Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Task mapper and application-aware virtual machine scheduler oriented for parallel computing[*]

Jing ZHANG[1,2], Xiao-jun CHEN[‡1], Jun-huai LI[1], Xiang LI[1]
([1]*School of Computer Science and Engineering, Xi'an University of Technology, Xi'an 710048, China*)
([2]*State Key Laboratory for Manufacturing Systems Engineering, Xi'an Jiaotong University, Xi'an 710048, China*)
E-mail: ZhangJing@xaut.edu.cn; army.net@163.com; lijunhuai@xaut.edu.cn; lixiang@163.com

Received July 27, 2011; Revision accepted Jan. 6, 2012; Crosschecked Feb. 9, 2012

Abstract: We design a task mapper TPCM for assigning tasks to virtual machines, and an application-aware virtual machine scheduler TPCS oriented for parallel computing to achieve a high performance in virtual computing systems. To solve the problem of mapping tasks to virtual machines, a virtual machine mapping algorithm (VMMA) in TPCM is presented to achieve load balance in a cluster. Based on such mapping results, TPCS is constructed including three components: a middleware supporting an application-driven scheduling, a device driver in the guest OS kernel, and a virtual machine scheduling algorithm. These components are implemented in the user space, guest OS, and the CPU virtualization subsystem of the Xen hypervisor, respectively. In TPCS, the progress statuses of tasks are transmitted to the underlying kernel from the user space, thus enabling virtual machine scheduling policy to schedule based on the progress of tasks. This policy aims to exchange completion time of tasks for resource utilization. Experimental results show that TPCM can mine the parallelism among tasks to implement the mapping from tasks to virtual machines based on the relations among subtasks. The TPCS scheduler can complete the tasks in a shorter time than can Credit and other schedulers, because it uses task progress to ensure that the tasks in virtual machines complete simultaneously, thereby reducing the time spent in pending, synchronization, communication, and switching. Therefore, parallel tasks can collaborate with each other to achieve higher resource utilization and lower overheads. We conclude that the TPCS scheduler can overcome the shortcomings of present algorithms in perceiving the progress of tasks, making it better than schedulers currently used in parallel computing.

Key words: Virtual machine, Virtualization, Application-aware, Parallel computing, Virtual machine mapping, Credit algorithm, Virtual machine scheduling
doi:10.1631/jzus.C1100217 **Document code:** A **CLC number:** TP391; TP393

1 Introduction

Virtualization abstracts the resources in computing systems, and transfers physical resources into logic resources which can be used and reused conveniently. Virtual machines are packaged and isolated by virtual machine monitors (VMM). If the quantity of the CPUs in a host is less than that of the virtual machines, the CPUs are scheduled to several virtual machines with round-robin (Chuzhoy and Naor, 2006). Virtual machine scheduling refers to selecting the virtual machines as current virtual machines to use CPUs. Thus, the nature of virtual machine scheduling is the scheduling of virtual CPUs (VCPUs) (Alessandro, 2004; Chen *et al.*, 2011a). If virtual machines have the same right to use the spaces of CPUs, a virtual machine scheduling algorithm requires us to determine the time slices for them in the execution cycle of systems (Ogata, 2002). The mainstream algorithms for virtual machine scheduling are Borrowed Virtual Time (BVT) (Duda and Cheriton, 1999), Strong Earliest Deadline First (SEDF) (Fu and Xu, 2006), Credit (Gupta *et al.*, 2006), and others (Lehoczky *et al.*, 1989; Jones *et al.*, 1997; Nieh and

[‡] Corresponding author
[*] Project (No. 2007AA010305) supported by the National High-Tech R&D Program (863) of China

Lam, 1997). These algorithms are often called on by VMM to change the CPUs' utilization of virtual machines, so they can solve most of the problems of resource allocation for virtual machines when facing different requirements (Pfoh *et al.*, 2009). But when the systems encounter tasks with parallel computing, these virtual machine scheduling algorithms may not run with a high efficiency because of their shortcomings in perceiving the knowledge in virtual machines. A lack of accuracy and effectiveness in determining the progress of tasks makes them unable to solve such problems (Chen *et al.*, 2011b).

In a virtualized cluster, large and complex tasks are generally divided into subtasks and mapped into several virtual machines to carry out parallel computing. In many parallel computing applications, one kind of subtask is concerned with the workflow parallel application (Chen *et al.*, 2011b). The input of a workflow task is normally an abstract workflow model. For the characteristics of the workflow parallel application, the workflows are usually described as a directed acyclic graph (DAG). In DAG, a task that does not have any parent task is called an entry task, and one that has no child task is called an exit task. Three typical workflow software applications are Montage (for astronomy applications), Broadband (for seismology applications), and Epigenome (for bioinformatics applications) (Juve *et al.*, 2009). Montage creates science-grade astronomical image mosaics using data collected from telescopes. The size of a Montage workflow depends upon the area of the sky (in square degrees) covered by the output mosaic. Broadband generates and compares seismograms from several high- and low-frequency earthquake simulation codes. Each workflow generates seismograms for several sources (scenario earthquakes) and sites (geographic locations). For each source/site combination, the workflow runs several high- and low-frequency earthquake simulations and computes intensity measures of the resulting seismograms. Epigenome maps short DNA segments collected using high-throughput gene sequencing machines to a previously constructed reference genome using MAQ software. The workflow splits several input segment files into small chunks, reformats and converts the chunks, maps the chunks to a reference genome, merges the mapped sequences into a single output map, and computes the sequence density for each location of interest in the reference genome.

Our study focuses on workflow parallel applications. In workflow parallel computing, a key problem is how to map the top tasks into the virtual machines efficiently. Also, a virtual machine scheduler is required that will consider fully the parallelism among subtasks, such as assignments, progress, and urgency. In this paper we design a mapper TPCM for assigning tasks to virtual machines and an application-driven virtual machine scheduler TPCS with a shorter completion time oriented for parallel computing task-oriented workflows. TPCM divides workflow tasks into several subtasks, among which the data flows tend to be serial and parallel. The data flows are used to determine the parallel virtual machines, and then serve as the base for mapping tasks to the virtual machines. Because the mapping from tasks to virtual machines affects the performance of systems, TPCM determines a modest quantity of virtual machines in a physical machine to maintain load balance. TPCS transmits the progress statuses of workflow applications into the Xen system, which allocates resources to CPUs based on the progress status of tasks. The TPCS scheduler ensures unified progress among tasks whatever their assignments, and keeps the tasks simultaneous to shorten the time in pending, synchronization, communication, and switching. TPCS can not only ensure high performance scheduling based on the progress of tasks, but also improve resource utilization and maintain load balance.

2 Related works

2.1 Parallel application scheduling algorithms in high performance computing

There are many task scheduling algorithms that schedule parallel applications to processors in high performance computing. In general, task scheduling is presented in two forms: static and dynamic (Boyer and Hura, 2005). In static scheduling algorithms, all information needed for scheduling, such as the structure of the parallel application, the execution times of individual tasks, and the communication costs among tasks must be known in advance. Static task scheduling takes place during the compilation time before parallel applications run. In dynamic scheduling, however, tasks are allocated to processors upon their arrival, and scheduling policies must be made at this time (Ilavarasan *et al.*, 2005; Kim *et al.*, 2005). Based on the challenges caused by the dynamicity of virtualization and the vagueness of availability requirements in the scheduling strategy of virtual data centers, some researchers have studied dynamic task scheduling with fuzzy prediction in virtualized data centers (Kong *et al.*, 2011). This is a dynamic algorithm to schedule tasks without dependence, which differs from our problem.

Static task scheduling algorithms are more suitable for high performance computing because many parallel applications have long execution time, and hence they require a high quality task scheduler to minimize the time. Also, the static scheduling time of several scientific and engineering applications is much shorter than their execution time on systems. For example, the execution times of more than 50% of the parallel applications that were run on four real parallel computing systems were between tens and thousands of minutes (Iosup *et al.*, 2006). The static scheduling times of parallel applications with diverse characteristics, scheduled using several static scheduling algorithms, are shorter than one second (Topcuoglu *et al.*, 2002). Static scheduling algorithms can be broadly classified into three main groups: heuristic, guided random, and hybrid algorithms (Topcuoglu *et al.*, 2002; Daoud and Kharma, 2011).

1. Heuristic scheduling algorithms move from one point in the search space to another, following a particular rule. Such algorithms, though efficient, search some paths in the search space and ignore others (Zhang *et al.*, 2007; Wang *et al.*, 2009; Daoud and Kharma, 2011). Heuristic scheduling algorithms can be divided into three groups: list-based, clustering, and duplication heuristics (Topcuoglu *et al.*, 2002). In list-based scheduling heuristics, each task is assigned a given priority. The tasks are inserted in a list of waiting tasks, such that tasks with higher priority are placed ahead of those with lower priorities. Three steps, task selection, processor selection, and status update, are then repeated until all the tasks in the list are scheduled. Clustering heuristics trade off inter-processor communication overhead with parallelization by allocating heavily communicating tasks to the same processor. In such heuristics, the tasks are grouped into an unlimited number of clusters (Topcuoglu *et al.*, 2002; Bansal *et al.*, 2003). Duplication algorithms start by running a clustering or list-based algorithm to create an initial schedule. This improvement in performance comes at the cost of increasing the complexity of the scheduling process.

2. Guided random scheduling algorithms mimic the principles of evolution and natural genetics to evolve near-optimal task schedules. Among the various guided random algorithms, genetic algorithms (GA) are the most widely used for the scheduling problem (Wu and Dajski, 1990; Grefenstette *et al.*, 1997; Daoud and Kharma, 2011). In attempts to obtain schedules of better quality, many well-known metaheuristics have been adopted, including Simulated Annealing (SA) (Grefenstette *et al.*, 1997; Zomaya and Teh, 2001; Baskiyar and Dickinson, 2005), Tabu Search (TS) (Radulescu and van Gemund, 2002; Topcuoglu *et al.*, 2002; Phinjaroenphan *et al.*, 2005), Artificial Immune System (AIS) (Wu and Dajski, 1990), Ant Colony Optimization (ACO) (Bansal *et al.*, 2003), Particle Swarm Optimization (PSO) (Nesmachnow *et al.*, 2010), and Variable Neighborhood Search (VNS) (Iverson *et al.*, 1999). GA usually takes more computing effort to locate the optimal solutions in the region of convergence (Topcuoglu *et al.*, 2002), owing to its lack of local search ability. On the other hand, trajectory methods, such as VNS (Sih and Lee, 1993), have shown their potential in exploiting the promising regions in the search space with high quality solutions. Nevertheless, they are still prone to premature convergence traps due to their limited exploration ability. Thus, it is natural to consider hybrid metaheuristics, also known as memetic algorithms (MA) (Grefenstette *et al.*, 1997; Iverson *et al.*, 1999), which have been applied to solve scheduling problems (Boyer and Hura, 2005).

3. Hybrid scheduling algorithms are also a main group. A hybrid scheduling algorithm combines heuristic algorithms and GA. The Genetic List Scheduling (GLS) algorithm (Grefenstette *et al.*, 1997) is an example of this class of algorithms, but it has greater complexity than other algorithms. There are also other highly efficient algorithms for the problem of task scheduling in heterogeneous distributed systems, including Dynamic Level Scheduling (DLS) (Sih and Lee, 1993), Heterogeneous Earliest Finish Time (HEFT) (Topcuoglu *et al.*, 2002), Critical Path on a Processor (CPOP) (Topcuoglu *et al.*, 2002), Mapping Heuristic (MH) (El-Rewini and Lewis, 1990), and Levelized Min Time (LMT) (Wu and Dajski, 1990). DLS and HEFT are improved heuristic scheduling

algorithms. They are two of the best existing scheduling algorithms for heterogeneous distributed systems (Topcuoglu *et al.*, 2002), and are employed as benchmark scheduling algorithms in many studies (Radulescu and van Gemund, 2002; Baskiyar and Dickinson, 2005). The DLS algorithm does not schedule tasks between two previously scheduled tasks. HEFT starts by setting the computation costs of tasks and the communication costs of edges to their mean values. Each task is assigned a value called the upward rank. In this algorithm, the upward rank of a task is the largest sum of the mean computation costs and mean communication costs along any directed path from this task to an exit task.

The above static and dynamic task scheduling algorithms focus on the allocation of CPUs to tasks, and these scheduling methods for parallel applications have already been well-studied and well-explored in the context of high performance computing. However, our research on CPU scheduling has a different scope:

1. Existing task scheduling algorithms take tasks or subtasks as the scheduling units to map to CPUs, focusing on the optimal combination and mapping of tasks to CPUs, so it is a scheduling problem in a macro field. Our CPU scheduling algorithm pays attention to dynamic resource adjustment during a period of time for some tasks or subtasks. The aim is to organize the CPU time slices with rationality. The CPU scheduling includes the determination of resource requirements, the setting of the interrupt cycle, and the selection of diversity scheduling algorithms, so it is a scheduling problem in a micro field.

2. Existing task scheduling algorithms concentrate on task scheduling in a non-virtualization environment that has only two layers, i.e., from CPUs to tasks. But our work focuses on CPU scheduling in a virtualization environment, and we solve the allocation of CPUs to virtual machines, not the tasks directly. Because of the increasing complexity of hierarchies in virtualized systems, the algorithm we propose is different from task scheduling algorithms.

2.2 CPU scheduling algorithms in virtualization computing environments

The mainstream VCPU scheduling algorithms are as follows: (1) BVT algorithm (Duda and Cheriton, 1999): BVT sets a weight to each of the domains in a system to allocate the time slices of CPUs in proportion, and the allocation is applied to the occasion oriented real-time demand. (2) SEDF algorithm (Fu and Xu, 2006): SEDF also allocates the time slices of CPUs in proportion, but in this case a domain cannot occupy all the CPU resources at one time, and we can reserve a portion of them for services in other domains. SEDF is applied to occasions with the demand of real-time. (3) Credit algorithm (Gupta *et al.*, 2006): Credit is designed for SMP hosts and each of their CPUs manages a local queue of VCPUs. Each VCPU in this queue has one of two priorities: over or under.

For real-time CPU scheduling algorithms, Earliest Eligible Virtual Deadline First (EEVDF) focuses on real-time tasks and is also classified as a proportional share real-time algorithm (Lehoczky *et al.*, 1989). SMART (Nieh and Lam, 1997) dynamically integrates a real-time scheduler and a conventional scheduler depending upon priorities and admission control. Resource reservations and a precomputed scheduling graph are used for scheduling real-time applications in the Rialto operating system (Jones *et al.*, 1997). BERT effectively schedules multimedia and best-effort jobs, but its implementation depends on a prediction mechanism that is tied to the Scout operating system (Bavier *et al.*, 1999).

For non-real-time CPU scheduling algorithms, studies have been focused on first-come first-served (FCFS), Shortest Job First (SJF), and PRIORITY (Shi *et al.*, 2007), in which the FCFS policy is a non-preemptive scheduling discipline that schedules the tasks in order of their arrival in the waiting queue. The earliest arriving task has the highest priority, so the ready task will get the CPU time slices until this process completes the task or the task is interrupted (Rawat and Rajamani, 2009). Many techniques have been imagined to efficiently manage threads, including the Work Stealing (WS) and the Parallel Depth First (PDF) techniques (Chen *et al.*, 2007). PDF approaches have been proposed for cache sharing as well as for WS, a popular scheduling technique that takes a more traditional approach. The WS policy maintains a work queue for each processor. When forking a new thread, the new thread is put on the top of the local thread. The Critical Path Method, CPM, is yet another method for scheduling threads. This approach tries to shorten the longest path in the application graph by removing communication requirements and mapping the adjacent tasks into a cluster (Cerin *et al.*, 2008).

A hierarchical CPU scheduler addresses this problem by statically partitioning the CPU bandwidth among various application classes. Hierarchical scheduling is a scheduling framework that enables the grouping together of threads, processes, and applications into service classes (Chandra and Shenoy, 2008). CPU bandwidth is then allocated to these classes based on the collective requirement of their constituent entities. In a hierarchical scheduling framework, the total system CPU bandwidth is divided proportionately among various service classes. Proportional-share scheduling algorithms (Nieh and Lam, 1997; Caprita *et al.*, 2005) are a class of scheduling algorithms that meet this criterion. Another requirement for hierarchical scheduling is that the scheduler should be insensitive to fluctuating CPU bandwidth available to it. A proportional-share scheduling algorithm, such as Start-Time Fair Queuing (SFQ) (Goyal *et al.*, 1996), has been shown to meet all these requirements in unprocessed environments and has been deployed in a hierarchical scheduling environment. However, SFQ can result in unbounded unfairness and starvation when employed in multiprocessor environments.

Schedulers that deal with performance issues can be classified as driven by deadlines or by proportionally sharing resources (Rau and Smirni, 1999). Deadline driven schedulers, such as Earliest-Deadline First (EDF) and Rate Monotonic (RM), are optimal under light load conditions but are not well suited to support best-effort applications. Proportional share schedulers try to allocate CPU resources to applications in proportion to their shares (Jin *et al.*, 2005). The idea was first presented for network packet scheduling as Weighted Fair Queuing, and later applied to processor scheduling as stride scheduling. From then on, many variants such as Virtual Clock, SFQ, SFS, FFQ, SPFQ, GRRR, and time-shift FQ have been proposed. GR3 (Chan and Nieh, 2003) also falls into this group, and provides a more accurate share with a low scheduling overhead. As proportional share scheduling is based on predefined shares of applications, it faces the challenge of setting reasonable shares for a set of tasks with dynamically changing resource requirements. Lottery scheduling (Waldspurger and Weihl, 1994) also proposes hierarchical allocation of resources based on the notion of tickets and lotteries. Lottery scheduling itself is a randomized algorithm that can meet resource requirements in a probabilistic manner, and extending it to multiprocessor environments is nontrivial.

There are currently some VCPU schedulers in the Quest operating system. The HARTIK kernel (Ghazalie and Baker, 1995) supports the co-existence of both soft and hard real-time tasks. To ensure temporal isolation between hard and soft real-time tasks, the soft real-time tasks are serviced using a constant bandwidth server (CBS). CBS guarantees a total utilization factor no greater than Qs/Ts, even in overload, by specifying a maximum budget in a designated window of time. CBS has bandwidth preservation properties similar to those of the dynamic sporadic server (DSS) (Abeni and Buttazzo, 1998), but with better responsiveness. TBS and DSS both assume the existence of server deadlines. We chose to assume the existence of deadlines for VCPUs in Quest, restricting VCPUs to fixed priorities. This avoided the added complexity of managing the dynamic priorities of VCPUs as their deadlines change. Also, for cases in which there are multiple tasks sharing a fixed-priority VCPU, the execution of one task will not change the importance of the VCPU for the other tasks (Govindan *et al.*, 2009; Danish *et al.*, 2011).

In addition to these mainstream CPU scheduling algorithms, a scheduling methodology based on the priority in accordance with I/O status has been presented (Aspnes *et al.*, 1997). The I/O performance of a domain scheduling algorithm in the Xen hypervisor with more emphasis on resource exchange in VMM was discussed by Govil *et al.* (2000). In studies of resource allocation over multiple virtual machines, the traditional CPU scheduling algorithms can allocate resources to processes with fairness. But in virtualization environments, the scheduler should adopt other flexible scheduling policies (Volkmar *et al.*, 2004), such as that of guest OS, to avoid preemptive blocking (Rosenblum and Garfinkel, 2005). Three CPU scheduling algorithms in Xen were evaluated by Hiroshi and Kenji (2007) by analyzing the effects of parameter settings. Dynamic configurations of CPUs are also a vital research area. Dynamic configuration policies orienting the virtual machines' status (Pfoh *et* *al.*, 2009) and applied requirements (Fumio, 2009) have been presented. These methodologies have two characteristics: first, they stress the effect that the I/O has on real time scheduling. Second, the open-source virtualization software is used to perform experiments to test I/O overheads and CPU multithreading abilities in the file, Web, and high performance computing servers.

There have been fewer studies on application-driven virtual machine schedulers oriented for parallel computing in clusters. Some researchers have presented scheduling algorithms with sense perception (Cota-Robles and Flautner, 2008). The virtual machine scheduling mechanism with sense perception was used to infer the I/O roundedness of user-level tasks combined with an event in I/O binding tasks. A scheduling algorithm based on the priority of tasks was designed to ensure the fairness of CPU for dynamic requirements of applications in symmetrical multi-processing (SMP) hosts (Shi *et al.*, 2009). Some scholars proposed that a modified VMM can perceive an implicit guest OS by inferring the information of guest OS (Laslo *et al.*, 2008). A CPU scheduling algorithm for communication perception with a better cost and performance was presented by Huai *et al.* (2007). Researchers have implemented sense perception to task load, but they have not solved the problem of allocating CPU resource based on the progress of tasks. In general, virtual machine monitors tend to lack the knowledge in virtual machines, and virtual machines lack knowledge of tasks in the workload, especially their progress (percentage of completed task assignment and percentage of remaining task assignment). Therefore, unexpected assignments make it difficult to allocate resources accurately according to the progress status of tasks. When faced with virtual machines with collaborative computing tasks, it is impossible to satisfy these requirements. In our study we aim to solve these problems in a virtualized parallel computing environment.

3 TPCM: a mapper for assigning tasks to virtual machines

In this section, we present a mapper, TPCM, to deploy tasks and virtual machines. The TPCM mapper is a tool which makes a plan for mapping tasks using the virtual machines mapping algorithm (VMMA). In this plan, the quantity of physical machines and the quantity of virtual machines in each physical machine are determined. After this work has been completed, TPCM collects the information from physical machines, and creates virtual machines based on such a plan. Then, the tasks are mapped into the virtual machines. Although the creation of virtual machines is part of the work of task mapping, it is not dealt with in VMMA. Therefore, it is not considered in the time measuring of VMMA, but it is considered in the time measuring of task mapping for TPCM. The mapper requires the inputting of the set of tasks and their relationships. First, we describe a method oriented for parallel computing tasks based on the workflow.

3.1 Description of tasks

In parallel computing, the division of tasks produces a set of subtasks with parallel and serial structures. We call the set of subtasks the taskSet. Thus, each pair of subtasks in the taskSet would be one of the seven relationships whose data flows are shown in Fig. 1.

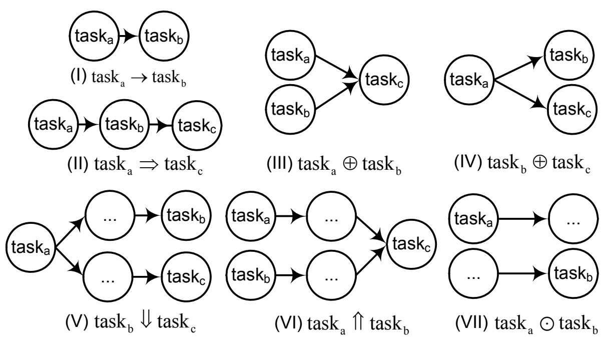

$task_a$
$task_b$
$task_c$
...
(I) $task_a \rightarrow task_b$
(II) $task_a \Rightarrow task_c$
(III) $task_a \oplus task_b$
(IV) $task_b \oplus task_c$
(V) $task_b \Downarrow task_c$
(VI) $task_a \Uparrow task_b$
(VII) $task_a \odot task_b$

Fig. 1 The relationships among tasks

1. Serial relation

The serial relation has two instances, the first shown as I in Fig. 1: for $task_a$, $task_b \in taskSet$, $task_b$ can begin only after $task_a$ is completed. That is, if a data flow from $task_a$ to $task_b$ exists, then it is a strict serial relation between $task_a$ and $task_b$, denoted by $task_a \rightarrow task_b$. The second instance, shown as II in Fig. 1, is: for $task_a$, $task_b$, $task_c \in taskSet$, if $task_a \rightarrow task_b \wedge task_b \rightarrow task_c$, then it is a loose serial relation between $task_a$ and $task_c$, denoted by $task_a \Rightarrow task_c$.

2. Parallel relation

The parallel relation has two instances, the first shown as III in Fig. 1: for task$_a$, task$_b$, task$_c$$\in$taskSet, only after task$_a$ and task$_b$ are both completed, can task$_c$ begin. That is, if task$_a$$\rightarrow$task$_c$$\wedge$task$_b$$\rightarrow$task$_c$, then it is a frontier strict parallel relation between task$_a$ and task$_b$, denoted by task$_a$$\oplus$ task$_b$. The second instance is shown as IV in Fig. 1: for task$_a$, task$_b$, task$_c$$\in$taskSet, task$_b$ and task$_c$ can begin only after task$_a$ is completed. That is, if task$_a$$\rightarrow$task$_b$$\wedge$task$_a$$\rightarrow$task$_c$, then it is a latter strict parallel relation between task$_b$ and task$_c$, denoted by task$_b$$\oplus$ task$_c$.

3. Indirect relation

The indirect relation has two instances. The first is shown as V in Fig. 1: for task$_a$, task$_b$, task$_c$$\in$taskSet, if task$_a$$\Rightarrow$task$_b$$\wedge$task$_a$$\Rightarrow$task$_c$, then it is the latter indirect relation between task$_b$ and task$_c$, denoted by task$_b$$\Downarrow$task$_c$. The second instance is shown as VI in Fig. 1: for task$_a$, task$_b$, task$_c$$\in$taskSet, if task$_a$$\Rightarrow$task$_c$$\wedge$ task$_b$$\Rightarrow$task$_c$, then it is a frontier indirect relation between task$_a$ and task$_b$, denoted by task$_a$$\Uparrow$task$_b$.

4. Connectionless relation

In Fig.1 (VII), for task$_a$, task$_b$$\in$taskSet, if there are no serial, parallel, or indirect relations between task$_a$ and task$_b$, then there is a connectionless relation because no data flow exists between any pair of tasks, denoted by task$_a$$\odot$ task$_b$.

The task information can be created manually or automatically. Some studies show that automatic decomposition might lead to irrational task sets, so we suggest manual decomposition. After we construct the relationship between tasks, the task information is enveloped as a data structure taskSet and inputted into virtual computing systems. The mapper TEVM in virtual computing systems completes the mapping from tasks to virtual machines.

3.2 The algorithm for mapping tasks to virtual machines

In general, there are two methods to make parallel computing for tasks with the same configuration requirement. In Method 1, the tasks are placed into a virtual machine and then run to make parallel computing as processes or threads (Katz *et al*., 2005). In Method 2, the tasks are placed into multiple virtual machines and then run to make parallel computing as multiple virtual machines (Bansal *et al*., 2003; Qi *et al*., 2006). Method 2 has at least two advantages:

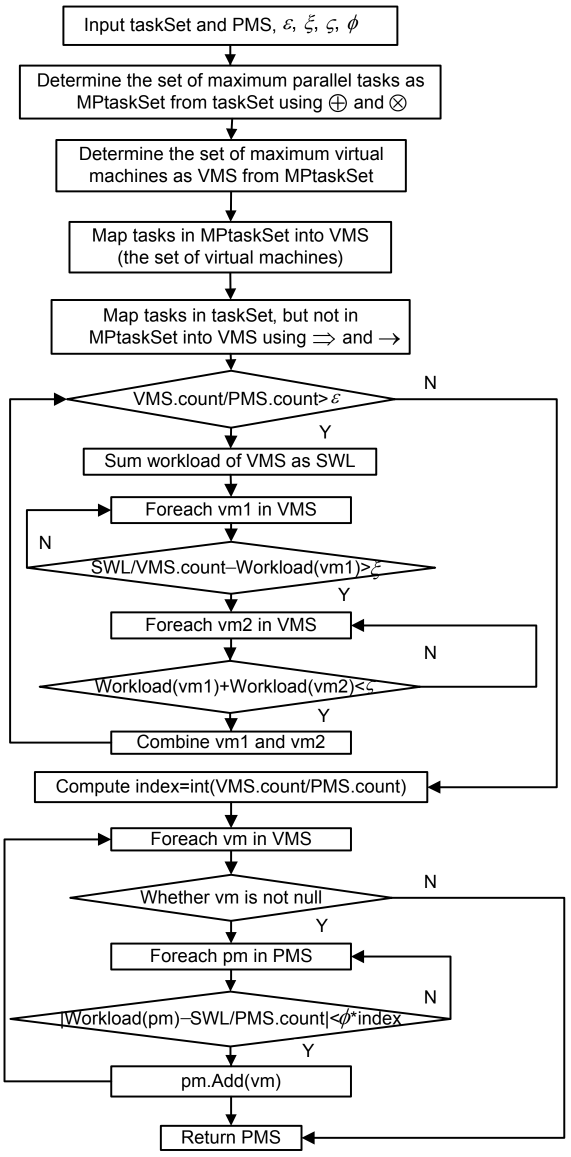
1. In a multiple virtual machine environment, a task occupies an operation system. Because of the isolation characteristics among virtual machines, there are fewer pending, deadlock, and synchronization phenomena than in multiple processes or threads.

2. Processes or threads are found in many data interactions among applications and operation systems, such as library calling in kernels, hardware reading and writing, and network transmission, not only in their own execution. The multiple virtual machines refer to multiple applications interacting with multiple guest OS. In the condition of power computing resources, the speed of this method is faster than that of a single operation interacting with many applications in a virtual machine.

As existing machines generally contain multiple CPUs and a CPU contains multiple cores, task collaboration over multiple virtual machines has the two advantages described above. Therefore, we adopt Method 2.

In our algorithm design, the quantity of virtual machines is determined by the relationships among subtasks in taskSet to mine the parallelism of subtasks and keep the assignments of tasks consistent with the resources allocated (Hamidzadeh *et al*., 2000). The mapping of tasks to virtual machines determines the performances of systems and affects the total running time of tasks. Also, we should maintain the load balance among virtual machines (Cherkasova *et al*., 2007) by considering the density of data flows in communication and synchronization (James and Ravi, 2007; Kim and Lim, 2009). Furthermore, the deployment of virtual machines to physical machines should be based on the load status of the virtual machines.

The key for mapping tasks to virtual machines is to estimate the workload of the tasks. In this section, the workload is decided by the quantity of codes from tasks and the communication with other tasks, and is ultimately determined by the estimated execution time in a certain hardware configuration. There are several techniques to estimate such information (Wu and Dajski, 1990). The distributed system is represented by a set P of m processors that have diverse capabilities. The $n\times m$ computation cost matrix $\boldsymbol{C}$ stores the execution costs of tasks. Each element $c_{i,j}\in\boldsymbol{C}$ represents the estimated execution time of task t_i on processor p_j. Precise calculation of the running time of the tasks on the processors is unfeasible before running the application (Phinjaroenphan *et al.*, 2005). One approach to estimating the execution time of task t_i on processor p_j is to use profiling information of t_i and p_j (Dail *et al.*, 2002; Zhang *et al.*, 2004). Another approach is to analyze past observations of the running times of similar tasks on p_j (Iverson *et al.*, 1999; Govindan *et al.*, 2007). Having determined the workload, the algorithm implementing the mapping from tasks to virtual machines (VMMA) is shown in Fig. 2.



Input taskSet and PMS, ε, ξ, ς, ϕ
Determine the set of maximum parallel tasks as MPtaskSet from taskSet using ⊕ and ⊗
Determine the set of maximum virtual machines as VMS from MPtaskSet
Map tasks in MPtaskSet into VMS (the set of virtual machines)
Map tasks in taskSet, but not in MPtaskSet into VMS using ⇒ and →
VMS.count/PMS.count>ε
N
Y
Sum workload of VMS as SWL
Foreach vm1 in VMS
N
SWL/VMS.count−Workload(vm1)>ξ
Y
Foreach vm2 in VMS
N
Workload(vm1)+Workload(vm2)<ς
Y
Combine vm1 and vm2
Compute index=int(VMS.count/PMS.count)
Foreach vm in VMS
Whether vm is not null
N
Y
Foreach pm in PMS
N
|Workload(pm)−SWL/PMS.count|<ϕ*index
Y
pm.Add(vm)
Return PMS

Fig. 2 Virtual machine mapping algorithm (VMMA)

The input parameters of VMMA are taskSet, PMS, ε, ξ, ς, and Φ. Besides taskSet, as defined above, PMS is a set of physical machines in a cluster, ε is the maximum quantity of virtual machines deployed in a physical machine, and ξ, ς, Φ are three parameters describing the boundary of load, where ξ is the minimum deviation among virtual machines, ς is the maximum load that a virtual machine can afford, and Φ is the maximum load that a physical machine can afford. To process the data easily, the assignments of tasks are normalized to (0, 1] to measure their load, so $\xi, \varsigma, \Phi \in (0, 1]$. In algorithm VMMA, the set of the maximum parallel quantity of subtasks MPtaskSet is collected from taskSet to create a virtual machine array VMS with the length of MPtaskSet as its initial length. The subtasks in MPtaskSet are mapped into the elements of VMS one by one, and then the other unmapped tasks in taskSet are mapped into VMS based on the serial relation among tasks. We decide whether the virtual machines need to be combined according to the relationship between the quantity of virtual machines and the quantity of physical machines. VMS.count/PMS.count>ε shows that too many virtual machines deployed in a physical machine would lead to inconvenience in their deployment, so the virtual machines are required to be combined. The function workload() computes the load of tasks deployed. If the load of a task in a virtual machine is less than ς, this virtual machine should be combined with other virtual machines. In the last part, the algorithm deploys the virtual machines into physical machines. The average number of virtual machines deployed in a physical machine is computed as index, and then Φ is taken as the reference to deploy the virtual machines with a balanced load.

The output of the VMMA algorithm is PMS, a set of physical machines in a cluster. We define the elements in PMS as PM=(VMs[]), where VMs is the set of virtual machines deployed in this physical machine, VMs=VM[]. The element in VMs is VM=(tasks[], PM), in which tasks[] is a set of tasks in this virtual machine, and PM is the physical machine that VM hosts. Based on the construction of elements in PMS, taskSet is allocated to virtual machines and the virtual machines are deployed to physical machines.

4 TPCS: an application-driven virtual machine scheduler

On the basis of tasks mapped, we propose a virtual machine scheduler which can perceive the progress of tasks. When the system performs parallel computing, the scheduler can use the progress of tasks to decide the CPU allocation of virtual machines.

In a system virtualization environment, a highly efficient scheduling algorithm can improve not only hardware usage, but also the efficiency of parallel computing tasks in multiple virtual machines. In the Xen hypervisor, the Xen kernel neither knows the workload of tasks in guest OS, nor perceives the progress of tasks packaged in virtual machines. Thus, it cannot make an efficient and accurate scheduling based on the status of tasks in virtual machines and is unable to reduce the time spent on pending, synchronization, communication, and switching among concurrent tasks. We propose an improvement to the Credit algorithm in Xen in which the progress statuses of applications are transmitted to the Xen kernel in an active way so that the scheduling algorithm can allocate the CPU resource to virtual machines with optimization according to the progress status.

We take Xen hypervisor 3.4.2 as the virtualized platform to achieve the goal of perceiving the progress of tasks in virtual machines. The tasks placed in virtual machines are allocated CPU resources based on their present statuses. TPCS, an application-driven virtual machine scheduler, integrates the virtual machines into a coherent whole. TPCS has three components: a middleware supporting parallel computing, a device driver, and a virtual machine scheduling algorithm (Fig. 3). To implement the application-ware in the Xen hypervisor, TPCS is designed from the top down using a hierarchical approach. Take X86 architecture as an example. Three components run at different CPU right levels: a virtual machine scheduling algorithm runs in CPU-ring0 in the Xen hypervisor, a device driver runs in CPU-ring 1 of the VM guest OS kernel, and a middleware runs in CPU-ring 3 of the application layer.

In Fig. 3, middleware calls the interface provided by the guest OS to transmit the percentage of completed task information to the guest OS, and then to the CPU virtualization subsystem of the Xen hypervisor. The Xen hypervisor analyzes the percentage of completed task information to decide the virtual machine scheduling policy. The scheduling is based on the amount of uncompleted assignments and the urgency of some tasks. Xen allocates or adjusts the CPU time slices for virtual machines to change their resource utilization, thus changing the speed of tasks and keeping the concurrent tasks in a simultaneous status. The synchronization of tasks can reduce the time spent in pending, communication, and switching.

The design components of TPCS are described in detail in the following subsection.

4.1 Middleware supporting application-driven scheduling

In existing virtual machine scheduling algorithms, CPU schedulers implement scheduling based on a static policy, not on the status of current tasks. Some researchers propose that the system software can perceive the workload of the application. For example, performance counters are known to be able to extract the characteristics of application behaviors at lower software layers, enabling estimation of the workload of the application. However, the system software cannot perceive the progress of an application because the workload has no relationship with the progress of the application. The percentage of completed tasks can be known only by the tasks themselves. As a result, we adopt a notification approach, application-driven scheduling, in which the top module in TPCS reports the progress of tasks to the underlying module in an active way.

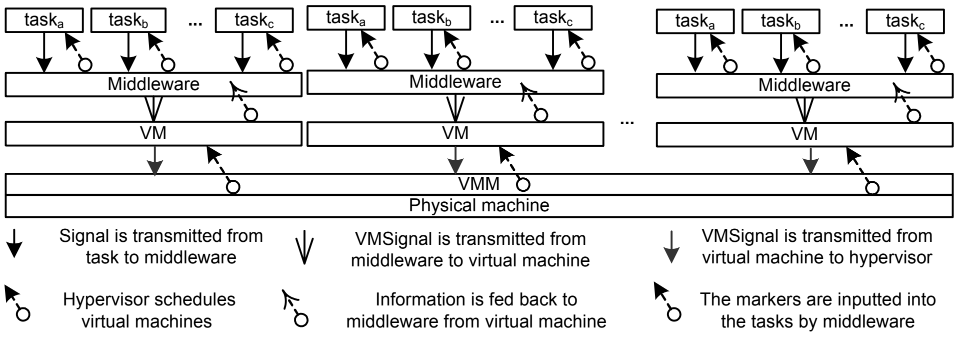

task$_a$
task$_b$
...
task$_c$
Middleware
VM
...
VMM
Physical machine
Signal is transmitted from task to middleware
VMSignal is transmitted from middleware to virtual machine
VMSignal is transmitted from virtual machine to hypervisor
Hypervisor schedules virtual machines
Information is fed back to middleware from virtual machine
The markers are inputted into the tasks by middleware

Fig. 3 The architecture of the TPCS scheduler

We have developed a middleware supporting application-driven scheduling in guest OS to perceive the progress of an application. The middleware has three functions:

1. Inputting progress markers into threads of tasks

The middleware inputs the progress markers of scheduling information into the threads of tasks to obtain the progress status from the threads of tasks at some key time points. Progress markers are collected via signals. Each signal is a tuple with two elements, signal=(taskid, mark), in which taskid, the identifier of a subtask, can be taken to compute the present location of parallel computing, and mark is labeled as the progress of this subtask. The signals are constructed in subtasks scanned by middleware periodically.

2. Progress information management for tasks

The middleware, as a daemon in guest OS with the task information, runs when its hosted virtual machine starts. After the tasks in taskSet are mapped into the virtual machines, the middleware describes the task information of taskSet as a tuple with two elements, Taskinfo=(tasks, relationship), in which, tasks is the set of subtasks in parallel computing, and relationship reflects the relation among these subtasks. For all task∈tasks, task=(taskID, workload, Markquantity, dynamicinfo), taskid is the identifier of a subtask, workload is an estimation of the assignments for this subtask, and Markquantity is the quantity of markers, that is the number of times of inputting progress marks. Dynamicinfo, a group of dynamic data for subtasks, can be described as dynamicinfo=(loadtime, previoustime, finishworkload, state), where loadtime is the time loading the subtask, previoustime is the time receiving the last signal, and finishworkload is the assignment completed in the last signal, and state describes the status of subtasks, such as preparing, completed, or running state.

3. Transmitting the progress signals to guest OS

To implement the signal transmission of progress marks, we define two concepts for progress status: first, the percentage of tasks completed denoted by Rate, and second, the time to complete a unit task denoted by TimeRate.

Definition 1 (Rate) The proportion of tasks completed, Rate, is the ratio of the number of completed assignments to the number of all assignments for tasks. The Rate is divided into the rate for the current task and the rate for the concurrent task set, denoted by thisRate and allRate, respectively. The current task refers to the recent task scanned by the middleware, and the concurrent task set refers to the set of tasks run concurrently with the current task in taskSet. In our design, allRate can be computed according to the properties of finishworkload in task.Dynamicinfo and the workload of task in Taskinfo, but thisRate needs only the location of the progress mark for the current task:

$$\text{thisRate} = \text{Currenttask.workload} \times \text{signal.mark}/ \\ \text{Currenttask.Markquantity}. \tag{1}$$

Definition 2 (TimeRate) The time to complete a unit task, TimeRate, is the ratio of the time to complete assignments for each task. It is also divided into the TimeRate for the current task and the TimeRate for the concurrent task set, denoted by thisTimeRate and allTimeRate, respectively.

In determining the task progress, we compare allTimeRate with thisTimeRate to determine the speed of the current task relative to the concurrent task set, thereby deciding the urgency of the current task in the progress of the synchronization process, and providing a reference for the Xen system to schedule the virtual machines.

The middleware scans the statuses of tasks periodically, and if it does not receive a signal for a long time, it judges whether a deadlock exists in the system (Govindan *et al.*, 2007). Otherwise, the signal is collected and processed by algorithm VSTA (Fig. 4).

The time points to execute VSTA are as follows: (1) the time before the running of the first subtask, (2) the time after the running of the last subtask, (3) the time before the running of each subtask, (4) the time after the running of each subtask, and (5) the period of time in the interval between the running of subtasks.

The input values of the VSTA algorithm are Taskinfo and signal, where Taskinfo is created automatically after the tasks are mapped into virtual machines in TPCM. The signal is obtained via the communication among threads when the middleware scans threads of tasks. The output value of this algorithm is a tuple VMsignal=(leftWorkload, allTimeRate, thisLeftWorkload, thisTimeRate), in which leftWorkload is the average uncompleted assignment for all tasks paralleling with the current task, allTimeRate

```
Input: Taskinfo, signal
Output: VMsignal
1  Currenttask←getTaskbySignaltaskID(Taskinfo,
     Signal.taskID);
2  Define coTaskset as the set of tasks paralleling
     with Currenttask;
3  coTaskset.add(Currenttask);
4  Foreach (task in Taskinfo.tasks)
5    If (Currenttask⊕task or Currenttask⊗task)
6       coTaskset.add(task);
7    Endif
8  Endfor
9  saveDynamicInfo(Currenttask);
   // for use in next scanning time
10 finishtime←0;
11 allworkload←0;
12 finishworkload←0;
13 Foreach (task in coTaskset) {
14      allworkload+←task.workload;
15      finishworkload+←task.dynamicinfo.
                            finishworkload;
16      finishtime+←sytemtime–task.dynamicinfo.
                      loadtime;
17 Endfor
18 leftWorkload←allworkload–finishworkload;
19 thisLeftworkload←Currenttask.workload–
        Currenttask.Dynamicinfo.finishworkload;
20 allRate←finishworkload/allworkload;
21 thisRate←Currenttask.workload×signal.mark/
              Currenttask.markquantity;
22 allTimeRate←finishtime/allRate;
23 thisTimeRate←(systemtime–Currenttask.
      dynamicinfo.loadtime)/thisRate;
24 leftWorkload←leftWorkload/coTaskset.count;
25 VMsignal←VMsignal(leftWorkload, allTimeRate,
      thisLeftworkload, thisTimeRate);
```

Fig. 4 Virtual machine signal transmitting algorithm (VSTA)

is the time rate of all completed tasks in the whole process, thisLeftWorkload is the uncompleted assignments for the current task, and thisTimeRate is the time rate of a completed assignment for the current task in recent time.

Besides the two parameters of timeRate and thisTimeRate (which reflect the synchronization requirement based on the urgency of tasks discussed above), the output of algorithm VSTA includes two other parameters, thisLeftWorkload and leftWorkload, which are the remaining assignments of the current task and the average remaining assignments of the concurrent tasks set, respectively. The value of thisLeftWorkload is the total assignment minus completed assignments for the current task. The leftWorkload is the mean value of thisLeftWorkload for all tasks in concurrent status. These two parameters reflect the general conditions based on the uncompleted assignments. We compute these values, because the relativities between thisLeftWorkload and leftWorkload provide another reference for the Xen system scheduling the virtual machines.

The essential points of VSTA: based on collecting the concurrent task set for the current task, the progress of the current task is compared to that of the concurrent task set. VSTA computes the total assignments of tasks in the concurrent task set, completed assignments of tasks, and the time consumption, to enable calculation of the remaining assignment of tasks (thisLeftWorkload and leftWorkload), the percentage of assignments completed (thisRate and allRate), and the completion time of a unit task (allTimeRate and thisTimeRate). They are packaged into a VMsignal transmitting into the guest OS.

In row 1 of VSTA (Fig. 4), the function getTaskbySignaltaskID gets the current task named Currenttask from taskid. In rows 2–8, the concurrent subtasks are identified by traversing the tasks in Taskinfo and are named coTaskset from the current task (Fig. 5). In row 9, the function saveDynamicInfo saves the dynamicinfo of the current task. In rows 10–17, coTaskset is taken to compute the time consumption of the whole process. In rows 18–21, the completed and uncompleted assignments are computed, and then, in rows 22–23, allTimeRate and thisTimeRate are determined. In row 24, the average remaining assignments of coTaskset is computed. In row 25, VMsignal is constructed and the system-call provided by guest OS transmits the VMsignal into the kernel of guest OS, and then passes it to the Xen hypervisor.

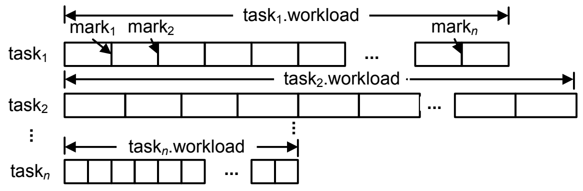


Fig. 5 coTaskset in the concurrent state of a virtual machine

For several tasks, $task_1$, $task_2$, ..., $task_n$, we can input $mark_1$, $mark_2$, …, $mark_n$ because of their different workloads (Fig. 5). Thus, when the middleware scans the tasks, we can determine the progress of the current task via the mark of signal. Therefore, VSTA implements the perception of task progress.

4.2 Device driver in the guest OS kernel

Virtual machine scheduling is carried out by a CPU virtualization subsystem in the Xen hypervisor based on progress signals. VMsignal must be transmitted to the CPU virtualization subsystem through hyper-call, because hyper-call is the only way to switch to the Xen kernel from top layers. But middleware in CPU-ring 3 is not entitled to request the hyper-call in X86 architecture. We cannot submit VMsignal into the Xen hypervisor in CPU-ring 0 directly, but we can use the hyper-call indirectly. In our design, VMsignal is transmitted into guest OS in CPU-ring 1 first, and then a hyper-call is called in guest OS to switch to the Xen hypervisor in CPU-ring 0.

We have designed a special device driver named vmscmd in the guest OS kernel to complete this work in the same way as a device driver in a Linux operation system. We complete this operation with the help of system-call ioctl(), and the service routine of ioctl() in the Linux kernel is sys_ioctl() provided by file_operation in vmscmd. The function vmscmd_ioctl is completed as shown in Fig. 6.

```
Function vmscmd_ioctl(*inode ,*file, cmd, data)
 1    { switch(cmd){
 2      case IOCTL_VMSCMD_VMscheduling:{
 3       vmscmd_hypercall hypercall;
 4         If (copy_from_user(&hypercall, data,
              sizeof(hypercall)))
 5          return –EFAULT;
 6          _asm__volatile_(
 7          “pushl %%ebx; pushl %%ecx; pushl %%edx;”
 8          “Movl 8(%%eax), %%ebx;”
 9          “Movl 16(%%eax), %%ecx;”
10          “Movl 24(%%eax), %%edx;”
11          “Movl (%%eax), %%eax;”
12          “shll $5, %%eax;”
13          “addl $hypercall_page, %%eax;”
14          “call *%%eax;”
15          “popl %%ebx; popl %%ecx; popl %%edx;”
16          : “=a” (ret): “0”(&hypercall): “memory”;
17          )
18          break; }
19          … }
20    }
```

Fig. 6 Function: vmscmd_ioctl

The function vmscmd_ioctl is the pointer of ioctl() in file_operation. The input parameters of vmscmd_ioctl are *inode, *file, cmd, and data, which have been determined by VZlinux 2.6.48 source files.

The essential points of vmscmd_ioctl: We define a new command code in ioctl, IOCTL_VMSCMD_VMscheduling. In this code, the function copy_from_user() obtains VMsignal from middleware and saves it into a variable hypercall. The data type of hypercall is a struct of vmscmd_hypercall including two elements: op, the command line of hyper-call, and arg[4], four parameters of hyper-call.

In rows 1–20, a new branch IOCTL_VMSCMD_VMscheduling is added to the function of hyper-call _HYPERVISOR_VCPU_OP, a hyper-call designed for a CPU virtualization subsystem. In this new branch, the function copy_from_guest() is called to transmit the value of hypercall into the CPU virtualization subsystem of the Xen hypervisor, as the input of the virtual machine scheduling algorithm discussed in Section 4.3. In row 14, vmscmd completes the work of calling the hyper-call through a jump instruction ‘call *%%eax’.

We implement vmscmd in the guest OS kernel via system-call based on VZlinux 2.6.48 for Xen 3.3. In our design, vmscmd is loaded by the function module_init() after the guest OS kernel starts.

4.3 Virtual machine scheduling algorithm

4.3.1 Algorithm design

The virtual machine scheduling algorithm first receives VMsignal from guest OS, and then allocates the CPUs based on task progress information. Related work shows that, if virtual machines have the same right to use the space of CPUs, CPUs would be allocated via time slices. A virtual machine scheduling algorithm requires the determination of the time slices of CPUs based on round-robin to decide the credit values of VCPUs in the next execution cycle. The execution cycle, as a fixed value in systems and named Δt in our study, is the time that the Xen hypervisor takes to traverse all virtual machines.

Here we discuss mainly the determination of the time slices. When a CPU clock interrupt occurs in systems, the CPU scheduler decides the time slices of virtual machines in the next execution cycle. If the time slices do not need to change, the virtual machines are scheduled in the next execution cycle according to the current distribution of time slices. Otherwise, the virtual machine scheduling algorithm (VSA) (Fig. 7) is called immediately. The CPU virtualization subsystem summarizes and stores the VMsignal into a special data structure to determine the time slices in the next execution cycle. We define VMinfo as this structure for managing the static and dynamic information for all currently active virtual machines, as the base of time slices for Xen in different execution cycles.

VMinfo is a tuple with three elements, VMinfo=(vms, lwl, tr), where vms is the set of virtual machines in the Xen hypervisor, lwl is the maximum limited load of this system, and tr is the maximum assignment allowed in this system. For all vm$\in$vms, vm={signal, schedulerate, timeslice}, signal is the structure used to save VMsignals received from guest OS, schedulerate describes the proportion of time slices in the Xen hypervisor (schedulerate$\in$[0, 1]), and timeslice is the proportion of the time slices for virtual machine vm in the next execution cycle.

Timeslices, as an element of VMinfo, is refreshed by VSA continuously. The latest value is taken as the basis to calculate the credits for VCPUs in virtual machines. In the Credit algorithm of the Xen hypervisor, the credit value of a VCPU decides the occupied frequency that this VCPU takes to the CPU. The CPU virtualization subsystem implements dynamic scheduling based on the progress of tasks in virtual machines by refreshing the values of VMinfo and credit.

We now discuss VSA in detail. The input parameters of VSA are the execution cycle Δt and VMinfo. The output of VSA is virtual machine scheduling information named VMinfo′ in the next execution cycle.

The essential points of VSA: VSA analyzes VMinfo to decide the virtual machine scheduling policy. There are two policies to calculate the time slices. Policy 1 is virtual machine scheduling in general conditions based on the uncompleted assignments. Policy 2 is virtual machine scheduling with a synchronization requirement based on the urgency of tasks. TimeRate represents the time to complete a unit task, which can embody the total progress of concurrent tasks in virtual machines. Thus, it can decide the virtual machine scheduling policy that Xen adopts. The policy is determined by the deviation between the

```
Input: $\Delta t$, VMinfo
Output: VMinfo′
 1  currentvm←VMinfo.vms.currentvm;
    // obtain current virtual machines
 2  Policy←1;  // the default value is virtual machine
               // scheduling based on the urgency of tasks
 3  Foreach (vm in VMinfo.vms)
 4    If (abs(vm.signal.timerate−getAvgTasksTimeRates
        (VMinfo))>VMinfo.tr)
 5      Policy←2;  // virtual machine scheduling
                   // based on the assignment uncompleted
 6    Endif
 7  Endfor
 8  If (Policy=1)  // virtual machine scheduling based
                   // on the urgency of tasks
 9    If (currentVm.signal.thisLeftWorkload>
        currentVm.signal.leftWorkload
          &&currentVm.signal.thisLeftWorkload>=
            VMinfo.lwl)
10      currentvm.schedulerate←100%;
11      Foreach (vm in VMinfo.vms)
12        If (vm<>currentvm)
13          Vm.schedulerate←0%;
14        Endif
15      Endfor
16    Else
17        sum←0;
18        Foreach (vm in VMinfo.vms)
19          Sum+←Vm.signal.leftWorkload;
20        Endfor
21        Foreach (vm in VMinfo.vms)
22          vm.schedulerate←vm.signal.
              leftWorkload/sum;
23        Endfor
24    Endif
25  Else
26      sum←0;
27      Foreach (vm in VMinfo.vms)
28        sum+←vm.signal.timerate;
29      Endfor
30      Foreach (vm in VMinfo.vms)
31        vm.schedulerate←schedulerate computed
            using Eq. (2);
32      Endfor
33  Endif
34  Foreach (vm in VMinfo.vms)
35      vm.timeslice←vm.schedulerate*$\Delta t$;
36  Endfor
37  Sort VMinfo.vms order by vm.timeslice desc;
38  Determine the credit value for all VCPUs based on
      their vm.timeslice;
```

Fig. 7 Virtual machine scheduling algorithm (VSA)

TimeRate of tasks in the current virtual machine and the average TimeRate of tasks in this physical machine. In our design, different virtual machine scheduling policies lead to different occupied frequencies (schedulerate and credit) in the next execution cycle.

We divide Policy 1 into two conditions: firstly, schedulerate is determined by the comparison between the remaining assignments of the current virtual machine and the average remaining assignments of virtual machines in the host. If the former is greater than the latter and also greater than the maximum limited load of the system (VMinfo.lwl), then the current virtual machine will be scheduled to run in the next executable cycle and others must stop running, because the remaining assignment of the current virtual machine is so great that it has exceeded the limit of the system and is far behind the tasks in parallel status. Otherwise, the second method is adopted; that is, the proportion of the remaining assignments of the current virtual machine in all assignments of virtual machines is taken as schedulerate. These two methods can ensure the system proceeds synchronously with the tasks in virtual machines.

With Policy 2, schedulerate is decided by the TimeRate in all virtual machines. We define it as

$$\begin{aligned}\text{schdulerate}_i = {} & [\max(\text{Vm.signal.TimeRate}) \\ & + \min(\text{Vm.signal.TimeRate}) \\ & - \text{Vm}_i\text{.signal.TimeRate}] \\ & \cdot \left(\sum \text{Vm.signal.TimeRate}\right)^{-1}.\end{aligned} \tag{2}$$

Eq. (2) shows that, the greater is the TimeRate in a virtual machine, the lower is the schedulerate, because a high TimeRate means the virtual machine is ahead of schedule. Then, the speed decreases in the next executable cycle to implement the synchronization among tasks.

As the schedulerate is defined as the ratio of time slices (schedulerate$\in$[0, 1]), the timeslice can be determined by multiplying Δt by the schedulerate after the schedulerate is calculated.

In rows 1–7 of the VSA (Fig. 7), we set the default virtual machine scheduling policy as Policy 1. Then, the function getAvgTasksTimeRates computes the average progress of tasks. If the deviation between the TimeRate of tasks in the current virtual machine and the TimeRate of concurrent tasks in this physical machine is greater than the maximum assignment allowed in this system, VSA will adopt Policy 2.

In rows 8–24, the algorithm takes the virtual machine scheduling in general conditions, in which the remaining task assignments directly affect the proportion of the time slices. In row 9, if the remaining assignments are greater than VMinfo.lwl, this virtual machine will be scheduled with the schedulerate=100% and the other virtual machines are set at 0% in the next execution cycle. Otherwise, the ratios among these remaining assignments decide their schedule-rate in the next execution cycle.

In rows 25–33, the algorithm takes the virtual machine scheduling with synchronization requirements, in which the urgency of tasks directly affects the proportion of the time slices. We aim to exchange the completion time of virtual machines with CPU resources, so more resources will be allocated to urgent tasks so as to reduce the pending time of slower tasks to implement the balance of all tasks.

In rows 34–35, the proportions of time slices in the next execution cycles are determined, in which fewer time slices will be allocated to the virtual machines with forward progress.

In row 37, the virtual machines are sorted by their time slices and form a queue for VCPU scheduling.

In row 38, the algorithm determines the credit values of VCPUs. We implement this algorithm in the Xen hypervisor, in which the scheduling unit is the credit of a VCPU. To ensure fairness and flexibility, each virtual machine is set the same quantity of VCPUs and CPUs, and we let each VPCU map to a different CPU. In this way, each virtual machine has the same right to occupy CPUs. The VCPUs occupy CPUs according to their credit values. Thus, the credit value of each VCPU is adjusted in the VSA algorithm based on the time slice of virtual machines in their execution cycles. As a result, the credit value is taken to determine the usage that a VCPU occupies in a CPU in an execution cycle.

4.3.2 Algorithm implementation

The VSA is implemented in the Xen CPU virtualization subsystem by changing the source code of Xen 3.3, and then it is defined as the scheduler sched_vsa_def. Meanwhile, VSA is set as a default algorithm saved in the variable opt_sched (Fig. 8).

```
// xen/com/schedule.c 55-61
static char opt_sched[10]="vsa";
extern structure scheduler sched_bvt_def;
extern structure scheduler sched_sedf_def;
extern structure scheduler sched_credit_def;
extern structure scheduler sched_vsa_def;
static struct scheduler *schedulers[]={
   &sched_bvt_def, &sched_sedf_def,
   &sched_credit_def, &sched_vsa_def, Null};
```

Fig. 8 Definition of the CPU scheduling algorithm

The interfaces of sched_vsa_def are composed of some properties such as 'char *name' and 12 functions such as '(*init)(void)', among which the most important is the function 'struct task_slice(*do_schedule)(s_time_t)' which implements VSA.

When the Xen hypervisor starts, VSA is initialized by the function init_idle_domain. The function scheduler_init obtains the current scheduler by comparing opt_sched and schedulers[]. Then, the macro SCHED_OP is called to initialize sched_vsa_def. VMinfo is defined by its initialization function sched_vsa_def. The vms in VMinfo=(vms, lwl, tr) is collected and saved in the Xen hypersvisor. lwl and tr are appointed the constants in the source codes of Xen 3.3, which can be adjusted by the hyper-call _HYPERVISOR_VCPU_OP. In vm={signal, schedulerate, timeslice} of vms, we change the hyper-call _HYPERVISOR_VCPU_OP to construct a VMsignal, and then transmit the VMsignal into the vm.signal in VSA. The starting values of schedulerate and timeslice are set as 0 in our experiments.

5 Experimental evaluations

Based on the source codes of Xen 3.3 and VZlinux 2.6.18, we have developed a prototype system in the desktop operation system Fedora core 12.0. We constructed a development environment in Eclipse for Linux by using a GCC compiler and C language. The machine was a PC with Intel Core 2, 2.8 GHz CPU, 2 GB DDR RAM, PAE, and a 160 GB hard disk.

First, we implemented TPCM, and obtained three key components of TPCS. Then we developed source codes which were deployed into a testing environment to verify their effectiveness. This testing environment was built on a pair of two-socket servers, with each socket having four Intel Xeon 1.6 GHz CPUs. One server had 4 GB DDR RAM and the other had 2 GB DDR RAM. The two servers were connected by a 1000 Mb/s Ethernet network. We used Linux 2.6.18 with Xen 3.3 as the operation system. The storage was exported to a migrated VM from a file system image, which was accessed via the Network File System (NFS) protocol. We pre-installed Red Hat Enterprise Linux 5 as the guest OS in VMs. After determining the reliability of the prototype system, we built them on a larger cluster to make system evaluations.

5.1 TPCM evaluation

5.1.1 Mapping efficiency with different parameter settings

Here we describe an example used to evaluate the virtual machine mapper TPCM and scheduler TPCS presented in this paper. We take the simulation of a cold flow impulsive experiment for a car engine (CFIE) as the instance. CFIE is a typical coupling process which considers the relations affecting flow field and structure. The CFIE project was constructed by three subprojects with 22 subtasks. Massively parallel computing was found to exist in these 22 subtasks after analyzing their requirements. The taskSet is shown in Fig. 9.

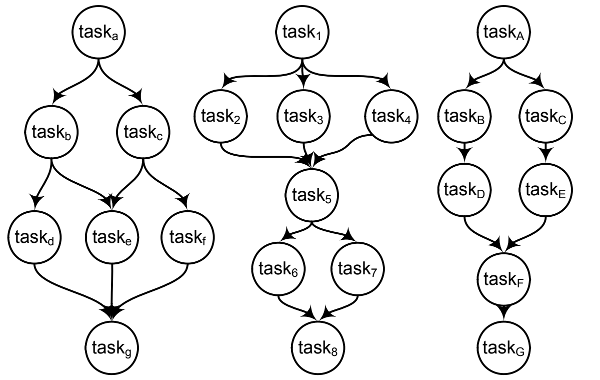


Fig. 9 Construction of parallel computing tasks

Based on the description of the parallel computing tasks, the subtasks in taskSet are described by their assignments and the relationships between them are identified as serial, parallel, indirect, or connectionless. To adapt to the threshold of inputting parameters in algorithm VMMA, the assignments of tasks are normalized to (0, 1].

We completed the mapping from tasks to virtual machines using VMMA. First, we used two hosts to make an evaluation, so PMS.count was set as 2. Based on the performances of the hardware, the quantity of virtual machines was limited to 4. To produce a higher efficiency, we set $\varepsilon=3<4$. Any group of parameters ξ, ς, Φ would produce a different value of 'index' (an average number of virtual machines) in algorithm VMMA. A higher ξ refers to a higher demand in a virtual machine combination. Therefore, it is very difficult to combine the virtual machines. A higher ς refers to a lower demand in a virtual machine combination. Therefore, it is easy to combine the virtual machines. A higher Φ refers to a lower demand to load balance in servers. Therefore, it is also easy to deploy the virtual machines. If we set different inputting values of ξ, ς, Φ to VMMA, different outputs would be shown after CFIE runs.

The values of 'index' meet the features of VMMA in virtual machine combination (Table 1). The data in the column headed 'weights of assignments in virtual machines' show that VMMA can maintain the load balance among virtual machines and physical machines. According to the characteristics of parallel computing among virtual machines and the quantity of physical machines in clusters, we set the input values of parameters in VMMA as PMS.count=2, $\varepsilon=3$, $\xi=0.1$, $\varsigma=0.9$, $\Phi=0.2$. We chose such settings for two reasons: first, too many virtual machines in a physical machine would increase the complexity of management and lead to a higher overhead in synchronization, communication, and switching. Therefore, the quantity of virtual machines was set as 6. Second, the performance of one server is better than the other because of its greater DDR RAM, so the weights of their assignments were set as 0.56:0.44. Based on such parameters, the mapping from tasks to virtual machines after VMMA runs is shown in Fig. 10. We conclude that TPCM is very effective for solving the mapping of tasks to virtual machines.

Table 1 The results of virtual machine mapping with different parameters

ξ	ς	Φ	Index	Weights of assignments in virtual machines	Weights of assignments in physical machines
0.1	0.7	0.1	10	0.09, 0.13, 0.06, 0.10, 0.14, 0.08, 0.14, 0.08, 0.07, 0.11	0.52, 0.48
		0.2	10	0.11, 0.13, 0.09, 0.09, 0.11, 0.09, 0.10, 0.07, 0.11, 0.10	0.53, 0.47
		0.3	10	0.12, 0.08, 0.11, 0.10, 0.09, 0.11, 0.10, 0.12, 0.08, 0.10	0.50, 0.51
	0.8	0.1	8	0.19, 0.06, 0.11, 0.08, 0.18, 0.08, 0.13, 0.18	0.44, 0.57
		0.2	8	0.08, 0.16, 0.16, 0.10, 0.10, 0.08, 0.17, 0.15	0.50, 0.50
		0.3	8	0.14, 0.12, 0.12, 0.13, 0.12, 0.14, 0.12, 0.12	0.51, 0.50
	0.9	0.1	6	0.15, 0.14, 0.18, 0.22, 0.16, 0.16	0.47, 0.54
		0.2	**6**	**0.20, 0.13, 0.23, 0.18, 0.12, 0.15**	**0.56, 0.44**
		0.3	6	0.17, 0.17, 0.17, 0.18, 0.16, 0.14	0.51, 0.48
	1.0	0.1	4	0.34, 0.19, 0.28, 0.18	0.53, 0.46
		0.2	4	0.22, 0.26, 0.26, 0.27	0.48, 0.53
		0.3	4	0.26, 0.23, 0.26, 0.25	0.49, 0.51
0.2	0.7	0.1	12	0.11, 0.08, 0.11, 0.10, 0.06, 0.11, 0.09, 0.10, 0.08, 0.05, 0.05, 0.08	0.57, 0.45
		0.2	12	0.11, 0.10, 0.11, 0.08, 0.07, 0.05, 0.07, 0.08, 0.08, 0.07, 0.08, 0.11	0.52, 0.49
		0.3	12	0.07, 0.08, 0.07, 0.08, 0.09, 0.09, 0.09, 0.08, 0.09, 0.09, 0.07, 0.10	0.48, 0.52
	0.8	0.1	10	0.13, 0.11, 0.07, 0.13, 0.13, 0.06, 0.08, 0.08, 0.08, 0.14	0.57, 0.44
		0.2	10	0.11, 0.11, 0.11, 0.06, 0.12, 0.12, 0.06, 0.12, 0.11, 0.10	0.51, 0.51
		0.3	10	0.09, 0.12, 0.09, 0.09, 0.10, 0.10, 0.12, 0.10, 0.10, 0.09	0.49, 0.51
	0.9	0.1	8	0.15, 0.09, 0.20, 0.14, 0.06, 0.09, 0.19, 0.08	0.58, 0.42
		0.2	8	0.12, 0.13, 0.08, 0.15, 0.14, 0.11, 0.12, 0.15	0.48, 0.52
		0.3	8	0.11, 0.13, 0.15, 0.13, 0.13, 0.11, 0.10, 0.13	0.52, 0.47
	1.0	0.1	5	0.24, 0.19, 0.12, 0.19, 0.26	0.55, 0.45
		0.2	5	0.31, 0.17, 0.14, 0.23, 0.16	0.62, 0.39
		0.3	5	0.17, 0.21, 0.22, 0.19, 0.22	0.60, 0.41

ξ: the minimum deviation among virtual machines; ς: the maximum load that a virtual machine can afford; Φ: the maximum load that a physical machine can afford

5.1.2 Mapping efficiency with different quantities of physical machines

We executed the algorithm VMMA in a larger cluster to verify the efficiency of TPCM. The cluster belongs to the High Performance Computing Lab, Xi'an University of Technology, China. This Lab was established in 2010, and now contains a total of 40 machines, including 38 computing nodes, one I/O node, and one management node. Each machine was configured to an IBM x3550 M2 Server, Intel Xeon (four cores, 5500 serial), three caches, each ≥8 MB, DDR3 RDIMM memory, 6*4 GB, Disk I/O:2.5″ SAS/SATA/SSD, integrated hardware RAID-0/1/10, Optional supporting RAID-5, and two 10/100/1000 Mb adaptive Ethernet cards. We set the number of computing nodes as 2, 8, 16, 26, or 38 to execute the CFIE. Meanwhile, the quantity of subtasks in CFIE increased by 21 times: we ran CFIE 21 times, and the workflow application had up to 462 subtasks. After the application was submitted to the prototype system, we recorded and summarized the completion time, and obtained the trend of changes in algorithm mapping efficiency with different quantities of physical machines (Fig. 11). Completion time increased with the increase in the quantity of tasks with different numbers of physical machines (Fig. 11). When there were only a few physical machines (PMS.count<8), the completion time changed significantly. But when there were many physical machines (PMS.count>26), the variation was relatively stable. For the same assignments, the completion time decreased with the increase in machines. When there were many (>352) subtasks, the completion time increased with the increase in the number of machines, but when there were fewer (<264) subtasks, the completion time changed only slightly. Thus, the algorithm performs well only when the ratio of the quantity of tasks to the quantity of machines is controlled within a range of 3–12. Fig. 11 shows that the TPCM showed high efficiency in our environment.

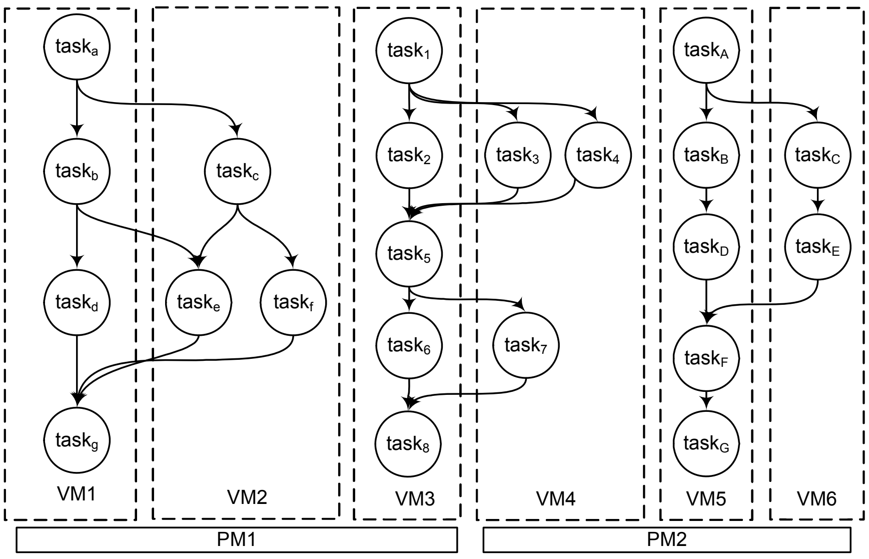


Fig. 10 The mapping of tasks to virtual machines

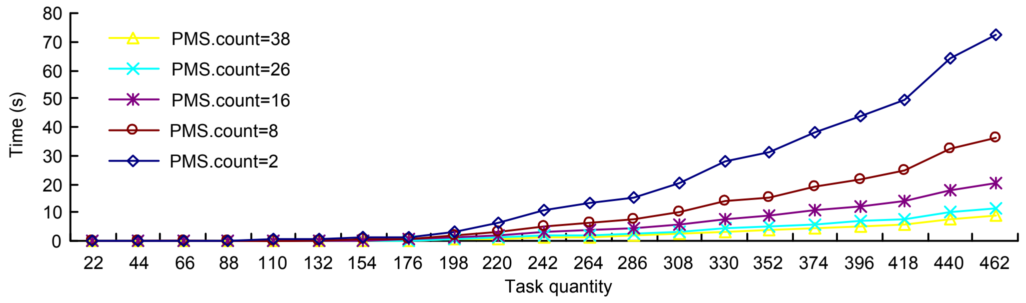


Fig. 11 Running time plotted against task quantity with different numbers of machines

5.2 TPCS evaluation

5.2.1 Efficiency with different parameter settings

We tested the performance of the TPCS with the parameter configuration shown in Fig. 10 to verify the feasibility of the three core components. The Credit algorithm is a CPU scheduling algorithm for SMP hosts in the Xen hypervisor, so TPCS or Credit was set as the scheduler and we collected the results to evaluate their performance in systems. First, we used TPCS to run CFIE. With different parameters, we collected the completion times of 22 subtasks and the CPU utilizations in different execution cycles. We then used the Credit scheduler to run CFIE to compare with TPCS. In VSTA, vmscmd_ioctl, and VSA, the factors affecting the efficiency of TPCS are the two parameters Markquantity and Δt. Thus, we tested the performance in the following settings: Markquantity=1000, 2000, 3000, 4000 and Δt=0.1, 0.2, 0.3, 0.4.

Fig. 12 shows the performance comparison between Credit and TPCS with different Markquantity (Δt=0.1). Markquantity=2000 led to the minimum completion time (147.279 s) in four settings, and the completion times in Markquantity=1000, 3000, 4000 were 174.755, 192.230, and 244.657 s respectively, so the quantity of markers inputted to the tasks had a vital effect on the performance of the system. We can determine a value of Markquantity whose completion time of tasks is the lowest of all. More markers in tasks result in more scanning time in middleware with a higher overhead, and the tasks may not be synchronized when meeting fewer markers in the tasks. A suitable Markquantity would keep the overhead in systems at the lowest level. We also see that when Markquantity=1000 or 2000 (Δt=0.1), the completion time of TPCS was shorter than that of Credit. We conclude that TPCS can fully utilize the parallelism among tasks to allocate the CPU resources with

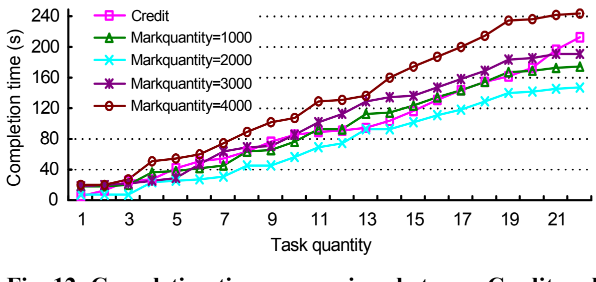


Fig. 12 Completion time comparison between Credit and TPCS with different Markquantity (Δt=0.1)

rationality to produce a lower overhead. The Credit scheduler would also make the completion time proportional to the number of assignments, but the TPCS scheduler enables some tasks to complete their assignments within the same time because of the synchronization in parallel computing.

Fig. 13 shows the performance comparison of Credit and TPCS with different Δt (Markquantity=2000). The curves of Δt=0.1, 0.2, 0.3, and 0.4 almost overlapped. We conclude that if the completion time of tasks is far greater than Δt, the execution cycle has no effect on the completion time of tasks, so all the curves in Fig. 13 fluctuate over the range 147–154, no more than 3% of the total time consumption. As long as we set a reasonable Markquantity, the performance of the TPCS scheduler should be better than that of the Credit scheduler.

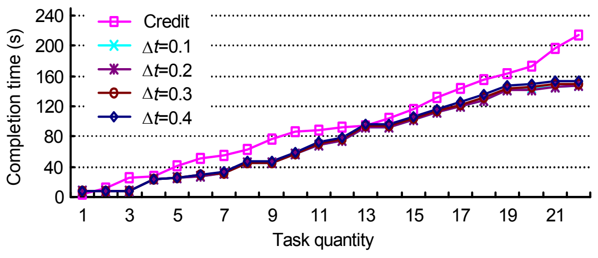


Fig. 13 Completion time comparison between Credit and TPCS with different Δt (Markquantity=2000)

Here, we summarize the resource utilization of the two algorithms. The resource utilization of TPCS, based on seven groups of parameters (Markquantity, Δt)=(1000, 0.1), (2000, 0.1), (3000, 0.1), (4000, 0.1), (2000, 0.2), (2000, 0.3), (2000, 0.4), was compared with that of Credit (Fig. 14).

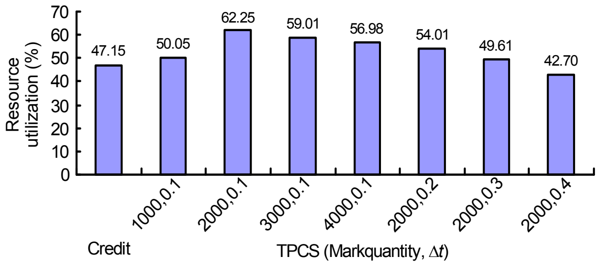


Fig. 14 CPU utilizations in different configurations

The eight average CPU utilizations (Fig. 14) for eight kinds of configuration in frontier 200 execution cycles show that Markquantity and Δt greatly affect the CPU utilization. When Markquantity=2000 and Δt=0.1, CPU utilization reached 62.25%, and at this time point, CFIE had the shortest completion time. Thus, the TPCS scheduler can achieve a higher performance by improving the resource utilization of virtual machines. In addition, the larger is Δ*t*, the lower is the CPU utilization, because a long execution cycle in systems would produce fewer times for collaboration in TPCS. That is, it would be difficult to implement the policy of exchanging the completion time for CPU resources. In our experiment, most of the CPU utilizations in the TPCS scheduler were greater than those in the Credit scheduler. Therefore, the TPCS scheduler can effectively mine the parallelism among tasks and implement a better dynamic resource scheduling by reducing the overheads from synchronization, communication, and switching, thereby accelerating the execution of tasks.

5.2.2 Efficiency with different quantities of machines

The efficiency of TPCS was verified again in the cluster composed of 38 computing nodes and 462 subtasks. The quantity of machines was set as 2, 8, 16, 26, or 38, and the quantities of tasks were set as 22, 44, 66, …, 462. After the subtasks were submitted to the prototype system, the completion time was recorded and summarized. Fig. 15 shows the change in the efficiency of TPCS with different quantities of machines.

The completion time increased with the increase in task quantity with a fixed quantity of machines. For the same quantity of tasks, the greater was the quantity of machines, the shorter was the completion time. Meanwhile, when there were only a few (<8) machines, the rate of increase was greater than when there were many (>26). Only if the quantity of tasks and the quantity of machines reached a certain ratio (a ratio of 3–12), can the algorithm achieve good performance. We conclude that TPCS maintains a high efficiency with different numbers of machines.

Fig. 16 shows the speedups with different numbers of machines and with different tasks. The speedups showed a trend of a slow increase with the increase in task quantity. When there were 38 machines and the quantity of tasks was less than 66 (especially 22), the speedup was much smaller than in other cases. This is because when the ratio of the quantity of tasks to the number of machines is far less than 3, the system cannot achieve a good performance.

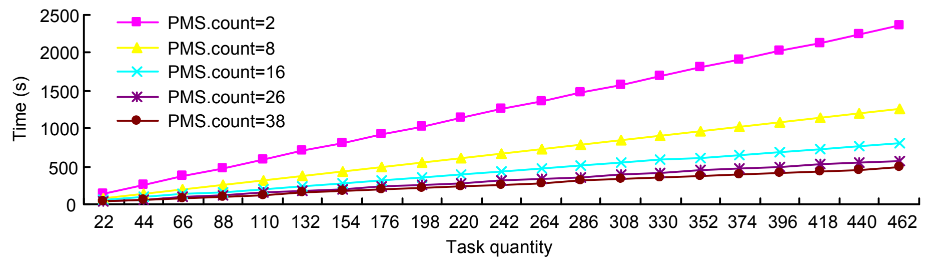


Fig. 15 Completion times under different numbers of physical machines using the TPCS scheduler with a cluster composed of 38 computing nodes and 462 subtasks

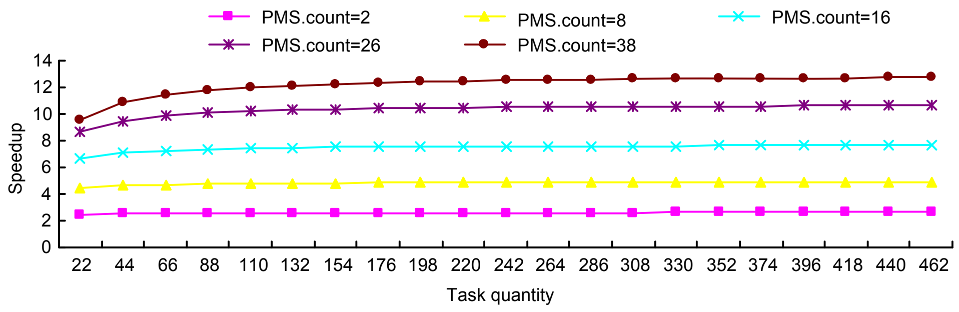


Fig. 16 Speedups under different numbers of physical machines using the TPCS scheduler with a cluster composed of 38 computing nodes and 462 subtasks

In a stand-alone environment, we ran CFIE with from 22 to 462 tasks and then summarized the completion times as 362.228, 652.010, 941.793, 1231.575, 1521.358, 1811.140, 2100.922, 2390.705, 2680.487, 2970.270, 3260.052, 3549.834, 3839.617, 4129.399, 4419.182, 4708.964, 4998.746, 5288.529, 5578.311, 5868.094, and 6157.876 s. These values are approximately proportional to the quantity of tasks. We took these values as the reference values to compute the speedups

For the same quantity of tasks, a larger number of machines would lead to a greater acceleration. When the ratio of the quantity of tasks to the number of machines was from 3 to 12, the speedups showed an equally spaced growth trend with the growth of the quantity of tasks. Thus, TPCS showed high acceleration efficiency in our experiment.

5.2.3 A comparison among scheduling algorithms

To further verify the performance of TPCS, we compared it with other CPU scheduling algorithms from related studies. We ran the CFIE with 462 subtasks in the Xen hypervisor for all kinds of algorithms. Since BVT and SEDF have been implemented in the Xen hypervisor, we need only to set the scheduler in the CPU subsystem as sched_bvt_def or sched_sedf_def. Other algorithms, SMART, SJF, EEVDF, SFQ, EDF, DSS, TBS, and CBS, were also created as corresponding schedulers. A virtual machine was taken as a task, a process, or a thread, because these algorithms were designed originally for non-virtualized environments. After the subtasks were completed, the completion time was recorded. We ran the CFIE program five times, and the completion times were averaged. The results are shown in Fig. 17.

The TPCS scheduler had the minimum completion time among all the schedulers for 462 subtasks in CFIE using different numbers of machines (PMS.count =2, 8, 16, 26, 38), followed by the Credit scheduler. Unlike in non-parallel computing systems, DSS and TBS algorithms in parallel computing had the maximum completion times, exceeding 300 s. We conclude that TPCS is the best scheduler oriented for parallel computing virtual machines.

To determine the speedups of all algorithms, we ran the CFIE program in a stand-alone machine in Credit and computed the speedups based on the data in Fig. 17. The results are shown in Fig. 18. The speedups in TPCS were greater than those in the other algorithms with the same number of physical machines.

5.2.4 Discussions

The three experiments above evaluated the performance of TPCS and Credit. We suggest that the ratio of the quantity of tasks to the number of machines should be controlled at 3–12 in our design. Within this scope, different parameter settings and different numbers of tasks and machines can lead to a high efficiency. The resource utilization of TPCS is higher than that of Credit. TPCS can also produce a shorter completion time and higher speedup compared with other similar algorithms. The experiments verified the feasibility of the TPCS scheduler. In a virtualized environment, a virtual machine scheduling policy that uses resources in exchange for completion time is feasible under the condition of adequate physical resources.

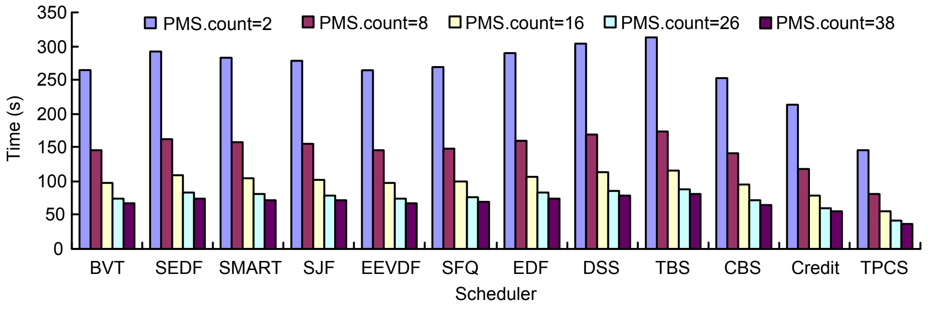


Fig. 17 Comparison of completion time among the different scheduling algorithms

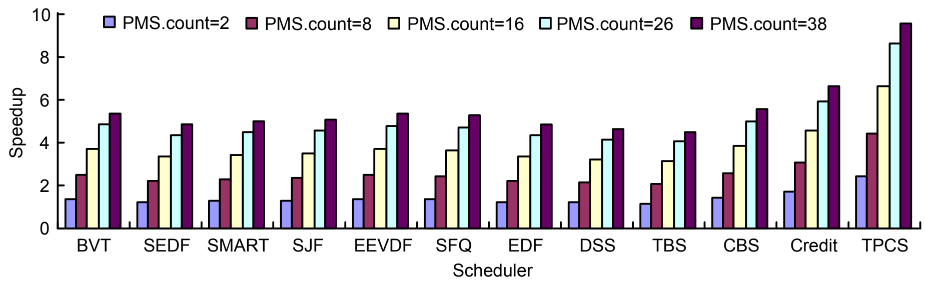


Fig. 18 Comparison of speedup among the different scheduling algorithms

6 Conclusions

In this paper, we studied virtual machine mapping and scheduling problems in parallel computing. For the problem of virtual machine mapping, a new methodology describing the tasks was presented to summarize the four relations, serial, parallel, indirect, and connectionless. We designed a mapper, TPCM, with the VMMA algorithm to implement load balance by fully mining the parallelism of tasks and deploying the virtual machines into balanced physical machines. For the problem of virtual machine scheduling, we designed a virtual machine scheduler, TPCS, including a middleware supporting application-driven mode, a device driver in the guest OS kernel, and a virtual machine scheduling algorithm (VSA). TPCS transmits the progress of tasks from the top layer to the underlying layer, and the Xen CPU virtualization subsystem schedules the virtual machines based on the progress of tasks to keep all subtasks simultaneous. Thus, we implemented a policy to exchange the completion time of tasks for CPU resources. This policy can ensure a higher collaboration among tasks to reduce the overheads in synchronization, communication, and switching. Experiments showed that TPCM can realize the load balance according to the parameters in clusters. The settings of the inputting information to tasks make the TPCS scheduler complete its tasks in a shorter time than Credit and other schedulers. The execution cycle has no effect on the performance of the virtual machine scheduling algorithm, but it has a great effect on CPU utilization. Overall, the TPCS scheduler can overcome the shortcomings of Credit and other schedulers in perceiving the progress of tasks, and thus is better suited to parallel computing than Credit and other schedulers.

References

Abeni, L., Buttazzo, G., 1998. Integrating Multimedia Applications in Hard Real-Time Systems. Proc. 19th IEEE Real-Time Systems Symp., p.4-13.

Alessandro, R., 2004. Linux Device Drivers (3rd Ed.). Power Press, Beijing, p.100-152.

Aspnes, J., Azar, Y., Fiat, A., Plotkin, S., Waarts, O., 1997. On-line routing of virtual circuits with applications to load balance and machine scheduling. *J. ACM*, **44**(3):486-504. [doi:10.1145/258128.258201]

Bansal, S., Kumar, P., Singh, K., 2003. An improved duplication strategy for scheduling precedence constrained graphs in multiprocessor systems. *IEEE Trans. Parall. Distr. Syst.*, **14**(6):533-544. [doi:10.1109/TPDS.2003.1206502]

Baskiyar, S., Dickinson, C., 2005. Scheduling directed a-cyclic task graphs on a bounded set of heterogeneous processors using task duplication. *J. Parall. Distr. Comput.*, **65**(8):911-921. [doi:10.1016/j.jpdc.2005.01.006]

Bavier, A., Peterson, L.L., Moseberger, D., 1999. BERT: a Scheduler for Best Effort and Realtime Tasks. Technical Report, Department of Computer Science, Princeton University, USA, p.12.

Boyer, W.F., Hura, G.S., 2005. Non-evolutionary algorithm for scheduling dependent tasks in distributed heterogeneous computing environments. *J. Parall Distr. Comput.*, **65**(9):1035-1046. [doi:10.1016/j.jpdc.2005.04.017]

Caprita, B., Chan, W., Nieh, J., Clifford, S., Zheng, H.Q., 2005. Group Ratio Round-Robin: *O*(1) Proportional Share Scheduling for Uniprocessor and Multiprocessor Systems. Proc. Usenix Annual Technical Conf., p.8-16.

Cerin, C., Fkaier, H., Jemni, M., 2008. Experimental Study of Thread Scheduling Libraries on Degraded CPU. 14th IEEE Int. Conf. on Parallel and Distributed Systems, p.697-704. [doi:10.1109/ICPADS.2008.102]

Chan, W.C., Nieh, J., 2003. Group Ratio Round-Robin: an *O(I)* Proportional Share Scheduler. Technical Report, Department of Computer Science, Columbia University, USA, p.1-5.

Chandra, A., Shenoy, P., 2008. Hierarchical scheduling for symmetric multiprocessors. *IEEE Trans. Parall. Distr. Syst.*, **19**(3):418-431. [doi:10.1109/TPDS.2007.70755]

Chen, S., Gibbons, P.B., Kozuch, M., Liaskovitis, V., Ailamaki, A., Blelloch, G.E., Falsafi, B., Fix, L., Hardavellas, N., Mowry, T.C., *et al.*, 2007. Scheduling Threads for Constructive Cache Sharing on CMPS. Proc. 19th Annual ACM Symp. on Parallel Algorithms and Architectures, p.105-115. [doi:10.1145/1248377.1248396]

Chen, X.J., Zhang, J., Li, J.H., Li, X., 2011a. Resource management framework for collaborative computing systems over multiple virtual machines. *Serv. Orient. Comput. Appl.*, **5**(4):225-243. [doi:10.1007/s11761-011-0087-6]

Chen, X.J., Zhang, J., Li, J.H., Li, X., 2011b. Resource virtualization methodology for on-demand allocation in cloud computing systems. *Serv. Orient. Comput. Appl.*, in press. [doi:10.1007/s11761-011-0092-9]

Cherkasova, L., Gupta, D., Vahdat, A., 2007. Comparison of the three CPU schedulers in Xen. *ACM SIGMETRICS Perform. Eval. Rev.*, **35**(2):42-51. [doi:10.1145/1330555.1330556]

Chuzhoy, C., Naor, J., 2006. New hardness results for congestion minimization and machine scheduling. *J. ACM*, **53**(5):707-721. [doi:10.1145/1183907.1183908]

Cota-Robles, E.C., Flautner, K., 2008. Real-Time Scheduling of Virtual Machines. U.S. Patent 7356817.

Dail, H., Casanova, H., Berman, F., 2002. A Decoupled Scheduling Approach for the GrADS Program Development Environment. Proc. ACM/IEEE Conf. on Supercomputing, p.55-62. [doi:10.1109/SC.2002.10009]

Danish, M., Li, Y., Richard, W., 2011. Virtual-CPU Scheduling in the Quest Operating System. 17th IEEE Real-Time and Embedded Technology and Applications Symp., p.169-179. [doi:10.1109/RTAS.2011.24]

Daoud, M.I., Kharma, N., 2011. A hybrid heuristic-genetic algorithm for task scheduling in heterogeneous processor networks. *J. Parall. Distr. Comput.*, **71**(11):1518-1531. [doi:10.1016/j.jpdc.2011.05.005]

Duda, K.J., Cheriton, D.R., 1999. Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler. Proc. 17th ACM SOSP, p.1-16.

El-Rewini, H., Lewis, T.G., 1990. Scheduling parallel program tasks onto arbitrary target machines. *J. Parall. Distr. Comput.*, **9**(2):138-153. [doi:10.1016/0743-7315(90)90042-N]

Fu, S., Xu, C.Z., 2006. Stochastic modeling and analysis of hybrid mobility in reconfigurable distributed virtual machines. *J. Parall. Distr. Comput.*, **66**(11):1442-1454. [doi:10.1016/j.jpdc.2006.05.006]

Fumio, M., 2009. Optimum Virtual Machine Placement and Rejuvenation Scheduling for High-Availability Consolidated Server Systems. CISUC, NEC, Japan, p.6-12.

Ghazalie, T., Baker, T., 1995. Aperiodic servers in a deadline scheduling environment. *Real-Time Syst.*, **9**(1):31-67. [doi:10.1007/BF01094172]

Govil, K., Teodosiu, D., Huang, Y., Rosenblum, M., 2000. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, **18**(3):229-262. [doi:10.1145/354871.354873]

Govindan, S., Nath, A.R., Das, A., Urgaonkar, B., Sivasubramaniam, A., 2007. Xen and Co.: Communication-Aware CPU Scheduling for Consolidated Xen-Based Hosting Platforms. Proc. 3rd Int. Conf. on Virtual Execution Environments, p.126-136. [doi:10.1145/1254810.1254828]

Govindan, S., Choi, J., Nath, A.R., Das, A., Urgaonkar, B., Sivasubramaniam, A., 2009. Xen and Co.: communication-aware CPU management in consolidated Xen-based hosting platforms. *IEEE Trans. Comput.*, **58**(8):1111-1125. [doi:10.1109/TC.2009.53]

Goyal, P., Vin, H.M., Cheng, H., 1996. Start-time fair queuing: a scheduling algorithm for integrated services packet switching networks. *ACM SIGCOMM Comput. Commun. Rev.*, **26**(4):157-168. [doi:10.1145/248157.248171]

Grefenstette, J., Back, T., Fogel, D.B., Michalewicz, Z., 1997. Handbook of Evolutionary Computation (1st Ed.). Oxford University Press, Oxford, UK, p.241-246.

Gupta, D., Cherkasova, L., Gardner, R., Vahdat, A., 2006. Enforcing Performance Isolation across Virtual Machines in Xen. Proc. 7th Int. Middleware Conf., p.342-362.

Hamidzadeh, B., Kit, L.Y., Lilja, D.J., 2000. Dynamic task scheduling using online optimization. *IEEE Trans. Parall. Distr. Syst.*, **11**(11):1151-1163. [doi:10.1109/71.888636]

Hiroshi, Y., Kenji, K., 2007. Foxy Technique: Tricking Operating System Policies with a Virtual Machine Monitor. Proc. 3rd Int. Conf. on Virtual Execution Environments, p.55-64.

Huai, J.P., Li, Q., Hu, C.M., 2007. Research and design on hypervisor based virtual computing environment. *J. Software*, **18**(8):2016-2026 (in Chinese). [doi:10.1360/jos182016]

Ilavarasan, E., Thambidurai, P., Mahilmannan, R., 2005. Performance Effective Task Scheduling Algorithm for Heterogeneous Computing System. Proc. 4th Int. Symp. on Parallel and Distributed Computing, p.28-38. [doi:10.1109/ISPDC.2005.39]

Iosup, A., Dumitrescu, C., Epema, D., Li, H., Wolters, L., 2006. How Are Real Grids Used the Analysis of Four Grid Traces and Its Implications. Proc. 7th IEEE/ACM Int. Conf. on Grid Computing, p.262-269. [doi:10.1109/ICGRID.2006.311024]

Iverson, M.A., Ozguner, F., Potter, L., 1999. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. *IEEE Trans. Comput.*, **48**(12):1374-1379. [doi:10.1109/12.817403]

James, E.S., Ravi, N., 2007. Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann, p.113-152.

Jin, H., Hu, Q.H., Liao, X.F., Chen, H., Deng, D.F., 2005. IMAC: an Importance-Level Based Adaptive CPU Scheduling Scheme for Multimedia and Non-Real Time Applications. 3rd ACS/IEEE Int. Conf. on Computer Systems and Applications, p.119-125. [doi:10.1109/AICCSA.2005.1387108]

Jones, M.B., Rosu, D., Rosu, M.C., 1997. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. Proc. 16th Symp. on Operating System Principles, p.198-211. [doi:10.1145/269005.266689]

Juve, G., Deelman, E., Vahi, K., Mehta, G., Berriman, B., Berman, B.P., Maechling, P., 2009. Scientific Workflow Applications on Amazon EC2. Proc. 5th IEEE Int. Conf. on E-Science Workshops, p.59-66. [doi:10.1109/ESCIW.2009.5408002]

Katz, D.S., Jacob, J.C., Deelman, E., Kesselman, C., Singh, G., Su, M.H., Berriman, G.B., Good, J., Laity, A.C., Prince, T.A., 2005. A Comparison of Two Methods for Building Astronomical Image Mosaics on a Grid. Proc. Int. Conf. Workshops on Parallel Processing, p.85-94. [doi:10.1109/ICPPW.2005.6]

Kim, H., Lim, H., 2009. Task-Aware Virtual Machine Scheduling for I/O Performance. Proc. ACM SIGPLAN/SIGOPS Int. Conf. on Virtual Execution Environments, p.101-110. [doi:10.1145/1508293.1508308]

Kim, J., Rho, J., Lee, J.O., Ko, M.C., 2005. CPOC: effective static task scheduling for grid computing. *LNCS*, **3726**:477-486. [doi:10.1007/11557654_56]

Kong, X.Z., Lin, C., Jiang, Y.X., Yan, W., Chu, X.W., 2011. Efficient dynamic task scheduling in virtualized data centers with fuzzy prediction. *J. Network Comput. Appl.*, **34**(4):1068-1077. [doi:10.1016/j.jnca.2010.06.001]

Laslo, Z., Golenko-Ginzburg, D., Keren, B., 2008. Optimal booking of machines in a virtual job-shop with stochastic processing times to minimize total machine rental and job tardiness costs. *Int. J. Prod. Econ.*, **111**(2):812-821. [doi:10.1016/j.ijpe.2007.03.018]

Lehoczky, J., Sha, L., Ding, Y., 1989. The Rate Monotonic Scheduling Algorithm: Exact Characteristics and Average Case Behavior. Proc. IEEE Real-Time Systems Symp., p.166-171.

Nesmachnow, S., Cancela, H., Alba, E., 2010. Heterogeneous computing scheduling with evolutionary algorithms. *Soft Comput. Fus. Found. Methodol. Appl.*, **15**(4):685-701. [doi:10.1007/s00500-010-0594-y]

Nieh, J., Lam, M.S., 1997. The Design, Implementation, and Evaluation of SMART: a Scheduler for Multimedia Applications. Proc. 16th ACM Symp. on Operating System Principles, p.184-197. [doi:10.1145/268998.266677]

Ogata, K., 2002. Modern Control Engineering. Prentice Hall, Upper Saddle River, p.52-67.

Pfoh, J., Schneider, C., Eckert, C., 2009. Formal Model for Virtual Machine Introspection. Proc. 1st ACM Workshop on Virtual Machine Security, p.1-10. [doi:10.1145/1655148.1655150]

Phinjaroenphan, P., Bevinakoppa, S., Zeephongsekul, P., 2005. A method for estimating the execution time of a parallel task on a grid node. *LNCS*, **3470**:226-236. [doi:10.1007/11508380_24]

Qi, X.T., Jonathan, F.B., Yu, G., 2006. Disruption management for machine scheduling: the case of SPT schedules. *Int. J. Prod. Econ.*, **103**(1):166-184. [doi:10.1016/j.ijpe.2005.05.021]

Radulescu, A., van Gemund, A.J.C., 2002. Low-cost task scheduling for distributed memory machines. *IEEE Trans. Parall. Distr. Syst.*, **13**(6):648-658. [doi:10.1109/TPDS.2002.1011417]

Rau, M.A., Smirni, E., 1999. Adaptive CPU Scheduling Policies for Mixed Multimedia and Best-Effort Workloads. Proc. 7th Int. Symp. on Modeling, Analysis, and Simulation of Computer Systems, p.45-52. [doi:10.1109/MASCOT.1999.805062]

Rawat, S.S., Rajamani, L., 2009. Experiments with CPU Scheduling Algorithm on a Computational Grid. IEEE Int. Advance Computing Conf., p.71-75. [doi:10.1109/IADCC.2009.4808983]

Rosenblum, M., Garfinkel, T., 2005. Virtual machine monitors: current technology and future trends. *Computer*, **38**(5):39-47. [doi:10.1109/MC.2005.176]

Shi, L., Sun, Y.Y., Wei, L., 2007. Effect of Scheduling Discipline on CPU-MEM Load Sharing System. 6th Int. Conf. on Grid and Cooperative Computing, p.242-249. [doi:10.1109/GCC.2007.64]

Shi, L., Zhou, D.Q., Jin, H., 2009. Xen Virtualization Techonolgy. Huazhong University of Science and Technology Press, Wuhan, China, p.222-224 (in Chinese).

Sih, G.C., Lee, E.A., 1993. A compile-time scheduling heuristic for interconnection constrained heterogeneous processor architectures. *IEEE Trans. Parall. Distr. Syst.*, **4**(2):175-187. [doi:10.1109/71.207593]

Topcuoglu, H., Hariri, S., Wu, M.Y., 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parall. Distr. Syst.*, **13**(3):260-274. [doi:10.1109/71.993206]

Volkmar, U., Joshua, L.V., Espen, S., Uwe, D., 2004. Towards Scalable Multiprocessor Virtual Machines. Proc. 3rd Conf. on Virtual Machine Research and Technology Symp., p.4.

Waldspurger, C.A., Weihl, W.E., 1994. Lottery Scheduling: Flexible Proportional-Share Resource Management. Proc. 1st Usenix Symp. on Operating System Design and Implementation, p.359-368.

Wang, J., Sun, J.L., Wang, X.Y., Yang, X.H., Wang, S.K., Chen, J.B., 2009. Efficient scheduling algorithm for hard real-time tasks in primary-backup based multiprocessor systems. *J. Software*, **20**(10):2628-2636 (in Chinese). [doi:10.3724/SP.J.1001.2009.00577]

Wu, M., Dajski, D., 1990. Hypertool: a programming aid for message passing systems. *IEEE Trans. Parall. Distr. Syst.*, **1**(3):330-343. [doi:10.1109/71.80160]

Zhang, W., Fang, B., He, H., Zhang, H., Hu, M., 2004. Multisite Resource Selection and Scheduling Algorithm on Computational Grid. Proc. 18th Int. Parallel and Distributed Processing Symp., p.105. [doi:10.1109/IPDPS.2004.1303052]

Zhang, W.Z., Tian, Z.H., Zhang, H.L., He, H., Liu, W.M., 2007. Multi-cluster co-allocation scheduling algorithms in virtual computing environment. *J. Software*, **18**(8):2027-2037 (in Chinese). [doi:10.1360/jos182027]

Zomaya, A.Y., Teh, Y.H., 2001. Observations on using genetic algorithms for dynamic load balance. *IEEE Trans. Parall. Distr. Syst.*, **12**(9):899-911. [doi:10.1109/71.954620]