Single-Address-Space FaaS with Jord

Yuanlong Li
EcoCloud, EPFL
Lausanne, Switzerland
yuanlong.li@epfl.ch

Atri Bhattacharyya
EcoCloud, EPFL
Lausanne, Switzerland
atrib.webmail@gmail.com

Madhur Kumar
EcoCloud, EPFL
Lausanne, Switzerland
madhur.kumar@epfl.ch

Abhishek Bhattacharjee
Yale University
New Haven, USA
abhishek.bhattacharjee@yale.edu

Yoav Etsion
Technion
Haifa, Israel
yetsion@technion.ac.il

Babak Falsafi
EcoCloud, EPFL
Lausanne, Switzerland
babak.falsafi@epfl.ch

Sanidhya Kashyap
EcoCloud, EPFL
Lausanne, Switzerland
sanidhya.kashyap@epfl.ch

Mathias Payer
EcoCloud, EPFL
Lausanne, Switzerland
mathias.payer@nebelwelt.net

Abstract

Function-as-a-Service (FaaS) has emerged as a popular cloud paradigm that simplifies software development and deployment by providing scalable and event-driven function execution without the burden of managing servers. FaaS was originally created with a *function as a function* semantics to enable standalone microservices with adequately short execution time to meet microsecond-scale service-level objectives (SLOs). Unfortunately, today's FaaS systems fundamentally suffer from millisecond-level performance bottlenecks that arise from isolating functions in separate address spaces inside containers or microVMs. Prior work has focused on optimizing FaaS performance, but these systems still fall short of meeting microsecond-level SLOs. In this paper, we present *Jord*, a FaaS system that revives the original function-as-a-function vision of FaaS. Jord leverages hardware/software co-design to colocate functions in a single address space with user-level in-process memory isolation, extending the capability of traditional virtual memory. By performing memory isolation and management in nanoseconds, Jord enables zero-copy cross-function communication and scalable function dispatch, thereby minimizing FaaS overheads. We demonstrate that Jord can meet microsecond-level SLOs for microservice workloads while performing within 16% of an idealized but insecure baseline and delivering over 2× higher throughput compared to enhanced state-of-the-art systems.

CCS Concepts

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Virtual memory**; • **Security and privacy → Hardware security implementation**.

Keywords

Function-as-a-Service, Single-Address-Space, Virtual Memory

ACM Reference Format:
Yuanlong Li, Atri Bhattacharyya, Madhur Kumar, Abhishek Bhattacharjee, Yoav Etsion, Babak Falsafi, Sanidhya Kashyap, and Mathias Payer. 2025. Single-Address-Space FaaS with Jord. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3695053.3731108

1 Introduction

The rise of cloud computing has transformed how applications are built and deployed, with Function-as-a-Service (FaaS) [3, 5, 53, 57, 75, 76] emerging as a particularly compelling cloud paradigm. FaaS abstracts away infrastructure management and provides users a pay-as-you-go billing model [4]. With such benefits, developers can focus purely on business logic implemented as single-purpose and short-running functions [25] without worrying about the details of deployment and autoscaling [2].

The original vision of FaaS was to enable running a *function as a function* [41] without managing servers. With this vision, programmers build standalone microservices with adequately short-running functions and run them on FaaS systems in the hope of meeting microsecond-level SLOs [35, 74]. Unfortunately, today's FaaS systems fall short of realizing this vision, because cloud providers favor security over performance when building FaaS as a public cloud service that must enforce strong isolation among functions coming from various third parties [67, 82].

The multi-address-space design adopted by today's FaaS systems for security and isolation—i.e., wrapping each function into its own sandbox, such as a container [77] or microVM [1]—fundamentally hinders performance. First, to invoke a function, a FaaS system must perform mediated communication [5, 53] to schedule the execution of the sandbox, which involves multiple messages passed through the address-space boundary of multiple FaaS processes, adding control-flow overheads to the invocation. Second, to share data with another function, a function must rely on indirect and slow communication channels [3, 5, 10, 38, 50, 55, 76], which incurs

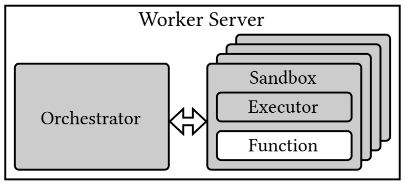

Figure 1: A worker server in a traditional FaaS system.

data-flow overheads. Finally, sandbox initialization can consume a substantial amount of time [40], adding extra latencies to each function invocation. These millisecond-level overheads effectively render today's FaaS systems completely unusable for latency-critical microservices running in microseconds [25].

Prior work has focused on mitigating these bottlenecks. Solutions based on shared memory optimize intra-server data transfer [35, 40, 45, 50, 56, 60, 68] but often sacrifice isolation guarantees or introduce OS-level inter-process communication overhead. Sophisticated cold start mitigations [9, 21, 39, 44, 51, 63, 64, 67, 82, 88] largely reduce the sandbox startup latency, yet the resulting sandbox initialization still requires milliseconds to finish—an unavoidable consequence of address-space-based isolation. In essence, optimizing software alone is not sufficient to realize FaaS's vision at the microsecond timescale.

The insufficiency of software-only solutions prompts us to rethink how to build FaaS systems from the perspective of hardware-software co-design. On one hand, while FaaS systems must provide scalability and isolation, achieving these goals does not inherently demand separate processes or sandboxes. On the other hand, when security is not a concern, traditional monolithic applications can achieve high performance with direct function calls and pointer-based data sharing. These two observations lead us to consider a new way of combining the operational benefits of FaaS with the performance benefits of monolithic applications. In such a system, functions can easily invoke other functions and share data with them without crossing address space boundaries. In contrast, each function is treated as a standalone schedulable entity that can be autoscaled transparently. The key to achieving this combination is efficient in-process memory isolation—a new hardware mechanism that exceeds the capability of traditional virtual memory systems and forgoes syscalls, page table manipulation, and TLB shootdown [7, 8, 47, 71, 90] that take tens to even thousands of microseconds to complete.

In this paper, we present *Jord*, a novel single-address-space FaaS system that revives the original vision of FaaS—running a function as a function without managing servers. Rather than relying on separate address spaces, Jord enforces strong isolation among functions using in-process protection domains [16, 85] with hardware-software co-design. Jord follows the trend of implementing virtual memory areas (VMAs) [13, 28], the software abstraction modern OSes use to manage virtual memory, in hardware, thereby extending the capability of traditional page-based translation. By virtue of the co-design, Jord operates entirely at the user level while performing memory isolation and management in nanoseconds. As such, Jord enables zero-copy cross-function communication and scalable function dispatch for FaaS, fully leveraging the performance benefits of a single-address-space design.

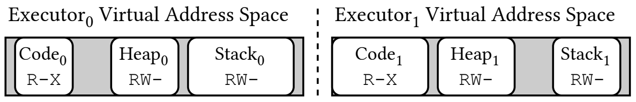
Executor$_0$ Virtual Address Space
Code$_0$
R-X
Heap$_0$
RW-
Stack$_0$
RW-
Executor$_1$ Virtual Address Space
Code$_1$
R-X
Heap$_1$
RW-
Stack$_1$
RW-

Figure 2: Isolated virtual address spaces and virtual memory areas (VMAs) in a traditional FaaS system.

We implement Jord on both QFlex, a family of full-system simulators [59], and OpenXiangShan [86], a configurable RISC-V out-of-order core on FPGA as an RTL-level proof of concept. Our results indicate that Jord can meet microsecond-level SLOs for microservice workloads. Furthermore, it performs within 16% of a baseline single-address-space system with no isolation, while outperforming enhanced state-of-the-art FaaS systems by over 2×.

We make the following contributions in this paper:

- We introduce Jord, a novel single-address-space FaaS system that revives the original vision of FaaS. Jord achieves microsecond-scale function execution enabled by zero-copy communication through shared memory and scalable function dispatch.
- We develop a hardware-software co-designed in-process memory isolation mechanism that provides nanosecond-scale memory and protection domain management.
- We demonstrate Jord's performance and scalability with the RISC-V ISA on a full-system simulator and FPGA running Linux and microservice workloads.

2 Background

This section first examines the architectural decisions and limitations of traditional multi-address-space FaaS systems. It then presents the challenges and opportunities of building a FaaS system using a single address space.

2.1 Traditional FaaS Systems

Figure 1 illustrates the basic anatomy of a worker server in a traditional FaaS system [3, 22, 53, 57, 75, 76]. The worker server runs an *orchestrator* that coordinates the execution of multiple *sandboxes*—i.e., isolated execution environments such as containers [77] and microVMs [1]. Inside each sandbox, an *executor* initializes the runtime environment for a function and executes that function. For each function invocation request, the orchestrator performs function dispatch to select a sandbox and relay the request to it. In this design, each orchestrator and executor runs in a separate virtual address space, and each virtual address space is further split by the OS into large contiguous regions representing various logical sections (e.g., code, heap, and stack) as virtual memory areas (VMAs), as shown in Figure 2. While this design enforces security through strong isolation among address spaces, it introduces substantial function invocation overhead, manifested in both the control and data flows, further compounded with sandbox initialization.

The primary source of control-flow overhead in traditional FaaS systems is function dispatch. Each function dispatch triggers multiple control messages among orchestrator processes to schedule the sandbox execution, while each message needs to traverse through multiple address space boundaries to finish. For example, both AWS Step Functions [5] and Azure Logic Apps [53] require each function to return control to the orchestrator before starting the subsequent function in the workflow. This orchestrator-mediated dispatch incurs multiple IPC roundtrips, usually taking more than 10 *ms* [46, 89, 91] per invocation.

In contrast, data-flow overhead in traditional FaaS systems is mainly incurred while copying data between two sandboxes. Traditional FaaS systems rely on indirect communication channels such as platform-specific messaging primitives [5, 49, 76], standalone message queues [10, 55], remote storage [3, 38, 46, 48, 50], or RESTful APIs [3]. These mechanisms require data serialization/deserialization [49], memory copying, and often network stack traversal, adding tens or even hundreds of milliseconds of additional latency and accounting for up to 70% of the total function execution time [40, 46, 48, 50].

There is much prior work focusing on optimizing communication by leveraging shared memory for intra-server communications [35, 40, 45, 50, 56, 60, 68], adopting data-centric workflows [46, 89], using fast network transports (e.g., RDMA) [49], or offloading data transfer to DPUs [48]. Unfortunately, shared memory techniques either compromise isolation or incur OS-level overheads of memory copying and process synchronization. Likewise, network-based techniques also suffer from memory copying and still incur several microseconds of latency at best with hardware acceleration.

Apart from communication, to start executing a function, the FaaS system must also pull the sandbox disk image from the registry, configure the sandbox, fork multiple processes to boot the sandbox, and launch the executor for language or library-level initialization. The entire startup sequence easily takes tens or even hundreds of milliseconds to finish, increasing overall function execution time by up to 95% [40]. There is a myriad of work dedicated to addressing the so-called cold start problem, including avoiding sandbox images from being swapped out [23, 64, 67], compressing [63, 82], fragmenting [15, 88], fusing [44, 51, 65], or checkpointing [9, 21, 39, 69]. While these optimizations dramatically decrease the sandbox startup latency, they still fall short of reducing the initialization overhead from milliseconds to microseconds.

Overall, the function execution overhead in traditional FaaS systems amounts to tens of milliseconds, essentially rendering these traditional FaaS systems completely unusable for latency-critical workloads, such as microservices [25], that heavily rely on nested function invocations with microsecond-level execution time.

2.2 The Case for Single-Address-Space FaaS

The order-of-magnitude disparity between microsecond-level function execution time and millisecond-level system overhead calls for rethinking of the design space for a FaaS system. In essence, from a pure functionality perspective, what users expect from the system is only the ability to execute their functions while managing the deployment and scaling automatically. Moreover, as a public cloud service, a FaaS system should also provide isolation during function execution for data confidentiality and integrity. Most notably, these bare minimum system requirements do not mandate the use of containers or microVMs.

This observation inspires us to consider an alternative system architecture: a FaaS system with a single address space as the runtime environment. In such a design, a function can invoke another function through direct function calls without address space management overhead incurred by the OS. Data sharing among functions also becomes drastically simpler as zero-copy pointer-based memory sharing [14, 41].

Adopting a single-address-space design, however, does not mean reverting to a monolith software architecture. Users can still write loosely coupled functions with clearly defined interfaces without explicitly chaining them with function calls. The FaaS system regards functions as standalone entities and scales them independently [26].

The primary challenge in realizing a practical single-address-space FaaS is enforcing memory isolation among functions. As each function can hold different permissions for various VMAs in the address space, the system must change VMA permissions accordingly when switching from one function to another. Unfortunately, updating VMA permissions by changing page permissions in page-based virtual memory is prohibitively slow because it involves multiple syscalls, traversal and modification of the page table, and TLB shootdowns, each of which can take tens to thousands of microseconds to complete [7, 8, 47, 71, 90]. The same overhead is also incurred when a function allocates or deallocates memory. Such overhead is hard to mitigate as only the OS can access the page table, which is inherently not scalable.

Memory protection key (MPK)-based approaches [58, 66, 78] or HFI [54] create another layer of protection on top of virtual memory. However, MPK-based approaches are not scalable as protection keys must be embedded into page table entries using reserved bits, which are highly limited in number (especially with the adoption of five-level paging [32]). Moreover, for both MPK-based approaches and HFI, this extra layer of protection only applies to a single core. They must rely on extra software modules to ensure the protection is consistent among all cores. Furthermore, they do not help in memory allocation/deallocation, as those operations are still provided by the page-based virtual memory.

Rather than relying on hardware, in-process memory isolation can also be provided at the language level through software fault isolation (SFI) [24, 68] or a single-address-space OS [16, 30, 31]. Unfortunately, SFI restricts the choice of language for software construction and can incur up to 40% runtime overhead due to frequent instrumentation and boundary checks [34]. Similarly, single-address-space OSes either assume software construction with safe languages or rely on virtual memory for memory isolation.

We argue that with a secure and fast memory isolation mechanism, a single-address-space FaaS can enjoy the best of both worlds: it can not only perform function invocation and zero-copy communication without involving the OS, but also allow the system to autoscale and functions to be deployed seamlessly as if they were written for a traditional FaaS system. Such a system can also be built with minimal modification to a CPU and OS without functional interference with existing workloads.

3 Jord's System Architecture

Jord follows four design principles to minimize function invocation overhead: 1) leveraging a *single address space* to enable zero-copy communication and scalable function dispatch, 2) maintaining strong isolation among functions through user-level in-process

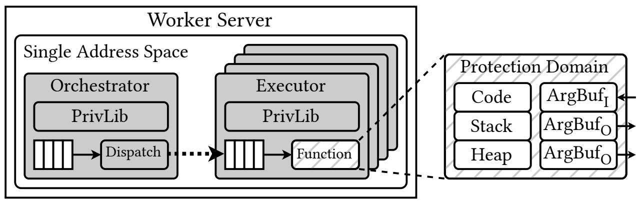

Figure 3: A worker server in Jord.

protection domains, 3) implementing isolation through nanosecond-scale VMA operations with hardware-software co-design, and 4) extending instead of replacing the traditional virtual memory to minimize the impact to existing hardware and software.

The key innovation in Jord lies in its in-process memory isolation [13, 85]. Rather than relying on multiple address spaces, Jord introduces in-process protection domains (PDs) [16] to provide each function with an isolated execution environment while maintaining performance benefits of a single address space—i.e., sharing data through memory and calling functions directly. In contrast to utilizing the OS for memory management, Jord manages memory and isolation by a trusted user-level *privileged library* (PrivLib) [66, 70] and leverages VMAs to achieve scalable address translation and permission check with nanosecond-level overhead.

Figure 3 depicts the anatomy of a worker server in Jord, with *orchestrator* and *executor* threads [1] running inside a single address space with zero-copy communication. A worker server runs one or more orchestrators that receive function invocation requests and distribute them to multiple executors with load balancing. Each executor, in turn, executes functions in isolated PDs. Function invocation requests are passed among an orchestrator and the executors it manages in *argument buffers* (ArgBufs). Each ArgBuf uses an individual VMA for address translation and access control.

Figure 4 illustrates a simplified flow of a function invocation request in Jord. When a request arrives, it enters an orchestrator's request queue and is dispatched to an executor based on a load-balancing policy. The executor then creates a new PD with a private stack and heap, makes the function code VMA accessible, and transfers the ArgBuf permissions to the PD. Next, the executor enters into the PD to execute the function. After it finishes, the executor transfers back the ArgBuf permissions, revokes the code VMA access, destroys the PD, dequeues the request, and notifies the orchestrator of the completion. Throughout the flow, the two key overheads are the dispatch overhead incurred by the orchestrator dispatching the request and the memory isolation overhead incurred by the executor managing PDs and VMAs.

3.1 Programming and Threat Models

Jord provides users with a sequential programming model similar to those in traditional FaaS systems. Listing 1 presents a synthetic function demonstrating function invocations, zero-copy communication, and VMA management in Jord. In this example, `SrcFunc` has an input ArgBuf pointer `req` as the only argument to retrieve inputs and drive outputs. To invoke other functions, in our case `Tgt1` and `Tgt2`, `SrcFunc` first creates two output ArgBufs with the desired types (line 3, 4) and populates them (line 6, 7). Then, the function hands ArgBufs along with target function IDs to Jord by calling `call` or `async` for synchronous (line 13) or asynchronous invocation (line 9) respectively. The synchronous invocation only returns when the target function finishes, while the asynchronous invocation immediately returns a cookie that the function can wait on (line 16). The function can also call POSIX-compatible PrivLib APIs to allocate and deallocate VMAs (line 19, 21) dynamically. Behind the scenes, Jord handles function invocation and enforces memory isolation transparently without user intervention.

Jord's threat model is centered around memory isolation among PDs. The system allows attackers to forge arbitrary memory addresses and access them through load/store instructions or code execution. The attackers can also arbitrarily call PrivLib. Jord enforces isolation by generating a hardware fault whenever untrusted code reads, writes, or executes a memory address that is either not mapped by a VMA or whose VMA does not have appropriate access permissions in the PD where the code executes. Much like traditional FaaS systems, Jord's sphere of protection is limited to memory isolation. More specifically, Jord cannot protect functions from corrupting their own data or defend against data corruption attacks through the syscall interface [18].

3.2 Memory Isolation

Jord achieves in-process memory isolation through PDs with the following security and functionality requirements. To enforce isolation, any untrusted code executing inside a PD can only access memory for which the PD has the appropriate permissions. To avoid PDs from being bypassed or destroyed by untrusted code, PDs themselves must be protected with special privileges. Furthermore, to support concurrent function execution, PDs should be able to be switched from one to another during runtime.

To meet these requirements, Jord introduces PrivLib to manage both PDs and their underlying VMAs with a set of OS-agnostic APIs as shown in Table 1. At the lower level, PrivLib leverages VMAs with per-PD permissions to create the abstraction of isolated PDs through VMA management APIs. On top of that, PrivLib additionally supports PD scheduling through PD management APIs. Even though the PrivLib APIs can be implemented with other in-process memory isolation approaches [13, 47, 66, 70, 85], Jord's PrivLib fully leverages the underlying hardware to make these APIs work on the nanosecond scale, minimizing their performance impact on functions with microsecond-level execution time.

PrivLib implements POSIX-compatible VMA operations to minimize programming complexity. The user code can call `mmap` to allocate a new VMA of a given size and permission into the current PD. It can further call `munmap` or `mprotect` to deallocate/resize or change the permission of an existing VMA. PrivLib also provides `pmove` and `pcopy` to atomically transfer or duplicate VMA permissions between the current PD and another PD. To maintain the integrity of PD configurations, PrivLib stores them within a special VMA that only PrivLib itself can access.

PrivLib manages the lifecycle and scheduling of PDs. PrivLib provides `cget` and `cput` to allocate and destroy them. The executor can call `ccall` to switch into one PD. When a function suspends itself due to a nested function invocation, it calls `cexit` to save

[1]For brevity, we refer to these threads as orchestrators and executors.

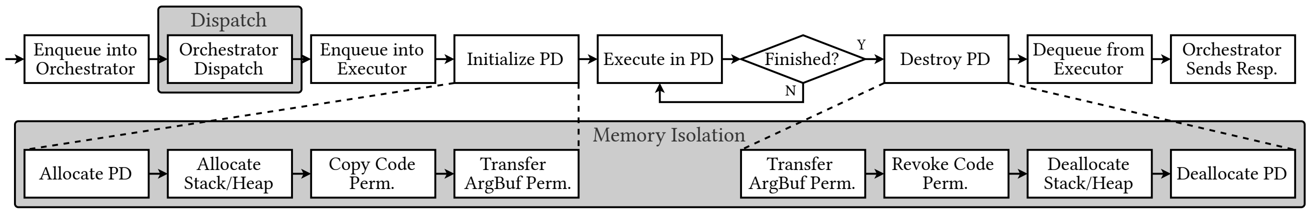

Figure 4: A simplified flow of how Jord handles a function invocation request.

```
int SrcFunc(SrcReq *req) {
  // each argBuf uses its own VMA
  jord::argBuf<Tgt1Req> r1;
  jord::argBuf<Tgt2Req> r2;
  // read inputs and populate argBufs
  r1->in = pre(req->in1);
  r2->in = pre(req->in2);
  // async call returns a cookie
  int c = jord::async(Tgt1, r1));
  // return value
  int r = 0;
  // sync call returns non-zero on failure
  if ((r = jord::call(Tgt2, r2)))
    return r;
  // wait on the cookie, return non-zero on failure
  if ((r = jord::wait(c)))
    return r;
  // allocate a new VMA
  void *buf = mmap(0, 0x1000, PROT_RW, 0, 0, 0);
  // generate the output
  req->out = post(buf, r1->out, r2->out);
  // deallocate the VMA
  munmap(buf, 0x1000);
  // no errors
  return 0;
}
```

Listing 1: A synthetic function in Jord.

Table 1: PrivLib APIs.

API function	Description
`mmap(addr, len, prot, flags, fildes, off)`	Allocate a new VMA
`munmap(addr, len)`	Deallocate a VMA
`mprotect(addr, len, prot)`	Change the permission of a VMA
`pmove(addr, cid, prot)`	Move the permission of a VMA to another PD
`pcopy(addr, cid, prot)`	Copy the permission of a VMA to another PD
`cget()`	Create a new PD
`cput(cid)`	Destroy a PD
`ccall(cid, func, args)`	Call into (and return from) a PD
`center(cid)`	Resume a PD
`cexit()`	Suspend a PD

its current state and switch back to the executor, which later calls `center` to resume the suspended PD.

The security and integrity of PrivLib is crucial to Jord. PrivLib enforces mandatory security policy checks on all APIs to prevent illegal or malicious calls. These checks are enforced through control-flow integrity (CFI) on top of the underlying hardware mechanisms.

3.3 Orchestrator

An orchestrator thread in Jord is conceptually similar to an orchestrator process in a traditional FaaS system. It receives external function requests through the network and dispatches them to executors. In Jord, orchestrators save these requests into ArgBufs to leverage the zero-copy communication inside the single address space. Orchestrators are pinned on separate CPU cores to avoid OS scheduling overhead.

Orchestrators require a dispatch policy to balance the load among executors while minimizing overhead. In this paper, without loss of generality, we use the Join-Bounded-Shortest-Queue (JBSQ) policy for load balancing inspired by state-of-the-art key-value stores [20, 33, 36]. This push-based policy requires each orchestrator to iterate over all executors it manages and push the request to the one with the fewest requests in its request queue. The dispatch overhead scales with the number of executors an orchestrator manages and varies with system size (i.e., the number of cores in a coherence domain and their organization in chips/chiplets). As such, a worker server employs multiple orchestrators, each managing a group of executors in proximity to limit dispatch overhead. We evaluate the impact of system size on dispatch overhead, but a further evaluation of dispatch policies is beyond the scope of this paper.

To support nested function invocation, each orchestrator also handles internal invocation requests generated from executing functions. These internal requests require special handling to prevent deadlocks, as external requests can fill the executor queues and block internal requests from executing. To ensure forward progress, the orchestrator maintains two separate request queues for internal and external requests and dispatches external requests only when the internal queue is empty. For internal requests that cannot be served on the current worker server, the orchestrator sends them through the network to find another worker server for execution.

3.4 Executor

An executor thread in Jord is also conceptually similar to an executor process in a traditional FaaS system. In Jord, however, an executor thread can place multiple functions inside their own PDs and execute them in parallel as functions generate nested invocations and wait for them to finish. The executor regards each function as a *continuation* with private register states, stack, and heap inside the isolated PD, allowing each function to be suspended or resumed independently, similar to cooperative user-level threads [81]. As with an orchestrator, each executor is also pinned on a separate CPU core to avoid OS scheduling overhead.

After retrieving a function request from the request queue, an executor initializes the continuation and PD with a private stack and heap. It then calls `ccall` in PrivLib to perform a user-level context switch into the continuation. During the context switch, the current register states are saved into the executor's stack, while new register states are loaded from the function's private stack. Consequently, the function only sees the ArgBuf pointer, the private stack pointer, and a return address (inside `ccall`) from the register values. As such, no extra information is leaked to the function.

The control flow automatically switches back to the executor when the function finishes, at which point the executor destroys the continuation and PD, dequeues the request, and continues to poll and execute new requests from the request queue. If the function suspends itself by calling `cexit` while waiting for a nested invocation to finish, the executor's context is also switched back, and the current state of the function is saved into the continuation. After the target invocation finishes, the executor can resume the suspended continuation by switching to it again using `center`.

4 Implementing Jord

Jord employs hardware-software co-design for nanosecond-scale in-process memory isolation without interfering with the traditional page-based virtual memory. In software, Jord manages virtual and physical address space regions reserved by the OS. Likewise, in hardware, Jord requires minimum microarchitectural modifications to provide a separate path for address translation as illustrated in Figure 5. In addition to the traditional TLB hierarchy, Jord introduces instruction and data virtual lookaside buffers (I/D-VLB), similar to those used in systems with intermediate address spaces [28, 29] to cache translations managed by Jord. A virtual table walker (VTW) handles I/D-VLB misses by traversing the translation table. Jord introduces a virtual translation directory (VTD) to track the sharing of translations among VLBs integrated together with the LLC.

In the following subsections, we present a Jord implementation with RISC-V [61] as the baseline ISA and describe its VMA management, VLB coherence, and PD management for nanosecond-scale in-process memory isolation.

4.1 VMA Management

The latency of VMA operations is critical to Jord's performance, as each function invocation requires multiple VMA allocations, deallocations, and permission transfers. Inspired by segregated list-based user-level heap allocators [43], we introduce *size classes* for VMA management without relying on complex data structures to manage free memory. Each size class represents a definite size

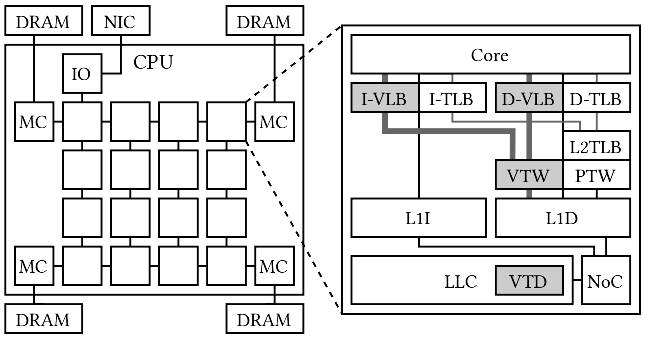

Figure 5: Microarchitectural modifications in Jord.

range to categorize VMA allocation requests, backed by a free list of memory chunks of that size. When allocating a new VMA, the allocator first calculates the size class of the allocation and then resorts to the corresponding free list to obtain a free memory chunk. Each VMA has the requested size, while the trailing part of the allocated memory chunk is reserved for future resizing. Each application can define its size classes. In our current implementation, we choose the size classes as all the power-of-two values between 128 bytes and 4 GB, as 99% of the VMAs in our target workloads are smaller than 1 KB [35, 42].

To achieve optimal VMA lookup performance in the translation table, hereafter referred to as the VMA table, without additional memory accesses, we encode the size class of a VMA into its base address [17, 29]. This encoding scheme results in a static partitioning of the virtual address space among size classes, as shown in Figure 6. Consequently, it also allows VMAs to be organized in the VMA table as a *plain list*, with definite positions determined by each VMA's size class and its index within the size class. More specifically, for a given VMA, the address of the VMA table entry (VTE) that stores the VMA metadata can be simply calculated as

$$A_{\mathrm{VTE}} = A_{\mathrm{Base}} + f(SC_{\mathrm{VMA}}, \mathrm{Index}_{\mathrm{VMA}})$$

where A_{Base} is the base address of the VMA table, SC_{VMA} and $\mathrm{Index}_{\mathrm{VMA}}$ are the size class and index of the VMA, and f is a simple two-input injective function that can be easily implemented in both software and hardware. Our current implementation uses a simple f that evenly interleaves VMAs of various size classes. Because of its simple structure, the plain list obviates the need to maintain two separate data structures for translations, as both software and hardware can use the same plain list concurrently.

We introduce two new user-level control and status registers (CSRs) into the RISC-V ISA, namely `uatp` (User Address Translation and Protection) and `uatc` (User Address Translation Configuration). `uatp` specifies the base address of the VMA table and whether the plain list-based address translation is enabled, while `uatc` defines the exact format of the VA encoding scheme as in Figure 6, such as the range of size classes, the position of size class bits, the value of `Top` bits, and the VMA table size.

Jord manages virtual and physical memory regions reserved by the OS. For virtual memory, when PrivLib is initializing, it asks the OS to reserve all VAs with the given `Top` bits according to the VA encoding scheme. Jord performs address translation for a VA only

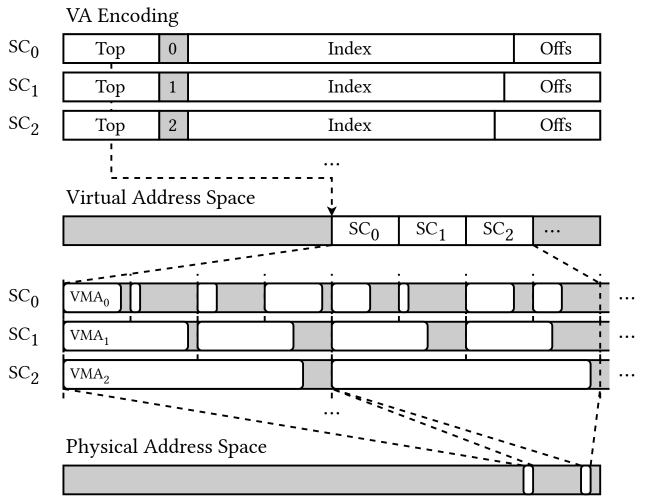

Figure 6: Size class-embedded VA encoding and partitioning of the virtual address space among size classes.

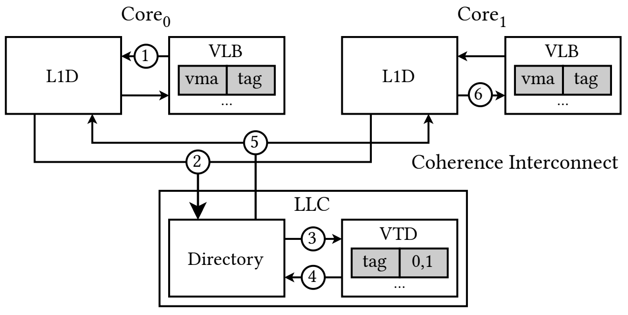

Figure 7: Hardware-based VLB shootdown in Jord.

if the process runs at the user level, the plain list-based translation is enabled in `uatp`, and the VA has the same `Top` bits as specified in `uatc`. The traditional page-based virtual memory still serves other VAs through the OS-managed page table.

Jord also relies on the OS to provide physical memory chunks to back VMAs. These chunks can be non-contiguous with various sizes, as shown in Figure 6. Instead of using a huge and contiguous physical memory for all VMAs, Jord only requires that each allocated VMA of size class *S* is backed by a contiguous physical memory chunk of at least *S* in size. For VMAs smaller than a page, Jord maps them into non-overlapping portions of a single physical page. To avoid interference with the traditional virtual memory, the OS must ensure that all physical memory chunks reserved for Jord are private to Jord and cannot be swapped out. The OS can either pin those physical memory chunks, or it only manages part of the physical memory, leaving the remaining to Jord.

At the microarchitectural level, we add instruction and data virtual lookaside buffers (I/D-VLBs) to cache recently used VMAs and a VMA table walker (VTW) to perform plain list traversal. The VLBs are range-based TLBs as in [12, 28], while the VTW is a simple finite state machine that calculates the address of the corresponding VTE and fetches it from the cache hierarchy.

This design introduces two main trade-offs. First, the plain list should be preallocated and overprovisioned to cover all VMAs in the process. However, given the moderate total number of VMAs [52], this overhead is acceptable, as a 64 MB VMA table can accommodate one million VMAs. Second, encoding the size class into VAs reduces the bits available for address space layout randomization (ASLR). In our implementation with 26 size classes, Jord only causes a modest 5-bit entropy reduction for ASLR, leaving 29 bits for randomization for the smallest 128-byte size class.

4.2 VLB Coherence

Jord frequently allocates, deallocates, and manipulates the permission of VMAs to maintain isolation among PDs. As one VMA can be shared by multiple functions running on multiple cores, Jord should also perform VLB shootdowns to invalidate stale translations from VLBs for coherence.

To match the frequency of VMA operations while minimizing silicon cost [62, 87] and modification to the existing cache coherence protocol and memory consistency model [6, 11], we implement VLB coherence in hardware managed by a virtual translation directory (VTD) [80]. VTD is a set-associative structure, co-located with the coherence directory inside each LLC slice. VTD tracks the sharing of VMAs using VTE addresses as a proxy, enabled by the one-to-one correspondence between VMAs and VTEs in the plain list design. As such, each VTD entry contains a tag from the corresponding VTE address and a sharer list for the translation.

We reuse the cache coherence protocol to piggyback VMA access information with a single-bit *Translation* (T) sideband signal. More specifically, both accesses to VMAs and the invalidation of VLB entries are performed through coherence messages with the T bit set. Similar to existing hardware TLB coherence approaches [62, 87], each VLB entry is extended with an extra tag from the corresponding VTE address to match against these coherence messages.

The core initiates VTE accesses through normal load/store instructions and VTW traversals. With the plain list design, the L1D can identify VTE accesses using the base address and size of the VMA table from the `uatp` and `uatc` registers and thus set the T bit for all coherence messages corresponding to VTE accesses. Figure 7 illustrates how coherence messages propagate among VTD and VLBs to implement VLB coherence. When a VTE read or write misses in the L1D ①, the L1D generates a new coherence message with the T bit set and sends it to the LLC and VTD through the coherence interconnect ②. For VTE reads, the VTD registers the translation by adding the initiator VLB into the sharer list of the tag-matched entry ③. For VTE writes, the VTD reads out the sharer list ④ from the tag-matched entry and generates invalidation messages with the T bit set to all sharer cores ⑤. Upon receiving these invalidation messages, the L1D forwards them to the VLB ⑥, which then performs a normal lookup and invalidates entries whose tag matches the address of the invalidation message. For a VTE write that hits in the L1D, a local VLB invalidation is generated without triggering any coherence traffic.

As the cache hierarchy, VLBs, and VTD evict cache blocks or translations independently, a corner case can occur when a VTE is in L1D, but the translation is absent in any VLB or VTD. In this case, the core can easily reinstall the translation into its VLB without informing VTD to track it. To fix this issue, when the VTD receives a coherence message, it retrieves the sharer list from the coherence directory and pessimistically regards all VTE sharers as translation sharers. Moreover, when the coherence directory evicts a cache block, it also pessimistically regards all VTE sharers as translation sharers if VTD does not track the corresponding translation. As such, the coherence directory behaves as a victim cache for the VTD, effectively increasing the VTD capacity.

4.3 PD Management

Jord allows multiple functions to execute in parallel inside multiple PDs. To manage this parallelism, we introduce a new user-level CSR `ucid` (User Continuation ID) to specify the currently executing continuation/PD. When the executor performs context switches among functions, it uses PrivLib to update `ucid` accordingly to assign each function the correct views to the shared memory.

To enable multiple PDs sharing the same VMA with different permissions, we extend each VTE with a sub-array of PD IDs and corresponding permissions, as shown in Figure 8. Each VTE spans an entire cache block to avoid false sharing. If the *Global* (G) bit in the `attr` fields is not set, the VTW considers the VTE as valid only when it finds a valid sub-array entry whose PD ID matches `ucid`. The VMA permission also comes from this entry rather than the `attr` field. In our implementation, the sub-array contains 20 entries to handle the common case of VMAs with up to 20 sharers. For rare cases with more sharers, the VTE contains an extra `ptr` field pointing to a complete list of PD IDs and permissions.

To protect the VMA table from being modified by untrusted code, we implement two complementary protection mechanisms. First, we introduce a *Privilege* (P) bit for each VMA to designate it as privileged. When the executing code attempts to access a privileged VMA through explicit load/store instructions, the hardware checks if the code itself is covered by a privileged VMA, similar to CODOMs [79]. The access proceeds normally only if the condition is met. Otherwise, the hardware generates a translation fault. This P bit protection also extends to the custom CSRs in Jord, effectively restricting privileged resource accesses to privileged code.

Second, to enforce control-flow integrity when entering PrivLib, we add a new `uatg` (User Address Translation Gate) instruction as a special call gate of privileged code. The hardware requires the first instruction when the control flow jumps from a non-privileged VMA to a privileged VMA to be `uatg`. Otherwise, the hardware generates an invalid instruction fault. Therefore, combining these mechanisms, the VMA table and PrivLib are protected by privileged VMAs, while untrusted code can only enter into PrivLib through pre-defined `uatg` call gates.

At the microarchitectural level, these protection mechanisms are implemented across multiple components. Each VLB entry caches the P bit obtained from the VTW. The I-VLB further attaches the P bit to each instruction, propagating through the pipeline to the decoder, load/store units, and finally to the reorder buffer. When the decoder determines that the current instruction is a CSR instruction that accesses `uatp`, `uatc`, or `ucid`, it allows the instruction to proceed only if it has the P bit set. Otherwise, the decoder marks the instruction as illegal. Similarly, when the D-VLB sees a load/store instruction targeting an address covered by a privileged VMA, it

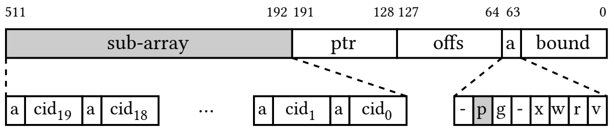

Figure 8: Structure of the VMA table entry.

also requires the instruction to have the P bit set. Otherwise, the D-VLB generates a translation fault to the instruction.

For control flow integrity, when the decoder sees a zero-to-one transition of the P bit in the instruction stream, it checks whether the first instruction with the P bit set is `uatg`. If not, the decoder marks this instruction as illegal. The P bits are also saved in the reorder buffer to allow P bits of executed instructions to be correctly restored when a branch misprediction happens. However, when an exception occurs, the P bit does not need to be backed up, as it will be refetched correctly when the user code executes again.

4.4 Software

PrivLib is the only user-level trusted software with privileges to access the VMA table and protected CSRs. Its code, stack, and heap are all protected with privileged VMAs, and its API entry points are all protected by `uatg` instructions, followed by mandatory security policy checks.

PrivLib manages all protected resources using free lists. During initialization, it populates the PD free list with all possible PD IDs and prepares VMA free lists with free memory chunks partitioned from the reserved memory according to the size class configuration. Resource allocation and deallocation (for both PDs and VMAs) are implemented through atomic pop and push operations on these free lists. For simplicity, these free lists are shared among all threads.

We introduce a new syscall `uat_config` that allows PrivLib to communicate with the OS. During initialization, the OS loads PrivLib code, initializes the VMA table, creates initial privileged VMAs, reserves the virtual memory region, and allocates a reserved physical memory chunk to PrivLib. Such bootstrapping is indispensable as PrivLib cannot load itself or create privileged VMAs before it is initialized. When the physical memory chunk is used up or the application is allocating huge VMAs, PrivLib calls `uat_config` to ask for more reserved physical memory from the OS to refill VMA free lists. Additionally, the OS treats `uatp`, `uatc`, and `ucid` registers as part of the process's context, which are saved and restored during OS context switches.

5 Methodology

We implement Jord's software stack in C++ with Linux 6.10 for the OS support, as described in §4.4. We implement Jord's hardware on QFlex [59], a family of full-system simulators built on top of QEMU, supporting both trace-driven (for functional warming [84]) and cycle-accurate simulation. We model a 32-core CPU running at 4 GHz clock frequency. The detailed parameters for the simulation are shown in Table 2. To study Jord's scalability with increasing core counts and the commensurate impact on latencies for dispatch, cross-function communication, and memory isolation, we also model single-socket systems with up to 256 cores and a

Table 2: System parameters for simulation.

Component	Configuration
Core	32-core, 4 GHz, 4-way OoO, 128-entry ROB, 32-entry SB
TLB Hierarchy	I/D: 48-entry, fully associative L2: 1024-entry, 4-way
VLBs	I/D: 16-entry, fully associative
Cache Hierarchy	I/D: 32 KB, 8-way, 2-cycle latency LLC: 2 MB/tile, 16-way, 6-cycle latency, non-inclusive
Coherence	Directory-based MESI
NoC	8x4 2D mesh, 16 B links, 3 cycles/hop
DRAM	4 MCs, 8 GB per MC

dual-socket system with 128 cores per socket along with a 260 *ns* inter-socket latency following AMD Zen5 Turin [19].

We mainly use cycle-accurate simulation to explore Jord's design space. In addition, as a proof of concept for full-system functionality, performance, and design feasibility at the RTL level, we also prototype Jord on OpenXiangShan [86] with a similar configuration as the simulator and deploy it on an FPGA board with Xilinx XCVU19P. The capacity constraints of the FPGA only allow us to instantiate two OpenXiangShan cores.

We use three microservice applications, Social, Media, and Hotel, from DeathStarBench [25] and OnlineBoutique (Hipster) from Google [27] as the target workloads. We port these workloads to Jord by rewriting them into functions following Jord's paradigm, as described in §3.1. We select two functions with non-trivial functionalities from each workload to study the breakdown of service time in more detail, as shown in Table 3. Function invocation requests are generated using a load generator similar to wrk2 [25] with configurable loads and a Poisson arrival process [25, 67, 72]. We instrument the workloads to obtain latencies in clock cycles for each function invocation, PrivLib operation, and orchestrator dispatch. The latencies are then converted into nanoseconds by dividing the measured clock cycles by the 4 GHz clock frequency.

We use NightCore [35] as the baseline FaaS system to compare with Jord. NightCore uses provisioned containers for concurrency and isolation while optimizing intra-server communication through OS pipes and SystemV shared memory. To assess the upper bound for the performance potential of NightCore, we optimize NightCore by running launchers and workers as normal threads in a single address space [92], with thread pinning and JBSQ dispatch in the same way as Jord. As such, the performance of this optimized version of NightCore is primarily limited by OS pipes.

We consider three variants of Jord to study memory isolation overhead in detail. The baseline Jord uses the PrivLib for isolation and plain list as the VMA table data structure. Jord_{NI} uses PrivLib to manage VMAs, but all isolation operations are bypassed. This configuration represents the upper bound of Jord's performance with no memory isolation. We also evaluate Jord_{BT} with B-tree [28, 37] as the VMA table data structure. In this case, the PrivLib performs B-tree (instead of plain list) operations for VMAs.

Table 3: Functions selected for service time breakdown.

Workload	Function	Abbreviation
Hipster	`GetCart`	GC
	`PlaceOrder`	PO
Hotel	`SearchNearby`	SN
	`MakeReservation`	MR
Media	`UploadUniqueId`	UU
	`ReadPage`	RP
Social	`Follow`	F
	`ComposePost`	CP

Following the methodology in [35, 89], we evaluate each workload in isolation on a dedicated worker server, where the single Jord/NightCore process provides the single-address-space runtime environment. We use throughput under 99-percentile latency as the main performance metric, with SLO set to 10× the minimal-load service time on Jord_{NI}, as is common in the literature [72, 73]. The measurement of a request's latency starts when an orchestrator receives it and ends when the orchestrator is informed by an executor that the request is finished.

6 Evaluation

In this section, we evaluate Jord across three dimensions to demonstrate its performance advantages as a single-address-space FaaS system. First, we compare Jord's performance against state-of-the-art systems to validate its efficiency. Next, we analyze Jord's overheads and their impact on the overall performance. Finally, we investigate the latencies of various operations in Jord over system size to understand its scalability.

6.1 Performance Comparison

Figure 9 compares the 99-percentile latency of Jord_{NI}, Jord and NightCore across various loads for our four workloads. The figure demonstrates, for almost all workloads, that Jord's single-address-space design successfully achieves its goal of minimizing overhead while maintaining SLO. Jord performs within 16% of Jord_{NI}, achieving throughput of 12, 7, and 0.9 MRPS under SLO for Hipster, Hotel, and Social, respectively. Media represents the only exception, where Jord achieves 70% of Jord_{NI}'s performance. The difference stems from Media's distinct behavior, where each function invokes an average of 12 nested functions, compared to three in other workloads.

Jord demonstrates consistently superior performance when compared to NightCore, achieving over 2× better throughput under SLO on average by eliminating the OS memory copying and scheduling overheads of pipes. The advantage becomes particularly pronounced in workloads with frequent cross-function communication, such as Hipster and Media, where NightCore fails to meet the SLO even under minimum load.

6.2 Memory Isolation

Jord's hardware-software co-design enables efficient memory isolation operations. As shown in Table 4, [2] all PD and VMA operations

[2]The latencies on FPGA are overly estimated compared to a real chip as the DRAM runs at a relatively higher frequency than cores.

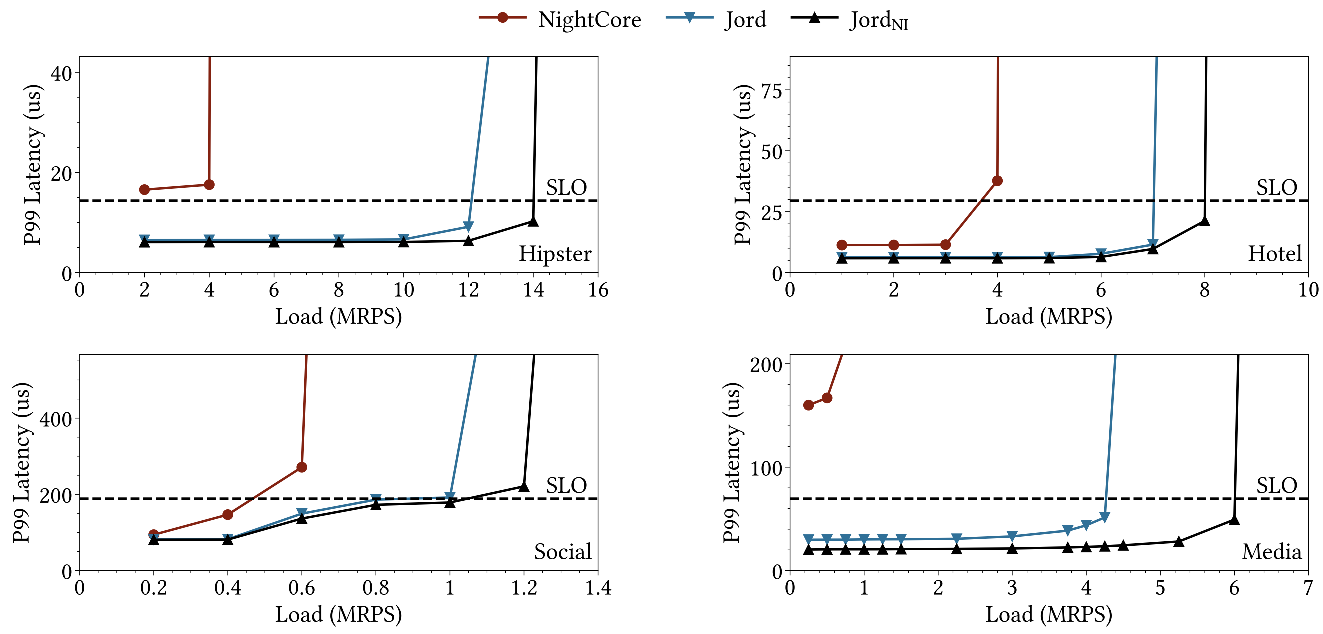

Figure 9: Performance of Jord compared to Jord$_{NI}$ and NightCore.

Table 4: VMA and PD operation latencies.

Operation	Simulator (*ns*)	FPGA (*ns*)
VMA lookup	2	2
VMA update	16	33
VMA insertion	16	37
VMA deletion	27	39
PD creation	11	25
PD deletion	14	30
PD switching	12	22

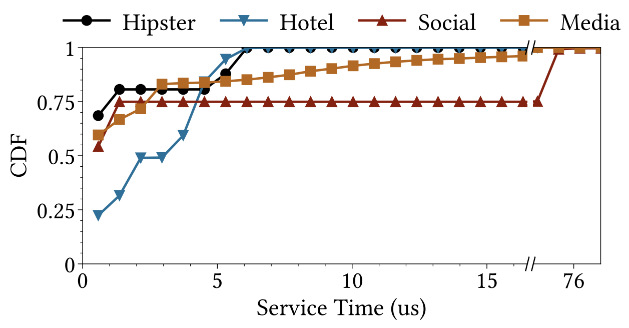

Figure 10: Cumulative distribution function (CDF) of function service time in Jord.

complete in 30 *ns* on the simulator, with total isolation overhead below 120 *ns* per function invocation. In contrast, NightCore takes 0.8 *ms* to prepare a worker process to execute a function [35]. Overall, raw hardware latencies (e.g., VMA lookup) are identical between the simulator model and the RTL model running on the FPGA. Similarly, SRAM structures are sized identically and incur identical latencies when accessed in the two models. However, operations involving instruction execution exhibit a lower IPC in the RTL model because our cycle-accurate simulator models a more aggressive pipeline with fewer control-flow and structural hazards than the RTL model. Nevertheless, both models show that VMA and PD operations can be performed in tens of nanoseconds.

Jord's performance benefits come from its ability to minimize FaaS overhead in the function service time. To illustrate this point, we show the function service time distribution of our four workloads in Figure 10. Across our workloads, 75% of function service times fall below ~5 μs, though Media and Social exhibit long tails, with Social having one function requiring ~75 μs to execute.

Jord's average overhead for dispatch and memory isolation is ~360 *ns* per request, representing only 8%, 4%, and 3% of the service time in Hipster, Hotel, and Social, respectively. While for Media, Jord incurs a higher overhead of ~30% due to excessive nested function invocations, which is still far less than the ~80% overhead incurred when memory isolation and cross-function communication were performed through OS pipes as in NightCore.

We further break down the service time of the eight selected functions into function execution time and various overheads for both Jord and NightCore in Figure 11. On average, Jord achieves 48% less service time than NightCore. Except for RP with excessive nested function invocations (more than 100), Jord's dispatch and memory isolation overheads only constitute ~11% of the service time on average. In contrast, NightCore's overhead exceeds function execution time in most cases. For RP, the overhead even reaches 3× the execution time.

Jord incorporates instruction and data VLBs to cache recently used VMAs. Unlike traditional workloads that typically require a dozen active VMAs [28], FaaS workloads need fewer VMAs due

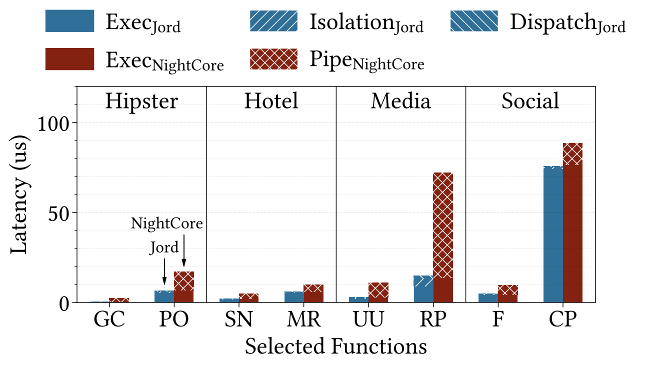

Figure 11: Breakdown of service time into execution time and overheads in Jord and NightCore for selected functions.

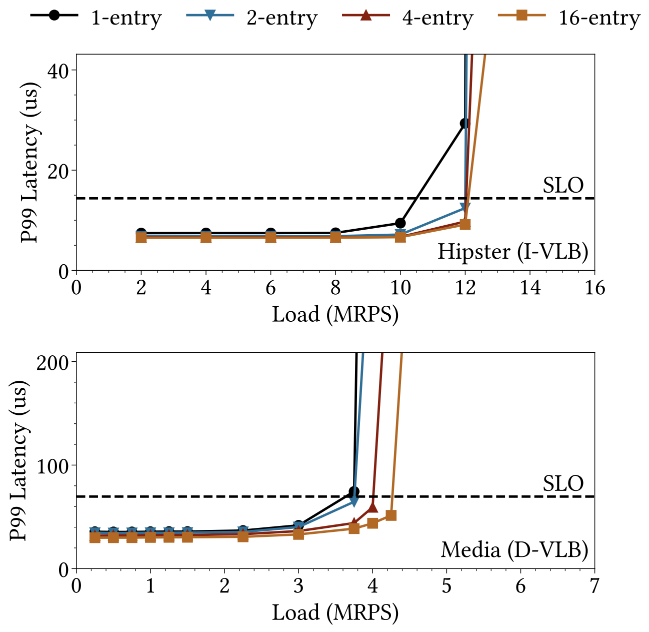

Figure 12: Sensitivity of performance on the number of I-VLB and D-VLB entries.

to their simple functionalities and minimal execution times. We therefore expect FaaS workloads to impose minimal requirements on VLBs, also considering the drastically low VLB miss penalty—2 *ns* in the common case when the traversal hits L1D. Figure 12 verifies our hypothesis for Hipster and Media, the two workloads with the most VLB-size sensitivity. For I-VLB, even two entries are sufficient to achieve 99% throughput under SLO as they essentially cover the instruction VMAs of the function itself and PrivLib. When the control flow switches to the executor or to another function, at most two I-VLB misses can occur and are then quickly served by the plain list, introducing minimal performance overhead.

The situation for D-VLB is similar. In Jord, each function only requires few standalone data VMAs for its private stack, heap, and ArgBufs. As the example workload with frequent cross-function communication, Hipster only requires four D-VLB entries for the best performance. Even for Media as the extreme case, eight entries are still sufficient. We anticipate that more D-VLB entries are required only when functions allocate and heavily use more VMAs than the functions in the workloads we study.

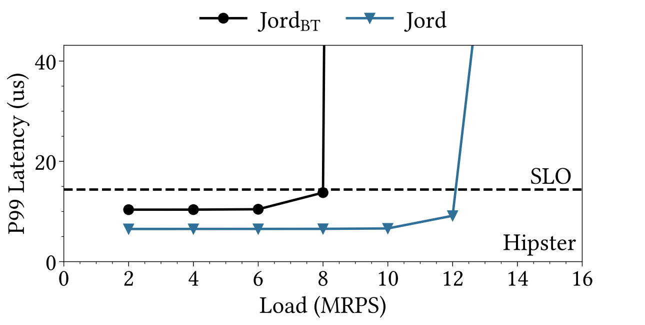

Figure 13: Performance comparison of Jord$_{BT}$ and Jord.

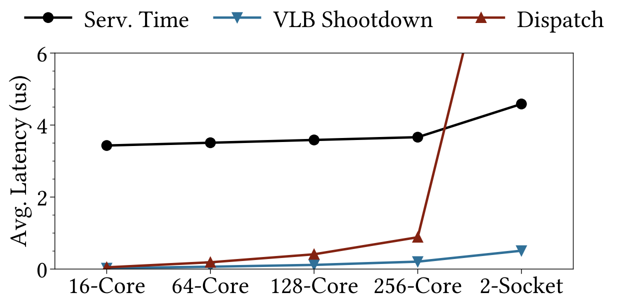

Figure 14: Sensitivity of average function service time, VLB shootdown latency, and dispatch latency on the system scale.

Apart from the plain list, Jord can also work with B-tree [28, 37] as the VMA table data structure, denoted as Jord$_{BT}$. As demonstrated with Hotel in Figure 13 (other workloads show similar results), Jord$_{BT}$ achieves ~60% of Jord's performance due to increased VMA lookup and modification costs. More specifically, the average function service time increases by 43% due to the much higher VLB miss penalty (2 *ns* vs. 20 *ns*). While at the same time, PrivLib spends 167% more time managing VMAs than in the case of plain lists because of the frequent B-tree rebalancing. Despite these costs, Jord$_{BT}$ still outperforms NightCore, achieving 8 MKPS throughput under SLO, while NightCore fails to meet the SLO.

6.3 Scalability

In Jord, three access patterns dominate the coherence traffic among cores. The first pattern is the access to ArgBufs. As the source and target functions can be dispatched to different cores, ArgBufs accesses can trigger multiple point-to-point coherence messages, impacting the service time. The second pattern is the executor's VLB shootdown, which, in the worst case, triggers a global cache invalidation on all executor cores. The third pattern is the orchestrator dispatch, which, in the worst case, generates one coherence message to each executor it manages for retrieving the queue lengths.

Figure 14 illustrates how system size affects the latencies of these access patterns. The average function service time shows modest scaling with the core count, as data transferred through ArgBufs spans only ∼15 cache blocks per request on average, independent of the system's scale. The VLB shootdown latency exhibits a similar sublinear growth with the core count, as the hardware can parallelize the invalidation to each core so that the shootdown latency only depends on the response time of the furthest core. However, the dispatch latency presents a significant scalability challenge. Even with memory-level parallelism that allows loads to multiple cores to be performed simultaneously, the software is still exposed to multiple cross-socket latencies, resulting in each dispatch taking ∼12 μs to finish in a dual-socket system with 256 cores.

This scalability analysis reveals a critical design implication for multi-socket systems: the dispatch latency can exceed average function execution time, causing a single orchestrator to become the performance bottleneck when managing executors across the entire system. To mitigate such inefficiency, the orchestrators should be per-socket, performing affinity-based scheduling and dispatching requests to executors only within the same socket. The potential performance loss due to increased load imbalance in such a multi-orchestrator deployment should not be a concern, as prior work [20] has shown that load imbalance is virtually identical in a multi-queue system and in a single-queue system as the number of cores assigned to each queue exceeds a certain threshold. The same consideration also applies to chiplet-based systems, as cross-chiplet latencies are in the same order as cross-socket latencies on current commodity CPUs [19].

7 Related Work

Recent research has explored novel approaches to constructing cloud applications. ServiceWeaver [26] introduces a programming model that allows developers to write distributed applications as monolithic programs as single binaries, with the runtime system managing communication, deployment, and monitoring. While this approach simplifies development, it lacks specialized support for microsecond-scale workloads. Dandelion [41] employs CHERI [83] capabilities for memory isolation among functions in the same address space, but it faces performance issues due to expensive capability management, especially revocation. Faasm [68] leverages WebAssembly to create lightweight sandboxes within a single address space, but it requires all code to be compiled to WebAssembly, imposing a significant practical limitation. Boucher et al. [14] propose running functions inside dedicated Rust worker processes to reduce code startup latencies. However, relying on Rust's type system for language-level isolation limits its applicability.

Many in-process memory isolation mechanisms have been proposed for isolating untrusted software components. MPK-based approaches [58, 66, 78] add domain specifiers to each page table entry and allow applications to restrict domain permission through the user-accessible permission register. However, due to the limited security model, these approaches must employ expensive measures for control-flow integrity and exclude self-modifying code to prevent unauthorized permission register writes. Mondrian [85] and SecureCells [13] employ segment-based address translation with separate permission tables. They provide mechanisms to accelerate protection domain switches and even permission transfers using special instructions. While these systems enable sub-microsecond domain switches, they still rely on the OS for segment management. HFI [54] introduces region registers to add segment-based permission checks on top of virtual memory. However, the limited number of registers requires manual management and careful allocation. Notably, none of these proposals adequately address TLB coherence among multiple cores.

Sentry [70] adds permission checks to each cache block on the L1 cache miss paths to form another layer of memory protection above the traditional paged virtual memory. Similar to Jord, a user-level supervisor manages these permissions in a per-core permission cache, while the hardware utilizes the existing coherence fabric to maintain coherence among multiple permission caches. However, Sentry leverages software to manage the permission cache and manipulate cache block permissions, resulting in a much higher performance overhead than Jord. Furthermore, as with other in-process memory isolation approaches, Sentry does not optimize the latency of memory allocation and deallocation.

8 Conclusion

We presented Jord, a single-address-space FaaS system that achieves microsecond-level SLOs with strong memory isolation. With a hardware-software co-designed in-process memory isolation mechanism that works on the nanosecond scale, Jord enables zero-copy cross-function communication and scalable function dispatch inside a single address space. Our evaluation shows that Jord performs within 16% of an idealized but insecure baseline system and delivers over 2× higher throughput under SLO compared to enhanced state-of-the-art systems. Jord demonstrates that efficient isolation need not compromise FaaS performance, opening new possibilities for building microsecond-scale FaaS systems.

Acknowledgments

We thank Haibo Chen, Dimitrios Skarlatos, Thomas Bourgeat, Boris Grot, Shanqing Lin, Sidharth Sundar, Ahmet Emre Eser, Leo Loch, and anonymous reviewers for their feedback and support. This work was partially supported by FNS projects "Hardware/Software Co-Design for In Memory Services" (200020B_188696), two Intel research donations (Midgard and CHIMP), National Science Foundation (award 2118851), and the European Research Countil (H2020 StG award 850868).

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications.. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*. 419–434.

[2] Amazon Web Services, Inc. 2024. Auto Scaling Documentation. https://docs.aws.amazon.com/autoscaling/.

[3] Amazon Web Services, Inc. 2024. Serverless Computing - AWS Lambda. https://aws.amazon.com/lambda/.

[4] Amazon Web Services, Inc. 2024. Serverless Computing - AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/.

[5] Amazon Web Services, Inc. 2024. Workflow Orchestration - AWS Step Functions. https://aws.amazon.com/step-functions/.

[6] AMD. 2024. AMD64 Architecture Programmer's Manual. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/40332.pdf.

[7] Nadav Amit. 2017. Optimizing the TLB Shootdown Algorithm with Page Access Tracking.. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*. 27–39.

[8] Nadav Amit, Amy Tai, and Michael Wei. 2020. Don't shoot down TLB shootdowns!. In *Proceedings of the 2020 EuroSys Conference.* 35:1–35:14.

[9] Lixiang Ao, George Porter, and Geoffrey M. Voelker. 2022. FaaSnap: FaaS made fast using snapshot-based VMs.. In *Proceedings of the 2022 EuroSys Conference.* 730–746.

[10] Apache Software Fundation. 2024. Apache Kafka. https://kafka.apache.org/.

[11] Arm. 2024. Arm Architecture Reference Manual for A-profile architecture. https://developer.arm.com/documentation/ddi0487/latest/.

[12] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient virtual memory for big memory servers.. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA).* 237–248.

[13] Atri Bhattacharyya, Florian Hofhammer, Yuanlong Li, Siddharth Gupta, Andrés Sánchez, Babak Falsafi, and Mathias Payer. 2023. SecureCells: A Secure Compartmentalized Architecture.. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P).* 2921–2939.

[14] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2018. Putting the "Micro" Back in Microservice.. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC).* 645–650.

[15] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast.. In *Proceedings of the 2020 EuroSys Conference.* 32:1–32:15.

[16] Jeffrey S. Chase, Miche Baker-Harvey, Henry M. Levy, and Edward D. Lazowska. 1992. Opal: A Single Address Space System for 64-Bit Architectures (Abstract). *ACM SIGOPS Oper. Syst. Rev.* 26, 2 (1992), 9.

[17] Dongwei Chen, Dong Tong, Chun Yang, Jiangfang Yi, and Xu Cheng. 2022. FlexPointer: Fast Address Translation Based on Range TLB and Tagged Pointers.. In *Proceedings of the 31st International Conference on Parallel Architecture and Compilation Techniques (PACT).* 532–533.

[18] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems.. In *Proceedings of the 29th USENIX Security Symposium.* 1409–1426.

[19] George Cozma. 2024. AMD's Turin: 5th Gen EPYC Launched. https://chipsandcheese.com/p/amds-turin-5th-gen-epyc-launched.

[20] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. 2019. RPCValet: NI-Driven Tail-Aware Balancing of µs-Scale RPCs.. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV).* 35–48.

[21] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting.. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV).* 467–481.

[22] fn. 2024. Fn Project - The Container Native Serverless Framework. https://fnproject.io/.

[23] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching.. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI).* 386–400.

[24] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. Sledge: a Serverless-first, Light-weight Wasm Runtime for the Edge.. In *Proceedings of the 2020 International Middleware Conference.* 265–279.

[25] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems.. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV).* 3–18.

[26] Sanjay Ghemawat, Robert Grandl, Srdjan Petrovic, Michael Whittaker, Parveen Patel, Ivan Posva, and Amin Vahdat. 2023. Towards Modern Development of Cloud Applications.. In *Proceedings of The 19th Workshop on Hot Topics in Operating Systems (HotOS-XIX).* 110–117.

[27] Github. 2024. GoogleCloudPlatform/microservice-demo. https://github.com/GoogleCloudPlatform/microservices-demo.

[28] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. 2021. Rebooting Virtual Memory with Midgard.. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA).* 512–525.

[29] Nastaran Hajinazar, Pratyush Patel, Minesh Patel, Konstantinos Kanellopoulos, Saugata Ghose, Rachata Ausavarungnirun, Geraldo F. Oliveira, Jonathan Appavoo, Vivek Seshadri, and Onur Mutlu. 2020. The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework.. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA).* 1050–1063.

[30] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. 1998. The Mungi Single-Address-Space Operating System. *Softw. Pract. Exp.* 28, 9 (1998), 901–928.

[31] Galen C. Hunt and James R. Larus. 2007. Singularity: rethinking the software stack. *ACM SIGOPS Oper. Syst. Rev.* 41, 2 (2007), 37–49.

[32] Intel. 2018. 5-level Paging and 5-level EPT White Paper. https://www.intel.com/content/www/us/en/content-details/671442/5-level-paging-and-5-level-ept-white-paper.html.

[33] Rishabh R. Iyer, Musa Unal, Marios Kogias, and George Candea. 2023. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling.. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP).* 466–481.

[34] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code.. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC).* 107–120.

[35] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices.. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI).* 152–166.

[36] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for usecond-scale Tail Latency.. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI).* 345–360.

[37] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. 2015. Redundant memory mappings for fast access to large memories.. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA).* 66–78.

[38] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics.. In *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI).* 427–444.

[39] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, João Carreira, and Pedro Fonseca. 2024. Pronghorn: Effective Checkpoint Orchestration for Serverless Hot-Starts.. In *Proceedings of the 2024 EuroSys Conference.* 298–316.

[40] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows.. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC).* 805–820.

[41] Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. 2023. Function as a Function.. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SOCC).* 81–92.

[42] Nikita Lazarev, Neil Adit, Shaojie Xiang, Zhiru Zhang, and Christina Delimitrou. 2020. Dagger: Towards Efficient RPCs in Cloud Microservices With Near-Memory Reconfigurable NICs. *IEEE Comput. Archit. Lett.* 19, 2 (2020), 134–138.

[43] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 2019. Mimalloc: Free List Sharding in Action.. In *Proceedings of the 17th Asian Symposium on Programming Languages and Systems, (APLAS).* 244–265.

[44] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing.. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC).* 69–84.

[45] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 2022. FaaSFlow: enable efficient workflow execution for function-as-a-service.. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVII).* 782–796.

[46] Zijun Li, Chuhao Xu, Quan Chen, Jieru Zhao, Chen Chen, and Minyi Guo. 2023. DataFlower: Exploiting the Data-flow Paradigm for Serverless Workflow Orchestration.. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVIII).* 57–72.

[47] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance.. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI).* 49–64.

[48] Guowei Liu, Laiping Zhao, Yiming Li, Zhaolin Duan, Sheng Chen, Yitao Hu, Zhiyuan Su, and Wenyu Qu. 2024. FUYAO: DPU-enabled Direct Data Transfer for Serverless Computing.. In *Proceedings of the 29th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIX).* 431–447.

[49] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Mingyu Wu, and Haibo Chen. 2024. Serialization/Deserialization-free State Transfer in Serverless Workflows.. In *Proceedings of the 2024 EuroSys Conference.* 132–147.

[50] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications.. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC).* 285–301.

[51] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing,

Bundling, and Prewarming for Serverless DAGs.. In *Proceedings of the 16th Symposium on Operating System Design and Implementation (OSDI)*. 303–320.

[52] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2019. Prefetched Address Translation.. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1023–1036.

[53] Microsoft. 2024. Azure Functions. https://learn.microsoft.com/en-us/azure/logic-apps/logic-apps-overview.

[54] Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, Dean M. Tullsen, and Deian Stefan. 2023. Going beyond the Limits of SFI: Flexible and Secure Hardware-Assisted In-Process Isolation with HFI.. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVIII)*. 266–281.

[55] NATS Authors. 2024. NATS.io - Cloud Native, Open Source, High-performance Messaging. https://nats.io/.

[56] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers.. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*. 57–70.

[57] OpenFaaS Ltd. 2024. OpenFaaS - Serverless Functions Made Simple. https://www.openfaas.com/.

[58] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK).. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. 241–254.

[59] QFlex. 2024. Quick & Flexible Computer Architecture Simulation. https://qflex.epfl.ch/.

[60] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-Chin Wang, and K. K. Ramakrishnan. 2022. SPRIGHT: extracting the server from serverless computing! high-performance eBPF-based event-driven, shared-memory processing.. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 780–794.

[61] RISC-V International. 2022. Specifications. https://riscv.org/technical/specifications/.

[62] Bogdan F. Romanescu, Alvin R. Lebeck, Daniel J. Sorin, and Anne Bracy. 2010. UNified Instruction/Translation/Data (UNITD) coherence: One protocol to rule them all.. In *Proceedings of the 16th IEEE Symposium on High-Performance Computer Architecture (HPCA)*. 1–12.

[63] Rohan Basu Roy, Tirthak Patel, Rohan Garg, and Devesh Tiwari. 2024. CodeCrunch: Improving Serverless Performance via Function Compression and Cost-Aware Warmup Location Optimization.. In *Proceedings of the 29th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIX)*. 85–101.

[64] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: warming serverless functions better with heterogeneity.. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVII)*. 753–767.

[65] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. 2022. Memory deduplication for serverless computing with Medes.. In *Proceedings of the 2022 EuroSys Conference*. 714–729.

[66] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys - Efficient In-Process Isolation for RISC-V and x86.. In *Proceedings of the 29th USENIX Security Symposium*. 1677–1694.

[67] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Irfan Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider.. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*. 205–218.

[68] Simon Shillaker and Peter R. Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing.. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*. 419–433.

[69] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. 2022. Fireworks: a fast, efficient, and safe serverless framework using VM-level post-JIT snapshot.. In *Proceedings of the 2022 EuroSys Conference*. 663–677.

[70] Arrvindh Shriraman and Sandhya Dwarkadas. 2010. Sentry: light-weight auxiliary memory access control. In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)*. 407–418.

[71] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls.. In *Proceedings of the 9th Symposium on Operating System Design and Implementation (OSDI)*. 33–46.

[72] Mark Sutherland, Babak Falsafi, and Alexandros Daglis. 2023. Cooperative Concurrency Control for Write-Intensive Key-Value Workloads.. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVIII)*. 30–46.

[73] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra J. Marathe, Dionisios N. Pnevmatikatos, and Alexandros Daglis. 2020. The NEBULA RPC-Optimized Architecture.. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*. 199–212.

[74] Ariel Szekely, Adam Belay, Robert Morris, and M. Frans Kaashoek. 2024. Unifying serverless and microservice workloads with SigmaOS.. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*. 385–402.

[75] The Apache Software Foundation. 2024. Apache OpenWhisk. https://openwhisk.apache.org/.

[76] The Knative Authors. 2024. Knative. https://knative.dev/docs/.

[77] The Kubernetes Authors. 2024. Kubernetes. https://kubernetes.io/.

[78] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK).. In *Proceedings of the 28th USENIX Security Symposium*. 1221–1238.

[79] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: Protecting software with Code-centric memory Domains.. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*. 469–480.

[80] Carlos Villavieja, Vasileios Karakostas, Lluís Vilanova, Yoav Etsion, Alex Ramírez, Avi Mendelson, Nacho Navarro, Adrián Cristal, and Osman S. Unsal. 2011. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory.. In *Proceedings of the 20th International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 340–349.

[81] Jérôme Vouillon. 2008. Lwt: a cooperative thread library.. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML (ML)*. 3–12.

[82] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute.. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*. 443–457.

[83] 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. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization.. In *IEEE Symposium on Security and Privacy*. 20–37.

[84] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. 2006. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro* 26, 4 (2006), 18–31.

[85] Emmett Witchel, Josh Cates, and Krste Asanovic. 2002. Mondrian memory protection.. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*. 304–316.

[86] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Quan, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, and Yungang Bao. 2022. Towards Developing High Performance RISC-V Processors Using Agile Methodology.. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1178–1199.

[87] Zi Yan, Ján Veselý, Guilherme Cox, and Abhishek Bhattacharjee. 2017. Hardware Translation Coherence for Virtualized Systems.. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*. 430–443.

[88] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. RainbowCake: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing.. In *Proceedings of the 29th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIX)*. 335–350.

[89] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing.. In *Proceedings of the 20th Symposium on Networked Systems Design and Implementation (NSDI)*. 1489–1504.

[90] Zhe Zhou, Yanxiang Bi, Junpeng Wan, Yangfan Zhou, and Zhou Li. 2023. Userspace Bypass: Accelerating Syscall-intensive Applications.. In *Proceedings of the 17th Symposium on Operating System Design and Implementation (OSDI)*. 33–49.

[91] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2023. AQUATOPE: QoS-and-Uncertainty-Aware Resource Management for Multi-stage Serverless Workflows.. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVIII)*. 1–14.

[92] Danyang Zhuo, Kaiyuan Zhang, Zhuohan Li, Siyuan Zhuang, Stephanie Wang, Ang Chen, and Ion Stoica. 2021. Rearchitecting In-Memory Object Stores for Low Latency. *Proc. VLDB Endow.* 15, 3 (2021), 555–568.