SkyBridge: Fast and Secure Inter-Process Communication for Microkernels

Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, Haibo Chen
Shanghai Key Laboratory for Scalable Computing Systems
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Abstract

Microkernels have been extensively studied over decades. However, IPC (Inter-Process Communication) is still a major factor of run-time overhead, where fine-grained isolation usually leads to excessive IPCs. The main overhead of IPC comes from the involvement of the kernel, which includes the direct cost of mode switches and address space changes, as well as indirect cost due to the pollution of processor structures.

In this paper, we present SkyBridge, a new communication facility designed and optimized for synchronous IPC in microkernels. SkyBridge requires no involvement of kernels during communication and allows a process to directly switch to the virtual address space of the target process and invoke the target function. SkyBridge retains the traditional virtual address space isolation and thus can be easily integrated into existing microkernels. The key idea of SkyBridge is to leverage a commodity hardware feature for virtualization (i.e., VMFUNC) to achieve efficient IPC. To leverage the hardware feature, SkyBridge inserts a tiny virtualization layer (***Rootkernel***) beneath the original microkernel (***Subkernel***). The Rootkernel is carefully designed to eliminate most virtualization overheads. SkyBridge also integrates a series of techniques to guarantee the security properties of IPC.

We have implemented SkyBridge on three popular open-source microkernels (seL4, Fiasco.OC, and Google Zircon). The evaluation results show that SkyBridge improves the speed of IPC by 1.49x to 19.6x for microbenchmarks. For real-world applications (e.g., SQLite3 database), SkyBridge improves the throughput by 81.9%, 1.44x and 9.59x for the three microkernels on average.

ACM Reference Format:
Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, Haibo Chen. 2019. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3302424.3303946

1 Introduction

Microkernels have been extensively studied over the past four decades [**? ? ? ? ? ? ? ? ?**]. The key design is to deprivilege most kernel functionalities into different servers residing in isolated user processes. The kernel provides basic functionalities, such as process management, capability enforcement and inter-process communication (IPC). Such a decentralized design makes the OS architecture robust against run-time errors, which means a fault within one server would not affect other servers and the kernel. Removing most functionalities from the kernel also results in a small Trusted Computing Base (TCB), making it less vulnerable to attacks and possible for comprehensive formal verification [**?**]. Given such advantages, microkernels [**? ?**] are widely used in various areas where high reliability matters, such as aerospace, automotive and medical devices.

In a microkernel, any communication between different user processes is based on IPC, which is an intensively-used operation. For example, if a client process writes data into an external block device, it first communicates with the file system, which in turn notifies the disk device driver to write data into the block device. All the communication is done via IPC. In fact, IPC is known as a major factor of run-time overhead [**? ? ? ?**], which determines the performance of applications on microkernels. Transferring control across process boundaries is expensive, which requires at least: a trap into the microkernel (SYSCALL instruction), data copying for arguments, one address space switch (even two switches if considering the recent Meltdown attack [**?**]), and an upcall back to the user level. Such operations must be repeated upon IPC return. Some asynchronous implementation of IPC even involves costly scheduling work.

A large body of research has been done to optimize the IPC performance. Software-based solutions try to shorten the IPC path by removing unnecessary operations. seL4 [**?**] uses the IPC fastpath for the case of Call and ReplyWait system calls where the IPC message fits in CPU registers, and no capabilities are transferred. For a fastpath, the message will be sent immediately and the control flow will be directly transferred without entering into the costly scheduling logic. Similarly, some software-based solutions like LRPC [**?**] also eliminate the scheduling overhead and allow a process's thread to execute requested procedures in the receiver's address space. However, all such approaches still require the involvement of the kernel and thus their performance (around 1000 cycles for an IPC roundtrip) do not satisfy the requirement of IPC-intensive workloads, as shown in Section 2. Hardware-based solutions propose new hardware extensions to boost IPC operation. dIPC [?] puts all IPC participants into a single address space, and the kernel is removed from the IPC path. The process isolation is achieved by the newly designed tagged memory. Such hardware-based solutions usually require non-trivial modification to both hardware and software, which have less potential for practical adoption.

Therefore, we argue that there is a need for an IPC technique that satisfies the following requirements.

- **Efficient**: the IPC path does not involve the kernel.
- **Lightweight**: the IPC can be readily deployed on commodity hardware and can be easily integrated into existing microkernel architecture.
- **Secure**: the IPC design does not break the microkernel isolation abstraction.

In this paper, we present a new IPC design that meets such requirements. Our design, called SkyBridge, allows one process (sender) to directly execute the requested procedure in another process's (receiver) address space without trapping into the kernel. SkyBridge has two main technical advantages. First, SkyBridge still places each process in its own virtual address space which fits well with the design and implementation of existing microkernels. Second, SkyBridge leverages one Intel hardware feature for virtualization, named EPT (extended page table) switching (the VMFUNC instruction), to change the virtual address space at the user level. By configuring the receiver's EPT, SkyBridge maps the page table of the sender to that of the receiver. Therefore, after switching the EPT by VMFUNC, the hardware uses the receiver's page table to translate all subsequent virtual addresses. SkyBridge also provides a separated stack for each receiver's thread in its virtual address space. To support long IPC, SkyBridge provides shared buffers for the IPC participants when large messages are transferred. Each buffer is bound to one receiver's thread for concurrency.

Although the approach of SkyBridge sounds intuitive, applying it to the microkernels imposes three practical challenges. First, leveraging EPT switching requires the virtualization layer, which may bring overhead to the whole system since the new layer could cause a large number of costly VM exits. To address this challenge, SkyBridge introduces a tiny virtualization layer (called ***Rootkernel***) only consisting of the most primitive functionalities for SkyBridge while eliminating VM exits during an IPC.

Second, existing ways [? ?] of leveraging VMFUNC require non-trivial modification to the microkernels and thus tremendous engineering effort. SkyBridge proposes a lightweight method to efficiently switch virtual address spaces among different processes which can be easily integrated into microkernel architectures.

Third, it is difficult to design a secure IPC facility without the involvement of the kernel, especially when one malicious process can exploit the VMFUNC instruction to corrupt other processes [?]. SkyBridge guarantees that there is only one legal entry point for switching address spaces among processes, which prevents a malicious process from invoking self-prepared VMFUNC instructions to corrupt other processes. SkyBridge also requires a process to register to other processes before communicating with them and introduces a calling-key table mechanism to enforce such a policy.

We have implemented SkyBridge on three different microkernels (seL4 [?], Fiasco.OC [?] and Google Zircon [?]) and deployed them on a commodity Intel Skylake machine. Our evaluation shows that SkyBridge significantly improves the performance of IPC by 1.49x, 5.86x, and 19.6x for seL4 (fastpath), Fiasco.OC and Zircon respectively. For a real-world application like a multi-tier SQLite3 workload, SkyBridge improves the performance by 81.9%, 1.44x and 9.59x for such three microkernels on average.

Contributions. The contributions of the paper are summarized as follows:

- A detailed analysis of the performance overheads of IPC in state-of-the-art microkernels.
- A new design which can significantly improve the performance of the microkernel IPC without any modification to the hardware.
- An implementation of SkyBridge and an evaluation using real-world benchmarks on three different microkernels.

2 Motivation and Background

2.1 Deconstructing Synchronous IPC

In this section, we evaluate the performance costs associated with the traditional synchronous inter-process call (IPC) in microkernels. We use seL4 [?] (v10.0.0) on an Intel Skylake processor to conduct all the experiments. seL4 is known to have a fast IPC facility, which we believe can represent state-of-the-art microkernels.

Although integrated with different optimization techniques, the current implementation of synchronous IPC still negatively impacts the performance of microkernel workloads, which are usually IPC-intensive. The synchronous IPC overheads can be classified into two categories: one is the direct cost from the kernel, and the other is the indirect pollution of processor structures.

2.1.1 Direct Cost of Microkernels

Mode Switch. For each IPC, the sender first invokes a *SYSCALL* instruction to trap into the kernel, which then saves

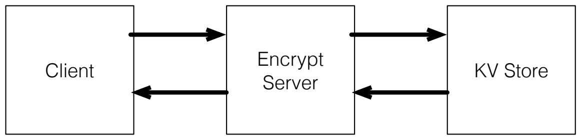


Figure 1. An example to measure the cost of IPC.

necessary user-mode states into a kernel stack. When the kernel resumes the execution of the receiver, it restores the receiver's user-mode states and finally invokes a *SYSRET* instruction to return to the user mode. The mode switch in an IPC also contains two *SWAPGS* instructions that change the base address of the *gs* registers when entering and exiting the kernel. We measure the mode switch cost by executing a *null* SYSCALL which directly returns to the user mode for one billion times. To measure the overhead of each operation, we read The Time Stamp Counter (TSC) values before and after each instruction. The cycles for SYSCALL, SWAPGS and SYSRET are 82, 26 and 75 respectively.

Address Space Switch. A microkernel uses different virtual address spaces for processes to isolate them. Thus, it is necessary to switch the virtual address space when delivering an IPC. The measured cost of an address space switch on our machine is 186 cycles with the PCID (process ID) enabled. Moreover, recent microkernels use different page tables for the kernel and the user space to defend against the Meltdown attack [**?**]. Hence, an IPC usually involves two address space switches, which costs 372 cycles in total.

Other Software IPC logic. To handle an IPC request, a microkernel usually contains various security checks, endpoint management and capability enforcement. The total cost of this part is 98 cycles for seL4 fastpath on our machine.

In total, the fastest IPC implementation may cost 493 cycles if the Meltdown mitigations are disabled. This result matches the recent performance results for the fastpath IPC of seL4 [**?**], which was also measured on an Intel Skylake machine.

Table 1. The pollution of processor structures.

Name	i-cache	d-cache	L2 cache	L3 cache	i-TLB	d-TLB
Baseline	15	10624	13237	43	8	17
Delay	15	10639	13258	43	9	19
IPC	696	27054	15974	44	11	7832

2.1.2 Indirect Cost of Microkernels

The overheads of the synchronous IPC are not limited to the direct cost of the kernel. During the execution of the kernel, it will evict the user-mode states in some important processor structures, including the L1 instruction and data caches, L2 and L3 shared caches and translation look-aside buffers (TLB). The state pollution makes an indirect effect on the following user-mode instructions, which triggers TLB misses and different levels' cache misses.

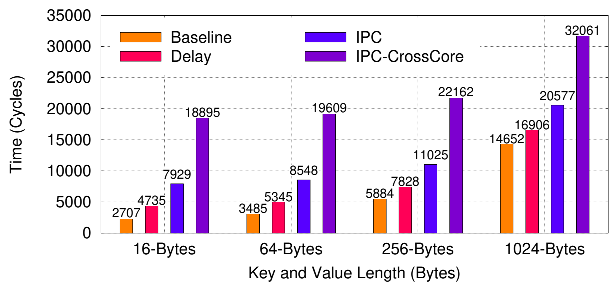


Figure 2. The average latency (in cycles) for the KV store operation. Lower is better.

To evaluate the overhead caused by the indirect pollution of processor structures, we build a simple Key-Value store in the seL4 microkernel. It consists of a client and two servers, which are an encryption server and a key-value (KV) store server as shown in Figure 1. For the insert operations, requests from the client reach the encryption server to encrypt the messages before getting to the KV store server to save the messages. For the query operations, the encryption server decrypts the query results from the KV store server and then returns them to the client.

There are three ways to organize the three processes:

- **Baseline**: putting them into the same virtual address space and utilizing function calls to connect them.
- **IPC**: putting them into different virtual address spaces and utilizing IPC to connect them. The seL4 kernel is configured without using Meltdown mitigations.
- **Delay**: putting them into the same virtual address space and utilizing the delay function calls to connect them. The delay function call uses a loop to delay for a period of time which is equal to the direct cost of an IPC (493 cycles).

We measure the impact of the key and value size on the benchmark throughput. The requests of the client consist of 50%/50% insert and query operations. Ideally, there should be no difference between the IPC and the Delay bar. However, as Figure 2 shows, the indirect cost of IPC is the reason for the gaps between the IPC and Delay bars.

We also count the different events occurring for 512 operations in the three experiments by leveraging Intel performance monitoring unit (PMU). Table 1 shows the footprints on several processor structures for them. The data indicates that the IPC version causes more significant impact on all levels of cache and TLB structure than that on the Delay and Baseline cases.

2.1.3 IPC Cost in Multicore

In a multicore machine, the servers and the client may reside on different cores, which is resulted from an oversubscribed condition or a scheduling decision. Under such a circumstance, a cross-core IPC involves a costly inter-processor

Table 2. Latency of different instructions and operations in cycles. KPTI [**?**] is a technique that uses two page tables for the kernel and the user space to defend against the Meltdown attack.

Instruction or Operation	Cycles
write to CR3	186±10
no-op system call w/ KPTI	431±13
no-op system call w/o KPTI	181±5
VMFUNC	134±3

interrupt (IPI). For example, the cross-core IPC degenerates into a slowpath version which contains an IPI. One IPI takes 1,913 cycles on our machine. We reevaluate the IPC version experiment by putting the client and its two servers to three different cores. The result is also shown in Figure 2. For various lengths of keys and values, the cross-core IPC incurs high overhead.

2.2 EPTP Switching with VMFUNC

VMFUNC [**?**] is an Intel hardware instruction that allows software in non-root mode (in both kernel and user modes) to invoke a VM function. VM functions are processor features managed by the hypervisor. EPTP (the pointer to an EPT) switching is one of these VM functions, which allows the guest to load a new value for the EPTP from an EPTP list stored in the Virtual Machine Control Structure (VMCS) configured by the hypervisor. The new EPT then translates subsequent guest physical addresses (GPA) to host physical addresses (HPA). The EPTP list can hold at most 512 EPTP entries. The typical usage of EPTP switching is to create multiple domains for one physical address space and these domains usually have different memory mappings and privileges [**? ?**]. Table 2 shows the latencies of different instructions and operations. With the Virtual Processor ID (VPID) feature enabled, the VMFUNC instruction does not flush TLB and costs only 134 cycles.

3 Overview

The traditional implementation of IPC requires the involvement of the kernel, which incurs the direct and indirect cost as we analyzed in Section 2. Therefore, to address the performance impact of the traditional synchronous IPC, SkyBridge aims at removing the kernel participation from synchronous IPC. It allows the client to directly switch to the server's virtual address space and execute the requested procedure, which not only avoids the direct cost of trapping into the microkernel but also partially eliminates the indirect cost of architectural state pollution.

3.1 Key Idea and Challenges

SkyBridge uses one VMFUNC to implement the switch of virtual address space without trapping into the kernel and eliminate the costly IPI because it allows one process to directly invoke other process's code. The general workflow of SkyBridge and the example code are shown in Figure 3 and Figure 4. SkyBridge provides a programming model similar to that of traditional IPC. To use SkyBridge, a program should be modified to use the SkyBridge user-level interfaces.

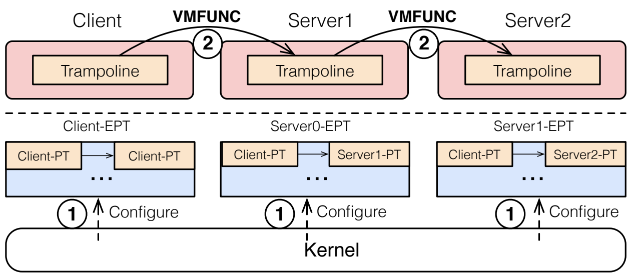


Figure 3. The general workflow of SkyBridge.

```
int server_handler(args)
{
    int res
    … // server handler logic
    return res;
}

void server (void)
{
     int connection_count = 8;
     int server_id = register_server(connection_count,
                          server_handler);
}

void client (void)
{
     register_client_to_server(server_id);
     direct_server_call(server_id, args)
}
```

Figure 4. Example code of a client and a server.

A server has to register into the kernel before other processes start to request its service. During registration, the server provides the kernel with a function address it allows other processes to directly invoke and a number indicating the maximum number of connections it can receive at the same time. Then the kernel maps trampoline-related code and data (shared pages and stacks) into the server's virtual address space and returns the server ID that can be used by clients to locate the server during registration. Similarly, the client also registers into SkyBridge by providing the server ID which it intends to call. The kernel then maps trampoline-related code and data pages into the client's virtual address space. It maps a server function list into the client virtual address space as well. Most importantly, the kernel creates one EPT for the client and all the target servers, as shown in Step ① in Figure 3. In these servers' EPTs, the page table of the client is mapped to the corresponding ones of the servers.

When the client invokes the direct_server_call (Step ② in Figure 3), the trampoline saves the client's states into its stack and invokes the VMFUNC instruction to switch to the

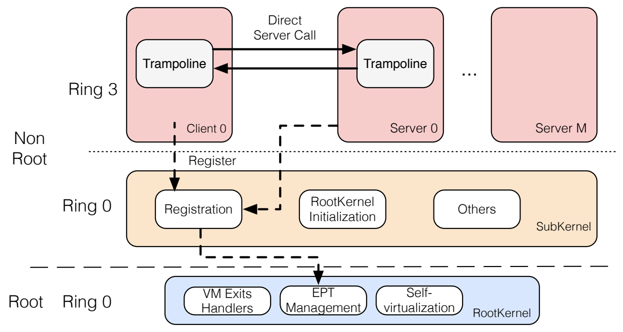


Figure 5. The overall architecture.

server's EPT. After switching EPT, the configuration in the server's EPT makes sure the hardware use the server's page table to translate all subsequent virtual addresses. Finally, the trampoline installs the server's stack and invokes the server's registered handling function.

However, applying VMFUNC imposes the following challenges.

- **Virtualization overhead**. The use of VMFUNC requires a hypervisor to be inserted at the bottom of the microkernel, which inevitably incurs overhead for the normal execution of the system.
- **Integration**. Existing techniques of leveraging VMFUNC are difficult to apply to microkernels without much modification.
- **Security**. New security threats are introduced by the design of the direct virtual address space switch.

3.2 Solutions

To leverage SkyBridge, SkyBridge proposes a series of solutions to addressing these challenges.

Efficient Virtualization: The overheads of virtualization mainly come from two sources. The first one is the two-level address translation and the other one is a large number of costly VM exits. To tackle the virtualization overheads, SkyBridge introduces a tiny hypervisor (Rootkernel) that contains only 1500 LoCs. It utilizes 1 GB huge pages to map most host physical memory except those reserved for Rootkernel to the microkernel (named Subkernel in our paper) in non-root mode. This memory mapping not only allows the execution of the microkernel not to trigger any EPT violation but also diminishes the cost of address translation from a GPA to its HPA. To tackle the second overhead, the Rootkernel configures VMCS to let the Subkernel handle most hardware events (external interrupts) or privileged instructions (e.g., HLT). By using such configuration, most VM exits are avoided. In our evaluation, there are no VM exits when running normal applications and the virtualization overheads are negligible.

Therefore, the architecture of SkyBridge is divided into two components, as shown in Figure 5. The Rootkernel consists of three parts, which are the management of EPT, handlers for inevitable VM exits and a self-virtualization module. The Subkernel has one line of code to call the self-virtualization module in Rootkernel to dynamically start the Rootkernel during the booting procedure. The process creation part is also modified to call the EPT management part of the Rootkernel to configure the EPT part for each new process. When creating a new process, the Subkernel maps a trampoline into the process, which helps the process to invoke the direct server call of SkyBridge. The details of the Rootkernel and Subkernel are described in Section 4.1 and Section 4.2.

Lightweight Virtual Address Switch: To use VMFUNC to efficiently switch the virtual address space in the user mode without the involvement of the kernel, there is one possible design, which is to combine all related processes into the same virtual address and use different EPTs to isolate them. Switching the virtual address space means to install a new EPT that enables the corresponding permission. Even if the solution sounds intuitive, it requires non-trivial modification to the microkernel to fix possible virtual address overlapping problems, which thus incurs tremendous engineering effort. SkyBridge proposes a lightweight and effective design that remaps the base address of the page table (CR3 value) in each EPT. Instead of putting all processes into the same virtual address space, SkyBridge still isolates them using different page tables. Before scheduling a new client, SkyBridge installs a new EPTP list for it, which contains the servers' EPT pointers the client is allowed to invoke. In each server's EPT, the GPA of the client's CR3 value is translated to the HPA of the corresponding server's CR3 value, which allows hardware to automatically use the new page table for later virtual address translation after the invocation of the VMFUNC instruction (Section 4.3).

Secure Trampoline: In the traditional microkernel design, each IPC gets trapped into the kernel, which then has the chance to do the security check and deny any illegal IPC communication. However, the kernel is unable to check each IPC communication in SkyBridge, which means SkyBridge has to provide new techniques to guarantee the same security level as the traditional IPC. First, a malicious process may use self-prepared VMFUNC instruction to bypass the trampoline and access sensitive data or code of other processes, which is called the VMFUNC faking attack in SeCage [**?**]. Yet, the defense proposed by SeCage fails to work in SkyBridge (we will explain it in Section 4.4). To defend against such attack, SkyBridge dynamically rewrites the binary of a process to replace any illegal VMFUNC with functionally-equivalent instructions. To prevent a sender from calling unregistered receivers, we provide a calling-key table for each process, which records a list of calling keys. For each IPC, the sender should provide a calling key for the receiver, which checks the key against its calling-key table and denies the IPC and notifies the kernel if the sent key does not exist in the table. This solution provides an optimistic security check which assumes most IPC is legal and does not require the kernel to check it (Section 4.4).

4 Detailed Design

4.1 The Rootkernel

To utilize VMFUNC, the processes have to run in non-root mode. However, there are two design choices of whether or not to put the kernel into non-root mode. One way like SeCage [?] and CrossOver [?] works like a virtual machine, where both the kernel and processes run in non-root mode. Usually, these systems reuse mature commercial hypervisors like KVM [?] and Xen [?], which are designed to support general virtual machines. Hence, this way will incur large overhead caused by the virtualization layer. The other design choice is to put the kernel in root mode while sustaining the processes in non-root mode, like Dune [?]. However, the costly VM exits still exist. In Dune, most system calls incur the cost of a VM exit that is significantly more expensive than a (nonvirtualized) system call.

SkyBridge offers a new solution different from the above two design choices. It eliminates the costly VM exits caused by previous solutions. SkyBridge provides a tiny hypervisor called the Rootkernel whose size is much smaller than the commercial hypervisors. The Rootkernel only contains necessary functionalities to support SkyBridge, which includes the EPT management, a dynamic self-virtualization module, and some basic VM exit handlers.

To eliminate the costly VM exits, the Rootkernel configures the hardware to let most VM behaviors not trigger any VM exits. VM exits include three categories: the privileged instruction exits, hardware event exits, and EPT violation exits. For the privileged instruction exits like changing CR3 value, the Rootkernel allows these instructions not to trigger any VM exits. To handle hardware events like an external interrupt in traditional hypervisors, a VM exit triggers to wake up the hypervisor when receiving this event. In SkyBridge, the Rootkernel allows the hardware to inject the external interrupts directly into the microkernel in non-root mode since it has the privilege to manage its external devices.

Commercial hypervisors use an EPT for each VM and specify in the EPT the memory regions belonging to the VM. This can limit this VM from accessing other VMs and the hypervisor's physical memory. When a VM accesses the physical memory which is not present in the EPT or it has not enough permissions to access it, an EPT violation VM exit triggers and the hypervisor wakes up to handle such violation. Furthermore, the 2-level address translation (from GVA to HPA) incurs higher overhead than the 1-level translation (from GVA to GPA). For example, one TLB miss in the 2-level address translation may require at most 24 memory accesses [?], which incurs large overhead. To eliminate the EPT violation VM exits and reduce the overhead of 2-level address translation, the Rootkernel creates a base EPT for the Subkernel and uses the maximum huge page (1 GB on an x86_64 machine) to map most physical memory address to the Subkernel. SkyBridge only reserves a small portion of physical memory (100 MB in our evaluation) for the Rootkernel. Hence, the microkernel is free to access almost all physical memory on the machine, and no more EPT violation will be triggered. Moreover, the huge page mapping not only reduces the number of memory accesses for handling a TLB miss, but also reduces the number of TLB misses.

The booting procedure of Rootkernel is different from traditional hypervisors. Inspired by CloudVisor [?], SkyBridge does not contain the machine bootstrap code that increases the complexity of the Rootkernel and is error-prone. Instead, the Rootkernel is booted by the Subkernel and downgrades the Subkernel to non-root mode.

The Rootkernel also retains handlers for inevitable exits. In our current implementation, the Rootkernel contains handlers for CPUID instructions, VMCALL instructions and EPT violations. The VMCALL instruction is leveraged by the Rootkernel to implement an interface to communicate with the Subkernel.

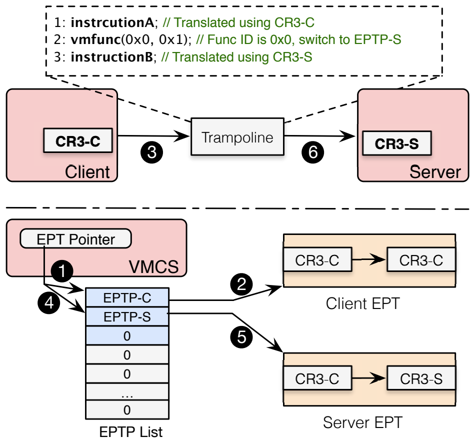
4.2 The Subkernel

The Rootkernel provides an interface for the Subkernel to manage each process's EPT. When a server registers into SkyBridge, the Subkernel maps the trampoline code page and a number of stack pages into the server's virtual address space. Then it allocates a free server ID for the server. When a client registers into SkyBridge and asks for getting bound to a server, the Rootkernel maps the trampoline code and stack pages into the client's virtual address space. Then the Subkernel invokes the Rootkernel interface to ask the Rootkernel to bind the client and server in the EPT level. The Rootkernel copies a new server EPT and maps the client page table to the server page table. Finally, the Rootkernel installs the newly created EPT into the client's EPTP list. Actually, the Rootkernel also writes all processes' EPTPs that the server depends on into the client's EPTP list.

SkyBridge does not modify the scheduling algorithm part of the microkernel. Yet, when the Subkernel decides to do a context switch from one process to a new process, it will notify the Rootkernel to install the next process's EPTP list.

SkyBridge provides shared buffers for processes to transfer large data. The Subkernel creates multiple shared buffers according to the number of registered threads for one server. Each time the client calls a server, it may use the server thread's buffer for large data transfer.

Process Misidentification: If a sender gets trapped into the kernel (e.g., caused by an interrupt) when executing in one receiver's virtual address space, it intends to invoke microkernel's services as a receiver. However, the microkernel will still treat the process as the original process, which we call a *process misidentification* problem. To address the problem,



```
1: instrcutionA; // Translated using CR3-C
2: vmfunc(0x0, 0x1); // Func ID is 0x0, switch to EPTP-S
3: instructionB; // Translated using CR3-S
```

Figure 6. Virtual address space switches in SkyBridge

SkyBridge allocates an identity page that records each process's identity information and maps this page into the same GPA in each EPT. This page is also mapped into the kernel address space which thus can be accessed by the Subkernel via a virtual address. The Subkernel checks this page to know which process it is serving for by accessing this virtual address.

4.3 Memory Mapping

SkyBridge guarantees that the virtual address spaces of different processes are isolated and provides an efficient user-level virtual space switch method for them. In fact, there are two known techniques for such purpose. The first technique [**?**] is to put different processes into the same virtual address space and use one EPT for each process to provide an isolated view of the shared virtual address space. Similar to SkyBridge, it also uses VMFUNC to switch among views without kernel involvement. This technique is easy to implement when the number of processes is small. When the number gets large, virtual address regions for different processes have to be carefully managed in order to prevent these regions from overlapping, which requires tedious engineering efforts.

The second technique is to leverage the recent Intel Memory Protection Keys for Userspace (PKU) to switch views. Applying PKU does not address the overlapping problem either. Moreover, it provides a limited number of security domains (16) and does not satisfy the requirement of microkernels.

SkyBridge sustains the traditional virtual memory isolation method and proposes an efficient virtual space switch technique, which does not require much engineering work to implement it. Different from the first technique, SkyBridge still uses separated page tables for these processes. To switch page tables without modifying the CR3 register in the user mode, it remaps the client's table page base address (CR3 value) to the HPA of server's CR3 value in the server's EPT. Therefore, the switch of EPT by invoking VMFUNC can install the server's virtual page table for later virtual address translations.

The technique SkyBridge employs to provide an efficient page table switch is depicted in Figure 6. A process is still created by the original mechanism of the microkernel and owns its virtual address space. In this example, the client and the server have their own page tables, whose base physical addresses (GPA) are client-CR3 and server-CR3 respectively. Once starting one new server, the Subkernel saves the CR3 value of the server (server-CR3). When the client registers, the Subkernel notifies the Rootkernel to copy two new EPTs from the base EPT for the client and server, which are EPT-C and EPT-S respectively. Then the Rootkernel remaps client-CR3 to the HPA of server-CR3 in EPT-S and does not make any modification to EPT-C.

During execution, the value of the CR3 register is *client-CR3* and will not be changed. When the client invokes direct_server_call interface, the trampoline invokes the VMFUNC instruction to change the value of EPT pointer from EPT-C to EPT-S. After using EPT-S, client-CR3 will be mapped to the HPA of server-CR3, which means all subsequent virtual address will be translated by the server page table. Therefore, the client can access any virtual address in the server's virtual address space.

Please note that the creation of EPT here is just a shallow copy that reuses most mapping in the base EPT. Only four pages that map client-CR3 to the HPA of server-CR3 are modified. All other EPT pages are kept intact.

4.4 Trampoline

The direct_server_call interface of SkyBridge is implemented by a trampoline, which is a code page mapped into the virtual space by the Subkernel during process registration. A client and all its bound servers should be inserted such trampoline. When binding a client to a server, the Subkernel creates multiple stacks and maps them into the server's virtual address space. The number of stacks is specified by the server during its registration and determines how many concurrent threads the server can support.

Usually, the sender needs to transfer some data to the receiver in an IPC. For small data transfer, SkyBridge puts these data into CPU registers which obeys the calling convention in x86_64. For large data transfer, SkyBridge creates a shared buffer for each pair of client and server thread and maps them into both the client and the server.

Trampoline Workflow: When the sender is a client, its ID is zero. Otherwise, the sender ID is the value returned by the register_server function. Before invoking the VMFUNC instruction, the trampoline copies data from the client's internal buffer into the shared buffer if the transferred data exceeds the capacity of CPU registers. After switching the virtual address space by invoking the VMFUNC instruction, the trampoline installs the server stack. Finally, it calls the server's registered function according to the server ID.

Security Threats: The design of the trampoline considers two possible attacks. The first attack is a self-prepared VMFUNC attack, where the malicious client or server invokes an illegal VMFUNC instruction which is not prepared by the trampoline to bypass the trampoline and access sensitive instructions or data in other processes. The previous defense against this attack is to put different pieces of application logic (PAL) code and data into different EPTs and guarantee that only the trampoline is mapped into these EPTs which makes it the only entry point to other PALs. However, this solution does not apply to SkyBridge due to the remapping of CR3 GPA technique, which allows the attacker to invoke VMFUNC at any virtual address to switch to the victim process's virtual address space. Therefore, the trampoline is not the only legal entry point to other processes.

To defend against such attack, we leverage the binary rewriting technique and scan the each process's code to replace any VMFUNC instruction or any sequences of instructions containing an inadvertent VMFUNC with other functionally-equivalent instructions. This solution has been used by different other systems [**? ? ?**] and we will describe our method in Section 5.

The second attack is called the illegal server call or client return. The illegal server call is that one client may bypass the server it should invoke and directly call unregistered servers, which is dangerous if these servers contain sensitive information. Similarly, the illegal client return is that one server does not return to the client that calls it, but to other client or servers. To defend against such attack, SkyBridge provides a calling-key table for each process, which records the processes bound to it and their calling keys. The server's calling keys are generated during the client registration. The Subkernel generates a random 8-byte key for the client and gives it to the client and the server. For example, the client gets the server's calling key and passes it to the server, which then checks the calling key against those in its table and notifies the Subkernel when it does not locate the given key in its table.

Each time a client will call its server, it also dynamically generates a client calling key and passes it to the receiver. The receiver should return this key to the sender, which rechecks it to ensure the receiver is what it calls before. A malicious process may deliberately leak its key to other processes. But the leaked key only exposes sensitive information belonging to the key owner and no other data will be exposed as servers can use calling keys to identify the calling processes.

5 Dynamically Rewriting Illegal VMFUNC

5.1 Rewriting Mechanism

When a process registers into SkyBridge, the Subkernel scans all code pages of the process to replace any illegal VMFUNC instructions outside the trampoline code page with functionally-equivalent instructions. After rewriting, one instruction may be changed to two or more instructions, which cannot be held in the original location. Therefore, SkyBridge replaces these instructions with a jump instruction which jumps to a page for rewriting. The rewriting page is inserted by the Subkernel and is mapped at an unused virtual address. We choose the second page in the virtual address space (starting from 0x1000). This page is deliberately left unmapped for most operating systems. In the inserted page, the Subkernel creates one code snippet for the new instructions. At the end of each snippet, the Subkernel also appends a new jump instruction to get back to the original code page.

5.2 Rewriting Strategy

The rewriting strategy is inspired by ERIM [**?**], which uses a similar strategy to replace the WRPKRU instruction. The strategy is ***complete***: any inadvertent VMFUNC instruction can be rewritten to functionally-equivalent instructions. The strategy is highly dependent on x86 variable-length instruction encoding. Intel x86 instruction encoding consists of five regions: 1) an opcode field possibly with a prefix (the opcode for VMFUNC contains three bytes: 0x0F, 0x01, 0xD4); 2) an optional 1-byte Mod R/M field describes the addressing mode and two operands for this instruction; 3) an optional 1-byte SIB (Second Addressing Byte) field specifying the indirect memory addressing; 4) an optional displacement which works as offset used in the addressing mode described by Mod R/M field; 5) an optional immediate field for the instruction.

There are three conditions where a VMFUNC instruction may be decoded from:

- **C1**: The instruction is indeed VMFUNC.
- **C2**: A VMFUNC encoded by spanning two or more instructions.
- **C3**: A long instruction contains the VMFUNC encoding.

To classify these three conditions, the Subkernel will bookkeep current instruction during scanning, which helps to determine instruction's boundary. For C1, the Subkernel just replaces the illegal VMFUNC with three NOP instructions (0x90). For C2, Illegal VMFUNC spanning two or more instructions can be "broken" by inserting a NOP between these consecutive instructions.

When VMFUNC exists in one longer instruction (C3), the Subkernel replaces this instruction with other functionally-equivalent instructions. Table 3 lists all possible cases and their corresponding rewriting strategies. The byte 0x0F is an escape prefix for opcode and will not occur in the middle bytes of any instruction opcode. Therefore, if the first byte

Table 3. Rewrite strategy for illegal VMFUNC instructions

ID	Overlap Case	Rewriting Strategy	Example
1	Opcode=VMFUNC	Replace VMFUNC with 3 NOP instructions	
2	Mod R/M=0x0F	Push/pop used register; use new register	imul \$0xD401, rdi, rcx; → push rax; mov rdi, rax; imul \$0xD401, rax, rcx; pop rax
3	SIB=0x0F	Push/pop used register; use new register	lea 0xD401(rdi, rcx, 1), rbx; → push rax; mov rdi, rax; lea 0xD401(rax, rcx,1), rbx; pop rax
4	Displacement=0x0F	Compute displacement value before the instruction	add 0xD4010F(rax), rbx; → add 0xD4000F, rax; add 0x0100(rax),rbx
5	Immediate=0x0F	Apply instruction twice with different immediates to get equivalent effect	add 0xD4010F, rax → add 0xD3010F, rax; add 0x10000, rax
		Jump-like instruction: modify immediate after moving this instruction	

of one instruction's opcode overlaps with 0x0F, this instruction is VMFUNC, whose rewriting strategy has already beed discussed (replace it with three NOP instructions).

If 0x0F equals the Mod R/M field (which is 1 byte), it determines that *rcx* (*r9* and *ecx*) and *rdi* (*r15* and *edi*) are the instruction's operands. The Subkernel replaces one of the registers (e.g., *rdi*) to another register, whose value will be pushed into the stack in advance. For example, the Subkernel replaces *rdi* with *rax* in Table 3. If 0x0F equals the SIB field (which is 1 byte as well), this instruction also uses fixed register and SkyBridge applies a similar rewriting strategy to replace it.

When the 0x0F overlaps with the displacement, the remaining two bytes (0x01 and 0xD4) may fit in the displacement or the immediate field. If the three bytes all reside in the displacement field, SkyBridge precomputes the displacement value before the instruction (Example is row 4). If some of the three bytes overlap with the immediate field, SkyBridge applies the instruction twice to get the same effect. For jump-like instructions, the immediate is treated as an offset which will be changed to a new value when we rewrite this instruction in the rewriting page.

6 Evaluation

This section tries to answer the following questions:

- **Q1**: What is the implementation complexity of SkyBridge?
- **Q2**: How does SkyBridge improve the IPC performance compared with other microkernels?
- **Q3**: How does SkyBridge improve the performance of the workloads introduced in Section 2?
- **Q4**: How do real-world microkernel applications perform when running using SkyBridge?
- **Q5**: How does the virtualization layer affect the performance of the original microkernel workloads?
- **Q6**: How many inadvertent VMFUNC instructions do we find?

6.1 Experimental Setup

Our test machine is equipped with an Intel Skylake Core i7-6700K processor, which has 4 cores and 8 hardware threads with the hyper-threading enabled. The memory size is 16 GB.

The microkernels we evaluated are seL4 (v10.0.0), Fiasco.OC and Zircon. Fiasco.OC is a 3rd-generation capability-based microkernel and provides synchronous IPC facility. Zircon is a microkernel developed by Google and also implements an IPC facility.

To ensure the evaluation results measured at the same CPU clock, we disabled the CPU frequency scaling. All experiments without using SkyBridge are conducted in the native hardware without using the virtualization layer.

6.2 Status Quo and Implementation Complexity

To answer the first question (**Q1**), we have implemented SkyBridge on different microkernels. The code sizes of the Rootkernel is 1.5 KLoC and modified lines of code to integrate SkyBridge into each microkernel are about 200 LoC. It is easy to integrate SkyBridge into an existing microkernel. We first implemented SkyBridge on seL4 and the porting cost of SkyBridge Fiasco.OC and Zircon took 2 and 4 person days accordingly.

6.3 IPC Performance

To answer the second question (**Q2**), we first evaluate and analyze the performance of various synchronous IPC implementations in different microkernels and compare them with the corresponding SkyBridge versions.

Figure 7 is the performance breakdowns of different IPC implementations. We measure the time period for an IPC roundtrip which starts from sending an empty message from the client until the client receives the response from the server. The results are the average value of 100,000 experiment runs. For IPC within the same core, seL4 has a fastpath implementation that we have analyzed in Section 2 and it performs the best among all the microkernel. For cross-core IPC, the fastpath IPC degenerates into a slowpath version. The slowpath version not only contains more IPC logics but also involves a costly IPI. Therefore, the IPC roundtrip costs 6764 cycles.

The Fiasco.OC microkernel also has a fastpath IPC implementation. However, the fastpath in Fiasco.OC may handle deferred requests (drq) during IPC which is the reason why its IPC is relatively slower than that in seL4. The cross-core IPC in Fiasco.OC also involves the costly IPI, which costs 8440 cycles.

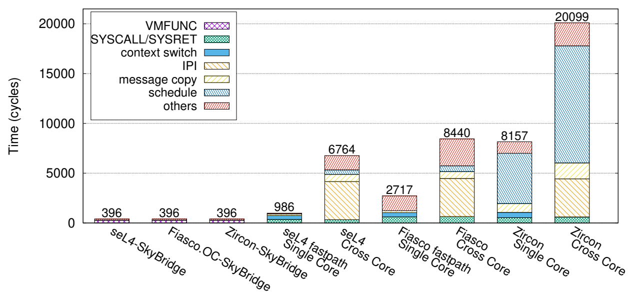


Figure 7. The performance breakdown of synchronous ipc implementations in different microkernels. lower is better.

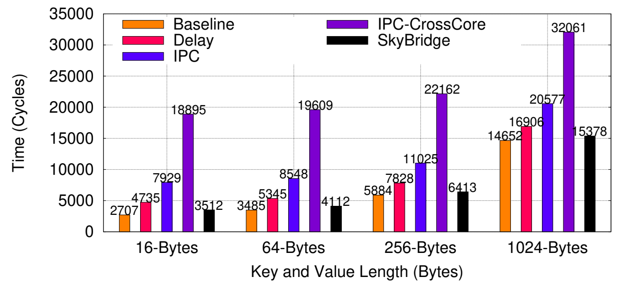


Figure 8. The performance of SkyBridge on the KV store benchmark. Lower is better.

The Zircon microkernel does not have a fastpath IPC, which means it may enter the scheduler when handling IPC. Moreover, the IPC path in Zircon may be preempted by interrupts. The message copying in Zircon is not well optimized, which involves two expensive memory copies for each IPC. These are the reasons why Zircon performs worst among the three microkernels. Its cross-core IPC also involves scheduling part and its average cost is 20099 cycles.

We implement SkyBridge on these three microkernels. The overheads of SkyBridge mainly comes from two sources. The first one is the VMFUNC instruction, which costs 134 cycles. The second one includes all other operations, such as saving and restoring register values and installing the target stack. The second source costs 64 cycles. Therefore, an IPC roundtrip in SkyBridge costs 396 cycles, which improves the performance of single-core IPC by 1.49x, 5.86x and 19.6x for seL4, Fiasco.OC and Zircon respectively. For cross-core IPC, the improvement is 16.08x, 20.31x and 49.76x.

6.4 Performance of the Key-Value Store in SkyBridge

To answer **Q3**, we modify the KV store benchmark and connect the processes using SkyBridge. The result is shown in Figure 8. Due to the space limit, we only report the results in seL4, which has the fastest IPC and we believe can represent other microkernels.

When the length of key and value is small, the IPC occupies a large portion of the whole benchmark. SkyBridge can reduce the latency from 7929 cycles to 3512 cycles. When the length of key and value is large, the overhead of SkyBridge is negligible.

6.5 SQLite3 Performance

Table 4. The throughputs (in ops/s) of four basic SQLite3 operations under different microkernels and server settings. ST-Server means that the file system and the block device has a single working thread, while MT-Server represents that these servers have multiple working threads and they are pinned to each physical cores. The rightmost column is the speedup of SkyBridge compared with the performance of MT-Server.

		ST-Server	MT-Server	SkyBridge	Speedup
seL4	Insert	4839.08	6001.82	11251.08	87.5%
	Update	3943.71	4714.52	7335.57	55.6%
	Query	13245.92	14025.37	18610.60	32.7%
	Delete	4326.92	5314.04	7339.31	38.1%
Fiasco	Insert	1296.83	1685.39	5000.00	196.7%
	Update	1222.83	1557.09	4545.45	191.9%
	Query	8108.11	8256.88	15789.47	91.2%
	Delete	1255.23	1607.14	4568.53	184.2%
Zircon	Insert	1408.42	2467.90	7710.63	212.4%
	Update	1376.77	2360.00	6643.24	180.4%
	Query	9432.34	9535.56	17843.54	87.1%
	Delete	1389.64	1389.64	7027.30	405.7%

To answer **Q4**, we evaluate the performance of a database application.

SQLite3 [**?**] is a widely-used and lightweight relational database. To use it, we put the client and the SQLite3 database into the same virtual address space. We also port a log-based file system named xv6fs [**?**], which is a formally verified crash-safe file system. The LibC of the three microkernels is modified to use the new file system. We use a RAM disk device to work as the block device and the file system communicates with the device with IPC. The client first uses the SQLite3 database to manipulate files and communicate with the first server (the file system). The file system finally reads and writes data into the block device server (The second server).

In an SMP scenario, there are two possible configurations of the server. One is to create only one server thread and each client may use cross-core IPC to communicate with the server. The second one is to create multiple server threads and pin them to each physical core. The client can directly communicate with the server thread sharing the same core with the client, which avoids the costly cross-core IPC.

We evaluate the performance of four basic SQLite3 operations, which are insert, update, query and delete, in three microkernels, as Table 4 shows. Among the four operations, the query operation performs the best since the SQLite3 has an internal cache to handle the recent read requests, which thus avoids a large number of IPC operations. For ST-Server, we create one working thread for each server and pin the

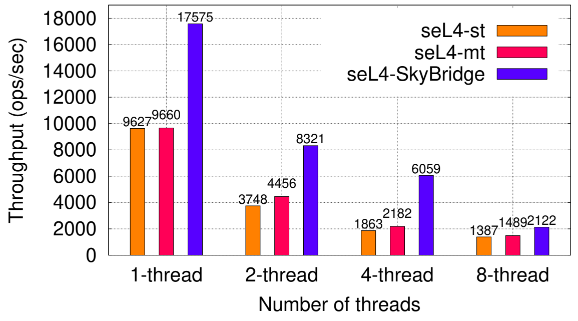


Figure 9. The throughput of YCSB-A (in op/s) for seL4. Higher is better.

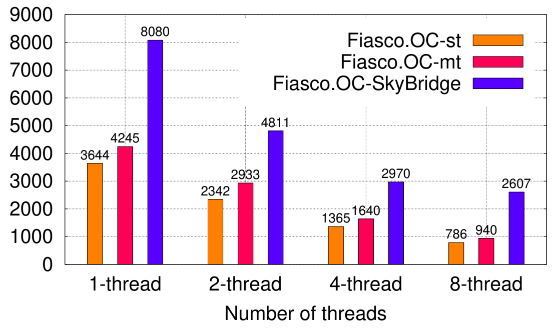


Figure 10. The throughput of YCSB-A (in op/s) for Fiasco.OC. Higher is better.

client and the two servers to three different physical cores and the IPC among them involves IPIs, which accounts for its bad performance. For MT-Server, we create multiple working threads for each server and pin these thread to different physical cores. The client can communicate with the local server thread without issuing the costly cross-core IPC. We also evaluate these four operations using SkyBridge, which allows the client to directly jump into the server's virtual space and call its functions. SkyBridge can greatly improve the performance of insert, update and delete operations except the query operation. The reason is that the query operation does not cause many IPC operations compared with other three operations.

Figure 9 shows the throughputs of SkyBridge for seL4 in an SMP scenario. Since the xv6fs does not support multi-threading, we use one big lock in the file system, that is the reason why the scalability is so bad for this benchmark. We use the YCSB workloads to test the throughput. All workloads have similar results and we only report YCSB-A result here due to the space limit. YCSB-A workload consists of 50% read (query) and 50% write (update) operations. We run the workload on a table with 10,000 records. seL4-st means that the servers only have one working thread, but we do not bind the thread to a specific core. In seL4-mt, we create

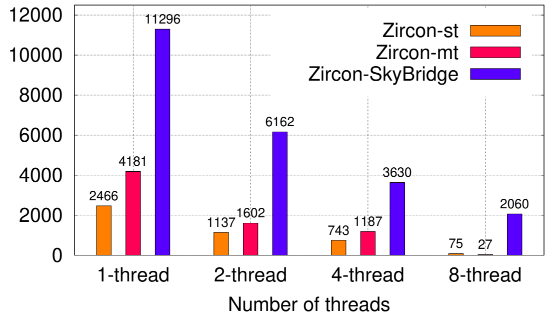


Figure 11. The throughput of YCSB-A (in op/s) for Zircon. Higher is better.

Table 5. The throughput (in ops/s) of SQLite3 using YCSB-A in the native and virtualized environments without using SkyBridge and the number of VM exits in SkyBridge Rootkernel.

	Native	Rootkernel	#VM exits
YCSB-A 1 thread	9745.15	9694.49	0
YCSB-A 8 thread	1465.95	1411.64	0

multiple threads for the servers and pin them to each core. With the increase of the cores, we also create more threads for the clients. The result shows that SkyBridge can outperform the old IPC mechanism in a different number of cores and the average speedups are 0.819x, 1.442x and 9.593x for seL4, Fiasco.OC (Figure 10) and Zircon (Figure 11) respectively.

We measure the number of IPI for each experiment. For example, in the 8-thread experiment in seL4, the number of IPI for seL4-st is 1,984,343 while the value in seL4-mt is 20.

6.6 Virtualization Overhead

To answer *Q5*, we count the number of VM exits during the execution of the SQLite3 benchmark with and without the Rootkernel. Table 5 shows the performance of the SQLite3 using the YCSB-A workload with different numbers of threads in seL4. We first evaluate the performance in the native environment and then run the same workload above the Rootkernel without using SkyBridge. The result demonstrates that the Rootkernel design incurs negligible overhead for the workload. Even if an application does not use SkyBridge, its performance is not affected by the virtualized environment introduced by SkyBridge. The number of VM exits collected during the experiments are **0**, in that the Subkernel can get access to most physical memory, handle hardware events and execute privileged instructions without triggering any VM exits, as we introduced in Section 4.1.

6.7 Rewriting VMFUNC

We do not find any occurrence of inadvertent VMFUNC instructions in our microkernel benchmarks. We scan many different programs in Linux and find only one occurrence of

Table 6. Inadvertent VMFUNC instructions found by SkyBridge

Program	Average Code Size (KB)	VMFUNC Count
SPECCPU 2006 (31Apps)	424	0
PARSEC 3.0 (45 Apps)	842	0
Nginx v1.6.2	979	0
Apache v2.4.10	666	0
Memcached v1.4.21	121	0
Redis v2.8.17	729	0
Vmlinux v4.14.29	10,498	0
Linux Kernel Modules v4.14.29 (2,934 Modules)	15	0
Other Apps (2,605 Apps)	216	1 (in GIMP-2.8)

inadvertent VMFUNC instruction in GIMP-2.8, as shown in Table 6. GIMP is an image manipulation program and the inadvertent VMFUNC is contained in the immediate region of a longer call instruction. The immediate is an offset and the call instruction can be replaced with our rewriting strategy for jump-like instructions.

7 Security Analysis

Malicious EPT switching: As we mentioned in Section 4.4, a malicious process may use a self-prepared VMFUNC to bypass the trampoline and jump to any address of one victim process. SkyBridge defends against such attack by dynamically scanning the binary of each process during loading time and replacing any VMFUNC instruction with functionally-equivalent instructions.

Meltdown Attacks [?]: Meltdown attacks allow unauthorized processes to read data of privileged kernel or other processes. Current OS kernels including the microkernels like seL4 defend against this attack by using two page tables for the kernel and user programs respectively. SkyBridge can also defeat such attack since it still puts different processes into different page tables..

DoS Attacks: During a direct server call, the called server may encounter internal errors which lead to itself failure or the server is deliberately waiting and does not return to the client, both of which causes a hang for the client. Like other microkernels, SkyBridge provides a timeout mechanism, which can force the server to return the control flow to the client.

Malicious Server Call: Due to the hardware features, the Rootkernel has to put all server's EPTP into the same EPTP list before one client gets scheduled, which may allow a malicious client or server to call other servers even if this call is forbidden. To prevent this illegal call, SkyBridge provides a calling-key table for each process to check whether the current caller is a legally registered process.

Refusing to Call SkyBridge Interface: One process may refuse to call the interface provided by SkyBridge. But it is in an isolated environment, which means this behavior only results in its own execution failure, not affecting other processes or the kernel.

8 Related Work

8.1 Software-based IPC Optimization

There is a long line of research on reducing the overheads of IPC over the last decades [? ? ? ? ? ?]. L4 [? ? ?] integrates many techniques, including using in-register message transfer to avoid redundant copying and leveraging tagged TLB to reduce the context switch overhead. One notable optimization technique is to migrate the client's thread to server's address space and run server's code, which is used in LRPC [?] and thread migration model [? ? ? ? ?]. This technique has two main benefits. The first one is that it avoids the costly scheduling since it does not block the client and allows it to directly switch to the server using its own scheduling quantum. The second one is that there is only a partial context switch where a small subset of registers and the address space are changed. However, it still requires the involvement of the kernel and the costly of address switches. In contrast, SkyBridge follows the thread migration model but designed a kernel-less IPC facility with extremely low overhead.

Modern microkernels such as seL4 have adopted many aforementioned optimization techniques. In seL4, there are two kinds of IPC: fastpath and slowpath. The fastpath IPC leverages the direct process switch technique to make the kernel directly switch to the target server without the involvement of the scheduler, which thus helps the fastpath to achieve the extremely small latency. However, as we analyzed in Section 2, the fastpath still involves the kernel and thus performs worse than SkyBridge.

The requirement for supporting more complex functionalities stimulates the design of asynchronous IPC like asynchronous notifications [?]. Current microkernels usually contain a mixture of both synchronous and asynchronous IPCs. FlexSC [?] also proposes asynchronous system calls for batching processing and avoiding the costly context switches between user space and kernel space in a monolithic kernel. The focus of this paper is mainly on improving synchronous IPC, which has already resulted in tremendous performance improvements.

Scheduling-related techniques are also proposed to improve the IPC performance. Lazy scheduling [?] avoids the frequent queue manipulation, but does not guarantee the bounded execution time of the scheduler, which is required by some hard real-time systems. Hence, seL4 [?] proposes Benno scheduling to address such problem.

For long message, one solution is to provide the shared buffer for the client and server, which requires two memory copies. To this end, L4 [?] proposes a technique called temporary mapping, which temporarily maps the caller's buffer into the callee's address space and avoids one costly message copying. This technique is orthogonal to SkyBridge and may also be combined with SkyBridge to achieve better performance.

8.2 Single Address Space Systems

Another direction is to put all domains into a single virtual address space and leverage other techniques to enforce the isolation among these domains, which can be divided into the software solutions [**? ? ? ?**] and hardware solutions [**? ? ? ? ? ? ? ? ?**].

SPIN [**?**] and Singularity [**? ?**] uses type-safe programming language to enable the low-overhead software-based memory isolation. Besides, different mainstream processor manufactures also present their products that support memory isolation within the same virtual space. ARM memory domain mechanism [**? ?**] assigns memory pages into different domains and the hardware Memory Management Unit (MMU) decides if an access should be allowed based on the current level stored in a per-core register (Domain Access Control Register). Intel introduces the Protection Keys for Userspace (PKU) mechanism to provide similar functionality. ERIM [**?**] leverages Intel PKU to provide isolated domains within a single virtual address space. However, these hardware features only supported limited domain numbers. Different from these works, SkyBridge provides an efficient IPC facility for a larger number of virtual address spaces (i.e., 512). SeCage [**?**] divides an existing application into different Pieces of Application Logic (PAL) by program analysis and puts them into isolated EPT space. SkyBridge focuses on connecting different virtual address spaces which is different from SeCage and proposes a series of techniques to address the challenges SeCage does not encounter. CrossOver [**?**] leverages VMFUNC to provide efficient cross-domain control transfers for virtual machines while SkyBridge focuses on IPCs in the microkernel world. Besides, CrossOver is mainly a hardware design that mandates hardware changes for high efficiency.

New hardware extensions are also proposed to provide efficient in-process isolation, like CODOMs [**? ?**], CHERI [**? ?**], Opal [**?**] and IMIX [**?**]. However, these solutions usually require non-trivial modification to the hardware and the microkernels.

9 Discussion and Limitation

Legacy Hypervisors. To run SkyBridge in cloud environments [**? ? ?**], legacy hypervisors need to be modified in order to support the Rootkernel. Fortunately, most functionalities of the Rootkernel have already been implemented in today's hypervisors. For example, Xen allows a VM to create up to 512 EPTs and use VMFUNC to switch the EPT pointer by using alt2pm [**?**]. Other required modifications in the hypervisors are to allow the Subkernel to change mappings in the EPTs, which can be implemented via hypercalls which accept and check the mapping information provided by the Subkernel.

W⊕X Code Pages. To defend against a malicious VMFUNC instruction, SkyBridge scans code pages to replace any inadvertent VMFUNCs with functionally-equivalent instructions. One implication of this technique is that it disallows direct modifications to code pages, which may prevent JIT compilation [**?**], dynamic software updating [**?**] and live updating of operating systems [**?**]. Therefore, to support dynamic code generation, the code generation process must be adapted to make code pages writable and non-executable. After code generation, these pages must be remapped as executable and non-writable, which allows SkyBridge to rescan them. The rescanning should be carefully implemented to avoid the instructions that span the newly mapped page and neighboring pages. The remapping and rescanning may impact application performance. However, these operations can be boosted by leveraging a batching technique and thus incur negligible overhead [**?**]. We will investigate the performance implication of W⊕X code pages in the future.

10 Conclusion and Future Work

This paper described the motivation, design, implementation and evaluation of SkyBridge, a microkernel IPC facility that eliminates the involvement of the kernel and allows a process to directly to switch to the virtual address space of the target process and execute its code. SkyBridge leverages VMFUNC to implement efficient IPCs and proposes a series of techniques to guarantee security properties and efficiency. We have integrated SkyBridge into three microkernels (seL4, Fiasco.OC and, Google Zircon) to evaluate its performance. The results show that SkyBridge can improve the performance of IPC by 1.49x to 19.6x for microbenchmarks. For real-world applications (SQLite3 database), SkyBridge improves the throughput by 81.9%, 1.442x and 9.593x for these three microkernels on average.

We plan to extend our work in three directions. First, we plan to investigate and extend the design of SkyBridge to monolithic kernels like Linux to boost applications that communicate through Linux IPC facilities. Second, we explore how to generalize the design of SkyBridge on more hardware architectures like ARM and AMD. Third, since the EPTP list can hold at most 512 EPTP entries, we plan to design a technique that dynamically evicts the least recently used EPTP entries from the EPTP list when the server number is larger than 512.

Acknowledgments

We sincerely thank our shepherd Mathias Payer, Martin Decky and the anonymous reviewers for their insightful comments. We also thank Dong Du for sharing his experience in porting SQLite 3.0 and xv6fs to microkernels. This work is supported in part by the National Key Research & Development Program (No. 2016YFB1000104), and the National Natural Science Foundation of China (No. 61772335). Haibo Chen is the corresponding author.

References