XPC: Architectural Support for Secure and Efficient Cross Process Call

Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, Haibo Chen
Shanghai Key Laboratory for Scalable Computing Systems
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
{Dd_nirvana,huazhichao123,xiayubin,byzang,haibochen}@sjtu.edu.cn

ABSTRACT

Microkernel has many intriguing features like security, fault-tolerance, modularity and customizability, which recently stimulate a resurgent interest in both academia and industry (including seL4, QNX and Google's Fuchsia OS). However, IPC (inter-process communication), which is known as the Achilles' Heel of microkernels, is still the major factor for the overall (poor) OS performance. Besides, IPC also plays a vital role in monolithic kernels like Android Linux, as mobile applications frequently communicate with plenty of user-level services through IPC. Previous software optimizations of IPC usually cannot bypass the kernel which is responsible for domain switching and message copying/remapping; hardware solutions like tagged memory or capability replace page tables for isolation, but usually require non-trivial modification to existing software stack to adapt the new hardware primitives. In this paper, we propose a hardware-assisted OS primitive, *XPC* (Cross Process Call), for fast and secure synchronous IPC. XPC enables direct switch between IPC caller and callee without trapping into the kernel, and supports message passing across multiple processes through the invocation chain without copying. The primitive is compatible with the traditional address space based isolation mechanism and can be easily integrated into existing microkernels and monolithic kernels. We have implemented a prototype of XPC based on a Rocket RISC-V core with FPGA boards and ported two microkernel implementations, seL4 and Zircon, and one monolithic kernel implementation, Android Binder, for evaluation. We also implement XPC on GEM5 simulator to validate the generality. The result shows that XPC can reduce IPC call latency from 664 to 21 cycles, up to 54.2x improvement on Android Binder, and improve the performance of real-world applications on microkernels by 1.6x on Sqlite3 and 10x on an HTTP server with minimal hardware resource cost.

CCS CONCEPTS

• Computer systems organization → Architectures; **• Software and its engineering → Operating systems**.

KEYWORDS

operating system, microkernel, inter-process communication, accelerators

ACM Reference Format:
Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, Haibo Chen. 2019. XPC: Architectural Support for Secure and Efficient Cross Process Call. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307650.3322218

1 INTRODUCTION

Microkernel has been extensively studied for decades [23, 28, 32, 39, 42, 45, 58]. It minimizes functionalities in privileged mode and puts most of them, including paging, file system and device drivers, in isolated user-mode domains to achieve fine-grained isolation, better extensibility, security and fault tolerance. Due to these benefits, microkernel-based OSes have been widely deployed in a broad range of devices including aircraft [22], vehicles [36] and mobile baseband systems [35]. Recently, we also witnessed a resurgent interest in designing and deploying microkernel-based operating systems, including seL4 [39], L4 on Apple's iOS secure enclave [4] and Google's next-generation OS called Fuchsia[1].

However, the implementations of current microkernel-based OSes still face a tradeoff between security and performance: more fine-grained isolation usually leads to better security and fault tolerance but also more IPCs (inter-process communications), which are known as the Achilles' Heel of microkernels [27, 46, 61]. For example, on a modern processor like Intel SkyLake, seL4 spends about 468 cycles [7] for a one-way IPC on its fast path[2] (687 cycles when enabling Spectre/Meltdown mitigations). Even worse, Google's Fuchsia's kernel (called Zircon) costs tens of thousands of cycles for one round-trip IPC. This brings a notable performance slowdown over a monolithic kernel like Linux for many IPC-intensive workloads.

Monolithic kernel-based OSes also suffer from the long latency of IPC. For example, Android is built on the monolithic kernel, Linux, and provides many user-level services for mobile applications. These applications frequently communicate with user-level services, like drawing a component in the surfaces through window manager, which causes high overhead. Android has introduced Binder [12] and anonymous shared memory [11] in Linux kernel to mitigate the issue, but the latency is still high.

Most of the cycles of an IPC are spent on two tasks: 1) domain switching, and 2) message copying. Since the caller and callee are in user mode, they have to trap into the kernel to switch the address

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322218

[1] https://fuchsia.googlesource.com/

[2] Fast path in seL4 is a heavily-optimized IPC routine without scheduling and does not consider long message copying.

space, which includes context saving/restoring, capability checking, and many other IPC logics. Sending messages through shared memory can reduce the number of copying, but may also lead to TOCTTOU (Time-Of-Check-To-Time-Of-Use) attack if both the caller and callee own the shared memory at the same time [49]. Adopting page remapping for ownership transfer can mitigate the above security problem, but the remapping operation still requires kernel's involvement. Meanwhile, remapping may also lead to costly TLB shootdown.

Previous work proposed various ways to optimize IPC performance, by either software [17, 29, 30, 45] or hardware [43, 44, 50, 55, 63, 66, 67]. For most software solutions, the overhead of trapping to kernel is inevitable, and message passing will lead to either multiple copying or TLB shootdown. Some hardware solutions, like CODOMs [62], leverage tagged memory instead of page tables for isolation. They adopt single address space to reduce the overhead of domain switching and message passing. These new hardware solutions usually require non-trivial modification of existing kernel implementations which are designed for multiple address spaces.

We advocate an efficient and secure IPC for microkernels and monolithic kernels, with the regeneration of microkernels on trending heterogenous systems [13], mobile OS [2] and the next generation data center [57], and the widespread IPC usage in monolithic kernel-based OSes like Android. In this paper, we propose a new hardware-assisted OS primitive, XPC (cross process call), to securely improve the performance of IPC. The design has four goals:

(1) Direct switching without trapping to kernel.
(2) Secure zero-copying for message passing.
(3) Easy integration with existing kernels.
(4) Minimal hardware modifications.

Specifically, our new primitive contains three parts. The first is a new hardware-aware abstraction, *x-entry*, which is similar to *endpoint* in traditional microkernel but with additional states. Each *x-entry* has its own ID and uses a new capability, *xcall-cap*, for access control. The capability is managed by the kernel for flexibility and checked by the hardware for efficiency. The second is a set of new instructions including *xcall* and *xret* that allows user-level code to directly switch across processes without the kernel involved. The third is a new address-space mapping mechanism, named *relay-seg* (short for "relay memory segment"), for zero-copying message passing between callers and callees. The mapping is done by a new register which specifies the base and range of virtual and physical address of the message. This mechanism supports ownership transfer of the message by ensuring only one owner of the message at any time, which can prevent TOCTTOU attack and requires no TLB flush after a domain switch. A *relay-seg* can also be passed through the invoking chain, aka *handover*, to further reduce the number of copying.

Although asynchronous IPC has the benefit of high throughput, synchronous IPC can achieve low latency and is easy to support semantics in existing POSIX APIs, thus has been used widely in existing systems [58, 65]. Even if Google's Zircon adopts asynchronous IPC, it uses the asynchronous IPC to simulate the synchronous semantics of the file system interfaces. This, unfortunately, introduces latencies as high as tens of thousands cycles per IPC.

XPC chooses to keep semantic of synchronous IPC across different address spaces, which makes it easy to be adopted to existing OS kernels. Meanwhile, XPC overcomes two limitations of traditional synchronous IPC [26], one is the relatively low data transfer throughput and the other is its not-easy-to-use model for multi-threaded applications. Specifically, XPC improves throughput with the *relay-seg* mechanism and provides easy-to-use multi-threaded programming interfaces with the *migrating thread* model [29].

We have implemented a prototype of XPC based on Rocket RISC-V core on FPGA board for evaluation. We ported two microkernel implementations (seL4 and Zircon) and one monolithic kernel implementation (Android Binder), then measured the performance of both micro-benchmarks and real-world applications. The result shows that XPC can reduce the latency of IPC by 5x-141x for existing microkernels, up to 54.2x improvement on Android Binder, and the performance of applications like SQLite and a Web Server can be improved by up to 12x (from 1.6x). The overall hardware costs are small (1.99% in LUT resource).

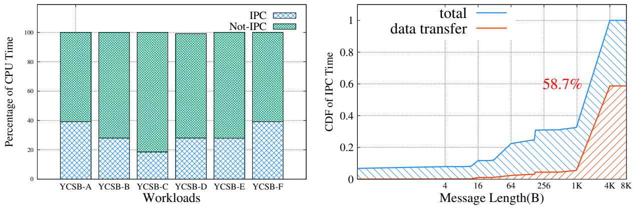
The main contributions of this paper are as follows:

- A detailed analysis of performance overhead of IPC and a comparison with existing optimizations.
- A new IPC primitive with no kernel-trapping and zero-copying message support along the calling chain.
- An implementation of XPC on FPGA with low hardware costs as well as Gem5 and the integration with two real-world microkernels and Android Binder.
- An evaluation with micro-benchmarks and applications on a real platform.

2 MOTIVATION

We start by analyzing the IPC performance of a state-of-the-art microkernel (i.e., seL4 [39]) and then present a detailed explanation of IPC. Our work is motivated by the performance analysis.

2.1 IPC Performance is Still Critical



(a) CPU time spent. (b) IPC time on YCSB-E.

Figure 1: (a): For Sqlite3 with YCSB workload, around 18% to 39% of the time is spent on IPC. (b): Distribution of IPC time, "data transfer" means the percentage of message transfer.

We took YCSB benchmark workloads and ran Sqlite3 on seL4 on a SiFive U500 RISC-V environment [8] (more setup details are in §5). Figure 1(a) shows that Sqlite3 with YCSB's workloads spends 18% to 39% of the time on IPC, which is significant. For each IPC, most of the time is spent on two tasks: domain switch and message transfer. For IPC with short message, the major performance overhead comes from domain switch; as the length of message increases, the

Table 1: One-way IPC latency of seL4. seL4 (4K) will use shared memory. The evaluation is done on a RISC-V U500 FPGA board.

Phases (cycles)	seL4(0B) fast path	seL4(4KB) fast path
Trap	107	110
IPC Logic	212	216
Process Switch	146	211
Restore	199	257
Message Transfer	0	4010
Sum	**664**	**4804**

time of data transfer dominates. Figure 1(b) shows the cumulative distribution of IPC time with different message sizes on the YCSB-E workload. In total, message transfer takes 58.7% of all the IPC time. The result is similar for other YCSB workloads, ranging from 45.6% to 66.4%. The rest is mainly spent on domain switch, which takes another half of the entire IPC time. This motivates us to design XPC with **both fast domain switch and efficient message transfer**.

2.2 Deconstructing IPC

In this section, we break down the process of IPC, measure the cost of each step, and analyze where the time goes. This quantitative analysis is done using a state-of-the-art microkernel, seL4, with a latest RISC-V FPGA board.

There are two paths of IPC in seL4: the "fast path" and the "slow path". The fast path contains five steps, as shown in Table 1. The slow path allows scheduling and interrupts, which introduce longer latency. Next, we will focus on the fast path and explain when seL4 will take the slow path.

Trap & Restore: A caller starts an IPC by invoking a system call instruction to trap into the kernel. The kernel will first save the caller's context and switch to the kernel's own context. After finishing the IPC handling code (e.g., *fastpath_call* in seL4), the kernel will restore the callee's context and return to its userspace. As shown in Table 1, these two phases take about 300 cycles which becomes a significant overhead of domain switch.

In existing systems, the costly switch between kernel mode and user mode is inevitable. Besides, the kernel will always save and restore all the context for isolation. The underlying assumption is that the caller and callee do not trust each other. However, we find that in certain cases, the caller and callee may have different trust assumptions, e.g., by defining their own calling conventions. Thus, it could be more flexible and efficient to let the caller and callee manage their context to achieve a balance between performance and isolation.

IPC Logic: In the IPC logic part, the major task is checking. seL4 uses capabilities to manage all the kernel resources, including IPC. The kernel first fetches the caller's capability and checks its validity (e.g., having *send* permission or not). It then checks if the following conditions are met to decide whether to take the slow path or not:

- the caller and callee have different priorities, or
- the caller and callee are not on the same core, or
- the size of a message is larger than registers (32-byte) and less than 120-byte (IPC buffer size).

The IPC logic takes about 200 cycles.

We find that these checking logic are more suitable to be implemented in hardware, which can be done in parallel to hide the latency. It inspires us to separate the logic to a control plane and a data plane, in which the former is done by software for more flexibility and the latter by hardware for more efficiency.

Process Switch: After running the IPC logic, the kernel achieves the "point of no return" and switches context to the callee. In this part, the kernel manipulates the scheduling queue to dequeue the callee thread and block the caller. To make the callee have the capability to reply, a *reply_cap* is added into the callee thread. Finally, the kernel transfers the IPC messages (only for messages ≤ 32B) and switches to the callee's address space. The process switch phase occupies about 150-200 cycles.

Process switch introduces several memory accesses (e.g., user contexts, capabilities, and scheduling queue). These memory accesses may trigger Cache and TLB misses, and thus affect the IPC performance.

Message Transfer: In seL4, there are three ways to transfer a message according to its length. If a message is small enough to be put into registers, it will be copied during the process switch phase, as mentioned. For medium-size messages (≤ IPC buffer and > register size), seL4 will turn to slowpath to copy the message (in our experiment, an IPC with 64B message takes 2182 cycles). For long message transfer, seL4 uses shared memory in user space to reduce data copying (e.g., 4010 cycles for copying 4KB data).

However, it is hard to achieve efficient and secure message transfer with shared memory. Although in-place update in shared memory can achieve zero-copying between caller and callee, it is not secure. For example, a multi-threaded caller can observe the operations performed by the callee and even affect the callee's behavior by modifying the shared memory. In most existing implementations, the data still needs to be copied to the shared memory at first. The message transfer dominates the IPC cycles when the message size is large.

Observations: In the analysis, we have two observations: first, a fast IPC that not dependent on the kernel is necessary but still missing. Second, a secure and zero-copying mechanism for passing messages while supporting handover is critical to performance. Our design is based on these two observations.

3 DESIGN

XPC consists of a new hardware component (XPC engine) and software support (OS kernel and a user library). The XPC engine provides basic functionalities for IPC, including capability checking, context switching, and a lightweight yet efficient message passing mechanism. The OS kernel acts as the control plane and manages IPC by configuring the XPC engine.

3.1 Design Overview

This section describes two hardware primitives provided by XPC engine and the programming model. The hardware changes are summarized in Figure 2.

User-level Cross Process Call: The XPC engine provides two new abstractions: *x-entry* and *xcall-cap* for this primitive. An *x-entry* is bound with a procedure that can be invoked by other processes. A

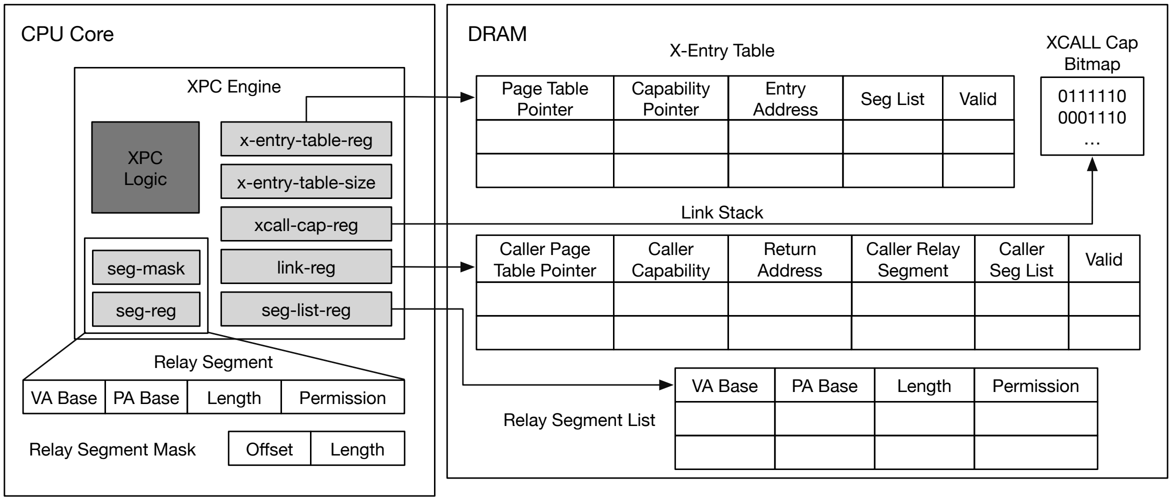


Figure 2: XPC Engine: *x-entry-table* holds all *x-entries*, each of which represents an XPC procedure. *xcall* capability bitmap indicates which *x-entries* can be invoked by current thread. A *link stack* is used to store the linkage record. Two new instructions *xcall* and *xret* perform the call and return operations, handled by XPC Logic. *seg-reg*, *seg-mask* and *seg-list-reg* provide a new address mapping method called relay segment to transfer message.

process can create multiple *x-entries*. All the *x-entries* are stored in a table named *x-entry-table*, which is a global memory region pointed by a new register *x-entry-table-reg*. Each *x-entry* has an ID, which is its index within the *x-entry-table*. A new register, *x-entry-table-size*, controls the size of *x-entry-table*, and makes the table scalable. A caller needs an *xcall-cap* to invoke an *x-entry*. *xcall-cap* is short for "XPC call capability", which records the IPC capabilities of each *x-entry*. Two new instructions are provided for IPC call and return, respectively: "*xcall #reg*" and "*xret*", where #reg records an *x-entry* index provided by the OS kernel.

Lightweight Message Transfer: XPC engine provides a lightweight mechanism named *relay-seg* (short for relay segment) for message transfer. A *relay-seg* is a memory region backed with continuous physical memory. The translation of a *relay-seg* is done by a new register, *seg-reg*, instead of page tables. The *seg-reg* can be passed from a caller to a callee, thus the callee can directly access the data within the virtual address range indicated in its *seg-reg*. The OS kernel will ensure that the mapping of a *relay-seg* will never be overlapped by any mapping of page table; thus, no TLB shootdown is needed on this region.

Listing 1 Example code of XPC.

```
void xpc_handler(void* arg) {
  ... /* handler logic */
  xpc_return();
}

void server() {
  ... /* Register an XPC Entry */
  xpc_handler_thread = create_thread();
  max_xpc_context = 4;
  xpc_ID = xpc_register_entry( entry_handler,
          xpc_handler_thread, max_xpc_context);
}

void client() {
  /* get server's entry ID and capability
     from parent process */
  server_ID = acquire_server_ID("servername");
  xpc_arg = alloc_relay_mem(size);
  ... /* fill relay-seg with argument */
  xpc_call(server_ID, xpc_arg);
}
```

XPC Programming Model: The programming model of XPC is compatible with both capability based permission check and page table based isolation. Listing 1 shows an example code snippet. The server first registers an *x-entry* by passing the procedure handler, a handler thread and a max context number (indicating the max number of simultaneous callers). The handler thread is used to offer the runtime state for client threads and can be shared by multiple *x-entries*. After that, the server finishes the registration and is ready to serve IPC requests. The client gets the server's ID as well as the IPC capability, typically from its parent process or a name server. The IPC is performed through *xcall #reg*, and the message could be transferred through general purpose registers and *relay-seg*.

3.2 XPC Engine

xcall: The *xcall #reg* instruction is used to invoke an *x-entry* whose ID is specified by the register. The XPC engine performs four tasks: ① It first checks the caller's *xcall-cap* by reading the bit at #reg in the *xcall-cap* bitmap. ② Then the engine loads the target *x-entry* from *x-entry-table* and checks the valid bit of the entry. ③ After that, a *linkage record* is pushed to the *link stack*. Here we use the term *linkage record* to indicate the information necessary for return, which will be stored in a per-thread stack (called *link stack*). ④ Then, the processor loads the new page table pointer (flushes TLB if necessary) and sets the PC to the procedure's entrance address. The engine will put the caller's *xcall-cap-reg* in a register (e.g., *t0* in RISC-V), to help a callee to identify the caller. Any exceptions that happen in the process will be reported to the kernel.

xret: The *xret* instruction pops a linkage record from the *link stack* and returns to the previous process. The CPU first checks the valid bit of the popped linkage record. It then restores to the caller according to the *linkage record*.

***xcall-cap*:** The *xcall-cap* will be checked during an *xcall*. For performance concern, we use a bitmap to represent *xcall-cap* (other design options are discussed in §6.2). Each bit with index *i* represents whether the thread is capable of invoking a corresponding *x-entry* with ID *i*. The bitmap is stored in a per-thread memory region pointed by a register *xcall-cap-reg*, which will be maintained by the kernel and checked by the hardware during *xcall*.

Link Stack: As mentioned, a *link stack* is used to record the calling information (*linkage record*), which is a per-thread memory region pointed by a register *link-reg*, and can only be accessed by the kernel. In our current design, a *linkage record* includes page table pointer, return address, *xcall-cap-reg*, *seg-list-reg*, relay segment and a valid bit. The XPC engine does not save other general registers and leaves to XPC library and applications to handle them. The entries in *linkage record* can be extended to meet different architectures by obeying a principle that *linkage record* should maintain information which can not be recovered by user-space.

At the point of pushing the linkage record, XPC engine is ready to perform switching and can save the *linkage record* lazily. Thus, we can optimize *link stack* using a non-blocking approach to hide the latency of writing stack. As shown in §5.2, we can save 16 cycles using the optimization.

XPC Engine Cache: We add a dedicated cache to optimize memory accesses of the XPC engine to fetch the *x-entry* and capability. We have two observations for this design decision: ① IPC has high temporal locality (for a single thread); ② IPC is predictable. Based on the two observations, we use a software manageable cache for XPC engine to store *x-entries*. Prefetch is supported so that a user application can load an *x-entry* into the cache in advance. As shown in §5.2, we can save 12 cycles by prefetching.

3.3 Relay Segment

***relay-seg*:** A *seg-reg* register is introduced as an extension of the TLB module for mapping a *relay-seg*. It includes four fields: virtual address base, physical address base, length, and permission. The virtual region (from VA_BASE to VA_BASE + LEN) is directly mapped to a physical region (from PA_BASE to PA_BASE + LEN). During address translation, the *seg-reg* has higher priority over the page table.

***seg-mask*:** Figure 3 shows the registers and operations of a *relay-seg*. User applications cannot directly change the mapping of *seg-reg*. Instead, they can use a new register *seg-mask* to shrink the range of current *relay-seg* and pass the new range to the callee. This is useful when only a part of the message should be passed to the callee, especially along a calling chain. During an *xcall*, both the *seg-reg* and *seg-mask* are saved in the linkage record, and *seg-reg* is updated to the intersection of *seg-reg* and *seg-mask*. After that, the callee can access *relay-seg* just as the caller does.

Multiple *relay-segs*: A server can create multiple *relay-segs*, which will be stored in a per-process memory region called *seg-list* managed by OS kernel, which is pointed by a new register *seg-list-reg*. If one process needs to perform a call with another *relay-seg*, it can use a new instruction, *swapseg #reg*, to atomically swap the current *seg-reg* with the one indexed by #reg in its *seg-list*. By swapping with an invalid entry, a thread can invalidate the *seg-reg*.

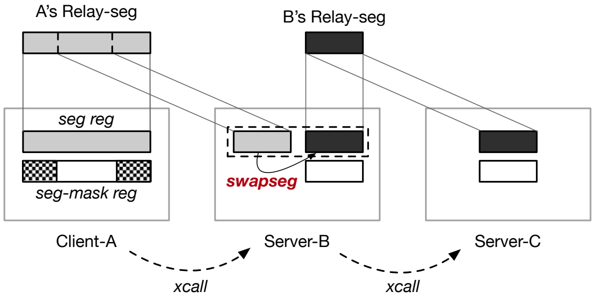


Figure 3: XPC mask and swap operations on *relay-seg*.

Ownership of *relay-seg*: To defend against TOCTTOU attacks, the kernel will ensure that each *relay-seg* can only be active on one CPU core at a time. In another word, an active *relay-seg* can only be owned by one thread, and the ownership will be transferred along its calling chain, so that two CPUs cannot operate one *relay-seg* at the same time.

Return a *relay-seg*: During an *xret*, the callee's *seg-reg* must be the same as when it is invoked. The XPC engine will ensure this by checking the current *seg-reg* with the *seg-reg* and *seg-mask* saved in the linkage record. The check is necessary; otherwise, a malicious callee may swap caller's *relay-seg* to its *seg-list* and return a different one. If the check fails, an exception will be raised, and the kernel will handle it.

4 IMPLEMENTATION

We describe our specific implementation of the general design, from four aspects: integration with the RocketChip RISC-V core, support for microkernels, support for monolithic kernels, and a user-level message handover mechanism.

4.1 Integration into RocketChip

We introduce the RTL prototype implementation in RocketChip and how OS kernel manages the XPC engine here.

XPC Engine: XPC engine is implemented as a unit of a RocketChip core. Table 2 shows detailed information about the new registers as well as instructions. The new registers are implemented as CSRs (Control and Status Registers) and can be accessed by *csrr* (CSR read) and *csrw* (CSR write) instructions. The three new instructions are sent to the XPC engine in *Execution* stage. XPC engine checks the validity of an IPC and returns the callee information to the pipeline. Five new exceptions are added, including invalid *x-entry*, invalid *xcall-cap*, invalid linkage, invalid *seg-mask*, and *swapseg* error.

The default implementation does not contain engine cache to minimize the hardware modifications, while a version with engine cache will be evaluated in the microbenchmark to show the performance. The engine cache contains only one entry and relies on software management including prefetching and eviction. The prefetch is invoked by *xcall* #reg, using a negative ID value (−ID) in the register to indicate a prefetch operation.

XPC Management: The kernel manages four XPC objects: 1) the global *x-entry* table, 2) per_thread *link stack*, 3) per_thread

Table 2: Registers and instructions provided by XPC engine.

Register Name	Access Privilege (R/W in kernel by default)	Register Length	Description
x-entry-table-reg		VA length	Holding base address of *x-entry-table*.
x-entry-table-size		64 bits	Controlling the size of *x-entry-table*.
xcall-cap-reg		VA length	Holding the address of *xcall* capability bitmap.
link-reg		VA length	Holding the address of *link stack*.
relay-seg	R/ in user mode	3*64 bits	Holding the mapping and permission of a relay segment.
seg-mask	R/W in user mode	2*64 bits	Mask of the relay segment.
seg-listp	R/ in user mode	VA length	Holding the base address of relay segment list.
Instruction	**Execution Privilege**	**Instruction Format**	**Description**
xcall	User mode	*xcall #register*	Switching page table base register, PC and *xcall-cap-reg*, according to the *x-entry* ID specified by the register. Pushing a linkage record to the *link stack*.
xret	User mode	*xret*	Returning to a linkage record poped from the *link stack*.
swapseg	User mode	*swapseg #register*	Switching current *seg-reg* with a picked one in the relay segment list and clearing the *seg-mask*.
Exception		**Fault Instruction**	**Description**
Invalid *x-entry*		*xcall*	Calling an invalid *x-entry*.
Invalid *xcall-cap*		*xcall*	Calling an *x-entry* without *xcall-cap*.
Invalid linkage		*xret*	Returning to an invalid linkage record.
Swapseg error		*swapseg*	Swapping an invalid entry from relay segment list.
Invalid *seg-mask*		*csrw seg-mask, #reg*	Masked segment is out of the range of *seg-reg*.

xcall capability bitmap and 4) per_address_space relay segment list. During the system boot, the kernel allocates the memory for the *x-entry* table and sets the table size (1024 entries in our current implementation). When creating a thread, it allocates 8KB memory for the thread's *link stack*, 128B memory as the capability bitmap and one 4KB page for the *seg-list*. During a context switch, the kernel saves and restores the per_thread objects.

4.2 Support for Microkernels

Capability: Capabilities have been widely used by microkernels [39, 58] for IPC. To transfer a *xcall-cap* between threads, our implementation introduces a software capability, *grant-cap*, which allows a thread to grant a capability (either *xcall* or *grant*) to another thread. The kernel will maintain and enforce the grant capability list for each thread. When a thread creates an *x-entry*, it will have the *grant-cap* of the new *x-entry*, and can grant the *xcall-cap* to other threads.

Split Thread State: The domain switch in user mode may lead to misbehavior of the kernel since the kernel is not aware of the current running thread. For example, caller A issues *xcall* to callee B, which then triggers a page fault and traps to the kernel. Since the kernel is now aware of the *xcall*, it will mistakenly use A's page table to handle B's page fault.

To solve this problem, we borrow the idea from migrating thread [29] and separate the kernel-maintained thread state into two parts: *scheduling state* and *runtime state*. The scheduling state contains all the scheduling related information, including kernel stack, priority, time slice, etc. The runtime state contains the current address space and capabilities, which are used by the kernel to serve this thread. Each thread is bound with one scheduling state but may have different runtime states when running. In our implementation, we use *xcall-cap-reg* to determine runtime states. Once a thread traps to the kernel, the kernel will use the value of *xcall-cap-reg* to find current runtime state, as this register is per-thread and will be updated by hardware during *xcall*.

Per-invocation C-Stack: The thread model of XPC supports one *x-entry* of a server to be invoked by multiple clients at the same time. In XPC, our library provides a per-invocation XPC context, which includes an execution stack (called C-Stack) and local data, to support simultaneous cross procedure calls.

When creating an *x-entry*, a server specifies a max number of XPC contexts for it. The XPC library will create these contexts in advance, and add a trampoline for each *x-entry*. The trampoline will select an idle XPC context, switch to the corresponding C-Stack and restore the local data before invocation, and release the resources before return. If no idle context is available, the trampoline either returns an error or waits for an idle one.

Such implementation may introduce DoS attacks, e.g., a malicious client may excessively invoke one *x-entry* to occupy all its available contexts. To address this problem, XPC allows each server to adopt specific policies to limit the invocation from clients. This problem can also be mitigated by adopting credit systems, as in M3 [13] and Intel QP [3].

Application Termination: Our implementation also considers that any procedure along a calling chain may terminate abnormally, which may affect the entire chain. For example, consider a calling chain: A → B → C, where B is killed by the kernel due to some exception. When C invokes *xret*, it may return to a wrong process. In this case, we need a way to trigger an exception and let the kernel handle it.

In our implementation, when a process terminates, the kernel will scan all the *link stack*s and mark all of the process's linkage records (by comparing the page table pointer) as invalid. Thus, in the previous example, once C returns, the hardware will trigger an exception, and the kernel will pop B's linkage record and return to A with a timeout error.

We also introduce an optimization to reduce the frequency of *link stack* scanning: when B is killed, the kernel will zero B's page table (the top level page) without scanning. Thus a page fault will be triggered when C returns to B, and the kernel gets a chance to handle. The kernel will also revoke the resource of B (§4.4).

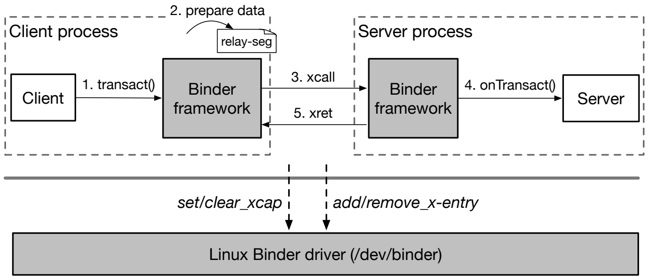


Figure 4: XPC for Android Binder. Rectangular boxes denote components in Android; shaded boxes denote modified parts.

4.3 Support for Android Binder

Android Binder is a significant extension introduced by Android for inter-process communication. It has been widely used in Android components, including window manager, activity manager, input method manager, etc [12]. Android Binder is composed of several layers, including the Linux Binder driver, the Android Binder framework (i.e., C++ middleware), and the API (e.g., Android interface definition language). Our modification focuses on the driver and framework but keeps the API unmodified to meet existing applications.

Binder uses *Binder transaction* to represent a cross-process method invocation and uses kernel "twofold copy" to transfer data (named *transaction buffer*). Besides, Android also introduces *ashmem* (anonymous shared memory [11]) to boost the performance of bulk memory transfer between processes.

Binder Transaction: The process of a Binder transaction between a client and a server involves the following steps:

(1) The client prepares a *method code* representing the remote method to call along with marshaled data (*Parcels* in Android).
(2) The client Binder object calls *transact().* This call will be passed through the Linux Binder driver, which copies the data from the userspace transaction buffer (through *copy_-from_user*), switches to the server side, and copies the data from the kernel (through *copy_to_user*). (**Two memory copyings and two domain switchings.**)
(3) The Binder framework in the server side receives the request and handles this call by invoking *onTransact()* method, which is prepared by the server in advance.
(4) The server replies the request through the Linux Binder driver.

As shown in the Figure 4, we optimize the above process using XPC. First, we keep the IPC interfaces provided by Android Binder framework (e.g., *transact()* and *onTransact()*) unmodified to support existing applications. Besides, we extend the Binder driver to manage the *xcall-cap* capabilities (i.e., the *set_xcap* and *clear_xcap* interfaces) and *x-entry table* (i.e., the *add_x-entry* and *remove_x-entry* interfaces). When a server process registers a service through Binder interfaces (e.g., *addService*), the modified framework will issue an *ioctl* command to the Linux Binder driver to add an *x-entry.* Similarly, the framework will issue *set-xcap* when a client process asks for a service through API (e.g., *getService*). Last, instead of invoking the *ioctl*, the framework will uses *xcall* and *xret* for remote method invocation, and uses relay segment to implement *Parcels* for data transfer. Moreover, the Linux kernel should also maintain the XPC registers for threads, just like in §4.2. Optimized by XPC, domain switchings and memory copying are eliminated.

Ashmem: The anonymous shared memory (ashmem) subsystem provides a file-based shared memory interface to userspace. It works like anonymous memory, but a process can share the mappings with another process by sharing the file descriptor. The shared memory can be accessed via both mmap or file I/O. In Android Binder, processes can share file descriptors of an ashmem through Binder driver.

Like conventional shared memory approaches, ashmem also needs *an extra copying* to avoid TOCTTOU attacks. We use relay segment in XPC to implement the ashmem.

- **ashmem allocation**: The Binder framework allocates an ashmem by allocating a relay segment from Binder driver.
- **ashmem map**: The memory map operation will allocate virtual addresses for the segment and set the relay seg register.
- **ashmem transfer**: The ashmem can be transferred among processes by passing the *seg-reg* register in the framework during *xcall.*

Using the relay segment, the framework can avoid the additional copying in the server side, as the ownership of the mapping has been transferred. However, one limitation is that, in the prototype implementation, there is only one active relay segment at a time. Thus we rely on the page fault (implicitly)/*swapseg* (explicitly) to switch the active relay segment when applications need accesses to several ashmems at the same time.

4.4 Handover along Calling Chain

The *relay-seg* mechanism allows a segment of data to be passed along the calling chain. In different situations, processes may have different handover usages. Suppose a calling chain: $A \to B \to C$, where A passes a message M along the chain. Here we need to overcome three challenges to support zero-copying handover. First, B may append data to M, e.g., a network stack may append headers to payload data. Such appending may exceed the boundary of a *relay-seg.* Second, C's interface may only accept small pieces of data, e.g., a file system server will split data into block size and send them to disk server one by one. Third, when C is executing, B may terminate abnormally, which requires revocation of its *relay-seg*s.

Message Size Negotiation: Message size negotiation is proposed to handle the first challenge. It allows all processes in a calling chain to negotiate a proper message size so that the client can reserve some space for appending. The negotiation is performed recursively. Given a calling chain: $A \to B \to [C|D]$, where B may call C or D. $S_{self}(B)$ represents the size B will add, $S_{all}(B)$ represents the size B and all its possible callees will add. Thus, $S_{all}(B)$ **equals** $S_{self}(B)$ **adds** $Max(S_{all}(C), S_{all}(D))$. For the first time A invokes B, it asks B the $S_{all}(B)$, so that A can reserve enough space for the whole chain. Servers can also have their implementations of size negotiation.

Message Shrink: Message shrink allows a caller to pass a part of the message to its callee, with the help of *seg-mask.* For example, A puts 1MB M in *relay-seg* and passes it to B. B can use the *relay-seg*

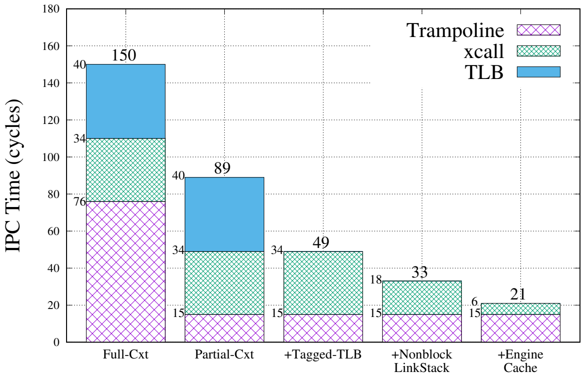


Figure 5: XPC optimizations and breakdown.

with different 4KB-size *seg-mask*s to pass data to *C* iteratively, just like a sliding window.

Segment Revocation: Segment revocation is done by the kernel when a process terminates. The kernel will scan *seg-list* of the process, return callers' *relay-seg*s to callers, and revoke other *relay-seg*s.

5 EVALUATION

To evaluate XPC, this section answers several questions:

- How XPC improves the IPC performance? (§5.2)
- How OS services benefit from XPC? (§5.3)
- How applications benefit from XPC? (§5.4)
- How the Android Binder benefits from XPC? (§5.5)
- How portable is XPC on other architectures? (§5.6)
- How much hardware resource XPC costs? (§5.7)

5.1 Methodology

We implemented the XPC engine based on two open-source RISC-V [64] implementations: siFive Freedom U500 [8] (on Xilinx VC707 FPGA board) and lowRISC [5] (on Xilinx KC705 FPGA board). We have ported two state-of-the-art microkernels, seL4[3] on siFive Freedom U500 and Zircon [2] on lowRISC, and added XPC support in both systems. Besides, we port the Android Binder framework, libBinder, to Freedom U500 (by modifying the synchronization assembly code to RISC-V) with Linux 4.15 and optimize the synchronous IPC in Binder with XPC. We evaluate the performance of six systems: Zircon, seL4, Android Binder, Zircon-XPC, seL4-XPC and Binder-XPC. Besides the FPGA hardware, we also port XPC on the GEM5 simulator for ARMv8 to validate the generality of XPC.

5.2 Microbenchmark

Optimizations and Breakdown: We use the following five configurations with different optimizations enabled to measure the latency of IPC and show the breakdown of performance benefit.

- **Full-Ctx**: saving and restoring full context.
- **Partial-Ctx**: saving and restoring partial context.
- **+Tagged TLB**: enabling previous optimizations and adopting tagged TLB to mitigate TLB miss.

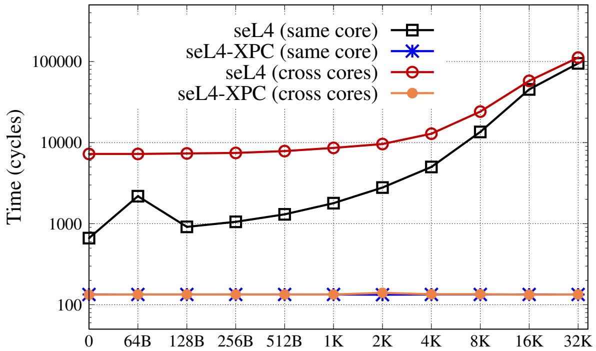


Figure 6: One-way call.

- **+Nonblock Link Stack**: enabling previous optimizations and adopting nonblock link stack.
- **+XPC Engine Cache**: enabling previous optimizations and adopting cache for XPC engine.

Figure 5 shows the cycles of one IPC call using different configurations. In the "Full-Cxt" configuration, as the RocketChip does not support tagged TLB yet, it will incur about 40 cycles of TLB flush/miss penalty. The trampoline code (mentioned in §4.2) takes 76 cycles to save and restore the general purpose registers. The logic of *xcall* takes about 34 cycles. The "partial-context" optimization only consider necessary registers (e.g., stack point register and return address register) and reduce the trampoline code to 15 cycles. The TLB misses could be mitigated by adopting tagged TLB. The "Nonblock Link Stack" hides the latency of pushing *linkage record*, which can reduce the latency by 16 cycles. The "Engine Cache" uses prefetching to further reduce the latency by 12 cycles. With all the optimization, one *xcall* can achieve 6 cycles and one IPC only spend 21 cycles.

In the following evaluation, XPC will use "Full-Cxt" with "Nonblocking Link Stack" optimizations, to ensure the fairness of the comparison.

One-way Call: We also evaluated the one-way call performance. A client calls a server with different message sizes. We calculate the cycles from the client invoking a *call* to the server getting the request. As shown in Figure 6, seL4-XPC has 5-37x speedup over the fast path of seL4. One reason is that seL4-XPC uses *relay-seg* to transfer messages, while in seL4, the kernel-copying is only used when the message is less than 120 bytes, and it uses shared memory to transfer large message. As the message size grows, the speedup comes more from the benefit of *relay-seg*. seL4 only uses slow path when the message size is medium (64B here).

Zircon-XPC can have 60x speedup when the message size is small, due to the elimination of scheduling and kernel involvement. Zircon uses kernel twofold copying to transfer messages and does not optimize the scheduling in the IPC path, which makes it much slower than seL4.

Multi-core IPC: As shown in Figure 6, the performance of cross-core IPC is improved from 81x (small message) to 141x (4KB message size). Thanks to the migrating thread IPC model adopted by XPC,

[3] seL4 already supports RISC-V. Our porting work mainly focuses on adding SMP support.

Table 3: Cycles of hardware instructions in XPC.

Instruction	Cycles
xcall	18
xret	23
swapseg	11

the server's code is running in the client's context. XPC also provides better TLB (in relay segment) and cache locality. Moreover, a client can easily scale itself by creating several worker threads on different cores and pull the server to run on these cores.

Other Instructions: We measure the cycles for *xcall*, *xret* and *swapseg* instructions (results are shown in Table 3). Besides the stated *xcall*, *xret* takes 23 cycles and *swapseg* takes 11 cycles. The costs of the three instructions are small and mainly come from the memory operations. Microkernels can implement efficient IPC based on these primitives.

5.3 OS Services

To show how IPC influences the performance of microkernel, we evaluate the performance of two OS services: file system and network subsystem.

File System: In microkernels, a file system usually includes two servers, a file system server and a block device server (e.g., in Zircon, the MiniFS and the in-memory ram disk server). We port a log-based file system named xv6fs from fscq [21], a formally verified crash-safe file system, to both Zircon and seL4. A ramdisk device is used as the block device server.

We test the throughput of the file read/write operations. The results are shown in Figure 7(a) and (b). Zircon uses kernel two-fold copy, and seL4 uses shared memory. We implement seL4-one-copy version which needs one copying to meet the interfaces (having TOCTTOU issue) and seL4-two-copy version, which requires two copying and provides higher security guarantee. XPC optimized systems can achieve zero-copying without TOCTTOU issue. On average, XPC achieves 7.8x/3.8x speedup compared with Zircon/seL4 for read operations, and 13.2x/3.0x speedup for write operations.

The improvement mainly comes from both faster switch and zero-copying of XPC, especially for write operations, which will cause many IPCs and data transfers between the file system server and the block device server.

Network: Microkernel systems usually have two servers for network: one network stack server (including all network protocols) and one network device server. We use lwIP [6], a network stack used by Fuchsia (a full-fledged OS using Zircon), as our network stack server. A loopback device driver, which gets a packet and then sends it to the server, is used as the network device server. We do not port lwIP to seL4, so we only consider Zircon in this test.

We evaluate the throughput of TCP connection with different buffer sizes. The result is shown in Figure 7(c). On average, Zircon-XPC is 6x faster than Zircon. For small buffer size, Zircon-XPC achieves up to 8x speedup, and the number decreases as the buffer size grows. This is because lwIP buffers the client messages for batching, so increasing buffer size will reduce the numbers of IPC, which improves the performance of the original Zircon due to its high IPC latency.

5.4 Applications

To show how XPC improves the performance of real-world applications, we evaluate the performance of a database and a web server.

Sqlite3: Sqlite3 [9] is a widely-used relational database. We use the default configuration with journaling enabled, and measure the throughput with different workloads (YSCB benchmark workloads). Each workload is performed on a table with 1,000 records. The result is shown in Figure 8(a) and (b). On average, XPC achieves 60% speedup in seL4 and 108% in Zircon.

YCSB-A and YCSB-F gain the most improvement because they have a lot of write/update operations which will trigger frequent file access. YSCB-C has minimal improvement since it is a read-only workload and Sqlite3 has an in-memory cache that can handle the read request well.

Web Server: Three servers are involved in this test: an HTTP server, ported from lwIP, which accepts a request and then returns a static HTML file; an AES encryption server which encrypts the network traffic with a 128-bit key; an in-memory file cache server which is used to cache the HTML files in both modes. The HTTP server is configured with both encryption-enabled mode and encryption-disabled mode. A client continuously sends HTTP requests to the web server. The throughput is measured and the result is shown in Figure 8(c). XPC has about 10x speedup with the encryption and about 12x speedup without encryption. Most of the benefit comes from handover optimization. Since in multi-server cases, the message will be transferred multiple times. Using handover can efficiently reduce the times of memory copying in these IPC.

5.5 Android Binder

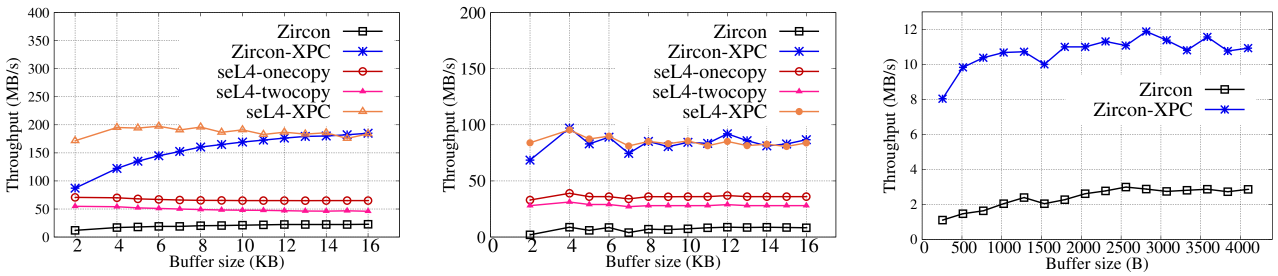
Binder. We evaluate the Binder by simulating the communication between the window manager and a surface compositor. In the case, the surface compositor will transfer the surface data to the windows manager through Binder, and then the windows manager need to read the surface data and draw the associated surface.

We consider two Binder facilities, passing data through Binder buffer and passing data through ashmem, and evaluate the latency for the communication. The result is shown in the Figure 9, where the latency time includes the data preparation (client), the remote method invocation and data transfer (framework), handling the surface content (server), and the reply (framework).

Figure(a) reveals the result using a buffer for communication. The latency of Android Binder is 378.4us for 2KB data and 878.0us for 16KB data (average value of 100 times run), while the Binder-XPC achieves 8.2us for 2KB data (46.2x improvement) and 29.0us for 16KB data (30.2x improvement). Notably, the buffer size is restricted in Android (e.g., less than 1MB).

The result of using ashmem for data transfer in Binder is shown in figure(b). The latency of Android Binder is from 0.5ms for 4KB surface data size to 233.2ms for 32MB surface data size, while the Binder-XPC achieves 9.3us for 4KB data (54.2x improvement) and 81.8ms for 32MB data (2.8x improvement).

Ashmem. We use the same case to evaluate the latency when we only adopt the relay segment (Ashmem-XPC in the figure) for optimizing ashmem. As shown in Figure(b), the Ashmem-XPC achieves

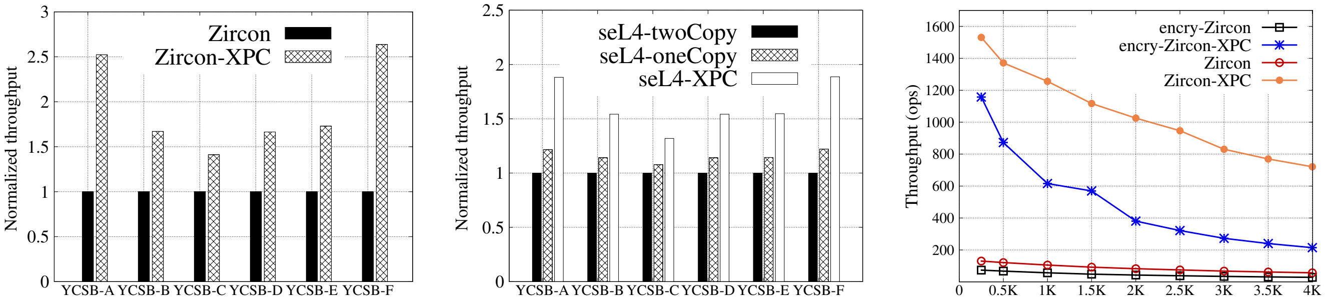


(a) FS read throughput.

(b) FS write throughput.

(c) TCP throughput.

Figure 7: Figure (a) and (b) show the read/write throughput of the file system with different buffer sizes. Figure (c) shows the throughput of TCP with different buffer sizes. Higher the better.

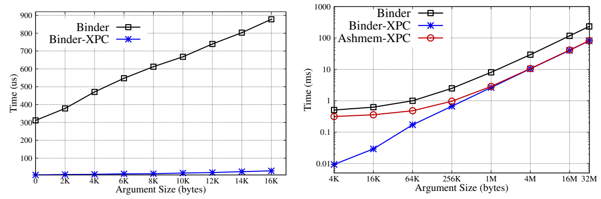


(a) Sqlite3(Zircon) throughput.

(b) Sqlite3(seL4) throughput.

(c) HTTP server throught.

Figure 8: Figure (a) and (b) show the normalized throughput of Sqlite3 with YCSB's workloads. Figure (c) shows the throughput of an HTTP server (with & without encryption). Higher the better.



(a) Binder using buffer.

(b) Binder using ashmem.

Figure 9: Android Binder evaluation. Figure (a) and (b) show the remote method invocation latency between the windows manager and surface compositor with different argument sizes. Lower the better.

0.3ms for 4KB data (1.6x improvement) and 82.0ms for 32MB data (2.8x improvement). The improvement mainly comes from the secure zero-copying message transfer.

Discussion: Overall, XPC can effectively optimize the performance of Android Binder and Ashmem. We also have some limitations. Currently, the prototype only optimizes synchronous IPC in Binder (asynchronous IPC usage like *death notification* is not supported yet). And we do not apply the split thread state approach in the Linux kernel. Instead, we leverage machine mode in RISC-V to trap and handle any exception between *xcall* and *xret* (rare in the experiments). Compared with *xcall* and *xret*, relay segment is more suitable for monolithic kernels as it introduces small modification.

5.6 Generality

XPC is a general design supporting different architectures. Besides a RISC-V implementation on FPGA, we also implement it on ARM platform using the cycle-accurate GEM5 simulator [18]. The implementation is based on ARM HPI (High-Performance In-order) model [1]. The simulation parameters, listed in Table 4, mimic a modern in-order ARMv8-A implementation.

Table 4: Simulator configuration.

Parameters	Values
Cores	8 In-order cores @2.0GHz
I/D TLB	256 entries
L1 I/D Cache	32KB, 64B line, 2/4 Associativity
L1 Access Latency	data/tag/response (3 cycle)
L2 Cache	1MB, 64B line, 16 Associativity
L2 Access Latency	data/tag (13 cycles), response (5 cycle)
Memory Type	LPDDR3_1600_1x32

We use microOps to implement the functionalities of XPC engine. By carefully choosing the order, we can avoid speculative issues in the *xcall* and *xret* instructions. We set the entries of endpoint table to 512, length of capability bitmap to 512 bits, and call stack to 512 entries. Any normal load/store instructions on these regions will trigger an exception. The implementation does not contain any optimizations like nonblocking *link stack* and engine cache.

We choose IPC of seL4 as a baseline. Since seL4 does not have GEM5 support, we dump the instruction trace of seL4's *fastpath_call* and *fastpath_reply_recv* using a record-and-replay tool named Panda [24]. We run the trace directly in GEM5. The result is shown in the table 5, which only consider the IPC logic part in seL4 and *xcall*/*xret* in XPC.

Table 5: IPC cost in ARM, (TLB flushing is about 58 cycles, which can be removed with tagged TLB).

Systems	IPC Call	IPC Ret
Baseline (cycles)	66 (+58)	79 (+58)
XPC (cycles)	7 (+58)	10 (+58)

Table 6: Hardware resource costs in FPGA.

Resource	Freedom	XPC	Cost
LUT	44643	45531	1.99%
LUTRAM	3370	3370	0.00%
SRL	636	636	0.00%
FF	30379	31386	3.31%
RAMB36	3	3	0.00%
RAMB18	48	48	0.00%
DSP48 Blocks	15	16	6.67%

Both the baseline and XPC have better performance than real hardware. One of the reason is that GEM5 do not simulate the TLB flushing costs (in ARM)[4]. We evaluate the cost of updating TTBR0 with instruction barrier (*isb* instruction) and data barrier (*dsb* instruction) in Hikey-960 board (ARMv8) and the cost is about 58 cycles.

Using XPC, the IPC logic part is improved: from 66 (+58 TLB cost) cycles to 7 (+58 TLB cost) cycles when the message transfer size is small and the cache is warm. The implementation on GEM5 confirms the generality of XPC design.

5.7 Hardware Costs

As we use Vivado [10] tool to generate the hardware, we can gain the resource utilization report in the FPGA from it. The hardware costs report is shown in Table 6 (without engine cache). The overall hardware costs are small (1.99% in LUT and 0.00% in RAM). By further investigating the resource costs, we found that CSRFile in XPC uses more 372 LUTs and 273 FFs than baseline (to handle the 7 new registers), while XPC engine uses 422 LUTs, 462 FFs, and 1 DSP48 blocks.

The utilization certainly could be further optimized, like using Verilog instead of Chisel in RocketChip. The low hardware costs make XPC possible to be applied in existing processors (e.g., Intel x86 and ARM).

6 DISCUSSION

6.1 Security Analysis

XPC Authentication and Identification: A caller cannot direct issue *xcall ID* to invoke an XPC without the corresponding *xcall-cap*. It may request the *xcall-cap* from a server with the corresponding *grant-cap*, just like the *name server* [23] in L4. A callee can identify a caller by its *xcall-cap-reg*, which will be put into a general purpose register by XPC engine and cannot be forged.

Defending TOCTTOU Attacks: TOCTTOU attacks happen due to the lack of *ownership transfer* of the messages. In XPC, a message is passed by a *relay-seg*, which is owned by only one thread at a time. Meanwhile, the kernel will ensure that a *relay-seg* will not overlap with any other mapped memory range. Thus, each owner can exclusively access the data in a *relay-seg*, which can inherently defend against TOCTTOU attacks.

Fault Isolation: During an *xcall*, a callee crash will not affect the execution of the caller and vice versa, as stated in §4.2 and §4.4. If the callee hangs for a long time, the caller thread may also hang. XPC can offer a timeout mechanism to enforce the control flow to return to the caller in this case. However, in practice the threshold of timeout is usually set to 0 or infinite [26], which makes the timeout mechanism less useful.

Defending DoS Attacks: A malicious process may try to issue DoS attacks by consuming far more hardware resources than it needs. One possible attack is to create a lot of *relay-seg* which requires many continuous physical memory ranges, which may trigger external fragmentation. In XPC, a *relay-seg* will use the process's private address space (i.e., untyped memory as seL4 [39]), which will not affect other processes or the kernel. Another case is that, a malicious caller may exhaust the callee's available contexts by excessively calling the callee. We can use credit systems [3, 13] to overcome the issue. The callee will first check whether the caller has enough credits before assigning an XPC context to it.

Timing Attacks: XPC Engine Cache may be the source of timing attacks, but very hard since the number of entries is small (one in the paper). Moreover, the issue can be mitigated by adding tag in the Engine Cache like tagged-TLB. As each Cache entry is private for a thread (with tag), the timing attacks could be mitigated.

6.2 Further Optimizations

Scalable *xcall-cap*: *xcall-cap* is implemented as a bitmap in our prototype. It is efficient but may have scalability issue. An alternative approach is to use a radix-tree, which has better scalability but will increase the memory footprint and affect the IPC performance.

Relay Page Table: The relay segment mechanism has a limitation that it can only support contiguous memory. This issue can be solved by extending the segment design to support a page table design. A relay page table can be similar with previous dual-page-table design [56]. The page table walker will choose the different page table according to the VA being translated. However, the ownership transfer property will be harder to achieve, and relay page table can only support page-level granularity.

7 RELATED WORK

There is a long line of research on reducing the latency of domain switch as well as message transfer to optimize IPC. We revisit previous IPC optimizations in this section and show the comparisons in the Table 7.

7.1 Optimizations on Domain Switch

Software Optimizations: One widely adopted optimization is to use caller's thread to run callee's code in callee's address space, as in PPC (protected procedure call) [17, 30] and migrating thread model [22, 29]. This optimization eliminates the scheduling latency and mitigates IPC logic overhead, and has been used in LRPC [17] and the new version of Mach [29]. Tornado [30] also adopts PPC as its execution model. Besides, it leverages another feature of PPC,

[4] We confirmed this with the GME5 community.

Table 7: Systems with IPC optimizations. Δ means TLB flush operations. N means the number of IPC in a calling chain.

Systems			Domain switch			Message passing				
Type	Name	Addr Space	Description	w/o trap	w/o sched	Description	w/o TOCTTOU	Hand over	Granularity	Copy time
Baseline	Mach-3.0	Multi	Kernel schedule	✗	✗	Kernel copy	✓	✗	Byte	2*N
Software optimization	LRPC [17]	Multi	Protected proc call	✗	✓	Copy on A-stack	✓	✗	Byte	2*N
	Mach (94) [29]	Multi	Migrating thread	✗	✓	Kernel copy	✓	✗	Byte	N
	Tornado [30]	Multi	Protected proc call	✗	✓	Remapping page	✓	✗	Page	0+Δ
	L4 [45]	Multi	Direct proc switch	✗	✓	Temporary mapping	✓	✗	Byte	N
Hardware optimization	CrossOver [44]	Multi	Direct EPT switch	✓	✓	Shared memory	✗	✗	Page	N-1
	SkyBridge [50]	Multi	Direct EPT switch	✓	✓	Shared memory	✗	✗	Page	N-1
	Opal [20]	Single	Domain register	✓	✓	Shared memory	✗	✗	Page	N-1
	CHERI [65]	Hybrid	Function call	✓	✓	Memory capability	✗	✓	Byte	0
	CODOMs [62, 63]	Single	Function call	✓	✓	Cap reg + perm list	✗	✓	Byte	0
	DTU [13]	Multi	Explicit	✓	✓	DMA-style data copy	✓	✗	Byte	2*N
	MMP [67]	Multi	Call gate	✗	✓	Mapping + grant perm	✗	✗	Byte	0+Δ
	XPC	Multi	Cross process call	✓	✓	Relay segment	✓	✓	Byte	0

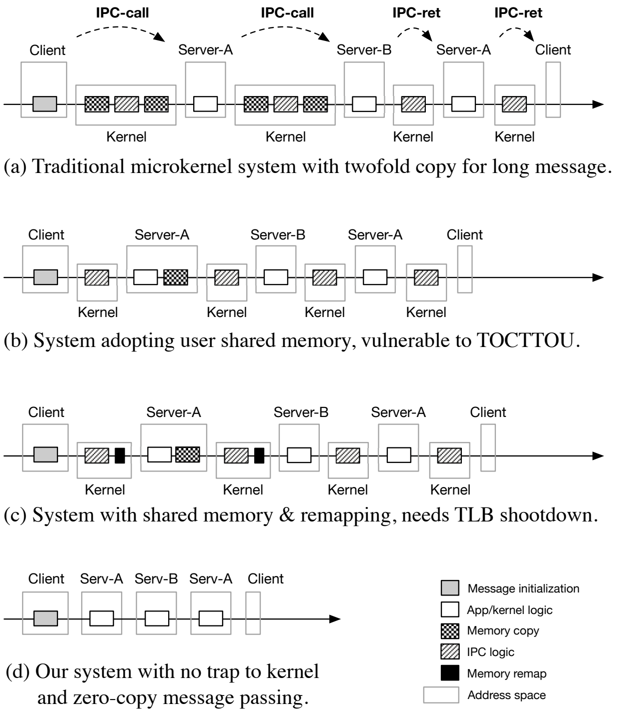
"fine data locality", to mitigate the data cache miss penalty caused by domain switching.

L4 [34, 45, 47, 48] uses a similar technology called "direct process switch" that supports address space switching between caller and callee with a small cost in kernel. It also adopts "lazy scheduling" to avoid frequent run-queue manipulations to reduce cache miss and TLB miss by careful placement.

Hardware Optimizations: New hardware extensions are also proposed to improve the performance of cross domain calls, like Opal [20], CHERI [65, 66] and CODOMs [62, 63]. Here a domain is not a process isolated by address space, but a new abstraction of execution subject (e.g., a piece of code) that has its own identity (e.g., domain ID). During a domain switch, the identity will be changed either explicitly (e.g., ID saved in register) or implicitly (e.g., ID implied by program counter). The switch can be done directly at unprivileged level without trapping to the kernel, which is a huge advantage against software optimizations. Meanwhile, multiple domains can share one address space, which can further reduce the overhead of TLB miss after domain switch. However, these systems usually require non-trivial changes to existing microkernels which are designed for address space based isolation mechanism. To achieve better compatibility, systems like CHERI adopt a hybrid approach using both capability and address space, but switching between address spaces still requires kernel involvement. CrossOver [44] and SkyBridge [50] leverage a hardware virtualization feature, VMFUNC, which enables a virtual machine to directly switch its EPT (extended page table) without trapping to the hypervisor. However, the feature is only suitable for virtualization environment.

7.2 Optimizations on Message Passing

Software Optimizations: For long message passing, one simple, secure but not efficient method is to adopt "twofold copy" (caller → kernel → callee), as shown in Figure 10(a). Some systems, e.g., LRPC [17], leverage user-level memory sharing to transfer messages and reduce the time of copying from two to one (caller → shared buffer), as shown in Figure 10(b). However, this design may affect the security since a malicious caller may change the message at any time when the callee is running, e.g., right after the callee checks the validity of the message, which leads to a TOCTTOU attack. One solution would be that the callee copies the message to its own space, but that will eat up the one-copy benefit [17]. Another solution is to change the ownership of shared memory by remapping (Figure 10(c)). However, memory remapping requires kernel's involvement and causes TLB shootdown. Meanwhile, since such memory is usually shared between **two** processes, if a message needs to be passed through **multiple** processes on an invocation chain, it has to be copied from one shared memory to another.



(a) Traditional microkernel system with twofold copy for long message.

(b) System adopting user shared memory, vulnerable to TOCTTOU.

(c) System with shared memory & remapping, needs TLB shootdown.

(d) Our system with no trap to kernel and zero-copy message passing.

Figure 10: Mechanisms for long message passing.

L4 [45] applies *temporary mapping* to achieve direct transfer of messages. The kernel will first find an unmapped address space of the callee, and map it temporarily into the caller's communication window, which is in caller's address space but can only be accessed by the kernel. Thus, one copy is achieved (caller → communication window). Meanwhile, since the caller cannot access the communication window, it has no way to change the message after sending it. However, the caller still requires the kernel to do the copying and remapping which will cause non-negligible overhead.

Hardware Optimizations: Many hardware-assisted systems [19, 62, 63, 65, 66] leverage capability for efficient message transfer among domains. CODOMs [62] uses a hybrid memory granting mechanism combined with permission list (domain granularity) and capability registers (byte granularity). By passing a capability register, a region of memory can be passed from caller to callee and forward. However, the owner of the region can access the region anytime which makes the system still vulnerable to TOCTTOU attack. CHERI [65] uses hardware capability pointers (which describe the lower and upper bounds of a memory range) for memory transfer. Although the design has considered TOCTTOU issues for some metadata (e.g., file descriptors), it still suffers TOCTTOU attacks for the data. Meanwhile, these systems are designed for single address space and use tagged memory for isolation.

Opal [40] and MMP [67, 68] propose new hardware design to make message transfer more efficiently. They use PLB (protection look-aside buffer) to decouple the permission from the address space translation to achieve byte granularity sharing. However, without additional data copy, they can neither mitigate the TOCTTOU attack nor support long messages handover along the calling chain.

M3 [13] leverages a new hardware component, DTU (data transfer unit), for message transfer. DTU is similar to DMA, which can achieve high throughput when the message is large, and allow efficient cross-core data transfer. However, it is known that DMA is not suitable for small and medium-size data [51], since the overhead of DMA channel initialization cannot be well amortized. HAQu [41] leverages queue-based hardware accelerators to optimize cross-core communication, while XPC can support a more general format of the message.

Table 7 summaries the characteristics of systems with different IPC optimizations. As it shows, the hardware methods can achieve better performance, e.g., faster domain switch by eliminating kernel trapping. However, these methods usually require significant changes to not only hardware (e.g., CHERI [65] adds 33 new instructions) but also software (e.g., not support existing microkernels designed for address space based isolation).

7.3 Other Related Work

Architectural Support for OS Performance: XPC continues the line of work in the community on architectural support for OS performance [14, 15, 31, 33, 38, 52–54]. Specifically, XPC contributes a hardware-assisted primitive that significantly improves the performance of IPC. This, combined with other recent architectural support, will significantly boost OS performance for various workloads.

Shared Memory for IPC: Many operating systems adopt the idea of using shared memory for message passing [20, 25, 44]. Fbufs [25] uses memory remapping and shared memory to achieve effective data transfer among protection domains. While appealing for performance, they are vulnerable to TOCTTOU attacks and can not support general handover.

Asynchronous IPC: Asynchronous IPC, as a complement to synchronous IPC, has been studied extensively [16, 26, 39, 59, 60]. FlexSC [59] proposes asynchronous system calls for batching processing and reducing mode switch between user and kernel. Barrelfish [16] uses asynchronous IPC to achieve non-blocking message exchanges among cores. Although asynchronous IPC can bring good throughput due to its batching strategy, it usually cannot achieve low latency at the same time.

Global Address Space Systems: Global virtual address systems [20, 37, 62, 63] put all the domains into a single address space. Enforcing the isolation among different domains within the same address space usually needs isolation mechanisms other than paging. For example, Singularity [37] relies on software verification and language support to enforce the isolation and behavior correctness of programs in the system. XPC has better compatibility and can be easily adopted by existing microkernels.

8 CONCLUSION

This paper has presented XPC, a new architectural extension to support fast and secure IPC. The extension is compatible with traditional address space isolation and can be easily integrated with existing OS kernels. Our evaluation shows that XPC can significantly improve the performance of various workloads of modern microkernels and Android Binder.

ACKNOWLEDGMENTS

This work is supported by the National Key Research & Development Program (No. 2016YFB1000104). The corresponding author of the paper is Yubin Xia.

REFERENCES

[1] 2018. Arm System Modeling Research Enablement Kit. https://developer.arm.com/research/research-enablement/system-modeling. Referenced November 2018.

[2] 2018. Fuchsia. https://fuchsia.googlesource.com/zircon. Referenced November 2018.

[3] 2018. An Introduction to the Intel QuickPath Interconnect. https://www.intel.de/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf. Referenced November 2018.

[4] 2018. iOS Security-iOS 12.1. https://www.apple.com/business/site/docs/iOS_-Security_Guide.pdf.

[5] 2018. lowRISC. https://www.lowrisc.org/. Referenced November 2018.

[6] 2018. lwIP. https://savannah.nongnu.org/projects/lwip/. Referenced May 2018.

[7] 2018. seL4 Benchmark Performance. https://sel4.systems/About/Performance/home.pml. Referenced November 2018.

[8] 2018. SiFive. https://www.sifive.com/. Referenced November 2018.

[9] 2018. SQLite. https://www.sqlite.org/index.html. Referenced May 2018.

[10] 2018. Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html. Referenced August 2018.

[11] 2019. Anonymous shared memory (ashmem) subsystem [LWN.net]. https://lwn.net/Articles/452035/.

[12] 2019. LKML: Dianne Hackborn: Re: [PATCH 1/6] staging: android: binder: Remove some funny usage. https://lkml.org/lkml/2009/6/25/3.

[13] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 2016. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA.

[14] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don'T Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA.

[15] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA.

[16] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles.*

[17] Brian N Bershad, Thomas E Anderson, Edward D Lazowska, and Henry M Levy. 1990. Lightweight remote procedure call. *ACM Transactions on Computer Systems (TOCS)* (1990).

[18] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* (2011).

[19] Nicholas P Carter, Stephen W Keckler, and William J Dally. 1994. Hardware support for fast capability-based addressing. In *ACM SIGPLAN Notices*. ACM.

[20] Jeffrey S Chase, Henry M Levy, Michael J Feeley, and Edward D Lazowska. 1994. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems (TOCS)* 12, 4 (1994).

[21] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*.

[22] Raymond K Clark, E Douglas Jensen, and Franklin D Reynolds. 1992. An architectural overview of the Alpha real-time distributed kernel. In *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*.

[23] Francis M David, Ellick Chan, Jeffrey C Carlyle, and Roy H Campbell. 2008. CuriOS: Improving Reliability through Operating System Structure.. In *OSDI*.

[24] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. 2013. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM.

[25] Peter Druschel and Larry L Peterson. 1993. Fbufs: A high-bandwidth cross-domain transfer facility. *ACM SIGOPS Operating Systems Review* (1993).

[26] Kevin Elphinstone and Gernot Heiser. 2013. From L3 to seL4 what have we learnt in 20 years of L4 microkernels?. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*.

[27] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, New York, NY, USA.

[28] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. 1996. Microkernels meet recursive virtual machines. In *OSDI*.

[29] Bryan Ford and Jay Lepreau. 1994. Evolving Mach 3.0 to A Migrating Thread Model. In *USENIX Winter*.

[30] Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. 1999. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *OSDI*, Vol. 99. 87–100.

[31] Jayneel Gandhi, Mark D Hill, and Michael M Swift. 2016. Agile paging: exceeding the best of nested and shadow paging. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE.

[32] Per Brinch Hansen. 1970. The nucleus of a multiprogramming system. *Commun. ACM* (1970).

[33] Swapnil Haria, Mark D Hill, and Michael M Swift. 2018. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM.

[34] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. 1997. The performance of μ-kernel-based systems. In *ACM SIGOPS Operating Systems Review*, Vol. 31. ACM.

[35] Gernot Heiser. 2008. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*.

[36] Dan Hildebrand. 1992. An Architectural Overview of QNX.. In *USENIX Workshop on Microkernels and Other Kernel Architectures*.

[37] Galen C Hunt and James R Larus. 2007. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review* (2007).

[38] Vasileios Karakostas, Jayneel Gandhi, et al. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA.

[39] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*.

[40] Eric J Koldinger, Jeffrey S Chase, and Susan J Eggers. 1992. *Architecture support for single address space operating systems*. Vol. 27. ACM.

[41] Sanghoon Lee, Devesh Tiwari, Yan Solihin, and James Tuck. 2011. HAQu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*.

[42] Roy Levin, Ellis Cohen, William Corwin, Fred Pollack, and W Wulf. 1975. Policy/mechanism separation in Hydra. In *ACM SIGOPS Operating Systems Review*.

[43] Henry M Levy. 1984. *Capability-based computer systems*. Digital Press.

[44] Wenhao Li, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2015. Reducing World Switches in Virtualized Environment with Flexible Cross-world Calls. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[45] Jochen Liedtke. 1993. Improving IPC by kernel design. *ACM SIGOPS operating systems review* (1993).

[46] Jochen Liedtke. 1993. A persistent system in real use-experiences of the first 13 years. In *Object Orientation in Operating Systems, 1993., Proceedings of the Third International Workshop on*. IEEE.

[47] Jochen Liedtke. 1995. *On micro-kernel construction*. Vol. 29. ACM.

[48] Jochen Liedtke, Kevin Elphinstone, Sebastian Schonberg, Hermann Hartig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. 1997. Achieved IPC performance (still the foundation for extensibility). In *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*. IEEE.

[49] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. 2018. DAMN: Overhead-Free IOMMU Protection for Networking. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM.

[50] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM.

[51] Changwoo Min, Woonhak Kang, Mohan Kumar, Sanidhya Kashyap, Steffen Maass, Heeseung Jo, and Taesoo Kim. 2018. Solros: a data-centric operating system architecture for heterogeneous computing. In *Proceedings of the Thirteenth EuroSys Conference*. ACM.

[52] Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. 2016. Efficient Synonym Filtering and Scalable Delayed Translation for Hybrid Virtual Caching. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA.

[53] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE.

[54] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. 2017. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA.

[55] Jerome H Saltzer. 1974. Protection and the control of information sharing in Multics. *Commun. ACM* 17, 7 (1974), 388–402.

[56] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, Todd C Mowry, and Trishul Chilimbi. 2016. Page overlays: An enhanced virtual memory framework to enable fine-grained memory management. *ACM SIGARCH Computer Architecture News* (2016).

[57] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed {OS} for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation OSDI 18)*.

[58] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. 1999. *EROS: a fast capability system*. Vol. 33. ACM.

[59] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association.

[60] Udo Steinberg and Bernhard Kauer. 2010. NOVA: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*.

[61] Dan Tsafrir. 2007. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Proceedings of the 2007 workshop on Experimental computer science*. ACM.

[62] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: Protecting software with code-centric memory domains. In *ACM SIGARCH Computer Architecture News*. IEEE Press.

[63] Lluís Vilanova, Marc Jordà, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2017. Direct Inter-Process Communication (dIPC): Repurposing the CODOMs Architecture to Accelerate IPC. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM.

[64] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. 2014. *The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0.* Technical Report. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES.

[65] Robert NM Watson, Ben Laurie, et al. 2015. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE.

[66] Robert NM Watson, Robert M Norton, Jonathan Woodruff, Simon W Moore, Peter G Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, et al. 2016. Fast protection-domain crossing in the cheri capability-system architecture. *IEEE Micro* (2016).

[67] Emmett Witchel, Josh Cates, and Krste Asanović. 2002. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM, New York, NY, USA.

[68] Emmett Witchel, Junghwan Rhee, and Krste Asanović. 2005. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. ACM.