Each cell of the table shows the average time of 25 runs of the benchmark in microseconds, each run completing 10^6 iterations. The sample standard deviation over the 25 runs is shown in parenthesis. For example, the first cell in the first row contains the values $0.49(0.03)$, which indicates that for the Raw benchmark, for 25 runs, the average elapsed time per iteration (one ping-pong) is 0.49 microseconds with a standard deviation of 0.03.

From the table, we can see that memory sharing via modified grant tables under Xen (first three rows) proceeds at native speeds. It may appear that, in fact, Xen is faster (the first element in the first column is larger than the others). However, in this case, the memory sharing had to be between user-level processes (since the kernel is not multi-threaded). We included this test as a control of our measurement infrastructure as we would expect user-space to user-space transfers to be more expensive. In all other cases shown in the first three rows, however, the transfers are kernel-to-kernel and the data indicates that the speeds are the same. More rigorously, comparisons of the means using a t-test for all but the first value in the first row provides no evidence contradicting the assertion that the means are the same.

The values in the bottom two rows of the table allow us to speculate on how shared-memory communication compares to the two socket-based approaches discussed in section 2. Specifically, Sync$_{1k}$ results are comparable to one data point reported for XenSocket [18] in which one thousand bytes were being transferred from one DomU to another. The bandwidth reported was 2250 Mbps, which corresponds to 3.4 microseconds per iteration involving transfer of 1000 bytes. This number is the same as the user-space number for Sync$_{1k}$ (left-most column), confirming that bulk data transfers are equally efficient with shared memory and message passing.

When it comes to latency for fine-grained synchronization, however, shared-memory programs perform better, as our TCP experiment (in the last row of Table 1) illustrates. Ping-pong in shared memory is at least 50 faster than a socket ping-pong. And the more domUs are involved the worse the performance. This degradation is mainly caused by the network layer implementation in Xen, in which a page-flipping technique – as explained in section 3.2 – is used to avoid memory copying.

These simple experiments indicate that it is possible to achieve memory sharing among OS instances at native execution speeds for multi-core systems using paravirtualization. They also indicate how existing Xen functionality might be adapted to support high-performance multi-kernels. They do not, however, demonstrate how memory management can be unified between hosted operating systems. In this case, we pinned the memory shared between threads so that Linux memory scheduling would not inadvertently interfere with the results. Moreover, in the cases with Xen, memory is partitioned, and there are as many as *three* memory schedulers active simultaneously – one in each domU and one in dom0. These schedulers are unaware of each other and uncoordinated, which could lead to thrashing if a single memory-intensive application were running in the hosted systems.

5 Conclusion and Future Work

This paper represents our first step towards developing support for efficient coordination and management of light-weight OS instances on parallel multi-core architectures. Our approach leverages recent developments in paravirtualization, particularly the Xen hypervisor, to enable multiple OS instances to share one machine. Although memory sharing among OS instances is not allowed in the standard Xen distribution, a small patch can enable such sharing. We used micro-benchmarks to study the overhead of our approach and to compare it to related (socket-based) approaches. Our results show that memory sharing can be achieved at the speed of the system without paravirtualization. In particular, our measurements agree with the bulk message-passing throughput reported for XenSocket (apparently on a similar hardware configuration). Furthermore, we confirmed that for fine-grained synchronization, which we consider crucial to maximizing performance of peta-scale applications, the latency is considerably lower with shared-memory than with message passing.

In this paper, we presented an approach to enabling efficient memory sharing and synchronization across OS instances running on a multicore machine. Our future plans are to continue investigating how to preserve this level of performance (perhaps through further modifications to the Xen grant tables mechanism) while eliminating the memory partitioning and the redundant – and possibly conflicting – memory schedulers. The next step will be to understand if the mechanism can be used to support full address space sharing and to develop a new mechanism if the generality of the current paravirtualization techniques cannot be sufficiently adapted. Further in the future, we plan to develop other mechanisms necessary for deploying one application across multiple OSs, such as inter-OS thread scheduling and delegation of I/O handling from one OS to another.

References

[1] Minnich, R., Sottile, M., Choi, S.E., Hendriks, E., McKie, J.: Right-weight kernels: an off-the-shelf alternative to custom light-weight kernels. SIGOPS Oper. Syst. Rev. **40**(2) (2006) 22–28

[2] Beckman, P., Iskra, K., Yoshii, K., Coghlan, S.: Operating system issues for petascale systems. SIGOPS Oper. Syst. Rev. **40**(2) (2006) 22–28

[3] Ong, H., Vetter, J., Studham, R.S., McCurdy, C., Walker, B., Cox, A.: Kernel-level single system image for petascale computing. SIGOPS Oper. Syst. Rev. **40**(2) (2006) 50–54

[4] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R.: Virtual machine monitors: Xen and the art of virtualization. In: Symposium on Operating systems principles (SOSP). (2003) `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/`.

[5] VMWare, I.: VMWare home page (2007) `http://www.vmware.com`.

[6] Lawton, K., Guarneri, B.D.N., Ruppert, V., Bothamy, C., Calabrese, M.: Bochs x86 PC emulator Users Manual (2003) `http://bochs.sourceforge.net/`.

[7] Bellard, F.: QEMU home page (2007) `http://fabrice.bellard.free.fr/qemu/`.

[8] J. Sugerman and G. Venkitachalam and B. Lim: Virtualizing I/O devices on VMware workstations hosted virtual machine monitor. In: USENIX Annual Technical Conference. (2001)

[9] Rosenblum, M., Garfinkel, T.: Virtual machine monitors: Current technology and future trends. Computer **38**(5) (2005) 39–47

[10] Whitaker, A., Shaw, M., Gribble, S.: Scale and performance in the Denali isolation kernel. In: Symposium on Operating Systems Design and Implementation (OSDI). (2002) `http://denali.cs.washington.edu/`.

[11] Clark, B., Deshane, T., Dow, E., Evanchik, S., Finlayson, M., Herne, J., Matthews, J.N.: Xen and the art of repeated research. In: USENIX Annual Technical Conference, FREENIX Track. (2004) 135–144

[12] Youseff, L., R. Wolski, B. Gorda, C.K.: Paravirtualization for hpc systems. In: Proceedings of Workshop on XEN in HPC Cluster and Grid Computing Environments (XHPC) *best paper award winner*. (2006)

[13] Youseff, L., Wolski, R., Gorda, B., Krintz, C.: Evaluating the performance impact of xen on mpi and process execution for hpc systems. In: Proceedings of the First International Workshop on Virtualization Technology in Distributed Computing (VTDC). (2006)

[14] Rashid, R., Julin, D., Orr, D., Sanzi, R., Baron, R., Forin, A., Golub, D., Jones, M.: Mach: A System Software Kernel. In: Computer Society International Conference. (1989)

[15] Black, D., Golub, G., Julin, D., Rashid, R., Draves, R., Dean, R., Forin, A., Barrera, J., Tokuda, H., Malan, G., Bohman, D.: Microkernel Operating System Architecture and Mach. In: Workshop on Micro-Kernels and Other Kernel Architectures. (1992)

[16] Menon, A., Cox, A.L., Zwaenepoel, W.: Optimizing network virtualization in xen. In: USENIX Annual Technical Conference. (2006) 15–28

[17] Jones, R.: `http://www.netperf.org/netperf/training/netperf.html` (2003) Netperf: a network performance Benchmark. Revision 2.0.

[18] Zhang, X., McIntosh, S., Rohatgi, P., Griffin, J.L.: Xensocket: A high-throughput interdomain transport for vms. Technical report, IBM Research Technical Report RC24247 (2007)

[19] Team, T.X.: Xway: Lightweight communication between domains in a single machine (2007) `http://sourceforge.net/project/platformdownload.php?group_id=191553`.

[20] Ben-Yehuda, M., Mason, J., Xenidis, J., Krieger, O., Doorn, L.V., Nakajima, J., Mallick, A., Wahlig, E.: Utilizing iommus for virtualization in linux and xen. In: Proceedings of the 2006 Linux Symposium. (2006) `http://www.linuxsymposium.org/2007/`.

[21] Team, T.X.: Xen V3.0 for x86 Interface Manual (2005) `http://www.cl.cam.ac.uk/research/srg/netos/xen/readmes/interface.pdf`.

[22] Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Communications of the ACM **17**(8) (1974) 453–455