Improving Performance by Embedding HPC Applications in Lightweight Xen Domains

Samuel Thibault, Tim Deegan

Improving Performance by Embedding HPC Applications in Lightweight Xen Domains

Samuel Thibault
XenSource, Cambridge
samuel.thibault@eu.citrix.com

Tim Deegan
XenSource, Cambridge
tim.deegan@eu.citrix.com

Abstract

Although they allow easy and cost-effective use of a wide range of machines, the programming interface and behavior of general-purpose Operating Systems (OS) often fail to meet, or even conflict with, the specific desires of High-Performance Computing (HPC) applications, such as low preemption or control over memory and I/O management. That often leads to poor performance. On the other hand, hypervisors are more and more commonly used on top of those OSes for various reasons, such as ease of dedicated environment deployment or load balancing. In contrast to the usual unix process model, hypervisors provide their guests with kernel-level facilities. In this paper, we show how an HPC application and its execution environment can be embedded within a lightweight guest domain, alongside a domain that runs a conventional OS which is only used for administrative purpose. That permits the execution environment to take advantage of kernel-level facilities to improve performance, which would be hard to achieve in the traditional process model because of lack of support or excessive overhead.

Keywords Virtualization, Microkernels, HPC, Memory management, Scheduling, I/O.

1. Introduction

The expected characteristics of a general-purpose Operating System (OS), such as the responsiveness of the user interface or the delaying of I/O operations, often mismatch or even conflict with the desires of HPC applications. Indeed, the latter typically use e.g. cache-oblivious BLAS routines which often perform very badly under a time-shared scheduling policy (some people even prefer to patch their kernel so as to almost completely disable preemption (13)!). For out-of-core applications, I/O could also be far better scheduled by the application itself (20). As a consequence, execution environments for HPC have to bypass OS algorithms, for instance by pinning threads on processors, locking memory, or using O_DIRECT operations.

One of the goals of microkernels is precisely to have minimum impact on applications and to delegate such resource management as much as possible. Their availability in HPC centers is however very low, and the preference is for general-purpose OSes, for various reasons such as support of the hardware or the availability of HPC tools.

However, virtualization solutions are more and more being considered in HPC centers (such as the French CEA) for several reasons, including the ability to migrate applications between physical machines or to easily deploy separate personal execution environments (and thus avoid *e.g.* library conflicts). That is actually an opportunity for *new execution models* for HPC. As Mergen *et al.* have foreseen (18), and as shown in figure 1, it becomes possible to run a hypervisor on top of a general-purpose OS, and then run the HPC application in its own virtualized domain, along with a specialized execution environment and kernel (often called library OS). As a result, the general-purpose OS is only used for supporting the hardware and administrative tasks, and it does not interfere with the HPC application during the computation. The hypervisor does, but that interference is very low: the scheduler of a hypervisor is usually very conservative for instance. In addition, the control that the application and the execution environment may have over the specialized kernel can be arbitrarily strong, opening the way to all sorts of memory, CPU, or I/O scheduling optimizations well adapted to the particular application. This is usually not possible in traditional OSes because such control requires kernel-level privileges.

In this paper, we present an actual implementation of such an execution model on top of the Xen (1) Hypervisor. For Xen-related purposes, we used the sample para-virtualized kernel 'Mini-OS', to which we added a POSIX interface thanks to the newlib C library and lwIP IP stack so as to run *stub* domains for various services related to Xen (hardware emulation, network services, ...). In this paper, we show

2nd Workshop on System-level Virtualization for High Performance Computing (HPCVIRT'08) 31st March, Glasgow, Scotland

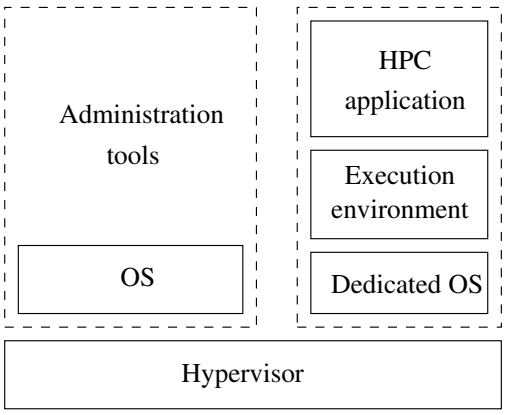

Figure 1. Running an HPC application in a dedicated domain under the control of a hypervisor.

how this work may be re-used for HPC and we describe various kinds of optimizations which become possible in such an environment. Through our application example, we both show how the reduced kernel call overhead brings performance benefits, and explain some optimizations which we were able to perform.

2. Related Work

Reducing kernel impact and allowing applications to handle scheduling, memory, and I/O management have been some of the key features of microkernels. Examples include Nemesis (21), the Exokernel (8) and the ExOS, K42 (5), and L4 (16). The amount of mechanisms implemented in these microkernels varies somewhat, the extreme cases being L4 and Nemesis, which are incredibly small (about 15 thousands lines of code for L4) and only handle very basic operations. Micro kernels often let applications schedule their own threads themselves with the help of Scheduler Activations (4) so as to better handle blocking system calls. They also delegate memory traps so as to let applications handle paging themselves (20), and provide very efficient inter-domain communications, typically based on RPC with IDLs. Engler *et al.* explain (8) that building a dedicated execution environment on top of such delegations can give far better performance and that resources should be safely provided instead of being emulated like in the case of the traditional process model. As a result, microkernels would be a target of choice for HPC execution environments. Unfortunately, from a pragmatic point of view, general-purpose OSes have much better support for modern hardware and can be more easily installed by administrators. As a result, on actual HPC machines microkernels are generally not available, and the use of a hypervisor (which is sometimes considered as 'MicroKernel Done Right' (10)) turns out to be a much more viable solution, benefitting from the hardware support of general-purpose OSes.

There are a few exceptions, however. The most notable one is probably the Operating System of the computing nodes of Blue Gene/L machines. While communication nodes are mere PowerPC processors running Linux, computation nodes are specialized processors which execute CNK (19) (Compute Node Kernel), a small kernel that provides applications with a flat fixed-size memory area without paging support (typically 512 MB). The kernel and the application are only separated by a privilege level. The I/O operations are delegated to the communication node, so that computation nodes only handle computations. The programming environment is the usual GNU toolchain which was ported to that special execution environment. This is however a quite special case developed for such particular machines, and thus not adapted to the typical HPC machines which, as the top500 shows (2), are often just clusters of standard PCs.

Several projects use a hypervisor to isolate the actual application from the general-purpose OS so as to keep an easy administration of the machine through that OS, and still get good performance or security for the application. The Libra (3) and JavaGuest (14) projects both aim at providing 'Java on silicon', i.e. to execute the JVM in an environment as close to the metal as possible. By using very simple virtualized kernels, they avoid all the complicated semantics of a full-featured kernel, and hence permit far easier certification of the semantics of the JVM. The virtual memory model of the JVM, for instance, can be implemented directly using the basic hypervisor memory management primitives, which boils down to handling a page table, and memory traps are directly handled by the JVM. By avoiding all the OS layers, the implementation is a lot faster. Of course, these projects only target the Java language in particular.

The Catamount (15) kernel is a very lightweight kernel for the Cray XT3 system, designed to have small overhead for parallel computing environments. It uses a very pruned version of the GNU libc that provides a POSIX environment for the applications. However, similarly to CNK, it provides almost no support for advanced features like threads and the mmap() function, and thus does not allow developers to easily port existing applications to it.

The Library OS (17) is very similar to our work: it uses Xen as a hypervisor and Xen's Mini-OS as a basic kernel. It links with the latter some libraries including the newlib C library and an efficient Inter-Domain Communication mechanism (IDC) based on the shared memory feature of Xen. However, the target is security rather than performance of the application itself: this environment supplies very good isolation for the use of *e.g.* TPM devices (Trusted Platform Module). In this paper, we use the same basic approach, but we target the performance of HPC applications.

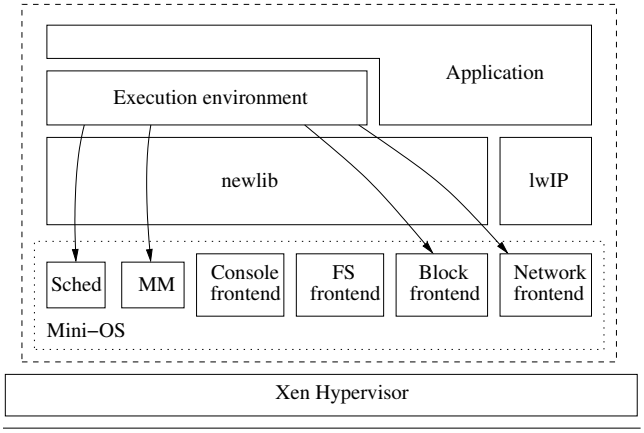

Figure 2. Structure of our lightweight guest domain.

3. A not so mini domain

In this section we describe how we embed an application within a lightweight Xen domain, on top of the Mini-OS kernel, the newlib C library, and the lwIP IP stack.

3.1 Mini-OS

Mini-OS is shipped in Xen as a sample para-virtualized guest for the Xen Hypervisor (6). It is meant to be very simple and it completely relies on the hypervisor to access the machine: it just uses the Xen network, block, and console frontend/backend mechanisms. It only supports non-preemptive threads, and supports only one virtual memory address space (no user space). It can access part of the file system of the general-purpose domain (dom0) through the FileSystem frontend/backend mechanism developed for the JavaGuest project (14).

3.2 A POSIX environment on top of Mini-OS

The design of our POSIX-compatible lightweight execution environment on top of a Mini-OS guest domain is shown in figure 2. Briefly, the idea is to just link together the code of the Mini-OS kernel, a C library, an IP stack, the execution environment, and the application in a single executable. Technically speaking, using the GNU tools we setup a plain ELF cross-compilation chain, i.e. we build a cross-compilation version of `binutils` and `gcc` with plain ELF as target, so as to avoid any specifics of the OS which is used to build our environment (the Thread-Local Storage model for instance). We then cross-compile some libraries with the same kernel C flags as for the Mini-OS compilation, and install them in the cross-compilation environment. To produce an embedded version of an application and its execution environment, we then just need to cross-compile them and link them with Mini-OS and the required libraries.

The POSIX interface is provided in several parts by newlib, lwIP, and some additional code. LwIP provides a lightweight TCP/IP stack which we just connect to the network frontend of Mini-OS. Newlib provides the standard C library functions. Another choice could have been the GNU libc, but newlib is meant for embedded projects and thus better suits our goal. The GNU libc would also have needed some porting work, while newlib just depends on a very small range of basic system functions like `sbrk`. These system functions and the Unix part of POSIX then have to be implemented on top of Mini-OS. Some functions are very trivial to implement: since we do not have the notion of Unix process, `getpid` and similar can just return *e.g.* 1. As we do not have signals either, `sig` functions can just be void. Other functions can be very easily implemented on top of Mini-OS: `sleep` and `gettimeofday` just require careful interfacing. `mmap` is only implemented for one case: anonymous memory, for which we just have Mini-OS project zeroes on a newly allocated address space.

The tricky part resides in the file-related functions. Since we want to have all of the console, main domain files (through the FileSystem frontend), block devices, network devices, and lwIP TCP/IP connections to work in a POSIX way, we have to multiplex among the corresponding parts of Mini-OS and lwIP. We therefore implemented a thin Virtual File layer, which for each file descriptor records the kind of file, and then all file-related functions are redirected to the appropriate low-level functions. The most tricky function is `select`, for which all kinds of file descriptors have to be mixed. The approach is to first register the caller thread on wait queues corresponding to those descriptors, then call the poll function for each of them, and if nothing comes out, block the thread. On wake up, we poll all file descriptors again and return the result.

The resulting lightweight environment is actually quite complete and we were even able to launch the Caml runtime with it. One of the key advantages of this approach is that, like Unix distributions, it assembles pieces of software which are well-maintained in other respects. Mini-OS, as the sample guest for Xen, is always kept up to date with the hypervisor interface, and newlib and lwIP are well maintained for other embedded projects. The hard work is thus to interface these existing implementations together.

4. Optimization opportunities for HPC

HPC applications often use an execution environment which acts as an adapted layer between the application and the Operating System, so as to provide advanced tools to the application, for instance computation, thread, or communication libraries. These environments are often dedicated to the particular purpose they serve, and as a consequence have quite a good idea of how resources should be used: how tasks should be scheduled, how memory or I/O management should be performed, etc. With traditional Operating Systems, these environments usually have quite a hard time interacting with the kernel so as to bypass the general-purpose algorithms that it implements, by pinning the kernel threads on processors, by locking, or by write-protecting memory

so as to manage memory from the user space, by using the non-standard O_DIRECT extension in order to control the actually performed I/Os, etc (12).

In a lightweight guest domain environment like ours, the situation is quite different. Since the execution environment is directly linked with the kernel, interaction between the two has very low overhead (it is reduced to a mere function call), and as shown in figure 2, there is no such barrier as a system call interface, the environment can interact with all interesting components of the kernel in whatever way it prefers. In the following subsections, we study the cases of scheduling, memory management, and disk Input/Output.

4.1 Scheduling

Mini-OS provides threads, but they are non-preemptive and there is support for only one processor. This actually meets the requirements of HPC components like BLAS routines quite well: they can run at full speed without seeing the cache being polluted by other tasks which might get woken up during the computation. The default scheduler is very simple: it uses a single runqueue without any priorities, and therefore it has very little overhead.

On the other hand, since the execution environment is directly linked with the kernel, it is easy to have the former provide its own scheduler, dedicated to the application, to be used by Mini-OS instead of the default trivial one.

The uniprocessor nature of the Mini-OS scheduler actually permits Xen to have very good control over the CPU usage. Indeed, with only one processor to manage, Mini-OS does not need to use techniques such as spinning, which would typically uselessly consume CPU while the hypervisor might have other domains to schedule.

Also, the scheduler interface of Xen itself is quite simple and already has a plug-in interface. As future work, it would be interesting to delegate the domain scheduling decisions to an advanced scheduler in a lightweight guest domain, which would permit to implement heuristics more easily, since that application may be developed and debugged in a standard OS first, and then just moved to a lightweight domain.

4.2 Memory management

The principle of memory management with Xen is very simple: domains are given a list of machine pages which they are allowed to use. Then they can build page tables in the way they prefer, provided that they only map the machine pages they are allowed to. The updates hence have to be validated by the hypervisor, but this can be efficiently run in batch. Memory traps are delivered to the domain the same way they would be on a genuine machine. This is a great opportunity for execution environment to actually have true control over the use of memory, just like a kernel usually has.

For instance, HPC often uses sparse data (11), in which whole parts can be zero for instance. In our environment, this can be achieved with very little cost by just changing the virtual memory mappings. The Copy on Write mechanism can then be used to have the sparse areas dynamically change according to updates. This behavior is of course exactly the same as what is achieved in a traditional Unix system when mapping anonymous pages. However, in our environment the overhead of memory maps is much lower since it reduces to a series of function calls and one call to the hypervisor to apply all page updates. Also, there is no arbitrary limit on the data that can be projected: identity matrix blocks could be projected the same way, copying matrix blocks could be performed through Copy on Write, etc. Doing so with a standard Unix environment would require first writing the data in a file, then mapping it, etc. which has much higher overhead.

Another interesting possibility is to exploit the Accessed/Dirty bits of page table entries, which track at no cost reads and writes in the pages. These are usually not available to Unix processes. In our environment, since the application has access to the actual data page tables thanks to the para-virtualization nature of Xen, reading these bits is as cheap as a memory dereference! This opens possibilities like various heuristics based on costless data access pattern detection, and also permits to implement an application-level DSM in a far more efficient way than when using the Unix process model.

On IA-64 machines the memory management model is not limited to a mere page table, but can use fixed TLB entries, hash tables, and trap-provided TLB entries. These could be usefully exploited by HPC execution environments.

4.3 Disk Input/Output

Since they have to go through dom0, disk Input/Output operations would *a priori* be slowed down. Actually, measurements show that the para-virtualization interface achieves quite good performance, very close to native Linux (the overhead is about 1%, as detailed in (6) and verified in (7)).

A possible benefit from our lightweight environment is that like the non-standard O_DIRECT interface of some Unix systems, the I/O operations are not buffered, they are directly enqueued to be completed as soon as possible. That means that the execution environment of an out-of-core application can exactly control the disk accesses and thus optimize requests.

5. Xen use case: HVM *stub* domains

Our lightweight guest domain environment was not originally intended for HPC and hence our use case is not HPC-related, but it still demonstrates the kind of improvements that can be expected, and it illustrates the optimization possibilities that have been described in the previous section.

Thanks to recent processor technologies (VMX and SVM), Xen 3 is able to run unmodified guests in fully-virtualized domains (also known as HVM domains, Hardware Virtual Machine domains) by using a new ‘VM mon-

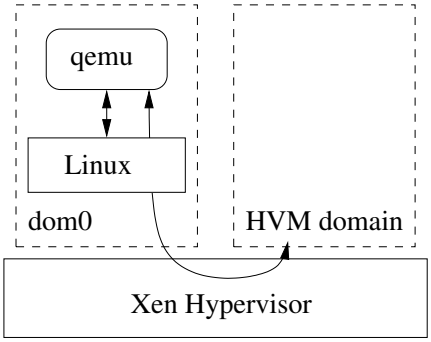

Figure 3. Traditional Xen approach: qemu, the device emulation component, runs as a process in dom0.

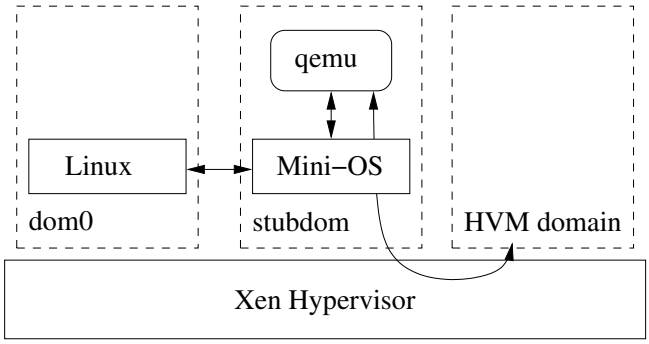

Figure 4. Stubdomain approach: qemu runs on top of Mini-OS in a specialized domain.

itor' level above the kernel level. The guest can hence run natively on the processor even with kernel privileges. However, when it performs operations which need the attention of the hypervisor, *e.g.* access to I/O ports, control is returned to the hypervisor so as to take appropriate actions. In the case of accesses to I/O ports, Xen has to emulate virtual devices, for the guest to believe it is running on a real machine. This is achieved by running a `qemu` process[1] in the dom0 administration domain, as shown in figure 3. However, emulation of virtual devices sometimes takes non-negligible CPU time, and this prevents correct accounting for *e.g.* scheduling quotas. Also, the latency is not so good: on an I/O operation, the Xen hypervisor immediately schedules dom0, but then the Linux scheduling policies come into play, and the hypervisor has no control over that. If dom0 is loaded with other tasks, the responsiveness can be very poor.

In a way similar to the *sidecore* approach described by Gavrilovska *et al.* (9), by moving the `qemu` application into a *stub* domain, as shown in figure 4, the hypervisor has better control over scheduling and accounting, since the Mini-OS scheduler is very simple and basically only runs the `qemu` application. Figure 5 shows the various performance measurements we could obtain with qemu running in dom0 or

[1] `qemu` is a complete machine emulator, of which Xen just uses the virtual device emulation part.

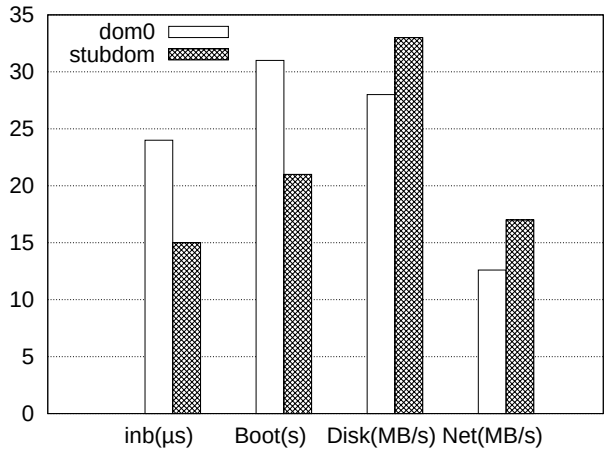

Figure 5. Performance of the fully virtualized guest, when qemu is running as a process in dom0, and when it is running in a *stub* domain: latency of guest I/O port access, guest boot time, throughput of disk and network.

in a *stub* domain. The `inb` bars show the latency of I/O port accesses as seen from the HVM domain, *i.e.* the round trip time between the application in the HVM domain and the virtual device emulation part of `qemu` (dom0 is not involved at all here, only the hypervisor and the *stub* domain are). These two bars show a dramatic reduction between the dom0 and the *stub* domain approaches. A closer analysis of the decomposition of the latency showed that most of the time was spent in the main loop of qemu, in which the overhead is due to calls which check for events. In the case of a *stub* domain, that is reduced a lot since there is no system call barrier, just straight forward function calls. The same kind of improvement can be seen for the boot time of a Linux kernel, shown in the second pair of bars, since booting a kernel in an HVM domain typically requires a lot of I/O operations. Lastly, the throughput achieved by the virtual disks and network boards emulated by `qemu` are improved quite well (between 20 and 30 %)[2]. Of course, an interaction with dom0 is still needed in order to perform the actual communications, but the main bottleneck is the interaction between the device drivers in the HVM domain and the virtual devices of `qemu`, since for each communication operation the guest has to perform a lot of basic I/O port operations.

Thanks to our lightweight environment, some parts of `qemu` can be optimized. For instance, for the 65 536 I/O ports of the `x86` architecture, qemu just uses a big table of handlers, initially filled with the address of a default handler. In order to save memory, it could instead be filled with `NULL` pointers which would be checked for, and the default handler be called in such a case. Allocating the table through

[2] Of course, the use of para-virtualized drivers gives far better results, but this typically can not be used during the installation of the guest for instance, and thus efficient full virtualization is essential too.

an anonymous `mmap` call would allow us to exploit Copy on Write to actually use a sparse table. However, a better approach is to fill one page with the address of the default handler, and then to project that page in the whole table, and enable Copy on Write for that page. This way, checking for `NULL` is not even needed as the address of the default handler address is efficiently exposed.

In addition to that, we plan to exploit the Dirty bits of page tables so as to more efficiently track the parts of the virtual screen that are updated by the HVM domain and thus need a refresh on the screen of the user, instead of the current approach which is using a `memcmp` with a shadow copy of the video memory.

6. Future Work

The few features of Mini-OS already present optimization opportunities for HPC. Some new features could bring yet more possibilities.

The use of super pages (2-4MB) can dramatically reduce TLB misses, but the usual issue is to decide which parts of the memory should use them, because kernels usually have few clues about the actual usage of application memory. The interface provided by standard Operating Systems to let applications use super pages is often clumsy, if at all available. By adding the support to Xen and Mini-OS, the execution environment of the application would be able to easily and efficiently use them, thus improving performance.

In the case of an MPI application for instance, it would be interesting to have the MPI execution environment use the Inter-Domain Communication facility of the Library OS (17), so as to optimize the intra-machine communications between MPI instances.

In order to take full benefit from hi-speed network cards, the use of VT-d technology would let Mini-OS (and thus the application itself) push packets directly to the network and hence get better performance since that completely skips interactions with dom0. Cisco is already considering our environment as a potential target for their IOS Operating System.

7. Conclusion

The performance of HPC applications executed by general-purpose Operating Systems are often poor just because the algorithms of the latter are not adapted to the special requirements of HPC. On the other hand, using hypervisors on computation machines is more and more being considered as a interesting solution, for various reasons like load balancing or ease of deployment.

In this paper, we have shown that it is possible to take the opportunity of the availability of such hypervisors to run applications more "close to the metal", by embedding them and their execution environment in a lightweight domain, along with a basic kernel with which they can have a very strong costless interaction (since they just run at the same privilege level). We have described some of the optimizations which are made possible in such an environment. All of that actually brings a initial positive answer to one of the research questions raised by Mergen *et al.* (18): 'Does virtualization make it possible to implement a hybrid full-function OS as a combination of two pieces: a small library OS that provides an optimized subset of performance-critical services for a class of HPC applications and runs in the VM with the application, plus a full-function OS that runs in a separate VM and uses introspection of the first VM to provide all other non-performance-critical services?'

By its very nature, this work opens of course optimization perspectives for all kinds of execution environments. For instance, being able to manage memory just like a kernel has long been considered to be reserved to the not-so-widespread microkernel systems. With the advent of hypervisors, the situation changes and this paper has effectively shown that we now have potential for a *viable* way to do it on normal computation systems too, which opens a new area of optimizations.

As machines are becoming increasingly parallel, it would be also interesting to determine the best way to schedule parallel applications in such an environment. A multiprocessor version of Mini-OS exists for the JavaGuest project, so it would be possible to run a parallel application in a single domain. The multiprocessor scheduling could for instance be performed by the dedicated execution environment used by the application. That said, a lot of legacy applications are not parallel, and the level of parallelism that can be used is to run several applications on the same machine. It would then be interesting to consider how to schedule different domains embedding HPC applications. That could even be achieved by having the hypervisor delegate scheduling decisions to a dedicated domain running a complex scheduler.

Software

This lightweight guest domain environment is available in the Xen unstable tree in the `stubdom/` directory.

References

[1] Open source xen hypervisor technology. Xen, Inc., Palo Alto, CA, USA, `http://www.xensource.com/`.

[2] Top 500. `http://www.top500.org/`.

[3] Glenn Ammons, Jonathan Appavoo, Maria Butrico, lma Da Silva, David Grove, Kiyokuni Kawachiya, Orran Krieger, Byran Rosenburg, Eric Van Hensbergen, and Robert W. Wisniewski. Libra: A Library Operating System for a JVM in a Virtualized Execution Environment. In *Virtual Execution Environments (VEE)*, 2007.

[4] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 1992.

[5] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Scheduling in K42. Technical report, IBM Reseach, 2002.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Wareld. Xen and the Art of Virtualization. In *nineteenth ACM symposium on Operating Systems Principles (SOSP'03)*, October 2003.

[7] Bryan Clark, Todd Deshane, Eli Dow, Stephen Evanchik, Matthew Finlayson, Jason Herne, and Jeanna Neefe Matthews. Xen and the Art of Repeated Research. In *FREENIX*, 2004.

[8] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. *Operating Systems Review*, 29(5):251–266, 1995.

[9] Ada Gavrilovska, Sanjay Kumar, Karsten Schwan Himanshu Raj, Vishakha Gupta, Ripal Nathuji, Adit Ranadive Radhika Niranjan, and Purav Saraiya. High-Performance Hypervisor Architectures: Virtualization in HPC Systems. In *1st Workshop on System-level Virtualization for High Performance Computing (HPCVirt 2007)*.

[10] Steven Hand, Andrew Wareld, Keir Fraser, Evangelos Kotsovinos, and Dan Magenheimer. Are Virtual Machine Monitors Microkernels Done Right? In *10th Workshop on Hot Topics in Operating Systems (HOTOS'05)*.

[11] Pascal Hénon, Pierre Ramet, and Jean Roman. PaStiX: A Parallel Sparse Direct Solver Based on a Static Scheduling for Mixed 1D/2D Block Distributions. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 519–527. Springer-Verlag, January 2000. ISBN 3-540-67442-X.

[12] Andreas Jacbosen. Implementing and Testing the APEX I/O Scheduler in Linux . Technical report, University of Oslo, 2007.

[13] Terry Jones, Shawn Dawson, Rob Neely, William Tuel, Larry Brenner, Jeffrey Fier, Robert Blackmore, Patrick Caffrey, Brian Maskell, Paul Tomlinson, and Mark Roberts. Improving the Scalability of Parallel Jobs by adding Parallel Awareness to the Operating System. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 10, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 1-58113-695-1.

[14] Mick Jordan. JavaGuest - A Research Java Virtual Machine on Xen. In *Xen Summit*, November 2007.

[15] Suzanne M. Kelly and Ron Brightwell. Software Architecture of the Light Weight Kernel, Catamount. In *47 th Cray User Group (CUG 2005)*, 2005.

[16] Jochen Liedtke. μ-kernels must and can be small. In *5th International Workshop on Object-Orientation in Operating Systems (IWOOS)*, pages 66–77, October 1996.

[17] Chris I. Dalton Melvin J. Anderson, Micha Moffie. . Technical Report HPL-2007-69, April 2007.

[18] Mark F. Mergen, Volkmar Uhlig, Orran Krieger, and Jimi Xenidis. Virtualization for High-Performance Computing. 40 (2):8–11, 2006.

[19] J. Moreira, M. Brutman, J. Castanos, T. Gooding, T. Inglett, D. Lieber, P. McCarthy, M. Mundy, J. Parker, B. Wallenfelt, M. Giampapa, T. Engelsiepen, and R. Haskin. Designing a highly-scalable operating system: The Blue Gene/L story. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Tampa, FL, USA, November 2006.

[20] Gil Utard Olivier Cozette, Abdou Guermouche. Adaptive paging for a multifrontal solver. In *18th annual international conference on Supercomputing*, pages 267 – 276, Malo, France, 2004.

[21] D. Reed and R. Fairbairns. The Nemesis Kernel – Overview. Technical report, University of Cambridge, 1997. `http://citeseer.ist.psu.edu/reed97nemesis.html`.