2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing

A Performance Study of Virtual Machines on Multicore Architectures

Jie Tao[1], Karl Fürlinger[2], Lizhe Wang[3], and Holger Marten[1]
[1]Steinbuch Center for Computing, Karlsruhe Institute of Technology, Germany
[2]Department of Computer Science, Ludwig-Maximilians-Universität München, Germany
[3]Center for Earth Observation and Digital Earth, Chinese Academy of Sciences, China
jie.tao@kit.edu, fuerling@nm.ifi.lmu.de, Lizhe.Wang@gmail.com, Holger.Marten@kit.edu

Abstract

Cloud computing has promoted the widespread use of virtualized machines. A question arises: How does virtualization influence the performance of running applications? The answer must be a common interest of application developers and users. This paper describes the results of our performance evaluation on a virtualized multicore machine. We tested a set of benchmark applications and detected some general features that should be considered when running applications on a virtualized multicore machine. We also studied the application execution behavior using profiling tools. We found the reason for unexpectedly poor performance of an OpenMP application in a virtualized setting and optimized the program. The optimization resulted in a significant performance gain.

1 Introduction

Cloud computing is currently a hot topic. Increasingly established Cloud platforms are attracting more and more users to develop or port their applications. A specific feature of Clouds is on-demand resource provisioning, which is enabled by the virtual machine technology. As Clouds show the benefit of virtualization, the use of virtual machines is widened to a variety of areas.

Computer architecture, on the other hand, is entering a new era. The design of microprocessors has been moved from single core to multicore (or manycore) and it is clear that emerging computing systems on all scales will be based on multicore nodes. This also means that the virtualized machines will be multicore machines in the future.

Virtualization changes the applications execution behavior because the operating system now runs on top of a virtualization layer rather than directly on the hardware of the host . It is commonly assumed that virtualization introduces a performance loss, especially for parallel applications. A comprehensive view of the performance issue on a virtualized multicore, however, is still a research topic.

This work aims at giving a performance study and analysis to tackle this research problem. We used performance tools and standard benchmark suites to understand the runtime behavior of a virtualized multicore machine, running in both a multi-programming and a shared memory parallel execution mode. During the study we detected several general rules, which can guide the programmers in the task of application development on virtual machines. We also studied the cause of unexpected behavior with an OpenMP application and optimized its code based on the detected problem. The optimization considerably improved the performance of this application.

The remainder of the paper is organized as follows. Section 2 gives a short introduction to the virtualization technology and the related work. Section 3 shows the initial experimental results and gives the details of performance analysis as well as the achieved optimization results. The paper concludes in Section 4 with a short summary and several future directions.

2 The Virtualization Technology

The virtualization technology was proposed in the late 1950s. A wide use of this technology was in the 70s with the purpose of running different application formats on the same hardware to increase the utilization of expensive computing resources. In the 90s, microcomputers were widely adopted to build client-server and peer-to-peer systems. These new computing environments brought with them several problems including security and increased administration complexity. As a solution, virtualization was applied [8] and became thereafter a hot topic.

A traditional computer system runs applications directly on the complete physical machine. Using virtualization, applications are executed on virtual machines (VM), with each VM typically running a single application and a different operating system. Users benefit from this execution model in the following aspects:

- On-demand OS and resource customization: the virtu-

978-0-7695-4633-9/12 $26.00 © 2012 IEEE
DOI 10.1109/PDP.2012.77

alization techniques allow the user to create a VM that provides a customized operating system and resource allocation.

- Performance isolation: the VMs are completely isolated from each other, as if they were separated physical machines.
- Security: the host system monitors the communication to the VMs, restricting the number of successful attacks.
- Availability: VMs can be easily migrated increasing the system's fault tolerance and availability.

These advantages widened the application area of the virtualization technology. Today, virtualization is widely used for server consolidation [21]. In this use case, different servers, like Web, application, and database servers, run on the same physical hardware but with separate VMs. The servers run safely on the shared hardware and can be migrated transparently, increasing server utilization, reliability, and availability while reducing the overall number of physical systems and related recurring costs. Grid computing uses virtualization to achieve interoperability and interoperation between different grid infrastructures [17], to gain administrative flexibility [7], and also to benefit from the traditional advantages of the virtualization technology [17]. Cloud computing adopts virtualization as a key technology to provide on-demand computing resources. Existing cloud infrastructures, including the Amazon EC2 [1], OpenNebula [19], and Eucalyptus [15], all use the virtual machine technology to provide Infrastructure as a Service (IaaS). Resource centers tend to be fully virtualized in order to enhance the resource availability and to reduce the administration cost.

The 1970s simply used binary transformation to run different code on an existing system. A comprehensive machine virtualization started in the 90s with a virtualization layer between the hardware and the guest operating systems. This layer is called Virtual Machine Monitor (VMM) or hypervisor [16]. Xen [2], VMware [22], and KVM [13] are three well known and widely used hypervisors. Xen is an open source development and is widely used for research purposes. KVM is also an open source product. It adds the virtualization capacities directly in the Linux kernel, achieving the thinnest hypervisor of only a few hundred thousand lines of code. VMware is a commercial product and used mainly for server consolidation.

The hypervisor's main task is to virtualize the memory, the devices, and the processor. Memory virtualization aims at mapping the physical memory of a VM to the actual machine memory. Device virtualization makes sure that each VM acquires a virtual device. Processor virtualization takes care of sensitive instructions, such as privileged instructions and exceptions, which cannot be executed directly because with a virtualization layer the operating systems run now at a lower privileged level. One solution is para-virtualization that deploys hypercalls to communicate with the hypervisor. The OS kernel has to be slightly modified to replace the sensitive instructions with hypercalls. The other approach is full virtualization which translates the sensitive instructions to a new sequence of instructions for the virtualized hardware without changing the OS running on a virtual machine.

Clearly, virtualization introduces overhead which results in a performance loss when applications run on a VM rather than directly on the physical machine. This issue was not the focus of interest in the 90s because virtualization was primarily used for the reason of security and system management. Today, however, scientific applications are running on virtualized machines, performance of virtualized machines becomes therefore a research topic.

The Xen developers evaluated their hypervisor, together with VMware and User Mode Linux [4], using several large applications. The results showed that all three hypervisors introduced a significant slowdown with database and web applications [2]. Developers of the ATLAS experiments also measured an up to 14% runtime overhead of VMware ESX with compute-intensive simulation applications in High Energy Physics [10].

For parallel execution Ibrahim, Hofmeyr, and Iancu [11] studied the performance of the NAS parallel benchmarks on VMs runing on a NUMA system. Their experimental results depicted an average performance degradation of 55%, which is caused by poor memory locality management of the underlying hypervisor. Evangelinos and Hill [6] studied the MPI performance on virtual machines. For this study they built a virtual cluster on top of the Amazon EC2 and tested a set of MPI implementations including OpenMPI, GridMPI, LAM and MPICH-2. The results showed quite poor latency and bandwidth performance. In all cases, the message latency is more than double of that measured on a physical, gigabit based cluster, and the asymptotic bandwidth is only a half. Similarly, Ekanayake and Fox measured the MPI runtimes [5] and reported a slowdown of 10% to 40%. Another test [23] showed an even worse performance. Jackson et al. [12] analyzed the HPC applications and found a substantial slowdown compared to dedicated clusters and HPC systems depending on the communication characteristics of the tested application.

In summary, the performance of applications running on VMs has been studied before. However, existing studies targeted on either single processor machines or virtual clusters. The performance of VMs on multicore is not well investigated and a complete performance study with various parallel benchmarks is not available. The goal of this work is to give a comprehensive performance evaluation of virtualized multicore machines. For this purpose we built a testbed with

an 8-core machine virtualized by the open source Xen. We studied the performance of applications in several standard benchmark suites.

3 Performance: Physical vs. Virtual

A virtualized multicore machine can be used to run several applications simultaneously, each on an individual single-core VM. This execution mode maintains the feature of performance isolation of virtual machines. The other execution mode is to run a shared memory parallel application, like a traditional multicore machine usually does. In this case, a VM with several cores is needed. We study both scenarios.

For the experiments, we first created two virtual machines, a fully virtualized VM called VM-full and a paravirtualized VM called VM-para. We then cloned the VM-full and created another seven VMs for running multiple applications. Each VM is equipped with 1-8 processing cores and an up to 4 GB memory. The operating system running on the VMs is Debian 2.6.26. The physical machine has two 2.3 GHz Quad-Core AMD Opteron 2376 ("Shanghai") processors and runs Scientific Linux 5.5.

The applications for the test were chosen from the NAS (NPB2.3-omp-C) and SPEC OpenMP benchmark suites. We also selected two applications from the OpenMP Source Code Repository [18]. One is the program *MolecularDynamics* (MD) which implements a simple molecular dynamics simulation using the velocity Verlet time integration scheme [20] and the other is the *Mandelbrot* program that computes an estimation to the Mandelbrot Set area using MonteCarlo sampling. All applications were compiled with gcc 4.3.2. The *gcc* compiler supports OpenMP since version 4.2. The NAS applications were compiled with a data size of class A, while the SPEC applications were executed using the reference data set. MD was executed with 16384 particles and 20 simulation steps, while the number of points in the *Mandelbrot* set was 1048576. The parameters for these two applications were specifically chosen to balance the execution time.

3.1 Multi-programming Performance

The first experiment was done for the multi-programming execution mode. In this test eight applications were executed once on the physical machine and once on the eight VMs with each application running on a separate VM. We compared the execution time of each application in both cases.

Figure 1 depicts the experimental results. For a better observation the SPEC applications are presented in a single diagram (the lower one) due to their long execution time. The execution time of each application is presented in two

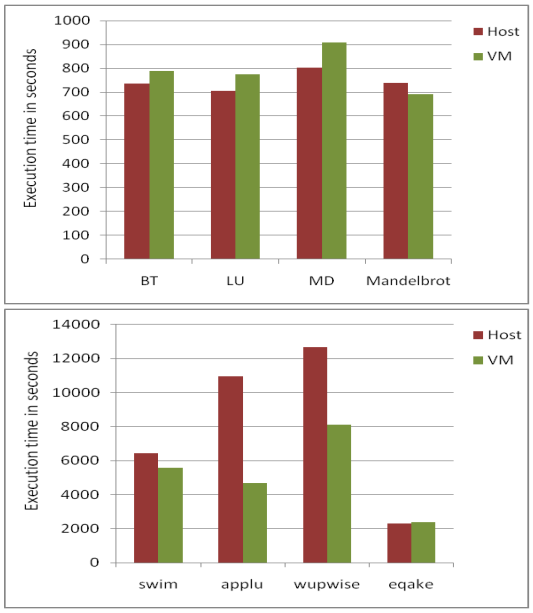

Figure 1. Simultaneous execution of eight applications on the host machine and the virtual machines.

bars with the left for the native execution on the host and the right for VM runs.

As depicted in the upper diagram, BT, LU, and MD show an expected behavior, where the programs run slower on the VM with a slowdown of 7%, 9.6%, and 5.1% individually. *Mandelbrot* presents a different behavior with a speedup of 6.5% on the VM. However, the results with the SPEC applications are more surprising. Observing the lower chart of Figure 1, it can be seen that three of the four applications run faster on the VM, where a speedup of 36% was measured with *wupwise* and the execution time of *applu* is even less than the half of the time measured on the host.

Our first suspicion of the reason for such unexpected behavior was access to memory, because the memory accesses significantly influence performance. We used Xenoprof [14], a profiling tool developed for the Xen virtual machine environment, to examine the memory access behavior of the execution on the host and the VM. Xenoprof relies on performance counters to acquire the runtime performance data. We studied four counter events: data cache misses, data cache line evicted, Data Translation Lookaside Buffer (DTLB) miss, and Instruction TLB (ITLB) miss. However, we only observed a slight difference between the performance data acquired on the VM and with the physical execution. The conclusion is that the better performance on the VM is not related to the memory virtualization.

We then studied the process activity in real time with the

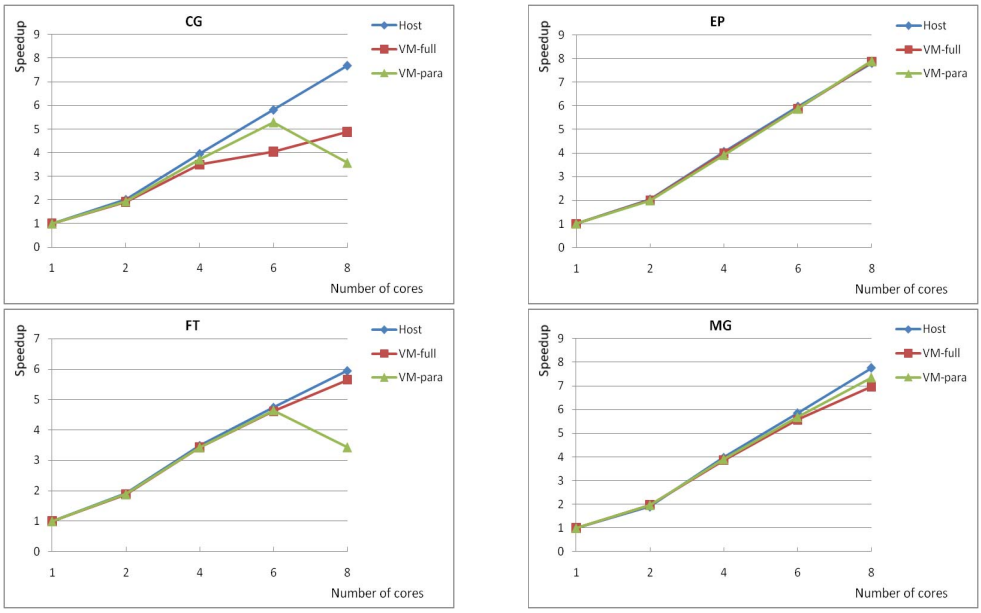

Figure 2. Speedup of NAS applications running on the host machine and virtual machines.

Linux command *top* for the case of physical runs. We observed that *wupwise* and *applu* were marked constantly with a status of D (disk sleep) with a CPU consumption of less than 1%. We found that the reason was related to *swim*. This application had been writing its calculation result to an output file of 85MB and was running the entire time with 100% CPU consumption. Occasionally *wupwise* and *applu* were put into run state but after a few seconds they slept again. Only after *swim* completed its execution both applications were executed with full CPU time.

The execution time of *wupwise* on the physical machine was 211 minutes 21 seconds. However, the actual CPU time was only 129 minutes. This is 6 minutes less than the execution time on the VM. The rest time was spent in waiting for I/O. For *applu* the measured execution time on the host was 182 minutes 43 seconds, among which the CPU time was only 73 minutes 23 seconds. That is 5 minutes less than the execution on the VM. It is clear that the better performance on VMs is contributed by the I/O virtualization. As mentioned in Section 2, each VM has a single virtual I/O device/interface for handling disk activities. Therefore, the I/O conflict, which can occur on the physical machine when running multiple programs simultaneously, is eliminated by virtualization.

3.2 Multi-threading Performance

Multicore is an ideal target platform for running shared memory parallel applications. To evaluate the virtualization impact on parallel execution we studied the performance of OpenMP runs on both the physical system and the virtual environment. We tested several SPEC and NAS OpenMP benchmark applications and measured their execution time on the host and the two VMs: VM-full and VM-para.

We studied the scalability because it is an essential criterion to evaluate the performance of a parallel system. For these experiments we ran only a single virtual machine on the hardware and we ran the applications using different number of threads and calculated the speedup. Figure 2 and 3 demonstrate the experimental results with the NAS applications and the SPEC applications separately.

Within the NAS applications, EP shows the best scalability: the application is scalable with a similar speedup on the VMs as on the host. MG also performs well on the VMs, where a slight difference is visible with the three lines in the diagram. For FT the para-VM scales only to 6 cores and a considerable decrease in speedup can be seen with 8 cores. The same behavior is presented by CG as well, where the para-VM does not run the application faster on 8 cores than on 6 cores. The speedup on the full-VM is also low with CG. The SPEC applications depict a better behavior, with only a bad scalability for *equake*. It would be interesting to see how the VMs behave on larger systems. Unfortunately, we have only machines with eight cores.

In summary, the speedup achieved by the parallel execution on VMs is not bad, except the case with para-VMs of eight cores. However, the overhead caused by virtualization can be clearly seen when examining the individual execution time. The application CG, for example, shows a performance loss of 57% on the 8-core fully virtualized

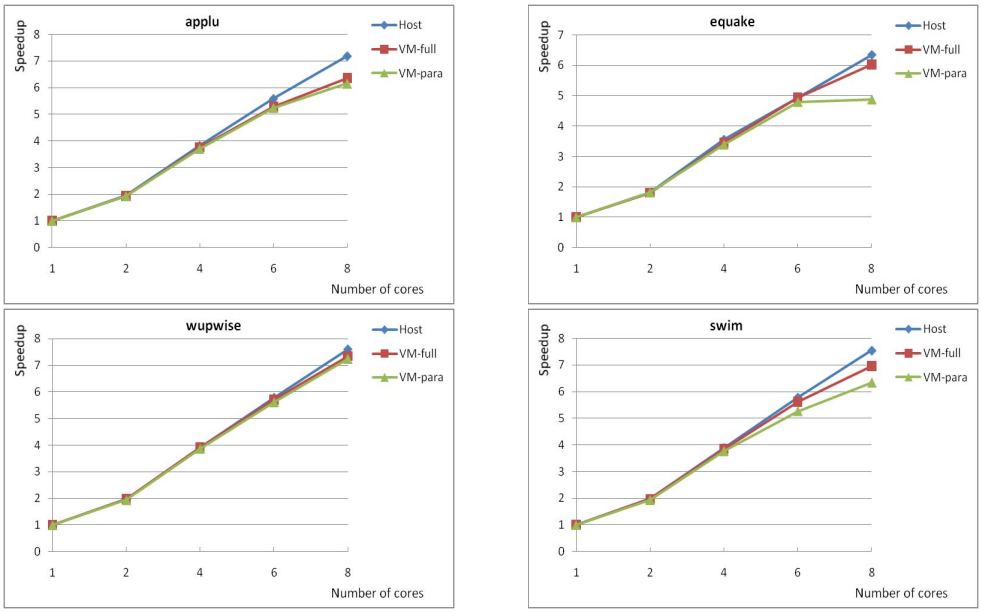

Figure 3. Speedup of SPEC applications running on the host machine and virtual machines.

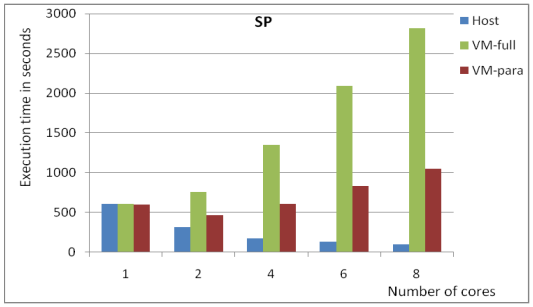

Figure 4. Execution time of the SP application.

VM. For all applications presented in Figure 2 and 3 the best OpenMP run on VMs introduced a slowdown of 0.8% and the worst case showed a slowdown of 75%.

However, the most interesting application is SP from the NAS benchmark suite. The execution time of this application on the host machine and the VMs is depicted in Figure 4. As can be seen the performance on the VMs is very poor. More surprisingly, the execution time even increases with the number of cores. As shown in the figure, it requires 2821 seconds to run SP on the fully virtualized machine using 8 cores. This is 28 times slower than the host run and a 368% slowdown to the case of sequential runs on the same VM. This means that the parallelization of SP causes a significant overhead.

In order to gain a deeper understanding of the problem, we systematically studied the overheads of OpenMP constructs on the virtualized machines with the EPCC OpenMP micro-benchmark suite [3].

Table 1 shows the results of all OpenMP constructs measured by the micro-benchmarks. The data were collected on both the host machine and VM-full with different number of cores.

The first OpenMP construct is PARALLEL which defines a parallel region. It can be clearly seen that the overhead with this construct on the VM is considerably larger than on the host. Additionally, the overhead rises with the number of cores in both cases, but with a smaller increase on the host. On the VM the overhead increases significantly.

PARALLEL FOR also shows a significant overhead on the VM. Similar to the construct PARALLEL, the overhead goes up with the system scale. However, unlike PARALLEL a slowdown with the overhead increase cannot be seen with PARALLEL FOR (the case with seven cores is an exception).

The situation with BARRIER, a construct for thread synchronization, is more critical. While the host machine presents a constant overhead with this construct, the overhead on the VM arises drastically with the number of cores at a rate of 49% in average.

SINGLE is a construct used to define code regions that are only executed by a single thread. This construct shows the worst behavior, where the overhead on the host is less than one second in all cases but the construct introduces an overhead as high as 293 seconds on the VM when executing the test code using eight cores.

Table 1. Overheads of the OpenMP constructs (in seconds).

Constructs		2	3	4	5	6	7	8
PARALLEL	Host	3.17	3.19	3.23	3.24	3.46	3.85	3.87
	VM	50.67	99.73	123	150.67	168.4	170.36	173.18
PARALLEL FOR	Host	3.24	3.25	3.28	3.34	3.67	3.96	4.04
	VM	51.39	100	121	151	174.43	175.31	220.23
BARRIER	Host	1.43	1.43	1.43	1.43	1.43	1.43	1.43
	VM	34.97	84.66	127	166.65	210.81	237.64	292.47
SINGLE	Host	0.56	0.63	0.7	0.76	0.57	0.91	0.63
	VM	40.59	81.7	126	168.56	212.13	236.61	293.1
CRITICAL	Host	0.25	0.3	0.3	0.36	0.57	0.7	0.83
	VM	0.56	1.66	1.86	2.54	3.13	3.15	3.49
LOCK/ UNLOCK	Host	0.28	0.44	0.44	0.44	0.62	0.77	0.89
	VM	0.53	1.68	1.8	2.51	3.04	3.05	3.49
ORDERED	Host	1.2	0.94	0.91	0.88	0.88	0.88	0.88
	VM	31.46	30.11	30.87	31.2	31.4	31.4	31.4
ATOMIC	Host	0.1	0.1	0.1	0.18	0.21	0.23	0.34
	VM	0.07	0.11	0.2	0.2	0.24	0.26	0.27
REDUCTION	Host	3.18	3.25	3.51	3.56	3.9	3.92	3.92
	VM	54.63	100	120	151	169.7	201.77	223.67

The next four constructs are used for mutual exclusion. The first two constructs, CRITICAL and LOCK/UNLOCK, show a larger overhead increase on the VM than on the physical machine. However, the overhead is not high in contrast to the aforementioned constructs. The overhead with ATOMIC is visible but maintains constant as the system scale changes. ATOMIC shows similar overhead on the VM as on the host machine.

The behavior with REDUCTION, a construct for calculating a sum of the partial results, is similar to BARRIER. The overhead caused by this construct goes up linearly with the number of cores on the VM, while the physical machine shows only a slight increase. In addition, the overheads on the VMs are much larger than on the host machine.

Overall, all OpenMP constructs introduce more overhead on VMs than on the host and the overhead on VMs increases fast with the number of cores for most of the constructs. Therefore, we assumed that this overhead is probably the reason for the unexpected performance of SP. However, why does only SP show the strange behavior while other tested programs not?

In order to find the answer, we applied another profiling tool, ompP [9], to analyze the OpenMP constructs in detail. ompP is an OpenMP profiler that collects performance data at the runtime by instrumenting the source code. It delivers per-region and per-thread timing statistics of the OpenMP constructs at the end of the program run. As an advanced feature, ompP produces an overhead analysis report which quantifies the overhead into four categories: load imbalance, synchronization, limited parallelism, and thread management. The first one defines the overhead caused by imbalanced work sharing across the collaborating threads and the subsequent idle waiting time. Synchronization overhead is the overhead that arises because threads need to coordinate their activity. An example is the waiting time to enter a critical section or to acquire a lock. The overhead of limited parallelism is resulted from unparallelized or only partly parallelized regions of code. Thread management overhead defines the time spent by the runtime system for managing the application's threads. That is, time for creation and destruction of threads in parallel regions and overhead incurred in critical sections and locks for signaling the lock or critical section as available.

The ompP report depicts that only 13% of the time needed for executing SP is used for useful computing and the rest is caused by overheads. ompP also shows that the overhead is almost fully caused by load imbalance in work-sharing regions. Examining the region overview delivered by ompP we found that SP contains 71 work-sharing regions, in which 69 regions are LOOPs parallelized with the construct PARALLEL FOR. This led us to further study the runtime behavior of these LOOPs.

Table 2 shows the time spent by all eight threads, T1-T8, in a sample LOOP of SP. The data were collected on both the host machine and VM-full. ompP reported the time needed for the LOOP body (bodyT) and the time for exiting the implicit BARRIER at the end of the LOOP (exitBarT).

It can be seen that the time used by each thread for the LOOP body varies only slightly between the VM and the host. However, each thread needs more than 310 seconds

Table 2. Execution time of a LOOP in SP (all threads, time in seconds).

		T1	T2	T3	T4	T5	T6	T7	T8
VM	BodyT	11.24	11.22	11.33	11.22	11.26	11.24	11.17	10.92
	ExitBarT	289.41	289.35	289.12	289.14	289.68	289.62	289.99	290.48
Host	BodyT	14.55	14.84	10.81	14.39	14.45	11.65	10.23	9.79
	ExitBarT	38.92	38.91	37.22	38.03	38.77	35.47	38.85	39.35

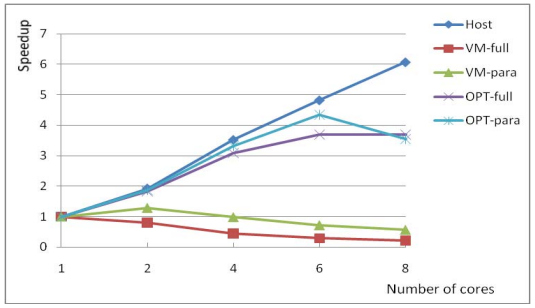

Figure 5. Speedup of the SP application: optimized vs. original.

for exiting the LOOP BARRIER on the VM while less than 40 seconds on the host.

Normally the overhead in a work-sharing region, like a LOOP, is caused by the inconsistent execution time of each thread, where some threads complete their work earlier and have to wait for other threads. For SP, however, Table 2 shows that the work is better distributed across the threads on the VM than on the physical machine. Therefore, we concluded that the high overhead on the VM is not really caused by load imbalance.

The conclusion led us to study the implementation of BARRIER in *gcc*. GNU uses a common approach to achieve the thread synchronization, with a counter combined with a BARRIER. The counter is initialized with the number of the parallel threads that work together for a sharing region. Each thread decreases the counter by one when arriving the BARRIER and then is blocked till the counter value is zero. The problem lies in that GNU uses the function *omp_get_num_threads* to acquire the number of total threads. The latter then uses a system call to enter the kernel space, which involves the hypervisor on a VM. Hence, the BARRIER operation is more expensive on the VM than on the host. For the studied LOOP ompP shows that each thread enters the LOOP more than 1.5 million times; therefore the overhead is considerable because each entering is combined with an implicit BARRIER. Other tested applications also contain a number of parallel LOOPs, but they are executed only a few hundred times and the overhand is not so large to limit scaling.

We went through the source code of SP. For the LOOPs with similar behavior as the example we moved the parallelization from the inner loop to the outer one to reduce the number of BARRIERs. Surprisingly, this simple optimization achieved a significant performance gain.

Figure 5 shows the speedup comparison of the optimized version vs. the original code. It can be seen that the lowest two lines, which present the original version on VMs, go down with the number of cores, while the two lines for the optimized version go up. The speedup of the optimized version on the VMs is still not close to that achieved on the host but it can be clearly seen that the application runs faster with more cores except the case of using eight cores on the para-virtualized machine. However, even with this case the execution time on the VM is reduced from 1051 seconds to 168 seconds, meaning that the optimized version runs 6.2 times faster than the original implementation.

4 Conclusions

Increasingly virtualized machines are adopted for scientific computing. An often asked question is: How does the virtualization change the applications execution behavior? The paper studied this issue with a multicore machine. We tested a set of standard benchmark applications and analyzed the runtime behavior with performance tools. We found that running applications separately on single VMs may be better than running them all on the same physical machine due to a better I/O performance contributed by virtualization. When running parallel applications, however, the performance is worse on the VM. A general guide for shared memory programmers is to use synchronization directives as few as possible because they could damage the performance. Hence, the traditionally applied fine-grained parallelization shall not be applied when programming virtualized multicore machines.

Currently, we are planning experiments with real applications to find more generic problems related to parallelization on virtual machines. We will also study novel virtualization techniques that take the specific feature of multicore machines into account with the goal of reducing the virtualization overhead.

Acknowledgements

This work was partially supported by the EU project MADAME: Multicore Application Development and Modeling Envi ronment.

References

[1] Amazon Web Services. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[2] P. Barham, B. Dragovic, and K. Fraser. Xen and the Art of Virtualization. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 164–144, 2003.

[3] J. M. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *The First European Workshop on OpenMP*, pages 99–105, 1999.

[4] J. Dike. *User Mode Linux*. Prentice Hall, April 2006.

[5] J. Ekanayake and G. Fox. High Performance Parallel Computing with Clouds and Cloud Technologies. In *Proceedings of the first International Conference on Cloud Computing*, October 2009.

[6] C. Evangelinos and C. N. Hill. Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. In *Proceedings of CCA-08*, 2008.

[7] R. Figueiredo, P. Dinda, and J. Fortes. Case for Grid Computing on Virtual Machines. In *Proceedings of the 23rd International Conference on Distributed Computin*, pages 550–559, May 2003.

[8] R. Figueiredo, P. A. Dinda, and J. Fortes. Resource Virtualization Renaissance. *Computer*, 38(5):28–31, 2005.

[9] K. Fürlinger and M. Gerndt. ompP: A profiling tool for openmp. In *Proceedings of the First and Second International Workshops on OpenMP (IWOMP 2005, IWOMP 2006)*, pages 15–23, Eugene, Oregon, USA, May 2005. LNCS 4315.

[10] L. Gilbert, J. Tseng, and R. Newman. Performance Implications of Virtualization and Hyper-Threading on High Energy Physics Applications in a Grid Environment. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, April 2005.

[11] K. Z. Ibrahim, S. Hofmeyr, and C. Iancu. Characterizing the Performance of Parallel Applications on Multi-socket Virtual Machines. In *IEEE International Symposium on Cluster Computing and the Grid*, May 2011.

[12] K. R. Jackson et al. Performance analysis of high performance computing applications on the amazon web services cloud. In *Proceedings of the 2nd International Conference on Cloud Computing Technology and Science (CloudCom 2010)*, 2010.

[13] KVM. Kernel Based Virtual Machine. http://www.linux-kvm.org/.

[14] A. Menon, J. R. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *The 1st ACM/USENIX international conference on Virtual execution environments*, pages 13–23, 2005.

[15] D. Nurmi, R. Wolski, and C. Grzegorczyk. The Eucalyptus Open-source Cloud Computing System. In *Proceedings of CCA-08*, 2008.

[16] M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *Computer*, 38(5):39–47, 2005.

[17] M. Ruda, J. Denemark, and L. Matyska. Scheduling Virtual Grids: The Magrathea System. In *The 3rd International Workshop on Virtualization Technology in Distributed computing*, November 2007.

[18] F. Sande. OmpSCR: OpenMP Source Code Repository. http://sourceforge.net/projects/ompscr/.

[19] B. Sotomayor, R. Montero, I. Llorente, and I. Foster. Capacity Leasing in Cloud Systems using the OpenNebula Engine. In *The First Workshop on Cloud Computing and its Applications*, October 2008.

[20] W. C. Swope, H. C. Andersen, P. H. Berens, and K. R. Wilson. A Computer Simulation Method for the Calculation of Equilibrium Constants for the Formation of Physical Clusters of Molecules: Application to Small Water Clusters. *Journal of Chemical Physic*, 76, 1982.

[21] VMware. Server Consolidation. http://www.vmware.com/solutions/consolidation/.

[22] VMware Inc. VMware. http://www.vmware.com.

[23] E. Walker. Benchmarking Amazon EC2 for High-Performance Scientific Computing. *The USENIX Magazine*, 33(5), October 2008.