LETTER Special Section on Parallel and Distributed Computing and Networking

Synchronization-Aware Virtual Machine Scheduling for Parallel Applications in Xen

Cheol-Ho HONG†a), *Student Member and* Chuck YOO†b), *Member*

SUMMARY In this paper, we propose a synchronization-aware VM scheduler for parallel applications in Xen. The proposed scheduler prevents threads from waiting for a significant amount of time during synchronization. For this purpose, we propose an identification scheme that can identify the threads that have awaited other threads for a long time. In this scheme, a detection module that can infer the internal status of guest OSs was developed. We also present a scheduling policy that can accelerate bottlenecks of concurrent VMs. We implemented our VM scheduler in the recent Xen hypervisor with para-virtualized Linux-based operating systems. We show that our approach can improve the performance of concurrent VMs by up to 43% as compared to the credit scheduler.

key words: *virtual machine scheduling, parallel application*

1. Introduction

Recently, system virtualization has attracted noteworthy attention as it has become the key enabling technology of cloud computing, which is one of the new paradigms related to distributed computing systems. Virtualization enables multiple operating systems (OSs) to run simultaneously on a single physical machine, thus achieving the effective utilization of system resources in a cloud environment.

However, parallel applications, which perform large-scale computationally intensive tasks, such as computational fluid dynamics (CFD) applications, can suffer from performance degradation when they are executed on the Xen hypervisor. This is because Xen asynchronously schedules the concurrent VMs that deploy parallel applications while the synchronization primitives in parallel applications assume the synchronous progress of the virtual machine (VM) [1]. Prior research [2], [3] has addressed this problem by adaptively co-scheduling the concurrent VMs. However, the co-scheduling policy has certain drawbacks such as CPU fragmentation, priority inversion, and a lack of scalability [4]. Therefore, a new mechanism that can overcome the above-mentioned disadvantages is required.

In this paper, we propose a synchronization-aware VM scheduler for parallel applications hosted by Xen. For this purpose, we first analyze the cause of the performance deterioration of the concurrent VMs in Sect. 2. Second, we develop a new scheduler that can improve the performance of parallel applications in Sect. 3. The proposed scheduler is based on the original credit scheduler of Xen. Finally, we evaluate the performance of the proposed scheduler in Sect. 4.

Manuscript received December 27, 2012.
Manuscript revised April 19, 2013.

†The authors are with the Department of Computer Science and Engineering, Korea University, Seoul, Korea.
a) E-mail: chhong@os.korea.ac.kr
b) E-mail: chuckyoo@os.korea.ac.kr

DOI: 10.1587/transinf.E96.D.2720

2. Performance of Concurrent VMs

A parallel application divides the total work into several concurrent threads that can be executed in parallel. Each thread in the parallel application consists of multiple phases, each of which has one computation part and one subsequent synchronization part. Through the synchronization part, all operations in the same phase across all threads must be completely finished before the next step begins. This synchronization procedure is required because the next step relies on the data written during the previous phase. As the synchronization method, barriers are commonly used. A barrier denotes a point where all threads must wait until all other threads arrive. The waiting threads then spin or are blocked according to the wait policy configuration. In a non-virtual environment, the waiting time is brief and acceptable.

For a virtual environment, however, the waiting time can be comparatively long. Figure 1 illustrates an undesired case where a concurrent VM has four virtual CPUs (VCPUs), each of which runs a thread of a parallel application. In this figure, the second phase of the thread running on CPU2 cannot proceed until the operations of the first phase on CPU3 are completed. Furthermore, the execution of the second phase on CPU2 may be delayed because the VCPU on CPU2 was inserted at the tail of the run queue when it was unable to proceed. Thus, the thread on CPU2 may have to wait for a significant amount of time before proceeding.

This long waiting time worsens the performance of the parallel application. According to the proportional-share scheduling model developed in our previous research [5], the completion time of the parallel application is as follows:

$$T_{Parallel} = \left\lfloor \frac{Lag + \lceil E_T \rceil - 1}{R_I} \right\rfloor + E_T - (\lceil E_T \rceil - 1) \qquad (1)$$

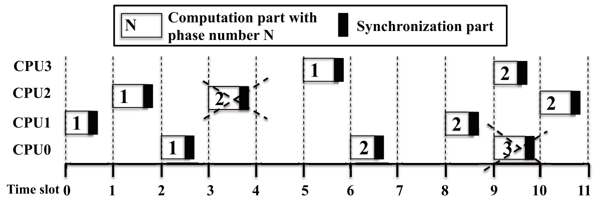

Fig. 1 Example of VM scheduling where a concurrent VM has four virtual CPUs, each of which runs a thread of a parallel application.

In Eq. (1), *Lag* is the upper bound of the difference between the ideally and the actually obtained CPU time of VM V_i during execution. $R_I = \frac{|P| \times \omega(V_i)}{|C(V_i)|}$, where $|P|$ is the number of physical CPUs in the system, and $\omega(V_i)$ is the weight of VM V_i. $|C(V_i)|$ indicates the number of VCPUs in VM V_i. E_T is the maximum of the sum of the execution times of all phases in any thread in the parallel application. According to Eq. (1), only the *Lag* value affects the completion time of the parallel application provided that other elements such as $|P|$, $\omega(V_i)$, $|C(V_i)|$, and E_T are fixed during the execution.

The completion time then increases with *Lag*. This means that as the VM receives less time from the CPU at any moment than what it ideally should, the performance degrades further. When we return to the previous example, as the waiting time of the threads increases above a certain threshold, the allocated CPU time of these threads is yielded to other non-concurrent VMs or is consumed by the idle domain; this situation is similar to the typical execution pattern of an I/O bound program in a VM. When this occurs, the *Lag* value increases because until this point, the CPU time actually obtained by the VM is less than the ideally allocated time. As *Lag* increases, the computation parts of the application are then delayed and become long overdue because of the lack of CPU time obtained. Consequently, the long waiting time increases *Lag* and results in the degradation of the performance of the parallel application.

3. Synchronization-Aware VM Scheduling

The proposed scheduler is based on the original credit scheduler of Xen. Our scheduling algorithm consists of an identification scheme and a scheduling policy. Before describing the scheduling algorithm, we make several simplifying assumptions:

1. As parallel applications commonly run using a number of threads equal to or less than the number of cores [6], we assume that the number of threads does not exceed the number of VCPUs in the VM. Under this assumption, a thread and a VCPU can be mapped with a one-to-one relationship.
2. Each parallel thread is assumed to be blocked rather than spin when the thread has to await other threads at a barrier. In virtualization, the wait policy is generally configured to such a condition in order to prevent superfluous CPU spinning.
3. To avoid the cost of thread migration in the guest OS and make our inference technique accurate, we assume that each thread is fixed to a VCPU.

In the identification scheme, the hypervisor identifies the threads that have awaited other threads for a significant amount of time. For this purpose, we provide the detection module for parallel applications by which the hypervisor can infer the internal status of guest OSs. In the guest OS, when a thread reaches a barrier, and other threads have not, the thread makes a system call that puts itself to sleep in the kernel according to the second assumption. When the last thread reaches the barrier, the thread makes a system call that awakens all the blocked threads. Examples of such system calls are *futex_wait()* and *futex_wake()* in Linux. The detection module in the hypervisor then snatches the system call by using the system call interception mechanism introduced in [7].

Table 1 Synchronization information.

Element	Description
dom_id	ID of a VM that executes the parallel application.
vcpu_id	ID of a VCPU on which the parallel thread executes the sleep call.
call_time	Time when the sleep call is issued.

Using the intercepted information, the module generates the synchronization information for the VM scheduler, as shown in Table 1. In this table, the detection module can directly utilize the ID of the VCPU to indicate the thread that executes the sleep call because, according to the third assumption, the guest OS scheduler cannot change the CPU assignment of the thread after it has been blocked. An element in the table is inserted when *futex_wait()* is executed; the elements that belong to the same VM are deleted when *futex_wake()* is executed because the occurrence of this call denotes that all the threads have reached the barrier. The detection module then periodically identifies the threads that have been awaiting other threads for a long time by subtracting *call_time* in the table from the current time. The advantage of utilizing the detection module is that we do not have to install any instrumentation on the guest OSs to know the synchronization phases in parallel applications.

In the scheduling policy, the hypervisor accelerates bottlenecks of concurrent VMs when the parallel threads have waited for a long time at the barrier. A bottleneck in the VM consists of multiple threads that need to reach the barrier before the waiting threads can proceed. Therefore, accelerating bottlenecks allows the waiting threads to resume execution in a short period of time and to obtain the CPU time allocated to them. In this scheduling policy, the *Lag* value illustrated in Sect. 2 decreases as compared to non-synchronization-aware scheduling, and parallel applications thus achieve improved performance.

When the hypervisor decides to accelerate a bottlenecks in a VM, *runnable* VCPUs in the VM become the bottleneck. In scheduling VMs, there are three states that a VCPU can have: *runnable*, *running*, and *blocked*. When the last thread reaches a barrier, it awakens all the blocked threads. Then, the state of each VCPU is changed to *runnable* from *blocked*, and the VCPUs are inserted into the run queue. After that, when a certain VCPU is selected to run, its state is changed to *running*. If this thread reaches the barrier, it sleeps in the kernel. Then, the VCPU state is changed to *blocked*. Because the bottleneck consists of the threads that have not reached the barrier, the states of VCPUs to which the threads belong then have to be *runnable*.

The hypervisor accelerates the bottleneck when the parallel threads have waited for a long time. The threshold of the waiting time is determined by the ideal VM schedul-

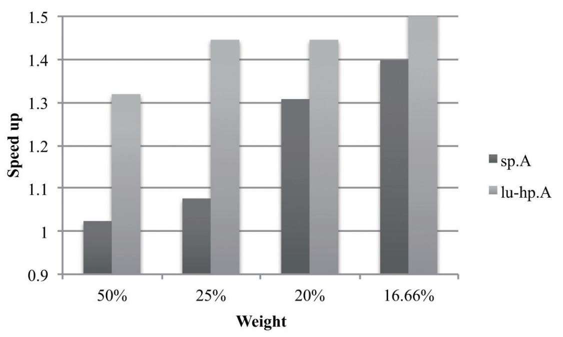

Fig. 2 Increases in performance for the synchronization-aware scheduler with weights of 50%, 25%, 20%, and 16.66%.

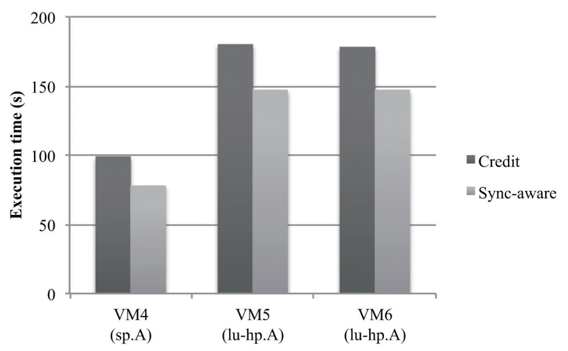

Fig. 3 Completion time for *sp.A*, *lu-hp.A*, and *lu-hp.A*.

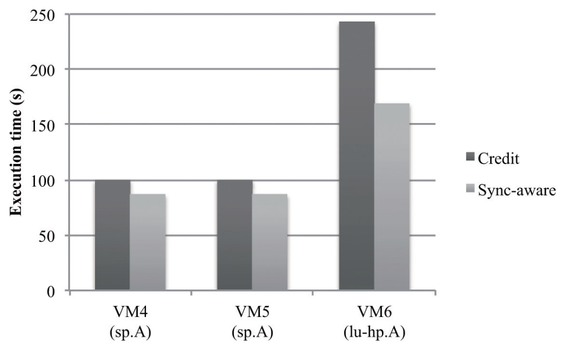

Fig. 4 Completion time for *sp.A*, *sp.A*, and *lu-hp.A*.

ing interval. The ideal interval is obtained under the assumption that each VCPU consumes exactly one time quantum in the interval and that value is $\frac{|C(V)|}{|P|} \times T$. $|C(V)|$ represents the total number of VCPUs in the system. $|P|$ denotes the number of physical cores in the system, and constant T indicates the size of the time quantum. Therefore, when any thread has waited for a time equal to the ideal VM scheduling interval, the bottleneck is accelerated. This acceleration is conducted by the boost mechanism in Xen [8]. To maintain system-wide fairness, however, we conditionally accelerate the bottleneck only when all VCPUs in the concurrent VM have still not consumed the allocated amount of CPU time.

4. Evaluation

We have implemented a synchronization-aware VM scheduler on an Intel Xeon E5-2620 hexa-core platform that has six 2.0-GHz cores with a disabled hyper-threading mechanism. The system is hosted by the recent Xen 4.1.3 hypervisor and para-virtualized Linux-2.6.32. We compile the Linux kernel with an enabled *paravirt_spinlocks* option in order to tolerate the lock holder preemption problem [9]. For the benchmark programs, we select the OpenMP version of *sp.A* and *lu-hp.A* from *NAS Parallel Benchmarks (NPB)* that are utilized to help evaluate the performance of parallel supercomputers [10].

First, we observe the performance impact of the synchronization-aware scheduler when the weight of the concurrent VM decreases. For this purpose, a target VM executing either *sp.A* or *lu-hp.A* with six VCPUs is run individually. To decrease the weight of the target VM, we deploy other multiple-CPU-bound VMs that simulate background workloads. The performance improvements compared to the original credit scheduler, where the weight is varied from 50% to 16.66%, are shown in Fig. 2. The results indicate that both programs can benefit from synchronization-aware scheduling, improving their speed by up to 49%.

Second, we evaluate the performance of the mixed workloads that consist of multiple single and parallel tasks. This experiment utilizes six VMs, each of which has six VCPUs. For the first three VMs, we run single-threaded CPU-bound programs to stress the CPU. In the other VMs, we deploy *sp.A* and *lu-hp.A* to measure the performance of the parallel applications. The completion times of the parallel applications are shown in Figs. 3 and 4. Figure 3 shows that our scheduler improves the performance of each *sp.A* by up to 26% and *lu-hp.A* by up to 22%. Similarly, in Fig. 4, our scheduler improves the performance of *sp.A* for each VM by up to 14% and *lu-hp.A* by up to 43% as compared to the credit scheduler.

5. Related Work

Uhlig et al. [11] proposed two techniques, intrusive and non-intrusive, to avoid CPU spinning incurred by the preemption of VCPUs that hold the kernel locks. The intrusive technique modifies the locking primitive in the guest OS and informs the hypervisor how long a lock should be held by a VCPU, thus avoiding preemption of the VCPU that holds the lock. The non-intrusive technique is for operating systems distributed in binary form. This technique avoids preemption of the VCPU that is in the unsafe state. In this state, kernel locks may be held by the VCPU currently executing at the kernel level. Friebel et al. [9] later investigated the influence of lock holder preemption in the Xen hypervisor. They proposed a technique to tolerate lock holder preemption by preventing unnecessary active waiting. They modified the spin lock code in the guest OS to issue a sleep hyper call when the wait period of any VCPU grows longer than a certain threshold.

These methods can successfully prevent CPU spinning at the kernel level. However, as pointed out in [11] and [9], they cannot handle synchronization methods utilized at the user level. In practice, the parallel application domain in virtualization environments relies on block-based synchronization barriers at the application level. Although we adopt the technique suggested by Friebel et al. [9] at the kernel level, we find that the long waiting time of parallel threads during the barrier synchronization worsens the performance of parallel applications. To alleviate this situation, our scheduler prevents threads from waiting for a significant amount of time throughout synchronization. Therefore, we further improve the performance of parallel applications compared to the methods of previous research.

6. Conclusion

In our synchronization-aware VM scheduler, we presented an identification scheme and a scheduling policy to achieve acceptable performance for parallel applications. We also presented a performance evaluation that shows that our scheduling algorithm is more efficient than the original credit scheduler in Xen.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No.2010-0029180).

References

[1] C. Xu, Y. Bai, and C. Luo, “Performance evaluation of parallel programming in virtual machine environment,” Sixth IFIP International Conference on Network and Parallel Computing, 2009, NPC ’09, pp.140–147, 2009.
[2] C. Weng, Z. Wang, M. Li, and X. Lu, “The hybrid scheduling framework for virtual machine systems,” Proc. 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp.111–120, 2009.
[3] C. Weng, Q. Liu, L. Yu, and M. Li, “Dynamic adaptive scheduling for virtual machines,” Proc. 20th International Symposium on High Performance Distributed Computing, pp.239–250, San Jose, 2011.
[4] O. Sukwong and H. Kim, “Is co-scheduling too expensive for SMP VMS?,” Proc. Sixth Conference on Computer Systems, pp.257–272, 2011.
[5] C.H. Hong, Y.P. Kim, S. Yoo, C.Y. Lee, and C. Yoo, “Cache-aware virtual machine scheduling on multi-core architecture,” IEICE Trans. Inf. & Syst., vol.E95-D, no.10, pp.2377–2392, Oct. 2012.
[6] Y. Nishitani, K. Negishi, H. Ohta, and E. Nunohiro, “Implementation and evaluation of OpenMP for hitachi sr8000,” High Performance Computing, pp.391–402, 2000.
[7] F. Beck and O. Festor, “Syscall interception in Xen hypervisor,” Technical report, Institut National Polytechnique de Lorraine (INPL), 2009.
[8] D. Ongaro, A. Cox, and S. Rixner, “Scheduling I/O in virtual machine monitors,” Proc. Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp.1–10, 2008.
[9] T. Friebel and S. Biemueller, “How to deal with lock holder preemption,” Presentation at Xen Summit North America, 2008.
[10] H. Jin, M. Frumkin, and J. Yan, “The OpenMP implementation of NAS parallel benchmarks and its performance,” Technical Report, NAS-99-011, NASA Ames Research Center, 1999.
[11] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski, “Towards scalable multiprocessor virtual machines,” Proc. 3rd Virtual Machine Research and Technology Symposium, pp.43–56, 2004.