Optimization of Parallel Applications under CPU Overcommitment

Tokiko Takayama
Kyushu Institute of Technology
tokiko@ksl.ci.kyutech.ac.jp

Kenichi Kourai
Kyushu Institute of Technology
kourai@ksl.ci.kyutech.ac.jp

***Abstract*—As cloud computing is widely used, even parallel applications run in virtual machines (VMs) of clouds. When CPU overcommitment is performed in clouds, physical CPU cores (pCPUs) can become less than virtual CPUs (vCPUs). In such a situation, it is reported that application performance degrades more largely than expected by the decrease of pCPUs available to each VM. To address this issue, several researchers have proposed optimization techniques of reducing the number of vCPUs assigned to each VM. However, their effectiveness is confirmed only in a limited VM configuration. In this paper, we have first investigated application performance under three configurations and revealed that the previous work cannot always achieve optimal performance. Then we propose *pCPU-Est* for improving application performance under CPU overcommitment. pCPU-Est dynamically optimizes the number of vCPUs on the basis of correlation between CPU utilization and execution time (*dynamic vCPU optimization*). In addition, it dynamically optimizes the number of application threads when possible (*thread optimization*). According to our experiments, dynamic vCPU optimization improved application performance by up to 42%, while thread optimization did by up to 72x.**

***Index Terms*—Virtual machines, virtual CPUs, CPU overcommitment, parallel applications, performance degradation, performance improvement**

1. Introduction

As the number of CPU cores is increasing, recent applications are parallelized to make their computation faster. The computation of an application is divided and is executed in parallel using multiple CPU cores. As cloud computing is widely used, such parallel applications often run in virtual machines (VMs) provided by public or private clouds, instead of directly in physical machines. In virtualized environments, each CPU core called a *physical CPU core (pCPU)* is virtualized and is provided as one or more *virtual CPUs (vCPUs)* to VMs. While there are a sufficient number of pCPUs, one pCPU is assigned to one vCPU. However, if clouds perform CPU overcommitment, the total number of vCPUs can exceed that of pCPUs. Under such a situation, it is reported that application performance degrades more largely than expected by the decrease of pCPUs available to each VM [1]–[3].

To address this issue, several researchers have proposed optimization techniques of reducing the number of vCPUs assigned to each VM [1]–[3]. These techniques can eliminate the root cause of lock-holder preemption [4] and vCPU stacking [5] by mapping one vCPU to only one pCPU. However, the previous work shows performance degradation under the shortage of pCPUs and performance improvement by the proposed methods only in a limited VM configuration. In this paper, we have first measured application performance and investigated the effectiveness of the previous work under three configurations on a pCPU shortage. Consequently, it was revealed that application performance was degraded under any configurations but the degree of the performance degradation depends on configurations. In addition, the number of vCPUs used in the previous work were often not optimal.

Therefore, we propose *pCPU-Est* for further improving the performance of parallel applications under CPU overcommitment. pCPU-Est provides two optimization techniques: *dynamic vCPU optimization* and *thread optimization*. Dynamic vCPU optimization determines the optimal number of vCPUs at runtime on the basis of correlations between CPU utilization and execution time. Thread optimization adjusts the number of application threads to overcome the limitation of vCPU optimization. It should be noted that thread optimization is not always applicable because it depends on applications and situations. For these two optimization techniques, pCPU-Est enables the number of pCPUs available to each VM to be estimated. When vCPU affinity is set to VMs, this estimation is not straightforward.

We have implemented pCPU-Est in Xen 4.4 [6] and examined whether pCPU-Est could improve the performance of the NAS parallel benchmarks [7]. The experimental results show that dynamic vCPU optimization could achieve larger performance improvement than the previous work in several benchmarks. Compared with the previous work [1], the performance was improved by up to 42%. This is because pCPU-Est could determine the number of vCPUs so that the number was nearer the optimal than that in the previous work. Although dynamic vCPU optimization was not effective for the rest of the benchmarks, thread optimization could improve performance in all the benchmarks. The maximum performance improvement was 72x.

The organization of this paper is as follows. Section 2 describes the existing optimization technique for CPU over-

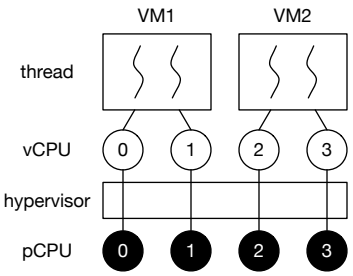


Figure 1: CPU assignment in a virtualized system.

commitment. Section 3 shows the results of our preliminary experiments on the previous work. Section 4 proposes pCPU-Est for further performance improvement. Section 5 shows the results of our experiments on pCPU-Est. Section 6 describes related work and Section 7 concludes this paper.

2. CPU Overcommitment

In virtualized systems, *physical CPU cores (pCPUs)* are virtualized and are provided to VMs as *virtual CPUs (vCPUs)*. While there are a sufficient number of pCPUs, one pCPU can be assigned to only one vCPU, as illustrated in Fig. 1. This assignment can be changed by the vCPU scheduler in the hypervisor. For example, pCPU 0 is assigned to vCPU 0 in Fig. 1, but it can be assigned to one of the other vCPUs later. To fix this assignment, the hypervisor provides a mechanism called *vCPU affinity*. Using this mechanism, each pCPU is always assigned to only one specified vCPU for strict performance isolation. When parallel applications are running in a VM, their threads are assigned to vCPUs by the process scheduler in the operating system.

Virtualized systems often overcommit CPUs and run more vCPUs than the number of pCPUs because all the vCPUs do not always use 100% of all the pCPUs. In general, CPU utilization is 5–20% in servers [8]. However, since parallel applications often occupy CPUs, CPU overcommitment easily leads to the shortage of pCPUs. For example, after under-utilized VMs are consolidated by VM migration, some of them can start running parallel applications. One option is to migrate such VMs to hosts with sufficient pCPUs, but it is often difficult because VM migration is also heavyweight and deteriorates the situation.

When a parallel application runs with insufficient pCPUs, its performance degrades more largely than expected by the decrease of pCPUs available to each VM [1]–[3]. For example, when the number of pCPUs available to a VM is reduced from 16 to 8, it is expected that the performance of a parallel application using 16 threads becomes half. However, the performance often degrades to less than half. The root causes are lock-holder preemption (LHP) [4] and vCPU stacking [5]. LHP is caused when a vCPU holding a spin lock is preempted and vCPUs waiting for the lock cannot run. vCPU stacking is caused when a pCPU is assigned to a vCPU waiting for a spin lock before one holding the lock.

To solve this problem, several optimization techniques have been proposed [1]–[3]. All of these techniques reduce the number of vCPUs of each VM according to that of pCPUs available to the VM. However, the previous work investigates performance degradation only when pCPUs are *shared* between *two* VMs and performance improvement by the proposed techniques at that time. Therefore, it is unclear *how* the performance is degraded and is improved at various VM configurations, e.g., when pCPUs are shared among *many* VMs, when CPU utilization of each VM is limited, and when pCPU assignment to each VM is reduced.

In addition, the previous work basically calculates the reduced number of vCPUs of VM_i as follows:

$$N_{all\_cpu} \times \frac{W_i}{\sum_{0 \le k < N_{vm}} W_k}$$

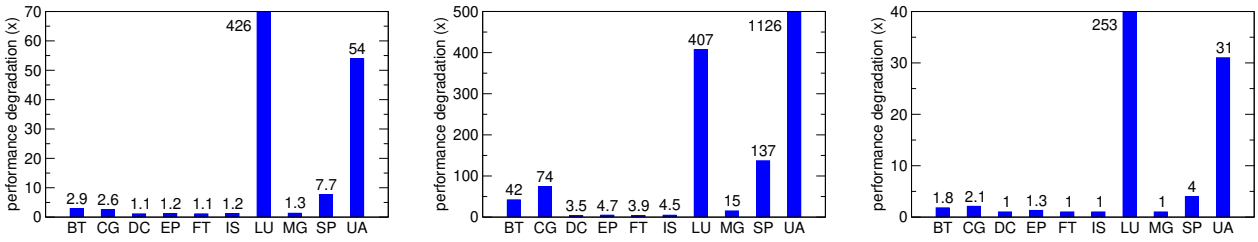
where N_{all_cpu} is the total number of pCPUs, N_{vm} is the number of VMs, and W_i is the weight of VM_i. The weight is used as CPU shares and is configured to relatively allocate CPU time to VMs. The calculated value means the number of pCPUs available to VM_i. If the weights of all the VMs are the same, the number of vCPUs is adjusted to the total number of pCPUs divided by the number of VMs. However, the previous work does not discuss whether this calculation is optimal or not. At least, this is not optimal when vCPU affinity is set so that each vCPU cannot use pCPUs equally.

3. Exploratory Experiments

We first conducted experiments to measure performance degradation under various VM configurations on a pCPU shortage. Then, we examined performance improvement by one of the previous work in Section 2, named VCPU-Bal [1], under these configurations. In addition, we experimentally searched for the optimal number of vCPUs.

We used three VM configurations. The configuration of *CPU sharing* shares pCPUs between VMs by assigning one pCPU to vCPUs of multiple VMs. The proportion of a pCPU available to each vCPU is determined by the weights configured to VMs. Note that surplus pCPUs can be used by arbitrary vCPUs when vCPUs do not use up the pCPUs assigned to them. The configuration of *CPU limit* limits the CPU utilization of VMs for sharing pCPUs between VMs. By default, each VM can use 100% multiplied by the number of pCPUs available to the VM. The vCPU scheduler determines how many pCPUs each VM uses and what percentage of each pCPU is used. The configuration of *CPU reduction* reduces pCPU assignment to VMs and assign one pCPU only to vCPUs of a specific VM using vCPU affinity. Each VM can occupy the entire pCPUs, while one pCPU is shared between multiple vCPUs of the VM.

We used a PC with two AMD Opteron 6376 processors (16 cores for each) and 320 GB of memory. We ran Xen 4.4.0 [6] with the default credit scheduler and VMs with 16 vCPUs and 4 GB of memory on top of it. Since these Opteron processors share several hardware units between adjacent two cores, we used only 16 cores in total for the VMs so that application performance was maximized [9]. In



(a) CPU sharing (b) CPU limit (c) CPU reduction

Figure 2: The maximum performance degradation on a pCPU shortage (D_{base}).

the VMs, we ran Ubuntu 14.04 and the Linux kernel 3.13. As parallel applications, we ran all the benchmarks in the NAS Parallel Benchmarks (NPB) [7] using OpenMP [10]. Those problem sizes were BT.A, CG.B, DC.A, EP.B, FT.B, IS.C, LU.A, MG.C, SP.A, and UA.A. We ran each benchmark for each VM configuration ten times and calculated the average and standard deviation.

3.1. Performance Degradation

As a baseline, we examined performance degradation when the three VM configurations were applied. The performance degradation is defined as

$$D_{base} = \frac{T_{base}}{E} \tag{1}$$

where T_{base} is the execution time measured using pCPUs available under each VM configuration and E is the execution time expected under that VM configuration. E is the possible shortest execution time and is calculated from the execution time T_{min} measured when one VM can use all the pCPUs. Note that the performance degrades more largely as the value of D_{base} becomes bigger and the value of 1.0 means no degradation.

For the configuration of CPU sharing, we increased the number of VMs running in one host, N_{vm}, from one to eight and measured the execution time for each benchmark. The previous work measured performance degradation when only two VMs were run. In this configuration, the expected execution time is defined as

$$E_{share} = T_{min} \times N_{vm}. \tag{2}$$

For example, when the number of VMs is increased from one to two, E_{share} becomes twice. Fig. 2(a) shows the maximum values of the D_{base} among various numbers of VMs sharing pCPUs for each benchmark. The performance of LU was 426x lower than expected when eight VMs were run. In contrast, the performance degradation of half the benchmarks was not so large. To compare the standard deviations of T_{base} between different degrees of CPU sharing, we calculated the coefficient of variation (CV), which is the standard deviation divided by the average. In this experiment, the CV was 0.17 at maximum and was relatively large.

For the configuration of CPU limit, we decreased the upper limit of CPU utilization, U, from 1600% to 100% and measured the execution time. We ran only one VM so that the VM could fully use 16 pCPUs when the limit was set to 1600%. The previous work did not measure performance degradation under this configuration at all. In this configuration, the expected execution time is defined as

$$E_{limit} = T_{min} \times \frac{1600}{U}. \tag{3}$$

For example, when the CPU limit is decreased from 1600% to 800%, E_{limit} becomes twice. Fig. 2(b) shows the maximum values of D_{base} among various CPU limits. Compared with the configuration of CPU sharing, performance degradation became larger except for LU. At least, the performance was degraded by 247%. The performance of UA was 1126x lower than expected when the CPU limit was set to 100%. The CV was 0.25 at maximum and was larger than in the configuration of CPU sharing.

For the configuration of CPU reduction, we decreased the number of pCPUs assigned to a VM, N_{cpu}, from 16 to 1 and measured the execution time. We ran only one VM so that the VM could fully use 16 pCPUs assigned. The previous work did also not measure performance degradation under this configuration. In this configuration, the expected execution time is defined as

$$E_{reduce} = T_{min} \times \frac{16}{N_{cpu}}. \tag{4}$$

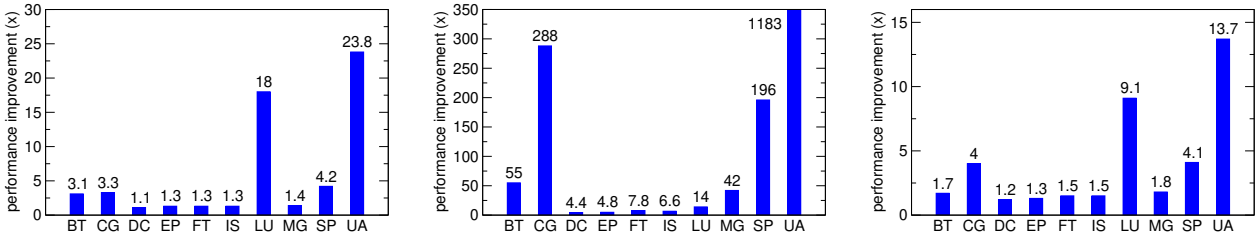
For example, when the number of assigned pCPUs are reduced from 16 to 8, E_{reduce} becomes twice. Fig. 2(c) shows the maximum values of D_{base} among various pCPU assignments. Compared with the other two configurations, the performance degradation was relatively small. However, the performance of LU was still 253x lower when 4 pCPUs were assigned. The CV was 0.16 at maximum.

3.2. Performance Improvement by VCPU-Bal

We examined performance improvement by VCPU-Bal to the baseline under the three VM configurations. The performance improvement is defined as

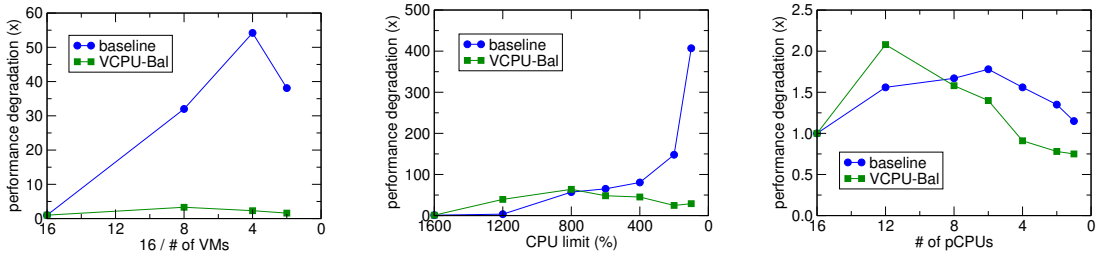
$$I_{bal/base} = \frac{D_{base}}{D_{bal}} \tag{5}$$

where D_{bal} is the performance degradation that is still left even when VCPU-Bal is applied. Note that the performance



(a) CPU sharing (b) CPU limit (c) CPU reduction

Figure 3: The maximum performance improvement by VCPU-Bal ($I_{bal/base}$).



(a) CPU sharing (UA) (b) CPU limit (LU) (c) CPU reduction (BT)

Figure 4: The detailed comparison of performance degradation in VCPU-Bal (D_{bal} and D_{base}).

improves more largely as the value of $I_{bal/base}$ becomes bigger. Like D_{base}, D_{bal} is defined as

$$D_{bal} = \frac{T_{bal}}{E} \tag{6}$$

where T_{bal} is the execution time measured with the optimization of VCPU-Bal.

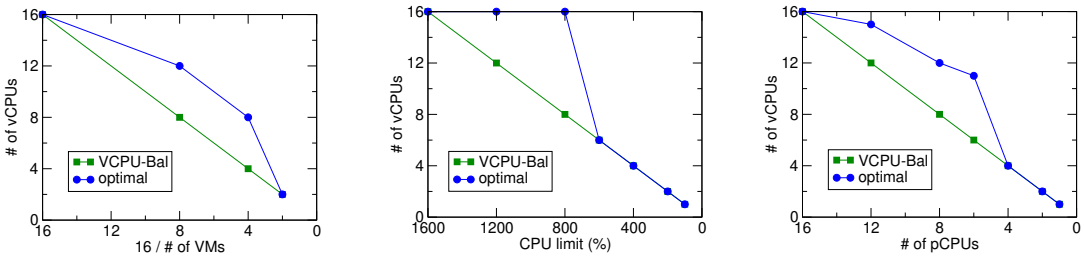
For the configuration of CPU sharing, we decreased the number of vCPUs of each VM to that of pCPUs available to the VM and measured the execution time for each benchmark. For example, when two VMs were run, the number of vCPUs was adjusted from 16 to 8 for each VM. The previous work showed performance improvement only for two VMs. Fig. 3(a) shows the maximum values of $I_{bal/base}$ for each benchmark. VCPU-Bal could improve the performance of all the ten benchmarks. In several benchmarks, the performance improvement exceeded the degradation in Fig. 2(a) because the measured execution time was shorter than the expected time. This happened when an application did not scale well. In LU and UA, the performance degradation was large, while the performance improvement was also large. For UA, Fig. 4(a) shows the detailed comparison between D_{bal} and D_{base}. VCPU-Bal could always improve the performance for any numbers of VMs. The performance was improved more largely when the number of VMs was more than two.

For the configuration of CPU limit, we decreased the number of vCPUs of a VM similarly and measured the execution time. For example, when the CPU limit of a VM was 800%, the number of vCPU was adjusted to eight. Fig. 3(b) shows the maximum values of $I_{bal/base}$. While the performance was degraded more largely than in the configuration of CPU sharing, it was improved more largely by VCPU-Bal. In particular, the performance improvement of UA was very large. Unlike the configuration of CPU sharing, VCPU-Bal could improve the performance of BT, CG, MG, and SP largely, but it could not improve that of LU so largely. Fig. 4(b) compares D_{bal} and D_{base} for LU in detail. As we configured CPU limit to a lower value, VCPU-Bal could improve the performance more largely. This is because the performance degradation became extremely large when the CPU utilization was limited more. However, when CPU limit was 1200%, the performance was rather 11x lower by reducing the number of vCPUs.

For the configuration of CPU reduction, we also decreased the number of vCPUs of a VM according to that of pCPUs assigned to the VM and measured the execution time. As shown in Fig. 3(c), the result was similar to that in the configuration of CPU sharing. Fig. 4(c) shows the detailed result of BT. The improvement was small in BT because the performance did not degrade so largely even without the optimization by VCPU-Bal. In addition, when the number of pCPUs was 12, the performance was 34% lower than that without the optimization. In any configurations, the CV of T_{bal} was 0.09 at most and was quite small.

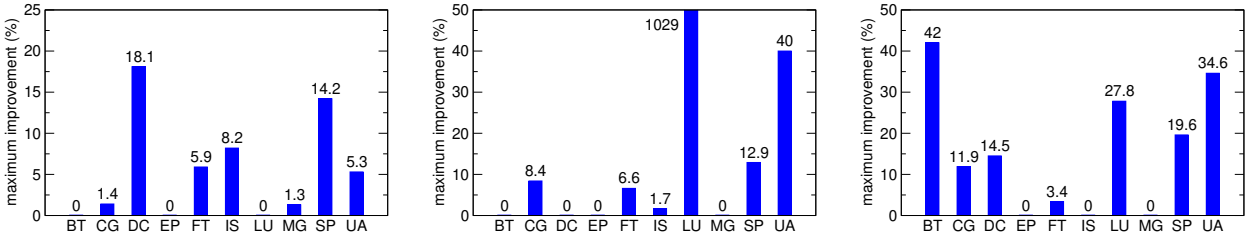
3.3. Optimal Number of vCPUs

From the results in the previous section, it was revealed that the performance rather degraded in several cases when VCPU-Bal was applied. Therefore, we thoroughly measured the performance for all the numbers of vCPUs and identified the optimal number. For the configuration of CPU sharing, the optimal number of vCPUs was exactly the same as the number of pCPUs available to each VM in BT, EP, and LU. However, that was often larger in the other seven



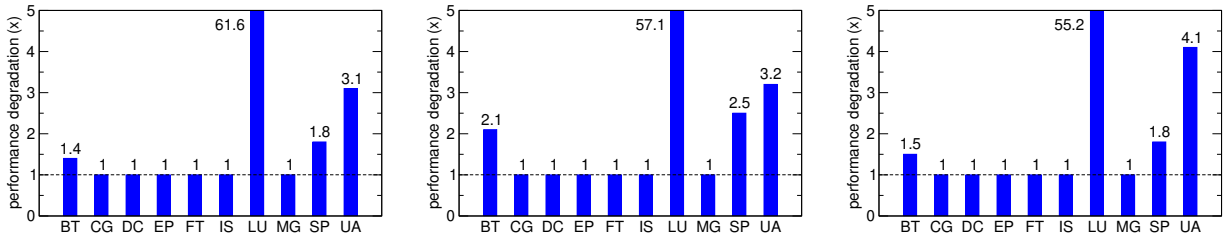
(a) CPU sharing (UA) (b) CPU limit (LU) (c) CPU reduction (BT)

Figure 5: The optimal number of vCPUs.



(a) CPU sharing (b) CPU limit (c) CPU reduction

Figure 6: The maximum possible improvement from VCPU-Bal by the optimal vCPU assignment ($I_{opt/bal}$).



(a) CPU sharing (b) CPU limit (c) CPU reduction

Figure 7: The maximum performance degradation even under the optimal vCPU assignment (D_{opt}).

benchmarks. For example, as shown in Fig. 5(a), the optimal number in UA was larger except when the number of VM was one and eight.

For the configuration of CPU limit, the optimal number of vCPUs was equal to the number of pCPUs available to a VM in BT, EP, DC, and MG. Unlike in the configuration of CPU sharing, the optimal number was much larger in LU when the CPU limit was larger than 600%, as shown in Fig. 5(b). In CG, FT, and IS, the optimal number was slightly smaller only when the CPU limit was 1200%. For the configuration of CPU reduction, the optimal number of vCPUs was different from the number of assigned pCPUs, except for EP, IS, and MG. As shown in Fig. 5(c), the larger number was optimal in most cases. In CG, FT, and LU, the slightly smaller number was sometimes optimal.

The maximum possible performance improvement from VCPU-Bal is defined as

$$I_{opt/bal} = \frac{D_{bal}}{D_{opt}} \tag{7}$$

where D_{opt} is the performance degradation that still exists even when the optimal number of vCPUs obtained is applied. D_{opt} is defined as

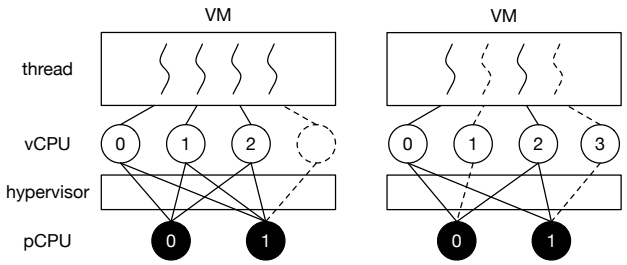
$$D_{opt} = \frac{T_{opt}}{E} \tag{8}$$

where T_{opt} is the execution time measured using the optimal number of vCPUs.

Fig. 6 shows the maximum values of $I_{opt/bal}$. The performance could be further improved by 18%, 1029%, and 42% at best for the three VM configurations, respectively. However, even the optimal vCPU assignment cannot improve the performance sufficiently in several benchmarks. Fig. 7 shows the maximum values of D_{opt} under the optimal vCPU assignment. It is shown that the performance of LU is still approximately 60x lower in any configurations. In BT, SP, and UA, the performance is also several times lower. The CV of T_{opt} was 0.08 at most and was small.

4. pCPU-Est

We propose pCPU-Est for further improving the performance of parallel applications under CPU overcommitment.



(a) vCPU Optimization (b) Thread Optimization

Figure 8: Two optimizations in pCPU-Est.

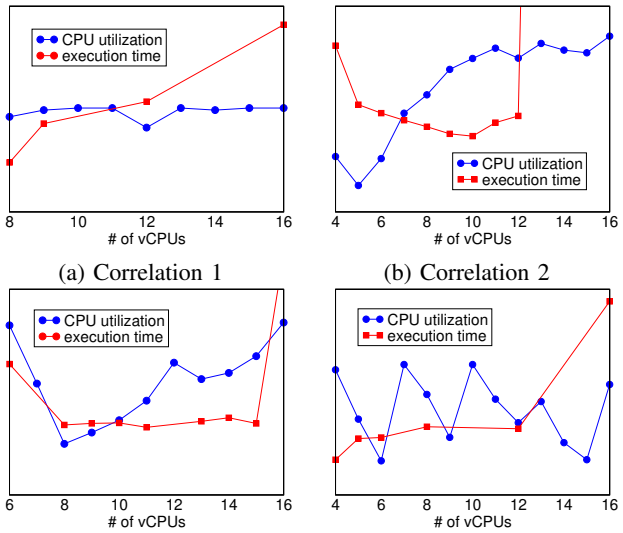
pCPU-Est provides two optimization techniques: *dynamic vCPU optimization* and *thread optimization*, which are illustrated in Fig. 8. Dynamic vCPU optimization adjusts the number of vCPUs assigned to a VM at runtime. Thread optimization adjusts the number of threads used by an application in a VM. For these two optimization techniques, pCPU-Est enables the number of pCPUs available to a VM to be correctly estimated even if vCPU affinity is set to VMs.

4.1. Dynamic vCPU Optimization

To dynamically optimize the number of vCPUs assigned to a VM using runtime information, we focused on the CPU utilization of a VM. As a preparatory experiment, we changed the number of vCPUs one by one and measured the execution time of benchmarks and the CPU utilization of a VM. As a result, we found that there are several correlations between the CPU utilization and the execution time. Fig. 9 shows four typical correlations, which were categorized on the basis of changes in CPU utilization when the number of vCPUs increased. Note that we omit data when the number of vCPUs is less than that of pCPUs available to a VM because we could not find clear correlation for less numbers of vCPUs.

Fig. 9(a) is the case where CPU utilization is constant. Larger the number of vCPUs is, lower the performance is. A VM uses up available pCPUs and the efficiency of pCPUs cannot increase even if the number of vCPUs increases. Rather, when the number of vCPUs is larger, performance degradation is caused due to LHP and vCPU stacking. Fig. 9(b) is the case where CPU utilization tends to increase and then becomes constant. When the number of vCPUs increases, the performance increases gradually. While CPU utilization increases, parallelism increases effectively. However, after CPU utilization becomes constant, the performance no longer increases and often decreases.

Fig. 9(c) is the case where CPU utilization decreases largely and then increases or becomes constant. While CPU utilization is decreasing, the performance is increasing. When CPU utilization does not increase any more, the performance no longer increases and often decreases. Fig. 9(d) is the case where CPU utilization largely increases and decreases repeatedly. Larger the number of vCPUs, lower



(a) Correlation 1 (b) Correlation 2

(c) Correlation 3 (d) Correlation 4

Figure 9: Four typical correlations between CPU utilization and execution time.

the performance is. This behavior is probably caused by application's CPU usage, but the root cause is unclear.

Using these four correlations, pCPU-Est determines the optimal number of vCPUs at runtime by examining only changes in CPU utilization. The algorithm is as follows. pCPU-Est begins at the number of vCPUs that is the same as that of pCPUs available to a VM and increases the number while CPU utilization increases. It considers the number of vCPUs as optimal when CPU utilization becomes constant. If CPU utilization largely drops at the beginning, pCPU-Est adopts that number of vCPUs when CPU utilization increases or becomes constant. If CPU utilization repeats increases and decreases largely, pCPU-Est considers the number of available pCPUs as the optimal number of vCPUs.

To change the number of vCPUs, we have developed a mechanism for externally changing the number from Dom0 in Xen. Dom0 is a privileged VM for managing VMs. The `vcpu-set` command in Xen's `xl` tool is provided for that purpose, but we could not change the number in Xen 4.4. First, pCPU-Est writes the new number of vCPUs of a VM to XenStore, which is the database for VM configuration. Then, the Linux kernel module installed in the VM detects that write. The kernel module reads the specified number from XenStore and performs hot-plug and hot-unplug of CPUs. Specifically, it changes the state of each CPU to online or offline to enable only the specified number of vCPUs. This kernel module uses standard support for Xen provided by Linux.

4.2. Thread Optimization

On the basis of the number of pCPUs available to a VM, pCPU-Est can change the number of application threads.

This results in indirectly reducing the number of vCPUs because vCPUs to which any threads are not assigned are not scheduled. In addition, this optimization method can overcome the limitation of vCPU optimization because it enables solving LHP and vCPU stacking not only at the kernel level but also at the application level. Note that the applicability of this method depends on applications. First, it is required that applications can run using the arbitrary number of threads. Second, applications need to have a mechanism for changing the number of threads at runtime. If they do not have such a mechanism, they would have to be restarted.

4.3. Estimation of Available pCPUs

pCPU-Est provides a new hypercall for returning the number of pCPUs available to the specified VM. A hypercall is a mechanism for invoking the hypervisor from VMs. This mechanism is similar to a system call used for the invocation to the operating system.

First, pCPU-Est divides pCPUs and vCPUs into several groups on the basis of the proportion of pCPUs available to each vCPU. Then, it estimates the number of pCPUs available to each VM. Such groups are created so that all the vCPUs in a group can use an equal proportion of pCPUs if the weights assigned to VMs are the same. As long as there is no restriction for the assignment of pCPUs, pCPUs and vCPUs create one group as a whole. However, they can be divided into several groups if vCPU affinity is set. Even if vCPU affinity is not set explicitly, some of the virtualized systems set NUMA affinity implicitly. NUMA affinity is the assignment of pCPUs considering NUMA.

pCPU-Est creates groups by dividing a graph of pCPUs and vCPUs as in Fig. 10. pCPUs and vCPUs are vortices and a pCPU and a vCPU are connected by an edge if the pCPU is assigned to the vCPU. When CPU affinity is set from several pCPUs to several vCPUs, each pCPU is connected to all of those vCPUs. Fig. 10(a) shows the graph when vCPU affinity is set from pCPUs 0 and 1 to vCPUs 0 and 1 and from pCPUs 1, 2, and 3 to vCPUs 2, 3, and 4.

The algorithm for graph division is as follows. First, pCPU-Est selects the pCPU that is connected to the minimum number of vCPUs. Then, it creates a group consisting of the selected pCPU and the vCPUs connected from the pCPU. This is because the proportion of pCPUs in the group is maximized as much as possible. The proportion is calculated by the number of pCPUs divided by that of vCPUs. In Fig. 10(b), pCPU 0 and vCPUs 0 and 1 form a group because the number of vCPUs connected from pCPU 0 is two and the smallest.

Next, among the rest of the pCPUs that is connected from the created group, pCPU-Est selects the pCPU that is connected to the minimum number of vCPUs. Then, it attempts to create a larger group including the selected pCPU and the vCPUs connected from that pCPU. If the proportion of the pCPUs in this new group is larger than that in the original group, pCPU-Est adopts the new group. It repeats this and creates as a large group as possible. If the

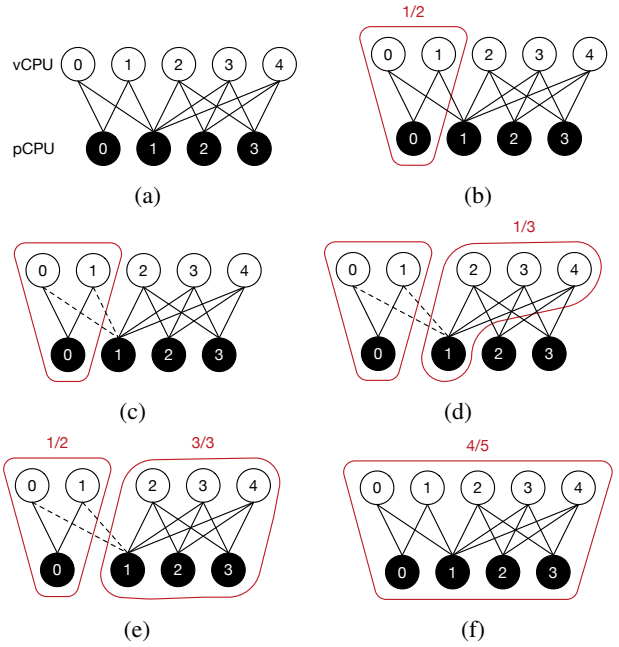


Figure 10: An example of the estimation of available pCPUs.

proportion of the pCPUs in the new group becomes smaller, pCPU-Est removes the edges connected to the outside of the group. For example, the proportion in the group created in Fig. 10(b) is $1/2$. If pCPU-Est adds pCPU 1 and vCPUs 2, 3, and 4 to the group, the proportion becomes $2/5$ and smaller. Therefore, pCPU-Est removes the edges between pCPU 1 and vCPUs 0 and 1 (dashed lines) and results in Fig. 10(c).

pCPU-Est repeats this for the rest of the graph. In Fig. 10(d), pCPU-Est selects pCPU 1 and creates a group consisting of pCPU 1 and vCPUs 2, 3, and 4. Similarly, among the pCPUs connected from this group, pCPU-Est attempts to add pCPU 2 to the group. Since the proportion of the pCPUs in the original group is $1/3$ but that in the new group becomes $2/3$ and larger, pCPU-Est adopts the new group. Finally, pCPU-Est adds pCPU 3 to the group and forms two groups, as in Fig. 10(e).

Once the entire graph is divided into several groups, pCPU-Est merges the created groups into the small number of groups as much as possible. This is because only local graph division does not result in correct graph division. First, pCPU-Est selects two groups that were connected by a removed edge. We call the group including the vCPU that was connected to the edge as group 1 and the group including the pCPU as group 2. If the proportion of the pCPUs in group 2 is larger or equal, pCPU-Est merges the two groups. It repeats this for all the groups that were connected by removed edges. For example, since two groups in Fig. 10(e) were connected by two removed edges, pCPU-Est attempts to merge them. The proportion of the pCPUs in group 1 is $1/2$, while that in group 2 is 1 and larger. Therefore, pCPU-Est merges them into one group.

For each created group, pCPU-Est equally assigns pC-

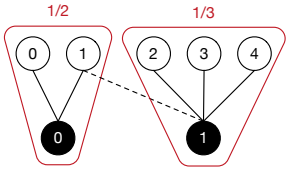


Figure 11: Another example of the estimation of available pCPUs.

PUs to vCPUs according to the weights set to VMs. In the case of Fig. 10, the proportion of pCPUs assigned to each vCPU is $4/5$ if the weights of VMs are the same. Next, pCPU-Est accumulates the proportions of pCPUs assigned to the vCPUs of each VM. It rounds up this value and considers the value as the number of pCPUs available to each VM. For example, if vCPUs 0 and 1 belong to one VM, the number of available pCPUs becomes two.

The example in Fig. 10 finally forms one group, but pCPUs and vCPUs form several groups in general. Fig. 11 is such an example. In this example, the proportion of pCPUs assigned to each vCPU is not $2/5$ equally because pCPUs 2, 3, and 4 can use only $1/3$ of pCPUs.

5. Experiments

We conducted several experiments to examine the effectiveness of dynamic vCPU optimization and thread optimization in pCPU-Est. We used the same experimental setup as in Section 3. We ran each benchmark ten times and calculated the average. We show the results under the configuration of CPU reduction.

5.1. Effectiveness of Dynamic vCPU Optimization

We applied dynamic vCPU optimization and measured the execution time of the ten benchmarks in NPB. The performance improvement from VCPU-Bal is defined as

$$I_{est/bal} = \frac{D_{bal}}{D_{est}} \tag{9}$$

where D_{est} is performance degradation in this optimization to the expected performance. D_{est} is defined as

$$D_{est} = \frac{T_{est}}{E} \tag{10}$$

where T_{est} is the execution time measured using this optimization. Fig. 12 shows the maximum values of $I_{est/bal}$ and compares them with those of $I_{opt/bal}$ obtained in Section 3.3. As a result, pCPU-Est accomplished almost the same maximum performance improvement as optimal in BT, CG, SP, and UA and that was up to 42%. In contrast, the performance was not improved at all in DC and LU although performance improvement is possible by the optimal vCPU assignment.

To further investigate performance improvement in pCPU-Est, we compared D_{est} of BT, CG, SP, and UA in pCPU-Est with D_{bal} and D_{opt} in detail. Fig. 13 shows

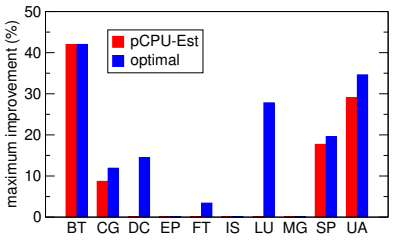
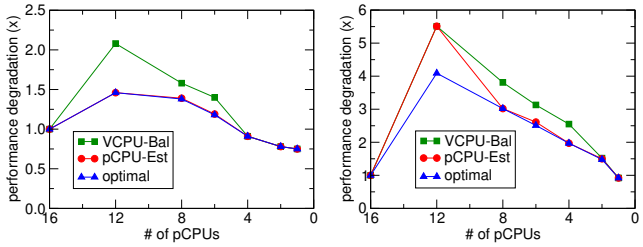


Figure 12: The maximum improvement from VCPU-Bal by dynamic vCPU optimization ($I_{est/bal}$ and $I_{opt/bal}$).



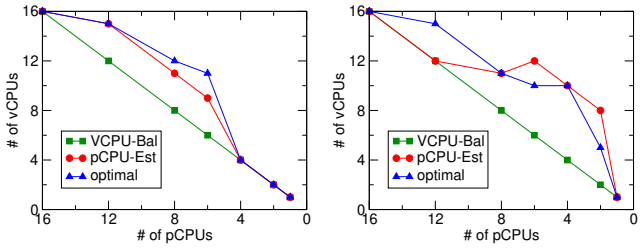
(a) BT (b) UA

Figure 13: The detailed improvement by dynamic vCPU optimization (D_{est}, D_{bal}, and D_{opt}).

the performance degradation of only two benchmarks due to space limitation. In BT, the performance was higher in pCPU-Est than in VCPU-Bal when the number of pCPUs were between 6 and 12. The performance was almost the same as that in the optimal vCPU assignment. Similarly, in SP, the performance was improved when the number of pCPUs were between 4 and 12. In UA, the performance was improved to near-optimal one, but it was the same as that in VCPU-Bal when the number of pCPUs was 12. In CG, the performance was also improved, but it is rather degraded when the number of pCPUs was 2.

To clarify the reasons of the above results, we compared the number of the vCPUs determined by pCPU-Est with that used in VCPU-Bal and the optimal one. Fig. 14 shows the comparison only in the two benchmarks. In BT and SP, the number of vCPUs in pCPU-Est was a bit smaller than the optimal one in several numbers of pCPUs, but that difference did almost not affect the performance. Similarly, in UA, the number of vCPUs in pCPU-Est was a bit larger in less numbers of pCPUs, but the performance was not affected. However, when the number of pCPUs was 12, the number of vCPUs in pCPU-Est was smaller than the optimal and the same as that used in VCPU-Bal. This is the reason why the performance was not improved in this number of pCPUs. In CG, the performance degraded when the number of pCPUs was 2 because the number of vCPUs in pCPU-Est was quite larger.

For LU and DC, on the other hand, we also examined the number of vCPUs determined by pCPU-Est. In LU, the optimal number of vCPUs was less than the number of pCPUs. In the current algorithm, pCPU-Est cannot optimize such applications. Worse, the difference between the number



(a) BT (b) UA

Figure 14: The number of vCPUs determined by pCPU-Est.

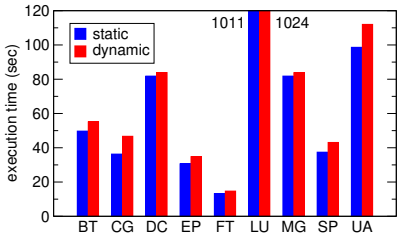


Figure 15: The maximum overhead of dynamic vCPU optimization.

of vCPUs in pCPU-Est and the optimal number was only one, but such slight difference affected the performance very largely. In DC, the number of vCPUs in pCPU-Est was far from the optimal when the number of pCPUs was less than four. This is the reason why pCPU-Est could not optimize this application.

Next, we examined the overhead of dynamically determining the number of vCPUs. While pCPU-Est searches for the optimal number of vCPUs, the benchmarks run in non-optimal numbers of vCPUs. Therefore, the application performance can be degraded in total. We measured the execution time when we used the number of vCPUs statically determined in advance and when we dynamically determined it at run time. Fig. 15 shows the execution time when the overhead is maximum for each benchmarks. We omit the data for IS because the execution time was too short to determine the optimal number of vCPUs. Although the maximum overhead reached 28% in CG, it would become negligible in longer-running real applications. Shorter the execution time is, larger the overhead tends to be. Also, the overhead depends on the sensitivity of application performance to the number of vCPUs.

5.2. Effectiveness of Thread Optimization

We applied thread optimization and measured the execution time of the ten benchmarks. We statically fixed the number of threads because the optimal number of threads was almost always the same as the number of available pCPUs according to our experiments. The performance improvement from VCPU-Bal is defined as

$$I_{est2/bal} = \frac{D_{bal}}{D_{est2}} \tag{11}$$

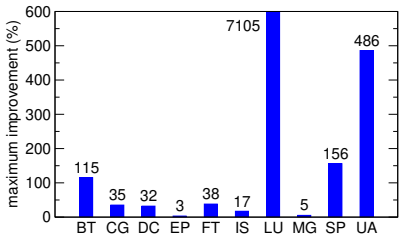
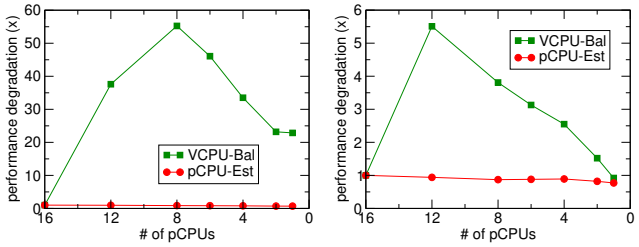


Figure 16: The maximum improvement from VCPU-Bal by thread optimization ($I_{est2/bal}$).



(a) LU (b) UA

Figure 17: The detailed improvement by thread optimization (D_{est2} and D_{bal}).

where D_{est2} is performance degradation to the expected performance. D_{est} is defined as

$$D_{est2} = \frac{T_{est2}}{E} \tag{12}$$

where T_{est2} is the execution time measured using this optimization. Fig. 16 shows the maximum values of $I_{ext2/bal}$. The performance was improved in all the benchmarks. In LU, in particular, the performance improvement was significant and up to 72x although dynamic vCPU optimization was not effective at all, as in Fig. 12. This is because the LHP problem inside the application was solved. In BT, SP, and UA, the performance became more than twice higher as well.

For LU and UA, Fig. 17 compares D_{est2} and D_{bal} in detail. In any numbers of pCPUs, pCPU-Est could achieve much higher performance than VCPU-Bal. Even compared with the result of the optimal vCPU assignment in the previous section, thread optimization was much more effective. However, it should be noted again that thread optimization is not always applicable due to its application dependence.

6. Related Work

VCPU-Bal [1] prevents LHP and vCPU stacking using vCPU ballooning, which increases and decreases the number of vCPUs of a VM. The paper points out that these issues rise due to double scheduling of vCPUs and application threads. Therefore, VCPU-Bal statically determines the number of vCPUs so that vCPUs are assigned to pCPUs one-to-one. In contrast, pCPU-Est dynamically determines the more appropriate number of vCPUs. FlexCore [2] is the implementation of VCPU-Bal and achieves the detection of

vCPU conflict, efficient communication between VMs and the hypervisor, and hot-plug of vCPUs in KVM [11]. We have implemented pCPU-Est in Xen.

vScale [3] adjusts the number of vCPUs in a finer-grained manner than VCPU-Bal. Like VCPU-Bal, vScale basically determines the number of vCPUs according to the weights of VMs. In addition, when some of the VMs do not use up assigned pCPUs, vScale decreases the number of vCPUs of these VMs and increases that of the other VMs. To enable frequently changing the number of vCPUs of VMs, vScale provides a mechanism for reconstructing CPUs in the operating system. VCPU-Bal and vScale assume only that multiple VMs share pCPUs, but pCPU-Est assumes that CPU utilization of VMs is limited and the assignment of pCPUs is reduced as well.

Co-scheduling [12] can solve LHP by simultaneously scheduling all the vCPUs of a VM. vCPUs of lock holders can always run whenever those of lock waiters are scheduled. However, it cannot schedule a VM until the necessary number of pCPUs are prepared. As a result, vCPUs with a higher priority can be scheduled after vCPUs with a lower priority. Balance scheduling [5] 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. These scheduling algorithms can be less efficient due to their scheduling constraints.

Intel Pause Loop Exiting [13] and AMD Pause Filter [14] are mechanisms for causing a VM exit when several PAUSE instructions are executed during a spin lock. If the hypervisor schedules another vCPU when it detects spin waiting, the performance degradation due to LHP and vCPU stacking is mitigated. However, frequent VM exits can affect the performance of VMs. In addition, this mechanism is ineffective for spin waiting that does not execute any PAUSE instruction, as used by LU in NPB.

For thread optimization in non-virtualized systems, there are a lot of work to estimate the optimal number of threads. Thread Reinforcer [15] runs an application with various numbers of threads for a short period and statically finds the best number. Aurora [16] redirects function calls to the OpenMP library and dynamically adjusts the number of threads by using the OpenMP function. These mechanisms can be combined with thread optimization in pCPU-Est.

7. Conclusion

This paper first investigated how the performance of parallel applications degraded under three VM configurations on a pCPU shortage. Then it revealed that the previous work could not always achieve optimal performance. Therefore, this paper proposes pCPU-Est for improving application performance under CPU overcommitment. Dynamic vCPU optimization determines the optimal number of vCPUs of VMs at runtime, while thread optimization optimizes the number of application threads. According to our experiments, dynamic vCPU optimization improved application performance by up to 42% and thread optimization did by up to 72x under the configuration of CPU reduction.

One of our future work is to confirm the effectiveness of pCPU-Est under the other two VM configurations. For ease of analysis, we ran only one VM in this paper, except for the configuration of CPU sharing. The impact of other VMs in the same host should be considered as the next step. In addition, it is necessary to apply pCPU-Est to other parallel applications, in particular, cloud-oriented ones such as MapReduce. Another direction is to develop a method for always determining the optimal number of vCPUs even in the cases where the current algorithm cannot optimize that. Using such methods, it is possible to further improve application performance and prevent performance degradation by dynamic vCPU optimization.

Acknowledgment

This work was supported in part by JSPS KAKENHI Grant Number 19H04087.

References

[1] X. Song, J. Shi, H. Chen, and B. Zang. Schedule processes, not VCPUs. In *Proc. Asia-Pacific Workshop on Systems*, 2013.

[2] M. Tianxiang and H. Chen. FlexCore: Dynamic Virtual Machine Scheduling Using VCPU Ballooning. *Tsinghua Science and Technology*, 20(1):7–16, 2015.

[3] L. Cheng, J. Rao, and F. Lau. vScale: Automatic and Efficient Processor Scaling for SMP Virtual Machines. In *Proc. European Conf. Computer Systems*, 2016.

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

[5] O. Sukwong and H. Kim. Is Co-scheduling Too Expensive for SMP VMs? In *Proc. European Conf. Computer Systems*, pages 257–272, 2011.

[6] 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*, pages 164–177, 2003.

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

[8] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, University of California at Berkeley, 2009.

[9] K. Kourai and R. Nakata. Analysis of the Impact of CPU Virtualization on Parallel Applications in Xen. In *Proc. Int. Symp. Parallel and Distributed Processing with Applications*, pages 132–139, 2015.

[10] OpenMP Architecture Review Board. OpenMP Application Programming Interface Version 5.0. https://www.openmp.org/, 2018.

[11] Red Hat, Inc. Kernel Based Virtual Machine. http://www.linux-kvm.org/.

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

[13] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*, 2016.

[14] AMD, Inc. *AMD64 Architecture Programmers Manual Volume 2: System Programming*, 2017.

[15] K. Pusukuri, R. Gupta, and L. Bhuyan. Thread Reinforcer: Dynamically Determining Number of Threads via O Level Monitoring. In *Proc. Int. Symp. Workload Characterization*, pages 116–125, 2011.

[16] A. Lorenzon, C. de Oliveira, J. Souza, and A. Beck. Aurora: Seamless Optimization of OpenMP Applications. *IEEE Trans. Parallel and Distributed Systems*, 30(5):1007–1021, 2019.