

New Method for Dispatching Waiting Logical Processors in Virtual Machine System

Hiddenori Umeno,
Dep. of CS, Fac. of Eng,
Kumamoto University
umeno@cs.kumamoto-u.ac.jp

Masato Kiyama,
Dep. of CS, Fac. of Eng,
Kumamoto University
masato@cs.kumamoto-u.ac.jp

Takafumi Fukunaga
Kumamoto Pref. College of
Technology
fukunaga@kumamoto-pct.ac.jp

Takashige Kubo,
Hitachi System & Service,
Ltd.
taka11k@dolphin.ocn.ne.jp

Abstract

A virtual machine system can run multiple conventional operating systems (OSs) in a single real host computer. A virtual machine is a logical computer with almost the same architecture as the host, and may contain several logical processors. A hypervisor is a control program to control this virtual machine system. Traditionally, the hypervisor has to receive an I/O interrupt pending for a waiting logical processor, and to simulate the I/O interrupt, consequently incurring the simulation overhead of the I/O interrupt. To avoid this overhead we present a new method which introduces a new self-wait state different from the conventional wait state, presents a new instruction for the hypervisor to detect the I/O interrupts pending for the logical processors in the self-wait state, and dispatches those logical processors on the detection ahead of the ready queue. This new method has eliminated the simulation overhead of those I/O interrupts, and enhanced the system performance to the near native.

1. Introduction

A virtual machine system can concurrently run multiple operating systems (OSs) in a single real host computer system [1], [2], [3], [4]. The virtual machine system has been used in mainframes, nowadays even in personal computers, as VMware [5], Virtual PC, etc. A Virtual Machine (VM), which may be called a guest, is a logical computer which has almost the same architecture as the real host computer. The real host computer here may be a uniprocessor (UP) system or a symmetric multiprocessor (SMP) system. As the SMP has multiple real processors, each of which has the same functions and shares its main memory, so the VM may have multiple logical processors in the same way,

which is called a guest multiprocessor system.

Recently, many virtual machine systems are supported in the personal computers with x86 architecture. Some of them give users only the partial hardware architecture of the real host computer, providing users with the simplified logical devices [7], [8]. On the other hand, some of them give users the full architecture by simulating all the guest I/Os [5]. As a result they have no capability of running the existing legacy operating systems intact, or they have the high I/O simulation overhead.

We address the VMs that have almost the same architecture as the real host computer and that are capable of running multiple existing OSs concurrently and with high performance due to the hardware assists for virtual machines [2], [3], [4]. They will be run as production systems.

This paper presents the virtual machine system's performance improvement, its development, and its evaluation. A hypervisor is the program which controls the whole virtual machine system, allocates real resources (real processors, real memory, and real I/O devices) to each VM, and schedules its logical processors.

The scheduler of the hypervisor schedules the logical processors in the similar way as a traditional scheduler of a usual OS schedules its processes. That is, it gives each logical processor one of the three states, which are running state, wait state, and ready state [9].

Generally, an interrupt to the hypervisor is called a host interrupt, which may be caused by the hypervisor itself such as a host timer interrupt, or may be caused by a VM and should be reported to the target VM. When a VM has the multiple logical processors, its I/O interrupt is reported to any one of them.

Conventionally, an I/O interrupt pending for a VM is directly executed by hardware when its target logical processor is running, therefore, it causes no

simulation overhead in this case [2], [4]. On the other hand, when all the target logical processors of that interrupt are in the wait state, it becomes a host interrupt. That is, the scheduler of the hypervisor recognizes the pending guest I/O interrupt through the host interrupt. Then, it simulates that interrupt to its target logical processor, and changes its state from the wait state to the ready state. This conventional scheduling is similar to the method a usual OS takes to schedule its processes [9], and causes the high simulation overhead of that guest I/O interrupt.

The hypervisor runs in the native mode, while a VM runs in the vm mode, which is not only a software mode but also a hardware supported mode [2], [3]. Changing the mode from vm to native or vice versa takes more than 100 times a usual instruction execution time. This is because the mode change contains the setting of many hardware control registers, resulting in suspending the hardware parallel execution temporally.

Conventionally an I/O interrupt pending for a target VM may cause a host interrupt when it (i.e. the target VM) is not running, while other VMs may be running and then intercepted by that host interrupt. If many guest I/O interrupts occur as host interrupts, they cause many mode changes from vm to native and vice versa, causing high transition overhead.

The above-mentioned I/O simulation overhead will be very high when a guest OS (i.e. an OS in a VM) runs heavy I/O workloads [5]. As an example of I/O simulation overhead we measured the I/O performance of VMware ESX Server, showing it in Figure 1 which indicates the VM I/O performance falls to 40 % - 60 % of the native performance in such workloads (i.e. the iotzone random read). This overhead is due to both the guest I/O instruction simulation and the guest I/O interrupt simulation.

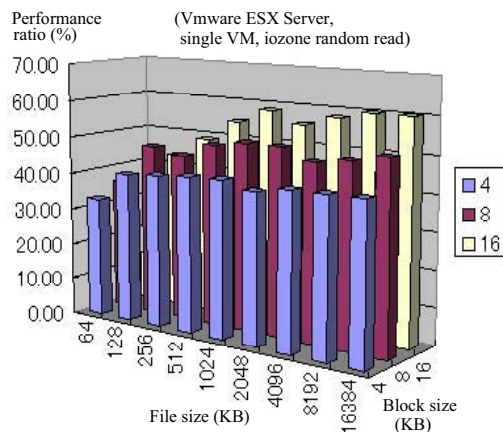


Fig. 1. VM I/O performance to native

To reduce those overheads, which consist of the

guest I/O interrupt simulation and the mode change overhead, we present a new method which introduces a new self-wait state different from the conventional wait state, presents a new instruction for the hypervisor to detect the I/O interrupts pending for the logical processors in the self-wait state, dispatches those logical processors in response to the detection ahead of the regular ready queue, and scans those logical processors to find out workloads by using the new instruction when workloads have been exhausted.

The detection and the finding workloads are carried out without any host I/O interrupt pending for VMs. This new method reduces those overheads and improves the performance of the virtual machine system to the near native even for heavy I/O workloads.

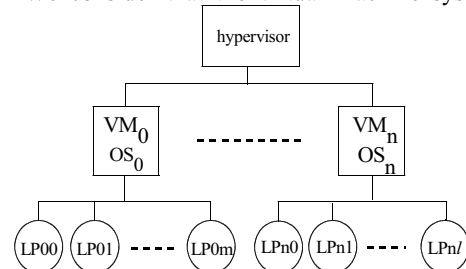
The system administrators will desire to run several VMs. The high I/O simulation overhead increases CPU utilization of the VM with heavy I/O workloads. This overhead reduces the CPU utilization for other VMs. Our method reduces I/O simulation overhead and gives benefit to the other VMs.

We state the concept of the virtual machine system in section 2, propose a new scheduling method in section 3, state the conventional related art and architecture in section 4, provide the experimental performance evaluation in section 5, and conclusions in section 6.

2. Virtual machine system

Figure 2 shows the configuration of the virtual machine system we are now considering, where virtual machines VM0, VM1, ..., VMn are defined, each of which (i.e. VMi) has logical processors LPi0, LPi1, ..., which share their main memory area. A virtual machine may have any number of logical processors. The number of logical processors may vary in each VM.

We consider that the virtual machine systems are



VMi: Virtual Machine - i, i=0, 1, ..., n

OSi: Operating System - i, i=0, 1, ..., n

LPij: Logical Processor - j of VMi

Fig. 2. Concept of virtual machine system

used for everyday production, where each VM has the high performance except for several testing VMs which may coexist in the system.

In mainframe computers the virtual machine system hypervisor runs in the native mode, while a VM runs in the vm mode supported by hardware [1], [2], [3]. When a logical processor has issued a system wait instruction, the overall system must not fall into the wait state due to it. Because of this, hardware detects such an instruction issued and suppresses its execution, and transfers control to the hypervisor which then puts the logical processor in the wait state and schedules the other logical processors.

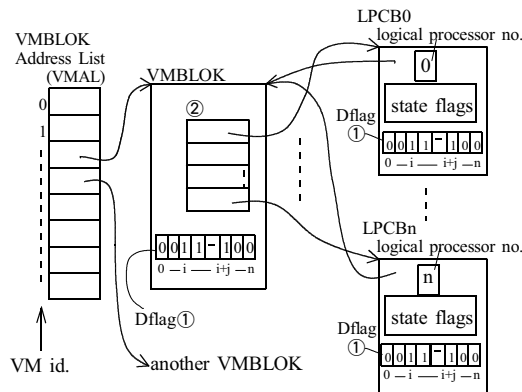


Fig. 3. Example of VM control blocks

On the other hand personal computers with IA32 architecture have no virtual machine hardware features such as the vm mode and the interception of the sensitive instructions. The hypervisor on those personal computers gives users the full architecture of the host by translating the guest binary codes containing sensitive instructions, tracking the guest segmented architecture by making shadow descriptor tables [7], which are modified copies of the guest descriptor tables. These simulations reduce the guest performance greatly for many processings such as the I/O processings shown in Figure 1.

Figure 3 shows an example of a configuration of various virtual machine control blocks, such as a VMBLOCK which stands for a control block for a VM, $LPCBi$ ($i = 0, 1, \dots, n$) each of which stands for a logical processor thereof and contains state flags for managing its states, and a VMAL which stands for address list for VMBLOCKS. The hypervisor creates and manages these control blocks in main memory, scheduling logical processors using those LPCBs, managing their state flags, and defining Dflag (①) explained in section 3. This configuration is conventional and not new.

The relation between the hypervisor and the guest

OSs is similar to that between an OS and its application processes [9]. Contrary to this similarity, the scheduler of the hypervisor differs from a process scheduler of an OS in the following three points.

(1) An interrupt may be reported to a running logical processor directly, that is, by hardware and without intervention of the hypervisor.

(2) A logical processor in the running state will keep running immediately after it has accepted its interrupts directly.

(3) Conventionally if an interrupt request occurs and no logical processors of its target are running, it becomes a host interrupt, then, the hypervisor simulates it according to the hardware architecture and reports it to one of the target logical processors.

The differences (1) and (2) enhance the total system performance because of no intervention of the hypervisor, while the last difference (3) causes the guest I/O interrupt simulation and the mode change overhead stated in section 1, and reduces the system performance. We propose a new scheduling method to reduce those overheads in the following section.

3. New scheduling method

We propose a new scheduling method consisting of the following four methods:

- (1) adding a new self-wait state different from the conventional wait state,
- (2) detecting the I/O interrupts pending for the logical processors in the self-wait state by using a new instruction,
- (3) scheduling the logical processors in the self-wait state,
- (4) searching for workloads in an active wait state with two stages when workloads have been exhausted.

3.1. Self-wait state

If a running logical processor falls into the wait state, it is due to (1) the exhaustion of its workloads, or due to (2) the hypervisor commands to stop it.

The operators do not issue the hypervisor commands to the logical processor during its production running, while they may issue them during its test running, which means a guest OS is tested in the logical processor.

In the former case, an OS in the logical processor usually falls into the system wait state, which will be waked up by an interrupt to it. In the latter case, the logical processor should be started also by some hypervisor commands. To distinguish these two kinds of wait state, we call the former the self-wait state.

Figure 4 shows conventional states (①, ②, ③,

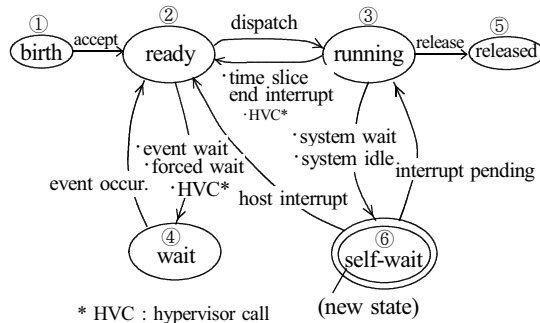


Fig. 4. Self-wait state of logical processors

(4), (5) and new state (6), that is, birth state (1), ready state (2), running state (3), released state (5), conventional wait state (4), and the new self-wait state (6). The logical processors in the self-wait state enter a self-wait queue, which is separated from a conventional wait queue, which consists of the logical processors in the conventional wait state (4). We present a method for detecting a VM's pending I/O interrupts by scanning this self-wait queue. This separation makes the scan quick. The interrupt delay is reduced because the scheduler of the hypervisor scans the self-wait queue ahead of the ready queue.

3.2. Conventional direct I/O execution method

Conventionally, I/O instructions issued by a guest OS are directly executed by hardware when they are issued to subchannels dedicated to the guest OS, and their interrupts are also directly executed during its running [2], [4]. This makes its performance close to that of the native machine. We premise that this direct execution method is already used in the real host machine. The conventional direct I/O execution method has been supported to the running logical processors. The I/O interrupts pending for VMs become host interrupts when all their target logical processors are in the wait state or in the ready state as stated in section 1. Our new method applies this direct execution method also to the self-wait logical processors though it may additionally delay the I/O interrupts to them.

Here, to clarify our method, we simply explain the conventional direct execution method having the following features,

(1) hardware logic recognizes the I/O interrupts pending for VMs without the intervention of the hypervisor,

(2) hardware logic judges whether or not a running logical processor is enabled for its pending I/O interrupts without the intervention of the hypervisor, (3) hardware logic informs a running logical processor of its I/O interrupts when it is enabled for them without the intervention of the hypervisor.

To support the feature (1), we allocate I/O interrupt priorities exclusively to VMs [2], [3]. Figure 5 shows I/O interrupt request queues. It illustrates an example of the exclusive allocation of the I/O

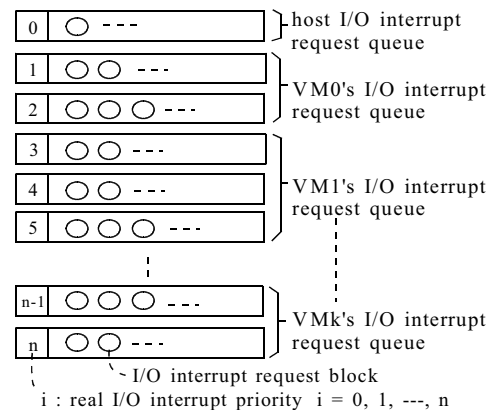


Fig. 5. I/O interrupt request queues (Conventional)

interrupt priorities 0, 1, 2, ..., n. It allocates priorities 1, 2 to VM0, priorities 3, 4, 5 to VM1, ..., priorities n-1, n to VMk, and allocates priority 0 to the hypervisor. An I/O interrupt request with priority (i) is enqueued into the request queue with priority (i). These I/O interrupt request queues are composed in real memory and managed by hardware, such as CPUs and I/O processors. With this queuing, hardware can recognize the I/O interrupt requests pending for VMs. For example, if an I/O interrupt request is enqueued into the queue with priority (i), and if the priority (i) is dedicated to VM1, the interrupt belongs to the VM1. As a result, hardware can recognize that the VM1 has the pending I/O interrupt requests.

This method does not require so many interrupt priorities, indeed, most OSs only use a few (probably two or three) priorities. This is because they will poll any I/O interrupts using some I/O instructions. These facts lead to this method which allocates a few interrupt priorities exclusively to VMs.

To support the feature (2) and (3), hardware has to have an interrupt control logic for a running VM which contains an I/O interrupt mask register and an I/O interrupt priority mask register. Several variations have been supported for those features, such as giving only one priority to each VM [3], or giving multiple priorities to each VM [2], or giving all the priorities

to each VM [4].

This conventional direct execution method can be applied only to the running logical processors. We apply it also to the self-wait logical processors.

3.3. Detecting I/O interrupts pending for self-wait logical processors

An I/O interrupt request occurs even when its target logical processor is in the wait state. Conventionally, it brings about a host interrupt. We present a method for suppressing this host interrupt. The I/O interrupt priorities are exclusively allocated to VMs as shown in Figure 5. We set the host masks of those exclusively allocated priorities always to be 0. This mask setting always suppresses the host I/O interrupts with those priorities. The pending I/O interrupts with those priorities are directly executed when one of its target logical processors are dispatched.

Conventionally, no scheduler dispatches the logical processors in the wait state or in the self-wait state. This is natural because the forced dispatching of the waiting logical processors will activate no activities and they will immediately fall into the wait state again, only increasing scheduling overhead.

After a logical processor falls into the self-wait state, its external system activities will generate a pending I/O interrupt or a pending timer interrupt or another interrupt pending for itself. When the logical processor is dispatched, those interrupts pending for itself can be directly executed by the hardware as long as the hardware can recognize those pending interrupts.

If the hypervisor is provided with a hardware function that can quickly detect the I/O interrupts pending for a VM, it can use the function to detect them. When detecting an interrupt pending for a self-wait logical processor, and it is enabled for that

interrupt, the hypervisor can dispatch it. Then, that interrupt request will be directly executed for it by the hardware.

Based on this consideration, we propose a new machine instruction called "Only Test Pending I/O Interrupts (OTPI)" shown in Figure 6, which detects an I/O interrupt pending with the priority i ($i = 0, 1, \dots, n$), which is an input parameter stored in a register R1, storing non-0 in a register R2 when detected, or 0 otherwise as a result. This instruction can be defined in any hardware architecture that has the I/O interrupt priorities $0, 1, \dots, n$ into which hardware enqueues the I/O interrupt requests.

Users allocate real I/O interrupt priorities exclusively to VMs using hypervisor commands. The hypervisor composes logical control blocks shown in Figure 3, which contains a Dflag (①), which indicates real interrupt priorities dedicated to the VM. By accessing it, the hypervisor can get those priorities, and can use this instruction with them, and can determine whether the VM has any pending I/O interrupts with those priorities.

The hardware usually has I/O interrupt request queues shown in Figure 5. This instruction is very simple and only checks the queue with the designated priority, and much faster than the I/O interrupt execution. This is because it only checks the designated queue and does not dequeue any I/O interrupt request element, while the I/O interrupt execution dequeues its request element and carries out the subsequent hardware interrupt procedures. Thus, the hypervisor can quickly determine whether a waiting logical processor has an I/O interrupt pending with its dedicated priorities without any host I/O interrupt for them.

We say that a logical processor is in the "self-wait-ready state," when it is in the self-wait state, at the same time, it has an I/O interrupt pending for itself, and it is enabled for that interrupt.

Conventionally, IBM provides an instruction called "Test Pending Zone Interrupt (TPZI)" [3], which checks the I/O interrupts of the designated guest, selectively can complete here, or continuously dequeues the I/O interrupt requests elements, and informs the hypervisor of the interrupt information. We think that this TPZI instruction is probably used to reap the guest I/O interrupt requests in the hypervisor of IBM's virtual machine system. This instruction seems to correspond to "Test Pending Interrupt (TPI)" instruction [12], [13] that native OSs use to check the pending I/O interrupts in the hardware queue and to dequeue (i.e. reap) the I/O interrupt request elements in the queue. Our OTPI instruction only tests the I/O interrupts pending in the queue and does not dequeue any request elements, therefore, much faster than those instructions. We do

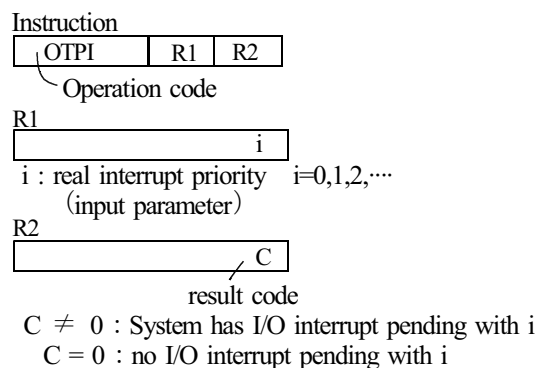


Fig. 6. Only Test Pending I/O Interrupt (OTPI) instruction

not know in detail how those other similar instructions are used in IBM's OSs or in the hypervisor of its virtual machine system.

3.4. Scheduling logical processors in self-wait queue

Figure 7 shows a new scheduling method using the self-wait queue ((a)). First, the scheduler checks the

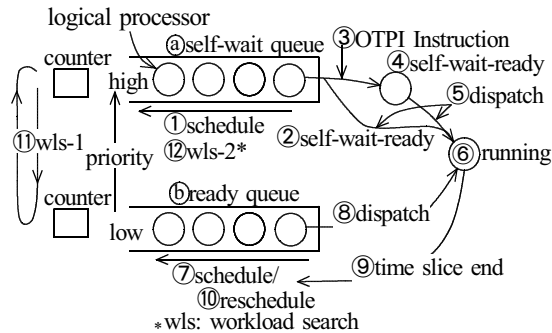


Fig. 7. Scheduling logical processors in self-wait queue

top entry of the logical processors in the self-wait queue (①). If it (i.e. the top entry) is in the self-wait-ready state (②), the scheduler dispatches it (⑤), otherwise checks whether it has an I/O interrupt pending with its dedicated priorities by using OTPI instruction (③). If it has an I/O interrupt pending for itself and if it is enabled for that interrupt, the scheduler sets it self-wait-ready (④), dispatches it (⑤), and sets it running (⑥).

Immediately after this dispatching, the real processor goes into vm mode, and directly executes the I/O interrupt pending for the running logical processor. If it (the top entry) has no pending I/O interrupts, the scheduler checks the next entry (i.e. a next logical processor) in the self-wait queue.

Second, when finding no dispatchable logical processors in the self-wait queue, the scheduler goes to the ready queue ((b)), and dispatches a ready logical processor if any (⑧). Thus the scheduler prefers the self-wait queue to the ready queue to shorten the additional delay of the I/O interrupts. Even in this ready queue, when finding no dispatchable entry, the scheduler goes to the workload search (⑪, ⑫), (Refer to 3.5).

The scheduler tries to keep running the logical processor in its time slice until it falls into the time slice end or the self-wait state. When the running logical processor falls into its time slice end (⑨), the scheduler reschedules it (⑩) according to its CPU service quantity, and gives higher priority to the logical processors with low CPU service quantities.

The additional delay of the guest I/O interrupts may occur because of no host interrupts pending for VMs. In order to shorten the additional delay, the scheduler gives the precedence to the self-wait queue over the ready queue, and checks the self-wait queue at every host event such as the host time slice end interrupts.

Conventionally a system has the interrupt delay due to the scheduling of its processes. That is, when the system has the processes with the same priority, and its process P0 is in the wait state, and an I/O interrupt to the process P0 makes it ready, and it is enqueued in its ready queue, and if there are k (≥ 1) ready processes before the process P0, then its reporting to the process P0 is delayed at most $k \cdot q$, where q is the time slice. Even if the system has several CPUs or many process priorities, a similar delay occurs.

To this conventional delay, our method adds at most one time slice because the host timer interrupt for the time slice end makes the scheduler quickly check the self-wait queue and dispatch the logical processor if it has an I/O interrupt pending for itself. The time slice is ordinary 1 ms to 10 ms, and thus usually causing no problem.

This additional delay may cause a timing problem for a VM that runs hard real time processes. In that case we can dedicate several processors to the VM in the SMP system [11], where they always run the VM in the vm mode and never fall into the self-wait state, and all its interrupts are executed directly with no additional scheduling delay.

3.5. Searching for workloads in active wait state with two stages

When finding no logical processors dispatchable in the current real processor, the scheduler will conventionally make it fall into the system wait state or actively search its workloads [10]. These conventional methods will find out its workloads by accepting host interrupts pending for guest VMs.

Instead of these methods, we present the method which causes no host I/O interrupts pending for VMs, and provides a workload search with two stages (see Figure 7), where the hypervisor has to accept the host timer interrupts and has to actively detect the I/O interrupts pending for the self-wait logical processors.

Figure 7 shows the workload search with two stages one of which is wls-1 (⑪) and the other is wls-2 (⑫). The first stage (wls-1) opens host interrupt masks for non-dedicated I/O interrupt priorities, and checks two counters, the self-wait queue counter and the ready queue counter.

Some waiting logical processors may come into the ready queue by host interrupts which may be host timer interrupts or may be host I/O interrupts with shared priorities. When finding the ready queue counter non-zero, the hypervisor exits from this stage (wls-1) and goes to the scheduler, when finding the ready queue counter zero and the self-wait queue counter non-zero, it exits from this stage, and goes to the second stage (wls-2), and when finding both counters zero, it loops between them.

In the second stage (wls-2) the hypervisor scans the self-wait queue, and checks if each logical processor has I/O interrupts pending for itself. The hypervisor checks the self-wait logical processor (we call this A) by issuing an OTPI instruction whose operand is an I/O interrupt priority dedicated to the logical processor A, and sets its self-wait-ready flag if it has the I/O interrupt pending for itself and is enabled for that interrupt.

In that case the hypervisor sets the self-wait-ready flags of all the self-wait logical processors that are brothers of the logical processor A, that is, belong to the same VM. This is because there may be several I/O interrupts pending for the VM in the interrupt request queues with its dedicated priorities (Refer to Figure 5).

The hypervisor can easily find dedicated I/O priorities by seeing the Dflag (①) of the logical processor as shown in Figure 3, where priorities $i, i+1, \dots, i+j$ ($j \geq 0$) are dedicated to a VM (represented as VMBLOCK), and to logical processors 0, 1, $\dots$, n (as LPCB0, LPCB1, $\dots$, LPCBn, respectively). It can also easily find all the logical processors belonging to the same VM by seeing its pointers ② in Figure 3. Thus when finding at least one self-wait-ready logical processor, the hypervisor exits from the stage 2 workload search and goes to the scheduler to dispatch it, when finding no such logical processors, it returns to the first stage (wls-1).

The pending I/O interrupts may occur asynchronously to the real processors. The hypervisor has to get the logical processor's most current state. Therefore, it has to issue the OTPI instruction each time it checks if the logical processor has a pending I/O interrupt. It is not sufficient for the hypervisor to read all the pending mask bits only once of all self-wait logical processors because the values of their mask bits may vary momentarily. Moreover, the hypervisor should schedule them in the order of their self-wait states and their pending I/O interrupts.

4. Related art and architecture

Our proposed method schedules and dispatches the self-wait logical processors. A conventional OS's

scheduler does not dispatch the waiting processes, usually accepting interrupts for them, changing their states from the wait to the ready, after that, dispatching only the ready state processes, thus having no methods for scheduling and dispatching the waiting processes directly.

Active wait method is supported conventionally [10]. It opens all the interrupt masks, causing any kind of interrupts, resulting in the increase of their simulation overhead in the VM environment. Instead of this, our active wait method disables for the I/O interrupts pending for VMs with dedicated interrupt priorities, detecting them without host interrupts, responsive to the detection, scheduling and dispatching the VMs.

Conventionally, Denali [7] has the lazy interrupt processing and Xen [8] has the batch processing for interrupts. In both methods the hypervisor accepts I/O interrupts pending for VMs, translating them into simple events, and reporting them at the scheduling of the VM. This is the I/O simulation and will cause high simulation overhead.

Contrary to this, in our method the hypervisor never accepts host I/O interrupts pending for a VM with dedicated interrupt priorities. The hypervisor detects those I/O interrupts pending for a waiting logical processor of a VM by using the new OTPI instruction, which can quickly detect them. After detecting them and checking if it is enabled for them, the hypervisor schedules it, then, they are directly executed in the vm mode hardware, thus causing no I/O simulation overhead.

Our method also has the conventional scheduling delay, and moreover may have the additional interrupt delay which is at most one scheduling time quantum due to the masking of the host I/O interrupts as described in section 3.4. We premise that our virtual machine system has the small number of VMs, that is, having at most ten VMs in the system. Therefore, these scheduling delays cause no problem because of the small time slice such as 1ms -10ms. The number of VMs is small because the guest OSs have almost the full use of the hardware and software functionalities and can run in near native performance, and will have the scalability to the number of their own processes.

Our method premises that the hardware architecture has a feature which supports direct I/O execution for VMs in their *running* states. It is widely supported in mainframe architectures [2], [4].

5. Implementation and performance evaluation

5.1. Implementation

As for hardware we have implemented the OTPI instruction, which detects I/O interrupts pending of the designated interrupt priority (See Figure 6). We used a hardware microcode program and two general purpose registers. This was very easy because the basic I/O interrupt queues were supported natively in hardware, only requiring additional control storage for its microcode.

As for software we have implemented the self-wait state, the mask setting to disable for the host I/O interrupts pending for VMs, the method for scheduling the self-wait logical processors, and the workload searching with two stages.

We have evaluated our scheduling method consisting of these proposed methods by measuring the performance for several workloads. Table 1 summarizes the performance comparison between the conventional host interrupt method and our proposed method. We describe the measurement method and the performance data in the following sections.

5.2. Measurement method

We address the users who daily operate multiple VMs concurrently under a single real host SMP. Several users have very large size systems consisting of several SMPs each of which has many (up to 128) processors sharing very large size (up to 2048GB) main memory and concurrently runs several (up to tens of) VMs. On the other hand, many users have medium size systems each of which will be a 2 to 4-way SMP system and runs 2 to 4 VMs concurrently.

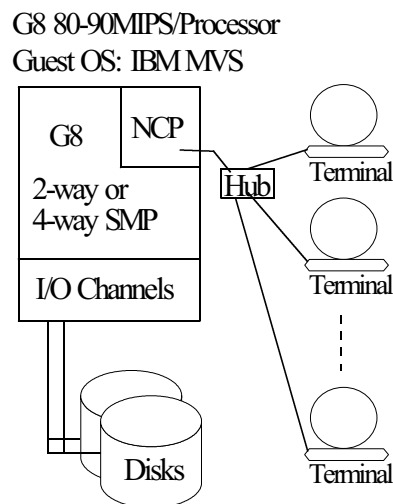


Fig. 8. Measurement Environment

Because our VMs can use almost the full functions of hardware and software, the number of VMs concurrently running is not so large, that is, up to ten. This is very different from the Denali VMs [7] and from the Xen VMs [8].

Considering that our virtual machine system has the small number of VMs running concurrently, we need not evaluate the scalability to the great number of VMs concurrently running. The scalability should be evaluated to the number of processes in each of the guest OSs. Here, the most important is the performance of each VM to the native machine.

We use the measurement environment shown in Figure 8, using a 4-way SMP, which consists of 4 processors sharing main memory, and a 2-way SMP, which consists of 2 processors sharing main memory.

Each processor has a 6.2ns machine cycle. This SMP has almost the same architecture as IBM System/390 [12]. The guest OS is the MVS, which is the OS of the IBM mainframe.

The number of logical processors of a VM is the same as that of the real host SMP. This is because the VM should have as many concurrently running logical processors as possible to improve its performance.

That is, when a real host is a 4-way SMP, we run 2 to 4 VMs, each of which is 4-way, that is, has 4 logical processors. When a 2-way SMP host, we run 2 VMs, each of which is 2-way, that is, has 2 logical processors.

The measurement of one case takes 1 to 3 hours including preparations, and measured workloads are I/O heavy jobs which seem to have the size and attributes comparable with real workloads. We use a performance monitor built in the guest OSs, summarizing the results in Table 1, explaining each case as follows.

Case 1 (#1 in Table 1) runs a 4-way SMP as a real host, in which running 2 VMs concurrently, each of which is 4-way, that is, has 4 logical processors. Its workloads are the Time Sharing System (TSS) jobs with 100 active terminals, and each VM has 50 terminals. We use a terminal simulator to generate the transactions each of which consists of a series of edit and its subcommands. We run the total system for about 15 minutes, getting performance data for two minutes (i.e. measurement interval in Table 1) of steady state during that running interval.

Table 1. Performance comparison of conventional method (a) with proposed method (b)

#	Workload	Item	Conventional method (a)	Proposed method (b)	Native mode (c)	b-a	(b-a)/a (%)	a/c (%)	b/c (%)	Comments
1	TSS 100 terminals	ETR ^{*1} (Ended/sec)	51.60	53.57	53.66	1.97	3.82	96.16	99.83	Host:4way 2 VMs each VM:4way $\rho^* = 26\%$
		Simul. ovh. ^{*3} (%)	3.21	1.85	-	-1.36	-42.37			
		Sched. ovh. ^{*4} (%)	15.38	1.77	-	-13.61	-88.49			
		Interval ^{*5} (sec)	120.00	120.00	120.00	0.00				
2	I/O_heavy	ETR ^{*2} (EXCPs/sec)	12.65	16.34	16.90	3.69	29.15	74.87	96.69	Host:2way 2 VMs each VM:2way $\rho^* = 49\%$ (estimated from VM mode)
		Simul. ovh. ^{*3} (%)	12.95	7.43	-	-5.52	-42.63			
		Sched. ovh. ^{*4} (%)	20.17	5.79	-	-14.38	-71.29			
		Interval ^{*5} (sec)	300.00	300.00	300.00	0.00				
3	I/O_heavy	ETR ^{*2} (EXCPs/sec)	13.30	16.31	16.70	3.01	22.63	79.64	97.66	Host:4way 2 VMs each VM:4way $\rho^* = 25\%$
		Simul. ovh. ^{*3} (%)	13.86	8.95	-	-4.91	-35.43			
		Sched. ovh. ^{*4} (%)	42.96	8.47	-	-34.49	-80.28			
		Interval ^{*5} (sec)	300.00	300.00	300.00	0.00				

*: ρ : Native CPU utilization

Host Machine Cycle : 6.2ns

*1: ETR: External Throughput Ratio: Transactions ended / Real time one sec.

*2: ETR: External Throughput Ratio: Execute Channel Programs issued / Real time one sec.

*3: CPU utilization for simulating system wait inst.(Load PSW wait) and host timer interrupts that should be reported to guests

*4: CPU utilization for scheduling logical processors

*5: Measurement interval

Case 2 (#2 in Table 1) runs a 2-way SMP as a real host, in which running 2 VMs concurrently, each of which is 2-way, that is, has 2 logical processors. Its workloads are I/O heavy jobs issuing many Execute Channel Program (EXCP) macro-instructions, which are I/O macro-instructions, very frequently. We get the performance data for 5 minutes (i.e. measurement interval in Table 1) of steady state during that running interval.

Case 3 (#3 in Table 1) runs a 4-way SMP as a real host, in which running 2 VMs concurrently, each of which is 4-way, that is, has 4 logical processors. The other attributes are the same as the case 2.

5.3. Performance data

(1) Simulation Overhead (CPU utilization)

The simulation overhead belongs to each VM. It reduces the total system performance because it decreases of the CPU times for other VMs. Table 1 shows that the simulation overhead remarkably decreases compared with that of the conventional

method, that is, in CPU utilization, 3.21% to 1.85% in case 1, 12.95% to 7.43% in case 2, and 13.86% to 8.95% in case 3. This data means that our proposed method reduces the number of host I/O interrupts pending for the guest OSs, and the self-wait logical processors directly execute their own I/O interrupts at the time they have been dispatched.

(2) Scheduling Overhead (CPU utilization)

The scheduling overhead is the CPU utilization for scheduling logical processors, and if it is high, it reduces the total system performance. The overhead is remarkably reduced compared with that of the conventional method as Table 1 shows that 15.38% to 1.77% in case 1, 20.17% to 5.79% in case 2, and 42.96% to 8.47%. The high scheduling overhead of the conventional method is due to the high frequency of the host I/O interrupts pending for the guest OSs and its poor workload searching.

(3) External Throughput Ratio (ETR)

The external throughput ratio (ETR) is the number of transactions processed in case 1, or the number of EXCP (Execute Channel Program) macro-instructions issued in cases 2 and 3 in a real time second. This is

the performance from the point of user view.

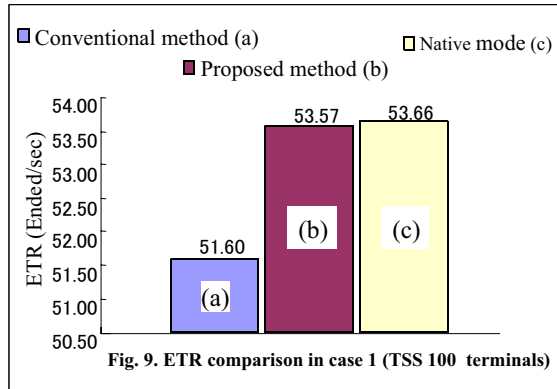


Fig. 9. ETR comparison in case 1 (TSS 100 terminals)

As for case 1 (TSS 100 terminals) Figure 9 shows a visible ETR improvement from 51.60 to 53.57.

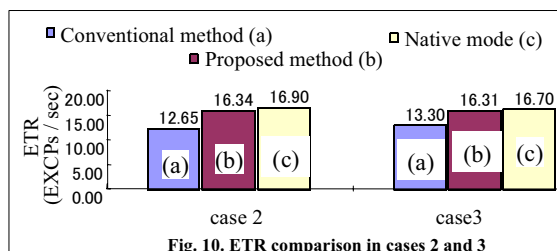


Fig. 10. ETR comparison in cases 2 and 3

As for cases 2 and 3 (I/O heavy jobs) Figure 10 shows a clear ETR improvement from 12.65 to 16.34 in case 2, and from 13.30 to 16.31 in case 3, additionally showing the ETR improvement reaching the near native performance.

These ETR improvements are almost due to the decrease of the I/O interrupt simulation for VMs and that of scheduling overhead.

6. Conclusions

We have presented a new method for scheduling logical processors in the self-wait state ahead of those in the ready state, and for searching workloads using two stages which scan the self-wait logical processors.

Having implemented this new method, we have evaluated it by measuring the performance for some workloads, and have confirmed their improvements by the performance data showing the decrease of the I/O interrupt simulation overhead and that of the

scheduling overhead. Moreover, we have confirmed the ETR performance improvement reaching the near native even for the I/O heavy workloads.

7. References

- [1] IBM, "Logical Partitioning", <http://www-1.ibm.com/servers/eserver/series/lpar/>.
- [2] Umeno, H., et al., "I/O execution method for a virtual machine system and system therefor", Hitachi, US 5109489, Priority Jan. 18, 1984.
- [3] Bean, George H., et al., "Logical Resource Partitioning of a Data Processing System", IBM, USP4843541, July 29, 1987 Filed.
- [4] Chou, Norman Cho-Chun, et al., "CPU expansive gradation of I/O interruption subclass recognition", US 752149, IBM, Filed Aug. 29, 1991.
- [5] Sugerman, J., et al., "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor", VMware, Inc., Proc. of the 2001 USENIX Annual Tech. Conf., Boston, Ma., USA, June 25-30, 2001.
- [6] Scott W. Divine, et al., "Virtualization System Including a Virtual Machine Monitor for a Computer with a Segmented Architecture", US6397242B1, VMWare Inc., Filed Oct. 26, 1998.
- [7] Whitaker, A., Shaw, M., Gribble, Steven D., "Scale and Performance in the Denali Isolation Kernel", In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, ACM Operating Systems Review, Winter 2002 Special Issue, Boston, MA, USA, Dec. 2002, pp. 195-210.
- [8] Barham, P., Dragovic, B., Fraser, K., et al., "Xen and the Art of Virtualization", SOSP'03, October 19-22, 2003, BoltonLanding, NY, USA, pp. 164-177.
- [9] Stallings, W., "Operating Systems: Internal and Design Principles Fourth Edition", Prentice Hall (2000).
- [10] Blandy, G. O., et al., "Active wait", US4631674, IBM, Filed Feb. 5, 1985.
- [11] Umeno, H., et al., "Development of Methods for Reducing the Spins of Guest Multiprocessors", Trans. IPS Japan, Vol. 36, No. 3, pp. 681-696 (March, 1995).
- [12] IBM, "ESA/390 principles of operation", SA22-7201-08.
- [13] IBM, "z/Architecture Principles of Operation", SA22-7832.