Vol. 36 No. 3 Transactions of Information Processing Society of Japan Mar. 1995

Regular Paper

Development of Methods for Reducing the Spins of Guest Multiprocessors

HIDENORI UMENO,[†] HIDEAKI AMANO[†] and KEIJI SAIJO[††]

Methods are presented for reducing the spins of operating systems (OSs) that are running in guest multiprocessors, which are virtual machines (VMs) that contain multiple logical processors sharing a main memory area. VMs are functional copies of a real host computer. Different OSs can be run in different VMs concurrently. A hypervisor is a program that allocates real resources to, controls, and schedules multiple VMs. Three methods are presented by which a hypervisor can inform a guest OS, which means an OS in a VM, of the processor allocation forms, which mean dedicated real processors or shared real processors by VMs. According to the information, the guest OS determines whether or not it should spin. Only the hypervisor knows the processor allocation forms, because it allocates real processors to guest OSs. Three other methods for reducing the spin of guest OSs are presented. The guest OSs call the hypervisor when they are going to spin, and the hypervisor schedules partner logical processors, which belong to the same VM, before scheduling the spinning logical processor again. These methods are different in the timing at which the hypervisor reschedules the spinning logical processor. The first, called WAPD, sets a spinning logical processor in a wait state until all its partners that are ready have been scheduled. The second, called WOPD, sets a spinning logical processor in a wait state until one of its partners that are ready has been scheduled. The third, called RSLP, requeues the spinning logical processor at the last position in the ready queue. These three methods have been experimentally implemented, and their effects have been measured and estimated quantitatively. For small workloads, the three methods have comparable performance. According to experiments, WOPD cannot suppress an excessive spin of a guest OS for large workloads that have heavy spin activities. WAPD has a slightly (0.02-0.27%) larger overhead than the RSLP, and its performance is comparable to that of RSLP. For large workloads, WAPD has a slightly (3-4%) better performance than RSLP.

1. Introduction

A virtual machine (VM) is a functional copy of a real host computer[1]. Most instructions of a VM are executed directly by the real host computer. A virtual machine system (VMS) consists of multiple VMs running concurrently on the real host computer. Different operating systems (OSs) can be run on different VMs concurrently. They are basically independent of each other. A guest OS means an OS on a VM. It runs on a VM and has to run in a real machine environment without modifications, managing its own allocated resources and scheduling its own tasks, processes, and threads. A hypervisor is a control program that allocates real processors to and schedules VMs. The widely used logical partition system[2] is one of the implementation forms of VMSs.

In mainframe computers "shared-main-memory multiprocessor" (hereafter simply called "multiprocessor") systems are generally used. An N-way multiprocessor means a multiprocessor consisting of N processors sharing main memory. A host multiprocessor is a real host computer that is run in multiprocessor mode. An N-way host multiprocessor consists of N real processors sharing main memory. Similarly, a guest multiprocessor is a VM that is run in multiprocessor mode. An N-way guest multiprocessor consists of N logical processors sharing their main memory area. We present several methods for improving the performance of the guest multiprocessors.

VMs are used as follows. Only one OS runs in a real machine environment. On the other hand, multiple OSs in VMs run concurrently in a virtual machine environment[2]. VMs are used to develop and test multiple OSs, to advance the normal production run of OSs while developing new OSs, and to run multiple OSs constantly for production use under one real host computer.

When VMs are run in an N-way real host computer, N1 ($<N$) real processors may be dedicated to an N1-way guest multiprocessor,

† Development Department 3, General Purpose Computer Division, Hitachi Ltd.
†† Software Development Center, Hitachi Ltd.

and N2 (=N−N1) other real processors may be shared by other VMs. Other forms may be used.

An OS running in a real multiprocessor mode will spin, (that is, loop) in disabled state, which means it cannot be interrupted, to get a lock, request a system process, wait for an event, and so forth. For this purpose, in a real machine a kernel of an OS has to spin on a real processor and cannot relinquish the processor because it has to manage and schedule its own tasks, processes, and threads. In this case, we can say that other real processors surely run a process that clears the spinning conditions, and therefore the spin finishes in a short time. Thus, in a real machine, the spin works well.

An OS running in a guest multiprocessor mode will run in the same way as in a real machine and will spin in the same cases. Such spins may waste CPU time, because logical processors that clear spin conditions are not always scheduled by the hypervisor. Moreover, the spin may cause system failure of a guest OS because it may spin excessively, that is, it may spin over the frequencies designated by itself.

Conventionally, handshaking is taken to avoid this disadvantage in a VM. The hypervisor and a guest OS are basically independent. The hypervisor gives only hardware architectures to the guest OS, and does not know which tasks and processes the guest OS is executing. A guest OS usually does not recognize that it is running in a VM. Handshaking means that the guest OS shakes hands with the hypervisor and lets the guest OS know that it is running in a VM, and take some action to improve its performance in the VM. There are many kinds of handshaking. A hypervisor-call issued by the spinning logical processor to suspend the spin is an example of this handshaking. When it spins, the guest OS determines whether it is running in a logical processor of a VM, and if so, it calls the hypervisor, which will suspend it and schedule another logical processor[2),7)].

We have to give additional consideration to reducing the occurrence of spins in guest multiprocessors for the following reasons:

First, issuing a hypervisor-call is not always better than continuing to spin. In VMs there are multiple OSs running concurrently, and two forms of processor allocation are used[2)]: One is processor dedication, and the other is processor sharing. In processor dedication, N real processors are dedicated one-to-one to N logical processors of an N-way guest multiprocessor. Therefore, spinning gives a better performance than issuing a hypervisor-call, because the spinning of the guest OS soon finishes in the VM, in the same way as in a real machine.

Second, it is not clear what the hypervisor should do in response to a hypervisor-call. When it is in multiprocessor mode, a guest OS runs on multiple logical processors. When the guest OS is going to spin and issues a hypervisor-call on a logical processor, the hypervisor should schedule other logical processors that clear the spin conditions. The scheduling is difficult, because the hypervisor is unable to determine which logical processor can clear the spin conditions.

Conventionally, revised scheduling disciplines are given for an OS and revised application programs are used to reduce spin times in a real machine[3),4)]. They revise process schedulers and application programs, respectively. First, let us consider revised scheduling disciplines. It is shown how multiprogramming and data-dependent behavior affect the spin times of processes, and so complicate the choice between spinning (busy waiting) and blocking (relinquishing the processor). Revised scheduling disciplines to reduce the spin times of processes[3)] are given. One such discipline is that the scheduler should never unschedule a process holding the lock. Next, we turn to revised application programs. The performance of application programs can be improved by a combination of spinning and blocking. That is, application programs may spin under thresholds to get a lock, and if they cannot get it, they may block[4)].

An OS can use the revised scheduling disciplines because it knows the activities of its processes. The OS knows when and which process gets and releases a lock, and when and which processes are synchronizing. It provides application programs with macro-instructions, such as Lock, Unlock, Wait, and Post, for supporting their activities. The application programs use these macro-instructions to request the services of the OS. Therefore, the OS can know their activities.

We cannot apply these conventional disciplines to the hypervisor, for the following reasons. First, the hypervisor and guest OSs are basically independent, except for the handshaking. All the guest OSs are also independent of each other. The hypervisor cannot recognize what a guest OS is spinning for, because the spinning is an internal process of the OS. It may be spinning to go and get a lock, to wait for an event, to request a process. The hypervisor cannot know which logical processor of a VM gets and releases the lock, or when, because the OS in the VM generally locks and unlocks by using nonprivileged instructions, such as the Compare & Swap instruction, which are directly executed by hardware. Moreover, the hypervisor cannot know which logical processor of a VM has finished synchronizing various events, or when, because the synchronizing is an internal activity of processes of the OS in the VM.

Second, an OS is usually made for a real machine, and not for a VM. If we try to apply the revised conventional scheduling disciplines to the hypervisor, we have to redesign the logic of guest OSs for VMs. That is, the guest OS has to inform the hypervisor of the timing of its dynamic locking and unlocking, and the completion of events associated with its spins. This requires large modifications of the OS. Therefore, this redesign is not practical, because the hypervisor has to run currently available OSs, which are widely used now and designed for real machines. We can only require the guest OSs to contain the basic and simple handshaking, which requires only small modifications.

Third, application programs run in enabled state, and therefore, even if they spin excessively, they are interrupted by timers set by an OS, and the OS will run normally. On the other hand, the kernel of an OS spins in disabled state for several reasons. The kernel contains a process-scheduler and runs in non-process mode. It cannot block—that is, wait or relinquish the processor—because no other kernel reactivates it. If the guest OS should spin excessively, it may fail or put the processor off-line, and its performance will be reduced.

Therefore, we present additional methods for reducing the spin of OSs in guest multiprocessors:

1. **How the hypervisor informs a guest OS of the processor allocation forms.** In processor dedication, spinning may be better than calling the hypervisor. On the other hand, in processor sharing, calling the hypervisor is better than spinning, because the hypervisor can allocate a real processor on which the guest OS has been spinning to other logical processors. The guest OS cannot recognize the processor allocation forms, because only the hypervisor manages all the real processors and allocates some of them to the guest OS. Therefore, the hypervisor determines the forms of the processor allocation to all VMs, and has to inform the guest OS of the processor allocation forms. According to the information, the guest OS can determine whether or not it should spin. We provide the following three new ways of handshaking, which are different as regards how the hypervisor informs the guest OS of the processor allocation forms. Only one of the three ways can be used.

(1) The hypervisor sets the data showing the processor allocation forms in the communication area between the guest OS and the hypervisor.

(2) The guest OS issues a hypervisor-call, which requires the hypervisor to inform the guest OS of the data showing the forms, in its Initial Program Loading (IPL) processes.

(3) The hypervisor supports a new external interrupt, which informs the guest OS of the forms.

2. **When to schedule a spinning logical processor again.** In responding to the hypervisor-call issued by a spinning logical processor to suspend the spin, the hypervisor has to determine when to schedule the spinning logical processor again. Below, we present three new methods by which the hypervisor can respond to the hypervisor-call. In them, the hypervisor sets the spinning logical processor in a wait state, and schedules the partner logical processors, which belong to the same VM as the spinning logical processor, before rescheduling the spinning logical processor. These methods are different as regards the timing at which the hypervisor schedules the spinning logical processor again.

(1) The hypervisor sets the spinning logical processor in a wait state due to spinning until one of its partners that are ready has been scheduled. This is called "Waiting for One Partner to be Despatched" (WOPD).

(2) The hypervisor sets the spinning logical processor in the wait state until all its partners that are ready have been scheduled. This is called "Waiting for All Partners to be Despatched" (WAPD).

(3) The hypervisor requeues the spinning logical processor at the last position in a

ready queue. This is called "Requeuing the Spinning Logical Processor" (RSLP).

We have experimentally implemented these three methods, and measured and evaluated their performance. The results show that the WAPD method has the best performance of the three for large workloads that have heavy spin activities in floating scheduling, where any real processor can schedule any logical processor.

We will be able to apply these methods to OSs that virtualize real processors. One such OS is Mach[11], which supports multiple tasks and multiple threads in a task. Each thread is a logical image of a real processor, and can run on any real processor in a multiprocessor. When threads manage system resources at a user level, the kernel, which schedules the threads, cannot know their behaviour. Therefore, we can find similar situations in them[12]. For example, a user level thread may spin for several reasons, such as to get a lock, or to wait for the processing of other threads. The spin may cause excessive spin because other threads are not always running. We will be able to apply the above methods to study how to schedule the spinning threads.

Section 2 describes conventional methods for reducing the spin of the guest OSs. Section 3 presents new methods for controlling the spin of guest OSs and their evaluation. Finally, our conclusion is presented in Section 4.

2. Conventional Methods for Reducing the Spins of Guest OSs

2.1 Spins of Application Programs

A process of an application program may spin (be busy wait) or block (relinquish a processor) to get a lock[3),4)]. There has been considerable research on how the spin times can be reduced. In a real machine there is only one OS to be run. In a single OS, several methods for reducing spin times have been adopted:

1. **Revised scheduling disciplines.** In a single OS it is shown how multiprogramming and data-dependent behavior affect the spin times of processes, and so complicate the choice between spinning (busy waiting) and blocking (relinquishing the processor). Scheduling disciplines revised to reduce the spin times of processes are given in Zahorjan and Lazowska[3]: In discipline A the scheduler never unschedules a process holding the lock. In discipline B it does not schedule a currently unscheduled spinning process unless the lock is free. In discipline C it does not reschedule a spinning process that happens to be descheduled because its time quantum has expired until all other processes have reached the synchronization points.

2. **Revised application programs.** The performance of application programs can be improved by a combination of spinning and blocking. That is, the application programs may spin below certain thresholds to get a lock, and if they cannot get it, they may block[4].

These two methods can help to provide us with effective concepts, but they cannot be applied unaltered to the hypervisor, for the following reasons:

1. **The hypervisor gives only hardware architecture to a guest OS (that is, an OS in a VM).** All the guest OSs are independent of each other. Similarly, the hypervisor and guest OSs are also basically independent except for the handshaking. Therefore, the hypervisor cannot recognize (a) what the guest OS is spinning for: it may be to go and get a lock, to wait for an event, to request a process, and so forth, (b) which logical processor of a VM gets and releases the lock, or when, and (c) which logical processor of a VM has finished synchronizing various events, or when.

2. **Application programs never know that they are running in a VM.** The hypervisor never knows the behaviour (locking, unlocking, spinning, and so forth) of the application programs under an OS. In turn, application programs never know whether or not they are running in a VM. Therefore, they do not call the hypervisor to improve their performance.

3. **An OS is usually made for a real machine, and not for a VM.** If we try to apply the above conventional methods to VMs, we have to redesign the logic of OSs for VMs. That is, the OS has to inform the hypervisor of the timing of its dynamic locking and unlocking, and the completion of events associated with its spins. The OS may be modified to spin under a threshold, and if it cannot detect an event, it may call the hypervisor. The threshold is dependent on the number of logical processors of VMs and that of the real host processors. This redesign and modification are not practical, however, because the hypervisor has to run currently available OSs, which are widely used now and designed for real machines.

Therefore, we have to consider further in

Section 3 how to reduce the spins of OSs in guest multiprocessors.

2.2 Spins of OSs

A kernel of an OS in multiprocessor mode will spin in disabled state, which means it cannot be interrupted, in the following cases:

1. **To get a lock.** A kernel of an OS has to get a lock to access shared resources (**Fig. 1**). When it succeeds in getting the lock, it can access the shared resources; otherwise, has to spin in disabled state. The OS sets a count for the spinning, and when the count expires, the OS may fail. It cannot relinquish its processor in a real machine environment. This is because it has to schedule its own processes, tasks, and threads. For this purpose, some instructions are provided to serialize the memory accesses of multiprocessors, namely, Compare & Swap (CS), Compare Double & Swap (CDS), and Test & Set (TS) instructions[10].

2. **To wait for a partner processor to complete a process.** An OS running on a processor (IP0) detects a process (P) running on a partner processor (IP1), which belong to the same multiprocessor (**Fig. 2**). The process (P) is a system process, and not a user process. Then the OS spins in disabled state on the processor (IP 0) to wait for the process (P) to complete on the partner processor (IP 1). At the completion of (P) some state is set by the partner processor (IP1). The OS spins in disabled state on the processor (IP0) until it detects the state. An example of this process (P) is queuing some entities. The OS sets a count for the spinning, and when the count expires the OS may fail. As in case 1, the OS cannot relinquish its processor in a real machine environment.

3. **To wait for a partner processor to complete a requested process.** An OS running on a processor (IP0) requests a partner processor (IP1), which belongs to the same multiprocessor, to carry out some process (**Fig. 3**). This process is a system process, and not a user process. Then, the OS spins in disabled state on the processor (IP0) to wait for the partner processor (IP1) to complete the process. The OS sets a count for the spinning, and when the

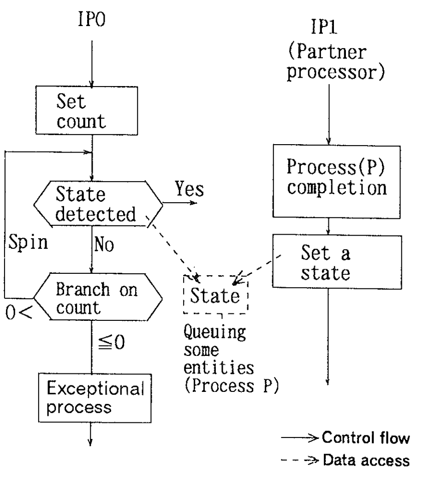


Fig. 2 Spin waiting for partner to complete its process.

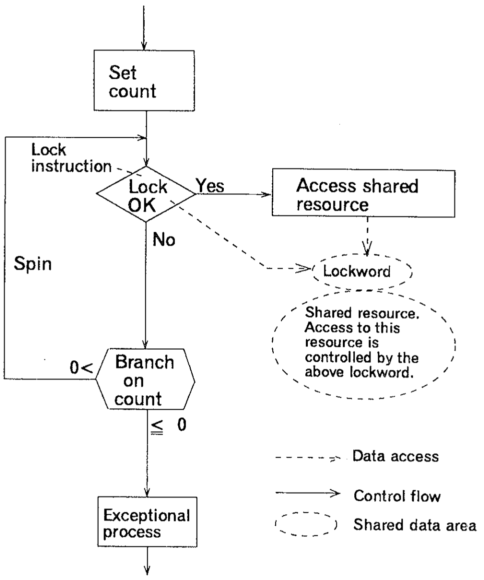


Fig. 1 Spin to get lock.

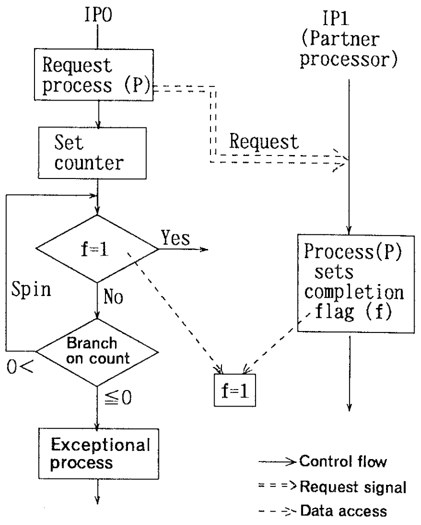


Fig. 3 Spin waiting for partner to complete requested process.

count expires, the OS may fail. An example of such a request is to purge hardware registers for address translation features. As in case 1, the OS cannot relinquish its processor in a real machine environment.

An OS is in disabled state in these spins, because the spins belong to the OS kernel, which runs in disabled and nonprocess mode. Certainly, these spins finish in a short time in a real machine environment, because a partner processor releases the shared resources or completes its waiting or requested process in a short time. On the other hand, in a VM environment, other partner logical processors, which belong to the same VM as the spinning logical processor, are not necessarily running, because real processors may run other VMs. Therefore, an OS may spin more times in a VM than in a real machine; that is, it may waste CPU time. Moreover, the OS may fail, because it may spin over the frequencies designated by itself.

On the other hand, a process of an application program is enabled for hardware interrupts. Therefore, the process will be suspended by the end of a time slice, which is detected by a timer interrupt, and other interrupts. Other processes will then be scheduled. Therefore, the system does not fail, even if the process spins over the frequencies designated by the application program.

2.3 Processor Allocation Forms

The hypervisor allocates the real processors of host multiprocessors to VMs, where guest OSs do not know the forms of the processor allocation. Below, we explain the conventional processor allocation forms, because spin problems are closely related to the processor allocation forms. There are two forms: processor dedication and processor sharing.

1. Processor Dedication

In processor dedication, all the logical processors of a guest multiprocessor (VM1) are dedicated one-to-one to real processors (**Fig. 4**)[9]. In this case, more real processors than there are logical processors of the guest multiprocessor (VM1) are required to run other VMs.

2. Processor Sharing

Conventionally, real processors are floatingly shared[9]. A logical processor of a VM may share a real processor with other logical processors of the VM or other VMs (**Fig. 5**). Moreover, a logical processor may be run on any real processors that have floating attributes. Figure 5 shows that the logical processor LIPi may be run on a real processor IPj for any (i, j), i=1, 2, 3, j=1, 2, 3.

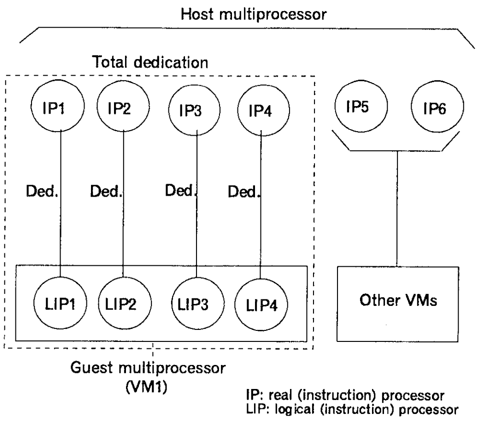

Host multiprocessor
Total dedication
IP1
IP2
IP3
IP4
IP5
IP6
Ded.
Ded.
Ded.
Ded.
LIP1
LIP2
LIP3
LIP4
Other VMs
Guest multiprocessor (VM1)
IP: real (instruction) processor
LIP: logical (instruction) processor

Fig. 4 Processor dedication.

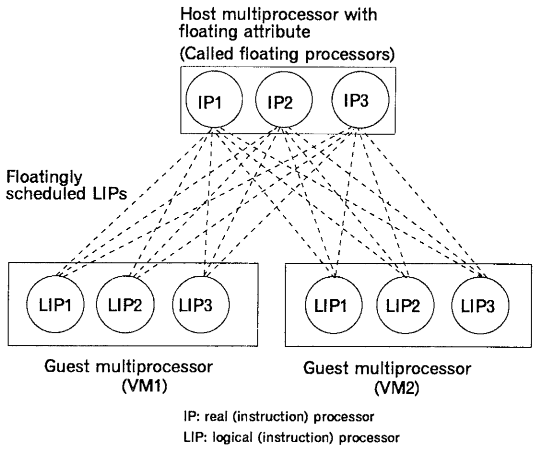

Host multiprocessor with floating attribute
(Called floating processors)
IP1
IP2
IP3
Floatingly scheduled LIPs
LIP1
LIP2
LIP3
LIP1
LIP2
LIP3
Guest multiprocessor (VM1)
Guest multiprocessor (VM2)
IP: real (instruction) processor
LIP: logical (instruction) processor

Fig. 5 Floatingly shared real processors.

2.4 Conventional Methods for Reducing the Spins of Guest OSs

Conventionally, handshaking is done to reduce the spin of a guest OS. The hypervisor and a guest OS are basically independent. The hypervisor gives only a hardware environment to a guest OS, and does not know which task and process the OS is scheduling. An OS runs in a VM as in a real machine. An OS usually does not recognize that it is running in a VM. Handshaking means that an OS shakes hands with the hypervisor and lets the OS know that it is running in a VM, and take some action to improve its performance in the VM. There are many kinds of handshaking. A hypervisor-call to suspend the spin is one of them: When it spins, an OS determines whether it is running in a VM, and if so, it calls the hypervisor, which

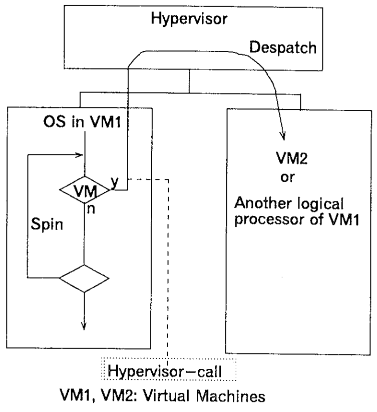


Fig. 6 Conventional Handshaking for suspending spin of OS.

will suspend it and schedule other logical processors of the VM (VM 1) or another VM2[2),7)] (**Fig. 6**).

The conventional methods have the following problems. Therefore, we have to consider further how to reduce the spin times of guest OSs.

1. **Calling the hypervisor to suspend the spin is not always better than spinning.** We have to give careful consideration to processor allocation forms, which mean dedicated or shared real processors by VMs. In the processor dedication, N real processors are dedicated one-to-one to N logical processors of an N-way guest multiprocessor (Fig. 4). Therefore, the spins perform better than the hypervisor-calls, because the spins expire in the VM in the same way as in a real machine. On the other hand, in processor sharing (Fig. 5), calling the hypervisor is better than spinning, because the hypervisor can allocate a real processor on which an OS has been spinning to other logical processors. OSs cannot recognize the processor allocation forms, because only the hypervisor manages all the real processors and allocates some of them to VMs. Therefore, the hypervisor has to inform a guest OS of the processor allocation forms. The guest OS can determine whether or not it should spin according to the information. Therefore, we have to consider how the hypervisor informs the guest OS of the forms of processor allocation.

2. **What the hypervisor should do in response to the hypervisor-call.** When it is in multiprocessor mode, a guest OS runs on multiple logical processors of a VM. When the guest OS is going to spin on a logical processor, the guest OS issues the hypervisor-call on it. In response to the hypervisor-call, the hypervisor should suspend it and schedule its partners, which are other logical processors belonging to the same VM as the spinning processor. This is because only the partners can clear the spin conditions. Therefore, the hypervisor has to schedule the partners, and determine when to schedule the spinning logical processor again. The hypervisor cannot know which logical processor of the VM clears the spin conditions and when.

We present, roughly speaking, two new methods for solving the above two problems. One contains three methods of handshaking, which informs guest OSs of processor allocation forms. On the basis of the information, the guest OSs determine whether or not they should spin. The other contains three methods, which present how and when the hypervisor should reschedule the spinning logical processor in relation to its partners, which are logical processors belonging to the same VM as the spinning logical processor.

3. New Spin Control for Guest Multiprocessors

3.1 How Hypervisor Informs Guest OSs of Processor Allocation Forms

According to the above discussion, in processor dedication (Fig. 4), the spin of a guest OS does not cause problems and is even desirable. On the other hand, in processor sharing, calling the hypervisor to suspend the spin is better than spinning. Therefore, we can present the following three types of handshaking based on the processor allocation forms:

1. **Setting a flag in a communication area between a guest OS and the hypervisor.** When it is IPLed, a guest OS calls the hypervisor to inform it of an address of the guest OS's communication area (**Fig. 7**). Then, the hypervisor

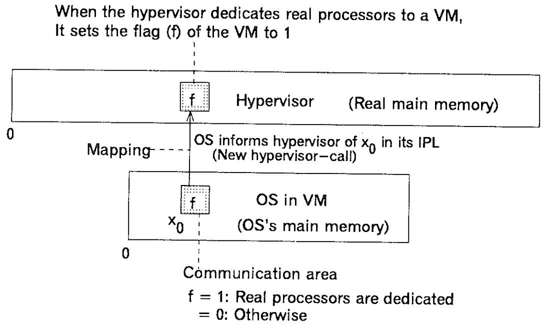

When the hypervisor dedicates real processors to a VM,
It sets the flag (f) of the VM to 1
f
Hypervisor (Real main memory)
0
Mapping
OS informs hypervisor of x0 in its IPL
(New hypervisor–call)
f
OS in VM
(OS's main memory)
x0
0
Communication area
f = 1: Real processors are dedicated
= 0: Otherwise

Fig. 7 Communication area between OS and hypervisor (New handshaking).

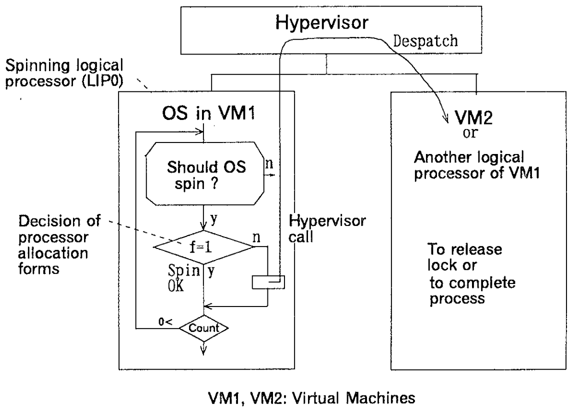

Hypervisor
Despatch
Spinning logical processor (LIP0)
OS in VM1
Should OS spin ?
n
y
Decision of processor allocation forms
f=1
n
Spin OK
y
0<
Count
Hypervisor call
VM2 or Another logical processor of VM1
To release lock or to complete process
VM1, VM2: Virtual Machines

Fig. 8 Determining whether to spin or to call hypervisor based on processor allocation forms (New handshaking).

sets a flag (f) in the area to control the spin of the guest OS:

f=1: The VM has dedicated real processors (Fig. 4). That is, real processors are dedicated one-to-one to logical processors of the VM. In this case, spins give a better performance than hypervisor-calls. Therefore, it is better for an OS in the VM to spin than to call the hypervisor. It depends on the policy of the OS whether or not the OS actually spins without the hypervisor-calls.

f=0: Otherwise (Fig. 5), real processors are shared. In this case, if the OS continues to spin, it may have an excessive spin; therefore, the OS should not continue to spin, and should call the hypervisor to suspend the spin.

Figure 8 shows how the guest OS uses the above-mentioned flag (f) as its spin control data. That is, when it has decided to spin, the guest OS additionally checks the flag f. It spins only if f=1; otherwise, it calls the hypervisor. The hypervisor despatches the partner logical processors that will release a lock or complete some processes expected or requested by the spinning logical processor (LIP0). When control is returned to the spinning logical processor (LIP0), the spin conditions are checked again. In most cases, the spin conditions are cleared.

Moreover, the hypervisor can dynamically change processor allocation forms by setting the flag f suitably.

2. **Returning codes of the hypervisor-call to the guest OS.** When it is difficult to get the above communication area, the hypervisor can inform the guest OS of the forms of processor allocation by means of return codes. The guest OS calls the hypervisor in its IPLing, and the hypervisor returns the codes to the guest OS. In this method, a change of processor allocation forms requires the guest OS to be IPLed again.

3. **Supporting a new external interrupt.** The hypervisor may support a new external interrupt to inform the guest OS of the processor allocation forms. The guest OS has to process the new external interrupt.

Of these three types of handshaking, the communication area method (1) is the most desirable, because it is simple, has a low OS overhead, and is the most flexible.

3.2 When Hypervisor Reschedules Spinning Logical Processors

In the processor dedication (Fig. 4) it is preferable for a guest OS to spin. In the processor sharing (Fig. 5) the guest OS should not spin, and should call the hypervisor. We discuss here what the hypervisor should do in response to the hypervisor-call.

A guest multiprocessor consists of multiple logical processors, which are called partners of each other. The hypervisor and the guest OSs are independent, and the hypervisor does not know which task and process the guest OSs are scheduling, or when. Therefore, the hypervisor does not know which logical processor can clear the spinning conditions, or when. We can say that the hypervisor has to suspend spinning logical processors, which are going to spin, in shared processor allocation, and schedule their partner logical processors, because only the partners can clear the spinning conditions. Therefore, we can present three methods that differ in the timing at which the hypervisor schedules the spinning logical processors again. We call these methods "Waiting for One Partner to be Despatched" (WOPD), "Waiting for All Partners to be Despatched" (WAPD), and "Requeue Spinning Logical Processors" (RSLP).

3.2.1 Methods for Rescheduling

1. WOPD Method

The hypervisor schedules logical processors of VMs by managing their states. When it is going to spin, a logical processor (LIP0) calls the hypervisor, which places it in "a wait state due to spinning." One of the partners of the logical processor (LIP0) may clear the spinning conditions of the logical processor (LIP0). Therefore, we can say in the first method that

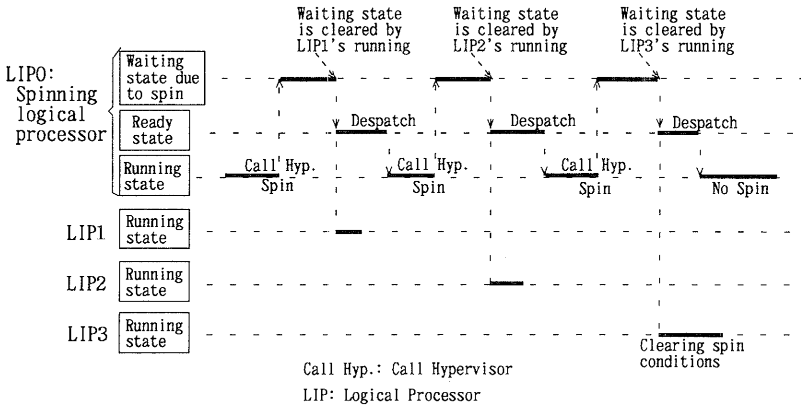

Waiting state is cleared by LIP1's running
Waiting state is cleared by LIP2's running
Waiting state is cleared by LIP3's running
LIP0: Spinning logical processor
Waiting state due to spin
Ready state
Running state
Despatch
Call Hyp.
Spin
No Spin
LIP1
LIP2
LIP3
Clearing spin conditions
Call Hyp.: Call Hypervisor
LIP: Logical Processor

Fig. 9 Example of LIP behaviour in WOPD method (LIP 3 clears spin conditions).

the hypervisor clears the wait state of the logical processor (LIP0) when it despatches at least one of the partners of the logical processor (LIP0). After the wait state of the LIP0 is cleared, the logical processor (LIP0) is set in ready state, and therefore will be despatched soon. We call this the "Waiting for One Partner to be Despatched" (WOPD) method.

Figure 9 shows an example of the behaviour of logical processors, one of which (LIP3) clears the spin conditions of LIP0 in the WOPD method. The figure shows that the spinning logical processor (LIP0) that is redespatched spins again, because the despatched partners (LIP1 and LIP2) do not clear the spin conditions. All the partners will be despatched before long, and the spin conditions of the logical processor (LIP0) will be cleared in the end. This method may not work well for guest OSs that have heavy spins, in which high activity levels are related to the spins, and therefore the activities of their kernels are also high. This is because the spin conditions are not always cleared when the spinning logical processor is rescheduled.

2. WAPD Method

The second method ensures that the spinning conditions of a spinning logical processor that issued a hypervisor-call are cleared by its partners. The hypervisor places the spinning logical processor in the wait state until all its partners that are in ready state have been despatched at least once since its last hypervisor-call. This is called the "Waiting for All Partners to be Despatched" (WAPD) method. It may decrease the response of the VM that contains the spinning logical processor. **Figure 10** shows an example of the behaviour of logical processors containing a spinning logical processor (LIP0). LIP0 issues the hypervisor-call at point t_1, and is set in a wait state due to spinning. The spinning conditions

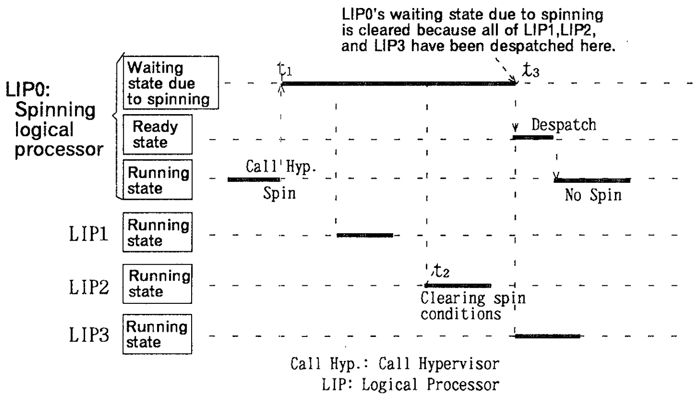

LIP0's waiting state due to spinning is cleared because all of LIP1, LIP2, and LIP3 have been despatched here.
t_1
t_3
t_2
LIP0: Spinning logical processor
Waiting state due to spinning
Ready state
Running state
Despatch
Call Hyp.
Spin
No Spin
LIP1
LIP2
LIP3
Clearing spin conditions
Call Hyp.: Call Hypervisor
LIP: Logical Processor

Fig. 10 Example of LIP behaviour in WAPD method (LIP 2 clears spinning conditions).

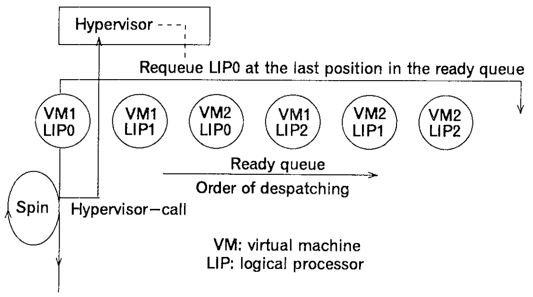


Fig. 11 Requeuing Spinning Logical Processors (RSLP).

are cleared at point t_2 by LIP2. At point t_3, logical processor 1 (LIP1), LIP2, and LIP3 have been despatched after the last hypervisor-call of LIP0. Therefore, LIP0 is set in ready state there, and is soon redespatched. LIP0 does not spin with the same spin conditions after being redespatched.

3. **RSLP Method**

The third method provides a queuing operation. That is, the hypervisor requeues a spinning logical processor (LIP0) that called the hypervisor at the last position in a ready queue (**Fig. 11**). When one of its partners is in a time-slice-end queue, LIP0 is enqueued into the last position of the time-slice-end queue. In floating scheduling, all ready logical processors are enqueued into the same ready queue. Therefore, the spinning conditions of LIP0 are probably cleared because all its partners are despatched before it.

3.2.2 Implementation

The implementation of the WAPD, WOPD, and RSLP methods is described below (see **Fig. 12-14**). The scheduler of the hypervisor implements these methods by setting real processors in disabled state and by using a lock related to ready queues. Therefore, the processing for the implementation is indivisible and exclusive.

1. **Structure of a ready queue.** A ready queue consists of three subqueues: a proper-ready queue, a time-slice-end queue, and an out-service queue. The proper-ready queue contains ready logical processors that can be run and despatched. Precisely speaking, when it is in a wait state due to spinning, an LIP is enqueued into the proper-ready queue. This is because it takes more overhead to make its own queue and enqueue it into the queue than to enqueue it into the ready queue. The LIP cannot be despatched until the wait state is cleared. When a running logical processor has consumed a full time slice, it is enqueued into the time-slice-end queue. When it has consumed a designated service quantity of CPU busy time, it is enqueued into the out-service queue. The designated service quantity is determined by the service ratios of VMs. For example, in two VMs we will specify VM1 : VM2=30 : 70. The designated service quantities of VM1 and VM2 are 30 and 70, respectively.

A scheduler of the hypervisor enqueues all the logical processors in the time-slice-end queue into the proper-ready queue, when the proper-ready queue is empty. In that case, when the time-slice-end queue is empty, the scheduler enqueues all the logical processors in the out-service queue into the proper-ready queue.

2. **Data areas.** The hypervisor defines the following control blocks in its memory:

(1) Virtual Machine Control Block (VMCB). This control block manages a virtual machine and contains its states.

(2) Logical Processor Control Block (LPCB). This control block manages a logical processor and contains its states.

For an N-way guest multiprocessor the hypervisor defines one VMCB and N LPCBs, where each LPCB contains the address of the next LPCB and an address of the VMCB that contains the address of the first LPCB (**Fig. 12**).

(3) An LPCB, which defines a logical processor (i), contains the following data areas that are used to indicate the states of the logical processor:

A SPin Wait state flag (ASPWi) : This consists of one bit.

ASPWi=1: The logical processor (i) is waiting for all its partners that are ready and runnable to be despatched.

Despatching MASK (DMASKi) : This consists

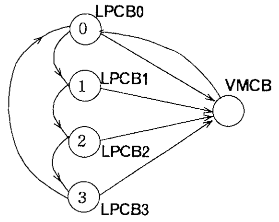

LPCB0
0
1
LPCB1
VMCB
2
LPCB2
3
LPCB3
VMCB: virtual machine control block
LPCB: Logical processor control block

Fig. 12 Chain of LPCBs in VM.

of N bits for an N-way guest multiprocessor.

DMASKi(j) means its j-th entry; this corresponds to the logical processor (j), which is a partner of the logical processor (i) when j≠i.

DMASKi(j)=1, where j≠i: A logical processor (j), which is not the logical processor (i), is ready and not yet despatched to an N-way guest multiprocessor j=0, 1, ···, (N−1), except i. In this case, the logical processor (i) is waiting for the logical processor (j) to be despatched.

(4) A VMCB contains the following counter:

Global SPin Wait state counter (GSPW): The number of LPCBs whose ASPW=1.

3. **WAPD method (Fig. 13 and 14).** For a hypervisor-call of a logical processor (i) (LIP(i)), the hypervisor checks the states of all its partners and updates its $DMASK_i$ (①, ③ in Fig. 14): For all j (≠i), when LIP(j) is ready, not in a wait state due to spinning, and not yet despatched,

DMASKi(j)←1.

This means that LIP(i) must wait for LIP(j) to

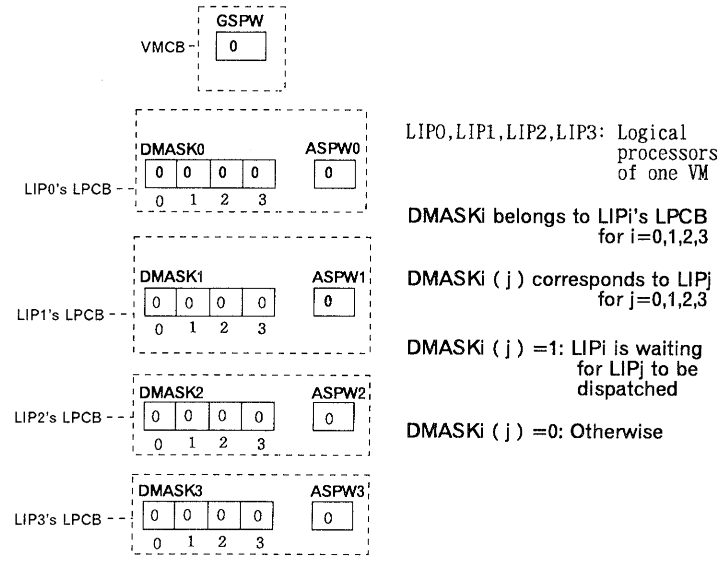

GSPW
0
VMCB
DMASK0
ASPW0
LIP0's LPCB
DMASK1
ASPW1
LIP1's LPCB
DMASK2
ASPW2
LIP2's LPCB
DMASK3
ASPW3
LIP3's LPCB
LIP0,LIP1,LIP2,LIP3: Logical processors of one VM
DMASKi belongs to LIPi's LPCB for i=0,1,2,3
DMASKi (j) corresponds to LIPj for j=0,1,2,3
DMASKi (j) =1: LIPi is waiting for LIPj to be dispatched
DMASKi (j) =0: Otherwise

Fig. 13 Control data for WAPD.

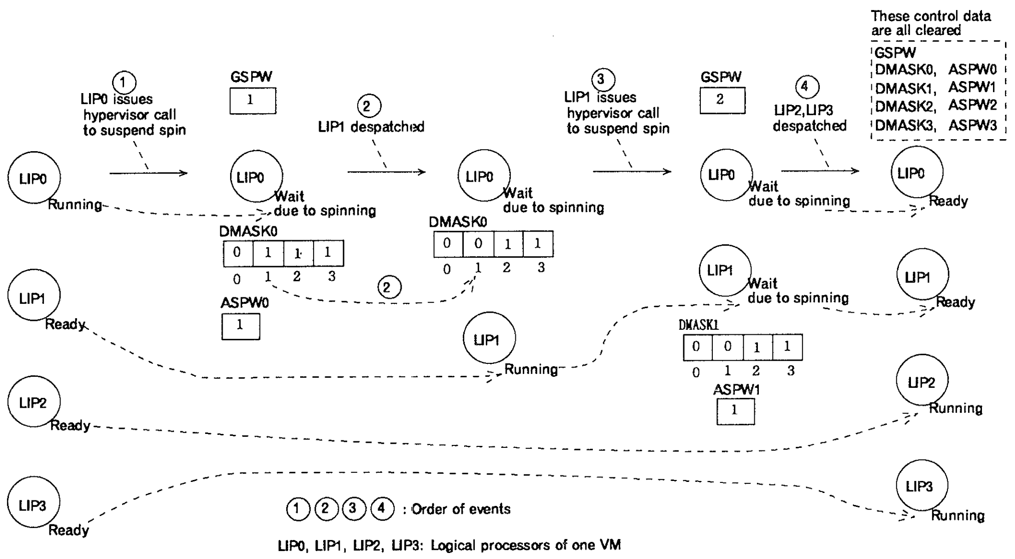


Fig. 14 State transition of logical processors in WAPD.

be despatched. When LIP(j) exists,

ASPW of the LPCB of LIP(i)←1, and

GSPW of the VM, which contains LIP(i), is increased by 1.

When the hypervisor despatches LIP(j) (its ASPW of LPCB=0) (LIP1, ② in Fig. 14) of a VM, and GSPW of the VM≠0, that is, at least one logical processor of the VM is in a wait state due to spinning, the wait state of the one logical processor may be cleared by despatching LIP(j). Therefore, the hypervisor checks the DMASKs of all partners of LIP(j) : For any LIP(k)

(k≠j) if DMASKk(j)=1 (this means that LIP (k) is waiting for LIP(j) to be despatched), the DMASKk(j)←0 (③ in Fig. 14) (that is, LIP(j) has been despatched), for k=0, 1, ⋯, N−1, except j.

Moreover, if DMASKk changes from≠0 to 0 as a result of the above (④ in Fig. 14), it means that all partners of LIP(k) have been despatched. Therefore, the wait state of LIP(k) is cleared, that is,

ASPW of LPCB of the LIP(k)←0, and

GSPW of the VM is decreased by 1.

4. **WOPD method.** When to set an LIP in a wait state due to spinning: Responding to a hypervisor-call issued by a logical processor (LIP0) of a VM to suspend its spin, the hypervisor checks the states of all the partners of LIP0. If one of its partners is in running state, no action is taken. If one of its partners is ready and not in a wait state due to spinning, the hypervisor sets LIP0 in a wait state due to spinning.

When to clear an LIP's wait state due to spinning: When it despatches a VM's logical processor (LIP1) that is ready and not in a wait state due to spinning, the hypervisor checks the state of the VM. If one of the logical processors of the VM is in a wait state due to spinning, the hypervisor clears the wait states of all the partners of LIP1.

5. **RSLP method.** Responding to a hypervisor-call issued by a logical processor (LIP0) to suspend its spin, the hypervisor enqueues LIP0 at the last position in the ready queue. To be more precise, the hypervisor checks the states of all the partners of LIP0. When one of them is in an out-service queue, the hypervisor enqueues LIP0 at the last position in the out-service queue. This is because LIP0 should run after all its partners. When one of them is in a time-slice-end queue, the hypervisor enqueues LIP0 at the last position in the time-slice-end queue. Otherwise, the hypervisor enqueues LIP0 at the last position in the proper ready queue.

3.3 Evaluation

3.3.1 Qualitative Evaluation

Here, we evaluate the three methods qualitatively.

1. **Waiting time.** The WOPD sets a spinning LIP in a wait state due to spinning until one of its partners that are ready has been despatched. WAPD sets a spinning LIP in a wait state due to spinning until all its partners that are ready have been despatched. RSLP enqueues a spinning LIP at the last position in a ready queue. The queuing makes the LIP wait for another LIP of another VM. Therefore, the waiting time of WOPD is shorter than that of WAPD and RSLP.

2. **Suppression of excessive spin.** WOPD may not be able to suppress excessive spin of a guest OS because it allows the hypervisor to redespatch a spinning LIP of the guest OS before its spin conditions are cleared. WAPD and RSLP certainly suppress excessive spin of any guest OSs, because they allow the hypervisor to redespatch the LIP only after its spin conditions are almost cleared.

3. **Influence of the number of VMs.** RSLP enqueues a spinning LIP at the last position in a ready queue. The queuing makes the LIP wait for another LIP of another VM. Therefore, when the number of VMs is large, the queuing will badly affect the performance of the VM that contains the spinning LIP. WAPD and WOPD enqueue the spinning LIP into a ready queue in normal order and set its flag for the wait-state-due-to-spinning so that it is on. When all its partners that are ready in WAPD or one of its partners in WOPD have been despatched, the LIP can be redespatched at once. Therefore, except that VMs normally compete with each other for real processors, the number of the VMs does not affect the timing at which the spinning LIP is redespatched.

4. **Overhead.** WAPD will have a slightly larger overhead than RSLP, because it has to manage the wait state due to spinning. WOPD will have a smaller overhead than WAPD, because its management of a wait state due to spinning is simpler than that of WAPD.

3.3.2 Quantitative Evaluation

We experimentally implemented the three

Table 1 Workloads.

#	Workload
1	Batch (4800 jobs) Compile, Link, Go
2	Batch (4000 jobs) Compile, Link, Go
3	Batch (3600 jobs) Compile, Link, Go
4	Heavy I/O requests issuing frequent EXCPs
5	TSS:Total 100 terminals (generated transactions)
6	TSS:Total 400 terminals (generated transactions)
7	Batch (endless test jobs issuing frequent SVCs)

All VMs have multiple virtual Storage OSs running.

CPU: 40 – 50 MIPs/real processor

EXCP: EXecute channel program

methods (WOPD, WAPD, RSLP), and measured the system performance to compare and evaluate them. **Table 1** shows the workloads of batch jobs (# 1, # 2, # 3,), a heavy I/O request job (# 4), Time Sharing System (TSS) jobs (# 5, # 6), and endless batch jobs issuing frequent SVCs (# 7). These workloads are comparable to those of real user systems.

We evaluated the following items:

1. The number of hypervisor-calls needed to suspend a spin. The number shows the activity level of OS kernel associated with the spin.
2. Internal throughput ratios (ITR), which means the total number of transactions processed per unit of real processor busy time (1 s).
3. External throughput ratios (ETR), which means the total number of transactions processed per unit of real time (1 s).
4. CPU overhead of the hypervisor. (This is the overhead of the scheduler of the hypervisor.)

The measurement conditions were as follows:

1. The host processors were large mainframes, which were two-way or four-way shared-main-memory multiprocessors.
2. Two or four VMs were used.
3. Real processors were allocated in floatingly shared forms.

We show the measurement data in **Tables 2, 3 and 4**.

The measurement methods were as follows:

1. In Table 2, batch jobs were measured once for each of the three methods (WOPD, WAPD, and RSLP), as shown in # 1 and # 4. The workload of the heavy I/O request was measured twice for each of the three methods, as shown in # 2 and # 3.
2. In Tables 3 and 4, batch jobs were measured once for each of the two methods (WAPD and RSLP), as shown in case-1 and case-4. The workload of the heavy I/O request was measured twice for each of the two methods, as shown in case-2 and case-3.
3. The data in cases 5-7 in Tables 3 and 4

Table 2 Comparison of three methods in floating scheduling.

#	Items		WOPD	WAPD	RSLP	Workloads & Environment
1	Hyp.Call	*1	–*5	893.27	680.39	#3 (Batch) in Table 1
	Guest ρ	*2	95.69	95.04	95.92	3–way host multiprocessor
	ETR	*3	6.91	6.85	6.25	3 VMs (each VM is 3–way)
	OVH	*4	1.73	1.99	1.76	
2	Hyp.Call		210.58	234.53	215.42	#4 (Heavy I/O) in Table 1
	Guest ρ		45.07	45.93	45.75	4–way host multiprocessor
	ETR		23.78	23.81	23.85	3 VMs (each VM is 4–way)
	ITR		33.71	32.79	33.08	Interval: 5 minutes
	OVH		14.15	14.61	14.48	
3	Hyp.Call		203.13	228.91	210.21	
	Guest ρ		44.92	45.89	45.45	Ditto
	ETR		23.77	23.79	23.54	
	ITR		33.76	32.78	32.89	
	OVH		14.17	14.62	14.39	
4	Hyp.Call		70.42	62.26	52.08	#7 (Batch) in Table 1
	Guest ρ		75.38	76.31	76.26	4–way host multiprocessor
	ETR		1.90	1.92	1.92	3 VMs (each VM is 4–way)
	ITR		2.26	2.26	2.25	Interval: 15 minutes
	OVH		4.84	4.77	4.95	

*1: Number of hypervisor–calls / CPU busy time (sec)
*2: Guest CPU utilization % / real processor
*3: External Throughput Ratio: Ends / Real time (sec)
*4: Overhead: Scheduler CPU utilization (%)
*5: Not measured

Table 3 Comparison of WAPD and RSLP in floating Scheduling.

Case	Items	WAPD	RSLP	Workload
	Hyp. Call *1	1,284.18	1,328.74	#1 (Batch) in Table 1
1	Guest ρ *2	43.75	43.03	4-way host multiprocessor
	ETR *3	4.729	4.585	4 VMs (each is 4-way)
	ITR *4	9.690	9.566	
	OVHD *5	2.48	2.37	
	Hyp. Call	267.77	274.21	#4 (Heavy I/O) in Table 1
2	Guest ρ	21.20	29.62	4-way host multiprocessor
	ETR	16.324	16.631	2 VMs (each is 4-way)
	ITR	34.265	34.658	ETR,ITR:
	OVHD	8.92	8.68	EXCP *6 counts/sec
	Hyp. Call	103.55	103.59	#4 (Heavy I/O) in Table 1
3	Guest ρ	49.01	49.72	2-way host multiprocessor
	ETR	16.415	16.722	2 VMs (each is 2-way)
	ITR	26.379	26.636	ETR,ITR:
	OVHD	5.86	5.67	EXCP counts/sec
	Hyp. Call	18.43	18.51 *7	#2 (Batch) in Table 2
4	Guest ρ	96.63	96.65	2-way host multiprocessor
	ETR	9.017	9.034	2 VMs (each is 2-way)
	ITR	9.279	9.296	
	OVHD	0.25	0.23	
	Hyp. Call	1,484.42	1,474.52	#5 (TSS) in Table 1
5	Guest ρ	56.34	56.10	2-way host multiprocessor
	ETR	53.438	53.438	2 VMs (each is 2-way)
	ITR	107.652	108.196	
	OVHD	1.20	1.15	
	Hyp. Call	12,091.00	8,037.23	#6 (TSS) in Table 1
6	Guest ρ	99.37	99.08	2-way host multiprocessor
	ETR	53.878	52.333	2 VMs (each is 2-way)
	ITR	53.911	52.370	
	OVHD	0.46	0.34	
	Hyp. Call	64,292.55	57,472.43	#6 (TSS) in Table 1
7	Guest ρ	97.02	97.24	4-way host multiprocessor
	ETR	76.839	73.735	2 VMs (each is 4-way)
	ITR	76.924	73.808	
	OVHD	1.28	1.01	

All VMs have multiple virtual storage OSs running.
*1: Hypervisor call: times/CPU busy time (sec)
*2: Guest CPU utilization % /real processor
*3: External throughput ratio: ends/real time (sec)
*4: Internal throughput ratio: ends/CPU busy time (sec)
*5: Overhead: scheduler CPU utilization (%)
*6: EXCP: EXecute channel program
*7: Estimated value

Table 4 Number of hypervisor-calls and throughputs in floating scheduling.

Case	Hyp. Call (WAPD)	Hyp. Call (RSLP)	ETR (WAPD)	ETR (RSLP)
4	18.43	18.51	9.017	9.034
3	103.55	103.59	16.415	16.722
2	267.77	274.21	16.324	16.631
1	*1,284.86*	*1,328.74*	*4.729*	*4.585*
5	*1,484.42*	*1,474.52*	*53.438*	*53.438*
6	*12,091.00*	*8,037.23*	*53.878*	*52.333*
7	*64,292.55*	*57,472.43*	*76.839*	*73.735*

Italic data show heavy spins.

are averages of the data sampled every two minutes during 10-14 minutes of measurement. Their workloads were TSS jobs (#5, #6 in Table 1). We used a terminal simulator, which generates transactions with edit commands and subcommands. We describe only the averages, because the sample data were almost the same and their deviations were very small (less than 0.1 in ETR and ITR) in the same conditions.

Table 2 shows that the three methods (WOPD, WAPD, and RSLP) have comparable performances when the workloads are small and the frequency of hypervisor-calls is low. It may be possible to say that WOPD has the best performance of the three because it has the largest number of best ETRs and ITRs. It is not preferable for large workloads, however.

We measured larger workloads. In WOPD we experienced many excessive spin-loops for large workloads. For example, there were excessive spin loops in five VMs, where all the VMs were 5-way multiprocessors, in a five-way host multiprocessor system, and in floating scheduling. Guest OSs were multiple virtual storage OSs. Three of the five OSs in the VMs had excessive spin loops and set several (two or three) logical processors off-line. Only after they had run for many hours, or sometimes for over ten hours, did the guest OSs detect excessive spin loops.

We consider the cause of the excessive spin loop to be as follows: In WOPD, the spinning logical processor does not wait for all its partners that are ready to be despatched. This is to avoid waiting too long, and to avoid decreasing the system performance. For this reason, the spinning logical processor is possibly redespatched before its partner, which clears its spin conditions, is despatched. The more guest OSs spin, the more the hypervisor redespatches their spinning logical processors before their spinning conditions are cleared. As a result, the guest OSs have excessive spin loops. Therefore, excessive spins are caused by heavy spin activities of guest OSs.

We exclude WOPD from performance measurement because a guest OS sets its processor off-line when detecting excessive spins; therefore, we cannot use its performance data. We focus our evaluation on the remaining two methods (WAPD and RSLP). Table 3 shows the measurement data for the two methods. The workloads contain batch jobs, a heavy I/O request job, and TSS jobs. Guest OSs are multiple virtual storage OSs.

A guest OS issues hypervisor-calls in its spin. Therefore, the high frequency of its hypervisor-call to suspend a spin signifies its heavy spin activities. In this sense, Cases 1 (batch), and 5-7 (TSS) in Table 3 have heavy spin activities.

Table 3 shows that in floating scheduling the two methods (WAPD and RSLP) have comparable performance. Table 4 focuses on the number of hypervisor-calls, which makes the hypervisor shake hands with the guest OSs to avoid their excessive spins, and system throughput. These tables show that:

1. WAPD has a slightly (0.02-0.27%) larger overhead than RSLP.
2. WAPD has a comparable performance to RSLP when the number of hypervisor-calls is small (Table 4, Case 4, 3, 2).
3. WAPD has a better performance than RSLP when the number of hypervisor-calls is high (Table 4, cases 1, 5-7). This means that for large workloads that have heavy spins, WAPD has a better performance than RSLP.

4. Conclusion

When all the logical processors of a VM have one-to-one-dedicated real processors, a spin of an OS in the VM works as well as in a real machine. When a real processor is shared by logical processors of VMs, it is better for OSs in the VMs to call the hypervisor to suspend their disabled spin. We have presented new handshakes whereby the hypervisor dynamically informs guest OSs of the processor allocation forms (i.e., dedicated or shared), and on the basis of the information the guest OSs determine whether or not they should spin.

Moreover, we have presented how the hypervisor should react in response to a hypervisor-call to suspend a spin of an OS in a VM. The hypervisor does not know which logical processor of the VM clears the spinning conditions, or when, because it is the guest OS that schedules the logical processors. Therefore, the hypervisor adopts one of the following three methods in response to the hypervisor-call:

1. Waiting for one partner ready to be despatched (WOPD). The hypervisor sets the spinning logical processor in the wait state until at least one of its partners that are ready has been despatched.
2. Waiting for all partners ready to be despatched (WAPD). The hypervisor sets the spinning logical processor in the wait state until all its partners that are ready have been despatched.
3. The hypervisor requeues the spinning logical processor (RSLP). The hypervisor enqueues the spinning logical processor again at the last position in a ready queue.

We have implemented these methods, and measured and evaluated them. As a result, we found that

1. For small workloads the three methods have comparable performances.
2. According to experiments, WOPD cannot suppress an excessive spin of a guest OS for large workloads that have heavy spin activities.
3. WAPD has a slightly (0.02-0.27%) larger overhead than RSLP.
4. For large workloads, WAPD has a slight-

ly (the difference of ETR is 3-4%) better performance than RSLP.

Acknowledgement We thank Mr. K. Une and Mr. K. Suzuki of General Purpose Computer Division (GPC), Hitachi Ltd., for their cooperation in performance measurement. We also thank Mr. T. Kuwabara and Mr. Y. Ohshima of GPC for their management of this work.

References

1) Goldberg, R. P.: Architectural Principles for Virtual Computer Systems, Ph.D. dissertation, Div. Eng. Appl. Phys., Harvard Univ., Cambridge, MA (1972).

2) Borden, T. L. et al.: Multiple Operating Systems on One Processor Complex, *IBM Syst. J.*, Vol. 28, No. 1, pp. 104-123 (1989).

3) Zahorjan, J. and Lazowska, E. D.: Spinning Versus Blocking in Parallel Systems with Uncertainty, *Performance of Distributed and Parallel Systems,* IFIP, pp. 455-472 (1989).

4) Karlin, A. R.: Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor, *ACM Symposium on Operating Systems Principles (13)*, Vol. 25, No. 5, pp. 41-55 (Oct. 1991).

5) Doran, R. W. et al.: Amdahl Multiple-Domain Architecture, *Computer,* pp. 20-28 (Oct. 1988).

6) Bean, G. H. et al.: Logical Resource Partitioning of a Data Processing System, IBM, PR/SM™ Patent, Priority, July 29 (1987).

7) Umeno, H. and Ohmachi, K.: A Method for Supporting Virtual Machine Multiprocessor Systems, *Proceedings of the 19th Annual Convention IPS Japan,* pp. 265-266 (1978).

8) IBM: IBM System/370-XA Start Interpretive Execution, SA22-7095.

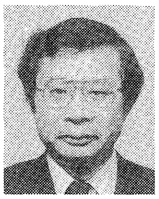
9) HDS: EX™ Series Multiple Logical Processor Feature™ (MLPF™) User's Guide, FE-91EX036-1.

10) IBM: System/370-XA Principles of Operation, SA22-7085.

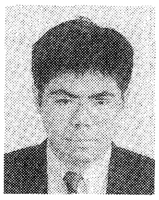
11) Open Software Foundation and Carnegie Mellon University: Mach 3 Kernel Interface (1991).

12) Marsh, B. D., Scott, M. L. et al.: First-Class User-Level Threads, *ACM Operating System Review,* Vol. 25, No. 5, pp. 110-121 (1991).

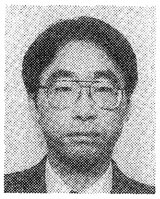
(Received May 6, 1994)
(Accepted November 17, 1994)



Hidenori Umeno was born in Ohita, in 1947. He received the B. S. in mathematics from Kyushu University in 1970. From 1970 to 1976, he was with Central Research Laboratory, Hitachi Ltd., where he made researches in productivity improvement of compilers. From 1976 to 1993, he was with Systems Development Laboratory, Hitachi Ltd., where he made researches in performance and reliability improvement of virtual machines, file systems, and operating systems. Since 1993, he has been with General Purpose Computer Division, Hitachi Ltd., where he has been engaged in the development of logical partition systems of mainframes. His main research fields are performance and function improvement of virtual machines, logical partition systems, operating systems, and computer architectures. He received a best paper award of Information Processing Society of Japan (IPSJ) in 1982. Since 1991, he has been a part-time instructor of Musasi Institute of Technology. He is a member of IPSJ and ACM, and an affiliate of IEEE Computer Society.



Hideaki Amano was born in Kyushu in 1968. He received the B. Eng. in Electrical Engineering from Fukuoka University in 1992. Since 1992, he has been with General Purpose Computer Division, Hitachi Ltd., where he is mainly developing virtual machines of general purpose computers.



Keiji Saijo was born in 1960. He received the B. Eng. and M. Eng. in Nucleonics from Nagoya University in 1982 and in 1984, respectively. Since 1984, he has been with Software Development Center, Hitachi Ltd., where he has been engaged in the development of virtual machine sysems and logical partition systems.