PAPER

Cache-Aware Virtual Machine Scheduling on Multi-Core Architecture

Cheol-Ho HONG[†a)], **Young-Pil KIM**[†b)], ***Nonmembers***, **Seehwan YOO**[†c)], ***Student Member***, **Chi-Young LEE**[†d)], ***Nonmember***, ***and*** **Chuck YOO**[†e)], ***Member***

SUMMARY Facing practical limits to increasing processor frequencies, manufacturers have resorted to multi-core designs in their commercial products. In multi-core implementations, cores in a physical package share the last-level caches to improve inter-core communication. To efficiently exploit this facility, operating systems must employ cache-aware schedulers. Unfortunately, virtualization software, which is a foundation technology of cloud computing, is not yet cache-aware or does not fully exploit the locality of the last-level caches. In this paper, we propose a cache-aware virtual machine scheduler for multi-core architectures. The proposed scheduler exploits the locality of the last-level caches to improve the performance of concurrent applications running on virtual machines. For this purpose, we provide a space-partitioning algorithm that migrates and clusters communicating virtual CPUs (VCPUs) in the same cache domain. Second, we provide a time-partitioning algorithm that co-schedules or schedules in sequence clustered VCPUs. Finally, we present a theoretical analysis that proves our scheduling algorithm is more efficient in supporting concurrent applications than the default credit scheduler in Xen. We implemented our virtual machine scheduler in the recent Xen hypervisor with para-virtualized Linux-based operating systems. We show that our approach can improve performance of concurrent virtual machines by up to 19% compared to the credit scheduler.

key words: *virtualization, cache-aware scheduling strategy, multi-core processor*

1. Introduction

For several years, new paradigms related to distributed computing systems such as grid, utility, and cloud computing have been introduced. Recently, among these paradigms, cloud computing has drawn the most attention. Cloud computing is intimately related to system virtualization technology, which allows multiple operating systems (OSs) to be consolidated simultaneously in a single physical machine. In many cloud computing environments, virtualization is the key enabling technology. For example, Amazons Elastic Compute Cloud (EC2) [1] adopts the Xen hypervisor [2] to offer Linux, Sun Microsystems' OpenSolaris, Windows Server 2008, and FreeBSD. By adopting a virtualization layer, EC2 can share system resources between applications from different users while the applications are consolidated in each virtual machine (VM). In addition to the consolidation service, in the cloud environment, virtualization technologies can provide additional and valuable benefits, including improved system utilization and greater fault-tolerance via reliable migration methods.

Current computing platforms used for distributed computing systems have one or more multi-core processors, and the number of cores is expected to see geometric increase in the near future. As multi-core processors are more prevalent, the development of technologies to fully exploit their capabilities is becoming an important research area. As recent multi-core processors share some hardware resources such as prefetching hardware, the front-side bus, and last-level caches on the same chip, methods for smart and efficient allocation of the shared resources need to be developed for improved application performance [3]–[6].

Among the shared resources of the processor, exploiting the locality of the last-level caches (L2 or L3 caches) makes inter-core communication efficient. Figure 1 illustrates two Intel Core2 Quad processor packages with each having four cores with two L2 caches. Each of the L2 caches is shared and accessed by two cores. In these multi-core topologies, if communicating threads are clustered among cores that share the same last-level cache, performance will increase as the threads can then reuse the cached data. By avoiding a cache coherence protocol and main memory references, hundreds of processor cycles on average can be saved in the communication process [3]. To exploit the multi-core topologies, cache-aware schedulers have been suggested at the OS level [3]–[5]. They detect data sharing patterns between threads and cluster them onto cores that share the same cache domain.

Unfortunately, virtualization software is not cache-aware or lacks the ability to fully exploit the locality of the last-level caches by clustering communicating virtual CPUs (VCPUs) into the same cache domain. For example, VM schedulers in Xen [2], including BVT, SEDF, and the credit scheduler, are not cache-aware. They implement a simple load-balancing algorithm that migrates VCPUs regardless of the multi-core topology. They migrate VCPUs on a physical core with high workloads to cores with lower workloads. In contrast to Xen, the recent ESX scheduler in VMware is cache-aware and has a smart load-balancing algorithm that prefers intra last-level cache migration to inter last-level cache migration [7]. The scheduler also supports clustering of VCPUs via the vSMP Consolidate facility that can be enabled by the administrator. However, the clustering mech-

Manuscript received April 25, 2012.

[†]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: ypkim@os.korea.ac.kr

c) E-mail: shyoo@os.korea.ac.kr

d) E-mail: cylee@os.korea.ac.kr

e) E-mail: chuckyoo@os.korea.ac.kr

DOI: 10.1587/transinf.E95.D.2377

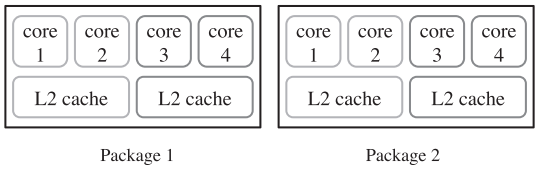


Fig. 1 A schematic view of two Intel Core2 Quad processors.

anism may not be honored depending on the availability of physical CPUs. As this hypervisor is a commercial product, we are not aware of its technical specifications. Therefore, we do not know how frequently clustering requests may be dismissed and the performance implications of this behavior.

The goal of this study is to design, implement, and evaluate a cache-aware VM scheduler on a multi-core architecture. For the implementation of the scheduling algorithm, we use the Xen hypervisor and its credit scheduler. We chose Xen because it is widely used and is an open source hypervisor for data centers and cloud computing. Concretely, when a VM has VCPUs communicating with each other, we proactively locate them in the same cache domain. This is to prevent them from suffering a lot of high-latency cross-chip or cross-cache domain communication. After clustering the communicating VCPUs, in order to increase utilization of the last-level cache, we schedule clustered VCPUs at the same time or in sequence. We have found that just clustering the communicating or related VCPUs into the same cache domain is insufficient to improve performance. The credit scheduler schedules VCPUs asynchronously among multiple VCPUs while guaranteeing fairness in sharing CPU bandwidth. Due to the asynchronous scheduling, the time quantum of any unrelated VCPU, generally 30 ms, is long enough for the cached content written by one of the communicating VCPUs to be displaced. Therefore, we co-schedule or schedule in sequence the relative VCPUs and this mechanism enables better performance in terms of last-level cache miss rates.

In scheduling clustered VCPUs, we assume that the strict co-scheduling algorithm, which is adopted in [7], [8] and keeps execution of VCPUs of an identical VM during some time slots, may disregard the I/O boost mechanism of the credit scheduler. The boost concept is introduced to improve performance of I/O intensive VMs in terms of both bandwidth and latency, and is implemented by prioritization of I/O VCPUs [9]. Then, the co-starting and the co-ending mechanism of the strict co-scheduling algorithm can preempt VCPUs performing I/O requests or responses. In reverse, I/O boosted VCPUs may not grab the physical CPUs while the co-scheduled VM is running.

To address this problem, we provide two scheduling options, an aggressive and a smooth method. While the aggressive method implements a strict co-scheduling mechanism, the smooth one applies a loose mechanism by preferentially respecting the I/O boosted VCPUs. Therefore, the concurrent VCPUs are scheduled next to the I/O boosted VCPUs. Administrators can choose these options on the basis of whether they place emphasis on the performance of the concurrent domain or the I/O domain while keeping high utilization of the last-level cache.

The main contributions of this paper are as follows. First, we provide a space-partitioning algorithm that migrates and clusters related VCPUs on the same cache domain. Second, we provide a time-partitioning algorithm that co-schedules or schedules in sequence clustered VCPUs. In this scheduling algorithm, we provide both the aggressive and smooth methods mentioned above. Finally, we present a theoretical analysis that proves our scheduling algorithm is more efficient in supporting concurrent applications than the default credit scheduler in Xen. We evaluate our scheduler using concurrent applications such as VolanoMark [10].

The remainder of this paper is structured as follows: In Sect. 2, we explain the background of Xen and the credit scheduler. In Sect. 3, we illustrate how our cache-aware VM scheduler works. Section 4 presents the scheduling model, and Sect. 5 describes the details of the implementation. Section 6 provides the evaluation results, and Sect. 7 explains related work. Section 8 presents discussion on several cache-aware schedulers. Finally, we present our conclusions in Sect. 9.

2. Background

2.1 Xen

Xen [2] is an open-source hypervisor that adopts a para-virtualization technique, rather than full virtualization, to minimize virtualization overhead. The para-virtualization technique enables collaboration between the hypervisor and a guest OS by providing communication channels named hypercalls. These calls are analogous to system calls between an OS and a user application. The hypercalls request the hypervisor to execute the instructions on behalf of the guest OSs. Hypercalls include processor state update operations, for example, memory management unit (MMU) updates and physical interrupt masking operations, that a guest OS cannot execute directly. Source code of a guest OS should be modified in the para-virtualized approach to contain hypercalls.

To consolidate multiple OSs, Xen provides CPU, memory, and I/O device virtualization. In CPU virtualization, it replaces sensitive instructions, which modify system states and are executed in the supervisor mode of guest OSs, with hypercalls. It also places guest OSs at a lower privilege level than the hypervisor so that they are protected from OS misbehavior. Guest OSs also register descriptor tables for exception and trap handlers that should be executed in the supervisor mode. In memory virtualization, it provides memory protection between a hypervisor, guest OSs, and applications. Although it allows guest OSs to have direct read access to hardware page tables, it validates all the page table update requests of the guest OSs. This mechanism prevents one guest OS from reading from or writing to the system

memory of the other guest OSs. In device virtualization, domain0, which is an administrator VM, runs most of the native device drivers. DomainU, which is an unprivileged domain, implements the front end device drivers for block and network interfaces to indirectly access the hardware that only domain0 has permission to use.

2.2 Credit Scheduler

In recent versions of Xen, the credit scheduler, which is a proportional fair share CPU scheduler, is used by default. Each domain has two properties associated with the scheduler, a weight and a cap. The weight decides allowance of the physical CPU time that the domain will obtain. The cap represents the maximum amount of CPU bandwidth that a domain will be able to consume [11].

The credit scheduler schedules domains fairly based on the credit amount, which is determined by the weight that each domain receives. Credit refers to CPU time or CPU bandwidth for which each domain can run. A VCPU of a domain consumes its credit every 10 ms; a global account thread recharges each VCPU's credit amount according to its weight every 30 ms.

The credit scheduler maintains per-CPU run queues, and at every scheduling moment, the next VCPU to run is chosen from the head of each run queue. The local run queue of each core is sorted by VCPU priority. Three VCPU priorities are defined in the current Xen implementation: UNDER (value of −1), OVER (−2), and BOOST (0). The VCPU's priority is determined by the remaining credit amount of the VCPU while the global account thread is running. If the credit amount of the VCPU is positive, the VCPU's priority is UNDER. Conversely, if the credit amount is negative, the VCPU's priority becomes OVER. BOOST priority is introduced to improve I/O performance of domains in terms of both bandwidth and latency. Whenever an I/O event is sent to a sleeping VCPU, the VCPU wakes up and changes its priority to BOOST and preempts a running VCPU immediately.

In a multi-core system, the credit scheduler supports VCPU load-balancing over all physical cores to guarantee fair share of total CPU bandwidth in the system. When a core cannot find a VCPU of UNDER priority on its local run queue, the core will search for any VCPU of UNDER priority on other cores. In addition, if any core has no VCPUs on its local run queue, it will search for any runnable VCPUs on other cores before it goes idle.

3. Cache-aware Scheduler

In this section, we cover the details of the cache-aware VM scheduler design. Our scheduler is based on the Xen hypervisor and modified from the credit scheduler. Figure 2 illustrates the structure of the Xen hypervisor with the cache-aware scheduler. Our scheduler depends on administrator's annotations and the detection module of concurrent applications. The former identifies which VM has concurrent applications with large shared regions, and the latter dynamically recognizes whether concurrent applications are running or not in the VM designated by the administrator. The detection module is implemented outside the scheduler. When the workloads in the VM are concurrent applications, our scheduler clusters the related VCPUs on the same cache domain. Then, the scheduling of the VCPUs occurs almost simultaneously by these two algorithms: the space-partitioning and time-partitioning algorithm. When the workloads of VCPUs of any domain are not concurrent, our scheduler schedules the VCPUs asynchronously according to the default proportional share algorithm of the credit scheduler.

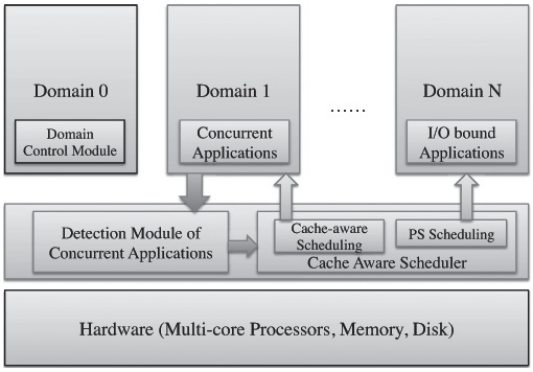


Fig. 2 The structure of Xen hypervisor with the cache-aware scheduler.

3.1 Identifying Communicating VCPUs

In our work, the more global memory regions communicating VCPUs share, the more they will benefit from the last-level cache utilization by the cache-aware VM scheduling. For this purpose, we require system administrators annotations to identify which VM has concurrent applications with large shared regions. The information can be obtained from prior research [12], [13] or from developers. In fact, it is very difficult or inefficient to detect the amount of shared regions automatically without the support of special hardware performance monitoring units (PMUs). The PMUs select data addresses on cache misses of the remote cache [3], [4]. The approach using PMUs uses a special data address register (DAR) for sampling data addresses and calculates similarity metric between threads. However, the hardware support is not available in many micro architectures including our Intel quad-core evaluation system; for reference, the hardware support is available in the IBM POWER5, Intel Itanium, Sun UltraSparc, and AMD Barcelona and Shanghai processors.

To support smart cache-aware scheduling of the designated VMs enabled by the administrator, we provide the detection module of concurrent applications. The module is implemented outside the scheduler and feeds information, into the cache-aware scheduler, about whether concurrent applications are running or not in the designated VM. In addition, it informs the scheduler of which VCPUs the applications are running on. The mechanism the module

uses is adopted from [5]. Generally, concurrent applications with shared regions interact with other threads by invoking mutex-related system calls when accessing the shared region. Using this characteristic, the detection module counts mutex-related system calls per VCPU of the designated VM in the hypervisor. When the preset threshold is reached on at least two VCPUs, the module makes the clustering information that contains interacting VCPUs in the designated VM (in our experiment, the preset value is 5 times per 10 ms). The clustering information is implemented as a list, and the hypervisor regards the head element of the list as a cluster representative.

The above-mentioned mechanism is conducted by the system call interception in the hypervisor. Therefore, in order to capture mutex-related system calls, we do not need to install any instrumentation on guest OSs. However, intercepting system calls in the hypervisor level has a high cost as illustrated in the evaluation section. Therefore, using timer interfaces in Xen, we periodically enable the detection module for a period of time disabling the module for the rest of the time; in our experiment, we enable the module for 100 ms per second. In current Xen-x86 implementation, because system calls bypass the hypervisor [14], we modify the hypervisor slightly and make system calls go through the hypervisor. The detail of x86 specific modification will be provided in the implementation section.

The advantage of both taking administrator's annotations and the detection module is simplicity. This approach can provide uncomplicated and practical mechanisms for the architecture that does not support data address sampling features. However, the disadvantage is that our technique used in the detection module does not cover several kinds of synchronization mechanisms. If user-level spin-lock libraries are used, interaction between VCPUs might not be detected. Nevertheless, we have tried to cover conventional thread libraries including NPTL(Native POSIX Thread Library) in Linux and the standard Java threads library that uses mutex-related system calls.

3.2 Space-Partitioning

The space-partitioning algorithm makes spatial scheduling partitions based on each cache domain and assigns communicating VCPUs in the same partition. It uses the clustering information created by the detection module in order to identify communicating VCPUs. With this algorithm, clustered VCPUs can reduce cost consumed by cache coherence protocols. Generally, in multi-core processors, if any data that are also located on another cache domain are modified, the duplicate copy of the data should be invalidated. Access from other cache domains to these invalidated data leads to cache coherence misses. These processes cause high latency inter-cache domain communications. Clustering the communicating VCPUs on the same cache domain can address this problem in the virtualization environment.

The algorithm periodically checks whether the clustering information lists are created or updated by the detection module. After identifying the created or updated cluster lists, the algorithm distributes the created clusters evenly among all of the cache domains. The reason for this is to utilize all cores across the system and guarantee the fairness policy of the credit scheduler. Concretely, the algorithm searches cache domains that have some idle cores first. When any cache domain with idle cores is found, the scheduling algorithm assigns a created cluster to the found cache domain. When no cache domain is found in the first phase, we assign the created cluster to the cache domain with lightweight workloads, which can be identified with the total length of the run queues of any cache domain.

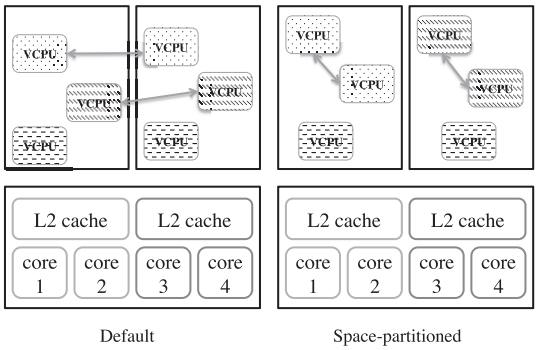


Fig. 3 Space-partitioning.

The space-partitioning algorithm is internally accomplished by using affinity interfaces provided by Xen. Affinity-related functions in Xen receive, as parameters, an address of a VCPU and a CPU mask structure that can designate the runnable cores in the target cache domain. Using the CPU mask variable, we migrate the communicating VCPUs in the cluster list to the same cache domain. In Fig. 3, the left side is the default system state where communicating VCPUs are located in different cache domains. Then, two cluster lists are formed to contain each two communicating VCPUs by the detection module. The space-partitioning algorithm assigns each cluster into the different cache domains by mapping the cache domain id, which is determined by the order in the multi-core topologies, to the descriptor of each cluster. Then, the algorithm migrates VCPUs in each cluster list to the assigned location using affinity interfaces as the right side of Fig. 3.

The advantage of the space-partitioning algorithm is that it can reduce the inter-cache domain communication cost by reducing cache coherence misses. The more global memory regions communicating VCPUs share, the more the algorithm can improve performance of concurrent applications. In addition, as the mutex lock variables, which are employed in threads in order to synchronize themselves, can be located in the same cache domain, the algorithm prevents the threads from wasting cycles consumed by a cache snooping mechanism.

3.3 Time-Partitioning

After related VCPUs are clustered into the same cache

domain by the space-partitioning algorithm, the time-partitioning algorithm makes each temporal scheduling partition inside each space partition and assigns communicating VCPUs in the same temporal partition. By doing this, the algorithm can schedule clustered VCPUs almost simultaneously and maximize utilization of the last-level cache in terms of data reuses among the VCPUs. As the default credit scheduler schedules VCPUs asynchronously among multiple VCPUs, without this algorithm, other non-concurrent VCPUs in the same cache domain can sweep out cached contents that may be used by communicating VCPUs; this situation prevents data reuses among communicating VCPUs and makes unavoidable cache misses (conflict misses). Therefore, we schedule communicating VCPUs as close as possible to each other in order to maximize the data reuses and reduce last-level cache misses incurred by redundant data evictions and reloads.

To gain the best performance of concurrent applications, it will be good to schedule communicating VCPUs at the same time such as the strict co-scheduling algorithm, which is adopted in [7], [8]. However, such a strict co-scheduling algorithm, which keeps execution of VCPUs of an identical VM during some time slots, may ignore the I/O boost mechanism of the credit scheduler. The boost mechanism is introduced in the credit scheduler to improve I/O intensive VMs in terms of both bandwidth and latency [9]. These improvements now make it an important characteristic of the scheduler. The boost mechanism is activated when a blocked VCPU is sent a virtual interrupt. As the boosted VCPU has higher priority than normal VCPUs, it is scheduled immediately. The problem is that co-starting and co-ending mechanisms of strict the co-scheduling algorithm can preempt the boosted VCPU. Conversely, the boosted VCPU may not grab the physical CPU until the concurrent VM releases the CPU. This situation may result in a deterioration of the performance of other I/O bound domains.

Therefore, considering side effects of the strict co-scheduling mechanism, we provide two options that administrators can choose, aggressive and smooth methods. While the aggressive method implements a strict co-scheduling mechanisms, the smooth one always respects the I/O boosted VCPUs. Detailed explanations are as follows.

3.3.1 Aggressive Method

To schedule related VCPUs at the same time, we introduce a new and highest ranked priority in the system: AGGRESSIVE (value of 2). When the head of a cluster list is selected to run on any physical core, time-partitioning algorithm changes priorities of other communicating VCPUs in the same cluster list to AGGRESSIVE. Then, it puts the VCPUs on the head of the run queue and tickles physical cores to reschedule VCPUs. Because the AGGRESSIVE priority is higher than BOOST priority, concurrent VCPUs preempt I/O intensive VCPUs in this mechanism. The AGGRESSIVE priority lasts for at most 20 ms to prevent other VCPUs from waiting a long time for CPU allocation. The left side of Fig. 4 illustrates the default system state, and the middle side illustrates the system state where the aggressive method is applied.

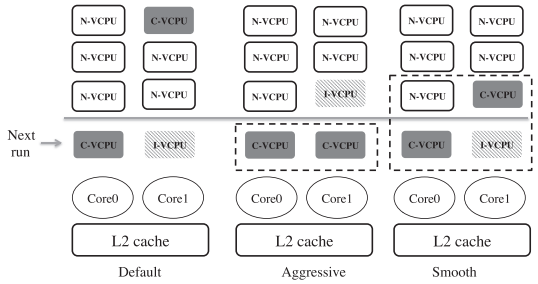


Fig. 4 Time-partitioning. The C-VCPUs indicate communicating VCPUs; the I-VCPUs indicate I/O bound VCPUs. The N-VCPUs mean normal VCPUs

3.3.2 Smooth Method

While respecting the I/O boosted VCPUs, the smooth method schedules in sequence communicating VCPUs in the same temporal partition. It operates in a similar way to the aggressive method. Instead of using AGGRESSIVE priority, the method defines the SMOOTH priority (value of 0) less than BOOST priority. Because the value of UNDER is -1, and BOOST is 0, we modify the value of BOOST to 1 and put our priority for the smooth method in the middle of them. Similar to the aggressive method, when the head of a cluster list is selected to run, the time-partitioning algorithm changes priorities of other communicating VCPUs in the same cluster list. In contrast to the aggressive method, concurrent VCPUs cannot preempt I/O intensive VCPUs with BOOST priority and are put second to the last I/O intensive VCPU with BOOST priority in the run queue. The right side of Fig. 4 illustrates the system state where the smooth method is applied.

3.3.3 Effects on Performance and Fairness

In terms of concurrent applications, the time-partitioning algorithm, which is composed of the aggressive and smooth methods, can improve performance by increasing data reuses and reducing conflict misses of a last level cache. In the smooth method, if there are a small number of boosted VCPUs, the time-partitioning algorithm can still keep high utilization of the last-level cache among communicating VCPUs. However, if there are a lot of not only I/O intensive but also memory-bound VCPUs that wait to be selected to run in the entire system, the smooth method could not have many opportunities to reuse cached data.

When the aggressive method is used, this algorithm can also solve the synchronization problem in the concurrent applications as the hybrid scheduling framework [8]. The synchronization problem in [8] occurs when threads in the same VM synchronize with each other through barrier synchronization, and the hypervisor adopts the asynchronous VM scheduling. Then, some threads may be blocked for several

time slots owing to the asynchronous CPU scheduling of the hypervisor. This situation deteriorates the performance of concurrent applications. By scheduling interacting VCPUs at the same time, the synchronization problem in [8] can be solved by using our aggressive method.

In regard to the fairness of the scheduler, we conditionally set communicating VCPUs to high priorities when the cluster representative is selected to run. We do not co-schedule any communicating VCPU with OVER priority (the credit amount is negative) to guarantee the fairness policy of the credit scheduler. As the credit scheduler puts a de-scheduled VCPU into the tail of a run queue, unconditional and frequent preemption by communicating VCPUs could compromise system-wide fairness.

4. Scheduling Model

In this section, we present a theoretical analysis of each scheduling policy, For convenience, we borrow some notations from [8], while building our scheduling model.

4.1 Scheduling Model

4.1.1 Concurrent Task Model

$T = \{T_1, T_2, \ldots, T_{|T|}\}$ denotes a concurrent task that consists of $|T|$ threads. Thread $T_i = \{T_i{}^1, T_i{}^2, \ldots, T_i{}^{|T_i|}\}$ consists of several thread units, where $|T_i|$ is the number of thread units. We define thread unit $T_i{}^k$ such that $T_i{}^k$ contains one computation part and one subsequent interaction part.

The interaction between threads is realized through synchronization and communication operations. Among the various thread synchronization methods, we focus on methods that allow for asynchronous interaction in order to simplify our scheduling model. In asynchronous interaction, when a thread reaches the interaction code, it can execute the interaction code without having to wait for other threads [15]. Similarly, among the various communication methods, we consider the method that achieves communication by accessing a shared memory.

In order to highlight the cache effect in the scheduling model, we define an execution time of $T_i{}^k$, $E(T_i{}^k)$, as follows:

$$E(T_i{}^k) = \rho_{ik} \times C_{ik} + (1 - \rho_{ik}) \times M_{ik}$$

, where $0 \leq \rho_{ik} \leq 1$.

In $E(T_i{}^k)$, ρ_{ik} is the cache utilization of $T_i{}^k$, that is, the ratio of cache hits to total memory references. C_{ik} denotes the average processing time of $T_i{}^k$ under the condition that all memory references induce cache hits. Further, M_{ik} denotes the average processing time of $T_i{}^k$ under the condition that all memory references induce last-level cache misses.

Cache utilization ρ_{ik} can vary with the scheduling policy. For example, when the space-partitioning or time-partitioning algorithm is applied, the value of ρ_{ik} increases compared to the default scheduler because of reduced cache coherence misses or conflict misses. Therefore, $E(T_i{}^k)$ can vary with the scheduling policy, even if the instructions of $T_i{}^k$ are identical.

4.1.2 Scheduling Model

$P = \{P_1, P_2, \ldots, P_{|P|}\}$ indicates physical CPUs, where $|P|$ is the number of physical CPUs in the system. $V = \{V_1, V_2, \ldots, V_{|V|}\}$ represents VMs running on the physical CPUs, where $|V|$ is the number of VMs in the system. The weight of VM V_i is represented by $\omega(V_i)$, and this is a relative proportion of physical CPU consumption. Thus, we have $\sum_{i=1}^{|V|} \omega(V_i) = 1$. $C(V_i) = \{v_{i1}, v_{i2}, \ldots, v_{i|C(V_i)|}\}$ indicates VCPUs running on VM V_i, and the number of VCPUs is $|C(V_i)|$.

To avoid the cost of thread migration, we assume that each thread is fixed to a VCPU. Moreover, to avoid the cost of frequent context-switching of threads on a VCPU, we assume that $|T| \leq |C(V_i)|$. Similarly, to avoid the cost of context-switching of VCPUs that belong to the same VM on a single CPU, we assume that $|C(V_i)| \leq |P|$.

4.1.3 Scheduling Policy Analysis

In the following theorems, we let i^* as the value of i when i^* satisfies the following equation.

$$\sum_{k=1}^{|T_{i^*}|} E(T_{i^*}{}^k) = \max_i \left\{ \sum_{k=1}^{|T_i|} E(T_i{}^k) \right\}$$

, where $1 \leq i \leq |T|$.

In our work, time is divided into fixed intervals referred to as time slots. We assume that a time slot is a time unit of virtual machine scheduling. Therefore, the virtual machine scheduler is called by the hypervisor at every time slot. In other words, the size of a time quantum of any VCPU equals that of a time slot. Then, $E(T_i{}^k)$ is normalized in proportion to the actual length of a time slot. At any moment, time t is also normalized in proportion to the actual length of a time slot.

Proportional-share scheduling. The proportional-share scheduling policy allocates a certain amount of CPU time to each VM in proportion to its weight.

Theorem 1. *In proportional-share scheduling, the worst completion time for a concurrent task T in VM V_i is as follows:*

$$\lfloor \tfrac{Lag + \lceil E_T \rceil - 1}{R_I(v_{ij})} \rfloor + E_T - (\lceil E_T \rceil - 1)$$

, where Lag is the upper bound of deviations between the ideally and the actually obtained CPU time, $R_I(v_{ij}) = \frac{|P| \times \omega(V_i)}{|C(V_i)|}$, $E_T = \sum_{k=1}^{|T_{i^*}|} E(T_{i^*}{}^k)$, and $\lceil E_T \rceil \geq 2$.

Proof. In an ideal fair scheduling model, the ideal amount of CPU time for VCPU v_{ij} is determined by the number of processors, the weight proportion of the VM where the

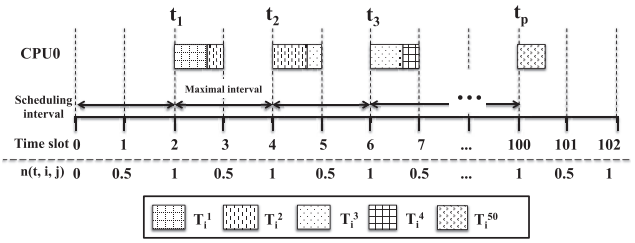


Fig. 5 An example of the proportional-share scheduling strategy for thread T_i in V_i, when $|P| = 2$, $|V| = 2$, $|C(V_i)| = 2$, $\omega(V_i) = 0.5$, and $Lag = 1$. The last time quantum is given at t_p.

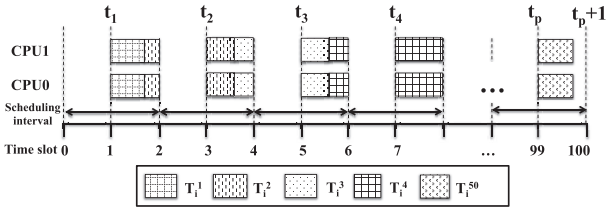


Fig. 6 An example of the co-scheduling strategy for two threads in V_i, when $|P| = 2$, $|V| = 2$, $|C(V_i)| = 2$, and $\omega(V_i) = 0.5$. The last time quantum is given at t_p.

VCPU belongs, and the number of VCPUs in the VM. Reflecting these factors, we define ratio $R_I(v_{ij})$ that gives the fair share of CPU to VCPU v_{ij} as follows:

$$R_I(v_{ij}) = \frac{|P| \times \omega(V_i)}{|C(V_i)|} \tag{1}$$

Because Eq. (1) indicates a ratio, for t (time), we should multiply t by $R_I(v_{ij})$, thus obtaining $t \times R_I(v_{ij})$ as the ideal amount of CPU time for VCPU v_{ij}.

However, in practice, for various reasons, the actual amount of obtained CPU time may be greater or smaller than the ideal amount. Thus, we need to define the actual amount of obtained CPU time of VCPU v_{ij} from time 0 to t, and we call it $Ob(t, i, j)$. Then, the deviation between the ideal and actual amount of CPU time from time 0 to t, $n(t, i, j)$, is defined as follows:

$$n(t, i, j) = t \times R_I(v_{ij}) - Ob(t, i, j) \tag{2}$$

This equation can be used in evaluating or controlling the fairness of the proportional-share scheduling strategies. The positive value of $n(t, i, j)$ means that VCPU v_{ij} has not obtained CPU bandwidth less than what it ideally should; the negative value means the opposite case. In general, the proportional-share strategies confine the upper bound of $|n(t, i, j)|$ to guarantee the fairness of CPU sharing less strictly than the ideal fair scheduling. Let us assign the upper bound of $|n(t, i, j)|$ to Lag. Then, we obtain the condition for fairness as follows:

$$\forall t, i, j, |n(t, i, j)| \leq Lag \tag{3}$$

Now, we can estimate the worst-case completion time of a task by using Eq. (3). For example, let us assume that a task is inserted in the ready queue at time 0. Then the first time quantum is given to VCPU v_{ij} at time t_1, and the last quantum is given at time t_p. We assume that p is larger than 1. This situation is illustrated in Fig. 5. In the figure, the execution time of each thread unit is not equal. Thread units are partitioned with different patterns of painting.

Then, the last time quantum can be given to VCPU v_{ij} while the condition $|n(t_p, i, j)| \leq Lag$ is true, according to Eq. (3). In order to consider the worst-case completion time, we let $n(t_p, i, j) = Lag$. Then, according to Eq. (2), we have

$$t_p = \lfloor \frac{Lag + Ob(t_p, i, j)}{R_I(v_{ij})} \rfloor \tag{4}$$

Because the virtual machine scheduler is called at every time slot, we use the floor function in Eq. (4). At the time of t_p, we have $Ob(t_p, i, j) = p - 1$ because VCPU v_{ij} has consumed $(p - 1)$ time slots until that time. Therefore, we have

$$t_p = \lfloor \frac{Lag + p - 1}{R_I(v_{ij})} \rfloor \tag{5}$$

The absolute amount of time quantum for the completion of thread T_{i^*} is determined by the sum of the normalized execution time of each thread unit. When we let $E_T = \sum_{k=1}^{|T_{i^*}|} E(T_{i^*}{}^k)$, the number of the required slots is $\lceil E_T \rceil$. When the last quantum is given at t_p, the last quantum is the pth time quantum. Thus, we obtain $\lceil E_T \rceil = p$. Because we assume that p is larger than 1, we have $\lceil E_T \rceil \geq 2$. Then, by (5), it follows that

$$t_p = \lfloor \frac{Lag + \lceil E_T \rceil - 1}{R_I(v_{ij})} \rfloor \tag{6}$$

The point where a task is finished in the last time quantum can be obtained by subtracting the amount of time until the $(p - 1)$th time quantum from E_T. The point is:

$$E_T - (p - 1) = E_T - (\lceil E_T \rceil - 1) \tag{7}$$

The maximum completion time for a task is obtained by summing Eq. (6) and (7). It follows that

$$\lfloor \frac{Lag + \lceil E_T \rceil - 1}{R_I(v_{ij})} \rfloor + E_T - (\lceil E_T \rceil - 1) \tag{8}$$

□

Co-scheduling. In this scheduling method, VCPUs in a VM are co-scheduled to the physical CPUs in the system, while the CPU bandwidth allocated to VMs is in proportion to the number of their weights [7], [8]. The co-scheduling policy forces every task to run on the VCPUs at the same time, as illustrated in Fig. 6. The co-scheduling strategy intrinsically guarantees the ideal fair-share of CPU sharing between virtual machines at every periodic interval.

Theorem 2. *In co-scheduling, the worst completion time for concurrent task T in VM V_i is as follows:*

$$\lfloor \frac{\lceil E_T \rceil}{R_I(v_{ij})} \rfloor + E_T - \lceil E_T \rceil$$

, where $R_I(v_{ij}) = \frac{|P| \times \omega(V_i)}{|C(V_i)|}$, $E_T = \sum_{k=1}^{|T_{i^*}|} E(T_{i^*}{}^k)$, *and* $\lceil E_T \rceil \geq 2$.

Proof. The co-scheduling method distributes CPU bandwidth to each VCPU in proportion to its weight in every periodic interval. Let us assume that a task is inserted in the ready queue at time 0, and VCPU v_{ij} has the final order in the periodic interval. As shown in Fig. 6, in the first interval, the time quantum of VCPU v_{ij} starts at t_1 and ends at $t_1 + 1$. In the last interval, the time quantum of VCPU v_{ij} starts at t_p and ends at $t_p + 1$. We assume that p is larger than 1.

Then, at the end of the last quantum, VCPU v_{ij} receives the ideal amount of CPU time by the scheduler. Then, by Eq. (2), we have $(t_p + 1) \times R_I(v_{ij}) = Ob(t_p + 1, i, j)$. Because VCPU v_{ij} has received p time slots until that time, the value of $Ob(t_p + 1, i, j)$ is p. Then, it follows that

$$t_p = \lfloor \frac{Ob(t_p+1,i,j)}{R_I(v_{ij})} \rfloor - 1 = \lfloor \frac{p}{R_I(v_{ij})} \rfloor - 1 \tag{9}$$

The absolute amount of time quantum for the completion of thread T_{i^*} is determined by the sum of the normalized execution time of each thread unit. When we let $E_T = \sum_{k=1}^{|T_{i^*}|} E({T_{i^*}}^k)$, the number of the required slots is $\lceil E_T \rceil$. When the last quantum starts at t_p, the last quantum is the pth time quantum. Thus, we obtain $\lceil E_T \rceil = p$. Because we assume that p is larger than 1, we have $\lceil E_T \rceil \geq 2$. Then, by (9), it follows that

$$t_p = \lfloor \frac{\lceil E_T \rceil}{R_I(v_{ij})} \rfloor - 1 \tag{10}$$

The point where a task is finished in the last time quantum can be obtained by subtracting the amount of time until the $(p-1)$th time quantum from E_T. The point is $E_T - (p-1) = E_T - (\lceil E_T \rceil - 1)$. Then, by (10), the maximum completion time for a task is obtained as follows:

$$\begin{aligned} &t_p + \Big(E_T - (\lceil E_T \rceil - 1)\Big) \\ &= \lfloor \frac{\lceil E_T \rceil}{R_I(v_{ij})} \rfloor + E_T - \lceil E_T \rceil \end{aligned} \tag{11}$$

□

Now, we analyze our two scheduling policies using the above models. First, we compare space-partitioning with the proportional-share strategy. Second, we compare time-partitioning with space-partitioning. These comparisons are meaningful because we theoretically reveal that even a small increase in cache utilization can improve the performance of parallel threads between the scheduling policies. In addition, if the cache utilization of each scheduler is given, we can estimate the performance improvement by obtaining the deviation of the completion time of each scheduler.

Space-partitioning.

Theorem 3. *When a concurrent thread shares some global memory regions with other VCPUs in the same VM, space-partitioning can improve the performance of the concurrent task at worst compared to the default proportional-share strategy.*

Proof. In general, a scheduling algorithm with the lower value of the worst completion time is preferable. Therefore, we compare the worst completion time of default proportional-share and space-partitioning scheduling. Then, we prove Theorem 3 by showing that the deviation of the worst completion time between default proportional-share and space-partitioning scheduling, D_S in the following context, is always positive.

In the default proportional-share strategy, we assume that parallel threads share large memory regions and are placed on different CPUs that do not share the last-level cache. Then, these threads would suffer from the coherent cache misses. Let ρ_N be the cache utilization in this case, and let ρ_S be the cache utilization in space-partitioning scheduling. Then, we have $\rho_N < \rho_S$ in all thread units because space-partitioning reduces cache coherence misses compared to the default proportional-share strategy by placing communicating threads in the same cache domain.

The space-partitioning algorithm basically adopts the proportional-share strategy. Then, the deviation of the worst completion time between the two scheduling algorithms, D_S, is obtained by subtracting Eq. (8) with ρ_S from Eq. (8) with ρ_N. When ρ_N is used, we refer to the sum of the execution time of all thread units as E_{TN}. Similarly, when ρ_S is used, we refer to the sum of the execution time of all thread units as E_{TS}. Then, D_S is:

$$\begin{aligned} D_S &= \Big(\lfloor \tfrac{Lag+\lceil E_{TN} \rceil - 1}{R_I(v_{ij})} \rfloor + E_{TN} - (\lceil E_{TN} \rceil - 1)\Big) \\ &-\Big(\lfloor \tfrac{Lag+\lceil E_{TS} \rceil - 1}{R_I(v_{ij})} \rfloor + E_{TS} - (\lceil E_{TS} \rceil - 1)\Big) \\ &= \Big(\lfloor \tfrac{Lag+\lceil E_{TN} \rceil - 1}{R_I(v_{ij})} \rfloor - \lfloor \tfrac{Lag+\lceil E_{TS} \rceil - 1}{R_I(v_{ij})} \rfloor\Big) \\ &+(E_{TN} - \lceil E_{TN} \rceil) - (E_{TS} - \lceil E_{TS} \rceil) \end{aligned} \tag{12}$$

In Eq. (12), the value of $(E_{TN} - E_{TS})$ is as follows:

$$\begin{aligned} E_{TN} - E_{TS} &= \sum_{k=1}^{|T_{i^*}|} \Big(\rho_{Ni^*k} \times C_{i^*k} + (1 - \rho_{Ni^*k}) \times M_{i^*k}\Big) \\ &- \sum_{k=1}^{|T_{i^*}|} \Big(\rho_{Si^*k} \times C_{i^*k} + (1 - \rho_{Si^*k}) \times M_{i^*k}\Big) \\ &= \sum_{k=1}^{|T_{i^*}|} \Big((\rho_{Si^*k} - \rho_{Ni^*k}) \times (M_{i^*k} - C_{i^*k})\Big) \end{aligned} \tag{13}$$

In Eq. (13), we have $\forall_k, \rho_{Si^*k} - \rho_{Ni^*k} > 0$, and $\forall_k, M_{i^*k} - C_{i^*k} > 0$ because M_{i^*k} means processing time of ${T_{i^*}}^k$ on the condition that all memory references induce last-level cache misses, and C_{i^*k} means the opposite case. Therefore,

$$E_{TN} - E_{TS} > 0 \tag{14}$$

Then, $\lceil E_{TN} - E_{TS} \rceil \geq 1$. Therefore, we obtain the condition as follows:

$$\lceil E_{TN} \rceil - \lceil E_{TS} \rceil \geq 0 \tag{15}$$

There are two cases to consider when $\lceil E_{TN} \rceil - \lceil E_{TS} \rceil \geq 0$. We first consider the case when $\lceil E_{TN} \rceil - \lceil E_{TS} \rceil = 0$. In this case, we have $D_S = E_{TN} - E_{TS}$ in Eq. (12). By (14), we have $D_S > 0$.

Next, we consider the case when $\lceil E_{TN} \rceil - \lceil E_{TS} \rceil > 0$ in

(15). In this case, we have

$$D_S = \left(\lfloor \frac{Lag+\lceil E_{TN}\rceil-1}{R_I(v_{ij})}\rfloor - \lfloor \frac{Lag+\lceil E_{TS}\rceil-1}{R_I(v_{ij})}\rfloor\right) + (E_{TN} - \lceil E_{TN}\rceil) - (E_{TS} - \lceil E_{TS}\rceil) \quad (16)$$

Then, in Eq. (16), the value of $\lfloor \frac{Lag+\lceil E_{TN}\rceil-1}{R_I(v_{ij})}\rfloor - \lfloor \frac{Lag+\lceil E_{TS}\rceil-1}{R_I(v_{ij})}\rfloor$ is as follows:

$$\lfloor \frac{Lag+\lceil E_{TN}\rceil-1}{R_I(v_{ij})}\rfloor - \lfloor \frac{Lag+\lceil E_{TS}\rceil-1}{R_I(v_{ij})}\rfloor = \begin{cases} \lfloor \frac{\lceil E_{TN}\rceil-\lceil E_{TS}\rceil}{R_I(v_{ij})}\rfloor & \text{if } \{\frac{Lag+\lceil E_{TN}\rceil-1}{R_I(v_{ij})}\} \geq \{\frac{Lag+\lceil E_{TS}\rceil-1}{R_I(v_{ij})}\} \\ \lfloor \frac{\lceil E_{TN}\rceil-\lceil E_{TS}\rceil}{R_I(v_{ij})}\rfloor + 1 & \text{if } \{\frac{Lag+\lceil E_{TN}\rceil-1}{R_I(v_{ij})}\} < \{\frac{Lag+\lceil E_{TS}\rceil-1}{R_I(v_{ij})}\} \end{cases} \quad (17)$$

Because we have $\lceil E_{TN}\rceil - \lceil E_{TS}\rceil > 0$, then it equals that $\lceil E_{TN}\rceil - \lceil E_{TS}\rceil \geq 1$. Moreover, the weight proportion of VM V_i cannot exceed the ratio of the number of its virtual CPUs to the number of physical CPUs in the system. Therefore, $\omega(V_i) \leq \frac{|C(V_i)|}{|P|}$. Then, $\frac{|C(V_i)|}{|P|\times\omega(V_i)} = \frac{1}{R_I(v_{ij})} \geq 1$. Thus, $\lfloor \frac{\lceil E_{TN}\rceil-\lceil E_{TS}\rceil}{R_I(v_{ij})}\rfloor \geq 1$. Therefore, in both cases in (17), we obtain the condition as follows:

$$\lfloor \frac{Lag+\lceil E_{TN}\rceil-1}{R_I(v_{ij})}\rfloor - \lfloor \frac{Lag+\lceil E_{TS}\rceil-1}{R_I(v_{ij})}\rfloor \geq 1$$

In Eq. (16), because we have $-1 < (E_{TN} - \lceil E_{TN}\rceil) - (E_{TS} - \lceil E_{TS}\rceil) < 1$, and $\lfloor \frac{Lag+\lceil E_{TN}\rceil-1}{R_I(v_{ij})}\rfloor - \lfloor \frac{Lag+\lceil E_{TS}\rceil-1}{R_I(v_{ij})}\rfloor \geq 1$, then $D_S > 0$.

We get our result of Theorem 3 because the deviation of the worst completion time between default proportional-share and space-partitioning scheduling (Eq. (12)), D_S, is always positive. □

Time-partitioning.

Theorem 4. *When a concurrent thread shares some global memory regions with other VCPUs in the same VM, time-partitioning with the aggressive method can improve the performance of the concurrent task at worst compared to space-partitioning.*

Proof. Like the proof of Theorem 3, we compare the worst completion time of the two scheduling algorithms. After the space-partitioning algorithm is applied, the time-partitioning algorithm with the aggressive method co-schedules VCPUs in the same temporal partition. Let ρ_S be the cache utilization in space-partitioning scheduling, and let ρ_T be the cache utilization in time-partitioning scheduling. Then, we have $\rho_S < \rho_T$ in all thread units because time-partitioning reduces cache conflict misses compared to space-partitioning by co-scheduling communicating threads in the same temporal partition.

The time-partitioning algorithm basically adopts the co-scheduling strategy while the space-partitioning algorithm adopts the proportional-share strategy. Then, the deviation of the worst completion time between the two scheduling algorithms, D_T, is obtained by subtracting Eq. (11) with ρ_T from Eq. (8) with ρ_S. When ρ_S is used, we refer to the sum of the execution time of all thread units as E_{TS}. Similarly, when ρ_T is used, we refer to the sum of the execution time of all thread units as E_{TT}. Then,

$$\begin{aligned} D_T &= \left(\lfloor \frac{Lag+\lceil E_{TS}\rceil-1}{R_I(v_{ij})}\rfloor + E_{TS} - (\lceil E_{TS}\rceil - 1)\right) \\ &-\left(\lfloor \frac{\lceil E_{TT}\rceil}{R_I(v_{ij})}\rfloor + E_{TT} - \lceil E_{TT}\rceil\right) \\ &= \left(\lfloor \frac{Lag+\lceil E_{TS}\rceil-1}{R_I(v_{ij})}\rfloor - \lfloor \frac{\lceil E_{TT}\rceil}{R_I(v_{ij})}\rfloor\right) \\ &+(E_{TS} - \lceil E_{TS}\rceil) - (E_{TT} - \lceil E_{TT}\rceil) + 1 \end{aligned} \quad (18)$$

Then, in Eq. (18), the value of $\lfloor \frac{Lag+\lceil E_{TS}\rceil-1}{R_I(v_{ij})}\rfloor - \lfloor \frac{\lceil E_{TT}\rceil}{R_I(v_{ij})}\rfloor$ is as follows:

$$\lfloor \frac{Lag+\lceil E_{TS}\rceil-1}{R_I(v_{ij})}\rfloor - \lfloor \frac{\lceil E_{TT}\rceil}{R_I(v_{ij})}\rfloor = \begin{cases} \lfloor \frac{(Lag-1)+(\lceil E_{TS}\rceil-\lceil E_{TT}\rceil)}{R_I(v_{ij})}\rfloor & \text{if } \{\frac{Lag+\lceil E_{TS}\rceil-1}{R_I(v_{ij})}\} \geq \{\frac{\lceil E_{TT}\rceil}{R_I(v_{ij})}\} \\ \lfloor \frac{(Lag-1)+(\lceil E_{TS}\rceil-\lceil E_{TT}\rceil)}{R_I(v_{ij})}\rfloor + 1 & \text{if } \{\frac{Lag+\lceil E_{TS}\rceil-1}{R_I(v_{ij})}\} < \{\frac{\lceil E_{TT}\rceil}{R_I(v_{ij})}\} \end{cases} \quad (19)$$

Because we have $\rho_T - \rho_S > 0$, then like the proof of Theorem 3, we have $E_{TS} - E_{TT} > 0$. This proof is omitted due to similarity. Then, $\lceil E_{TS} - E_{TT}\rceil \geq 1$. Therefore, $\lceil E_{TS}\rceil - \lceil E_{TT}\rceil \geq 0$.

Because general proportional-share strategies have the value of *Lag* usually equal to or larger than one [8], we have $Lag - 1 \geq 0$. Moreover, according to the proof of Theorem 3, we have $\frac{1}{R_I(v_{ij})} \geq 1$. Thus, $\lfloor \frac{(Lag-1)+(\lceil E_{TS}\rceil-\lceil E_{TT}\rceil)}{R_I(v_{ij})}\rfloor \geq 0$. Therefore, in both cases in (19), we obtain the condition as follows:

$$\lfloor \frac{Lag+\lceil E_{TS}\rceil-1}{R_I(v_{ij})}\rfloor - \lfloor \frac{\lceil E_{TT}\rceil}{R_I(v_{ij})}\rfloor \geq 0$$

In Eq. (18), because we have $0 < (E_{TS} - \lceil E_{TS}\rceil) - (E_{TT} - \lceil E_{TT}\rceil) + 1 < 2$, and $\lfloor \frac{Lag+\lceil E_{TS}\rceil-1}{R_I(v_{ij})}\rfloor - \lfloor \frac{\lceil E_{TT}\rceil}{R_I(v_{ij})}\rfloor \geq 0$, then $D_T > 0$.

Therefore, we get our result of Theorem 4 because the deviation of the worst completion time between space-partitioning and time-partitioning scheduling, D_T, is always positive. □

We are only concerned with the aggressive method in time-partitioning because the smooth method actively adopts both the co-scheduling and default proportional strategy according to the existence of I/O boosted VCPUs. Therefore, its performance would lie between that of the aggressive method and the space-partitioning algorithm.

5. Implementation

This section describes the major implementation of our cache-aware scheduler in the Xen hypervisor. We define a new scheduler, sched_cache-aware_def, based on the credit scheduler using common scheduler interface provided by Xen.

5.1 System Call Interception in the Hypervisor

As explained in Sect. 3.1, we modify the hypervisor slightly and make system calls go through the hypervisor to support

the detection module. Xen-x86's direct trap optimization, which is also referred to as fast system call optimization, allows the system calls to bypass the hypervisor. Therefore, we disable the optimization by commenting the following lines in do_set_trap_table() function as illustrated in [14].

```
if ( cur.vector == 0x80 )
  init_int80_direct_trap(curr);
```

Then, the system call number can be intercepted in do_guest_trap() function by adding following lines.

```
static void do_guest_trap(
    int trapnr, const struct cpu_user_regs *regs, int use_error_code) {
  ..... /* omitted */

  int system_call_number;
  struct cpu_user_regs *user_regs;
  user_regs = guest_cpu_user_regs();

  /* retrieve the system call number */
  system_call_number = (int) user_regs->eax;

  ..... /* omitted */
}
```

5.2 Time-Partitioning Implementation

As illustrated in Sect. 3.3, when the cluster representative is selected to run on any physical core, the hypervisor prepares to co-schedule other related VCPUs. In the end of cache-aware_schedule() function, which is our do_schedule() function, the hypervisor executes co-schedule() function indirectly by softirq(), because cache-aware_schedule() is in the critical path and should be designed to be simple and fast.

```
/* cluster_svc means a cluster representative. */
if (cluster_svc != NULL) {
  raise_softirq(COSCHEDULE_SOFTIRQ);
}
```

The co-schedule function() is executed on the core where the cluster representative is running. The objective of the co-schedule() function is to change the priority of clustered VCPUs to AGGRESSIVE or SMOOTH and to make them get ready to run. If any related VCPU is in RUNSTATE_runnable state, the VCPU is already in the local run queue. We change the priority of the VCPU to AGGRESSIVE or SMOOTH and set a CPU map variable to notify the core, where the VCPU belongs, that run queue needs to be sorted.

```
/* cluster_svc means an individual cluster member in traversing phase of
      cluster list. */
if (vcpu->runstate.state == RUNSTATE_runnable && cluster_svc->pri
      >= CSCHED_PRI_TS_UNDER) {
  /* if the method is aggressive one. */
  cluster_svc->pri = CSCHED_PRI_TS_AGGRESSIVE;
  /* if the method is smooth one. */
  cluster_svc->pri = CSCHED_PRI_TS_SMOOTH;
  cpu_set(cluster_svc->vcpu->processor, co_map);
}
/* if co_map is set, the hypervisor tickles other cores */
cpumask_raise_softirq(co_map, SCHEDULE_SOFTIRQ);
```

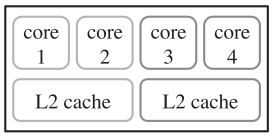


Fig. 7 A schematic view of an Intel Core2 Quad processor.

If any related VCPU is in RUNSTATE_blocked state, the co-schedule() function wakes the VCPU up. Then, the function inserts the VCPU in the run queue according to their priority and tickles the core where the processor field of the VCPU belongs.

```
/* cluster_svc means an individual cluster member in traversing phase of
      cluster list. */
if(vcpu->runstate.state == RUNSTATE_blocked && cluster_svc->pri
      >= CSCHED_PRI_TS_UNDER) {
  /* the code of priority setting is omitted */
  co-schedule_vcpu_unblock(cluster_svc->vcpu);
}
```

6. Evaluation

We have implemented a cache-aware virtual machine scheduler on an Intel Q6600 quad-core platform that has four 2.40 GHz cores with two 4 MB of L2 caches, each of which is shared by two cores. The CPU layout of this system is illustrated in Fig. 7. The system is hosted by the recent Xen 4.0.1 and para-virtualized Linux-2.6.31 (paravirt_ops kernel) with a Xenoprofile patch.

Although our experiments are conducted on an Intel quad-core processor that has two cache domains, our scheduler can also work for any SMP systems that have at least two cache domains. A system that contains only one cache domain naturally cannot take advantage of the space-partitioning algorithm, but the time-partitioning algorithm alone.

6.1 Workloads

6.1.1 Workloads for Concurrent VMs

VolanoMark [10] is a benchmarking program that consists of a chat server along with a driver that simulates thousands of chat client threads. When the program is run, chat threads belonging to one chat room experience intensive data sharing while communicating with each other. We designate the VM that runs this benchmark as the concurrent VM with a considerable number of shared regions. We run VolanoMark with its default setting, where the program creates client connections in groups of 20 for each loopback connection.

SPLASH-2 (Stanford Parallel Applications for Shared Memory) [16] consists of scientific and graphical parallel applications and is being used as a multi-processor benchmarking program since many years. In SPLASH-2, we use an LU kernel program, which factors a dense matrix into the product of a lower triangular and an upper triangular

matrix. As the LU kernel is designed to reduce communications among the processors, the benchmark does not have large shared regions. However, to observe the performance implications when our scheduling algorithm is applied to general parallel applications, we apply the space and time-partitioning algorithms to the VM running LU by force. In order to reflect the current multi-core's performance, we aggressively run the LU with large memory footprints. The settings are n = 4096, and B = 16, and the number of processors is 2.

For reference, according to the prior researches [3], [5], other server benchmarking programs including SPECjbb2005 [17] and RUBis [18] can benefit from our cache-aware scheduling algorithms.

6.1.2 Workloads for Other VMs

In general, in the virtualization environment, as different users run various applications, we attempt to cover miscellaneous types of workloads such as CPU-intensive or IO-intensive applications.

Iperf [19] measures the maximum TCP and the UDP bandwidth performance. We have another multi-core system that stresses Iperf programs through the network.

SPEC CPU2006 [20] is a CPU-intensive benchmark suite, stressing a system's processor and the memory subsystem. In our experiment, in order to stress both CPU and memory, we use the mcf benchmark in CPU2006.

STREAM [21] is a benchmark program that measures the sustained memory bandwidth at all levels of the cache hierarchy.

Matrix is our own micro-benchmark for CPU-intensive workloads. It calculates the matrix before a user terminates it while using small memory footprints and generating mainly CPU-intensive works.

6.1.3 Experimental Methodology

We only focus on the performance impact of the concurrent VMs and the VMs running I/O intensive works because our scheduler targets an improvement in the cache utilization of the concurrent VMs while not sacrificing the I/O intensive domains. We measure the performance of VolanoMark, LU in SPLASH-2, or Iperf in each VM while running other workloads repeatedly.

In this experiment, only in the case of the concurrent VMs, we restrict the maximum number of VCPUs to be equal to the size of the cache domain in order to thoroughly exploit the locality of the last-level cache. For example, in our quad-core systems, the number of VCPUs for the concurrent domain would be 2, and in dual six-core systems with each processor sharing one last-level cache, the number of VCPUs would be 6.

6.2 Scheduling Fairness

As illustrated in Sect. 3.3.3, in order to guarantee fair scheduling by the credit scheduler, we conditionally set VCPUs to a high priority when the cluster representative is selected. In particular, we do not co-schedule the related VCPUs with the OVER priority. We carry out an evaluation to know whether fairness is guaranteed or not.

Table 1 Credit usage with space partitioning.

Domain	Cache Domain	Execution Time	Credit Amount
1	0	66.45 s	612,242
2	0	65.19 s	601,498
3	1	65.89 s	608,808
4	1	65.66 s	604,990

Table 2 Credit usage with time partitioning (aggressive method).

Domain	Cache Domain	Higher Priority	Execution Time	Credit Amount
1	0	yes	61.78 s	575,669
2	0	no	62.44 s	584,265
3	1	yes	62.55 s	579,353
4	1	no	62.25 s	580,839

We develop four virtual machines with each machine having two VCPUs and running an LU kernel program inside them. First, we apply the space-partitioning algorithm by force to all VMs. Because the space-partitioning algorithm scatters groups of clustered VCPUs among cache domains, there will be two domains in each cache domain. As summarized in Table 1, the performances of the LU program in all VMs are almost the same, and each LU program consumes subequal credit amounts. This result is a performance baseline. Second, while keeping the space-partitioning algorithm, we apply the time-partitioning algorithm with an aggressive method to only one of the VMs in the same cache domain. Then, as summarized in Table 2, the performances of the LU are slightly faster than those in the case when using only the space-partitioning algorithm. This is because, although the LU program does not have large shared regions, the strict co-scheduling algorithm lightens the synchronization problem in the LU as pointed out in [8]. In this experiment, although one of the VMs in the same cache domain gets higher priority, each LU program consumes subequal credit amounts. Therefore, we believe that the strict co-scheduling algorithm certainly does not hamper fairness.

6.3 Single Workloads

For testing the basic performance of our scheduler, we measure the throughputs of VolanoMark and LU. In each experiment, either for VolanoMark or for LU, we configure two virtual machines with two VCPUs, as summarized in Table 3. Each VM's weight is 256 and memory size is 384 M. We measure the performance of each application when the default (the credit scheduler in Xen), space, and time partitioning with the aggressive method are used. As there are no I/O intensive domains, the experiment involving the time-partitioning algorithm with the smooth method is omitted.

Table 3 Single workloads. IDD stands for isolated driver domain (IDD) that runs most of the native device drivers. Domain0 plays the role of an IDD in this configuration.

Dom	Workloads	VCPUs	Memory	Option
0	IDD	4	1024 M	
1	VolanoMark /LU	2	384 M	designated VM
2	VolanoMark /LU	2	384 M	designated VM

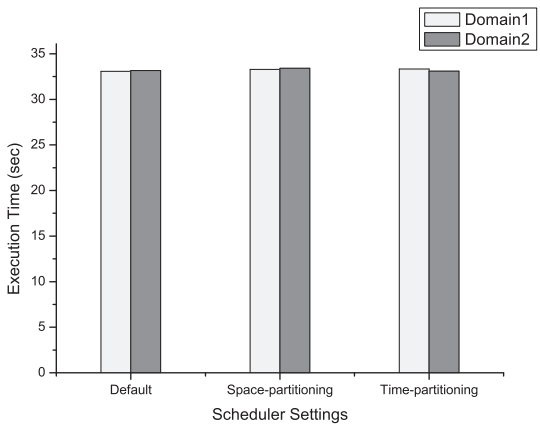


Fig. 8 Performance of LU in the single workloads (lower is better).

Table 4 Average last-level cache misses of LU in the single workloads. We measure samples of last-level cache (LLC) misses using Xenoprofile with the reset counter value of 6,000.

Config	Default	Space partitioned	Aggressive
LLC-user	44,186	43,014	43,416
LLC-kernel	283	199	201
LLC-total	44,469	43,213	43,617

Figure 8 shows the impact of different scheduling policies on the performance of single LU program. Table 4 exhibits the average last-level cache misses. The results indicate that there are few differences in performance and cache misses for all scheduler settings. Because LU has few shared regions between processes in a VM, exploiting the topology of multi-core processors has little meaning to the LU.

Figure 9 shows the performance of VolanoMark according to the different scheduling policies. Table 5 exhibits the average last-level cache misses. As the results show, our scheduler improves the performance of VolanoMark by reducing cost consumed by cache coherence protocols. Our scheduler can decrease the last-level cache misses by up to 38% and improve the performance of the application by up to 13% as compared to the default credit scheduler. In this experiment, space and time-partitioning algorithms have almost the same performance impact, because there are no other VMs except domain0 that is mainly in idle state.

6.4 Mixed Workloads

We construct mixed workloads, as described in Table 6.

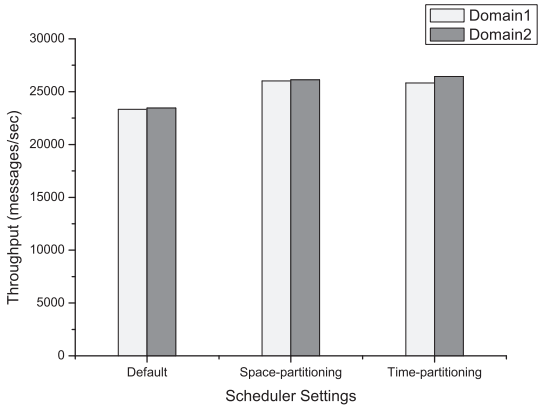


Fig. 9 Performance of VolanoMark in the single workloads (higher is better).

Table 5 Average last-level cache misses of VolanoMark in the single workloads.

Config	Default	Space partitioned	Aggressive
LLC-user	30,224	19,916	19,795
LLC-kernel	35,821	21,487	21,200
LLC-total	66,045	41,403	40,995

Table 6 Mixed workloads.

Dom	Workloads	VCPUs	Memory	Option
0	IDD	4	1024 M	
1	VolanoMark /LU	2	384 M	designated VM
2	VolanoMark /LU	2	384 M	designated VM
3	STREAM	4	256 M	4 processes
4	mcf	2	512 M	2 processes
5	Matrix	4	256 M	4 processes
6	Iperf	4	256 M	4 threads
6	Iperf	4	256 M	4 threads

The mixed workloads consist of two concurrent domains (VolanoMark or LU workload), two I/O intensive domains (Iperf), three other domains, and an IDD domain. Each domain's weight is 256 and memory size varies according to the workload characteristics. We measure the throughput of concurrent VMs when the each execution of concurrent applications is terminated. Because the space-partitioning scatters groups of clustered VCPUs among the cache domains, concurrent domains are located in the different cache domains except the default scheduler.

Figure 10 shows the performance of LU in the mixed workloads. Table 7 exhibits the average last-level cache misses. Similar to the results of the single workloads, our scheduler does not improve the performance of LU, because LU has few shared regions. However, the execution time of LU slightly becomes shorter when the time-partitioning is used, because the co-scheduling algorithm lightens the synchronization problem as pointed out in [8].

Figure 11 shows the impact of different scheduling policies on performance of VolanoMark in the mixed work-

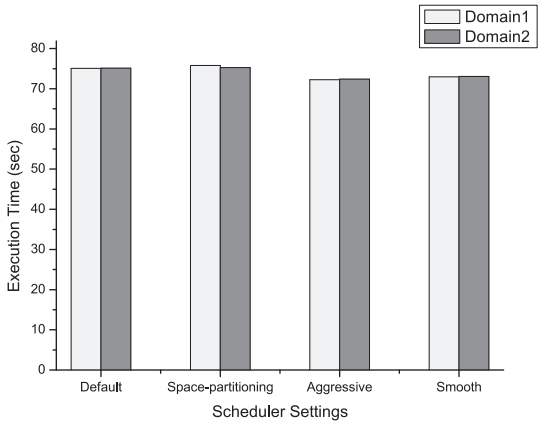


Fig. 10 Performance of LU in the mixed workloads (lower is better).

Table 7 Average last-level cache misses of LU in the mixed workloads.

Config	Default	Space partitioned	Aggressive	Smooth
LLC-user	40,637	42,215	42,663	42,640
LLC-kernel	573	665	513	477
LLC-total	41,210	42,880	43,176	43,117

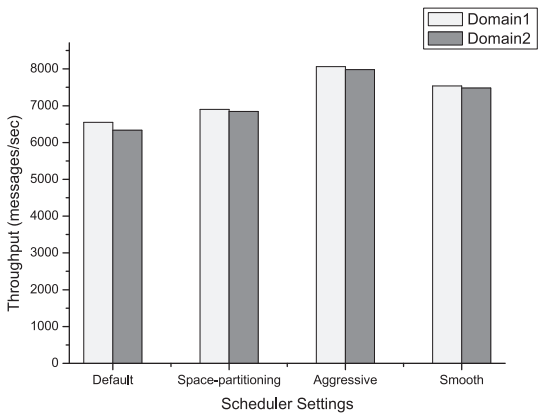


Fig. 11 Performance of VolanoMark in the mixed workloads (higher is better).

Table 8 Average last-level cache misses of VolanoMark in the mixed workloads.

Config	Default	Space partitioned	Aggressive	Smooth
LLC-user	35,045	32,832	29,244	30,420
LLC-kernel	40,534	35,467	31,741	33,040
LLC-total	75,579	68,299	60,985	63,460

loads. Table 8 exhibits the average last-level cache misses. As the results show, the space-partitioning in the mixed workloads could slightly improve the performance, because other VMs may sweep out cached contents due to the asynchronous scheduling. After adopting the time-partitioning, our scheduler can decrease the last-level cache misses by up to 16% and improve performance of VolanoMark by up to 19% as compared to the default credit scheduler. These results indicate that the time-partitioning can maximize utilization of the last-level cache by reducing redundant data evictions and reloads.

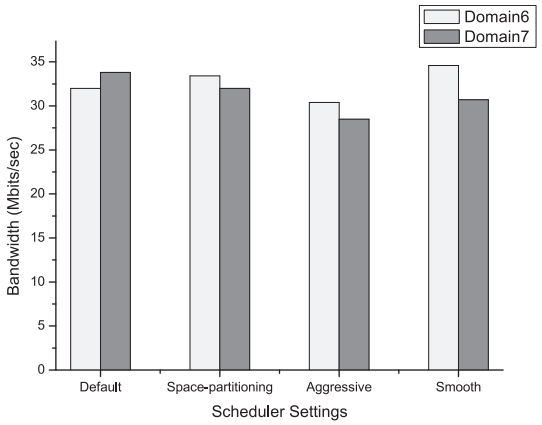


Fig. 12 Performance of Iperf when running LU in the mixed workloads.

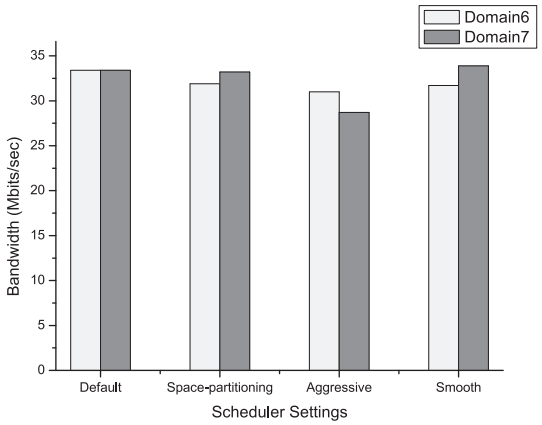


Fig. 13 Performance of Iperf when running VolanoMark in the mixed workloads.

Figure 12 and 13 exhibit the bandwidths of Iperf in each experiment. These results present the impact of the aggressive method on the performance of I/O domains. The aggressive method decreases the performance of Iperf by up to 14% because the strict co-scheduling algorithm in the aggressive method may disregard the I/O boost mechanism of the credit scheduler. We can address this problem by adopting the smooth method that always respects the I/O boosted VCPU. The smooth method also enables high utilization of the last-level cache, and the result is summarized in Table 8.

6.5 Overhead of System Call Interception

We use the system call interception mechanism in the detection module of concurrent applications. We measure the cost of system call interception using the lmbench program [22]. The cost of executing the system calls are listed in Table 9. Unfortunately, the fast syscall implementation outperforms our mechanism with respect to the execution of the read and write calls. Our interception mechanism exhibits a slightly slower performance than the original implementation with

Table 9 Overhead of system call interception: lmbench system calls - times in micro second.

Config	null syscall	read	write	stat	open close
Fast syscall	0.4	0.3	0.3	5.5	12.3
Interception	0.5	0.6	0.6	5.8	13.2

Table 10 Average throughput (messages/s) of each VolanoMark in four VMs, each of which has a different number of VCPUs (1, 2, 3, and 4 VCPUs) from each other.

Config	1 VCPU VM	2 VCPU VM	3 VCPU VM	4 VCPU VM
Default scheduler	11340	8430	2336	1042
Time-partitioning	11250	9697	2111	927

respect to the execution of the null, stat, and open/close system calls. Therefore, we periodically enable the detection module for a certain time quantum in order to not degrade system performance.

6.6 Discussion of Bad Cases

To evaluate the bad case of the space-partitioning algorithm, we conduct an experiment in which the number of concurrent VMs is not even. In this experiment, the VCPUs of any cluster should be executed across the cache domain in order to maintain the fairness policy of the credit scheduler. We develop three virtual machines, each of which has two VCPUs, and run VolanoMark or LU on them.

In the case of VolanoMark, when the default credit scheduler is used, the average throughput of the three virtual machines is 12965 messages/s. When the space-partitioning algorithm is applied, the average throughput is 14116 messages/s. The proposed algorithm improves the performance by approximately 9% as compared to the default credit scheduler; in the best situation, the improvement is 13%, as discussed in Sect. 6.2. In this experiment, although any one of the three clusters is executed across the cache domain, the other two clusters can benefit from the space-partitioning algorithm. Therefore, our scheduler can improve performance more significantly than the default credit scheduler and cover this bad case adequately.

In the case of LU, because it is designed to minimize communications among the processors, the benchmark does not have large shared regions. Therefore, there is no performance difference between the default credit scheduler and our space-partitioning algorithm.

To evaluate the bad case of the time-partitioning algorithm, we conduct an experiment in which the number of the communicating VCPUs is different among VMs. We prepare 4 VMs, each of which has a different number of VCPUs (1, 2, 3, and 4 VCPUs) from each other, and run VolanoMark or LU on them. The experimental result, in the case of VolanoMark, is summarized in Table 10. Optimal performance is achieved in the 2-VCPU VM case, and in the 3- and 4-VCPU VM cases, the average throughput begins to deteriorate gradually as compared to that in the case of the default credit scheduler.

However, this performance decline is caused not by the low utilization of the last-level cache but by the inherent characteristic of VolanoMark. VolanoMark has the client-server architecture; it consists of a chat server along with a driver that simulates thousands of chat client threads. In the worst case, if the co-executing VCPUs only have client threads, the VCPUs cannot proceed because they have to wait for the responses from a server thread that waits to be executed in another time partition. Because of this reason, the average throughput cannot help but decrease. Therefore, in particular, when the workload has the client-server architecture, administrators have to pay attention to the decision on the number of VCPUs in the concurrent VMs.

In the case of LU, there is no performance difference as in the case of the space-partitioning algorithm.

7. Related Work

For the credit scheduler in Xen, several pieces of work, [8], [9], [23], [24], present credit specific problems. Although credit is widely accepted for general-purpose guest domains, mixed workload and time sensitive applications still have limitations since boost highly depends upon simple prioritization. So, original credit scheduling cannot support complicated cases such as simultaneous boosting multiple guest domains or complex domains that have mixed workloads.

[9] focused on the performances of multiple VMs when all domains are boosted simultaneously. The authors revealed that boost domain is always traversed from fixed guest domain, and unfair performance is presented. The authors have presented a fair-event-channel notification mechanism, with several optimizations, which includes preemption minimization, and VCPU ordering by remaining credit.

Another paper [23] tackles the performance problem of mixed workload guest domains. Since the hypervisor cannot understand the workload characteristics of the guest domain, it simply boosts a domain whenever a guest OS triggers an I/O event, so the authors make the hypervisor aware of the workload characteristics of the guest domain. Using gray-box knowledge, hypervisor can distinguish I/O bound domains from the others, and separately boost guest domains.

Recently, another Xen scheduler for supporting soft real-time VMs has been presented [24]. For VMs running time-sensitive workloads, such as virtual media servers, poor voice quality has presented even though credit has not been consumed. By introducing laxity into VM scheduling, the authors can accurately prioritize a VCPU so that the hypervisor can timely schedule guest domains.

Regarding the co-scheduling of VCPUs in VMs, hybrid scheduling [8] has been proposed. Since VCPUs in Xen are asynchronously dispatched and scheduled by the hypervisor, applications in a concurrent VM cannot take advantage of co-scheduling algorithm inside guest OSs. The authors proposed a new scheduling framework that can run both a concurrent VM and a high-throughput VM. However, the framework does not consider the cache awareness and the

boost mechanism in Xen as compared to our research.

In terms of cache-aware scheduler research, [3] deals with cache sharing issues between parallel threads. The authors proposed a thread clustering scheme based on sharing patterns detected on-line. They dynamically identify threads that frequently access shared data and group the threads to the same cache domain. To identify shared regions between threads, they used special features of performance monitoring units available in Power5 processors. They concentrated on a technique to detect shared regions rather than a scheduling algorithm.

Another cache-aware scheduler [25] deals with the cache contention issues between threads. The authors have found that the co-executing memory-bound threads in the same cache domain deteriorate the performance of threads because of a shared resource contention including a last-level cache in a processor. They show that scattering memory-bound threads into different last-level caches increases performance by reducing inter-core cache misses.

[26] proposed cache-aware credit scheduler to achieve high network bandwidth. Xen hypervisor processes network packets by the inter-domain communication between a network driver domain and a domain requesting network processing. By clustering each VCPU of domain0 and domainU into shared cache regions, they reduced network performance overhead. Our research is different from their research in that we focus on the intra-domain communication to improve the performance of concurrent applications. In addition, we provide a time-partitioning algorithm to maximize cache utilization.

8. Discussion

To exploit the multicore topologies, cache-aware schedulers dealing with cache sharing and contention have been suggested mainly at the OS level [3]–[5], [25]. In this section, we present the reasons why we cannot simply apply the existing cache-aware schedulers to our cache-aware VM scheduler.

With respect to cache sharing, [3], [4], and [5] focus on the thread-clustering scheme that gathers communicating threads to the same cache domain. These methods are effective if the threads are scheduled on the same cache domain as close as possible in terms of time. Because these methods assume that threads execute on a single operating system, the above condition is met easily.

However, in the virtualization environment, there may be many VCPUs from different operating systems. The problem is that VCPUs are scheduled asynchronously in many hypervisors including Xen. Therefore, as the number of virtual machines increases, the communicating threads in a single virtual machine have relatively few opportunities to be scheduled as close as possible in terms of time. To address this problem, we develop a new time-partitioning algorithm that schedules the communicating VCPUs as close to each other as possible in terms of time.

In terms of cache contention, [4] and [25] first identify threads that continually evict the cache content of other applications and then arrange such threads in order such that other applications are not hurt. In order to identify offensive memory-bound threads, [4] makes the cache miss-rate curve (MRC) online and [25] simply obtains the cache miss rate using hardware performance counters.

They assume that all the threads in the system are independent and do not share any data. Therefore, they adopt single-threaded and scientific benchmark programs such as SPEC CPU2006 [20]. The problem here is that they develop scheduling algorithms only for single-threaded programs. Because these methods will not work appropriately with parallel threads, we cannot apply them to our VM scheduler. For example, [25] distributes memory-bound threads into different last-level caches to decrease cache contention. In our case, if parallel threads in a VM are scattered into different last-level caches, they certainly would suffer from the coherent cache misses as shown in the evaluation of the space-partitioning algorithm.

9. Conclusion

Virtualization software has become a foundation technology of cloud computing. At the same time, as multi-core processors are pervasive in the cloud computing environments, the issue of utilizing multi-core resources in the virtualization layer has become a challenging problem. From that point of view, we endeavored to design a cache-aware VM scheduler that could allocate shared resources of the processor efficiently on multi-core architecture.

In our cache-aware VM scheduler, we presented the detection mechanism of concurrent applications in order to support smart scheduling of the concurrent VMs. We also provided a space-partitioning algorithm that clusters communicating VCPUs on the same cache domain and a time-partitioning algorithm that co-schedules clustered VCPUs at nearly the same time. Using these two algorithms, we could exploit the locality of the last-level caches and improve the performance of concurrent applications. Finally, we presented the theoretical analysis that proves our scheduling algorithm is more efficient than the default credit scheduler in Xen.

Acknowledgements

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No.2011-0029848).

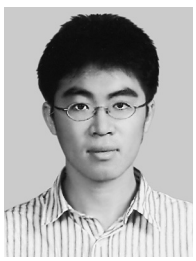
References

[1] E. Amazon, "Amazon elastic compute cloud," Retrieved Feb, vol.10, 2009.

[2] 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," Proc. Nineteenth ACM Symposium on Operating Systems Principles, pp.164–177, 2003.

[3] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors," Proc. 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, pp.47–58, 2007.
[4] R. Azimi, D. Tam, L. Soares, and M. Stumm, "Enhancing operating system support for multicore processors by using hardware performance monitoring," ACM SIGOPS Operating Systems Review, vol.43, no.2, pp.56–65, 2009.
[5] S. Vaddagiri, B. Rao, V. Srinivasan, A. Janakiraman, B. Singh, and V. Sukthankar, "Scaling software on multi-core through co-scheduling of related tasks," Linux Symp, pp.287–295, 2009.
[6] M. Bhadauria and S. McKee, "An approach to resource-aware co-scheduling for CMPs," Proc. 24th ACM International Conference on Supercomputing, pp.189–199, 2010.
[7] E. VMware, "Server," User Manual, Version, vol.1.
[8] 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.
[9] 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.
[10] L. Volano, "VolanoMark benchmark," 1999.
[11] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three CPU schedulers in Xen," Performance Evaluation Review, vol.35, no.2, p.42, 2007.
[12] D. Wang, D. Gao, and D. Wang, "A sharing-aware actively pushing cache on CMP," 2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE), pp.286–291, 2010.
[13] C. Xu, Y. Bai, and C. Luo, "Performance Evaluation of Parallel Programming in Virtual Machine Environment," 2009 Sixth IFIP International Conference on Network and Parallel Computing, pp.140–147, 2009.
[14] F. Beck and O. Festor, "Syscall Interception in Xen Hypervisor," 2009.
[15] K. Hwang and Z. Xu, Scalable parallel computing: technology, architecture, programming, McGraw-Hill, 1998.
[16] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," Proc. 22nd Annual International Symposium on Computer Architecture, pp.24–36, 1995.
[17] A. Adamson, D. Dagastine, and S. Sarne, "Specjbb2005—A year in the life of a benchmark," 2007 SPEC Benchmark Workshop, 2007.
[18] C. Amza, A. Chanda, A. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite, "Specification and implementation of dynamic web site benchmarks," 2002 IEEE International Workshop on Workload Characterization, 2002, WWC-5, pp.3–13, 2002.
[19] A. Tirumala, J. Ferguson, J. Dugan, F. Qin, and K. Gibbs, "Iperf," Avaliable from: http://dast.nlanr.net/Projects/Iperf
[20] J. Henning, "SPEC CPU2006 benchmark descriptions," ACM SIGARCH Computer Architecture News, vol.34, no.4, pp.1–17, 2006.
[21] J. McCalpin, "STREAM benchmark," 1995.
[22] L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," Proc. 1996 annual conference on USENIX Annual Technical Conference, p.23, 1996.
[23] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee, "Task-aware virtual machine scheduling for I/O performance.," Proc. 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, pp.101–110, 2009.
[24] M. Lee, A. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik, "Supporting soft real-time tasks in the xen hypervisor," Proc. 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, pp.97–108, 2010.
[25] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," ACM SIGARCH Computer Architecture News, pp.129–142, 2010.
[26] G. Liao, D. Guo, L. Bhuyan, and S. King, "Software techniques to improve virtualized I/O performance on multi-core systems," Proc. 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, pp.161–170, 2008.

Cheol-Ho Hong received the B.S. and M.S. degrees in computer science and engineering from Korea University, Seoul, Korea, in 2001 and 2003, respectively. He worked as a senior researcher at Netville, Seoul, Korea, from 2003 to 2006. Currently, he is a Ph.D. candidate of Korea University, Seoul, Korea. His research interests include hypervisor, multi-core architecture, and embedded system.



Young-Pil Kim received the B.S. and M.S. degrees in computer science and engineering from Korea University, Seoul, Korea, in 2002 and 2004, respectively. He is currently a Ph.D. candidate at Korea University, Seoul, Korea. His current interest is cloud computing.

Seehwan Yoo received the B.S. and M.S. degrees in computer science and engineering from Korea University, Seoul, Korea, in 2002 and 2004, respectively. He is currently a Ph.D. candidate at Korea University, Seoul, Korea. His current interest is system virtualization.

Chi-Young Lee received the B.S. degree in computer science from Chungnam National University, Daejeon, Korea, in 2006 and the M.S. degree in computer science from Korea University, Seoul, Korea, in 2008. Currently, he is a Ph.D. candidate of Korea University, Seoul, Korea. His research interests include network and embedded virtualization.

Chuck Yoo received the B.S. degree in electronic engineering from Seoul National University, Seoul, Korea and the M.S. and Ph.D. in computer science from University of Michigan. He worked as a researcher in Sun Microsystems Laboratory, from 1990 to 1995. He is now a professor in department of computer science and engineering, Korea University, Seoul, Korea. His research interests include high-performance networks, multimedia streaming, and operating systems. He served as a member of the organizing committee for NOSSDAV 2001.