Boosting Inter-Process Communication with Architectural Support*

YUBIN XIA, DONG DU, ZHICHAO HUA, BINYU ZANG, HAIBO CHEN, HAIBING GUAN,
Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, China,
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, China, and
Shanghai AI Laboratory, China

IPC (inter-process communication) is a critical mechanism for modern OSes, including not only microkernels like seL4, QNX and Fuchsia where system functionalities are deployed in user-level processes, but also monolithic kernels like Android where apps frequently communicate with plenty of user-level services. However, existing IPC mechanisms still suffer from long latency. Previous software optimizations of IPC usually cannot bypass the kernel which is responsible for domain switching and message copying/remapping across different address spaces; hardware solutions like tagged memory or capability replace page tables for isolation, but usually require non-trivial modification to existing software stack to adapt to the new hardware primitives. In this paper, we propose a hardware-assisted OS primitive, *XPC* (Cross Process Call), for efficient and secure synchronous IPC. XPC enables direct switch between IPC caller and callee without trapping into the kernel, and supports secure message passing across multiple processes without copying. We have implemented a prototype of XPC based on the ARM AArch64 with Gem5 simulator and RISC-V architecture with FPGA boards. The evaluation shows that XPC can reduce IPC call latency from 664 to 21 cycles, 14x–123x improvement on Android Binder (ARM), and improve the performance of real-world applications on microkernels by 1.6x on Sqlite3.

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

Additional Key Words and Phrases: operating system, microkernel, inter-process communication, hardware-software co-design

ACM Reference Format:
Yubin Xia, Dong Du, Zhichao Hua, Binyu Zang, Haibo Chen, Haibing Guan. 2022. Boosting Inter-Process Communication with Architectural Support. *J. ACM* 37, 4, Article 111 (October 2022), 35 pages. https://doi.org/10.1145/1122445.1122456

*This article extends a prior conference version that appeared in the proceedings of the 46th ACM/IEEE International Symposium on Computer Architecture (ISCA'19) by proposing new designs to support the XPC hardware extensions in ARM architectures, implementing the system on Gem5 and Linux, and evaluating new test cases.

Author's address: Yubin Xia, Dong Du, Zhichao Hua, Binyu Zang, Haibo Chen, Haibing Guan,
Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, Shanghai, China ,
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China ,
Shanghai AI Laboratory, Shanghai, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
0004-5411/2022/10-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

IPC (Inter-Process Communication) is a critical mechanism of operating system, which enables processes to send/receive messages to each other for data sharing, service invocation, etc. IPC is essential for microkernels [30, 40, 54] , which move most of the functionalities from privileged-mode to isolated user-mode domains that cooperate with each other through IPC. IPC is also important for monolithic systems with many user-level services, like Binder in Android [10]. For example, Android deploys multiple system services as user-level processes. Applications frequently communicate with these services through IPC, e.g., for drawing a component on the surface through a window manager. The efficiency of IPC directly affect the performance of applications as well as user experiences.

Unfortunately, existing IPC mechanisms still cause long latency, in either microkernels or monolithic kernels. For example, on Intel SkyLake processor, seL4 microkernel [34] spends about 468 cycles [6] for a one-way IPC on its fast path[1] (687 cycles when enabling Spectre/Meltdown mitigations). For Google Fuchsia's microkernel (called Zircon) [2] and Linux, one IPC may take tens of thousands of cycles.

Most of the cycles of an IPC are spent on two tasks: 1) domain switching, and 2) message copying. Domain switching is usually done by the kernel, which includes context saving/restoring, capability checking, and other IPC-related logics.

One way to reduce the cost of message copying is to pass the message through shared memory. However, such a method may lead to TOCTTOU (Time-Of-Check-To-Time-Of-Use) attack, i.e., the callee may check the state of a resource before using it, but the state can be changed by a malicious caller between the check and the use that violates the results of checking [11]. TOCTTOU can cause the victim callee to perform invalid actions because of the unexpected state [44]. Adopting page remapping for ownership transfer can mitigate the above security problem, but the remapping operation further increases latency and may also lead to costly TLB shootdown due to page table modification.

Previous work proposed various ways to optimize IPC performance, by either software [21, 31, 32, 39] or hardware [37, 38, 46, 51, 56, 59, 60]. For most software solutions, the overhead of trapping to kernel is inevitable, and message passing will lead to either multiple data copying or TLB shootdown. Some hardware solutions, like CODOMs [55], 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 modifications of existing kernel implementations which are designed for hosting multiple address spaces.

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-assisted IPC 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

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

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 addresses of a message. This mechanism supports memory ownership transfer by ensuring only one owner of the message at any time, which can prevent the TOCTTOU attack and requires no additional TLB flush. If process A invokes B (through IPC), and B invokes C, we call it an *invokcing chain*. A *relay-seg* can also be passed through an invoking chain to further reduce the time of memory copying.

XPC chooses to keep the semantic of synchronous IPC. XPC overcomes two limitations of traditional synchronous IPC [29], one is the relatively low data transfer throughput and the other is its not-easy-to-use model for multi-threaded applications. XPC improves throughput with the *relay-seg* mechanism, and provides easy-to-use programming interfaces with the *migrating thread* [31] and thread pool model. Besides, we only focus on intra-host IPC in this paper. Cross-host IPC (aka., Remote Procedure Call, RPC) is not considered in this paper.

We have implemented a prototype of XPC based on Gem5 simulator (a hardware simulator encompassing system-level architecture and processor microarchitecture) [23] and Rocket RISC-V core on FPGA board for evaluation. We ported two microkernel implementations (seL4 and Zircon) and one monolithic kernel implementation (Linux with Android Binder [10]), then measured the performance of both micro-benchmarks and real-world applications. The results show that XPC can reduce the latency of IPC by 5x-141x for existing microkernels, 14x–123x improvement on Android Binder (ARM), and the performance of applications like SQLite can be improved by up to 12x (from 1.6x). The overall hardware costs are small (1.99% in FPGA 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 as well as Gem5, integrated with two microkernels and Android Binder IPC.
- An evaluation with both microbenchmarks and real workloads on real platforms.

2 MOTIVATION

The semantic of a synchronous IPC is like a cross process function call: the caller thread blocks until the callee thread returns. The steps during a synchronous IPC can be grouped into two tasks: **domain switch** and **data transfer**. Typically, a domain switch requires trapping to the kernel which includes mode switch, context switch, states save/restore, and IPC logic (including scheduling). Data (message) transfer is used for passing arguments and returning values between the caller and callee. While short messages can be passed through registers, for long messages, memory copying is the most commonly used method.

We start by analyzing the IPC performance of state-of-the-art OSes (i.e., seL4 [34] and Android) and then present a detailed explanation of IPC. Our work is motivated by the performance analysis.

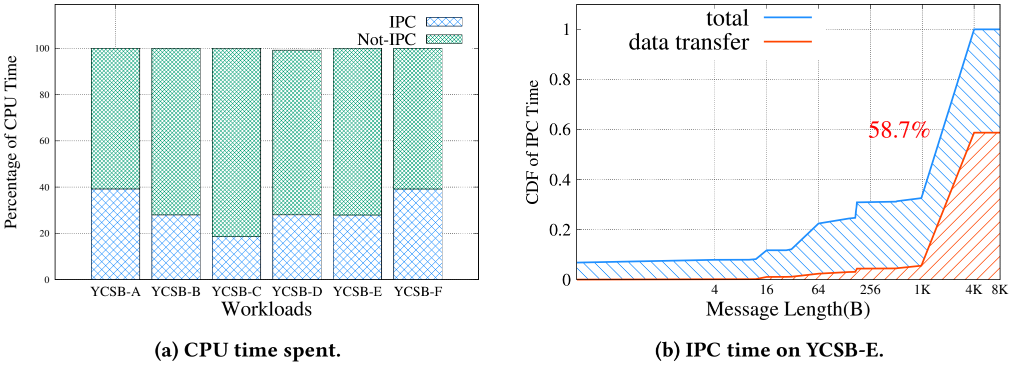
2.1 IPC Performance is Still Critical

We took YCSB benchmark workloads and ran Sqlite3 on seL4 on a SiFive U500 RISC-V FPGA board [14] (more setup in §8). 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 time of data transfer

Table 1. One-way IPC latency of seL4. seL4 (4KB) 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**

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**.



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

Fig. 1. (a): For Sqlite3 with YCSB workload, around 18% to 39% of the time is spent on IPC. **(b)**: Distribution of IPC time (during YCSB-E test). “total“ means the CDF of the total IPC costs related to different message sizes. “data transfer” means the ratio of message transfer costs and total IPC costs related to different message sizes.

2.2 Microkernel IPC Analysis

In this section, we break down the process of IPC in microkernel, measure the cost of each step, and analyze where the time goes. This quantitative analysis is also done using seL4 microkernel with U500 board.

There are two paths of IPC in seL4: a “fast path” and a “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. The kernel then saves the caller’s context and switches to its 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 is nearly half the time of domain switch.

In existing systems, the kernel will always save and restore all the context during switching for isolation. The underlying assumption is that the caller and callee do not trust each other. However, in certain cases, the caller and callee may have different trust assumptions, e.g., when they can trust each other, they only need to save and restore few registers like the traditional function call (according to the calling convention). Thus, existing kernel-involved IPC misses the optimization opportunities.

IPC Logic: In the IPC logic part, the major task is checking the validness of the IPC call. 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 any of the following conditions are met; if so, the slow path will be taken:

- 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 (32B) and less than 120B (IPC buffer size).

The IPC logic takes about 200 cycles.

We find that it is possible to separate the checking 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. Finally, the kernel transfers the IPC messages (only for messages $\leq$ 32B) and switches to the callee's address space. The process switch phase occupies about 150-200 cycles.

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 (> the space of available registers but $\leq$ IPC buffer), seL4 will turn to slowpath to copy the message (in our experiment, an IPC with 64B message takes 2182 cycles). For transferring long messages, seL4 uses shared memory in user space to avoid data copying (e.g., 4010 cycles for copying 4KB data in Table 1).

However, transferring message through shared memory may bring security issues. For example, a multi-threaded caller can observe the callee's operations performed on the shared memory, or even affect the callee's behavior by modifying the shared memory while the callee is processing. Further, in most existing implementations, the message still needs to be copied to the shared memory at first. The message transfer dominates the IPC cycles when the message size is large.

2.3 Monolithic Kernel IPC Analysis

Low IPC latency becomes significant for modern operating systems. However, a single-minded pursuit of performance-oriented design is insufficient because today's IPC is feature-rich especially on monolithic kernel-based OSes. An IPC design should be capable of supporting the necessary features for better usability.

This section takes Android as an example to analyze the IPC facilities provided by a monolithic kernel (Linux), which forms the requirements for our design.

Binder IPC: Binder IPC is an extension introduced by Android to Linux for inter-process communication. It has been widely used in Android components, including window manager, activity manager, input method manager, etc [10]. Binder IPC uses *Binder transaction*, which is a set of

Table 2. Marshaling/Unmarshaling with Binder Parcel. The marshaling and unmarshaling methods for basic data types (i.e., Integer, Float, and String) are in Binder's parcel library, e.g., *writeFloat* and *readFloat*. The results are latencies transferring 1,000 objects. The marshaling for the linked list is based on the basic data types, and the results are transferring one list with 1,000 entries. It is evaluated in gem5 (setting in §8.1), and the results are represented using ticks.

	Integer	Float	String	Linked list
Marshaling (kticks)	34,524	34,958	100,746	44,536
Unmarshaling (kticks)	24,766	29,016	10,766	112,648

protocols and commands to establish an IPC channel, configure IPC, and process IPC operations. More details will be introduced in §8.5. Binder IPC relies on the kernel to copy the transfer data from the caller to the callee processes (i.e., two copying). Besides, Android combines Binder with *ashmem* (anonymous shared memory [9]), which allows a process to share a memory region with another process by sharing a file descriptor (i.e., zero-copying), to boost the performance of bulk memory transfer between processes.

Marshalling: The transferred data is named *parcel* in Binder IPC. Binder provides a library to read/write objects with basic data types like integer, float, boolean, or string from/to a parcel. Binder also allows processes to implement customized data structures, e.g., linked list, which must implement the *Parcelable* interface. A data structure implementing the interface must provide methods to serialize an object on the caller and deserialize it on callee. It is required to break the high-level structures into basic data types for serialization and deserialization.

The procedure of building a parcel from objects (with different data types) is called *marshalling*, and the procedure of rebuilding objects from a parcel is called *unmarshalling*. To reveal marshalling's performance implications, we present an evaluation using Binder's parcel, as shown in Table 2. It shows that the costs for different data types, from the basic data types to high-level structures. The string's unmarshalling costs are less than others (i.e., Integer, Float and Linked List in Table 2) because the Binder will directly return a pointer pointing to the string without copying. The unmarshalling costs for the linked list are much more than marshalling costs because it needs to re-construct the whole list, including allocating memory. The analysis implies that marshalling and unmarshalling are costly and should be optimized. In this work, we use a hardware-software co-design to mitigate the costs of marshalling and unmarshalling (§5.4 and §6.2).

Concurrent Communication: Binder IPC allows multiple clients [2] to communicate with the same server simultaneously. It leverages the thread pool model to achieve the goal. The server process should register a set of working threads (called *Looper*) into a thread pool managed by the OS. The working threads are blocked in the kernel. When a communication request arrives, the kernel will select an idle working thread, pass messages to it, and wake up the thread so it can handle the requests. Supporting concurrent communication brings challenges to a hardware-assisted IPC mechanism. We will elaborate on the challenge, and how XPC (this work) overcomes it in §5.2 and §6.2.

2.4 Goals

Based on the analysis, we summarize three goals: first, a fast IPC that does not depend on the kernel (**Goal-1**) is necessary but still missing. Second, a secure and zero-copying mechanism for passing messages while supporting handover (**Goal-2**) is critical to performance. Third, an IPC

[2]In the paper, we also use "client/server" to represent "caller/callee".

Table 3. 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 TO-CTTOU**	**Hand over**	**Granu-larity**	**Copy time**
Baseline	Mach-3.0	Multi	Kernel schedule	✗	✗	Kernel copy	✓	✗	Byte	2*N
Software opt.	LRPC [21]	Multi	Protected proc call	✗	✓	A-stack copy	✓	✗	Byte	2*N
	Mach (94) [31]	Multi	Migrating thread	✗	✓	Kernel copy	✓	✗	Byte	N
	Tornado [32]	Multi	Protected proc call	✗	✓	Remapping	✓	✗	Page	0+Δ
	L4 [39]	Multi	Direct proc switch	✗	✓	Temporary mapping	✓	✗	Byte	N
Hardware opt.	CrossOver [38]	Multi	Direct EPT switch	✓	✓	Shared mem.	✗	✗	Page	N-1
	SkyBridge [46]	Multi	Direct EPT switch	✓	✓	Shared mem.	✗	✗	Page	N-1
	Opal [25]	Single	Domain register	✓	✓	Shared mem.	✗	✗	Page	N-1
	CHERI [58]	Single	Function call	✓	✓	Memory cap.	✗	✓	Byte	0
	CODOMs [55, 56]	Single	Function call	✓	✓	Cap reg + perm list	✗	✓	Byte	0
	DTU [19]	Multi	Explicit	✓	✓	DMA-style data copy	✓	✗	Byte	2*N
	Mondrian [60]	Multi	Call gate	✗	✓	Mapping + grant perm	✗	✗	Byte	0+Δ
	XPC	Multi	Cross process call	✓	✓	Relay segment	✓	✓	Byte	0

design should support concurrent communication and transfer messages efficiently (**Goal-3**). Our design is based on these three goals.

3 RELATED WORK

There is a long line of research on reducing the latency of domain switch as well as long message transfer for IPC optimization. This section aims to answer the following questions:

- Why software-only approaches cannot achieve the proposed goals, and what are the missing parts in hardware?
- Why state-of-the-art hardware-assisted approaches can not achieve the goals?
- Why existing hardware "direct switching" mechanisms like VMFUNC and MPK can not achieve the goals?

We will focus on the first two goals and leave the third one in §5.2. We classify related approaches into two categories: optimizations on domain switch and optimizations on message passing, as shown in Table 3.

3.1 Optimizations on Domain Switch

Software-based Optimizations. One widely adopted optimization is to use a caller's thread to run callee's code in callee's address space, as in PPC (protected procedure call) [21, 32] and migrating thread model [27, 31]. This optimization eliminates the scheduling latency and mitigates IPC logic overhead, and has been used in LRPC [21] and the new version of Mach [31]. Tornado [32] also adopts PPC as its execution model. Besides, it leverages another feature of PPC, "fine data locality", to mitigate the data cache miss penalty caused by domain switching. L4 [33, 39, 41, 42] 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.

Limitations of Software-based Optimizations. As shown in Table 3, two key requirements to achieve **Goal-1** for a domain switch are no trapping into the kernel and no scheduling. Although software-only approaches can mitigate the scheduling costs using a carefully designed IPC routine,

they can not achieve the "without trapping" requirement as they still rely on the OS kernel to perform context switchings from caller to callee.

Barrelfish [20], as a state-of-the-art kernel design based on the multikernel model, utilizes URPC [22] for cross-core communication. URPC is an asynchronous IPC design and can achieve both requirements by polling on user-space shared memory. It can achieve significant low latency even on complicated settings, e.g., only 618 cycles (with two-hop) on an 8x4-core AMD.[3] However, URPC brings few benefits if both the caller and callee are on the same core. Besides, it usually requires communicating entities to poll on well-structured messages (usually using a cache line) to gain the best performance, which is too restrictive for scenarios without redundant CPU cores, like mobile devices, cars, etc. Although modern OSes tend to utilize both asynchronous IPC and synchronous IPC for different scenarios, in this paper, we only focus on optimizing the performance of synchronous IPC, which usually has a stricter latency requirement.

Hardware-based Optimizations. New hardware extensions are also proposed to improve the performance of cross domain calls, like Opal [25], CHERI [58, 59] CODOMs [55], and Mondrian memory protection (MMP) [60, 61]. In these extensions, a *domain* is a new abstraction of execution subject (i.e., a piece of code) that has its own identity (i.e., domain ID), instead of a traditional process isolated by address space. For example, CODOM [55] provides efficient support for protecting multiple software components (domains) inside a single address space. It associates a tag for every page and a list of tags for code pages, representing the data/code they can access/invoke. Therefore, a domain includes a set of code pages as well as the data and privileged instructions it can use. CHERI [58, 59] implements a capability system on hardware to provide fine-grained isolation in a single address. The capability system isolates different domains. The domain switching of CODOMs and CHERI can be done directly at an unprivileged level without trapping to the kernel, which is a huge advantage against software optimizations.

Limitations of Hardware-based Optimizations. A distinct flaw of CODOMs and CHERI is that they only optimize **intra-process** communication, i.e., requiring all the communicating components to locate in a single address space. Most of the existing applications use the multi-address space model. It is hard for them in a single address space to use the traditional OS primitives like the fork, in which the parent and child processes can use the same address. Besides, domain-based isolation usually incurs many hardware changes. For example, CODOM needs to check the permission for each memory operation (dTLB, new capability registers, access protection lists, and others) and switch the capability for calls/rets (similar but much more complicated than xcall/xret). Moreover, these systems usually require non-trivial changes to existing OS kernels designed to multiple address space scenarios. Systems like CHERI adopt a hybrid approach using both capability and address space to achieve better compatibility, but switching between address spaces still requires kernel involvement.

MMP [60, 61] achieves efficient domain switching based on protection domain and call gate, and can retain the address space-based isolation. However, it still requires a privileged domain for permission checking (i.e., trapping into the privileged domain) and therefore can not achieve **Goal-1**.

Limitations of Hardware-assisted Direct Switching Mechanisms. CrossOver [38] and Sky-Bridge [46] 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, VM-FUNC is not designed for inter-process communication and has several limitations. First, VMFUNC only switches the EPT but leaves all other privilege states (e.g., page table pointers) unchanged. This

[3] "8x4-core" means 8 CPUs interconnected, each with 4 cores; each connection is one hop.

leads to a complex trampoline to handle all the states [46, 49]. Besides, VMFUNC does not include a capability-based checking mechanism — any user process can switch to any valid EPTs. Prior works utilize binary scanning to avoid potential malicious instructions. Last, VMFUNC requires support from the hypervisor, which is extremely hard when the system is running in the cloud.

User-level Interrupts. Intel released its prototype of an optimized IPC design based on user-level interrupts (i.e., UIPI [15]), which will bypass the kernel and the scheduler during communication. UIPI-based IPC design can also achieve **Goal-1**, but with two fundamental limitations. First, UIPI is only useful for cross-core communication — a caller and callee in the same core can not communicate using UIPI. Besides, a caller process can only notify a *running* callee process; otherwise, the kernel will record the events, and the callee can only handle the call when scheduled again.

3.2 Optimizations on Message Passing

As shown in Table 3, four key requirements to achieve **Goal-2** are (1) no TOCTTOU (Time-Of-Check-To-Time-Of-Use) vulnerability, (2) supporting long messages handover along the calling chain (without additional copyings), (3) fine-grained message granularity, and (4) few copying times.

Software-based Optimizations. For long message passing, one simple, secure but not efficient method is to adopt "twofold copy" (caller → kernel → callee), as shown in Figure 2(a). Some systems, e.g., LRPC [21], 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 2(b). However, a malicious caller may change the message at any time when the callee is running, which leads to a TOCTTOU attack. One solution is to change the ownership of shared memory by remapping (Figure 2(c)). However, memory remapping requires kernel's involvement and causes TLB shootdown. Further, 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, i.e., cannot support *handover*.

L4 [39] 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). However, the caller still requires the kernel to do the copying and remapping which will cause non-negligible overhead.

Hardware-based Optimizations. Many hardware-assisted systems [24, 55, 56, 58, 59] leverage capability for efficient message transfer among domains. CODOMs [55] uses a hybrid memory granting mechanism combined with a permission list (domain granularity) and capability registers (byte granularity). By passing a capability register, a memory region 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 attacks. CHERI [58] 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. Intel MPK [50] allows a process to classify its pages into 16 domains (0 is used by default); each domain can have its own read/write permissions through the (per-thread) PKRU register. Based on MPK, a process can assign different domains to different threads and allow zero-copying data transfer among threads by updating the PKRU register. However, MPK can only be used for intra-process communication.

Limitations: *A fundamental limitation of capability or domain-based approaches is that they are designed for single address space, i.e., only improving intra-process communication.*

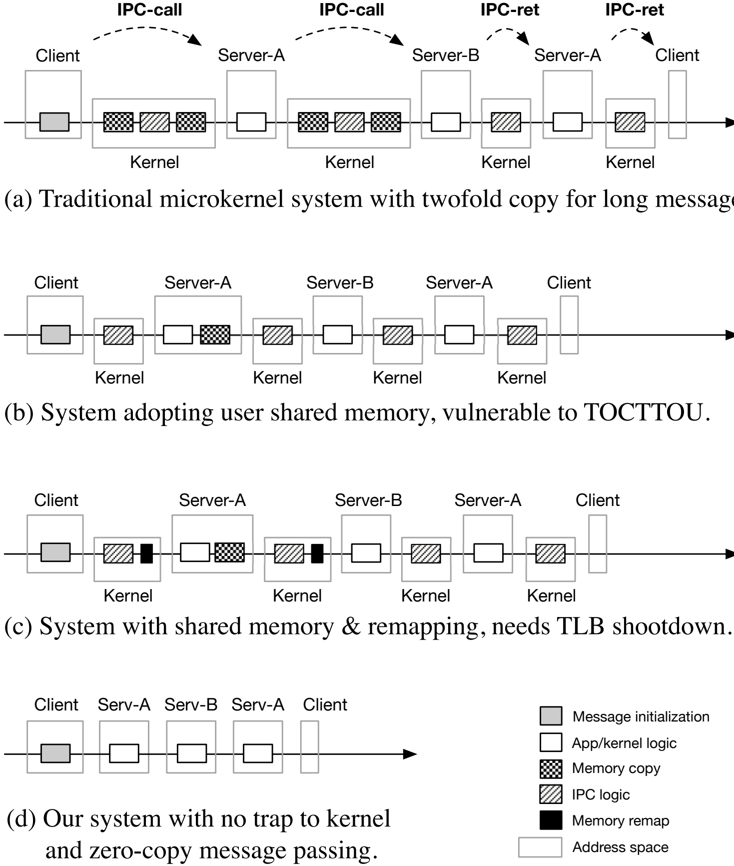


Fig. 2. Mechanisms for long message passing.

Opal [35] and Mondrian [60, 61] propose new hardware designs 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 attacks nor support long messages handover along the calling chain.

Limitations: *A fundamental limitation of PLB-based approaches is that they still suffer from TOCTTOU attacks and can not support message handover efficiently.*

M3 [19] leverages a new hardware component, DTU (data transfer unit), for message transfer. However, DTU is not suitable for small and medium-size data [47], since the overhead of channel initialization cannot be well amortized. HAQu [36] leverages queue-based hardware accelerators to optimize cross-core communication, while XPC can support a more general format of the message.

Limitations: *A fundamental limitation of DMA or Queue-based approaches is that they can only optimize the cross-core (or cross-PU) communication.*

4 DESIGN OVERVIEW

This paper proposes XPC, a hardware-software co-design IPC solution. XPC consists of two extensions on hardware: XPC engine and relay segment. XPC engine provides two new instructions,

"*xcall #reg*" and "*xret*", for IPC call and return respectively, while relay segment is a hardware design used for zero-copying message transfer. The paper also explores challenges to implementing the hardware extensions on full-featured architectures (e.g., ARM). and proposes techniques to overcome them, e.g., a decoupled version of *xcall* and *xret*. Besides hardware extensions, XPC also consists of two software components: the OS kernel acting as the control plane and manages XPC hardware, and a user-level library that can ease applications to use XPC.

The hardware and software design will be introduced in §5 and §6. This section presents the high-level insights on how XPC can achieve fast IPC.

User-level Cross Process Call. To achieve fast domain switching, XPC decouples the domain switching into the control and data planes. The OS kernel is responsible for control-plane tasks. For example, for a specific connection (e.g., *Client-A* and *Server-B*), the kernel will perform the IPC logics offline, including checking capabilities, affinity, priority, and any other OS-specific tasks. If the check is passed, the kernel can allow *Client-A* to directly invoke *Server-B* without trapping into the kernel by configuring XPC engine. The "direct invoke" is achieved through XPC engine, which will be introduced in §5. Since the control plane checking is performed offline, the runtime costs of a domain switching are only the costs of "direct invoke", which eliminates the trap and restore costs, optimizes the IPC logic costs by only checking whether the kernel allows the invocation (much simpler), and avoids other costs like scheduling queue modification.

Zero-copying Message Transfer. XPC relies on *relay-seg* (short for "relay segment") for secure and zero-copying message transfer. The main idea is that, instead of transferring data, XPC transfers a mapping from the caller to the callee. A similar design is page remapping, that the kernel unmaps a set of pages from the caller process and then maps these pages to the callee process. XPC overcomes two challenges in page remapping. First, remapping requires TLB shootdowns due to page table modification, which may incur long latency [17]. Second, remapping requires the message size to be aligned with page size (e.g., 4KB), which may waste memory and cause fragmentation.

The challenges are overcome using *relay-seg*. A *relay-seg* is a memory region with its virtual address ranges and physical address ranges, i.e., the virtual address ranges are backed with continuous physical memory. The address range information is recorded in a new register, *seg-reg*, and an extension in MMU is responsible for translating the virtual addresses into physical addresses using *seg-reg*. The *seg-reg* can be passed from a caller to a callee through XPC engine; thus, the callee can directly access the data within the virtual address range indicated in its *seg-reg*. In our prototype, the OS kernel is responsible for assigning the same virtual address region for all *relay-seg*s and ensuring that the mapped region of a *relay-seg* will never be overlapped by any mapping of the page table. Thus, no TLB shootdown is needed on this region.

5 XPC HARDWARE DESIGN

5.1 XPC Engine

This section introduces how XPC engine achieves "direct invoke" in userspace. Specifically, XPC engine introduces several new abstractions, i.e., *x-entry*, *xcall-cap* and *link-stack*, and two new instructions, i.e., *xcall* and *xret*, for the goals. We introduce the abstractions first and then explain how they are used together in the two instructions. The hardware extensions are summarized in Figure 3. For now, we ignore the credit-related extensions (e.g., credit-base-reg, credit table, and credit index) and will introduce them in §5.2.

x-entry. An *x-entry* is bound with a procedure that can be invoked by other processes. A 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 register *x-entry-table-reg*. *x-entry-table* is system-wide global

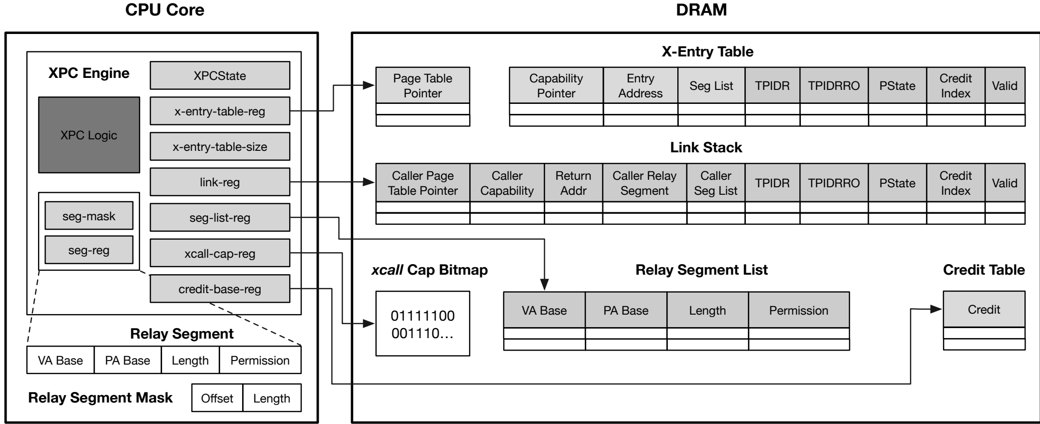


Fig. 3. XPC Engine: *x-entry-table* holds all *x-entries*, each of which represents an XPC procedure. *xcall* capability bitmap indicates which *x-entries* can be invoked by current thread. A *link stack* is used to store the linkage record. A *credit table* is used to store credits for *x-entries*. Two new instructions *xcall* and *xret* perform the call and return operations, handled by XPC Logic. *XPCState* maintains the execution states of decoupled *xcall* and *xret* instructions. *seg-reg*, *seg-mask* and *seg-list-reg* provide a new address mapping method called relay segment to transfer message.

and shared by all processes. 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 adjustable.

An *x-entry* contains fields describing a remote procedure, as shown in Figure 3. In our current design, it includes *page table pointer* of remote procedure, *entry address* (i.e., the program counter of the handler function in the page table), *seg-list-reg*, *credit index* and a valid bit. The *credit index* and *seg-list-reg* will be introduced in §5.4 and §5.2 individually. For ARM, an *x-entry* also include three ARM-specific states, *TPIDR*, *TPIDRRO*, and *PState*, which are per-thread registers/states for ARM. With all the information, XPC can switch from a caller process to a callee process without kernel involvement.

A caller needs an *xcall-cap* (short for "XPC call capability") to invoke an *x-entry*.

xcall-cap. *xcall-cap* records the XPC invocation capability of each *x-entry*. We use a bitmap to represent *xcall-cap*, as shown in Figure 3. Each bit with index i represents whether the thread[4] 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* (also *per-thread*), which is maintained by the kernel and used by the hardware.

An *xcall-cap* will be checked during an *xcall*. Therefore, the kernel can set a bit on the bitmap to allow a specific client to call a specific server directly if the offline checking passes; otherwise, the kernel can reject the direct call.

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 in the processor during this instruction: checking capability, fetching *x-entry*, pushing *linkage record*, and switching contexts.

(1) The XPC engine checks the caller's *xcall-cap* by reading the bit at #reg in the *xcall-cap* bitmap, which is per-thread bitmap and indicated by *xcall-cap-reg* (per-thread).

[4] As a process may contain multiple threads in most modern OSes, we use *thread* instead of *process* here for accuracy.

(2) The XPC engine loads the target *x-entry* from *x-entry-table* (a global table indicated by *x-entry-table-reg*) and checks the valid bit of the entry.
(3) The XPC engine prepares a *linkage record* to record the caller's information, and pushes it to the *link stack*. *Linkage record* and *Link stack* will be used for *xret*.
(4) The XPC engine update processor states according to the fetched *x-entry*, including page table pointer, program counters (set to the procedure's entrance address), and others.

The XPC engine does not switch other general registers, e.g., stack pointer, and leaves them to XPC library and applications to handle. The engine will put the value of the caller's *xcall-cap-reg* in a general-purpose register as an unforgeable badge, which can help a callee identify the caller as used in modern microkernels like seL4 [16]. Any exceptions that happen in the process will be reported to the kernel.

Linkage Record & Link Stack. Besides *x-entry* and *xcall-cap*, we still need an abstraction to save the caller's information, e.g., caller's page table and return address, which is necessary for IPC return. We use the term, *linkage record*, to represent the information. A *linkage record* maintains calling information that can only be accessed by the kernel, which is very similar to the fields in *x-entry*.

In our current design, a *linkage record* includes *page table pointer*, *return address*, *xcall-cap-reg*, *seg-list-reg*, *relay segment credit index*, and a valid bit. Besides, it can also include architecture-specfic registers when necessary, e.g., three ARM-specific states, *TPIDR*, *TPIDRRO*, and *PState* in Figure 3. The XPC engine does not save other general registers and leaves them to the XPC library and applications to handle. A *link stack* is used to store *linkage record*s, which is a per-thread memory region pointed by a register, *link-reg*. Both *link stack* and *link-reg* can only be accessed by the kernel and used by the hardware.

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 in the processor during this instruction: checking capability, fetching *x-entry*, pushing *linkage record*, and switching contexts.

(1) The XPC engine checks the caller's *xcall-cap* by reading the bit at #reg in the *xcall-cap* bitmap, which is per-thread bitmap and indicated by *xcall-cap-reg* (per-thread).
(2) The XPC engine loads the target *x-entry* from *x-entry-table* (a global table indicated by *x-entry-table-reg*) and checks the valid bit of the entry.
(3) The XPC engine prepares a *linkage record* to record the caller's information, and pushes it to the *link stack*. *Linkage record* and *Link stack* will be used for *xret*.
(4) The XPC engine update processor states according to the fetched *x-entry*, including page table pointer, program counters (set to the procedure's entrance address), and others.

The XPC engine does not switch other general registers, e.g., stack pointer, and leaves them to XPC library and applications to handle. The engine will put the value of caller's *xcall-cap-reg* in a register (e.g., *t0* in RISC-V) as an unforgeable badge, which can help a callee identify the caller as used in modern microkernels like seL4 [16]. 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 checks the valid bit of the popped record and restores the caller's context accordingly.

Optimizations. Two optimizations are proposed to reduce the latency of XPC instructions further. First, we add a dedicated cache to optimize memory accesses of the XPC engine when fetching *x-entry* and capability. This design is based on two observations: ① IPC has high temporal locality

(for a single thread); ② IPC is highly predictable and friendly to prefetching. Thus, we use a software-managed cache to store *x-entries*. Prefetching is supported so that an *x-entry* can be loaded to the cache in advance. As shown in §8.2, we can save 12 cycles by prefetching.

Besides, at the point of pushing a *linkage record*, XPC engine is ready to perform switching. Thus, we can lazily save the *linkage record* to *link stack* using a non-blocking approach to hide the latency of stack writing. As shown in §8.2, the optimization can save 16 cycles.

5.2 Concurrent Communication and Credit System

The design of *xcall* and *xret* can support concurrent communication, i.e., multiple caller processes can invoke *xcall* to communicate with the same callee process. However, we still lack a mechanism restricting the maximal concurrent callers.

Motivation. Without such a mechanism, supporting concurrent communication may impose a substantial burden on software. First, it requires all server threads to be able to handle multiple invocations. However, some applications, e.g., Android services that use Android Binder for IPC, assume each server thread to handle one request at a time.

Besides, it requires the kernel to handle concurrent traps from the different threads (with the same context), which complicates the OS implementation. For example, two caller threads may invoke the same callee thread simultaneously. As a result, there will be two running threads using the same context, including page tables, per-thread registers, and others, inherited from the callee thread. During the execution, the two threads may trap into the kernel at the same time, e.g., to handle a syscall, and the kernel will get confused by finding that two threads are trapped with the same context. This will usually lead to kernel panic without (usually complicated) kernel modification.

Another possible solution is to use the wrapper on the callee side to reject invocation when the last call is not finished. However, there is still a time window (although very short) that multiple callers are running using the callee's contexts after *xcall*. These callers are still possible trapping into the kernel (e.g., timer interrupt), which will require OS kernel modifications to handle the case.

In XPC, we introduce *Credit system* [3, 19], which allows the kernels to explicitly control the concurrent communication and limit the maximal concurrent callers. Therefore, OSes like Linux can enforce that one callee thread can only be invoked by one caller process at one time and rely on the thread pool to handle concurrent communication.

Credit system. XPC's credit system includes a new global table, *credit table*. Each entry of the table records a credit value, which indicates the allowed invocations. The *x-entry-table-size* manages the number of entries. We add a new field in each *x-entry*, *credit index*, which is an index in the *credit table*. As a result, the credit of an *x-entry* would be $credit_table[credit_index]$. The main reason to use *credit index* instead of *credit value* in *x-entry* is to allow sharing credits between different *x-entries*. For example, a server thread may register two services for clients but can only handle one request at a time. In this case, the server thread can ask the kernel to use the same credit index for the two services, which can control the maximum concurrent invocations for the two services. The initial value of credits is managed by the kernel.

During *xcall*, XPC engine will first check the credit. If the value is 0, the *xcall* returns an error code to the caller. Otherwise, it atomically decreases the credit in the credit table (i.e., $credit_table[credit_index] \ -= 1$). To ease the recovery of the credit, *xcall* will push the credit index in *linkage record*. *xret* will increase the credit according to the index information (i.e., $credit_table[credit_index] \mathrel{+}= 1$).

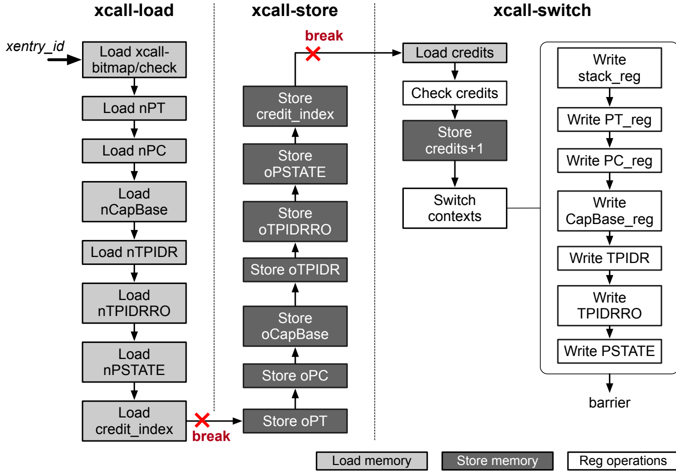


Fig. 4. XPC(xcall)+credits in ARM using μOPs. A single instruction with too many μOPs is hard to be implemented in ARM devices. Relay segment-related registers are omitted here.

5.3 Decoupled XPC Instructions

Motivation. To achieve better performance, *xcall* and *xret* are suggested to be implemented in the CPU core as two complete instructions. However, the quite complex logic in the two instructions may violate their practicality. For example, the implementation of ARM architecture uses μOP, which is the basic operations that form ISA-level instructions to implement complicated instructions. Although an ARM instruction can be composed of multiple μOPs, there is usually an upper bound of the number of μOPs because too many μOPs may cause several issues, e.g., the instruction would be easy to be interrupted during execution or affect the pipeline design. The concrete upper bound depends on the design and implementation of hardware.

An implementation of *xcall* using the μOP (in Gem5) is shown in Figure 4. Because of the complex logic in *xcall*, it needs at least 19 μOPs, which is a long path that is very hard to be implemented in real devices.

Instruction Decoupling. To solve the issue, this paper proposes a new mechanism, *instruction decoupling*, that decouples a complex instruction into multiple simple instructions, and the composition of these simple instructions has the same semantics as the complex instruction. The insight behind the design is that we can sacrifice some performance and usability for practicality.

For example, *xcall* is decoupled into three simpler instructions, *xcall-load*, *xcall-store* and *xcall-switch*, as shown in Figure 4. *xcall-load* will load the target process's states which are recorded in *x-entry*. *xcall-store* will store the current states into the link stack, while *xcall-switch* will finally check credits and switch to the new states. We add internal registers to record values prepared by a prior instruction but used in successive instructions, e.g., *xcall-load* saves the loaded credit index into *tmp-credit-index-reg* which is used by *xcall-store* to push *linkage record* into link stack. Therefore, a process can sequentially invoke the three instructions to achieve the same results as a single *xcall*. Similarly, *xret* is decoupled into two instructions: ① *xret-load* to load a linkage record into temporal registers; and ② *xret-switch* to switch the context to the caller.

Challenges. We need to solve two challenges to decouple *xcall* and *xret*. First, we need a generic way to split μOPs into different decoupled instructions that achieves both great performance and usability. Second, we should ensure the security of the decoupled instructions. As invocations of

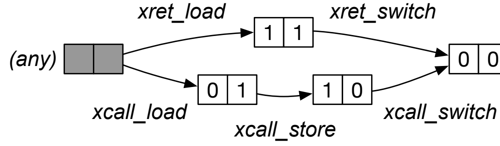


Fig. 5. State transition with XPCState.

decoupled instructions can be interrupted, some states dependent on the successive instructions may have been changed. For example, the capability is checked in *xcall-load*. Nevertheless, before *xcall-save* and *xcall-switch*, the capability may have been revoked. As the successive instructions do not recheck the value, they will still succeed and lead to errors. Moreover, a malicious process may randomly invoke the three instructions in the wrong order.

Techniques. We propose two techniques to resolve the above challenges. First, we decouple *xcall* and *xret* based on two rules: (1) each decoupled instruction should perform similar μOPs, and (2) each decoupled instruction should touch a few cache lines. As a result, the *xcall-load* in Figure 4 performs (mostly) memory loading, in which a single *x-entry* could be aligned in a single cache line (one cache miss at the worst case).

Second, XPCState is introduced to overcome the second challenge, as shown in Figure 5. It includes two bits[5] in XPC. Different values of XPCState represent the different states in the state machine shown in the figure. XPCState is changed by the kernel or the decoupled instructions. The figure shows the allowed transitions performed by the instructions. For example, *xcall-load* is allowed to perform on any state, and will transfer the XPCState into 0b01, while *xcall-store* is only allowed to perform on state 0b01, and transfer to 0b10. *xcall-switch* is only allowed to perform on state 0b10. These transitions enforce only the sequence: *xcall-load*→*xcall-store*→*xcall-switch* will succeed (for *xcall*). The XPCState is visible to the kernel, so the kernel can leverage it to manage the transition. For example, the kernel can turn a process's XPCState into 0b00 when it changes the process's capability. The simple strategy can avoid the prior case that the capability is changed during *xcall-load* and *xcall-store*.

Notably, the paper does not aim to provide a generic or optimal solution to implementing complex instructions. The decoupled instruction method proposed here is designed for supporting XPC on complicated hardware like ARM-based devices. However, we believe other systems can still learn from our approach to solve similar issues.

5.4 Relay Segment

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 address base and the length together represent a virtual memory region (i.e., from VA_BASE to VA_BASE + LEN). Similarly, the physical address base and the length together represent a physical memory region. A valid *seg-reg* means the virtual memory region is directly mapped to the physical memory region. Moreover, the permission indicates how a user can access the virtual memory region, the same as permission bits used in page table entries. During address translation, the *seg-reg* has higher priority over the page table and TLB.

seg-mask register. User applications cannot directly change the mapping of *seg-reg*. Therefore, to help applications dynamically change the range of *seg-reg*, we introduce a new register *seg-mask*,

[5] The size could be extended in more complicated cases.

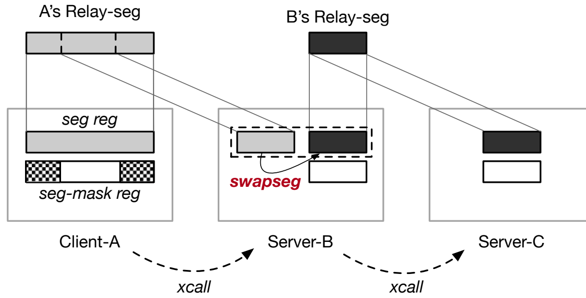


Fig. 6. Operations on *relay-seg*. (1) In Client-A, the process will only use the relay-seg memory masked by the *seg-mask* register; (2) During *xcall* from Client-A to Client-B, the masked relay-seg is transferred to the Client-B; (3) Client-B utilizes the *swap-seg* instruction to swap the *relay-seg* with an entry in *seg-list*; (4) During *xcall* from Client-B to Client-C, the new *relay-seg* is transferred to Client-C.

which can be used by user applications 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. Figure 6 shows the registers and operations of a *relay-seg*.

Multiple relay-segs. A server can create multiple *relay-seg*s, which will be stored in a per-process memory region called *seg-list* managed by kernel and pointed to by a new register, *seg-list-reg*. One process 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*.

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 other words, an active *relay-seg* can only be owned by one thread, and the ownership will be transferred along its calling chain, so that two CPU cores 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 check the current *seg-reg* with the *seg-reg* and *seg-mask* saved in the *linkage record*. An exception will be raised if the check fails, and the kernel will handle it.

6 XPC SOFTWARE DESIGN AND IMPLEMENTATION

XPC's software components include kernel support and an XPC library. The kernel is in charge of managing *x-entry*, *xcall-cap*, *relay-seg*, and exception handling. The XPC library is optional, which can ease the usage of the XPC mechanism.

6.1 OS Kernel Support

We have supported three OS kernels for XPC: two state-of-the-art microkernels (seL4 and Zircon [2] and Linux. Most modifications to the OS kernels are similar, and we will clarify the differences between microkernel systems and Linux when necessary.

System Calls. The kernels provides new syscalls to allow a user-level process to: register and unregister an *x-entry* (i.e., *xpc_register_x_entry* and *xpc_unregister_x_entry*), allocate and free a *relay segment* (i.e., *xpc_alloc_relay_seg* and *xpc_free_relay_seg*), grant and revoke the *xcall* capabilities to other threads (i.e., *xpc_grant_service* and *xpc_revoke_service*).

- *xpc_register_x_entry*: The user thread should pass the entry address, service name, and initial credits to the OS kernel. The kernel will construct an *x-entry* including provided information with *page table pointer*, *xcall-cap-reg* and other informations of the thread, and add the entry into the global *x-entry-table*. The return value is an index (representing the *x-entry* ID) on success, and error code on error, e.g., no space in *x-entry-table*.
- *xpc_unregister_x_entry*: The kernel will check whether the user thread is the owner of an *x-entry* and remove the entry from *x-entry-table*. Besides, the kernel will clear relative bits in all xcall bitmaps.
- *xpc_alloc_relay_seg*: The kernel will allocate a relay segment with indicated length. The mapping of the relay segment will be writen into the *seg-reg* if *seg-reg* is invalid now or into an empty entry of *seg-list* if *seg-reg* is used.
- *xpc_free_relay_seg*: The kernel will free a list of relay-segment in *seg-reg* or *seg-list*.

In the above implementations, we omit some checking details, e.g., the OS kernel should check that the provided entry address is not in kernel space.

Capability. Capabilities have been widely used to manage IPC [34, 52]. Our software implementation introduces two capabilities for each *x-entry*: *xcall-cap* and *grant-cap*. *xcall-cap* represents the capability to invoke an *x-entry* — a caller thread can only use *xcall* to invoke an *x-entry* (of calee) when it has the *x-entry*'s *xcall-cap*. *grant-cap* represents the capability to grant/revoke an *xcall-cap* or *grant-cap*.

The kernel will maintain a capability list for each thread. When a thread creates an *x-entry* (using *xpc_register_x_entry*), it will have both the *grant-cap* and *xcall-cap* of the new *x-entry*, and can grant the *xcall-cap* to other threads (i.e., *xpc_grant_service* and *xpc_revoke_service*).

In our prototype, the *xcall-cap* is implemented by the hardware *xcall* capability bitmap, and the *grant-cap* is implemented by the OS kernel to transfer capabilities.

XPC Contexts in Kernel. The OS kernels introduce a new context, named *xpc_context*, including XPC's per-thread registers, i.e., *xcall-cap-reg*, *seg-reg*, *seg-mask*, *seg-list* (can be per-thread or per-process), and *link-reg* (per-process). The *xpc_context* will be saved and restored during context switching.

Split Thread Model. A significant challenge to the kernel is that kernel-bypassed domain switching 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 into the kernel. If the kernel is not aware of the *xcall*, it will mistakenly use A's page table to handle B's page fault.

To solve this problem, we propose a generic model, *split thread model*, which is inspired by the idea of migrating thread [31] to 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. The OS kernel (or XPC's kernel module) will maintain the map in its memory. When a new thread is created or destroyed, the xcall-cap-reg and the xcall-bitmap will be allocated or freed, and the kernel module will update the map accordingly. As *xcall-cap-reg* is per-thread and will be updated by hardware during *xcall*, once a thread traps into the kernel, the kernel will use the value of *xcall-cap-reg* as an index to find the current runtime state through the map.

```
1 struct Node {
2   /* the data being stored in the node */
3   data;
4   /* reference to the next node, NULL for last node */
5   struct Node* next;
6   /* ... specific fields */
7 };
8 struct List {
9   /* first node of list; NULL for empty list */
10  struct Node* firstNode;
11 };
12
13 /* caller ensure list is in the relay segment */
14 void handle_list(struct List list) {
15   ► assert(node->next >= relay-seg.begin) ◄
16   ► assert(node->next+sizeof(node) <= relay-seg.end) ◄
17     struct Node * node =
18       xpclib_get_pointer(list->firstNode);
19   ► Loop_start ◄
20   while (node != NULL) {
21     /* do something with node.data */
22    ► assert(node->next >= relay-seg.begin) ◄
23    ► assert(node->next+sizeof(node) <= relay-seg.end) ◄
24    node = xpclib_get_pointer(node->next);
25    ► Loop_validate ◄
26   }
27 }
```

Fig. 7. Linked list protected by XPC library. The *xpclib_get_pointer* is provided to fetch pointers from the relay segment, which will validate the fetched pointer's boundary.

The idea of migrating thread needs to carefully decouple the thread state in the whole kernel, which is hard to be implemented in a monolithic kernel like Linux. Fortunately, most of the monolithic kernel relies on a set of specific interfaces to get thread-specific resources, e.g., Linux uses *current* and *current_thread_info()* to get the resources and handle exceptions and syscalls. Therefore, we can solve the issue by updating these interfaces: ① check whether the thread has been changed (using *xcall*) by comparing the *xcall-cap-reg* (per-thread); ② find the correct thread by *xcall-cap-reg* (OS maintains a map between the *xcall-cap-reg* to *task_struct*); and ③ return the correct thread to the kernel.

Implementation Efforts. Compared with prior systems [55, 58, 59], the OS implementation efforts for XPC are small for two reasons. First, XPC only affects IPC-related modules in OS kernels but will not affect other modules like memory and address space management, file system, networks, etc. Instead, prior hardware-assisted systems (e.g., CODOM) usually requires running multiple processes (or domains) in a single address space, which will break abstractions like *fork*.

Although we should modify the scheduler to save and restore XPC contexts, we do not need to modify existing logic but simply add our new routines, e.g., in Linux, the function to save and restore XPC contexts is wrapped by a new config (CONFIG_XPC) and can be easily disabled. In this way, our modifications are easy to be maintained along with the kernel development.

Besides, our experiences of supporting three different kernels have proven that most modifications are similar to different kernels and XPC is easy to integrate with existing OSes. A developer can follow previous patches to support XPC in a new OS.

6.2 User Library Implementation

We propose XPC library to ease the usage of the XPC mechanism. In our evaluation (§8), all applications, including the Android Binder optimized with XPC, utilize the library to use XPC instead of directly invoking syscalls and XPC instructions. Developers are also free to design and use their own libraries.

Concurrent Communication: The XPC library can support different concurrent communication methods.

First, XPC library allows multiple clients to invoke a single *x-entry* at the same time. When creating an *x-entry*, a server process should specify a maximal number of concurrency, e.g., CN. Then, XPC library will prepare CN invocation contexts, which include an execution stack (called C-Stack) and local data (e.g., TLS, states dedicated to the server thread) to support simultaneous IPC calls. Besides, the library will add a trampoline for each *x-entry*. The trampoline will select an idle invocation 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. The credit value of the *x-entry* is set to the number of concurrencies, i.e., CN.

Besides, XPC library also supports concurrent communication based on the thread pool. When creating an *x-entry*, the library will prepare a thread pool with worker threads registered in the kernel. The kernel will allocate different *x-entries* for different worker threads. When a client connects to a service, the kernel will pick an *x-entry* of the service to the client. After connecting, the client can invoke a server directly using *xcall* with the *x-entry* of the service. The current implementation uses a round-robin policy to pick the *x-entry*. The credit value of the *x-entry* is set to one.

Message Passing. Message marshaling is one of the major bottlenecks for data transfer among different address spaces. XPC can mitigate the costs of marshaling as the relay segment uses the same virtual address range in caller and callee processes; therefore, we can use any self-contained data structures in the relay segment without marshaling. Figure 7 presents a case of using a linked list on the relay segment. The *handle_list* function will traverse a list in the relay segment and perform some work on each node.

To avoid a malicious caller putting an out-of-boundary pointer in the data structure, the library has provided some helper functions for pointer operations. For example, *xpclib_get_pointer* will check the boundary of a pointer to ensure it is located in the range. The library does not allow any pointers pointing to non-relay segment memory without explicit confirmation. Besides, to avoid infinite loop, the library will initialize a counter before a loop start (i.e., *Loop_start*) and validate the counter (i.e., *Loop_validate*). The threshold is user-defined. With all these helper functions, processes can safely use the relay segment's data structures without marshaling.

Notably, checking pointers may incur higher costs when many pointers are in the relay segment. In such cases, users can choose to marshall/unmarshal the data instead.

7 HARDWARE IMPLEMENTATION

Integration into RocketChip. XPC engine is implemented as a unit of a RocketChip core, which is an open-sourced RISC-V core FPGA implementation [18]. We implement the prototype on the Xilinx VC707 FPGA board.

Table 4 shows detailed information about the new registers as well as instructions. A simplified version of engine cache is implemented for evaluation, which contains only one entry and relies on software management including prefetching and eviction. The prefetching is also invoked by *xcall* #reg, but with a negative ID value (−ID). We do not implement the credit system extension and decoupled instructions in RocketChip.

Integration into Gem5. We also implement XPC on the ARM platform using Gem5 simulator [23]. The implementation is based on ARM HPI (High-Performance In-order) model [1], which mimics a modern in-order ARMv8-A implementation. We use μOPs to implement the functionalities of

Table 4. Registers and instructions provided by XPC engine.

Register Name	**Access Priv.** (R/W in kernel)	**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.
credit-base-reg		VA length	Holding the base address of credit table.
XPCState		VA length	Execution state of decoupled XPC instructions.
Instruction	**Execution Priv.**	**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.*
xcall-load/-store/-switch	User mode	*xcall-load #register; xcall-store; xcall-switch*	Decoupled instructions for *xcall.*
xret-load/-switch	User mode	*xret-load; xret-switch*	Decoupled instructions for *xret.*
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.*

XPC engine. By carefully choosing the order, we can avoid speculative issues in the *xcall* and *xret* instructions. We set the endpoint table entries to 512, length of capability bitmap to 512 bits, and call stack to 512 entries. Any load/store instructions issued by user-space on these regions will trigger an exception, which is enforced by hardware during the permission checking of load/store instructions. The implementation contains basic XPC functionalities, as well as credit system extension and decoupled instructions. It does not contain any optimizations like non-blocking *link stack* and engine cache.

8 EVALUATION

To evaluate XPC, this section answers several questions:

- How XPC improves the IPC performance? (§8.2)
- How OS services benefit from XPC? (§8.3)
- How applications benefit from XPC? (§8.4)
- How much hardware resource XPC costs? (§8.6)

Table 5. 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

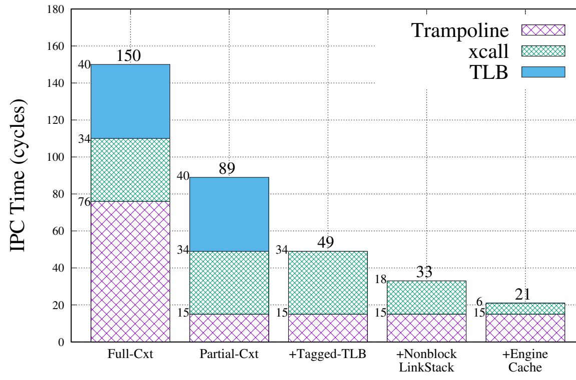


Fig. 8. XPC optimizations and breakdown (FPGA RISC-V).

8.1 Methodology

We implement the XPC engine based on GEM5 simulator (ARMv8) and two open-source RISC-V [57] implementations: siFive Freedom U500 [14] (on Xilinx VC707 FPGA board) and lowRISC [4] (on Xilinx KC705 FPGA board). The Gem5 implementation includes all features (i.e., the basic XPC, decoupled XPC and credit system), while FPGA implementations include the basic functionalities (i.e., the basic XPC). We have ported two state-of-the-art microkernels, seL4[6] on SiFive Freedom U500 and Zircon [2] on lowRISC, and added XPC support in both systems.

We also support XPC in Linux (Gem5). We port the Android Binder framework, libBinder, to both Gem5 (with Linux 4.4) and Freedom U500 (with Linux 4.15) and optimize the synchronous IPC in Binder with XPC. The simulation parameters of Gem5, listed in Table 5, mimic a modern in-order ARMv8-A implementation. We evaluate the performance of six systems: Android Binder, Zircon, seL4, Android Binder-XPC, Zircon-XPC, and seL4-XPC.

8.2 Microbenchmark

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

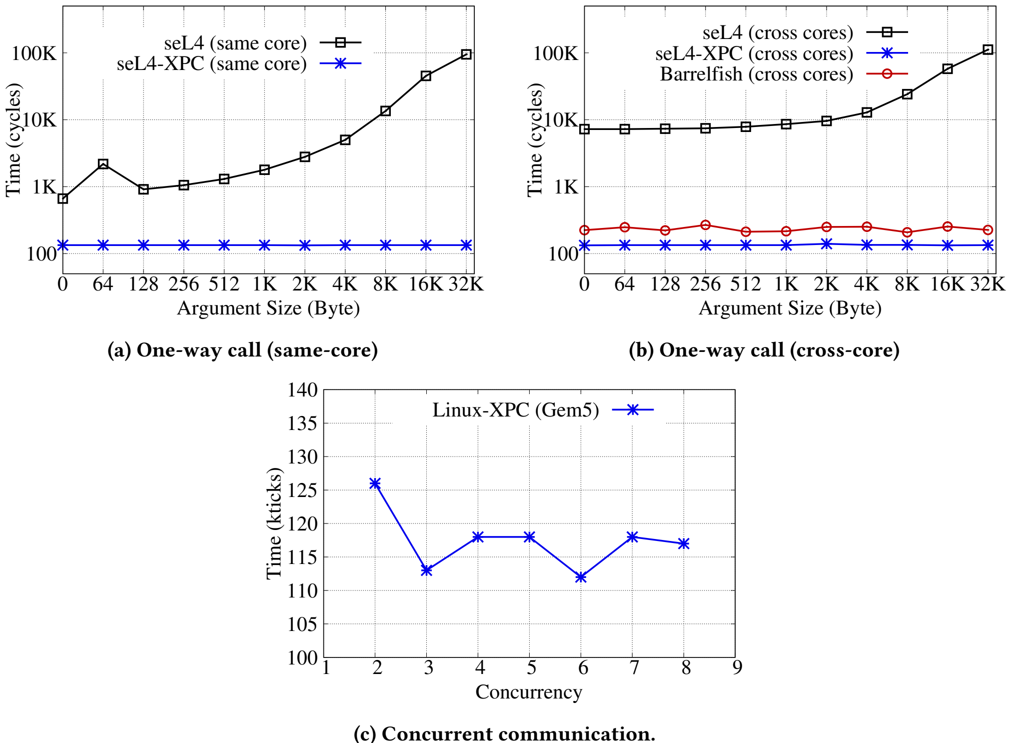
- **Full-Ctx**: saving and restoring full context.
- **Partial-Ctx**: saving and restoring partial context.

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

- **+Tagged TLB**: enabling previous optimizations and adopting tagged TLB to mitigate TLB miss.
- **+Non-blocking Link Stack**: enabling previous optimizations and adopting non-blocking link stack.
- **+XPC Engine Cache**: enabling previous optimizations and adopting cache for XPC engine.

Figure 8 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 §6.1) 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 flushing could be mitigated by adopting tagged TLB. The "Non-blocking 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 "Non-blocking Link Stack" optimizations, to ensure the fairness of the comparison.



(a) One-way call (same-core)

(b) One-way call (cross-core)

(c) Concurrent communication.

Fig. 9. IPC performance.

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 9 (a), 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

Table 6. Concurrent communication with shared server thread. The table presents end-to-end latency to call a remote shared service. "x"c means there are "x" concurrently running client processes invoking the same service. The service has only one worker thread, which is protected by the credit system. The service handler will perform long-latency operations (e.g., invoking syscalls) to increase the chances of contention.

Systems	1c	2c	4c	8c
XPC (kticks)	63,284	63,507	64,369	64,060

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).

We also evaluated the same test for Zircon and Zircon-XPC, and the results show that Zircon requires 20,569 cycles for 0B messages and 74,825 cycles for 4KB messages, while Zircon-XPC requires 324 cycles[7] for 0B messages and 371 cycles for 4KB messages. 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.

Cross-core IPC. We also evaluate the cross-core IPC performance as shown in Figure 9 (b). We compared three methods, IPC of seL4, IPC of seL4 optimized with XPC, and URPC [22] on Linux. We learned the communication implementation of Barrelfish [20] and LXDs [48] and carefully implemented and optimized the URPC[8]. The seL4 and seL4 optimized with XPC are evaluated on RISC-V FPGA boards, while URPC on Linux is evaluated on an x64 machine (i7-6700 CPU @ 3.40GHz). We choose to evaluate URPC on x64 machine because the FPGA RISC-V implementation does not have the shared LLC now, which is not suitable to illustrate the benefits of URPC's cache line-sized messages. In each case, we pined the client and server on different cores and evaluate one-way call (from caller issues IPC to the server receives the call) latency.

As shown in Figure 9 (b), the performance of cross-core IPC on seL4 is improved from 81x (small message) to 141x (4KB message size). This is because seL4 turns to use a slow path for all cross-core invocations. XPC optimized seL4 can achieve great performance because it pulls the server thread (on another core) to the current core, which is the same as the same-core IPC. URPC represents the state-of-the-art cross-core IPC performance. Since URPC and seL4 are not evaluated on the same platform, it is unfair to compare them directly. However, it is sufficient to show how XPC can significantly improve synchronous IPC (e.g., IPC in seL4) performance which is very close to URPC results. The result also shows that both synchronous IPC and asynchronous IPC can achieve great cross-core communication performance.

Last, we want to highlight two points that URPC is better and worse than XPC for cross-core IPC. First, URPC is better than XPC considering the load balance. A service can easily create multiple threads/processes on different cores to dispatch requests. Instead, XPC relies on the client to balance the loads as the server thread is running on the client's core. Second, URPC relies on polling to achieve great performance. When the server shares the core with other processes, it may significantly increase the latency [13]. We believe applications should choose different methods according to their needs on load balancing, CPU utilization, etc.

Concurrent IPC. We evaluate the performance of concurrent communication using XPC, as shown in Figure 9 (c). The test is performed on Gem5 (ARM) with Linux. We first start a server process, which will register a service configured with eight maximum concurrencies and assigned with

[7]XPC needs more cycles in lowRISC because of the CPU implementation.
[8]The source code and results are available at https://github.com/Ddnirvana/urpc_tests.

Table 7. Cycles of hardware instructions in XPC (FPGA).

Instruction	Cycles
xcall	18
xret	23
swapseg	11

Table 8. XPC instruction costs in ARM. dXPC is short for decoupled XPC, which includes three instructions for IPC call and two instructions for IPC ret. TLB flushing is not included (about 58 cycles), which can be removed with tagged TLB. It is evaluated in Gem5, and the results are represented using ticks.

Systems	Call	Return
XPC (kticks)	51	19
dXPC (kticks)	56	20

eight worker threads. After that, we start a set of client processes (from 2 to 8, pinned to different cores), to get a connection with the service and call the services using XPC (100 times). We present the average latency for the client processes to finish a single communication. As shown in the figure, the latency is quite stable among different concurrency. This is because the kernel will assign a different worker thread to each client; therefore, they can directly communicate to the server without contention with other clients.

We also evaluate the end-to-end latency of sharing the same worker thread, as shown in Table 6. The table presents the average latency of calling a remote service by changing the concurrency. The results show that the latency will increase by 1,085 kticks from 1c to 4c. The reason is that the server has only one worker thread and will handle each request one by one (guaranteed by the credit system).

Instructions costs. We measure the cycles for *xcall*, *xret* and *swapseg* instructions (results are shown in Table 7). 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, e.g., *xcall* needs to fetch an *x-entry* and push a *linkage record* to the *link stack*. Kernels can implement efficient IPC based on these primitives.

Marshaling. To reveal the benefits of the relay segment on mitigating marshaling/unmarshaling, we evaluate the performance of transferring a linked list using the relay segment. The list has 1000 entries, and each entry contains a 64bit data and a pointer pointing to the next entry (NULL for the last one), as shown in Figure 7. The baseline performance is using Binder's parcel to transfer the list, 44,536 kticks for marshaling, and 112,648 kticks for unmarshaling, as shown in Table 2.

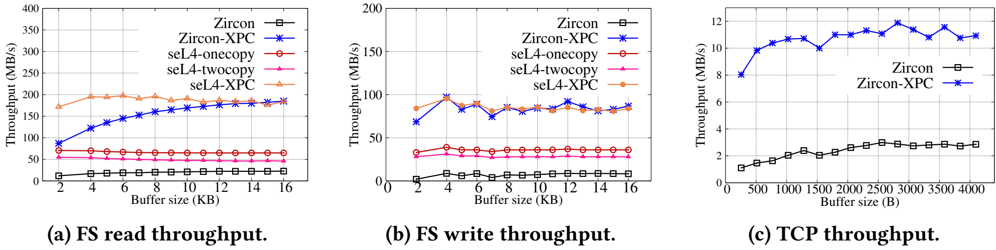
Our optimized implementation will first construct a linked list in the relay segment with an existing one on the caller side. The costs of this procedure are 25,851 kticks. It is 1.7x better than the marshaling in the baseline system. Notably, this procedure can even be mitigated if the caller directly constructs the list in the relay segment, which is reasonable as it is private unless the caller explicitly transfers to others. The unmarshaling procedure is much simpler. The callee can directly use a pointer in the relay segment to represent the list and rely on the helper functions to protect the security. The procedure to construct the linked list pointer takes 106 kticks and is magnitudes orders better than the unmarshaling baseline.

The results confirm that the relay segment can effectively transfer the messages by eliminating marshaling and unmarshaling costs.

Decoupled XPC. To know whether the decoupling will afect the performance, we evaluate the latency for *xcall*, *xret* and the decoupling version of *xcall* (*xcall-load*, *xcall-store* and *xcall-switch*) and *xret* (*xret-load* and *xret-switch*) on Gem5. The result is shown in Table 8. As the GEM5 does not simulate the TLB flushing costs (in ARM)[9], 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.

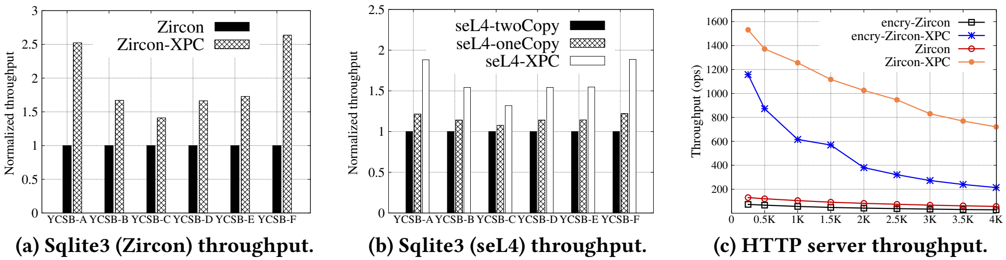
The result shows that decoupling the *xcall* and *xret* will incur minor performance impacts, while it can achieve better practicality to be implemented in real hardware.

Credit System. We also evaluate the concurrent IPC test case by setting credits to the maximum. As a single server worker thread is allowed to be invoked by concurrent clients, the kernel will see the same thread trapped into the kernel simultaneously. This can trigger kernel errors, e.g., “unhandled translation fault”, as our prototype Linux does not support a worker thread to be invoked and run concurrently. Thus, the credit system eases our concurrent communication implementation, especially on the complicated monolithic kernel like Linux.



(a) FS read throughput. **(b) FS write throughput.** **(c) TCP throughput.**

Fig. 10. 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. It is evaluated in FPGA (RISC-V).



(a) Sqlite3 (Zircon) throughput. **(b) Sqlite3 (seL4) throughput.** **(c) HTTP server throughput.**

Fig. 11. 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. It is evaluated in FPGA (RISC-V).

8.3 OS Services

To show how IPC influences the performance of microkernels, we evaluate the performance of two OS services: file system and network subsystem. The setting is seL4 (using SiFive Freedom on FPGA) and Zircon (using lowRISC on FPGA), and we use lmbench [45] to evaluate the file system and network performance.

[9]We confirmed this with the GME5 community.

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 ramdisk server). We port a log-based file system named xv6fs from fscq [26], 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 10(a) and (b). Zircon uses two-fold copying, 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 [5], 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 10(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.

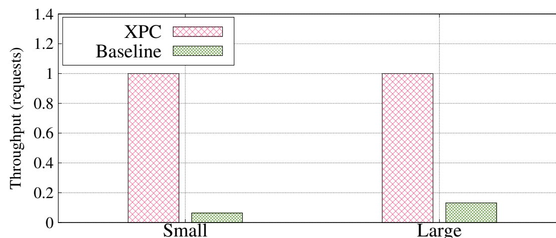


Fig. 12. Encryption serverless functions. There are three serverless functions in the chain. The throughput is normalized and higher the better. It is evaluated in FPGA (RISC-V).

8.4 Applications

To show how XPC improves the performance of real-world applications, we evaluate the performance of a database, a web server and a serverless application. In the evaluation, applications may need one copying (even using relay-segs) to match existing APIs, e.g., POSIX interfaces in microkernel-based applications. The copying costs are included in all cases.

Sqlite3. Sqlite3 [7] is a widely-used relational database. In this case, we evaluate its throughput on both seL4 and Zircon. 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 11(a) and (b). On average, XPC achieves 60% speedup in seL4 and 108% in Zircon.

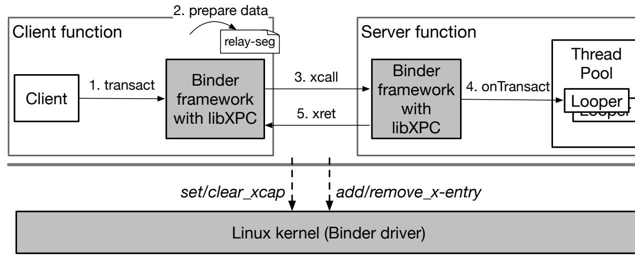


Fig. 13. XPC for Android Binder. Rectangular boxes denote components in Android; shaded boxes denote modified parts.

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 requests well.

Web Server. In this case, we evaluate the web server throughput on Zircon. Three services are involved in the web server: an HTTP service ported from lwIP, which accepts a request and then returns a static HTML file; an AES encryption service that encrypts the network traffic with a 128-bit key; an in-memory file cache service that is used to cache the HTML files in both modes. The HTTP service 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 11(c). XPC has about 10x speedup with the encryption and about 12x speedup without encryption. Most of the benefit comes from the relay segment. For a multi-service server (i.e., three services in the web server), a message will be transferred multiple times. Using relay segment can efficiently reduce the times of memory copying in these IPC.

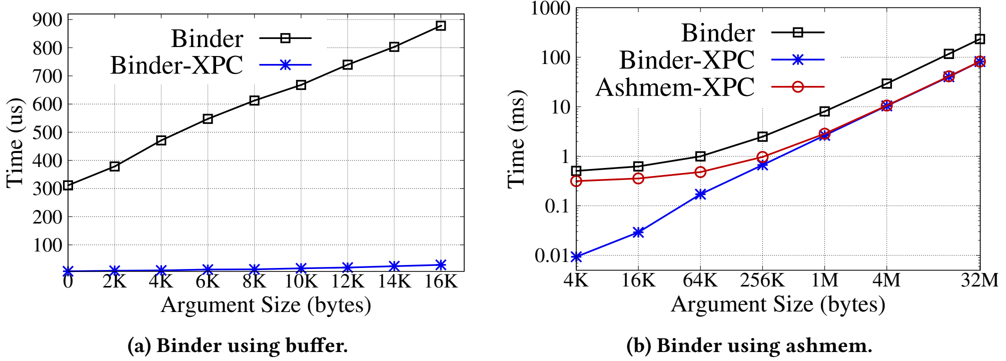
Serverless Functions. To evaluate the communication performance in real-world serverless applications, we port an online encryption application to Linux. The online encryption application includes three functions: the front-end function, the data provisioning function, and the encryption function. The front-end function will receive a request from a client, which is a file name. The data provisioning function will fetch a file according to the file name. Last, the encryption function will process the file data and return it to the client.

We compare the performance in Linux with and without XPC. The throughput is measured, and the result is shown in Figure 12. The optimized has about 15.7x speedup when the file is small and 7.5x speedup when the file is large. This is because the increased computation costs of a large file are much more than the costs of communication.

8.5 Case Study: IPC in Android

We use the widely-used IPC framework in Android, Binder, as a case study to show how XPC can achieve efficient communication in real-world scenarios. Android Binder comprises 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 (almost) unmodified. This study explains how XPC optimizes Binder's domain switching and message passing.

8.5.1 Binder Transaction for Domain Switching. The Binder framework provides a set of interfaces, named *Binder transaction*, to establish an IPC channel, configure IPC, and process IPC operations.



(a) Binder using buffer. **(b) Binder using ashmem.**

Fig. 14. Android Binder evaluation (RISC-V). Figure (a) and (b) show the remote method invocation latency between the windows manager and surface compositor with different argument sizes. Lower the better.

Table 9. Android Binder evaluation (ARM). The results are the latency of transferring messages with different sizes. Binder-XPC is the optimized implementation, and it only saves and restores minimal registers. Binder-XPC-full will save and restore all general-purpose registers for security. Binder-buffer and Binder-ashmem are two baseline systems, using transaction buffer and ashmem (zero-copying), respectively. It is evaluated in Gem5 (setting in §8.1), and results are represented using ticks.

Systems	0B	4KB	16KB
Binder-XPC (kticks)	220.8	1,133.1	3,554.9
Binder-XPC-full (kticks)	777.5	1,144.9	3,489.4
Binder-buffer (kticks)	27,682.2	42,470.7	69,445.2
Binder-ashmem (kticks)	\	47,524.9	52,075.6

We first explain five interfaces for domain switching used on the case study and then explain how to optimize these facilities' implementation using XPC.

First, a server process should register a service through Binder*addService* (❶) interface, and implement a handler function with the *onTransact* (❷) interface. Second, it registers a set of working threads through an *ioctl* command, *BC_REGISTER_LOOP* (❸). Third, the client establishes a communication connection with a server through API, *getService* (❹). Last, the client performs *transact* (❺) to invoke the services in the server side.

The optimized Binder transaction is shown in Figure 13. First, we extend the Linux kernel (Binder driver) to manage the *xcall-cap* capabilities, *x-entry-table*, and other XPC states. When a server process registers a service through *addService*, the modified framework will issue an *ioctl* command to the Linux Binder driver to add an *x-entry*. The registered handler function (i.e., *onTransact()*) is wrapped with a wrapper, which is writeen in the entry address of the *x-entry*. The wrapper will invoke the *onTransact()* handler and return the results using *xret*. The server will also register worker processes using *BC_REGISTER_LOOP*, in which the kernel will assign different *x-entry* for each worker thread, and maintains these worker threads in a thread pool. Each *x-entry* is assigned with one credit.

From the perspective of the client process, the framework will issue an *ioctl* command to acquire the *xcall-cap* capabilities when the client asks for a service through *getService* API. Besides, the kernel will pick an *x-entry* for the service from its thread pool to the client. The modified framework will use *xcall* to invoke a remote service and use the relay segment to implement *Parcels* for data transfer (explained in §6.2).

Results: We evaluate Binder by simulating the communication between two services in Android, the window manager and a surface compositor. The surface compositor will transfer the surface data to the windows manager through Binder, and then the windows manager needs 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 (see §8.5.2), and evaluate the latency for the communication. The result is shown in Figure 14(a), 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). 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). Optimized by XPC, domain switchings and memory copying are eliminated. The results are similar in the ARM, as shown in Table 9. XPC reduces the latency of Binder by 35x–123x for 0B message and about 19x for 16KB message.

8.5.2 Anonymous Shared Memory for Message Passing. Android Binder utilizes the anonymous shared memory (ashmem) subsystem in Linux to provide a file-based shared memory interface to userspace. It works like anonymous memory (i.e., ashmem does not have backed files), but a process can share the mappings with another process by sharing the file descriptor. In Android Binder, processes can share file descriptors of an ashmem through the Binder driver.

Since ashmem is one of the shared memory, applications can utilize it to achieve zero-copying message transfer. However, one-copying ashmem (e.g., the callee copies the data from ashmem to its private memory) is mostly used to avoid TOCTTOU issues. In the case study, we compare XPC with both methods. We optimize the ashmem with the relay segment.

- **ashmem allocation**: The framework allocates an ashmem by allocating a relay segment from Binder driver.
- **ashmem map**: The *map* operation will allocate virtual addresses for the segment and set the *seg-reg* register.
- **ashmem transfer**: The ashmem can be transferred among processes by passing the *seg-reg* register in the framework during *xcall*.
- **ashmem usage**: The client and server can safely use the ashmem without TOCTTOU issues as the ownership will be transferred. Furthermore, they can build complex data structures directly on the memory with several helper functions provided by XPC's library.

Using the relay segment, the framework can avoid copying, syscall operations, and marshaling/unmarshaling costs; therefore, it achieves better performance. However, one limitation is that, in the prototype implementation, there is only one active relay segment[10] at a time. Thus we rely on the page fault (implicitly)/*swapseg* (explicitly) to switch the active relay segment when applications need access to several ashmems at the same time.

Results: The result of using ashmem for data transfer in Binder is shown in Figure 14(b). The latency of Android Binder is 0.5ms–233.2ms for 4KB–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-XPC represents the results when we only optimized ashmem using the relay segment (no *xcall* support). As shown in the figure, the ashmem-XPC achieves 0.3ms for 4KB data (1.6x improvement) and 82.0ms for 32MB data (2.8x improvement). The baseline will use the one-copying ashmem. The improvement mainly comes from the secure zero-copying message transfer. We

[10]It is possible to extend to more active relay segments, and the OS should ensure they have different virtual memory regions.

Table 10. 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%

also confirm the results on ARM, as shown in Table 9. The Binder-ashmem uses the zero-copying ashmem, but still has 14x higher latency.

Summary: Overall, XPC can effectively optimize the performance of Android Binder and ashmem. Currently, the prototype only optimizes synchronous IPC in Binder (asynchronous IPC usage like *death notification* is not supported yet). As we do not support XPC in RISC-V Linux, we leverage machine mode in RISC-V to trap and handle any exception between *xcall* and *xret* (rare in the experiments).

8.6 Hardware Costs

As we use Vivado [8] tool to generate the hardware, we can gain the resource utilization report in the FPGA. The hardware costs report is shown in Table 10 (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.

9 DISCUSSION

9.1 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* [28] 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. 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 [29], which makes the timeout mechanism less useful.

Message security without marshaling. Since the pointers/addresses in a *relay-seg* are the same in caller and callee's address space, XPC library provides a simplified way to check pointer's validity

and does not require marshaling and unmarshaling. However, the feature may not suit complicated data structures, and developers should carefully use it in such cases.

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 [34]), 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, 19] 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 is very hard since the number of entries is small (only one in the paper). Moreover, the issue can be mitigated by adding tags in the Engine Cache like tagged-TLB. As each Cache entry is private for a thread (with tags), the timing attacks could be mitigated.

9.2 Scheduling Properties

A common concern of kernel-bypassed IPC is that it may violate scheduling properties like fairness and real-time scheduling. Here, we explain how these issues are addressed in XPC.

First, XPC is not a *fully* scheduler-bypassed IPC design; instead, only the data-plane of IPC invocation will bypass the scheduler. The OS kernel is still responsible for the control plane of an IPC invocation, and the hardware extension is proposed to boost the data plane. In this way, the OS kernel can carefully select channels (usually pairs of client and server threads) that can communicate directly through XPC. The same approach to decoupling a system into a control plane and data plane (which is kernel-bypassed) is also used in high-performant applications like DPDK and SPDK and other IPC designs like LVDs [49] and SkyBridge [46]. Commercial hardware, e.g., Intel, also released its prototype of an optimized IPC design based on user-level interrupts [15], which will bypass the kernel and the scheduler during communication.

Besides, the proposed generic design, the split thread model, helps the kernel understand how it should work with XPC. The model abstracts each thread with two sets of states: the scheduling state and the runtime state. During the IPC, only the runtime state is changed by the XPC hardware, while the scheduling context (includes all the information for scheduling) is reserved. That means the *the client thread should donate its time slices to invoke the server thread* (called *time-slice donation*). Time-slice donation is already used in many microkernels like Mach 3.0 [31], LRPC [21], Fiasco.OC [12], Nova [53], and seL4 [34]. XPC does not introduce anything others besides time-slice donation to the scheduling.

Last, the design of XPC is orthogonal to scheduling properties like fairness and real-time scheduling. For scheduling fairness, threads are allocated with time slices and scheduled normally using OS's policies, e.g., a client thread will still be scheduled even it *xcall*s to a server thread when it uses up its time slices. The main concern for methods like "direct switch" (terms used in L4/seL4 microkernel) for real-time scheduling has been addressed by *priorities* [29]. XPC relies on the OS kernel for the control plane. Therefore, the OS kernel can reject to use *xcall* and *xret* for IPC channels that will violate the real-time requirements. As a result, XPC can have the same fairness and real-time scheduling properties as prior OSes.

Prior works [43] explore the limitations of the IPC system based on time-slice donation. For example, scheduling context [43] illustrates that time-slice donation is insufficient to provide strong spatial isolation in seL4 and propose *scheduling context donation* during IPC. The extension could be

easy to achieve in XPC with our split thread model. Specifically, the OS can manage the scheduling context in a thread's scheduling state, which will be inherently donated to its callee thread.

9.3 Relay Segment Support

Some communication models used by applications, e.g., scatter-gather, require transferring a batch of messages. This can be achieved by using *seglist* to support multiple relay segments (§5.4). For example, to support the scatter-gather model, a caller process can split the data into multiple relay segments, and these relay segments can be saved into the *seglist*. These relay segments can be passed to different callees (for *scatter*). Similarly, a callee process can gather segments from multiple callers. Two memory copyings may be necessary during data splitting and merging, which will not be worse than existing methods (e.g., shared memory with two copyings). Besides, the new instruction, *swapseg* is proposed to switch relay segments between *seg-list* and *seg-reg*, which has small costs (i.e., 11 cycles in the best case).

Moreover, we want to clarify that the relay segment does not aim to replace existing methods like shared memory. Users are free to use relay segments with other methods if the relay segment does not bring significant benefits in some cases. We believe that the relay segment gives applications a new option to transfer messages with low latency.

10 CONCLUSION

This paper presents XPC, a hardware/software co-design for 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

The corresponding author of the paper is Haibo Chen. This work is supported by the National Natural Science Foundation of China (No. 62132014, 61925206, U19A2060), and the Program of Shanghai Academic/Technology Research Leader (No.19XD1401700).

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. lowRISC. https://www.lowrisc.org/. Referenced November 2018.
[5] 2018. lwIP. https://savannah.nongnu.org/projects/lwip/. Referenced May 2018.
[6] 2018. seL4 Benchmark. https://sel4.systems/About/Performance. Referenced November 2018.
[7] 2018. SQLite. https://www.sqlite.org/index.html. Referenced May 2018.
[8] 2018. Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html. Referenced August 2018.
[9] 2019. Anonymous shared memory (ashmem) subsystem [LWN.net]. https://lwn.net/Articles/452035/.
[10] 2019. LKML: Dianne Hackborn: Re: [PATCH 1/6] staging: android: binder: Remove some funny usage. https://lkml.org/lkml/2009/6/25/3.
[11] 2021. CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition (4.5). https://cwe.mitre.org/data/definitions/367.html. Referenced Sep. 2021.
[12] 2021. The Fiasco microkernel - Overview. https://os.inf.tu-dresden.de/fiasco/. Referenced Oct. 2021.
[13] 2021. Message Notifications, Barrelfish Technical Note 9. http://www.barrelfish.org/publications/TN-009-Notifications.pdf. Referenced Sep. 2021.
[14] 2021. SiFive. https://www.sifive.com/. Referenced November 2018.
[15] 2021. User Interrupts: A faster way to signal. https://linuxplumbersconf.org/event/11/contributions/985/attachments/756/1417/User_Interrupts_LPC_2021.pdf. Referenced Oct. 2021.

[16] 2022. seL4 Dynamic Libraries: IPC. https://docs.sel4.systems/Tutorials/dynamic-2.html. Referenced Mar. 2022.

[17] Nadav Amit, Amy Tai, and Michael Wei. 2020. Don't Shoot down TLB Shootdowns!. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 35, 14 pages. https://doi.org/10.1145/3342195.3387518

[18] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).

[19] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 2016. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *ASPLOS* (Atlanta, Georgia, USA). ACM, New York, NY, USA.

[20] 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*.

[21] 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).

[22] Brian N Bershad, Thomas E Anderson, Edward D Lazowska, and Henry M Levy. 1991. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)* 9, 2 (1991), 175–198.

[23] 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).

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

[25] 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).

[26] 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 *SOSP*.

[27] 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*.

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

[29] Kevin Elphinstone and Gernot Heiser. 2013. From L3 to seL4 what have we learnt in 20 years of L4 microkernels?. In *SOSP*.

[30] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *SOSP'95* (Copper Mountain, Colorado, USA). ACM, New York, NY, USA.

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

[32] 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.

[33] 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.

[34] 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*.

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

[36] Sanghoon Lee, Devesh Tiwari, Yan Solihin, and James Tuck. 2011. HAQu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. In *HPCA*.

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

[38] Wenhao Li, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2015. Reducing World Switches in Virtualized Environment with Flexible Cross-world Calls. In *ISCA*.

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

[40] 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.

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

[42] 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.

[43] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. 2018. Scheduling-context capabilities: a principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference*. ACM.
[44] 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.
[45] Larry W McVoy, Carl Staelin, et al. 1996. lmbench: Portable Tools for Performance Analysis.. In *USENIX annual technical conference*. San Diego, CA, USA, 279–294.
[46] 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.
[47] 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 *EuroSys*. ACM.
[48] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. 2019. LXDs: Towards Isolation of Kernel Subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 269–284. https://www.usenix.org/conference/atc19/presentation/narayanan
[49] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2020. Lightweight Kernel Isolation with Virtualization and VM Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 157–171. https://doi.org/10.1145/3381052.3381328
[50] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 241–254. https://www.usenix.org/conference/atc19/presentation/park-soyeon
[51] Jerome H Saltzer. 1974. Protection and the control of information sharing in Multics. *Commun. ACM* 17, 7 (1974), 388–402.
[52] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. 1999. *EROS: a fast capability system*. Vol. 33. ACM.
[53] Udo Steinberg and Bernhard Kauer. 2010. NOVA: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*.
[54] 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.
[55] 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.
[56] 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 *EuroSys*. ACM.
[57] 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.
[58] 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.
[59] 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).
[60] Emmett Witchel, Josh Cates, and Krste Asanović. 2002. Mondrian Memory Protection. In *ASPLOS* (San Jose, California). ACM, New York, NY, USA.
[61] Emmett Witchel, Junghwan Rhee, and Krste Asanović. 2005. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In *SOSP*. ACM.