Capability-Based Efficient Data Transmission Mechanism for Serverless Computing

Abstract

Serverless computing has enjoyed wide popularity due to its rapid deployment, elastic scaling, and pay-as-you-go advantages. Large-scale pipeline applications designed for big data processing, video analytics, and machine learning are gradually evolving toward the serverless architecture. However, the inefficiency of large data transmission among functions has limited the further adoption of the serverless paradigm in these applications. With the emergence of capability Instruction Set Architecture (ISA), the multi-tenant security isolation model and data transmission methods in the cloud have ushered in new opportunities.

This paper proposes Hummer, an efficient communication and data transmission scheme based on the capability ISA. The capability ISA allows multiple functions to run in separate isolation domains in a single address space. Function instances can efficiently share memory by directly passing hardware capabilities. This paper extends the CHERI capability ISA, introducing the novel *shared capability* and *shared capability store tracking (SCST)* mechanism to manage the shared memory access rights, enabling a zero-copy communication mechanism. We compare Hummer with existing communication mechanisms and show that it achieves a $6-40\times$ data transmission rate improvement. Redis implemented with Hummer APIs doubles its throughput.

1 Introduction

On serverless or FaaS (Function as a Service) platforms, such as AWS Lambda [3], Google Cloud Functions [10], and Azure Functions [5], functions are the minimum units of computation. Complex applications can be built as serverless workflows. The serverless workflow [4, 9, 11] transforms large-scale pipeline applications, e.g., big data processing [53, 72, 73], video analytics [20, 59, 77], and machine learning [19, 23, 67], into the Directed Acyclic Graph (DAG), where the nodes represent the serverless functions and the edges represent the data flow among the functions.

Serverless computing has dramatically improved the flexibility, elasticity, and usability of cloud computing. Users only need to decompose the application logic into functions and upload them to the serverless platform, and the requests are processed through the serverless workflow. The platform manages the underlying infrastructure and provides the runtime for functions, hiding trivial details from users. The above advantages come with the following requirements for the isolation model of functions:

(1) Security. The serverless platform should provide an isolated domain for each function. Co-located functions should not interfere with each other.

(2) Lightweightness. The function should be lightweight enough to reduce the physical resource occupation, improving the deployment density of functions in the cloud.

(3) Efficiency. The startup and destruction latency of the functions should be low enough to respond quickly to workload bursts.

(4) Compatibility. The serverless runtime should be able to support various types of applications.

Containers and virtual machines (VMs) [21, 22, 39] are still the mainstream isolation models for serverless computing. Containers rely on the mechanisms afforded by the operating system (OS), such as *cgroups*, *namespace*, and *seccomp-bpf*, to provide security guarantees. However, the kernel-sharing problem makes containers susceptible to isolation issues such as error propagation, e.g., the famous DirtyCow [18] vulnerability. VMs achieve better isolation by emulating given ISA. Despite the hardware-assisted virtualization technologies [51], VMs are still too heavy for the serverless functions. Considering the isolation issues of containers and the inefficiency of VMs, many researchers are exploring novel isolation models for the serverless runtime. The candidates are safe containers, lightweight VMs, WebAssembly, and other lightweight sandboxes with solid isolation.

Besides, the large intermediate data transmission in the serverless workflow is another obstacle to adopting the serverless paradigm in large-scale applications [50, 62]. The widely-used remote storage in the serverless workflow, e.g., Amazon

S3 [1], suffers from redundant network traffic, degrading the performance of the serverless workflow. [54] shows that running the CloudSort benchmark on AWS Lambda to process 100TB data is 500× slower than running it on VMs. To resolve the issue, prior work [17,45,64] optimizes the functions' scheduling policy and replaces the network-based storage with the in-memory storage, transmitting intermediate data through memory sharing.

We describe Hummer, which builds multiple isolation domains in a single address space using CHERI [27, 75] (Capability Hardware Enhanced RISC Instructions). It achieves efficient inter-function communication by passing only the *hardware capabilities*. The CHERI hardware capability extends the conventional pointer semantics in programming languages with abundant metadata, e.g., the corresponding memory region's base address, length, and permissions. The CHERI-enabled hardware guarantees memory access and control flow transfer legitimacy when executing capability instructions [31, 52]. Multiple serverless functions can run in different compartments built with CHERI hardware capabilities in a single address space. While the single address space is visible to all functions, the source function can grant the destination function access to the specific piece of data by delivering hardware capabilities directly without copying memory.

The main contributions of this paper are listed as follows:
(1) Serverless Isolation Model Analysis. This paper surveys the existing isolation models and inter-function communication methods for the serverless scenario. We observe the inefficiency of inter-function communication in the existing isolation models and base our work on CHERI.
(2) Secure Zero-Copy Data Transmission. To eliminate redundant memory coping in existing capability-based data transmission mechanisms, this paper proposes Hummer, a secure zero-copy data transmission mechanism based on the novel Shared Capability Store Tracking (SCST) mechanism.
(3) Purely User-mode Communication. To avoid costly user-kernel switches, Hummer enables purely user-mode communication based on the user-mode exceptions and cross-domain invocations. Each function sandbox enters the userspace program monitor through cross-domain invocations to establish communication channels with others and transmit data.
(4) Application-compatible Communication APIs. To bridge the gap between the conventional programming models and the capability-based systems, Hummer provides socket-style Shared Device communication APIs and integrates them with the existing I/O multiplexing framework.

To show that the Hummer mechanism is promising, we compare it with the existing communication mechanisms (PIPE/TCP/CP_FILE/CP_STREAM) using micro-benchmarks. The end-to-end communication latency is reduced by 69.49%/80.38%/ 8.77%/98.21% on average, and the data transmission rate is improved by 6 – 40×. In addition, we apply the Hummer APIs to Redis and run redis-client and redis-server in two capability-based compartments. Compared to the TCP-version Redis, the average end-to-end latency of Redis GET/SET requests is reduced by 45.94%/50.89%, and the throughput is improved by 1.91/2.02×.

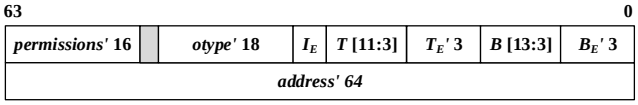

Figure 1: CHERI128 Capability Format

2 Background and Motivation

This section provides an overview of the CHERI ISA. First, we describe the CHERI hardware capability format (§2.1). Then we introduce the system architecture and corresponding software stack of the existing CHERI systems (§2.2). §2.3 finally elaborates on the motivation of our work.

2.1 CHERI Hardware Capability

CHERI hardware capability is a hardware-supported privileged boundary-checking pointer that can be used to build application-level or system-level isolation domains [28,47]. Fig. 1 shows the CHERI128 capability format[1]. In addition to the memory address pointed to by the capability, the capability contains other metadata, such as the upper and lower bounds of the memory range, capability type, and access permissions. The `otype` field identifies the type of capability (ordinary or sealed). The `permissions` field represents the access rights of the pointed memory region, such as readable, writable, and executable. The hardware performs the boundary and permission checks when jumping or accessing memory with the capability-related instructions. The `tag` field indicates whether the capability is valid. Each capability is derived from the existing valid capabilities, namely *provenance validity*, to disable the hardware capability forging. The derived hardware capabilities must not exceed the boundary and permissions of their parent capabilities, namely *capability monotonicity*.
Capability Usage Model. CHERI extends the C/C++ compiler to compile conventional programming language pointers into hardware capabilities. However, the capability pointers consume more memory space than traditional integer pointers to store the extra metadata, leading to the explosion of the binary code size. For example, replacing all pointers with capabilities results in a 43.48% growth of the binary code size. To address this issue, the CHERI ISA proposes the *Pure Mode* and *Hybrid Mode* shown in Fig. 2. Pure Mode allows only capabilities for memory accesses, and the compiler compiles the pointer dereference statements into capability-related memory access instructions. In Hybrid Mode, only the pointers

[1]CHERI provides three capability formats of different sizes: CHERI64, CHERI128, and CHERI256.

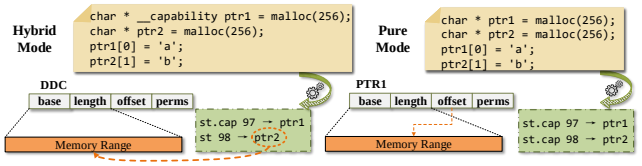

Figure 2: Pure and Hybrid Modes of the CHERI ISA

annotated with the *__capability* keyword are compiled into capabilities.

CHERI Isolation Boundary. CHERI hardware capability enforces the isolation boundary of the memory regions in a single address space. However, traditional memory accesses or control flow jumps issued through the integer pointers in the Hybrid Mode are not limited. Thus, the CHERI ISA introduces the PCC (Program Counter Capability) register and the DDC (Default Data Capability) register. The PCC register extends the traditional PC register, records the memory boundary of the current code segment, and limits the range of program control flow jumps. The DDC register limits the range of conventional pointer memory accesses. PCC and DDC ensure the security of the application control and data flow, enabling isolated domains in a single address space.

Cross-Domain Invocation Mechanism. The CHERI ISA also provides a secure cross-domain invocation mechanism for inter-domain communication [70]. The caller executes the CInvoke instruction, loads the PCC and DDC registers of the target domain, and jumps to the specific entry. To avoid malicious cross-domain invocations with the tampered PCC and DDC registers, the CHERI ISA provides the Seal mechanism to encapsulate the ordinary capability into the sealed capability. Applications cannot derive or tamper with the sealed capability. Accessing the target domain's memory area using the sealed capability is disabled as well. Unlike system calls, CHERI cross-domain invocations do not switch the privilege mode. Hummer combines the cross-domain invocation mechanism with the user-mode exception to implement purely user-mode communication.

2.2 CHERI System Architecture

The hardware capability enables CHERI to implement memory isolation and process abstraction without the traditional MMU (Memory Management Unit) architecture. The OSes and applications can run in a single virtual or physical address space in the CHERI-based system. Considering the difference between the CHERI ISA and current MMU-based architectures, CHERI requires the evolution from the state-of-the-art MMU-based OSes to single address-space capability systems [71]. Currently, there are two mainstream system architectures based on the Hybrid Mode and Pure Mode:

(1) Hybrid Capability/MMU OSes. Hybrid Capability/MMU OSes try to integrate the capability hardware with MMU, providing stronger safety guarantees. CheriBSD [8]

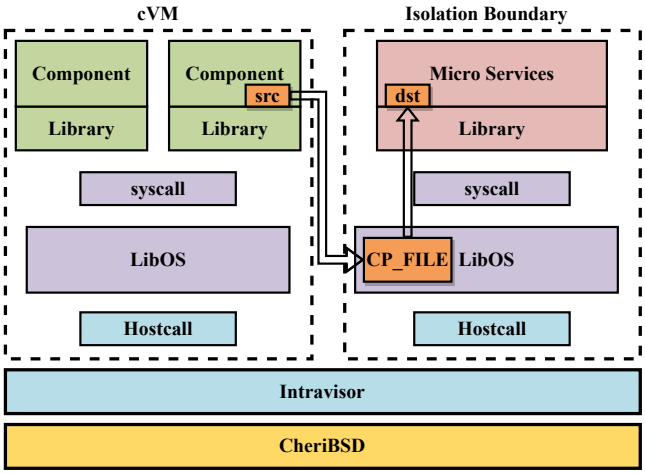

Figure 3: System Architecture and Communication Mechanisms of CAP-VMs

uses CHERI hardware capabilities to enforce the isolation boundary between user applications and the kernel. CAP-VMs [61] build the cVM sandbox for microservices in the cloud using the CHERI hardware capabilities. Fig. 3 presents the architecture of the cVM. Each cVM contains the application code, shared libraries, and the Library OS (LibOS). The Intravisor [12] is responsible for creating and destroying the cVMs. Besides, the Intravisor provides Hostcall interfaces based on the CHERI cross-domain invocation mechanism for privilege operations, such as memory allocation, thread management, and inter-cVM communication.

(2) Pure Capability OSes. Thanks to the memory safety and control flow integrity guaranteed by the CHERI ISA, CheriOS [30] proposes to run all applications, even the microkernel, in a single address space. CheriOS considers the microkernel untrustworthy and provides the Nanokernel abstraction. The Nanokernel is implemented with 2000 lines of assembly code and provides the fundamental primitives, such as *reservation* for memory management and *foundation* for cross-domain invocations.

2.3 Capability-Based Communication

§2.2 demonstrates that all capability-based systems run the applications in a single address space and use the cross-domain invocation to request services. The single address space feature offers new opportunities to eliminate memory copies and reduce communication overhead. CAP-VMs provide the single-copy CP_FILE and CP_STREAM communication mechanisms by transferring the hardware capabilities directly. The usage of CP_FILE is shown in Fig. 3. The sender sends the capabilities pointing to the message buffer to the LibOS of the receiver. Then the receiver copies the message from the original buffer to the destination buffer. CP_STREAM is similar to CP_FILE and delegates the task of coping memory

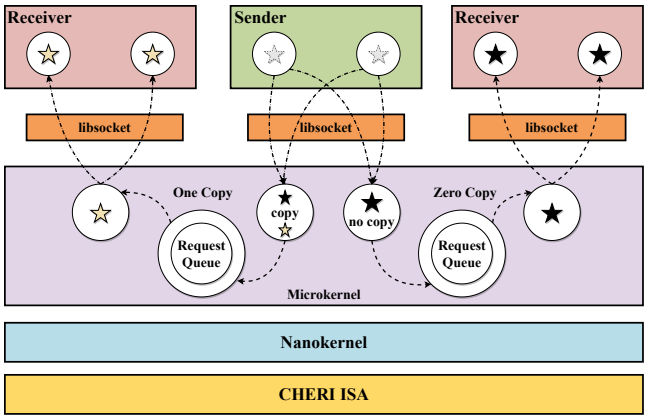

Figure 4: System Architecture and Communication Mechanisms of CheriOS

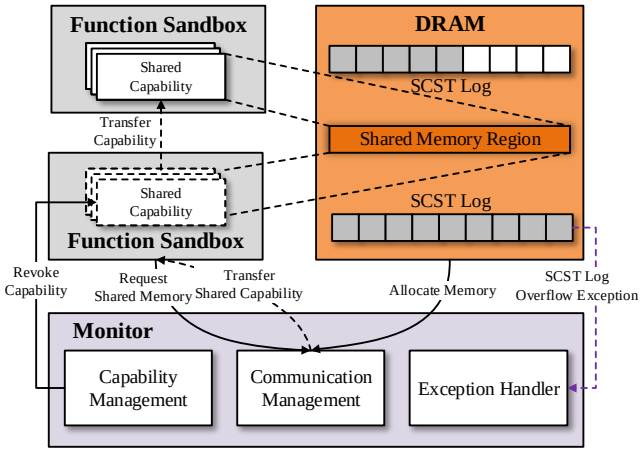

Figure 5: Hummer Architecture

to the sender.

CP_FILE and CP_STREAM mechanisms of CAP-VMs require at least one memory copy to ensure security. On the one hand, the data buffer lies within the boundary of the source cVM, so the source cVM can maliciously modify the data by forging a conventional pointer to the message buffer. On the other hand, CAP-VMs prevent programs and the LibOS from storing hardware capabilities into memory due to the lack of capability store tracking mechanisms. Programs must use the hardware capability rigidly and avoid any behaviors, e.g., function calls and variable assignments, that result in storing hardware capabilities into memory. The secure single-copy and the non-secure zero-copy communication mechanisms of CheriOS are depicted in Fig. 4. The former is similar to the CP_FILE and CP_STREAM mechanisms and suffers from the same inefficiency problem. It copies the message from the original buffer to the auxiliary buffer and then transfers the capability pointing to the auxiliary buffer to the receiver. While the non-secure zero-copy method directly transmits the capability pointing to the message buffer to the receiver. However, once the capability is transmitted, both sides can arbitrarily access the shared message buffer using the hardware capability. The CheriOS allows applications to revoke the transmitted capability, which is done by scanning the physical memory globally.

We argue that the existing capability-based communication mechanisms could be more efficient and secure. This paper presents Hummer, a capability-based zero-copy communication and data transmission mechanism that is safe and efficient simultaneously.

3 Hummer Design

As described in §2.2, existing communication mechanisms based on the hardware capability suffer from security and inefficiency issues. CP_FILE, CP_STREAM, and the secure single-copy communication mechanism of CheriOS require at least one memory copy to ensure security for the lack of efficient capability store tracking mechanisms. CheriOS avoids copying memory by directly passing the hardware capabilities without security considerations, which is vulnerable to the TOCTTOU (Time of Check to Time of Use) attack. The sender can tamper with the data using the capability, bypassing the receivers' security check. CheriOS allows programs to revoke specified capabilities by globally scanning the physical memory to find all in-memory capabilities.

To address the above limitations, this paper proposes three design principles for the capability-based communication system.

(1) Security. The system should control the data buffer access rights of both sides by transferring ownership of the hardware capabilities. In this way, both sides of the communication cannot simultaneously modify the data. Thus, attacks like TOCTTOU are eliminated.

(2) Efficiency. The system should avoid copying memory by transferring only hardware capabilities, which are tokens of the data buffers, to the destination. In addition, the system should avoid costly operations in the communication process.

(3) Compatibility. Considering the heterogeneity between the capability hardware and traditional MMU (Memory Management Unit), the system should provide compatible interfaces for the existing software and minimize the engineering efforts to adopt our mechanism.

In §3.1, we present the system architecture of Hummer following the above principles.

3.1 System Architecture

We present the Hummer architecture in Fig. 5. First, we extend the CHERI capability ISA and introduce the novel hardware *shared capability* (§3.2) pointing to the shared data buffer. The monitor prevents the source function from accessing the shared data buffer by revoking the shared capability. To avoid the costly global memory scan when revoking the shared capa-

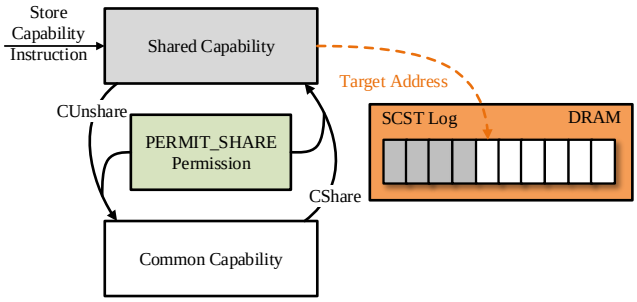

Figure 6: The Shared Capability Extension

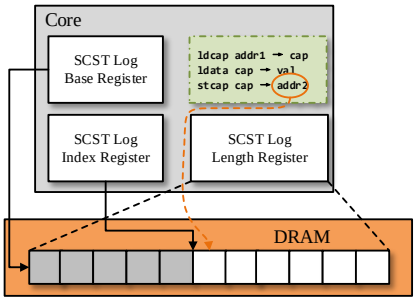

Figure 7: SCST Mechanism

bility, we propose the *shared capability store tracking (SCST)* mechanism (§3.3) to record the in-memory shared capability addresses in the SCST log region. When the sender transmits data to the receiver, the monitor traverses the sender's SCST log region and seals or revokes the corresponding shared capabilities of the sender. Then the receiver could acquire exclusive write or shared read access rights to the shared data buffer. The sender cannot maliciously modify the data after transferring the shared capability to the receiver, guaranteeing **security**. Since the CHERI ISA enables multiple functions and the monitor to run in different isolation domains in the same address space in any RISC-V privilege level, we implement Hummer as a purely user-mode communication system.

Conventional methods of memory sharing, e.g., `mmap`, require a costly trap into the kernel. We integrate the user-mode exception mechanism (§4.2) and shared capability management into the user-mode monitor, eliminating extra user-kernel switches and achieving **efficiency**. As for **compatibility**, Hummer implements easy-to-use communication interfaces (§4.3) based on the CHERI Hybrid Mode. Applications access the shared data buffer with capabilities like traditional pointers. We prototype Hummer based on CAP-VMs, by implementing communication management, capability management, and user-mode exception in Intravisor. Currently, we build the function sandbox runtime upon cVM. §6 discusses further cVM memory footprint optimizations for serverless computing.

3.2 Shared Capability Extension

To distinguish our system from other capability systems for tracking purposes, we describe the shared capability. It contains the following extensions, as shown in Fig. 6.

(1) Hardware Shared Capability. As described in §2.1, the `otype` field in hardware capability indicates the capability type. We choose reserved `otype` 4 to represent the shared capability.

(2) `CShare` Instruction. The `CShare` instruction constructs shared capabilities. It converts an ordinary capability into a shared capability.

(3) `CUnshare` Instruction. The `CUnshare` instruction destroys shared capabilities. It converts a shared capability into an ordinary capability.

(4) `PERMIT_SHARE` Permission. The program can execute `CShare`/`CUnshare` instructions to construct/destroy shared capabilities only when its `PCC` `PERMIT_SHARE` permission bit is set. Hummer grants the `PERMIT_SHARE` permission to the monitor. In case functions destroy shared capabilities to bypass the SCST mechanism, Hummer clears the `PERMIT_SHARE` permission bit of the function sandboxes.

3.3 SCST Mechanism

Due to the lack of capability store tracking mechanisms, existing capability systems are unable to revoke or seal functions' access rights to the data buffers after transferring the buffers to other functions. Therefore, Hummer designs the SCST mechanism depicted in Fig. 7. The shared capability store instructions trigger hardware logging. Then the CPU writes the target virtual address of the store instruction into the predefined SCST log region. To eliminate the address translation when logging, we allocate a continuous physical memory region for the SCST log and pin it in physical memory. The SCST log region is defined by the following three registers: **(i) SCST log base register**, which records the starting physical address of the SCST log region; **(ii) SCST log length register**, which records the length of the SCST log region; **(iii) SCST log index register**, which records the index of the next SCST log. The SCST log base and length registers are only accessible in RISC-V Supervisor Mode since we allow only CheriBSD to access the physical address of the SCST log region. The index register is accessible in RISC-V User Mode for the SCST log overflow handler (described below) to reset the SCST log index. When the SCST log region is full, we need a mechanism to prevent the loss of SCST log. We introduce the user-mode SCST log overflow exception to inform the monitor that the SCST log region is full. The monitor maintains an ordered address list for each shared capability. When the SCST log region is full, the exception handler flushes the SCST log to the ordered list. The overhead of SCST mechanism is negligible compared with Intel Page Modification Logging (PML) [24], which has been widely used for accelerating VM live migration. The PML mechanism logs the Guest Physical Address (GPA) on each memory

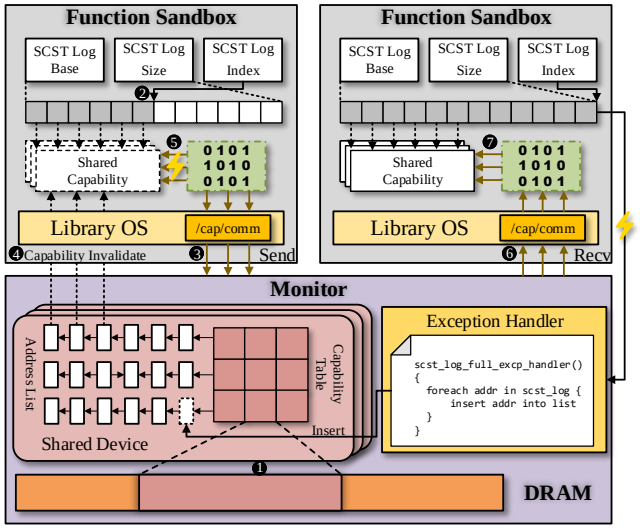

Figure 8: Efficient Data Transmission with Hummer

write by the VM (Virtual Machine). While SCST only logs the target address of the shared capability store instructions, which is rare compared to memory write operations.

3.4 Data Transmission Process

Fig. 8 details the communication and data transmission process based on the shared capability extension and the SCST mechanism. ❶ The sender requests shared data buffers from the monitor through the cross-domain invocations. The monitor allocates fixed-size memory regions, constructs shared capabilities, and delivers them to the sender. ❷ The sender fills the shared data buffers using capability-based memory access instructions. The CPU automatically appends the target addresses to the SCST log if the sender attempts to store shared capabilities into memory. The SCST log overflow exception handler flushes the addresses into the ordered address list if the SCST log gets full. ❸ The sender calls into the monitor domain to transfer the ownership of the shared data buffers. ❹ The monitor revokes the sender's access rights to the shared data buffers by sealing or destroying shared capabilities recorded by the address list and SCST log. The shared capabilities in the capability register files also need to be revoked. ❺ Invalid access exception occurs if the sender tries to access the shared data buffers again after finishing the send request. Besides, the sender cannot access the shared data buffers by forging pointers because the buffers are outside its isolation boundary. The sender's `DDC` register will automatically forbid out-of-boundary memory accesses. ❻ The monitor transfers the shared capabilities to the receiver when it launches the receive request through the cross-domain invocation. ❼ When the receiver accesses the shared data buffers through the shared capabilities, it also follows the SCST rules, writing the target addresses of the shared capability store instructions to its SCST log region.

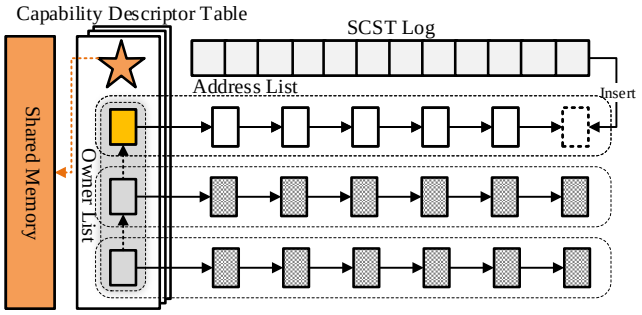

Figure 9: Shared Capability Descriptor Table

4 Implementation

This section elaborates on the implementation of Hummer. We implement the shared capability extension and SCST mechanism by adding 730 LoC (Lines of Code) to the CHERI-QEMU emulator and 150 LoC to the CHERI-LLVM compiler. We implement our communication system based on CAP-VMs with 3000 LoC in Intravisor, 2000 LoC in Linux Kernel Library, and 850 LoC in CheriBSD. Besides, additional 250 LoC assembly enables user-mode exception context switches. We plan to open source Hummer in the near future.

4.1 Shared Capability Descriptor Table

Hummer introduces the Shared Capability Descriptor Table (Fig. 9) to manage the shared data buffers and related shared capabilities. The Shared Capability Descriptor Table stores shared capability descriptors, which point to the shared data buffers between function sandboxes. The descriptor contains the shared buffer size, the current owner of the shared data buffer, the shared capability pointing to the shared data buffer, and the ordered address list of the shared capability. The shared capability descriptor also contains a previous owner list, recording the owner function sandboxes of the shared capability and their address list heads. When a function sandbox transmits data to other sandboxes, the monitor revokes the sender's ownership of corresponding shared capability by traversing the ordered address list.

4.2 SCST log overflow exception

Hummer follows the RISC-V N-Extension requirements and implements the user-mode exception mechanism. RISC-V provides three privilege modes: User mode, Supervisor mode, and Machine mode. All interrupts and exceptions are delivered to the Machine mode by default. The Machine mode delegates the interrupts and exceptions to the Supervisor mode by setting `mideleg` and `medeleg` control and status registers (CSRs), respectively. RISC-V N-Extension proposes the `sideleg` and `sedeleg` CSRs to delegate the interrupts and exceptions to User mode. Moreover, the RISC-V N-Extension uses other CSRs to track the interrupt and excep-

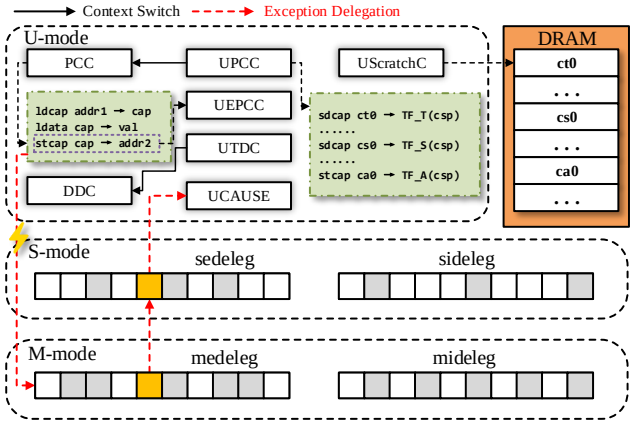

Figure 10: User-mode Exception Context Switch

tion information: `uepc` records the interrupted instruction PC; `utvec` stores the vectored interrupt or exception handler entry; `ucause` reports the interrupt or exception reason; `utval` provides additional information of the interrupt or exception, e.g., fault addresses; `uscratch` records the top address of the user exception stack. The user-mode exception handler executes the `uret` instruction to return from the user-mode interrupt or exception handler.

The CHERI ISA puts more constraints on the user-mode exception mechanism. The CHERI ISA extends `PC` to `PCC` to guarantee control flow transfer security, and adds the `DDC` capability register for secure memory accesses (§2.1). To prevent user-mode exception handler from crossing the isolation boundary, the CHERI ISA extends `utvec`, `uepc` and `uscratch` to `UPCC`, `UEPCC`, and `UScratchC` capability registers, respectively. The CHERI ISA also introduces the `UTDC` capability register to control the user-mode exception handler's memory accesses.

Fig. 10 demonstrates the process of user-mode exception context switch. Domain switch (`PCC` and `DDC`) is required in this process as well since the user-mode exception handler and the function belong to separate isolation domains. Once the function sandbox triggers a user-mode exception, the hardware sets the handler's privilege mode according to `medeleg` and `sedeleg` CSRs. The CPU saves `PCC` to `UEPCC` and loads `UPCC` into `PCC`. `UPCC` points to a piece of assembly code. The assembly code first loads `UScratchC` into the stack pointer and then saves general-purpose registers to the user exception stack. These registers in the stack are called `trapframe`. Then the assembly code writes `UTDC` value to `DDC`, completing the domain switch. The `trapframe` serves as the arguments of the user-mode exception handler. The user-mode exception handler recovers the context of the interrupted program and executes `uret` instruction to return when exception handling is finished.

Correctly configuring the user-mode exception stack is also crucial. Currently, Hummer reuses the isolation mechanism of CAP-VMs. The monitor and function sandboxes are different

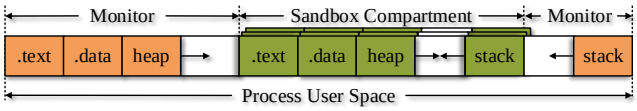

Figure 11: Memory Layout of the Monitor Process

threads of a single process in CheriBSD. The monitor is the main thread of the process and splits the address space into several isolation domains. Each function sandbox contains some POSIX threads. Fig. 11 depicts the memory layout of the monitor process. Hummer places the main thread's user-mode exception stack at the top of the user space. Threads in function sandboxes use the top memory area of the isolation domain as the user-mode exception stack.

4.3 Communication APIs

Tab. 1 presents two sets of Hummer communication APIs: (i) the simple communication interfaces named **Shared Region**; (ii) application-compatible communication interfaces named **Shared Device**. Shared Region related interfaces are implemented as system calls based on the Shared Capability Descriptor Table described in §4.1. Each shared region corresponds to a shared capability and can be delivered through the serverless workflow using the `shr_send` interface. During this process, the monitor transfers the ownership of the shared data buffer by invalidating all shared capabilities of the source function.

Unfortunately, Shared Region interfaces are incompatible with the current communication mechanisms based on file descriptors and cannot be integrated with the existing I/O multiplexing frameworks, e.g., Select/Poll/Epoll. Thus, Hummer proposes the Shared Device framework in Fig. 12. Hummer installs a communication module to the LibOS and exposes it to the function as a character device. The function sandboxes establish peer-to-peer communication channels using the `sd_listen`, `sd_connect` and `sd_accept` interfaces through the character device. Once the connection is established, the monitor creates a Shared Device. Both sides of the communication channel will get a file descriptor representing the Shared Device. Later, they initiate read and write requests through the file descriptor to send and receive data. The Shared Device consists of a shared capability descriptor table and two ring queues. The ring queues store the messages or data to be handled by the target function sandbox. The target function sandbox handles these messages sequentially. The message or shared data are transferred by simply passing shared capabilities, as described in §4.1, without any memory copying.

The Shared Device interface is in the socket style and can be conveniently integrated with the existing I/O multiplexing framework. Currently, the I/O multiplexing frameworks are maintained by the LibOS of the function sandboxes. The LibOS should be able to inform the functions of the incom-

Type	API	Description
Shared Region	**shr_alloc**(size_t size, void *addr)	Allocate shared buffer and write shared capability to addr
	shr_destroy(int shrid)	Free shared buffer with shrid
	shr_send(int shrid, int dst)	Send shared buffer with shrid to dst sandbox
	shr_recv(int shrid, void *addr)	Receive shared buffer with shrid
Shared Device	**sd_listen**(void)	Wait for connection from other sandboxes
	sd_accept(int flags)	Accept the connection from other sandboxes
	sd_connect(int dst, int flags)	Connect to the dst sandbox
	sd_alloc(int size, void *addr)	Allocate shared buffer and write shared capability to addr
	sd_free(int shrid)	Free shared buffer with shrid
	sd_read(void *addr)	Read shared capability into addr
	sd_write(int size, int shrid)	Send shared capability to destination

Table 1: Hummer Communication APIs

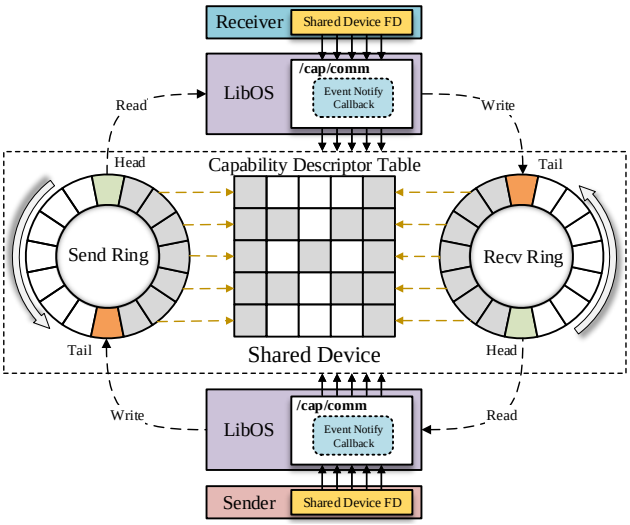

Figure 12: The Shared Device

ing readable/writable events. However, the LibOS has no information about the events' occurrence since function sandboxes jump to the monitor to send or receive messages. Thus, Hummer proposes the internal event notification mechanism. When the sender injects a message into the ring queue, the monitor executes `CInvoke` instruction, switches the `PCC` and `DDC` registers, jumps to the predefined trampoline in the receiver domain, and wakes up waiters. Each function sandbox reserves a memory region in the isolation domain as the event notification stack for the trampoline code to execute.

5 Evaluation

We now examine the performance of the Hummer communication mechanism and the proposed interfaces for using Hummer in applications. We first describe the evaluation platforms and configurations in §5.1. We then compare the performance of Hummer with cVM communication mechanisms and existing OS mechanisms using micro-benchmarks in §5.2. In §5.3, we deploy Redis with the socket-style Shared Device APIs of Hummer to validate the benefits of the Hummer communication mechanism and break down the latency of Redis requests to explore the performance bottleneck of our system.

5.1 Experiment setup

The physical and emulated testbed. We evaluate our system on an emulated environment on CHERI-QEMU [14] (version 5.2.0), which extends QEMU with CHERI support. The CHERI-LLVM [7] version is 11.0.0. All experiments are conducted on a physical server with 32 Intel Xeon Gold 6151 cores and 32GB memory. The physical server runs Linux Kernel v5.4.0 and the Ubuntu 18.04.6 LTS OS. We run CheriBSD [8] OS (version 13.0) on CHERI-QEMU, which provides an emulated RISC-V platform with CHERI extensions. Since Multi-TCG [6] is not supported in CHERI-QEMU, i.e., the vCPUs are implemented as a single host OS thread, we run one vCPU and pin it to a physical CPU to avoid the scheduling overhead. We allocate 4GB memory for CHERI-QEMU to avoid swapping.

Benchmark configurations. We compare the Hummer communication mechanism (SCST) with existing OS mechanisms (MEMCPY/TCP/PIPE) and cVM mechanisms (CP_FILE/CP_STREAM). For traditional OS mechanisms, benchmarks run on CheriBSD without modification. For other mechanisms that require the CHERI capability, all benchmarks run in a single address space using dedicated APIs, each in a separate sandbox with non-overlapping address ranges. All benchmarks run on the emulated environment to ensure fairness.

Time measurement. To get the time, applications running with CHERI capability enabled first invokes the APIs provided by musl libc[2], which calls into the LibOS (Linux Ker-

[2]https://www.musl-libc.org/

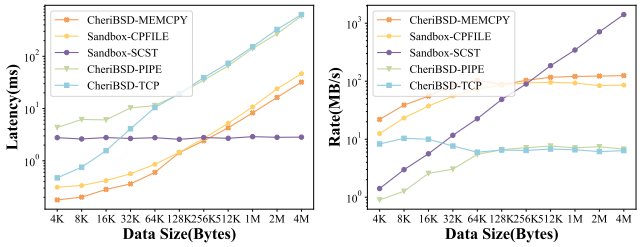

Figure 13: Latency and Data Transmission Rate of the Server

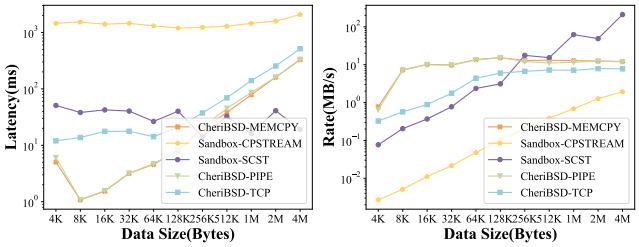

Figure 14: Latency and Data Transmission Rate of End-to-End Requests

nel Library [55]) and finally gets the result. While for the applications with CHERI disabled, they directly invoke the C library functions to get the time. To avoid this inequity, we leverage the `rdtime` instruction of RISC-V, which reads the RISC-V `time` CSR [69] of the single vCPU in the emulated environment to get the time. The value of the time CSR is increased by one every 100ns, guaranteeing 100ns accuracy.

5.2 Micro-benchmarks

To measure the latency and throughput of the communication mechanisms (MEMCPY/TCP/PIPE/CP_FILE/CP_STREAM/SCST), we use a synthetic client-server micro-benchmark. In the benchmark, the client issues requests of varying sizes (from 4KB to 4MB) and transfers them to the server through the above communication mechanisms. After the server receives and checks the integrity of the transferred data, the server returns an acknowledgement to the client. Configurations for each communication mechanism are detailed as follows:

1. CheriBSD-{MEMCPY, TCP, PIPE}: The client and server are each a separate process on CheriBSD. The server receives data through `memcpy`, TCP sockets, or pipes.

2. Sandbox-{CPFILE, CPSTREAM}: The client and server each run in a separate sandbox and communicate through CP_FILE or CP_STREAM provided by the CAP-VMs.

3. Sandbox-SCST: The client and server each run in a separate sandbox, and communicate through the Hummer APIs.

The traditional mechanisms (CheriBSD-TCP, CheriBSD-PIPE) require multiple memory copies, including a copy between kernel and user buffers on both client and server sides. The CAP-VMs mechanisms (Sandbox-CPFILE, Sandbox-CPSTREAM) implement communication using only one memory copy. For CP_FILE, the copy is performed on the server side, while for CP_STREAM, the copy happens on the client side. The SCST mechanism proposed by Hummer enables zero-copy communication by only passing the shared capabilities between the client and server.

Server-side latency and data transmission rate. Since the emulated environment is configured to have one vCPU, the client-server data transmission incurs at least one context switch. We record the timestamps before and after the non-blocking `recv` function call at the server side and get the latency, thus eliminating the overhead of the context switch. Fig. 13 presents the `recv` latency and data transmission rate. All results are the average of five runs. Sandbox-CPSTREAM results are not presented since the receive side needs to call `recv` to register a shared capability. Then the sender can issue the request according to the shared capability. Thus context switches cannot be avoided. We show Sandbox-CPSTREAM in the end-to-end tests. Moreover, the LibOS and CheriBSD implementations of `memcpy` differ. Thus we instead use the RISC-V `ld` and `st` instructions to copy data in 8-byte granularity. For capability-enabled settings, we use `cld` and `cst` to do memory copy.

In general, latencies of all mechanisms except Sandbox-SCST grow linearly as the request size grows, since at least one memory copy is involved. The data transmission rate stops increasing after the data size gets to around 64KB, as the memory copy rate is limited by the memory bandwidth of the platform. CheriBSD-MEMCPY results are the upper bound performance of all memory-copy involved mechanisms. While for Sandbox-SCST, the data transmission rate is not limited by the memory bandwidth, and can continue to rise linearly as the data size grows. However, a minor performance decrease arises when request sizes are below 32KB. The SCST mechanism requires the applications to call into the LibOS, which then invokes the monitor through the cross-domain invocation and does shared capability revoking. This overhead cannot be reset by the benefits of zero copy when the request size is small.

End-to-end latency and data transmission rate. We explore the end-to-end latency of all communication mechanisms in Fig. 14 by recording the timestamps before the client sends requests and after the server receives data. CP_FILE is unusable for asynchronous communication and cannot run our test, so Sandbox-CPFILE results are not presented. To avoid the influence of server sandbox initialization, the client issues a test begin request to the server and waits for the server's acknowledgement before sending other requests. Similar to the server-side tests, the benefits of the Hummer zero-copy mechanism are remarkable when the data size is large. The data transmission rate of our mechanism (Sandbox-SCST) is 6 – 40× of other mechanisms. Sandbox-CPSTREAM presents much higher latency compared with the CHERI-disabled

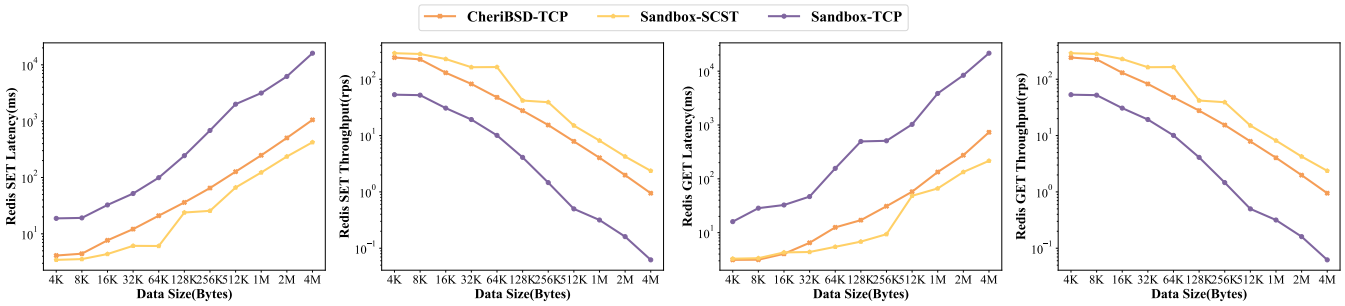

Figure 15: Latency and Throughput of Redis GET/SET Requests

mechanisms (CheriBSD-TCP, CheriBSD-PIPE) due to the frequent context switches between the sandbox and the monitor and the synchronization overhead. We observe a latency jitter in Sandbox-SCST results, although SCST latency is independent of the request size, varying between 16ms to 50ms. This fluctuation is due to the uncertainty of the thread scheduling by CHERI-QEMU TCG. However, when the data size is larger than 1MB, Sandbox-SCST still significantly outperforms CheriBSD-MEMCPY in latency and data transmission rate.

Conclusion. The Hummer communication mechanism decouples request latencies from request data sizes and presents tremendous performance advantages over traditional mechanisms when the data size is large.

5.3 Deployment with Redis

Hummer provides socket-style Shared Device APIs for applications to facilitate its wide use. We rewrite the redis-server and redis-client (version 7.0.4[3]) with Shared Device APIs, and measure the Redis GET/SET latencies. To use the Shared Device APIs, the redis-server opens the `/cap/comm` file instead to get the shared device file descriptor. For the `read,write` APIs, the shared device sends and receives the shared capabilities instead of data buffers. We compile Hummer version of Redis using CHERI-LLVM. Since CP_FILE and CP_STREAM versions of Redis and NGINX are not open-sourced in the CAP-VMS paper, we do not have a comparison with them. The configurations are listed as follows:

1. Sandbox-{TCP, SCST}: The redis-client and redis-server each run in a separate sandbox in the same address space, communicating with TCP sockets or Shared Device.

2. CheriBSD-TCP: The redis-client and redis-server run as separate processes on CheriBSD, communicating through the TCP sockets.

Fig. 15 plots the results. On average, the Hummer mechanism reduces the Redis GET and SET latency by 92.4% and 94.6% compared to Sandbox-TCP, which is a considerable improvement. The Sandbox-SCST configuration eliminates the traditional network stack from the communication path, while requiring only two switches to the monitor. For Sandbox-TCP, multiple transitions from Redis to monitor are required, and both LibOS and CheriBSD network stacks are involved, leading to overwhelmingly high latency. Although the CheriBSD-TCP setting only goes through the single CheriBSD network stack, Sandbox-SCST still presents a 45.9% and 50.9% reduction in Redis GET and SET latency, respectively. The throughputs of Redis GET and SET are also boosted by 1.9× and 2×, respectively. This is due to the zero-copy capability provided by Hummer, which overcomes the major communication bottleneck of Redis compared to TCP.

Redis GET/SET latency breakdown. To demystify the reason behind the Sandbox-SCST improvements, we breakdown the Redis GET/SET process of 4MB data size, and plot the detailed time cost in Fig. 16. The stages of the processes are listed as follows:

1. Build Request. The redis-client constructs the request before the Hiredis library calls `write` to issue the request.

2. Transfer Request. Start when the Hiredis library issues `write` and stop when the redis-server's `read` returns.

3. Handle Request. Start when the redis-server's `read` call returns and stop when the redis-server calls `write` to reply.

4. Transfer Reply. The reply from the redis-server goes through sockets and is received by the redis-client.

5. Handle Reply. The redis-client processes the redis-server's reply.

As expected, transferring request (Stage 2) and transferring reply (Stage 4) are the dominating parts of the Sandbox-TCP setting (96.76% and 99.77%, respectively), which suffers from a complicated TCP stack and frequent application-monitor transitions. Meanwhile, the TCP stack requires several rounds of memory copying, aggravating the performance problem. For CheriBSD-TCP, the Stage 2 and 4 are shortened to 66.77% and 86.69%, respectively, due to the shorter network stack. Thanks to the zero-copy communication mechanism, Sandbox-SCST shortens the percentage of Stage 2 and 4 to 18.47% and 78.51%. However, the Stage 4 latency is not accurate. It reckons in the additional memory copy overhead since the sender needs to copy the reply to the destination buffer immediately in the CheriBSD-TCP setting. Thus, we count in the overhead in Sandbox-SCST as well for equity. The accurate latency of transferring the reply is 32.5ms, which

[3] https://github.com/redis/redis/tree/7.0.4

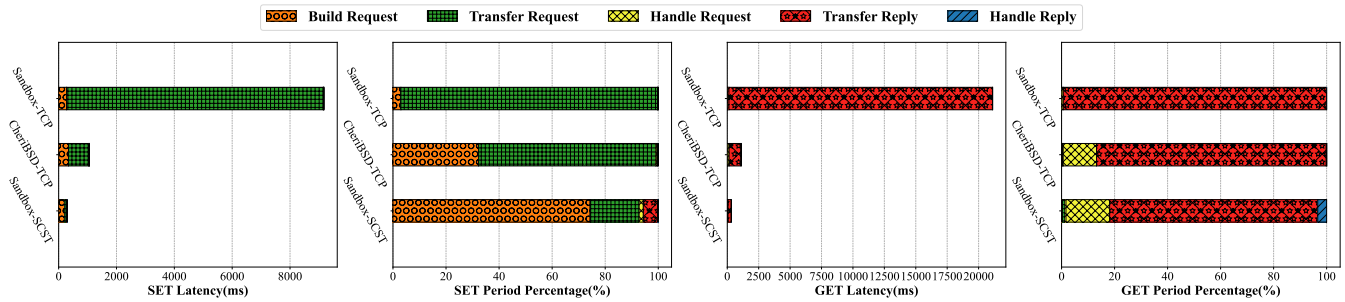

Figure 16: Latency and Percentage of Each Period in Redis GET/SET Requests

accounts for only 4.4% of the process.

Conclusion. Hummer provides easy-to-use interfaces and enables ultra-low latency when data size is large for cloud applications. The breakdown results attribute the improvement to the zero-copy communication mechanism of Hummer.

6 Related Work

Serverless Isolation Models. Some researchers focus on safe containers. gVisor [76] and Quark Container [15] narrow the attack surfaces by intercepting and filtering system calls of applications. Kata Container [56] further strengthens the isolation of containers by running containers in virtual machines. Some researchers are interested in lightweight VMs. Firecracker and StratoVirt [16] simplify the hypervisor to adapt to the serverless architecture. Unikernel [25, 44, 46] sacrifices the generality of conventional operating systems to improve performance, causing compatibility issues. NanoVMs [13] propose the *ops* tool, which can convert container images into Unikernel images, to maintain compatibility. Unikraft [42] and OSv [40] implement Linux-compatible OSes from scratch. X-Container [63] directly uses Linux as the exokernel to adapt to applications. To adapt Unikernel to the serverless platform, Jitsu [43] proposes the first event-driven Unikernel framework. UniFaaS [48] implements a Unikernel-oriented serverless platform based on Solo5 [74] and compares it with OpenWhisk [2], demonstrating the superiority of Unikernel in terms of startup latency and memory footprint. SEUSS [26] starts up functions from the Unikernel snapshots, reducing the initialization time to 10ms. FAASM [64] proposes to leverage the Software Fault Isolation (SFI) [66] mechanism to build isolation domains for serverless functions in a single address space. [32, 33, 36] all choose WebAssembly [34] as the isolation model for serverless functions in IoT (Internet of Things) and edge computing scenarios. Sledge [33] implements efficient user-mode scheduling and resource sharing atop the WebAssembly. Photons [29] runs multiple functions in a Java Virtual Machine (JVM) to improve resource utilization.

Inter-function Communication. Boki [38] constructs a message queue based on the shared log for communication and collaboration between serverless functions. FAASM [64] extends the WebAssembly memory model to transfer large amounts of data between Faaslets through shared memory. Pocket [41] provides different storage methods for short-term data of different data volumes while maintaining the scalability of the serverless architecture. CloudBurst [65] implements the scalable key-value store Anna to share function state. Photons [29] builds shared object storage through key-value mapping to realize application state sharing. SAND [17] presents a hierarchical message bus mechanism. SONIC [45] adopts three data transmission methods: virtual machine storage, direct message transmission, and remote storage, according to the message size and function attributes. [60] establishes a shared memory-based communication between function containers on the same physical node for efficient communication. [37, 49, 57, 58, 68, 78] aiming at optimizing communication between virtual machines and containers are also applicable to serverless platforms.

7 Conclusion and Future Work

Serverless computing demands secure, lightweight, and flexible isolation models with high-efficiency data transmission. Hummer proposes to build function sandboxes with the emerging CHERI ISA and provides a secure zero-copy data transmission method by transferring the hardware capability directly. The shared capability extension and SCST mechanism enable Hummer to efficiently manage functions' access rights to the shared data buffers. Hummer can be integrated with the prevalent I/O multiplexing frameworks, thanks to the application-compatible communication APIs.

Currently, Hummer relies on the user-mode exception handler to remove reduplicate addresses in the SCST log, which can be filtered by the hardware address cache. Besides, Hummer reuses CAP-VMs' isolation mechanism and suffers from the static memory division problem. Thus, we propose to combine Hummer with the virtual memory block mechanism like VBI [35], supplying dynamic memory management in the single address space. Moreover, we plan to provide a modular LibOS. The function sandbox loads the necessary LibOS components on demand. Multiple sandboxes can share a single LibOS component, reducing memory footprint.

References

[1] Amazon S3. Object storage built to store and retrieve any amount of data from anywhere. https://aws.amazon.com/pm/serv-s3/. Last Accessed: January 9, 2023.

[2] Apache OpenWhisk. An open source, distributed Serverless platform. https://openwhisk.apache.org/. Last Accessed: January 4, 2023.

[3] AWS Lambda. Run code without thinking about servers or clusters. https://aws.amazon.com/lambda/. Last Accessed: January 9, 2023.

[4] AWS Step Functions.Visual workflows for distributed applications. https://aws.amazon.com/step-functions/. Last Accessed: January 9, 2023.

[5] Azure Functions. Execute event-driven serverless code functions with an end-to-end development experience. https://azure.microsoft.com/en-us/products/functions/. Last Accessed: January 9, 2023.

[6] Features/tcg-multithread - QEMU. https://wiki.qemu.org/Features/tcg-multithread. Last Accessed: January 9, 2023.

[7] Fork of llvm adding cheri support. https://github.com/CTSRD-CHERI/llvm-project. Last Accessed: January 11, 2023.

[8] FreeBSD adapted for CHERI-RISC-V and Arm Morello. https://github.com/CTSRD-CHERI/cheribsd. Last Accessed: January 11, 2023.

[9] Google Cloud Composer. fully managed workflow orchestration service built on Apache Airflow. https://cloud.google.com/composer. Last Accessed: January 9, 2023.

[10] Google Cloud Functions. Run your code in the cloud with no servers or containers to manage with our scalable, pay-as-you-go functions as a service (FaaS) product. https://cloud.google.com/functions. Last Accessed: January 9, 2023.

[11] IBM Composer. Composer is a new programming model for composing cloud functions built on Apache OpenWhisk. https://github.com/ibm-functions/composer. Last Accessed: January 9, 2023.

[12] Intravisor. Virtualisation platform using CHERI for isolation and sharing. https://github.com/lsds/intravisor. Last Accessed: January 9, 2023.

[13] NanoVMs. Run your applications faster, safer and with less cost using the new generation of cloud infrastructure. https://nanovms.com/. Last Accessed: January 9, 2023.

[14] QEMU with support for CHERI. https://github.com/CTSRD-CHERI/qemu. Last Accessed: January 11, 2023.

[15] A secure container runtime with oci interface. https://github.com/QuarkContainer/Quark. Last Accessed: January 4, 2023.

[16] StratoVirt. Next generation virtualization platform for all scenarios. https://gitee.com/openeuler/stratovirt. Last Accessed: January 4, 2023.

[17] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: towards high-performance serverless computing. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 923–935. USENIX Association, 2018.

[18] Delwar Alam, Moniruz Zaman, Tanjila Farah, Rummana Rahman, and M. Shazzad Hosain. Study of the dirty copy on write, a linux kernel memory allocation vulnerability. In *2017 International Conference on Consumer Electronics and Devices (ICCED)*, pages 40–45, 2017.

[19] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In Christine Cuicchi, Irene Qualters, and William T. Kramer, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 69. IEEE/ACM, 2020.

[20] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, pages 263–274. ACM, 2018.

[21] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 164–177. ACM, 2003.

[22] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 41–46. USENIX, 2005.

[23] Anirban Bhattacharjee, Yogesh D. Barve, Shweta Khare, Shunxing Bao, Aniruddha Gokhale, and Thomas Damiano. Stratum: A serverless framework for the lifecycle management of machine learning-based data analytics tasks. In Bharath Ramsundar and Nisha Talagala, editors, *2019 USENIX Conference on Operational Machine Learning, OpML 2019, Santa Clara, CA, USA, May 20, 2019*, pages 59–61. USENIX Association, 2019.

[24] Stella Bitchebe, Djob Mvondo, Alain Tchana, Laurent Réveillère, and Noël De Palma. Intel page modification logging, a hardware virtualization feature: study and improvement for virtual machine working set estimation. *CoRR*, abs/2001.09991, 2020.

[25] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E. Engelstad, and Kyrre M. Begnum. Includeos: A minimal, resource efficient unikernel for cloud services. In *7th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2015, Vancouver, BC, Canada, November 30 - December 3, 2015*, pages 250–257. IEEE Computer Society, 2015.

[26] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: skip redundant paths to make serverless fast. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 32:1–32:15. ACM, 2020.

[27] David Chisnall, Colin Rothwell, Robert N. M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the PDP-11: architectural support for a memory-safe C abstract machine. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, pages 117–130. ACM, 2015.

[28] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey D. Son, and Jonathan Woodruff. Cheriabi: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C runtime environment. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 379–393. ACM, 2019.

[29] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: lambdas on a diet. In Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi, editors, *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, pages 45–59. ACM, 2020.

[30] Lawrence Esswood. *CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor*. PhD thesis, University of Cambridge, 2021.

[31] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey D. Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia: Temporal safety for CHERI heaps. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 608–625. IEEE, 2020.

[32] Philipp Gackstatter, Pantelis A. Frangoudis, and Schahram Dustdar. Pushing serverless to the edge with webassembly runtimes. In *22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2022, Taormina, Italy, May 16-19, 2022*, pages 140–149. IEEE, 2022.

[33] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Sledge: a serverless-first, light-weight wasm runtime for the edge. In Dilma Da Silva and Rüdiger Kapitza, editors, *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*, pages 265–279. ACM, 2020.

[34] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with webassembly. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 185–200. ACM, 2017.

[35] Nastaran Hajinazar, Pratyush Patel, Minesh Patel, Konstantinos Kanellopoulos, Saugata Ghose, Rachata Ausavarungnirun, Geraldo F. Oliveira, Jonathan Appavoo, Vivek Seshadri, and Onur Mutlu. The virtual

block interface: A flexible alternative to the conventional virtual memory framework. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*, pages 1050–1063. IEEE, 2020.

[36] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In Olaf Landsiedel and Klara Nahrstedt, editors, *Proceedings of the International Conference on Internet of Things Design and Implementation, IoTDI 2019, Montreal, QC, Canada, April 15-18, 2019*, pages 225–236. ACM, 2019.

[37] Wei Huang, Matthew J. Koop, Qi Gao, and Dhabaleswar K. Panda. Virtual machine aware communication libraries for high performance computing. In Becky Verastegui, editor, *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007, November 10-16, 2007, Reno, Nevada, USA*, page 9. ACM Press, 2007.

[38] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 691–707. ACM, 2021.

[39] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.

[40] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. Osv - optimizing the operating system for virtual machines. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 61–72. USENIX Association, 2014.

[41] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 427–444. USENIX Association, 2018.

[42] Simon Kuenzer, Vlad-Andrei Badoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Stefan Teodorescu, Costi Raducanu, Cristian Banu, Laurent Mathy, Razvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: fast, specialized unikernels the easy way. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 376–394. ACM, 2021.

[43] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, David J. Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian M. Leslie. Jitsu: Just-in-time summoning of unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 559–573. USENIX Association, 2015.

[44] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: library operating systems for the cloud. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, pages 461–472. ACM, 2013.

[45] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: application-aware data passing for chained serverless applications. In Irina Calciu and Geoff Kuenning, editors, *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 285–301. USENIX Association, 2021.

[46] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 218–233. ACM, 2017.

[47] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the depths of C: elaborating the de facto standards. In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 1–15. ACM, 2016.

[48] Chetankumar Mistry, Bogdan Stelea, Vijay Kumar, and Thomas F. J.-M. Pasquier. Demonstrating the practicality of unikernels to build a serverless platform at the edge. In *12th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2020, Bangkok, Thailand, December 14-17, 2020*, pages 25–32. IEEE, 2020.

[49] Hamid Reza Mohebbi, Omid Kashefi, and Mohsen Sharifi. ZIVM: A zero-copy inter-vm communication mecha-

nism for cloud computing. *Comput. Inf. Sci.*, 4(6):18–27, 2011.

[50] Maziyar Nazari, Sepideh Goodarzy, Eric Keller, Eric Rozner, and Shivakant Mishra. Optimizing and extending serverless platforms: A survey. In *Eighth International Conference on Software Defined Systems, SDS 2021, Gandia, Spain, December 6-9, 2021*, pages 1–8. IEEE, 2021.

[51] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3), 2006.

[52] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony C. J. Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1003–1020. IEEE, 2020.

[53] Alfonso Pérez, Sebastián Risco, Diana M. Naranjo, Miguel Caballer, and Germán Moltó. On-premises serverless computing for event-driven data processing applications. In Elisa Bertino, Carl K. Chang, Peter Chen, Ernesto Damiani, Michael Goul, and Katsunori Oyama, editors, *12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019*, pages 414–421. IEEE, 2019.

[54] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In Jay R. Lorch and Minlan Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 193–206. USENIX Association, 2019.

[55] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. Lkl: The linux kernel library. In *9th RoEduNet IEEE International Conference*, pages 328–333. IEEE, 2010.

[56] Alessandro Randazzo and Ilenia Tinnirello. Kata containers: An emerging architecture for enabling MEC services in fast and secure way. In Mohammad A. Alsmirat and Yaser Jararweh, editors, *Sixth International Conference on Internet of Things: Systems, Management and Security, IOTSMS 2019, Granada, Spain, October 22-25, 2019*, pages 209–214. IEEE, 2019.

[57] Yi Ren, Ling Liu, Xiaojian Liu, Jinzhu Kong, Huadong Dai, Qingbo Wu, and Yuan Li. A fast and transparent communication protocol for co-resident virtual machines. In Calton Pu, James Joshi, and Surya Nepal, editors, *8th International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom 2012, Pittsburgh, PA, USA, October 14-17, 2012*, pages 70–79. ICST / IEEE, 2012.

[58] Yi Ren, Ling Liu, Qi Zhang, Qingbo Wu, Jianbo Guan, Jinzhu Kong, Huadong Dai, and Lisong Shao. Shared-memory optimizations for inter-virtual-machine communication. *ACM Comput. Surv.*, 48(4):49:1–49:42, 2016.

[59] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In Carlo Curino, Georgia Koutrika, and Ravi Netravali, editors, *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, pages 1–17. ACM, 2021.

[60] Andrea Sabbioni, Lorenzo Rosa, Armir Bujari, Luca Foschini, and Antonio Corradi. A shared memory approach for function chaining in serverless platforms. In *IEEE Symposium on Computers and Communications, ISCC 2021, Athens, Greece, September 5-8, 2021*, pages 1–6. IEEE, 2021.

[61] Vasily A. Sartakov, Lluís Vilanova, David M. Eyers, Takahiro Shinagawa, and Peter R. Pietzuch. Cap-vms: Capability-based isolation and sharing in the cloud. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 597–612. USENIX Association, 2022.

[62] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless computing: a survey of opportunities, challenges, and applications. *ACM Computing Surveys*, 54(11s):1–32, 2022.

[63] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 121–135. ACM, 2019.

[64] Simon Shillaker and Peter R. Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless

computing. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 419–433. USENIX Association, 2020.

[65] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M. Faleiro, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *CoRR*, abs/2001.04592, 2020.

[66] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, page 203–216, New York, NY, USA, 1993. Association for Computing Machinery.

[67] Hao Wang, Di Niu, and Baochun Li. Distributed machine learning with a serverless architecture. In *2019 IEEE Conference on Computer Communications, INFOCOM 2019, Paris, France, April 29 - May 2, 2019*, pages 1288–1296. IEEE, 2019.

[68] Jian Wang, Kwame-Lante Wright, and Kartik Gopalan. Xenloop: A transparent high performance inter-vm network loopback. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, page 109–118, New York, NY, USA, 2008. Association for Computing Machinery.

[69] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. The risc-v instruction set manual. *Volume I: User-Level ISA', version*, 2, 2014.

[70] Robert N. M. Watson, Robert M. Norton, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, Nirav H. Dave, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Ed Maste, Steven J. Murdoch, Colin Rothwell, Stacey D. Son, and Munraj Vadera. Fast protection-domain crossing in the CHERI capability-system architecture. *IEEE Micro*, 36(5):38–49, 2016.

[71] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 20–37. IEEE Computer Society, 2015.

[72] Sebastian Werner, Jörn Kuhlenkamp, Markus Klems, Johannes Müller, and Stefan Tai. Serverless big data processing using matrix multiplication as example. In Naoki Abe, Huan Liu, Calton Pu, Xiaohua Hu, Nesreen K. Ahmed, Mu Qiao, Yang Song, Donald Kossmann, Bing Liu, Kisung Lee, Jiliang Tang, Jingrui He, and Jeffrey S. Saltz, editors, *IEEE International Conference on Big Data (IEEE BigData 2018), Seattle, WA, USA, December 10-13, 2018*, pages 358–365. IEEE, 2018.

[73] Sebastian Werner and Stefan Tai. Application-platform co-design for serverless data processing. In Hakim Hacid, Odej Kao, Massimo Mecella, Naouel Moha, and Hye-young Paik, editors, *Service-Oriented Computing - 19th International Conference, ICSOC 2021, Virtual Event, November 22-25, 2021, Proceedings*, volume 13121 of *Lecture Notes in Computer Science*, pages 627–640. Springer, 2021.

[74] Dan Williams and Ricardo Koller. Unikernel monitors: Extending minimalism outside of the box. In Austin Clements and Tyson Condie, editors, *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016*. USENIX Association, 2016.

[75] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 457–468. IEEE Computer Society, 2014.

[76] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The true cost of containing: A gVisor case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[77] Miao Zhang, Fangxin Wang, Yifei Zhu, Jiangchuan Liu, and Zhi Wang. Towards cloud-edge collaborative online video analytics with fine-grained serverless pipelines. In Özgü Alay, Cheng-Hsin Hsu, and Ali C. Begen, editors, *MMSys '21: 12th ACM Multimedia Systems Conference, Istanbul, Turkey, 28 September 2021 - 1 October 2021*, pages 80–93. ACM, 2021.

[78] Qi Zhang and Ling Liu. Shared memory optimization in virtualized cloud. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 261–268, 2015.