IPSJ Transactions on Advanced Computing Systems Vol. 2 No. 4 64–82 (Dec. 2009)

Regular Paper

Inter-kernel Communication between Multiple Kernels on Multicore Machines

Taku Shimosawa[†1] and Yutaka Ishikawa[†1,†2]

We propose a new inter-kernel communication mechanism for multicore architectures in order to communicate with multiple kernels within a single machine. Using this mechanism, multiple kernels share I/O devices, such as network and disk devices. The mechanism has been integrated into another mechanism called SHIMOS that partitions the CPUs, the memory, and I/O devices. Multiple Linux kernels on multicore architectures have been realized using the integrated SHIMOS mechanism. Several sets of benchmark results demonstrate that SHIMOS is faster than modern virtual machines. For system calls, SHIMOS achieves about seven times faster than the Xen virtual machine. When two Linux compilation jobs run on two Linux kernels, SHIMOS is 1.35 and 1.005 times faster than Xen and the native single Linux, respectively.

1. Introduction

Multicore processors have been adopted in various computer systems, from commodity machines 1),2) to embedded systems 3)–5). Although such environments have attracted a lot of attention, not every application is able to exploit the entirety of the increasing parallel computation capacity.

To utilize these multicore machines more efficiently, a promising way is to run multiple applications on one machine or share resources among different users in computer centers. Though each application may not be able to utilize the entirety of the computation resources, those resources can be used by multiple applications or by several users. In such a situation, instead of using a single operating system environment, multiple operating systems may run on one machine. There are two reasons for this: One is that some applications require different operating system environments or that the users require isolation of the operating system. Another reason is that running multiple operating systems can provide better performance isolation by avoiding contention within a kernel caused by system calls from multiple applications that are running in parallel.

There are two main approaches to running multiple kernels: logical partitioning (LPAR) 6) and virtual machines (VMs) 7). In the existing logical partitioning approach, the CPUs, memory and devices of a machine are partitioned by a hardware or firmware mechanism. There is nearly no overhead caused by the partitioning in LPAR methods unless a processor is time-shared. This has been equipped with IBM mainframes 8) and HP-UX servers 9), but they require dedicated hardware/firmware support, and thus it has not been available for commodity machines.

The virtual machine approach basically requires no special hardware support, and has been widely used in commodity machines. Popek and Goldberg provided a definition of virtual machines 10). According to their definition, a virtual machine monitor (or VMM for short) must execute most instructions directly and also must provide resource control. Following the definition, VMMs should simulate privileged CPU instructions and I/O, as well as system calls and external interrupts. Although many techniques are proposed to accomplish efficient simulation, the inherent overheads cannot be eliminated in the VM approach.

In some systems that emphasize performance, this overhead of virtual machines is not acceptable. For example, users of computing clusters are not willing to have their programs slowed down by virtualization. In the emerging multicore embedded systems, there can be almost no room of computing and memory capacity to spare for virtualization. Nevertheless, they are running multiple kernels and providing multiple environments in these systems: a process-per-processor system like Virtual Node mode in IBM's BlueGene/P supercomputer 11), or running a normal kernel and real-time kernel. For these situations, a higher performance at the risk of a weak isolation of the kernels may be realistic.

In order to provide a multiple kernel execution environment for multicore machines without sacrificing performance, a new partitioning mechanism, called SHIMOS (Single Hardware with Independent Multiple Operating Systems) has been designed and implemented by the authors 12). SHIMOS is implemented in the Linux kernel by adding several pieces of kernel code and a kernel module

---

[†1] Graduate School of Information Science and Technology, The University of Tokyo
[†2] Information Technology Center, The University of Tokyo

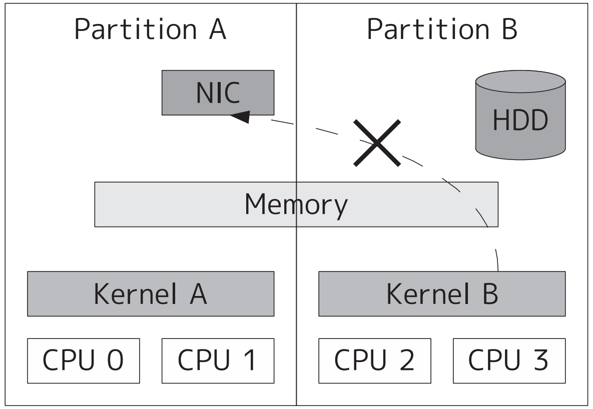


Fig. 1 Resource partitioning in the former SHIMOS.

called KLoader to boot another kernel from a kernel. **Figure 1** shows the partitioning of resources in a machine by SHIMOS. As the figure shows, all Linux kernels run without anything such as a hypervisor under them, and so the kernels run under the privileged CPU mode and manipulate I/O devices directly. Thus, SHIMOS does not introduce any additional overheads as do VMMs. It is implemented by relatively little program code compared to the existing virtual machine monitors. Moreover, it requires no special hardware for support, and thus it runs on commodity x86 multicore machines unlike traditional partitioning. In the paper 12), the basic functionality and feasibility of partitioning hardware, such as partitioning CPUs, memory, and I/O devices, were presented.

SHIMOS is not a VMM as described in Popek and Goldberg's definition because it violates the one definition of VMMs: provide "resource control." SHIMOS restricts the resource usage of each kernel, but the limit is established by the kernel itself. There is no mechanism to prevent a kernel from accessing resources dedicated to another kernel. If a malfunctioning kernel accesses a resource belonging to other kernels, the system will be in chaos. In the SHIMOS design philosophy, providing a highly efficient multiple kernel execution environment takes priority over providing a protection mechanism against such a malfunction. As a result of this design philosophy, SHIMOS is faster by 134% than the Xen paravirtualization, as shown in the paper 12).

Though the SHIMOS partitioning approach supports an efficient multiple kernel execution environment on multicore machines, it has one major limitation that there is no capability of sharing I/O devices. This means that as the number of kernels increases, the number of devices should increase as well. This limitation is not acceptable in computer centers that require sharing of networks and storage.

In this paper, an inter-kernel communication mechanism is proposed and integrated into the SHIMOS mechanism in order to provide a general framework to share I/O devices. The integrated SHIMOS achieves the best performance in most cases when compared to Xen paravirtualization and KVM. Moreover, this paper shows that partitioning is better in some cases than running on the native single kernel.

This paper is structured as follows: In the next section, we present an overview of the inter-kernel communication and virtual devices in the SHIMOS mechanism. The implementation for Linux on x86 architecture is described in Section 3. Related work is presented in Section 4. In Section 5, SHIMOS is evaluated using both micro benchmarks and application-level benchmarks, and compared with Xen and KVM. Microbenchmarks measure the `getpid`, `fork` and `wait` system calls, and the network and disk performance. Application-level benchmarks use the SPECint 2006 benchmark, the Linux kernel compilation, the Apache benchmark, and the mix of these applications. The paper is concluded with pointing out future work in Section 6.

2. Design of Inter-kernel Communication in SHIMOS

In this section, we describe the basic design of the inter-kernel communication used in SHIMOS to implement the virtual devices. Via virtual devices, a kernel can share the physical devices in another kernel. We take into consideration that each operating system kernel running in a machine can differ.

The inter-kernel communication mechanism is composed of three functions: sharing, transfer and notification. In the following subsections, we describe how the shared memory area is constructed in the independent kernels, how the data of one kernel is transferred to other kernels, and how the kernels notify each other of things like packet arrivals. We also describe the management of the virtual devices.

2.1 Shared Memory

In order to provide data available to all the kernels with the same virtual addresses, we introduce a memory area shared by all kernels to SHIMOS. Queues of inter-kernel communication packets (IKC packets in short) are located in this shared memory area. The number of queues can be more than one for each kernel, to implement virtual devices efficiently. The virtual devices send requests to the kernel where the physical device resides via inter-kernel communication, by passing IKC packets.

This design of the virtual devices is derived from the design principle that performance is prioritized over protection, and the SHIMOS design that a CPU core is dedicated to a kernel to avoid overhead. Therefore, this IKC design is for a multicore CPU because sharing a CPU core among multiple kernels requires context switching among the kernels, and thus it results in overhead for IKC packet transmission.

Another design choice is to create a device kernel that manages all devices. However, it requires at least one CPU to be dedicated for device processing, which results in the smaller number of available CPUs for users. Therefore, we have selected a design whereby each device is assigned to one of the kernels, and the other kernels request the kernel which has that device.

2.2 Transfer Methods

There are two issues concerning the transfer. One is that kernels with data representations that may differ must communicate with one another. Another issue is that the overhead for accessing transferred data should be minimized. Copying of data should be avoided.

The first issue exists because kernels use various structures and data buffers, so the virtual devices must transfer some of those to another kernel. In general, structures differ among operating systems, and even among versions of the same operating system. Therefore, the virtual device converts some of the data structures to the IKC packet and then reconstructs to its own kernel structure. While the data structures cannot be shared, data associated with the data structures may be shared. For example, consider the `sk_buff` structure used in the network layer in the Linux kernel. An `sk_buff` structure represents a network packet in the kernel, and it contains a lot of information about the packet in its member variables, such as pointers to the headers of different network layers, time stamps and the actual sizes of data. However, the members of the structure can vary from version to version of Linux, thus the structure cannot be simply passed to another kernel. In contrast, the contents of the packet, pointed to by a pointer in the `sk_buff` structure, are independent of the kernel implementations.

The basic design of the IKC packets and virtual devices is conducted as follows: An IKC packet contains a pointer to the data, and contains the abstracted information for the data. Upon a request, the virtual device converts the request into an IKC packet and sends the packet to the destination kernel. The virtual device in the destination kernel reconstructs a kernel request structure from the IKC packet.

There are two possible locations for the data to be transferred. One is in the shared memory area, and the other is in the dedicated area. We provide two methods for data transfer to support both cases. The first is the shared memory transfer method, and the other is the page transfer method. We describe the details of both methods in the following subsections, and show how they alleviate the second issue, that is, minimizing the cost of accessing transferred data.

2.2.1 Shared Memory Transfer

In the shared memory method, data allocated from the shared memory is transferred. Since the portion of the shared memory is accessible to all the kernels in the machine by use of the same address, there is no need to perform any additional procedure. Obviously, no copy of the data is required during the transfer. The IKC packets can only contain the pointer to the data as their payload, and so the overhead of the IKC packet construction is small. To realize this mechanism, there needs to be a common memory allocator for the shared memory. Memory portions are allocated without sleeping by this memory allocator. Sleeping is not allowed within the allocator because it is shared by multiple kernels. If it were allowed, a further facility for waking up would have been necessary and it would have been difficult to implement in the situation that different operating systems are running.

The receiver kernel is able to free the data using the memory allocator because the sender kernel has allocated data using the same allocator. If no response is expected, the receiver is responsible for freeing the data. Therefore, the data for inter-kernel communication is managed by the shared memory allocator.

2.2.2 Page Transfer

The page transfer method is used to transfer the memory area outside the shared memory area. It achieves another zero-copy transfer, but memory mapping costs exist compared to the shared memory method.

In the page transfer method, the physical address of the data is transferred. The receiver kernel maps the given physical address to its virtual memory map to access the data. The overhead is the time needed to alter the page map table and the TLB misses which occur every time. Therefore, this method should be used for communication where large-sized data are sent with relatively low frequency, such as block device requests.

2.3 Notification

Other than just queuing IKC packets, it is necessary to notify the target kernel of packet arrivals. In SHIMOS, the notification is accomplished with IPIs (Inter-Processor Interrupts). Upon an IPI from another kernel, a kernel examines its queues and handles packets in the queues properly, possibly by waking up the virtual device codes.

Since modern device drivers use the polling method to reduce the interrupt cost when they are highly loaded, the queues are associated with flags which indicate whether a notification is necessary or not. The flag can be used to enable or disable interrupts adaptively by virtual device drivers.

2.4 Device Management

For a simple efficient IKC mechanism, SHIMOS itself does not provide access control of the virtual device requests. The control of the requests are responsible for the kernel which holds the physical device because modern operating systems have capabilities for controls such as packet filtering and disk resource limit. Also, because virtual devices access physical devices via the abstracted request described in Section 2.2 the configuration mismatches among the virtual devices in different kernels cannot occur.

3. Implementation in Linux

We have implemented the SHIMOS mechanism in Linux 2.6.26 for the x86 architecture. As a basic implementation principle, patches to the kernel should be as small as possible. The numbers of lines of modifications to the Linux kernel, and the additional kernel modules, are shown in **Table 1**.

In the following, we show the implementation of inter-kernel communication and virtual devices in SHIMOS, following the design shown in the previous section. The subsections below describe the support of the shared memory area, the implementation of the shared memory allocator, the inter-kernel communication interface, and the implementation details of the virtual network and block devices. Finally, we describe the improvement to the SHIMOS partitioning method by adding the shared memory.

3.1 Shared Memory Support

To support inter-kernel communication, the shared memory area must be added to the SHIMOS memory map. Since the area must be addressable by the same address from all the kernels, and the address calculation should be as simple as possible, we put the shared area at the lowest physical address, and the dedicated area at the next higher address.

The physical and kernel virtual memory map after the initialization phase is illustrated in **Fig. 2**. To construct this memory map, some initialization codes in Linux have been modified.

The introduction of the shared memory area is not only for the inter-kernel communication but also for alleviating some remaining issues on partitioning. This is described in the later section.

3.2 Memory Allocator

The shared memory allocator must use the same implementation or, at least, the same algorithm and protocol for meta information, such as the allocation map, the pointer to the last free area, and the lock variable to protect the infor-

Table 1 Patched lines.

Patch to Existing Source	522 lines
Additional Kernel Source	798 lines
Kernel Loadable Modules	1,798 lines

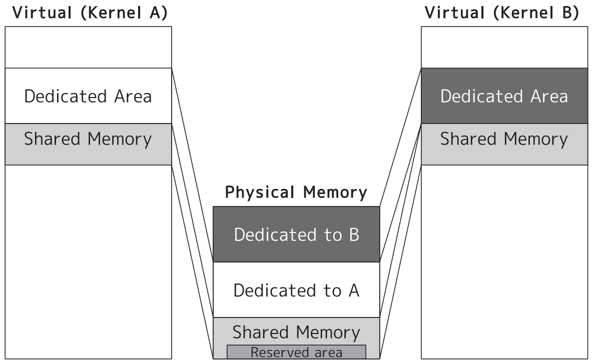


Fig. 2 Kernel memory maps in SHIMOS.

mation. However, the same implementation is not generally appropriate because the implementation of the spinlock primitive may differ in each kernel. This difference is caused by the required procedures before and after the lock, for example, preemption counters in Linux. Therefore, the memory allocator is implemented in each kernel.

The information for the allocator is located at a fixed address in the shared area, along with the lock variable for the information. Because sleeping is not allowed at the allocator, we use the spinlock method for a lock to the information.

The allocator itself is implemented using a very simple allocation algorithm. The worst time complexities are $O(n)$ for allocation, and $O(1)$ for de-allocation. To reduce external fragmentation, the allocator allocates memory by using a unit of a kilobyte, which can be changed at boot time.

The interface of the allocator is also simple: a function used to allocate and a function used to free.

3.3 Inter-kernel Communication

The general inter-kernel communication in SHIMOS is composed by queues and IKC packets. The queues are allocated in the shared memory during the initialization of virtual devices. They are implemented as ring buffers so that they do not need locks if there is a single reader and writer. The interface for inter-kernel communication, including an overview of the structure of an IKC packet, is shown in **Fig. 3**: there are functions to put an IKC packet into a specified queue, to pick one from the queue, and to notify the destination kernel that a packet has arrived or the status of a queue has changed. The notification is accomplished by an inter-processor interrupt (IPI). Since the interrupt cannot contain more information within itself, the response message is sent by another IKC packet.

```
int shimos_ikc_write(struct ikc_packet *pkt, int kern, int qn);
/* Puts an IKC packet 'pkt' to the queue number 'qn'
   of the kernel 'kern' */

int shimos_ikc_read(struct ikc_packet *pkt, int qn);
/* Gets an IKC packet from the queue number 'qn' of its own */

int shimos_ikc_notify(int kern, int qn);
/* Sends an IKC notification message
   so that the kern can process the message */

struct ikc_packet{
    short cmd;      /* NetworkPacket, BlockRead, etc. */
    short sender;   /* Sender kernel number */
    int param[6];   /* Depends on cmd */
    void *data;     /* Pointer to the data if any */
};
```

Fig. 3 Inter-kernel communication interface.

3.4 Virtual Network Device

To reduce the cost of data transfer between kernels, as we have presented in Section 2.2.1, the data to be sent is allocated from the shared memory.

In order to allocate network packets from the shared memory area, we modified the `alloc_skb` function. This is worth patching because copying during network communication is avoided by modification of just a few lines. Since the network packets in the Linux kernel are represented as `sk_buff` structures, the structures and the packet data are always allocated using the `alloc_skb` function. However, the members in an `sk_buff` structure may vary according to the versions of the kernels, so the `sk_buff` structure is not shared between kernels and not allocated from the shared memory, but from the normal kernel memory area. Only the packet data in the `sk_buff` structure is allocated from the shared memory. We modified this function so as to use the allocator function of the shared memory allocator. Also, we changed the free function of the `sk_buff` structure, the

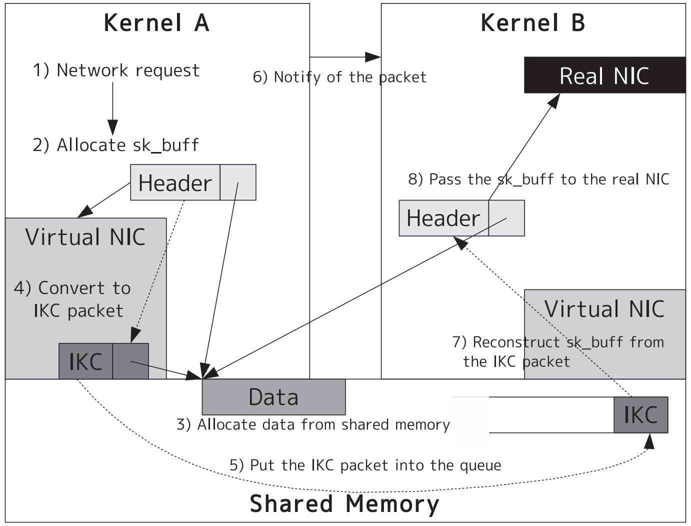


Fig. 4 Virtual network device implementation in SHIMOS.

`free_skb` function similarly.

The flow of network communication is illustrated in **Fig. 4**. In this figure, a packet is transferred from kernel *A* to the real NIC in kernel *B* as follows: 1) We assume that a network request is issued by some activity in kernel *A*. 2) Kernel *A* allocates an `sk_buff` structure by using the `alloc_skb` function. 3) The packet data is allocated from the shared memory area, but the header of the structure is allocated from the normal kernel memory, as described above. 4) When the `sk_buff` structure is passed to the virtual network device, the device converts it to an IKC packet, and discards the `sk_buff` structure, except for the data. 5) The converted packet is put into the queue of kernel *B*. Obviously, if the destination does not exist, the packet is just discarded. 6) Kernel *A* issues an inter-processor interrupt as a notification to kernel *B*. 7) Kernel *B* gets the packet from the queue, and the virtual device reconstructs an `sk_buff`. It throws the `sk_buff` structure to the upper network layer, and finally, 8) the packet reaches the physical network controller.

The IKC packet representing a network packet contains the pointer to the buffer, the offset, and the size of the actual data in the buffer, which are sufficient to reconstruct the `sk_buff`.

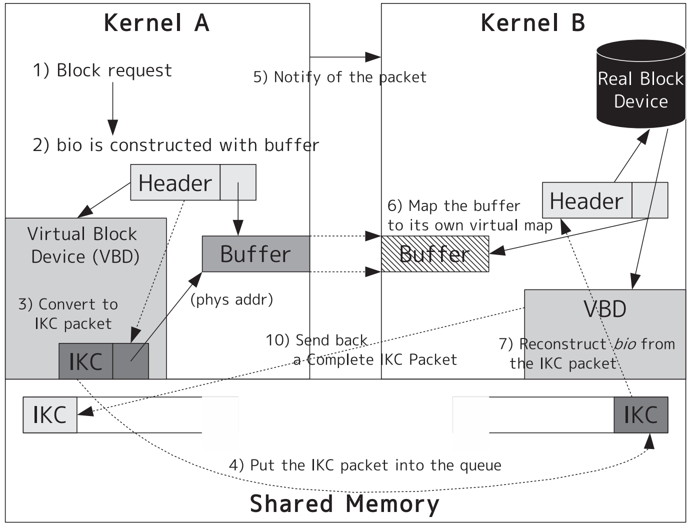

Kernel A
Kernel B
1) Block request
2) bio is constructed with buffer
Header
Virtual Block Device (VBD)
3) Convert to IKC packet
IKC
Buffer
(phys addr)
5) Notify of the packet
Real Block Device
6) Map the buffer to its own virtual map
Buffer
Header
VBD
7) Reconstruct bio from the IKC packet
10) Send back a Complete IKC Packet
IKC
IKC
4) Put the IKC packet into the queue
Shared Memory

Fig. 5 Virtual block device implementation in SHIMOS.

3.5 Virtual Block Device

For a virtual block device, it is difficult to use shared memory because the request for the block device can be passed with any page in the kernel. Therefore, virtual block devices use the page transfer method for the inter-kernel communication presented in Section 2.2.2. **Figure 5** shows the flow of a block device request from kernel *A* to the physical devices in kernel *B*. 1) A thread in the kernel *A* tries to read or write to a disk provided by the virtual block device. 2) The kernel constructs a *bio* structure, which consists of those pages that have buffers to be read or written to, and it requests a virtual device, using the *bio* structure. 3) The virtual device translates the *bio* structure to an IKC packet with the physical addresses of the pages of the request. 4) It puts the IKC packet into the block queue of kernel *B*. 5) It notifies kernel *B* of the request using an IPI, if necessary. 6) The driver in kernel *B* receives the packet, and it maps the page to be read or written into its virtual memory map. 7) It reconstructs a *bio* structure with the newly mapped pages. 8) Finally, it makes a request to the real block device. 9) Upon the response from the real device, the driver unmaps the region to be read or written to. 10) Kernel B sends a simple packet that notifies kernel A of the completion of the request. If necessary, kernel B also issues a notification to kernel A.

The mapping of memory outside the kernel is not simple because actions in the kernel to memory areas generally require a *page* structures. In addition, the `page_to_pfn` function, which converts *page* structure to its corresponding physical page number, assumes that the *page* structures are just a linear array. Naturally, the corresponding *page* structure for a memory area outside the kernel is not in the array, or may not even exist. The virtual block device in SHIMOS allocates a temporary *page* structure for the transferred page. For implementation simplicity, we also modified the members of the *page* structures and the `page_to_pfn` function so that kernels can handle irregular *page* structures. Unmapping is done by only freeing the temporary *page* structure.

3.6 Improvement in Booting Procedure

The shared memory area, introduced in Section 3.1 provides more flexibility to the partitioning compared to the former SHIMOS mechanism.

There is a limitation of the boot order in the previous implementation of SHIMOS that a kernel which uses the lowest memory address must be booted last. This limitation comes from the property of the x86 architecture that all CPUs boot in the "real" mode, being able to access only the lowest 1 MB memory. With the simple partitioning of the memory, the kernels to which the higher memory areas are dedicated even cannot boot at all without using the low memory area.

Now that the shared memory is provided, the booting mechanism is improved to use the area. In SHIMOS, the kernel module named *KLoader* in one kernel prepares and boots up another kernel. The KLoader module loads the bootstrap routine in a reserved area in the shared memory, and puts the remaining part of a kernel to the memory area dedicated to that kernel. Finally, it wakes up one of the CPU cores dedicated for that kernel by an IPI. In addition, the code to wake up another CPU cores in the Linux SMP kernel is modified to use the reserved area.

With this improvement, the booting of the kernels can be performed in any order. Also, it is possible to reboot a kernel while the other kernels are running.

4. Related Work

4.1 Virtual Machines and I/O

As a common machine partitioning method, the virtual machines approach has been widely adopted. Many techniques have been proposed to reduce the overhead of the VMs, such as the paravirtualization technique 13), and hardware virtualization support 14),15).

One of the most successful VMMs for commodity machines is Xen 16), which makes use of the paravirtualization technique. In the paravirtualization technique, the guest kernels are modified to call the VMM, or the hypervisor in Xen terms, instead of privileged instructions and I/O. It uses this so-called "hypercall" to reduce the costs of trapping in the traditional virtualization method. **Figure 6** shows an example of partitioning a machine using Xen. In this figure, on the top of the hypervisor, there run two types of domains, guest operating systems: a privileged domain (Dom0), and guest domains (DomUs). The former domain holds all the physical devices, and the latter domains access the devices

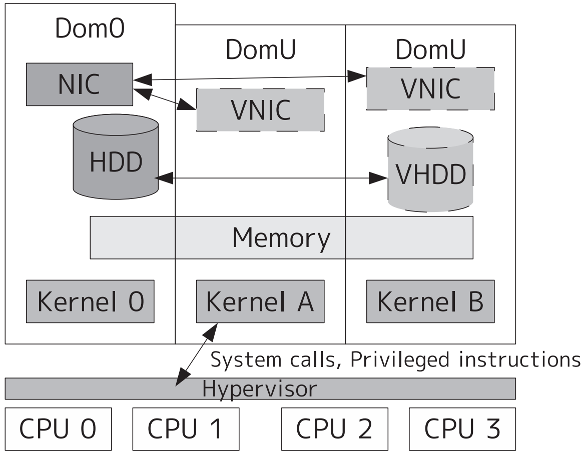


Fig. 6 Virtual machines and resources in Xen.

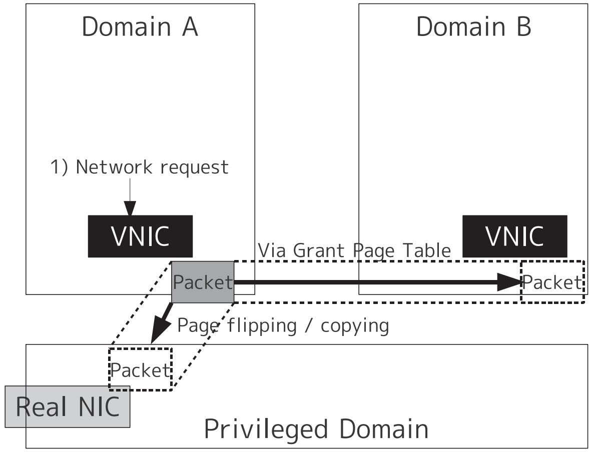


Fig. 7 Communication between domains in Xen.

via the privileged domain using virtual devices. System calls or privileged instructions are simulated by the hypervisor, thus the cost for this simulation still remains.

The other popular VMMs for the x86 architecture are VMWare [17] and Kernel-based Virtual Machine (KVM) [18]. VMWare uses dynamic binary modification of the privileged instructions and I/O processing of the guest operating systems. Therefore, it can run any operating system, including proprietary operating systems for which source codes are not generally available. KVM exploits the hardware virtualization support recently introduced in commodity machines, as well as the paravirtualization technique. It uses "trap and emulate" hardware support to run any operating system, but to reduce the costs of trapping, it also uses paravirtualization for some operating systems, such as Linux.

The device sharing and inter-domain communication in Xen achieve significant performance improvement. The virtual network devices in Xen communicate with each other by using a grant table [19]. **Figure 7** illustrates communication in Xen. Because each of the domains, except Dom0, is independent and unable to access the memory of others, the hypervisor provides the interface used to establish the grant page tables. The grant page tables are the list of pages to which a domain "grants" access to another domain. As a result, it achieves zero-copy communication between independent domains. Although the grant page table method achieves zero-copy communication, it incurs a cost for page table modification and managing the protection for the grant pages.

There are many ways to achieve better communication or device sharing among virtual machines on one physical host machine. XWAY [20] is an inter-domain socket architecture for Xen. It provides a user-transparent socket layer, which provides more direct access to the communication mechanism in Xen. The layer intercepts between INET and TCP and uses the XWAY channel to communicate with other domains. Therefore, it assures binary compatibility for applications that use sockets. Thus, it accomplishes faster inter-domain communications for applications running on Xen guests without any modification to those applications. However, it has the inevitable overhead for `connect` and `close` operations, and for some applications using the kernel socket interface directly. Thus, more work will be needed.

Huang, et al. [21] proposed efficient inter-domain communication in Xen, and integrated it to an MPI library. In this method, the receiver maps pages of the sender using a kernel driver from the MPI library, and achieves one-copy, i.e., a copy from the library to the user space, communication. Unlike a new socket, MPI is a widely used interface, and extension to the MPI library is an acceptable solution for high performance computing. The MPI library is, however, not used in all applications. To make efficient communication available generally, a virtual network device would be better.

Xen-IB [22] uses InfiniBand network cards, which are capable of user-level networking, in the virtual machine to provide fast communication on virtual machines. It enables user processes to communicate directly to the network device bypassing kernels and the hypervisor using the facility of user-level networking of InfiniBand. However, it requires a special hardware that supports user-level networking, and so it is not directly applicable to network devices in general.

4.2 Communication across Address Spaces

In the sense that the communication must be done across different address spaces, inter-process communication (IPC), inter-domain communication in Xen and the proposed IKC have the same issue. There are a large number of researches on IPC, and for example, the use of the shared memory is adapted as A-stack in LRPC[23], the page transfer is a common IPC technique in microkernels such as DASH[24] and L4[25].

Since these IPC techniques and inter-domain communication provide security, they require interference of an operating system or a hypervisor. If a CPU is shared among processes or virtual machines, it can incur switching of the virtual page tables. Under the assumption that the kernels can be trusted and that a CPU is dedicated to a kernel, the proposed IKC does not rely on such intervention, and can eliminate the overhead of the protection and switching of virtual address spaces.

4.3 Partitioning and Scalability

As for the partitioning of a machine, there are various methods other than virtual machines. Linux VServer[26] and OpenVZ[27] partition resources within an operating system. They provide virtual environments, each of which has a set of resources. This method is implemented in a single kernel. The applications in the environments provided by this method do not suffer from overhead for system calls and I/O, unlike VMs. However, the environments are still on the same kernel, these methods do not alleviate the issues of synchronization costs in the kernel.

Microvisor[28] provides online maintenance of operating systems of server machines. During the maintenance, an operating system is virtualized and a copy of the operating system runs concurrently. The copied OS updates itself, and after the maintenance, the processes in the original OS are migrated to the copied OS. Finally, the copied OS is devirtualized and continues providing services. This method provides the online maintenance within a single node, and reduces the cost of virtualization by devirtualizing during the normal operation. To run multiple operating systems, however, it uses virtualization, and does not meet demand of a server with multiple operating systems during normal operations without virtualization cost. Moreover, this method requires spare resources, such as network cards for maintenance, which is inefficient since they are not used during normal operation.

TwinOS[29] is a method that runs multiple Linux kernels on a uniprocessor machine. It divides memory and other devices, and dedicates them to the kernels, but time-shares the CPU. The time sharing of the CPU requires overhead for switching operating systems, and the target of TwinOS is not a multiprocessor environment, and so it does not offer the efficient use of multicore processors.

Many operating systems have been proposed to attain scalability in multicore environments by, for instance, reducing the cost of synchronization, or using the knowledge of memory distances, which are more often different from processor to processor, such as in ccNUMA machines.

Corey[30] is an operating system that focuses on scalability in multicore machines. It provides an interface to specify a separate memory area and a shared memory area, and a sharing of kernel objects, to reduce false sharing of memory and objects, and thus reduces wasteful locks. However, it does not improve scalability of existing operating systems like Linux, and applications must be modified or recompiled to work on Corey.

Barrelfish[31] is designed to have a knowledge base of the hardware, and provides user programs with access to it so that they can find the capabilities of the machine and request them, or they can notify the scheduler of the preferences for the functions. By providing a knowledge base, Barrelfish is able to run user programs properly on various machines, which are getting more and more diversified. However, it is hard to design and build such a knowledge base that describes enough information of machines, and the efficient use of the knowledge in the operating system still requires more investigation.

In AsyMOS[32] and Piglet[33], new methods are proposed to dedicate one of the CPU cores in a machine to a lightweight kernel, and run a single Linux operating system on the others. I/O and some system services are executed by the lightweight kernel. Their goal is to reduce cache contention by kernel codes and synchronization in kernels by assigning a dedicated core for the lightweight kernel. This can address the scalability issue of Linux, but the practical effect is not clear. Also, it is unable to concurrently run independent multiple operating systems on a machine.

4.4 Multiple Operating Systems

OS Switching[34] executes multiple operating systems natively by the “suspend” and “resume” power management features. Switching is done as follows: first, an operating system is suspended. Then, the switching code is executed. It restores hardware states and runs resume code for another operating system. This method enables each operating system to use the entire resources of a machine, but it may be seen as time sharing with a long time span, so it is not able to run multiple operating systems at the same time.

The KLoader module in SHIMOS resembles *coreboot* [35], formerly called LinuxBIOS [36], and other OS-based bootstrappers [37], which provide alternatives to the BIOS bootstrap programs. To support booting from various devices, they use a part of an operating system to boot other operating systems. This idea is similar to KLoader, but the bootstrap operating systems replace themselves with the new operating systems, while the kernel where the KLoader module is loaded stays alive after it boots the other kernels.

5. Evaluation

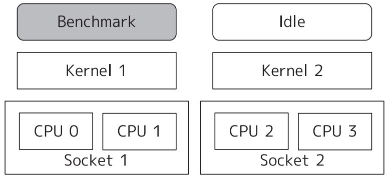
We evaluated the implementation of SHIMOS on Linux 2.6.26 for x86 described in Section 3. The evaluation environment is a dual-core dual-processor SMP machine (four cores in total) with three hard disks connected to a SATA controller, and a hard disk connected to a PATA controller. It is also equipped with a network interface card. The specification is shown in detail in **Table 2**.

We conducted benchmarks on the native kernel, on the kernels of SHIMOS, on the guest domains (DomUs) of Xen 3.2 with the OpenSuSE version of Linux 2.6.26 for the privileged domain (Dom0), and on the guest machines of KVM 62.

Let a benchmark execution environment be represented by the list of b/p where p is the number of CPU cores dedicated to the kernel, and b is the number of benchmark instances executed on the kernel. For example, $\{1/2, 1/2\}$ means that the evaluation machine is partitioned into two kernels each of which has two CPU cores and runs a benchmark program. To make this clear, the typical configurations are illustrated in **Fig. 8**. The size of memory dedicated to the kernel is proportional to the number of dedicated cores. When b is more than one, we show the average of scores of all the benchmark programs as the score for that configuration.

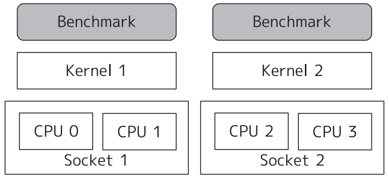
Table 2 The evaluation machine.

Model	DELL Precision 490
CPU	Intel Xeon 5130 (dual-core, 2.0 GHz) x 2
Memory	DDR2 667 MHz FB-DIMM 1,024 MB x 2
HDD	SATA 250 GB x 3, PATA 120 GB
OS	Linux 2.6.26
NIC	Intel (`e1000`, PCI-X)

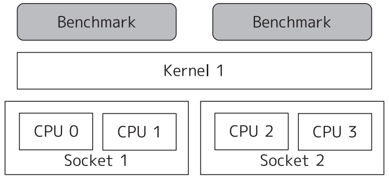


*CPU2 and CPU3 are stopped in the native case.

(a) {1/2} configuration



(b) {1/2, 1/2} configuration



(c) {2/4} configuration

Fig. 8 Typical configurations in this evaluation.

Two DomUs in the Xen case and two guest machines in the KVM case ran on the evaluation machine when we ran two kernels in the SHIMOS case. One of the kernels uses an SATA hard disk, 896 MB of memory and an `e1000` NIC; the other kernel has a PATA hard disk and 896 MB of memory. We used the *ext3* file system for all the hard disks, and we accessed via this file system otherwise noted. Although the memory size in the evaluation machine is 2,048 MB, we only use 1,792 MB in total. This is because the hypervisor or the host kernel uses a certain amount of memory in the virtual machines, while SHIMOS can use all of the memory available by giving 1,024 MB to each kernel. To make the conditions fair, we used 896 MB per kernel in SHIMOS, Xen and KVM. The native single kernel with four CPU cores had 1,792 MB of memory.

We used two methods for measurement of elapsed time: the `gettimeofday` system call and the time stamp counter of the CPU. The system call is called or the counter is read before and after the benchmark. It may incur a small cache miss for the system call, or may effect on the CPU pipeline, but the side effects of these time measurements are ignorable. The precision was 1 ms for the system call, and one clock (i.e., 0.5 ns) for the counter. We disabled the DVFS feature of the machine to obtain a constant clock.

First, we present the microbenchmark results to show the overhead of the SHIMOS mechanism compared to the virtual machines and the effect of partitioning on the kernel behavior. We show the performance of the virtual network and block devices next. Then, we experiment using actual applications to have a look at the overall performance of the SHIMOS mechanism.

5.1 Microbenchmarks

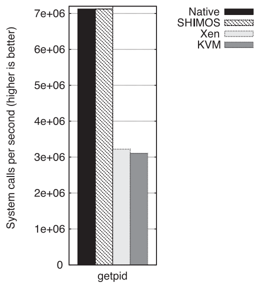
5.1.1 System Calls

The existence of overhead upon execution of privileged instructions is the major downside of virtual machines. In this section, we measured the overhead by calling system calls many times. We used the `gettimeofday` system call to measure the time.

First, the time required for calling the simple system call `getpid` ten million times is measured. The number of system calls per a second is shown in **Fig. 9**. It shows that solely changing the privilege level costs a lot, slowing the execution time by about 56% in both VMs, while SHIMOS results in the same execution time as the native single kernel.

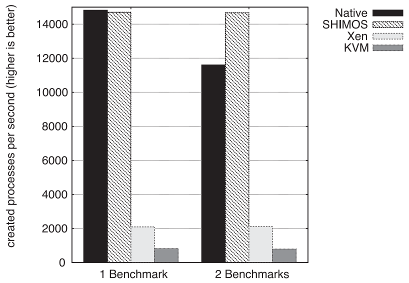
The *forkwait* benchmark is a simple loop of the `fork` and `waitpid` system calls, that is, a loop of process creation and destruction. **Figure 10** shows the results of this benchmark. It shows that the virtual machines get much worse scores than in the *getpid* benchmark.

It is also remarkable that SHIMOS achieved a 1.26 times better score than the native kernel when two benchmarks are executed in parallel (the $\{1/2, 1/2\}$ and $\{2/4\}$ configurations). To clarify the reason, we examined the native case further by changing the number of CPUs and the binding of CPUs. The results are shown in **Fig. 11**. As the number of CPUs exceeds one, and then two, when the kernel switches to SMP and starts to use a core on the second CPU socket, the execution time gets much longer, but the time with four cores is not very different from that with three cores. When we bind the benchmark process and its children to a core, CPU 0, the difference between two and three cores almost disappears. This suggests that the degradation in the three cores is caused by an inter-socket cache protocol. When more than one CPU socket is used, the scheduler may choose another core in the different CPU socket for a newly created process, thus resulting in inter-socket communication. Therefore, when we bind



The configuration is $\{1/2\}$ in all the cases.

Fig. 9 The `getpid` benchmark results.



The configurations are $\{1/2\}$ for the "1 benchmark" cases, $\{2/4\}$ for the Native case in "2 benchmarks", and $\{1/2, 1/2\}$ for the others in the "2 benchmarks" cases.

Fig. 10 The `forkwait` benchmark results.

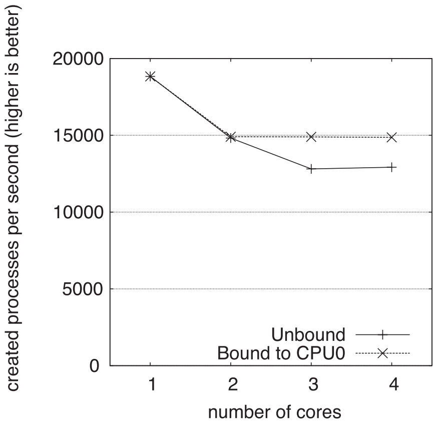


Fig. 11 The `forkwait` results and the number of cores.

Table 3 The native `forkwait` benchmark results with binding.

	Unbound	Each socket	The same socket
# of created procs	11,600	11,631	13,678

The configuration is {2/4}.

the benchmark instances to the configuration similar to the SHIMOS case, that is, each benchmark for each CPU socket, the score is worse than the score when we bind both benchmark instances to the same socket (see **Table 3**). Still, the best score in that situation is about 5% worse than the score of SHIMOS.

5.1.2 IKC Primitive Performance

In this section, we evaluate the performance of inter-kernel communication in SHIMOS. We measured the round-trip time and the number of IKC packets transmitted by writing a small kernel module that uses IKC. Because the manipulation of IKC packets is totally a kernel activity, we used time stamp counter to measure the elapsed time.

The modules are loaded to both kernels, one of them sends an IKC packet to the other and waits for receiving an IKC packet from the other. During the measurement, the module does not wake up another thread but sends back a packet in the reception interrupt handler, and so it measures nearly pure overhead of the IKC. The module also bursts 1,000,000 packets to the other kernel, and the receiving module measures the time for receiving the all the packets in order to evaluate how many packets can be transmitted in certain time. The result is shown in **Table 4**.

Table 4 Results of the IKC primitive benchmark.

RTT	2.723 μs
# of packets per second	1.209×10^6

Table 5 The `ping` benchmark results.

	Native	SHIMOS	Xen	KVM
ping RTT	5 μs	10 μs	66 μs	4,043 μs

5.1.3 Network Performance

In this section, we evaluate the performance of the virtual network devices. We used the same configuration as the system call benchmarks, with the addition of a virtual network device in SHIMOS, bridged to the physical NIC in the first kernel. For the virtual machines, the virtual network devices for both kernels are connected via a bridge in the host. For the native kernel, a loopback device is used for benchmarks in this section. We used a default configuration for network packet size (MTU = 1500).

Table 5 shows the round-trip times (RTTs) of inter-kernel communication measured by the `ping` command with a microsecond precision, and **Fig. 12** shows the various bandwidths measured by the `iperf` program 38). Excluding the loopback device, the RTT of SHIMOS is the best among the virtual devices. However, with regard to the bandwidth, it is worse than Xen. When we turn off the TSO (TCP Segmentation Offloading) feature in the Xen network devices, the performance in Xen becomes worse than SHIMOS. Therefore, it is a problem of optimization of the network device, whether the TSO feature is simulated or not. The TSO effect in Xen is pointed out by the papers 39),40).

Comparison between the loopback device and SHIMOS indicates that half of the latency in the SHIMOS virtual network device seems to originate from the IKC and the virtual network implementation. Also, from the results in Table 4, it seems that the SHIMOS network device implementation consumes about half

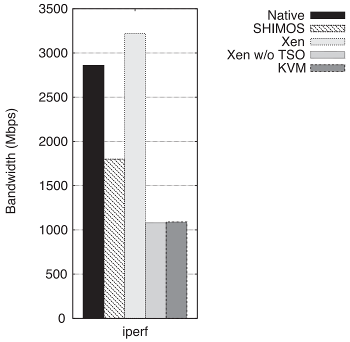


Fig. 12 The `iperf` benchmark results.

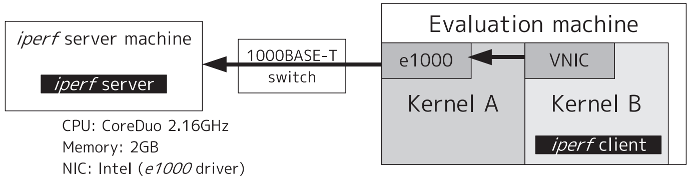


Fig. 13 Evaluation configuration for measurement of virtual network effects.

of the processing time. Comparing the maximum IKC packet transmission rate in Table 4, there seems to be room for improvement of the network device implementation, since one IKC packet corresponds to a network packet.

Next, we experimented with the effect of the virtual network devices. The configuration is illustrated in **Fig. 13**. We measured the bandwidth using the `iperf` program from a kernel to a machine outside the environment via a virtual network device. The results in **Fig. 14** show that SHIMOS and Xen exhibit no performance degradation in the network, while KVM shows a decrease in bandwidth of about 5.8%.

5.1.4 Disk Performance

To evaluate the virtual block devices in SHIMOS, we first conducted a measurement of the speed of reading and writing to a RAM disk. Since a simple memory access is faster than inter-kernel communication and faster than operations involving disks, this measures the potential speed of the virtual block device. We created a RAM disk on a kernel or on a host kernel, and shared it using a virtual block device with another kernel or with a guest kernel. We created a program to read and write data to the specified device with raw access (i.e., via no file system). The benchmark program enabled the `O_DIRECT` option to remove the effects of disk caches in the operating system. We compared the speed of reading and writing 2 GB of data to the device in each environment. The result is shown in **Fig. 15**. In this figure, SHIMOS achieves far better performance among the virtual devices. The virtual block device in SHIMOS performs 3.98 times better than the one in Xen, and 6.75 times better than the one in KVM.

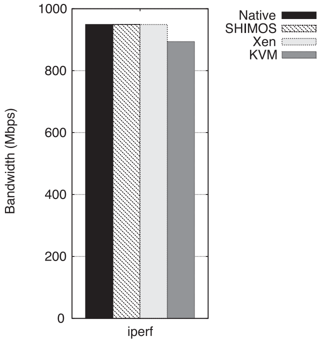


Fig. 14 Effects on network performance of virtual devices.

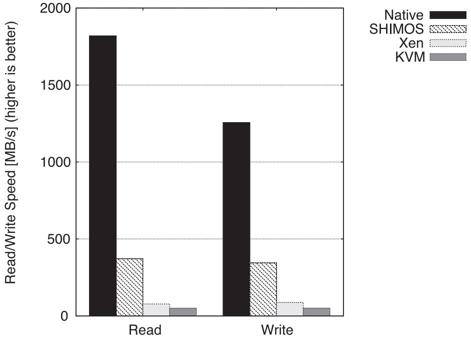


Fig. 15 Results of RAM disk read/write tests via virtual devices.

Table 6 Results of Linux compilation by various methods.

	Elapsed time (s)
Native (Direct)	205.845
SHIMOS	206.146
iSCSI	206.856
NFS	220.766

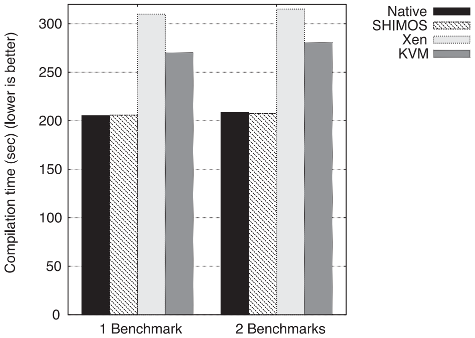
Next, we tried to clarify the significance of the virtual block device in SHIMOS compared to some alternative methods of remote hard disk sharing by network because real hard disks are typically slower than the network. We conducted a Linux compilation benchmark experiment in a kernel using an SATA hard disk on another kernel with various methods in SHIMOS: the proposed virtual block device, iSCSI[41)] and NFS[42)]. The last two methods use the virtual network device for communication. The comparison of these methods and the direct use of the hard disk is shown in **Table 6**. The elapsed time is measured by the `gettimeofday` system call with a millisecond precision. This table shows that the performance of the shared block device in SHIMOS is the best one among the sharing methods using network.

5.2 Application-level Benchmarks

5.2.1 Linux Compilation

The Linux compilation benchmark is to measure the time it takes to compile the Linux kernel. It accesses disks and produces many child processes. We measured the time needed to compile the Linux 2.6.26 kernel of the default configuration with four concurrent processes. When we ran two compile jobs in parallel, we used different hard disk drives connected to different hard disk controllers for the compilations, namely, an SATA hard disk and a PATA hard disk. The elapsed time is measured by calling `gettimeofday` system calls with millisecond precision before and after execution of a `make` process.

The results of the Linux compilation benchmark are shown in **Fig. 16**. In this figure, "1 benchmark" denotes a single benchmark, and "2 benchmarks" denotes the parallel execution of the two benchmarks. In the single execution, SHIMOS is a little (0.3%) worse than the native kernel because of the existence of another kernel. However, the results of multiple executions show that SHIMOS is slightly (0.5%) better than the native single kernel. This slight difference in multiple executions is statistically significant according to a statistic test called Welch's test at the significant level of 1.0% with the detailed data shown in **Table 7**.



The configurations are $\{1/2\}$ for the "1 benchmark" results. $\{2/4\}$ for the Native case in "2 benchmarks" and $\{1/2, 1/2\}$ for the others in the "2 benchmarks" results.

Fig. 16 The Linux compilation results.

Table 7 Results of compilation benchmark in Native and SHIMOS environments.

	Native	SHIMOS
Mean	208.3918	207.335
Variance	3.754×10^{-2}	0.1013
Number of Samples	10	10

Elapsed time in second. Smaller is better.

5.2.2 SPEC CPU Benchmark

Next, we ran the *int* set of the SPEC CPU2006 Benchmark[43)] on the evaluation machine. This benchmark contains 12 sub-benchmarks that mainly stress CPUs and memory. We ran two instances of the SPECint_2006 benchmark on the evaluation machine, and the configurations were $\{2/4\}$ for native, $\{1/2, 1/2\}$ for the others. The total score is shown in **Fig. 17**, and the scores for sub-benchmarks are presented in **Fig. 18**. Note that the results are not officially reportable SPECCPU 2006 scores because they do not satisfy the hardware description criteria and they have some unknown flags for benchmark program compilation.

The results showed that SHIMOS achieves the best score in all sub-benchmarks and, consequently, the best total score in this evaluation. As for the total

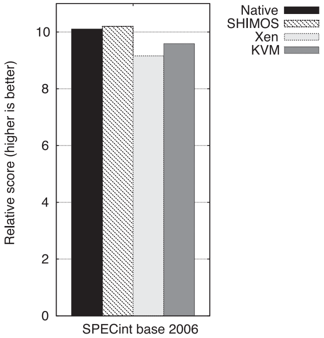


Fig. 17 The SPECint 2006 base results.

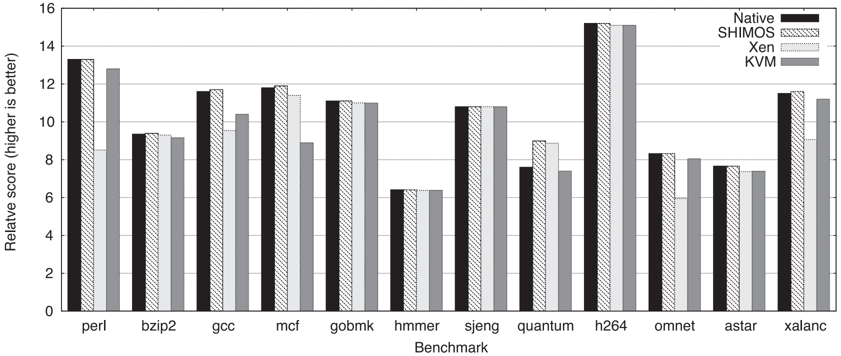


Fig. 18 Detailed results for SPECint_base2006.

scores, SHIMOS scores 6.2% better than KVM, the best of the VMs, and scores 1.8% higher than the native single kernel. In the sub-benchmarks, the scores of SHIMOS are from 0.8% to 16% better than the native kernel, or at least, identical.

To clarify the performance difference between the native and SHIMOS cases, the results in detail are shown in **Table 8**. By applying Welch's test at 1.0% significance level to the results, there is statistically performance difference.

5.2.3 Mixed Jobs

Table 8 Results of SPECint benchmark in Native and SHIMOS environments.

	Native	SHIMOS
Mean	10.2	10.021
Variance	0	7.104×10^{-3}
Number of Samples	5	6

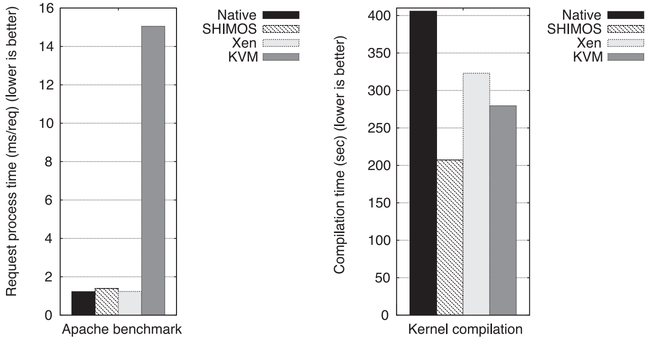


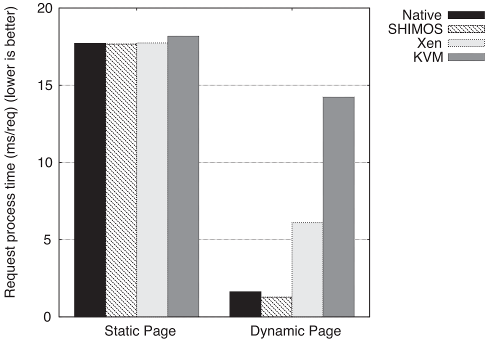
Fig. 19 The mixed benchmark results.

Next, we conducted a mix of benchmark experiments. We ran an Apache server and Linux compilation at the same time on one machine. While the kernel was being compiled, we executed an Apache benchmark[44], which sends requests to the Apache server, and the effect on the results of each was observed. The Apache benchmark program sent requests to a dynamic page, a CGI, which simply shows the current date. The Apache benchmark setting was 5 concurrent connections for static, 10 concurrent connections for dynamic, and the Linux compilation setting was the same as that of the previous evaluation. Note that the Apache server and Linux compilation used the different hard disks in all cases (an SATA hard disk and a PATA hard disk, respectively).

The results are shown in **Fig. 19**. Although the Apache score for SHIMOS is about 10% worse than those in the native kernel and Xen, we must point out that the Linux compilation time in the native kernel is about twice the time in SHIMOS. We consider that the CPUs were used to process network rather than the compilation in the native case, and so the compilation time was far worse than the single benchmark, while the Apache score decreased slightly.

5.2.4 Apache Benchmark

To evaluate the virtual network device by application, we ran two Apache



The configuration is $\{2/4\}$ for the Native case, and $\{1/2, 1/2\}$ for the others.

Fig. 20 The Apache Benchmark results.

servers, one in each kernel. The network configuration was the same as the one shown in Fig. 13. The *iperf* server in this figure was used to run the Apache benchmark. In the native case, we ran two Apache process sets in a kernel with different port numbers. We sent requests from the Apache benchmark programs to the two servers concurrently. The Apache benchmark setting is the same as that of the previous benchmark.

The static page used in this benchmark is a one-megabyte file, and the dynamic page is the same as the one in the previous benchmark. The results are shown in **Fig. 20**. Since the static page access does not have an impact on CPUs, and the bandwidth is limited by the physical network device, the results did not show much difference among the SHIMOS and the VMMs. In contrast, the dynamic page access does stress CPUs, and the performance decreases in the cases other than SHIMOS, due to the loads attributable to the dynamic pages. SHIMOS achieves 375% better performance than Xen, and 22% better than the native single kernel. Note that the Apache score in Xen drops to 4.93 times worse than the previous benchmark. It may be because the load on the virtual device certainly affected the performance of the Apache servers.

5.3 Discussion

Compared to the Xen virtual machine, the SHIMOS kernel achieves better performance in most benchmarks. The SPEC int2006 benchmark showed that not only I/O oriented programs, but CPU and memory intensive programs can perform better in SHIMOS than on the virtual machines. The benchmarks where SHIMOS performs worse are those involving the virtual network bandwidth and the request process speed in the Apache benchmark. The former reflects the fact that the virtual network device is more optimized in Xen. The latter is related to the configuration, in that virtual machines in Xen are not bound to one CPU socket, thus the cache available to them may vary. However, the overall performance of the mixed jobs seems to be better in SHIMOS.

As for the network performance, the result without TSO shows that the number of transmitted packets in SHIMOS are more than in Xen. Also, from the IKC primitive benchmark, the transmission capacity of IKC packets are more than the transmitted network packets, and so the SHIMOS can achieve a higher performance by applying some optimization techniques to the network device implementation. We expect that reducing interrupts and TSO are clearly applicable, and at least, TSO can improve the performance because it can increase the size of data in a packet and decrease the processing time of the packetization.

On the other hand, the kernels in SHIMOS can achieve better performance than that of a single native kernel when some of the benchmarks are executed in parallel. As described in Section 5.1.1, the parallel process creation in the native Linux kernel performs worse than the single process creation in the native Linux kernel, and than the concurrent creation in the separate SHIMOS kernels. We consider that it is because accesses to shared resources in the process creation procedure were contended. Also, as Fig. 11 and Table 3 imply, parallel access to the shared resources from different CPU sockets can be a reason of the slowdown in the Linux kernel, because the serialization in such a situation requires inter-socket communication and cache misses. The compilation benchmark also generates many processes, and the worse performance in the Linux kernel should be due to the same reason as the `forkwait` benchmark. As for the other application benchmarks, it may be the shared resource accesses in the other part of the kernel and the I/O and process scheduling in one kernel. We consider that it is implied from the mixed benchmark results that the compilation in native kernel performs poorly when it runs with Apache benchmark. In the benchmark, the I/O load on different parts, the network and the disk, seemed to pose performance degradation on, at least, one side.

6. Conclusion

There have been demands for a multiple operating system environment running on a single multicore machine because it enables users to run applications that require different operating system environments, provides separation of multiple environments for users, and reduces contention in operating system kernels by reducing the synchronization costs. Virtual machines have been recognized as one promising approach. However, VMs entail overhead when simulating privileged CPU instructions and I/O.

There are two main contributions in this paper, as described below. The first is that a new and efficient mechanism, SHIMOS, providing multiple kernels in a multicore machine, has been designed and integrated into the Linux kernel without any specific architectural support. In SHIMOS, all the kernels run in the privileged CPU mode under the assumption that kernels do not break other kernels, and thus no extra overhead is incurred, unlike with VMs. Thus, an inter-kernel communication mechanism has been designed and implemented to create virtual devices, as explained in this paper. In order to provide efficient kernel data transfer mechanisms, two functions, a shared memory transfer function and a page transfer function, have been designed. The former function is mainly used to create the virtual network or character devices, and the latter function is mainly used to create virtual block devices.

The other contribution in this paper is to demonstrate that the SHIMOS Linux achieves better performance than existing virtual machines, such as Xen and KVM, and that it performs better than a single Linux kernel in some benchmarks. We have evaluated the implementation and showed that it performs about 7 times faster than the Xen virtual machines for process creation and destruction system calls. We have also shown that a virtual network device using inter-kernel communication is as fast as the native case, and achieved 1.67 times better inter-kernel bandwidth compared to Xen without TSO. And, we have shown that the virtual block device performs 3.98 times better than Xen, and has almost the same performance as a real hard disk drive. Finally, we have presented the results of application-level benchmarks. Using the multiple Linux compilation benchmarks, we have shown that SHIMOS is 1.35 times faster than Xen, and 1.006 times faster than the native single Linux. Using the SPECint 2006 benchmarks, we have shown that SHIMOS is 1.06 times better than KVM and 1.10 times better than Xen, and also 1.01 times better than the native single kernel. We have also shown that SHIMOS with a virtual network device is 4.77 times faster than Xen and 1.26 times faster than the single Linux in the Apache benchmark. With all of the benchmarks, with the exception of the SPECint benchmark, KVM performs worse than SHIMOS.

As future work, we need to further investigate the scalability of the method we have proposed. We used four core machines for the experiments, and we used the spinlock method for our shared memory. Scalability issues, such as the effect of cache consistent protocols in these shared memory accesses from multiple kernels, should be examined, especially when the number of cores is increased to, for example, 16 or more.

Acknowledgments This work is partially supported by a CREST project of JST (Japan Science and Technology).

References

1) Intel: Intel Core i7 Processor Extreme Edition and Intel Core i7 Processor, http://download.intel.com/design/processor/datashts/320834.pdf.
2) Advanced Micro Devices: AMD Opteron Processor Product Data Sheet, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/23932.pdf.
3) Intel: Intel Atom Processor Z5xx Series Datasheet, http://download.intel.com/design/processor/datashts/319535.pdf.
4) ARM: ARM11 MPCore, http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html.
5) Yoshida, Y., Kamei, T., Hayase, K., Shibahara, S., Nishii, O., Hattori, T., Hasegawa, A., Takada, M., Irie, N., Uchiyama, K., Odaka, T., Takada, K., Kimura, K. and Kasahara, H.: A 4320MIPS Four-Processor Core SMP/AMP with Individually Managed Clock Frequency for Low Power Consumption, *2007 IEEE International Solid-State Circuits Conference* (*ISSCC2007*), pp.100–101, 590 (2007).
6) Borden, T.L., Henessy, J.P. and Rymarczyk, J.W.: Multiple operating systems on one processor complex, *IBM Systems Journal*, Vol.28 No.1, pp.104–123 (1979).
7) Meyer, R.A. and Seawright, L.H.: A virtual machine time-sharing system, *IBM Systems Journal*, Vol.9, No.3, pp.199–218 (1970).
8) International Business Machines Corporation: *Logical Partitions on System i5: A*

Guide to Planning and Configuring LPAR with HMC on System i (2006).
9) Hewlett-Packard Company: *Installing and Managing HP-UX Virtual Partitions* (*vPars*) *Ninth Edition* (2006).
10) Popek, G.J. and Goldberg, R.P.: Formal Requirements for Virtualizable Third Generation Architectures, *Comm. ACM*, Vol.17, No.7, pp.412–421 (1974).
11) Alam, A., Barrett, R., Bast, M., Fahey, M.R., Kuehn, J., McCurdy, C., Rogers, J., Roth, P., Sankaran, R., Vetter, J.S., Worley, P. and Yu, W.: Early evaluation of IBM BlueGene/P, *Proc. 2008 ACM/IEEE Conference on Supercomputing* (2008).
12) Shimosawa, T., Matsuba, H. and Ishikawa, Y.: Logical Partitioning without Architectural Supports, *IEEE International Computer Software and Applications Conference*, pp.355–364 (2008).
13) Whitaker, A., Shaw, M. and Gribble, S.D.: Scale and performance in the Denali isolation kernel, *OSDI '02: Proc. 5th Symposium on Operating Systems Design and Implementation*, New York, NY, USA, pp.195–209, ACM (2002).
14) Advanced Micro Devices: *AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual* (2005).
15) Neiger, G., Santoni, A., Leung, F., Rodgers, D. and Uhlig, R.: Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization, *Intel Technology Journal*, Vol.10, Issue 3, pp.167–177 (2006).
16) Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the art of virtualization, *Proc. ACM Symposium on Operating Systems Principles*, pp.164–177 (2003).
17) Adams, K. and Agesen, O.: A comparison of software and hardware techniques for x86 virtualization, *ASPLOS-XII: Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, pp.2–13, ACM (2006).
18) Qumranet Inc.: KVM—Kernel-based Virtualization Machine, http://www.qumranet.com/files/white_papers/KVM_Whitepaper.pdf.
19) Fraser, K., Hand, S., Neugebauer, R., Pratt, I., Warfield, A. and Williamson, M.: Safe hardware access with the Xen virtual machine monitor, *Proc. 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure* (*OASIS*) (2004).
20) Kim, K., Kim, C., Jung, S.-I., Shin, H.-S. and Kim, J.-S.: Inter-domain socket communications supporting high performance and full binary compatibility on Xen, *VEE '08: Proc. fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp.11–20, ACM (2008).
21) Huang, W., Koop, M.J. and Panda, D.K.: Efficient one-copy MPI shared memory communication in Virtual Machines, *2008 IEEE International Conference on Cluster Computing*, pp.107–115 (2008).
22) Liu, J., Huang, W., Abali, B. and Panda, D.K.: High performance VMM-Bypass I/O in virtual machines, *Proc. USENIX 2006 Annual Technical Conference*, pp.29–42 (2006).
23) Bershad, B.N., Anderson, T.E., Lazowska, E.D. and Levy, H.M.: Lightweight remote procedure call, *ACM Trans. Comput. Syst.*, Vol.8, No.1, pp.37–55 (1990).
24) Tzou, S.-Y. and Anderson, D.P.: The Performance of Message-passing using Restricted Virtual Memory Remapping, *Software—Practice And Experience*, Vol.21, No.3, pp.251–267 (1991).
25) Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S. and Wolter, J.: The Performance of μ-Kernel-Based Systems, *16th ACM Symposium on Operating Systems Principles* (*SOSP '97*) (1997).
26) Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A. and Peterson, L.: Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors, *EuroSys '07: Proc. 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, New York, NY, USA, pp.275–287, ACM (2007).
27) OpenVZ Development Team: OpenVZ, http://wiki.openvz.org/.
28) Lowell, D.E., Saito, Y. and Samberg, E.J.: Devirtualizable virtual machines enabling general, single-node, online maintenance, *Proc. 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.211–223 (2004).
29) Tabuchi, M., Itoh, K., Nomura, Y. and Taniguchi, H.: Design and Evaluation of a System for Running Two Linuxes Coexistently, *IECIE Trans. Inf. Syst.* (*Japanese Edition*), Vol.J88-D1, No.2, pp.251–262 (2005).
30) Wickizer, S.B., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y. and Zhang, Z.: Corey: An operating system for many cores, *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI '08*), pp.43–57 (2008).
31) Schüpbach, A., Peter, S., Baumann, A., Roscoe, T., Barham, P., Harris, T. and Isaccs, R.: Embracing diversity in the Barrelfish manycore operating system, *Workshop on Managed Multi-core Systems* (*MMCS 08*) (2008).
32) Muir, S. and Smith, J.: AsyMOS—an asymmetric multiprocessor operating system, *Proc. Open Architectures and Network Programming*, pp.25–34 (1998).
33) Muir, S. and Smith, J.: Functional divisions in the Piglet multiprocessor operating system, *Proc. 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, pp.255–260 (1998).
34) Sun, J., Zhou, D. and Longerbeam, S.: Supporting Multiple OSes with OS Switching, *USENIX Annual Technical Conference*, pp.357–362 (2007).
35) coresystems: coreboot, http://www.coreboot.org/.
36) Agnew, A., Sulmicki, A., Minnich, R. and Arbaugh, W.: Flexibility in ROM: A Stackable Open Source BIOS (2003).
37) Minnich, R.G.: Give your bootstrap the boot: Using the operating system to boot the operating system, *CLUSTER '04: Proc. 2004 IEEE International Conference on Cluster Computing*, Washington, DC, USA, pp.439–448, IEEE Computer Society

(2004).

38) Gates, M., Tirumala, A., Ferguson, J., Dugan, J., Qin, F., Gibbs, K. and Estabrook, J.: iperf Project Page, http://sourceforge.net/projects/iperf.

39) Menon, A., Santos, J.R., Turner, Y., Janakiraman, G.J. and Zwaenepoel, W.: Diagnosing performance overheads in the xen virtual machine environment, *VEE '05: Proc. 1st ACM/USENIX International Conference on Virtual Execution Environments*, New York, NY, USA, pp.13–23, ACM (2005).

40) Menon, A., Cox, A.L. and Zwaenepoel, W.: Optimizing network virtualization in Xen, *Proc. USENIX 2006 Annual Technical Conference*, pp.15–28 (2006).

41) Satran, J., Meth, K., Sapuntzakis, C., Chadalapaka, M. and Zeidner, E.: Internet Small Computer Systems Interface (iSCSI), RFC 3720.

42) Callaghan, B., Pawlowski, B. and Pawlowski, B.: NFS Version 3 Protocol Specification, *Network Working Group RFC 1813* (1995). http://www.ietf.org/rfc/rfc1813.txt.

43) Standard Performance Evaluation Corporation: SPEC CINT2006 Benchmarks, http://www.spec.org/cpu2006/CINT2006/.

44) The Apache Software Foundation: ab—Apache HTTP server benchmarking tool, http://httpd.apache.org/docs/2.2/programs/ab.html.

(Received May 11, 2009)
(Accepted September 7, 2009)

Taku Shimosawa received his B.S. and Master of Information Science and Technology degrees from the University of Tokyo in 2007 and 2009, respectively. He is currently a Ph.D. candidate of Graduate School of Information Science and Technology, the University of Tokyo. His interests include operating systems on multicore machines and system software.

Yutaka Ishikawa is a professor of the University of Tokyo, Japan. Ishikawa received his B.S., M.S., and Ph.D. degrees in electrical engineering from Keio University. From 1987 to 2001, he was a member of AIST (former Electrotechnical Laboratory), METI. From 1993 to 2001, he was the chief of Parallel and Distributed System Software Laboratory at Real World Computing Partnership. His interests include the next generation supercomputer, cluster technologies, dependable parallel and distributed systems.