Analysis of the Impact of CPU Virtualization on Parallel Applications in Xen

Kenichi Kourai
Department of Creative Informatics
Kyushu Institute of Technology
Fukuoka, Japan
kourai@ci.kyutech.ac.jp

Riku Nakata
Department of Mechanical Information Science
Kyushu Institute of Technology
Fukuoka, Japan
rikuhana@ksl.ci.kyutech.ac.jp

***Abstract*—Recently, Infrastructure as a Service (IaaS) is being used for parallel computing. In IaaS clouds, parallel applications are run in virtual machines (VMs), which own virtual CPUs (vCPUs). Application threads are scheduled to vCPUs and then vCPUs are scheduled to physical CPUs (pCPUs). This CPU virtualization can affect the performance of parallel applications. According to our experiments in Xen, the scalability of parallel applications in a VM was largely different from that in a physical machine (PM). In this paper, we analyze the root cause of such a difference in scalability. As a result of our investigation, we found that the root cause was both resource conflicts between pCPUs and conflict in vCPU scheduling. In addition, we provide three methods for avoiding these conflicts and improving scalability. Furthermore, we confirmed that the optimal vCPU scheduling could be effective for not only applications parallelized by Tascell but also most of NAS Parallel Benchmarks.**

***Keywords*-vCPU scheduling, virtual machines, scalability, chip multi-threading, parallel computing**

I. Introduction

Traditional parallel computing is usually performed using dedicated computer clusters. Recently, cloud computing is being used for running parallel applications. Since cloud computing provides elasticity, users can use necessary resources when they need. For example, they can use clouds only when running parallel applications. According to running applications, they can use the necessary number of CPUs and the necessary amount of memory. In addition, cloud computing can reduce the cost for parallel computing because users can pay only for used resources.

In particular, Infrastructure as a Service (IaaS) is used for maximum flexibility. Users can run any parallel applications with the operating system and libraries required by the applications. IaaS clouds provide virtual machines (VMs) to users and users can set up the entire system. A VM often owns more than one virtual CPU (vCPU). Application threads are scheduled to vCPUs by the operating system in a VM. Then vCPUs are scheduled to physical CPUs (pCPUs) by the hypervisor. This CPU virtualization can affect the performance of parallel applications. Extra vCPU scheduling is performed and the thread scheduler in a VM cannot obtain physical information precisely.

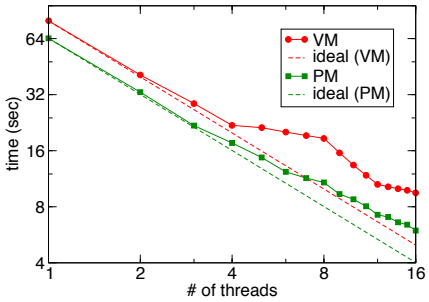


Figure 1. The performance in a PM and a VM.

To compare the performance between a VM and a physical machine (PM), we executed an application parallelized by Tascell [1], which is a framework for dynamic load balancing. Fig. 1 shows the execution time of this application for various numbers of threads in Xen 4.4 and Linux 3.13. We explain the details of this experiment in Section III-B. From this result, we found that scalability was largely different even if we exclude the fixed virtualization overhead. The performance improvement gradually slowed down in a PM, while it steeply slowed down in 4 threads in a VM. Strangely, the performance in a VM improved well again in more than 8 threads.

In this paper, we analyze the root cause of such a difference in scalability between a VM and a PM. Consequently, we found that the reason why performance improvement steeply slowed down was resource conflicts between pCPUs. AMD Opteron 6376 shares several resources between two CPU cores. In addition, the root cause of the strange scalability was conflict between the vCPU and NUMA-node affinities in vCPU scheduling. On the basis of our observation, we provide three methods for avoiding these conflicts. Disabling the node affinity and/or the vCPU affinity could improve scalability. The optimal vCPU affinity could achieve the highest scalability and the performance became comparable to that even in a PM in 16 threads.

Furthermore, we conducted the same investigation for other parallel applications such as another application parallelized by Tascell and the NAS Parallel Benchmarks [2]. Most of the benchmarks showed the similar difference in

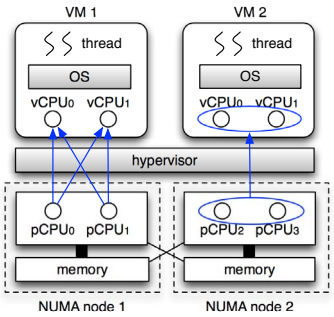


Figure 2. vCPU scheduling.

scalability between a VM and a PM and the optimal vCPU affinity largely improved the performance in a VM.

The organization of this paper is as follows. Section II explains the basics of CPU virtualization. Section III shows the performance comparison between a VM and a PM and analyzes the root cause of the difference in scalability. Section IV provides three methods for avoiding two kinds of conflicts. Section V shows the performance in other parallel applications. Section VI discusses related work and Section VII concludes this paper.

II. CPU Virtualization

A VM often owns more than one vCPU, whose number is usually less than that of pCPUs. Application threads are scheduled to vCPUs by the operating system in a VM and then vCPUs are scheduled to pCPUs by the hypervisor, as illustrated in Fig. 2. vCPU scheduling is indispensable when the total number of vCPUs owned by VMs is more than the number of pCPUs. Also, it is necessary when pCPUs are temporarily used for other purposes, e.g., I/O and system management. If there are no constraints for vCPU scheduling, pCPUs are assigned to arbitrary vCPUs.

As one of the constraints, users can set the vCPU affinity to a VM. The vCPU affinity enables the static binding of vCPUs to specific pCPUs. A vCPU can be assigned to a pCPU, or a group of vCPUs can be assigned to a group of pCPUs. In the latter case, the vCPUs in a group are dynamically scheduled to only an assigned group of pCPUs. One of the advantages of the vCPU affinity is to reduce communication between processors by scheduling vCPUs to only one processor. Another advantage is to use CPU cache effectively by one-to-one assignment.

The other constraint on vCPU scheduling is the awareness of non-uniform memory access (NUMA). In the NUMA architecture, a processor and its local memory consist of a NUMA node, as shown in Fig. 2. A processor can access local memory faster than remote memory of the other NUMA nodes. Therefore, the vCPU scheduler attempts to assign one or small number of nodes to a VM if possible. This is called the node affinity. Only pCPUs included in assigned nodes are scheduled to vCPUs.

III. Analysis

A. Experimental Setup

We used a PC with two AMD Opteron 6376 2.3 GHz processors (16 cores for each) and 320 GB of memory. For virtualization software, we used Xen 4.4.0 [3] and ran Linux 3.13.0 in Dom0, which is a special VM for managing VMs. We created a target VM with 16 vCPUs and 4 GB of memory and assigned 16 CPU cores to it. In the VM, we ran Linux 3.13.0 as a fully virtualized guest. When we used this PC as a PM without Xen, we ran Linux 3.13.0.

For a parallel application, we used the calculation of the Fibonacci number that was parallelized using Tascell [1]. Tascell is a framework for backtracking-based dynamic load balancing. Whenever a worker is requested by another idle worker, it temporarily backtracks and restores the oldest task-spawnable state. Thereby it spawns as a large task as possible. For example, the calculation of `fib(10)` needs the recursive calculation of `fib(9)` and `fib(8)`. If worker A is requested by worker B while it is calculating `fib(9)`, it spawns `fib(8)` as a new task and then worker B calculates it. Fibonacci is a representative of searching applications. We call Fibonacci parallelized by Tascell as tascell-fib. In our experiment, tascell-fib calculated `fib(48)`.

For all experiments, we ran tascell-fib 10 times and calculated the average.

B. Comparison between a VM and a PM

To examine the virtualization overhead in the execution of a parallel application, we measured the execution time of tascell-fib in a VM and a PM. The execution time includes only parallel computation, not application setup such as dynamic loading of shared libraries. We changed the number of threads that tascell-fib used from 1 to 16. We assigned 16 cores in only one processor to a VM, considering NUMA nodes. Fig. 1 shows the results. Note that this figure uses a logarithmic scale for both axes. We also show the ideal execution time calculated from the execution time in a single thread. The coefficients of variance (CV) were 1.4% for a VM and 11% for a PM at maximum.

These results show that fixed virtualization overhead was 24% at least from the performance in a single thread. This overhead comes from vCPU scheduling and memory virtualization using Intel Extended Page Table (EPT). In both a VM and a PM, the performance improvement slowed down as the number of threads increased. However, scalability in more than 4 threads was largely different. Whereas the performance improvement gradually slowed down in a PM, it steeply slowed down in a VM. The performance difference was maximized in 8 threads and the execution time in a VM was 72% longer than that in a PM. Strangely, the performance improved largely again after the number of threads exceeded 8. Then the improvement slowed down again in more than 12 threads.

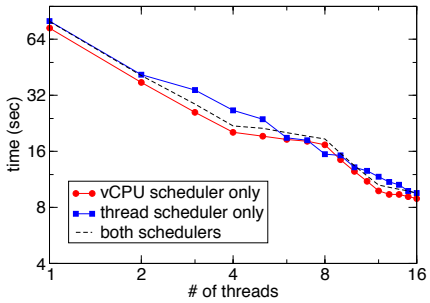


Figure 3. The performance with one scheduler disabled.

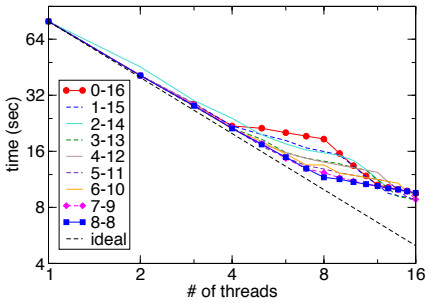


Figure 4. The performance in various pCPU distributions.

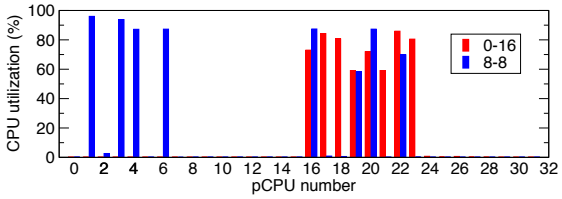


Figure 5. The pCPU usage in 8 threads for distributions 0-16 and 8-8.

C. Identifying a Problematic Scheduler

The strange scalability can be due to the thread scheduler in the operating system, the vCPU scheduler in the hypervisor, or both. To examine which scheduler causes such a phenomenon, we first disabled the thread scheduler. For this purpose, we configured tascell-fib so that it assigned each thread to a different vCPU exclusively. This CPU affinity can prevent the operating system from scheduling threads to arbitrary vCPUs. Next, we disabled the vCPU scheduler by assigning pCPUs to vCPUs one by one. This vCPU affinity can prevent the hypervisor from scheduling vCPUs to arbitrary pCPUs.

Fig. 3 shows the execution time of tascell-fib for various numbers of threads. When the thread scheduler was disabled, the execution time became 6-9% shorter than that when both schedulers were enabled. However, scalability was almost the same in both cases. On the other hand, when the vCPU scheduler is disabled, the scalability was different from that when both schedulers were enabled. As the number of threads increased, the performance improvement slowed down more gradually like a PM in Fig. 1. From these results, we conclude that only the vCPU scheduler is problematic.

D. Impact of pCPU Distribution

We assumed that the reason of the strange scalability in a VM was pCPU assignment from only one processor. To examine whether this assumption is correct or not, we changed pCPU distribution m-n between two processors by configuring the vCPU affinity. m-n means that m and n pCPUs are assigned from processor 1 and 2, respectively. We assigned 16 pCPUs in total from two processors to a VM. pCPUs in each processor were consecutively assigned from the lowest number (0 and 16, respectively). Fig. 4 shows the execution time for nine different distributions. The CV was less than 10% at maximum, except for 2-14.

These results show that the performance was largely affected by pCPU distribution in more than 4 threads. 0-16 is the worst until the number of threads exceeded nine. One of the best distributions was 8-8, which means that pCPUs are evenly assigned from two processors. The performance improved almost ideally until 8 threads. Compared with 0-16, this distribution improved the performance by up to 38%. However, the performance improvement still slowed down in more than 8 threads. Interestingly, the performance in 16 threads was better than 8-8 when the m and n in distributions are odd. For example, 7-9 was similar to 8-8 and outperformed 8-8 in 16 threads. Instead, the performance was worse than 8-8 in 8 threads.

E. Difference of pCPU Usage

To examine the difference of pCPU usage, we measured the pCPU utilization in 8 threads, using xenmon. We focused on 0-16 and 8-8, where the performance difference was the largest in 8 threads. Fig. 5 shows the results. Xen assigned the number 0-15 to pCPUs in processor 1 and the number 16-31 to those in processor 2. In 8-8, four pCPUs (1, 3, 4, and 6) in processor 1 and four pCPUs (16, 19, 20, and 22) in processor 2 were mainly used. In 0-16, in contrast, eight pCPUs (16-23) in processor 2 were used. These results match the configured pCPU assignments. However, only the first half of the pCPUs in processor 2 was densely used in 0-16, whereas both processors were sparsely used in 8-8.

F. Resource Conflicts between pCPUs

From the difference of pCPU usage between 0-16 and 8-8, we suspected that the reason why performance improvement slowed down was resource conflicts between pCPUs. Opteron 6376 processors we used adopt the Clustered Multi-Thread microarchitecture. In this architecture, two cores are packaged as a *module* and share several resources such as an instruction decoder, L2 cache, and a floating-point unit (FPU). Therefore, the performance can degrade when two cores in a module are competing. To examine such competing modules, we measured the pCPU utilization every

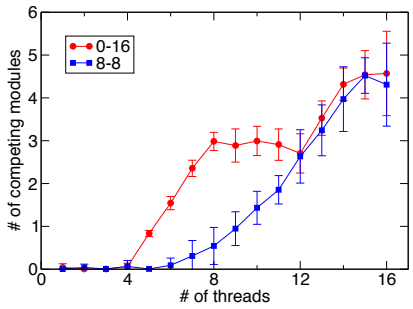


Figure 6. The number of competing modules.

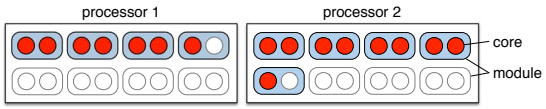


Figure 7. The pCPU usage by 16 threads in 7-9.

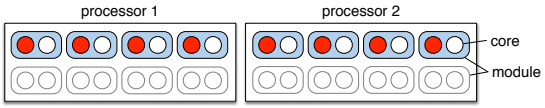


Figure 8. The pCPU usage by 8 threads in 8-8.

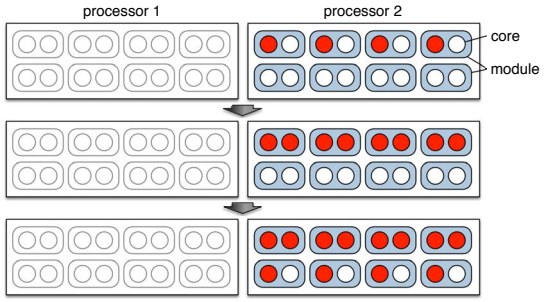


Figure 9. The change of the pCPU usage in 0-16.

second and calculated the average number of modules in which the two cores were used simultaneously. We multiplied the utilization of two pCPUs belonging to the same module and considered it the ratio of module contention. For example, when 50% of one core and 50% of the other core are used, the ratio is 0.25.

Fig. 6 shows the average number of competing modules and the standard deviation in distributions 0-16 and 8-8. Compared with Fig. 4, the number of competing modules has a strong correlation with the performance. When the number of threads increases, the performance improvement is small if the number of competing modules also increases. Otherwise, the performance improves almost ideally in proportion to the number of threads. For 0-16, the performance did not improve well between 4 and 8 threads and between 12 and 16 threads because the number of competing modules increased. In contrast, the number of competing modules did not increase between 8 and 12 threads and therefore the performance improvement was significant. For 8-8, the change of the number of competing modules was largely different. This caused the difference of scalability between 0-16 and 8-8.

It should be noted that, when m and n in distributions are odd, the number of used modules becomes nine, not eight, in 16 threads. For 7-9, for example, four modules are used in processor 1 and five modules are used in processor 2, as illustrated in Fig. 7. Therefore the number of competing modules is only seven. This means that only one core is used for the remaining two modules. From this reason, the performance in 16 threads is better than when m and n are even.

G. Conflict in vCPU Scheduling

The next research question is why module contention happens so early only in 0-16. First, we inspected how CPU cores were used when we increased the number of threads. For 8-8, only one core in each module was used until 8 threads, as illustrated in Fig. 8. This means that CPU cores were used so that they did not compete. In more than 8 threads, there were no modules with two idle cores because only cores in 8 modules were assigned to a VM. Therefore the other core in each module with one busy core was used. This pCPU usage means that the vCPU scheduler in Xen correctly considers modules in Opteron. For 0-16, on the other hand, the other core in a module with one busy core was used when the number of threads exceeded four, as illustrated in Fig. 9. In more than 8 threads, modules with two idle cores were used again. This pCPU usage is largely different from that in 8-8.

To examine why the vCPU scheduler performed such pCPU usage, we investigated the source code of Xen. Consequently, we found that the root cause was the conflict between the vCPU and NUMA-node affinities. Opteron 6376 has two NUMA nodes and each node consists of eight CPU cores. In our PC, Xen assigned the numbers 0 and 1 to the nodes in processor 1 and the numbers 2 and 3 to those in processor 2. When a VM is created by the xl command, the command automatically sets the node affinity to the VM. In our experiments, nodes 0 and 2 were assigned to a VM.

In addition to such automatically set node affinity, we manually set the vCPU affinity to a VM. In our PC, nodes 0, 1, 2, and 3 contain pCPUs 0-7, 8-15, 16-23, and 24-31, respectively. For distribution 8-8, we assigned pCPUs 0-7 from processor 1 and 16-23 from processor 2 to a VM. These pCPUs were contained in nodes 0 and 2. In this case, the vCPU affinity did not conflict with the node affinity. Therefore, the vCPU scheduler first used eight non-competing cores in nodes 0 and 2 and then competing cores in these nodes, as shown in Fig. 8.

For 0-16, on the other hand, we assigned pCPUs 16-31

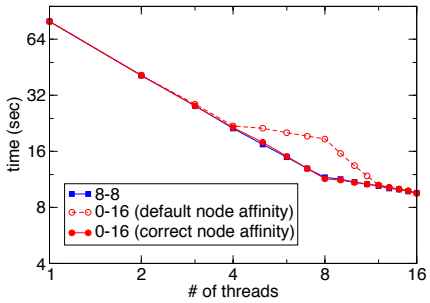


Figure 10. The performance of 0-16 with the correct node affinity.

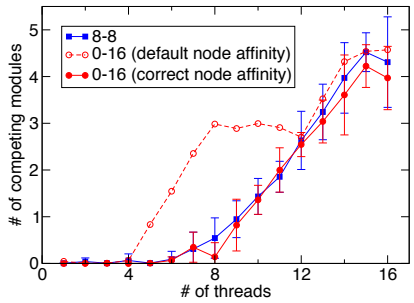


Figure 11. The number of competing modules with no node affinity.

from processor 2 to a VM. These pCPUs were contained in nodes 2 and 3. This vCPU affinity conflicted with the node affinity, which assigned nodes 0 and 2. Since pCPUs in node 2 preceded those in node 3 due to the node affinity, the vCPU scheduler first used four non-competing cores in node 2 and then four competing cores in node 2, as shown in Fig. 9. After that, it used four non-competing cores in node 3 and then four competing cores in node 3.

IV. Improving Scalability

We provide three methods for avoiding resource conflicts between pCPUs and/or conflict between the vCPU and node affinities in vCPU scheduling.

A. Disabling the Node Affinity

To avoid conflict between vCPU and node affinities, we have developed a tool for disabling the node affinity. Specifically, the tool sets the affinity to all the nodes by issuing a hypercall to the hypervisor. When we set the vCPU affinity after that, the corresponding nodes are automatically reset to the VM as the correct node affinity. For example, the node affinity was set to nodes 2 and 3 for the vCPU affinity to pCPUs 16-31. It is more desirable to disable the node affinity when we set the vCPU affinity, but this requires modification to Xen itself. Therefore we have developed an independent tool.

Using this tool, we measured the performance for 0-16 with the correct node affinity. Fig. 10 and Fig. 11 show the execution time and the number of competing modules, respectively. The performance improved by disabling the node affinity and became almost the same as that in 8-8. However, the degree of module contention was slightly different. In 0-16 with the correct node affinity, the number of competing modules was less than that in 8-8 when the number of threads was more than 12.

It should be noted that the node affinity is not set when we set the vCPU affinity, not manually, in the configuration file of a VM. However, this is not a good way because the vCPU affinity strongly depends on processors. When we migrate a VM to another host, the numbers of processors and their cores can be different. According to processors, we should change the vCPU affinity. For example, if a destination host has only one processor with 16 cores, the vCPU affinity to pCPUs 16-31 is invalid.

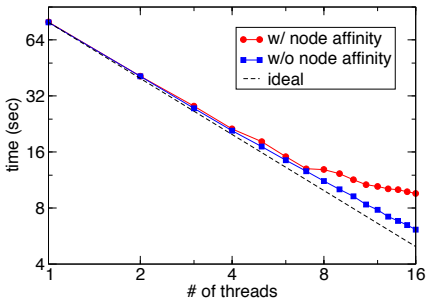


Figure 12. The performance with no vCPU affinity.

B. Disabling the vCPU Affinity

As another method, we disabled the vCPU affinity simply by not setting it. By default, the node affinity is automatically set to a VM when we do not set the vCPU affinity. Therefore, the VM could use pCPUs 0-7 and 16-23, which were corresponding to nodes 0 and 2. This pCPU assignment is the same as 8-8. In addition, when we disabled the node affinity using our tool, the VM could use all the pCPUs 0-31. Fig. 12 and Fig. 13 show the execution time and the number of competing modules, respectively, for both cases. With the node affinity, the performance was similar to that for 8-8 although it was not exactly the same.

With no node affinity, on the other hand, the performance improved well until 16 threads. Theoretically, module

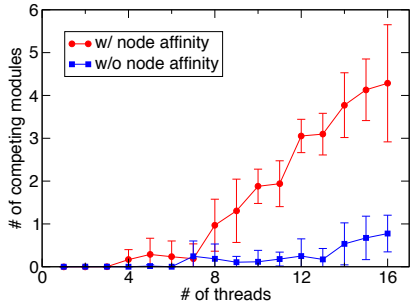


Figure 13. The number of competing modules with no vCPU affinity.

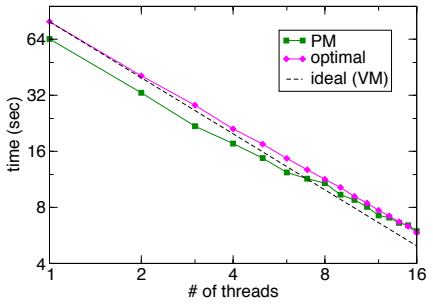


Figure 14. The performance with the optimal vCPU affinity.

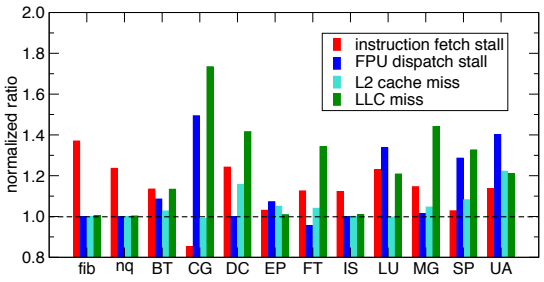


Figure 16. The severity of resource conflicts in 0-16 over 8-8 in 8 threads.

contention should be avoided completely because all the 16 modules in two processors can be used. However, the number of competing modules was not zero, particularly, between 14 and 16 threads.

C. Optimal vCPU Affinity

To avoid module contention completely, we assigned only one core per module to a VM using the vCPU affinity. Specifically, we used only pCPUs with even numbers. In this case, the node affinity was not substantially affected because usable pCPUs were across all the nodes. Fig. 14 shows the result in this optimal vCPU affinity. The CV was 2.7% at maximum. The number of competing modules became zero and the performance improved well until 16 threads. In 16 threads, the performance was 4.6% higher than that with no vCPU and node affinities.

Even compared with the performance in a PM, that in a VM was slightly better in 16 threads. This is because CPU assignment in a PM was not optimal. In the default CPU scheduling in a PM, the number of competing modules was 4.0. When we set the CPU affinity using Linux cgroup as it was optimal, it could also avoid resource conflicts in each module. As a result, the performance in a PM became the same as that in a VM when we used 16 threads.

V. Other Parallel Applications

For other parallel applications, we examined whether the conflict between the vCPU and node affinities occurred and whether the optimal vCPU affinity improved performance. We executed each application in distributions 0-16, 8-8, and the optimal vCPU affinity. We ran each application 10 times and calculated the average.

A. N-Queen Parallelized by Tascell

As another application parallelized by Tascell, we ran N-Queen, called tascell-nq. Fig. 15a shows the execution time when N was 16. Like tascell-fib, the performance in 0-16 was also lower than that in 8-8, but the difference was smaller than that in tascell-fib. The performance degradation due to using 0-16 was 24% at maximum in tascell-nq, while that was 38% in tascell-fib.

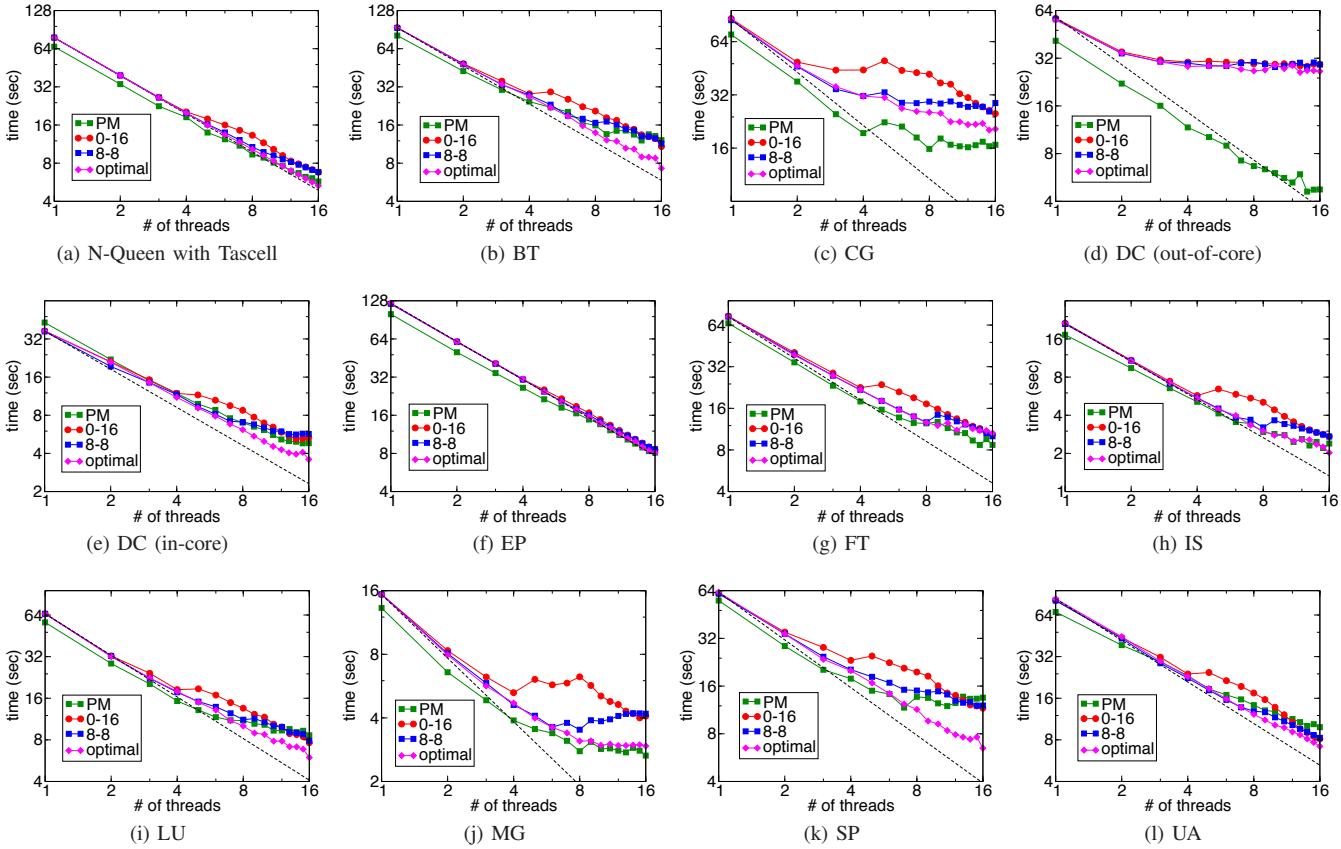
To examine the cause, we measured detailed resource conflicts in 8 threads using hardware performance counters. Fig. 16 shows the ratio of the number of events in 0-16 to that in 8-8 for shared resources in each module. The ratio is normalized by the number of events per second to compare different resources and benchmarks. For tascell-fib and tascell-nq, the cause of the performance degradation in 0-16 was instruction fetch stalls in the instruction decoders. In tascell-nq, the ratio in the instruction fetch stall was lower than that in tascell-fib. This is the reason why the performance difference was smaller.

Like tascell-fib, the optimal vCPU affinity improved the performance. Surprisingly, in 16 threads, the performance in a VM was 7.5% higher than that in a PM. At that time, the number of competing modules was 5.2 and this was larger than that in tascell-fib. When we set the optimal CPU affinity in a PM, the performance became 2.3% higher in 16 threads than that in a VM.

B. NAS Parallel Benchmarks

Without Tascell, we ran the NAS Parallel Benchmarks [2] using OpenMP. These benchmarks are different from tascell-fib and tascell-nq in that most of them use floating point and access a larger amount of memory. Fig. 15(b)-(l) show the results of 10 benchmarks. Note that we measured DC in both the out-of-core and in-core modes, which use disks and only memory to store results, respectively. In most of the benchmarks, the performance in 0-16 was less than that in 8-8, except for DC in the out-of-core mode and EP. For DC in that mode, the scalability in a VM was very low as shown in Fig. 15(d). This is due to the overhead of storage virtualization. Since we focus on the impact of CPU virtualization in this paper, we consider only the in-core mode below. In contrast, EP mainly executed floating-point computation with little communication and therefore scaled very well without regard to pCPU assignments. As in Fig. 16, the degree of resource conflicts was almost the same between 0-16 and 8-8 for EP.

According to Fig. 16, there are several causes for the performance differences between 0-16 and 8-8. For DC and LU, instruction fetch stalls were the cause of the performance degradation in 0-16. Unlike tascell-fib and tascell-



(a) N-Queen with Tascell (b) BT (c) CG (d) DC (out-of-core)

(e) DC (in-core) (f) EP (g) FT (h) IS

(i) LU (j) MG (k) SP (l) UA

Figure 15. The performance of various parallel applications.

nq, FPU dispatch stalls in the instruction decoders degraded the performance for CG, UA, LU, and SP. In addition, last-level cache (LLC) misses were another cause for CG, MG, DC, FT, and SP. In Opteron 6376, LLC is shared in each node, not in each module. In 8 threads, only one node was used in 0-16, while two nodes were used in 8-8. Therefore 0-16 was subject to LLC misses more. In contrast, the impact of L2 cache misses was limited.

In all the benchmarks, the performance in the optimal vCPU affinity was higher than that in 8-8 as well. Furthermore, in BT, DC, LU, SP, and UA, it outperformed even that in a PM when the number of threads was large. This is because the number of competing modules in a PM was between 0.7 and 4.1. When we used the optimal CPU assignment in a PM, the performance became slightly better than that in a VM. For CG, FT, and MG, on the other hand, the performance in a PM was higher than the optimal vCPU affinity by default. Since the number of competing modules for them was similar to the others, the root cause is not that the default CPU scheduling was better for them. We examined the causes of resource conflicts using hardware performance counters, but we could not find any strong correlation. Identifying this root cause is our future work.

VI. Related Work

There are several researches for thread schedulers that are aware of chip multi-threading (CMT). MASA [4] assists thread scheduling for Intel's Hyper-Threading by dynamically adjusting the CPU affinity. It detects resource conflicts in a CPU core (package) using the hardware performance counter and swaps two threads between two CPU cores to equalize the loads. Bulpin et al. [5] has implemented a similar thread scheduler in the Linux kernel. The scheduler calculates a performance ratio from single-thread and dual-thread executions in one CPU core and schedules threads to maximize throughput. Fedorova et al. [6] simulates a CMT processor to understand resource conflicts and proposes a CMT-savvy thread scheduler. The scheduler schedules threads with high cycles per instruction (CPI) and those with low CPI in one CPU core. These approaches can be also applied to the vCPU scheduler.

AMD's Clustered Multi-Thread has similarities to Intel's Hyper-Threading, but these have different performance bottlenecks. In the former microarchitecture, two CPU cores in one module share one instruction decoder and one L2 cache, while they have two independent integer units. In the latter, on the other hand, two threads in one CPU core have

two independent instruction decoders, while they share one integer unit and one L1/L2 cache. Therefore, the instruction decoder is a bottleneck only in AMD's processors, whereas the integer unit and the L1 cache are bottlenecks only in Intel's processors. In any cases, it is important to perform microarchitecture-aware vCPU scheduling.

There are two well-known problems on vCPU scheduling. These problems cause a long waiting time for lock waiters in VMs with multiple vCPUs. One problem is lock-holder preemption (LHP) [7]. A lock-holder vCPU can be preempted by the vCPU scheduler when vCPUs in other VMs are waiting for pCPU. If there is a lock waiter on another vCPU, the lock waiter has to wait for the lock holder to release the lock. However, the lock holder cannot release the lock until the lock-holder vCPU is rescheduled after other vCPUs use up pCPU. The other problem is vCPU stacking [8]. When a lock-holder vCPU is preempted, a lock-waiter vCPU can be scheduled on the same pCPU. In this case, the lock-holder vCPU can wait in the run queue on the same pCPU. Since the lock waiter is waiting for the lock holder to release a lock, the lock-waiter vCPU does nothing.

To solve these problems, co-scheduling has been proposed [9]–[11]. It schedules all the vCPUs of the same VM to sufficient number of pCPUs simultaneously. Whenever a lock-holder vCPU is preempted, lock-waiter vCPUs are also preempted. Each vCPU is scheduled to a different pCPU. However, co-scheduling introduces CPU fragmentation and priority inversion. Even if several pCPUs are available, vCPUs cannot use them until the sufficient number of pCPUs becomes available. Such available but unused pCPUs are wasted. In addition, a vCPU with higher priority can be scheduled after a vCPU with lower priority. If all the pCPUs are used, emergent vCPUs has to wait until the end of the current time slice.

Balance scheduling [8] spreads vCPUs of a VM on different pCPUs and prevents both lock-holder and lock-waiter vCPUs from entering the run queue of the same pCPU. This scheduling can avoid vCPU stacking, but it cannot prevent LHP. Demand-based scheduling [12] is scheduling driven by inter-processor interrupts (IPIs) between vCPUs. This can effectively reduce synchronization latency without sacrificing the throughput of non-communicating vCPUs. However, this scheduling lacks support for spinlock-based synchronization. VCPU-Bal [13] dynamically adjusts the total number of runnable vCPUs of VMs. Thereby, the hypervisor can assign each vCPU to a different pCPU and avoid LHP and vCPU stacking. One disadvantage is that the operating systems have to be modified.

VII. Conclusion

In this paper, we analyzed the root cause of the difference in scalability between a VM and a PM. Our main findings are (1) that resource conflicts occur between two CPU cores in each module of Opteron 6376 processors and (2) that strange scalability in a VM is caused by conflict between the vCPU and NUMA-node affinities in vCPU scheduling. On the basis of our observation, we provided three methods for avoid these conflicts. Furthermore, we confirmed that the optimal vCPU affinity was effective for not only applications parallelized by Tascell but also most of NAS Parallel Benchmarks.

Our future work is to develop a vCPU scheduler that can automatically disable the NUMA-node affinity according to workloads. To be comparable to the optimal vCPU affinity, the scheduler should completely avoid contention between two CPU cores in each module.

Acknowledgment

This work was supported in part by JSPS KAKENHI Grant Number 26280023.

References

[1] T. Hiraishi, M. Yasugi, S. Umatani, and T. Yuasa, "Backtracking-based Load Balancing," in *Proc. Symp. Principles and Practice of Parallel Programming*, 2009, pp. 55–64.

[2] NASA Advanced Supercomputing Division, "NAS Parallel Benchmarks," http://www.nas.nasa.gov/publications/npb.html.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proc. Symp. Operating Systems Principles*, 2003, pp. 164–177.

[4] J. Nakajima and V. Pallipadi, "Enhancements for Hyper-Threading Technology in the Operating System – Seeking the Optimal Scheduling," in *Proc. Workshop on Industrial Experiences with Systems Software*, 2002.

[5] J. Bulpin and I. Pratt, "Hyper-Threading Aware Process Scheduling Heuristics," in *Proc. USENIX Annual Technical Conf.*, 2005.

[6] A. Fedorova, C. Small, D. Nussbaum, and M. Seltzer, "Chip Multithreading Systems Need a New Operating System Scheduler," in *Proc. ACM SIGOPS European Workshop*, 2004.

[7] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski, "Towards Scalable Multiprocessor Virtual Machines," in *Proc. Conf. Virtual Machine Research and Technology Symposium*, 2004.

[8] O. Sukwong and H. S. Kim, "Is Co-scheduling Too Expensive for SMP VMs?" in *Proc. Conf. Computer Systems*, 2011, pp. 257–272.

[9] VMware, Inc., "Co-scheduling SMP VMs in VMware ESX Server," 2008.

[10] C. Weng, Z. Wang, M. Li, and X. Lu, "The Hybrid Scheduling Framework for Virtual Machine Systems," in *Proc. Intl. Conf. Virtual Execution Environments*, 2009, pp. 111–120.

[11] Y. Bai, C. Xu, and Z. Li, "Task-aware Based Co-scheduling for Virtual Machine System," in *Proc. Symp. Applied Computing*, 2010, pp. 181–188.

[12] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng, "Demand-based Coordinated Scheduling for SMP VMs," in *Proc. Intl. Conf. Architectural Support for Programming Languages and Operating Systems*, 2013, pp. 369–380.

[13] X. Song, J. Shi, H. Chen, and B. Zang, "Schedule Processes, not VCPUs," in *Proc. Asia-Pacific Workshop on Systems*, 2013.