# Janus: OS Support for a Secure, Fast Control-Plane

Wenyuan Shao, Xinyu Han, Evan Stella, Linnea Dierksheide, Phani Kishore Gadepalli, Gabriel Parmer
The George Washington University, Washington, DC
{shaowy,kevin_han,evanstella,ldierksheide,phanikishoreg,gparmer}@gwu.edu

***Abstract*—The emergence of the edge cloud, empowered by advanced wireless technologies such as 5G, is aimed at delivering predictable, interactive services within low milliseconds. Even in the traditional cloud, latency-sensitive services are pervasive. To enable low-latency software, much focus has been on data-plane optimizations for fast processing of requests. Unfortunately, these efforts alone are insufficient: effective, low-latency services also require advances in the control-plane. However, control-plane operations require strong spatial and temporal isolation between tenant computations, thus can result in significant overhead.**

**This paper introduces Janus, an OS abstraction for a flexible control-plane which provides both strong isolation and low latency through the use of pervasive kernel-bypass. Janus leverages Memory Protection Key (MPK) hardware to enable low-cost control operations for protected procedure call (PPC) and thread dispatch – two essential building blocks of spatial and temporal isolation. By transparently improving these fundamental control-plane operations, Janus enables efficient and predictable user-defined and customizable system control policies and mechanisms, while maintaining strong isolation. We evaluate Janus's ability to extensibly define new control mechanisms, increase the efficiency of an existing RTOS, and support low-latency services in a `memcached` server. Compared to a Linux approach, a specialized latency-sensitive control plane using Janus provides over a 6x improvement in throughput, while providing 99th percentile tail latencies almost 3x lower. Demonstrating the utility of an efficient control plane, Janus improves tail latency in a multi-tenant system by orders of magnitude.**

## I. Introduction

Predictably low-latency services have become increasingly prevalent in our computational infrastructure. For example, within data-centers, microseconds in the tail matter [1], [2] and must be designed around [3], [4], [5]. Tail latency is a core optimization due to low-latency networking and fanout/fanin architectures [2] where service time is bounded by the worst-case request latency. Similarly, the proliferation of low-latency wireless (*e.g.* 5G) enables the network edge to be accessed in the low-milliseconds. This enables software running on the edge to provide cloud services with tighter latency properties, but emphasizes the need for tightly controlled latencies in software. To provide low-latency services, much of the focus has been on optimizing the *data-plane* [6], [7] – the fast-path software for processing on requests and providing replies. Similarly, real-time systems often focus on interrupt response time as the core system optimization – to enable processing on the data quickly and within a bounded latency.

Unfortunately, optimizing the data-plane isn't sufficient. The promise of cloud infrastructures is to provide strong isolation facilities to enable third parties to provide potentially low-latency services. Similarly, real-time systems are adapting to provide the strong isolation facilities required for consolidation [8] in embedded systems. In such systems, the *control-plane* – the software that coordinates resources between services, processes, and threads – must intelligently multiplex resources in a latency-aware manner.

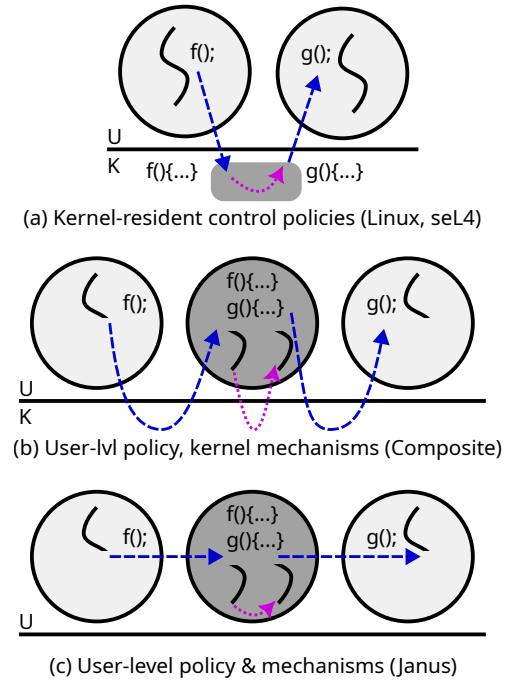



Fig. 1: An interaction between two threads in different protection domains (processes). Blue dashed lines are calls to receive some service and correspond to logically invoking function `f()` and returning from function `g`, purple dotted lines indicate thread dispatch, and dark grey denotes the implementation of the control policy (including scheduling and thread coordination policies). Examples: `f` and `g` might be sending and receiving on a channel, or unlocking and locking a lock. (a) OSes such as Linux or seL4 [9] that have control policies and mechanisms in the kernel. (b) Composite [10] microkernel that enables user-level control policies, but uses kernel-resident IPC and dispatching mechanisms. (c) Janus that *also* enables user-level control mechanisms, thus avoiding mode switches with customizable control policies.

Despite their importance, control-plane operations often involve significant overhead due to the need for strong tenant isolation which imposes restrictions on access to memory, processing cycles, and I/O. Isolation mechanisms include page tables to restrict memory accesses, scheduling, and execution of I/O management in hardware-isolated protection domains (either by page table or dual mode protection). These mechanisms entail significant overhead, due to interactions between

user- and kernel-level [11], [12], [13] and between virtual memory address spaces. This includes micro-architectural effects [11] from mode-switching, the cache overheads of kernel paths [12], [13], [14], and TLB cache effects of multiple address spaces (even with tagged TLBs [15], [16]). Traditional RTOSes avoid isolation to maintain predictable execution control through direct scheduling and thread coordination [17]. Other approaches leverage single address space operating systems to avoid hardware-provided isolation [18], [19], [4], [20], [21]. Unfortunately, the major focus of existing research is to optimize the performance of the data-plane while focusing less on the control-plane, which can become the bottleneck when strong isolation is required.

In this paper, we present Janus[1], depicted in Figure 1(c), an OS abstraction for a flexible control-plane that provides both strong isolation and controlled latency. Janus achieves this by utilizing intra-page-table isolation hardware facilities to avoid page-table and mode switching overheads, while integrating them cleanly into a $\mu$-kernel's capability-based access control to maintain strong security. Customized preemptive scheduling, coordination, and multiplexing policies are implemented extensibly at the user-level. When paired with kernel-bypass-based I/O management techniques, Janus can avoid all mode- and page table switches enabling *a latency-sensitive control-plane that provides strong, customizable isolation, coordination, and scheduling*. Janus's efficient control-plane enables 1) customized CPU management and scheduling policies, 2) inter-thread coordination policies, 3) lightweight I/O multiplexing, and 4) the fine-grained software partitioning required for multi-tenancy.

Janus leverages the Memory Protection Keys [22] (MPK) x86 extension which provides a mechanism for low-overhead switching between protection domains at user-level (§II-A). MPK hardware has a number of limitations: protection domains must share a virtual address space (as in Single Address-Space OSes [23], [24]), there are a limited number of MPK keys (*i.e.* 15), untrusted user-level code can change MPK domains [25], [26], and underlying OSes are susceptible to confused deputy attacks [27], [28], [29] as they don't understand intra-process protection domains.

Janus leverages MPK to enable low-cost control-operations for IPC, scheduling, and inter-thread coordination. It does so while integrating MPK-based mechanisms into the core OS ensuring that they provide consistent semantics and security properties of the underlying capability-based system. This prevents attacks within existing MPK infrastructures including confused deputy and control-flow attacks. It additionally provides mechanisms to control virtual address spaces to avoid the 15-key limitation (§III-D).

Janus demonstrates that 1) the control-plane can be customized with new coordination mechanisms by implementing L4-style IPC entirely at user-level that is faster than seL4, 2) RTOS latencies in Patina [30] can be significantly decreased with efficient control, and 3) latency-sensitive cloud systems using `memcached` can achieve significantly improved throughput and latency due to a customized control-plane and efficient isolation mechanisms, especially in multi-tenant scenarios.

[1]Janus is the Roman mythological god of gates and transitions, and their protection. Similarly, our system creates new abstractions and mechanisms for gate-based control transitions, while maintaining the strong protection of the underlying OS.

**Contributions.** The key contributions of this paper follow.

- Janus introduces a latency-sensitive and customizable control-plane abstraction for managing executions of multiple, latency-sensitive and untrusted tenants.
- Janus transparently integrates the existing isolation mechanisms in a security-centric OS with those provided by hardware extensions for memory isolation (MPK), thereby enabling efficient and predictable extension of control-plane policies.
- We evaluate Janus and compare it with existing $\mu$-kernels, a $\mu$-kernel-based RTOS, and Linux.

## II. Background and Related Work

Janus uses x86's MPK to enable direct, user-level switching of memory protection domains. It expands the execution model of the open source and publicly available Composite [10] OS to support fast, predictable control-plane operations while maintaining backward compatibility with application and system code. This section covers the background for MPK (§II-A) and Composite (§II-B).

### A. Memory Protection Keys (MPK)

MPK [22], [31] is an x86 architectural feature that provides memory protection facilities within a page table. MPK enables the restriction of memory access *within* a page table to a set of pages marked with domains or keys, and provides user-level instructions to quickly switch active domains. The Protection Key register's (`PKRU`) bits represent active access rights (read/write) to specific domains. Similarly, each page in a process is associated with a set of domains. To implement this, each page table entry contains a bitmap containing up to 16 domains. The user-level accessible `wrpkru` instruction directly updates the read/write access permissions to each domain in `PKRU`, thus enabling the dynamic, user-level update of memory access rights.

As an example, two different ranges of pages in a process have page table entries that tag them separately as domains 1 and 2. The process executes `wrpkru` making only domain 1 read/writable. At this point, any accesses to the memory in domain 2 would page-fault. After executing `wrpkru` to make only domain 2 read/writable, memory access to pages in domain 1 would correspondingly fault. At first glance, this provides the core of what might be necessary to provide memory protection domains, with fast switching.

MPK has a number of limitations:

- *Unrestricted access to MPK instructions.* Any user-level code can update the active MPK domains, which can easily subvert the required protection.
- *Limited number of MPK domains.* As MPK enables only 15 protection domains, it does not scale to all protection domains in an OS.

- *Control Flow Integrity (CFI).* When used to communicate between isolated components, restricting memory accesses is a necessary but not sufficient metric for security. CFI [32] must also be maintained. The set of instruction entry points for Protected Procedure Calls (PPCs), and the instruction address to be returned to when a client performs a PPC, must be carefully maintained.
- *Semantic gap between OS and MPK protection domains.* MPK provides memory protection domains *within* existing process abstractions. This can lead to a mismatch [28] between OS process access control, and the access control of MPK protection domains.

**Restricting access to MPK instructions.** Janus must prevent user-level code from being able to directly update memory access controls through the `PKRU` register. Janus takes a conventional approach [25], [33], [34] to restricting access to MPK instructions: we scan the binary and disallow any memory patterns that match `wrpkru` or `xrstor` instructions, detailed in §V-D.

**More than 15 memory protection domains.** Research projects have added interfaces to program and use MPKs [31], to virtually expand the number of keys [35], or to use both MPK and `VMFUNC` [36], [37], [38], [39] to provide a larger effective number of protection domains. Dynamic approaches that virtualize many more keys require recoloring the keys at runtime [40], which slows down the system, and incurs unpredictable delays. Instead, Janus co-designs the page table management of address spaces with MPK domains (§III-D) to enable an unlimited number of clients to use MPK-based PPC to call the set of control-plane services.

**Ensuring PPC CFI.** Most MPK approaches generate *call-gates* [25], [26], [13] that transition a function call across protection boundaries directly using the MPK instructions. While these approaches carefully validate that a specific protection domain is allowed to call another, they do not validate that a PPC *return* specifically corresponds to a PPC invocation. In contrast, Janus is the first system we know of that ensures PPC CFI by devoting an MPK domain to logic to track the sequence of PPCs (§III-B).

**Unification of MPK and OS security primitives.** In-process isolation facilities [25], [26], [13] are treated by the kernel as all part of the encompassing process. Thus system calls can be used to bypass the MPK isolation facilities, for example by `mm`apping memory in another domain [28]. $\mu$Switch [34] integrates MPK protection domains with those in Linux, but only in restricted cases to provide stronger isolation of libraries. Janus integrates MPK-based PPC and protected dispatch into the strong capability-based security model of an existing $\mu$-kernel.

### B. Composite

Composite is a $\mu$-kernel based on a capability-based access control model [41], [42]. In Composite, *components* include the code and data for execution, a set of functions they export to be called by other components, and a set of functional dependencies. Components are isolated using page tables, and function dependencies exported by another component are invoked via thread migration-based IPC[2]. Thread migration-based IPC [43], [44], [45] is synchronous and mimics function call semantics. When invoking a function in the server protection domain, a thread discontinues execution in a client protection domain, and resumes at a defined entry point in a server. When returning from that invocation, that thread's execution resumes in the client, thereby emulating a function call and return, but between protection domains. From the scheduler's perspective, the same thread executes *across protection domains*, thus requiring no thread context switch. In this paper, we'll assume that all PPC adheres to thread migration semantics.

### C. Slite

Slite [46] is a user-level scheduling framework designed for predictable, efficient, and customizable scheduling. Unlike traditional schedulers that rely on kernel-mediated thread dispatch, Slite enables near-zero-cost thread dispatch by directly saving and restoring thread registers. When dispatching to a preempted thread, the kernel path is still taken to enable restoring full register context and potentially switching active components. Slite enables schedulers to switch between threads without kernel involvement, which leads to inconsistencies between the tracking of the active thread in the kernel and user-level. User-level dispatch changes the active thread, as does kernel-handled interrupts. As both user- and kernel-level change the active thread, they must coordinate to understand which thread and component are active. As part of that coordination, the kernel validates that the scheduler has a capability to the thread.

To enable this scheduler/kernel coordination, Slite introduces two key data structures: a per-core Scheduler Control Block (SCB), and a per-thread Dispatch Control Block (DCB). The SCB tracks the (capability to the) currently active thread, and the DCB tracks preemption status and register state. While Janus is built on Slite, the PPC path must have a trustworthy understanding of the currently active thread, without kernel coordination. As such, Janus introduces a Protected Dispatch operation with which the SCB and DCB operations are MPK-protected.

## III. Janus System Design

Janus focuses on two main control operations as building blocks for OS spatial and temporal isolation: *thread dispatch*, and *Protected Procedure Calls (PPC)*. Thread dispatch enables schedulers to multiplex threads across cores. PPC facilitates logical function calls between protection domains, supporting cross-domain coordination and allowing OS abstractions, policies, and mechanisms to be defined within user-level compo-

[2]Note that thread-migration-based IPC is different from the "thread migration" used to colloquially refer to moving a thread from a core to another. Thread-migration-based IPC used for communication between protection domains with all execution on the same core. When we reference "thread migration" in this paper, we always mean thread-migration-based IPC.

nents. These operations represent the kernel-resident control-plane foundation for memory and processor multiplexing in Composite. Janus enables both kernel- and user-level control operations with consistent semantics and security properties.

Scheduling logic executes at user-level in isolated protection domains. Scheduling policies use thread dispatch to directly switch between threads. Unlike library-based dispatch (*e.g.* in go), Janus switches between actual system threads (not user-level threads [46]) and supports interrupt-driven thread preemption. Scheduling and synchronization policies are built on thread dispatch.

PPCs provide cross-protection-domain function-call semantics whereby a thread in a *client* protection domain invokes a function in a separate *server* protection domain. Client and server executions are spatially isolated (*e.g.* in separate page tables), thus each access separate memory sets, including disjoint execution stacks. PPCs use thread migration [45] to avoid switching between threads, thus also avoiding scheduler interaction during IPC. This approach enables the IPC necessary for protected coordination, in which a client requests service from a server that provides an abstraction using its heightened resource access.

Thread dispatch and PPC are the core building blocks for system control-plane policies for resource management and coordination. PPC enables $\mu$-kernel-style isolation of system services, including the system scheduler. PPCs enable client components to safely invoke the scheduler component's functions that provide abstractions for locks, semaphores, and timing, which in turn utilize thread dispatch to manage concurrency. In Janus, when paired with kernel-bypass-based I/O, direct, user-level implementations of thread dispatch and PPC can avoid all mode-switches while providing traditional, protected OS services.

### A. Goals, Challenges, and Mechanisms

The core goals and key properties of Janus's design focus on enabling the integration of customized, efficient, and predictable control operations into data-paths to enable fine-grained multi-tenant computation.

**G1: Efficient control operations.** Janus avoids processor mode transitions – thus their overheads – on most control-path operations including both thread dispatch and PPC. This enables performance-sensitive data-path computations to efficiently integrate tightly with predictable control operations.

**G2: Customizable control-plane operations.** Specialized systems can benefit from customized and extensible scheduling, control-flow, and communication abstractions. Micro-second-scale systems [1], [4], [47], [5], [20] utilize specialized, tail-latency-aware scheduling. Real-time systems require specific policies that maintain the predictable temporal properties of tasks.

**G3: Uniform security model.** Unlike in-process isolation mechanisms [25], [26], [48], [13], [49], which separate in-process protection domains from OS processes, Janus unifies protection domain access-control around capability-based security, ensuring consistent semantics and security across in-kernel and user-level control paths.

The core challenge to achieving these goals is that kernel-level mechanisms for PPC and thread dispatching must co-exist alongside user-level mechanisms and policies. MPK hardware only enables 15 different protection domains (§II-A), thus requiring them to be used alongside normal page-table-based protection domains to support more components. This raises the question of *how user- and kernel-resident policies coordinate to identify the currently active protection domain?* To enable preemptive scheduling, interrupts must directly dispatch between threads within the kernel, necessitating kernel-resident dispatching logic. Consequently, with user-level scheduling policies and kernel-level dispatching for preemption, *how can user- and kernel-resident policies consistently determine the currently active thread?* To achieve these goals, Janus's design focuses on three core mechanisms.

**Direct, user-level control operations.** Key control operations to multiplex threads on cores and to conduct IPC between protection domains proceed transparently and directly at user-level using abstractions constructed around MPK. Despite this, to maintain the system's security properties, switching threads and doing IPC between protection domains is only possible if corresponding kernel capabilities exist.

**Co-design of kernel- and user-level control operations.** Key data-structures integral to the control fast-paths of thread dispatch and synchronous IPC are directly accessed by both user- and kernel-level. This enables the elision of system calls, instead relying on asynchronous coordination through these data-structures between user-level policies and the kernel. Nevertheless, it preserves the semantics of system operations, replacing them transparently with efficient, protected, user-level variants wherever possible.

**Asynchronous protocols for delayed kernel consistency.** User-level policies perform thread dispatch and IPC invocations without mode-switching into the kernel, instead updating the shared data-structures at user-level. Janus maintains normal, non-shared, kernel data-structures. Control operations must asynchronously interact with the kernel, so that it can correctly update its state (*i.e.* which thread and protection domain is active) when it is next activated.

### B. Janus PPC Design

Protected Procedure Calls enable resource management services to multiplex their resources across multiple clients. They require the following operations:

1) save client context (including ip and sp),
2) switch to the server memory protection domain,
3) establish a stack,
4) execute the target server function,
5) upon return, switch back to the client protection domain,
6) restore its context.

The Composite kernel provides IPC by saving client context in the kernel in a per-thread *invocation stack* – where the top of the stack indicates the thread's active component – and uses page tables to provide memory protection domains. When

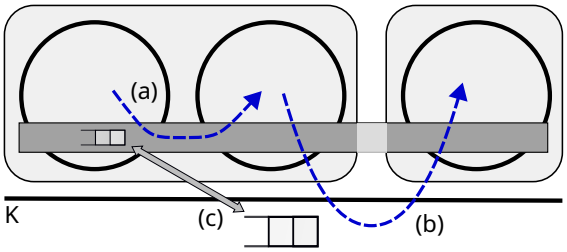



Fig. 2: Protected Procedure Calls in Janus use MPK-protected (grey box) per-thread stacks to track PPCs in (a). The rounded rectangle boxes around components depict address spaces. The first two components are in the same address space, thus they use direct PPC through user-level in (a). The second two are not, thus they must use the kernel PPC path (b). The user- and kernel PPC stacks are synchronized in (c) to determine the thread's active component.

calling the server, steps 1 and 2 are performed by an IPC system call, context is saved in the invocation stack, and a capability identifies which server to switch to. The client uses a capability to a PPC endpoint resource, which denotes which function in which server is invoked. Returning from server to client in steps 5 and 6 requires a system call, activating a *return capability* (with value `0`), which pops the client's context from the current thread's invocation stack to return to it.

Janus's PPC replaces steps 1, 2, 5, and 6 with user-level operations by using hardware support to switch protection domains via MPK. Yet it must still a) provide the capability-based protection of invocations and returns identical to that of the kernel, b) isolate the client context from both the client and server so that the PPC returns to the correct context in the client, and c) enable the kernel to understand which protection domain is active for future system calls.

**User-level Invocation Stack.** To maintain a trace of user-level invocations, Janus defines a per-thread *user-level invocation stack*. Each PPC invocation pushes the client context onto the stack, along with the capability corresponding to the callgate. Each return pops the context off the stack, restoring the client. Figure 2 depicts Janus's data-structures.

This data-structure serves three purposes. (1) It provides the control-flow integrity of PPC returns into the client, by accurately restoring client context (instruction pointer and stack). (2) It maintains a proper, nested return order to the correct protection domain, thus maintaining CFI around the return semantics of PPCs. In contrast, existing codegates [25], [13], [26] validate only that *a return is from the proper (server) protection domain*, but *not* that the return was paired with a specific active invocation from the client to the server. (3) It tracks the capabilities corresponding to the callgates. This enables the kernel to asynchronously consider both a thread's kernel-resident invocation stack and the user-level invocation stack when identifying which protection domain is active.

The user-level invocation stack must be isolated from both the client and the server, so that neither can inhibit proper PPC semantics. Thus, Janus isolates this stack into a separate system protection domain (with a dedicated MPK domain), inaccessible from normal components.

**Janus's MPK Callgates.** The Janus PPC implementation uses MPK-based callgates for direct, user-level switching between protection domains. During an invocation and before updating the invocation stack, the callgate validates that the intended client is invoking the callgate. Before switching to the server, the callgate validates that the invocation stack was properly updated. Similarly, when returning, the callgates validate that the server is returning, and that the proper client is returned to.

The user-level invocation stack must be isolated from component code, thus Janus protects in a separate *system MPK protection domain*. This requires a callgate that switches from a client to a server to go through the intermediately user-level invocation stack domain, thus doubling the number of protection-domain switches. While this adds overhead, it is key in Janus to maintain the CFI around PPC and access control guarantees of the existing OS. §V-A presents a detailed callgate implementation.

**User-level and Kernel PPC Synchronization.** While a core goal of Janus is to provide user-level, direct PPCs, the kernel must be able to coordinate through the user-level invocation stack to understand, for a given thread, which protection domain it is active in. For example, when a component makes a system call (*e.g.* to map memory or to delegate a capability), the kernel must ascertain which capability-table and page table it should use to identify and constrain access rights. Janus determines the currently active component and protection domain by examining both the user and kernel invocation stacks, despite the asynchronous and preemptive execution of user-level PPCs.

### C. Janus Scheduling Design

Janus is built on the user-level scheduling component in Composite, extending it to provide protected user-level thread switching. The kernel has no blocking APIs, which empowers the scheduler to efficiently define abstractions and policies to manage concurrency across all components in the system. A scheduler in Composite is a component that has access to threads in its capability-table. Dispatching to a thread via a capability contains at least the following operations:

1) save the current thread's registers,
2) potentially reprogram the timer to fire at a specific (scheduler-chosen) time in the future,
3) if the next thread is not executing in the scheduler component (as it was previously preempted), switch active page tables,
4) restore the register contents of the target thread,

All four steps are executed by the Composite kernel during the dispatch system call (*i.e.* using a capability to a thread). While all steps are generally required, context switches for concurrency primitives often maintain the previously programmed timer value, and switch to a thread that intentionally blocked itself (thus was not preempted). In such cases, only steps 1 and 4 are required. Previous research on Slite [46] leverages this observation to enable direct, user-level dispatch. Janus integrates user-level dispatch into the capability-based

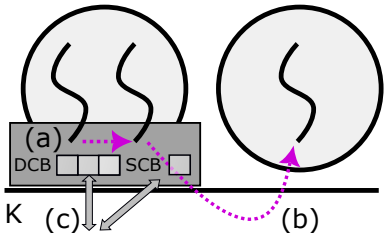



Fig. 3: Protected dispatch in Janus uses MPK-protected (grey box) per-thread context structures in (a). The scheduling component (left) can direct switch between threads, except those that were preempted and require (b) kernel dispatch. The kernel synchronizes these structures in (c) to update its view of the currently active thread.

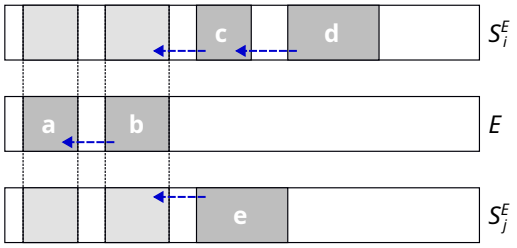



Fig. 4: An example of explicit address space management with *split*. Address space $E$ is split into $S_i^E$ and $S_j^E$. This results in the components in address space $E$ being invoked directly via user-level PPC from components in all of $E$, $S_i^E$, and $S_j^E$.

security of Composite by providing *dispatch gates* that validate a scheduler's access right to dispatch to a thread.

**Protected dispatch data-structures.** In Slite (§II-C), the scheduler has direct access to the SCB and DCB, which means the PPC path must trust the entire scheduler to properly update the active thread. While this may be a reasonable decision in many cases, Janus's protected dispatch wraps all of these data-structure accesses and logic in the *system MPK protection domain*, thus ensuring that the active thread is accurately updated during dispatch. Figure 3 depicts the key data-structures and their interactions in Janus. When a thread is *not* in a preempted state (tracked by DCB), the scheduler component can switch to it with user-level thread dispatch in (a). Otherwise, the scheduler component must use kernel dispatch (b) to switch to that thread. When it next executes, the kernel synchronizes user- and kernel-level states in (c) using the information in SCB and DCB (§V-C).

**User- and kernel-level synchronization.** As the user-level scheduling logic and the kernel execute asynchronously, they must synchronize around the key thread state. The SCB tracks the capability to the thread of the active thread. The user-level scheduler uses protected dispatch to switch between threads, updating this value. When the kernel activates (*i.e.* due to an interrupt, or system call), it looks up the SCB's thread capability in the component's capability table. When an interrupt preempts a thread, the kernel notes this in the DCB, so that a future dispatch to the thread through the protected dispatch path will know it needs to go through the kernel.

### D. Janus Address Space Management

As discussed in §II, MPK hardware is limited to 15 different memory protection domains, motivating the continued use of traditional protection domains provided with page tables. MPK does not switch page tables so only components that share a virtual address space can use MPK for IPC. Thus, the straightforward combination of page tables and MPK is limiting: the small number of components that share a page table with the control-plane component (*e.g.* the scheduler) will have fast access to it, but components in other page tables will require kernel-mediated control operations.

Janus enables control-plane policies to be quickly accessed at user-level by a large number of components by 1) providing an API to explicitly control the mapping of component protection domains to virtual address spaces, and 2) creating a *split* operation that enables sets of components (*i.e.* in the control-plane) to be shared across many virtual address spaces. Address spaces are created explicitly, and when a component is created, its virtual address space is an explicit parameter. Two components that are created using the same virtual address space will share it (*i.e.* they cannot have overlapping addresses). Later, when PPC IPC capabilities link these components, Janus will use through-kernel IPC if components are in separate address spaces, and MPK-based PPC gates otherwise.

The *split* operation enables a set of components to be in *multiple address spaces* – thus providing direct, user-level PPCs from components in any of those address spaces. To describe this operation, we'll treat each address space as a set of all resident components. As shown in Figure 4, split is performed on an existing address space, $E$, to generate a new address space, $S_j^E$, that includes all components in $E$ *and* any other components: $E \subseteq S_j^E$. Any two address spaces $S_j^E$ and $S_i^E$ *split* from $E$ share the same components, $E = S_j^E \cap S_i^E$. As such, any of the components in $S_j^E$ or $S_i^E$ will use user-level, MPK-based PPC to any component in their address space or in $E$. Janus ensures that PPCs and protected dispatch will use the kernel-based path from a component in $E$ to any component in $S_j^E$ (as $S_j^E$ might not be the active address space at the time of the PPC). Split enables an unlimited number of components to be able to use MPK-based PPC to the control-plane components in the shared subset $E$. This solution enables *predictable and guaranteed* access to user-level PPCs in contrast to approaches that dynamically manage MPK keys and have significant overhead [36].

## IV. Janus Security and Predictability Analysis

This section analyzes Janus's security properties and predictability of its PPC, thread dispatch, and user-kernel synchronization.

### A. Janus Security Properties

While Janus's callgates and dispatch gates enable kernel-bypass, protection domain switches using `wrpkru` introduces challenges. These include (1) access control of the gates themselves, (2) mid-gate hijacking, and (3) unauthorized access to MPK instructions.

**Access control for PPC gates.** A client protection domain gains PPC callgate access only when a corresponding capability is added to its (kernel resident) capability-table. If access to the server function is removed by revoking the capability from the capability-table, the client should correspondingly no longer have PPC access to the server's function. To provide this one-to-one mapping of capability and PPC access, PPC code is dynamically generated when a PPC capability is created at component creation time. Likewise, the PPC code is zeroed out to remove access upon capability revocation. Similarly, in thread dispatch, a protected dispatch gate dispatches to threads for which the schedulers have capabilities in their capability table. When a thread is added to (removed from) the scheduler's capability-table, Janus provides (removes) dispatch gate access to the corresponding thread's DCB.

MPKs present another challenge by constraining only read/write permissions for load/store instructions, without restricting execution permissions [13], [22]. Thus, Janus's callgates and dispatch gates must include logic to prevent unintended components from leveraging callgates and dispatch gates to which they should not have access. Both the callgate and dispatch gate code validate that they are being executed by the proper component and thread (§V-D).

**Control Flow Integrity for PPC.** Unlike existing callgate implementations [25], [26], the user-level invocation stack provides expected "return" semantics: that each return corresponds to a previous call. The invocation stack ensures that a PPC returns to the proper client, at the correct instruction and stack pointer.

**Preventing mid-gate hijacking.** While Janus ensures a component only has access to PPC callgates if it has the corresponding kernel capability, malicious components can still intentionally jump into the middle of a gate in an attempt to bypass its checks. Were a malicious component to bypass the necessary checks to validate execution by a proper component, it could hijack the callgate to illegally switch to other protection domains. Janus detects these malicious jumps into the middle of a gate using tokens that are obtained at the start of the gate, and checked for validity at the end. A jump into the middle will not properly acquire the token, which will be detected. Details are in §V-D.

**Limiting Access to MPK Instructions.** MPK-based isolation requires that applications never directly execute the `wrpkru/xrstor` instructions to change active `PKRU` domains thereby bypassing Janus protection. To ensure that no `wrpkru/xrstor` instructions exist in component binaries outside the Janus callgates, we adopt ERIM's static binary scanning [25] approach. A thorough analysis [25] of five large Linux binary distributions (including over 100s of MLoC), found only 1213 wrpkru/xrstor instruction sequences, and all were addressed with standard binary rewriting tools. We found no instances of these instructions outside of gates in Janus.

### *B. Janus Predictability Analysis*

The user-level PPC path, similar to the kernel implementation, is bounded including no loops. Call and return paths each access only two entries on the stack, and the pointer to the head of the stack. Janus ensures that the kernel PPC path for synchronization between the kernel and user-level stacks is also bounded. The number of user-level invocation stack entries it must process is bounded by the maximum PPC call-depth, which is a constant. As such, Janus has predictable PPC control paths for both user- and kernel-level implementations.

The dispatch gate's logic includes no loops, and is bounded – involving only looking up the DCB, and directly updating it and the SCB. Same as with the kernel path, the dispatch operation is predictable. When the kernel is activated (due to a system call or interrupt), it must determine the active thread. It synchronizes with previous protected dispatch logic by deriving the currently active thread from the SCB. As such, dispatch retains predictable execution properties for both user- and kernel-level implementations.

Synchronization between user-level PPC and dispatch does impact the performance of kernel operations. To assess the impact of this, we induce synchronization overheads by making multiple direct PPCs and dispatch actions, then measuring the performance of a kernel-level PPC. This requires synchronizing both with SCB and user-level invocation stacks, and we find that the kernel PPC slows from 1129 to 1445 cycles – a 27% slowdown. With Janus, users can explicitly optimize (through the high-level configuration of address spaces) component PPCs to use the direct, user-level paths where heightened performance is beneficial, and pay a small cost of synchronization otherwise.

## V. Implementation

This section provides a detailed description of the implementation of MPK-based callgates for PPC (§III-B) and the dispatch gate for direct, protected thread dispatch (§III-C). This section also details the state synchronization between the user and kernel, as well as the implementation of security properties in §IV-A.

### *A. MPK-based Callgates*

According to §III-B, Janus allows clients to make PPC to servers via an MPK-based callgate if the client possesses the necessary capability in its capability table to invoke the corresponding server. The callgates are generated by a trusted constructor component which is responsible for creating components and allocating capabilities according to a system specification. The constructor keeps track of the address spaces, capabilities and PPCs, enabling it to generate callgates only when adding such a capability. The callgate switches the active protection domains by updating the `PKRU` register with `wrpkru`. The following pseudocode is abridged callgate code. The callgate starts by saving the caller's state. From line 8 to line 12, the callgate switches to the MPK system protection domain to push entries into the user-level invocation stack in lines 14 and 15. Subsequently, a second protection domain switch is performed to the callee's protection domain.

Implementation details regarding the security and integrity of the callgate are discussed in §V-D.

```
1  JANUS_SAVE_CALLER_STATE
2  ; Save AUTHENTICATION_TOKEN
3  mov $AUTHENTICATION_TOKEN, %r15
4  ; Validate the PPC is from the correct caller
5  rdpkru
6  cmp $CALLER_PROTDOM, %rcx
7  jne handle_abuse
8  mov $SYS_PROTDOM, %rax ; Janus system domain
9  xor %rcx, %rcx
10 xor %rdx, %rdx
11 ; Switch to Janus MPK system protection domain
12 wrpkru
13 ; Push PPC capability and %rsp into INVSTK
14 JANUS_ULINV_GET_INVSTK
15 JANUS_ULINV_PUSH_INVSTK
16 mov $CALLEE_PROTDOM, %rax
17 xor %rcx, %rcx
18 xor %rdx, %rdx
19 ; Switch to callee
20 wrpkru
21 cmp $AUTHENTICATION_TOKEN, %r15
22 jne handle_abuse
23 call callee_fn
```

### B. Thread Dispatch

As described in §III-C, Janus enables user-level direct switches to threads that were not preempted. The scheduler determines whether the current thread dispatch is eligible for using user-level dispatch by checking if the DCB of the next thread has its context (instruction and stack pointer). Otherwise, it uses the kernel dispatch to switch to the preempted thread. Janus implements a user-level thread dispatch gate in the scheduler. The following psuedocode is an abridged version of its code.

```
1  SAVE_THREAD_STATE
2  ; Check if we are in the scheduler
3  mov %rsp, ($ADDR_IN_SCHEDULER)
4  mov $AUTHENTICATION_TOKEN, %r15
5  mov $NEXT_DCB_ADDR, %rsi
6  mov $CURR_DCB_ADDR, %rax
7  ; Switch to system MPK protection domain
8  JANUS_SWITCHTO_SYSTEM_PROTDOM
9  ; Save sp, ip for current thread
10 mov $RESTORE_ADDR, (%rax)
11 mov %rsp, 8(%rax)
12 mov 8(%rsi), %r14
13 mov (%rsi), %r13
14 ; Check if next thread is preempted
15 cmp $0, 8(%rsi)
16 je slow_path
17 JANUS_UPDATE_SCB_ACTIVE_THREAD
18 ; Fast Path
19 JANUS_SWITCHTO_SCHED_PROTDOM
20 cmp $AUTHENTICATION_TOKEN, %r15
21 jne handle_abuse
22 mov %r14, %rsp
23 jmp *%r13
```

The protected dispatch assumes that registers outside of instruction and stack pointers have already been saved to the stack. To save the instruction and stack pointers, lines 10-11 move them to their corresponding DCB entry. We note that these values being saved indicates to future dispatches to this thread that protected dispatch can be used as the thread was not preempted. Lines 12-13 read these register values for the thread being switched to, and line 15 checks the `sp` of the next thread. If zero, this indicates the next thread was preempted, and the scheduler proceeds with the slow path through the kernel. Otherwise, if there are valid sp and restore address, the scheduler continues with the fast-path, updating the active thread in the SCB on line 17. Finally, the scheduler will switch to the destination thread update by restoring its stack and instruction pointer (in line 22 and 23).

Janus offers a secure protected dispatch gate as the SCB and DCB track critical information including the current active thread id which is also used in PPC. System designers configure Janus for either direct dispatch with lower overhead or protected dispatch with enhanced security. With protected dispatch, the scheduler does not require trust outside of its core functionality, as SCB and DCB updates are performed in protected code. Similar to PPC, using `wrpkru` to switch protection domains requires additional checks to prevent malicious accesses, details in §V-D.

### C. User-kernel Synchronization

As discussed in §III-B and §III-C, Janus leverages three user-space structures for synchronization between user and kernel state: the user-level invocation stack, SCB and DCB. When the kernel attempts to determine the active thread, it checks if its (per-core) variable tracking the current thread differs from the scheduler's SCB. If so, the inconsistent values mean that protected dispatches have updated the active thread, thus the kernel updates its current thread. To determine the currently active component, Janus is able to locate the corresponding user- and kernel-level invocation stacks of the active thread. Each of the kernel's invocation stack entries tracks the corresponding offset in the user-level stack active when an address space is invoked (through kernel IPC). To synchronize execution contexts, Janus's kernel iterates from this offset to the current user-level invocation stack top, each iteration identifying the next component that was invoked through PPC. The top of the stack denotes the active component.

Both PPC and dispatch paths are lock-free as they must assume they are preempted at any point. Thus, key variables serve as synchronization points: the reference to the head of an invocation stack for PPC, and the active thread id for protected dispatch. Only when these are updated is the gate's operations "committed", thus accessible to the kernel. Additionally, references to components or threads placed into shared structures are always in the form of capabilities, thus the kernel must translate (through the corresponding) capability tables, which component or thread is being referenced.

### D. Implementation of Security Properties

Janus's callgate and thread dispatch gate implement the security properties described in §IV-A.

**Gate access control.** Both callgate and dispatch gate must validate that the proper client is calling the code. PPC callgates use a similar approach to Underbridge [13]. As shown in the callgate implementation in §V-A, the callgate retrieves the active domain via `rdpkru` (line 5), and compares it to the

domain that is intended to use the gate (line 6). On return, the callgate makes similar checks by validating that the component is returning from a call, thus it is at the head of the invocation stack. To verify if the scheduler is calling the dispatch gate, the gate in §V-B issues a store to a pre-defined global address only valid in the scheduler's memory (line 3).

**Preventing mid-gate hijacking.** To prevent a malicious component from jumping into the middle of a gate, Janus's implementation of callgates and the thread dispatch gate relies on a randomly generated 64-bit AUTHENTICATION_TOKEN loaded into a register at the start of the gate, which is later confirmed to properly hold the correct token. This detects jumps into the gate, past where the token is loaded. It is highly unlikely for a malicious adversity to guess the value of the AUTHENTICATION_TOKEN given the $2^{64}$ namespace.

**Limiting Access to MPK Instructions.** Janus adopts ERIM's static binary inspection which scans a sliding window over each component's code segment for the `wrpkru/xrstor` instruction sequences at system image creation time [25]. We have not found any components that contain such sequences.

### *E. Split, Address Space, and Page-Table Management*

Janus decouples address spaces and protection domains by enabling the creation of multiple component protection domains into shared address spaces. This enables fast PPC between those components. The *split* operation (shown in Figure 4) enables the creation of multiple separate virtual address spaces that all share a set of PPC-accessible components. Component page-tables are safely created from constituent memory frames using retype [10], [9] operations, and operations to link page-table nodes. Janus adds new operations into the library that abstracts component creation: a virtual address space abstraction, and a split operation. Split is implemented by linking the shared component's page-tables (at the first level) into the different virtual address spaces. As the component's virtual memory is shared by aliasing within the page-table, only a single system call is required to link the second level of the shared component's page-table into the new address space's page-table. Most of the cost of creating new components is in the page initialization operations (zeroing them out, or `memset`ing into them), so split adds negligible overhead, measuring around 7.3 $\mu$s on a Cincoze Dx1200 embedded system equipped with a 1.1 GHz processor with cold cache.

## VI. Evaluation

In this section, we evaluate the system-level overhead of Janus and performance of real-world applications on Janus comparing to other existing systems. In particular, in our evaluation, we (1) leverage microbenchmarks to evaluate Janus; (2) assess the extensibility of Janus by comparing with seL4 [50] on seL4 style IPC, the baseline Composite, and Linux; (3) evaluate the performance of the Patina [30], security-focused RTOS; and (4) demonstrate the performance of Janus by running a latency-sensitive application, `memcached`.

| | Janus | Composite | Linux | seL4 |
|---|---|---|---|---|
| IPC | 408(*416*)**496** | 1129(*1226*)**1235** | 8741(*9379*)**218480** | 1273* |
| Dispatch | 718(*866*)**897** | 2226(*2708*)**2726** | 1720(*1858*)**14185** | 806(*1193*)**1202** |

TABLE I: Microbenchmarks study the round-trip latencies of inter-process communication and thread dispatch. We use `pipe()` as the Linux equivalent of IPC. All results are in cycles. We organize the results as average(*99th percentile*)**Maximum**. We only show the average IPC round-trip latency results of seL4 (marked with *), since there's no direct equivalent for round-trip IPC in seL4 microbenchmarks. Detailed results of seL4 are in Table II.

All experiments are conducted on a Cincoze Dx1200 embedded system equipped with Intel(R) 12th i9-12900TE CPU, utilizing 8 performance cores at 1.1GHz. The embedded system is equipped with an Intel X550T 10GbE NIC for networking. A client machine with an Intel(R) i7-14700F CPU with 16 cores and Intel(R) X540 10GbE NIC is used to drive the workload generation. The Linux evaluations are performed on kernel version 6.8.0-47-generic. For `memcached` evaluations, the client machine uses a modified `mcblaster` open-loop workload generator to measure throughput and round-trip latency. Janus uses DPDK version 22.03.0-rc0 and `memcached` version 1.4.39 while the Linux comparison cases use `memcached` server version 1.6.14.

### *A. Microbenchmarks*

We first measure the overhead of inter-process communication (IPC) and thread dispatching in Janus and compare to the Composite-baseline, Linux, and seL4 in Table I. Both Composite and Janus thread-migration-based PPC for communication between protection domains. We use a round-trip pair of `pipe()`s as the Linux equivalent of inter-process communication. To evaluate the overhead of thread dispatching, we measure the latency of one thread yielding to another.

**Discussion.** Janus achieves the lowest round-trip latency IPC across all metrics thanks user-level control operations. Composite and seL4 show similar IPC latency but are nearly 3 times slower than Janus, while Linux `pipes` are significantly slower than both systems. Section VI-B provides a detailed analysis of L4-style IPC latency. Janus reduces the cost of PPC to that of a Linux system call (*e.g.*the `getppid()` call) of around 304 cycles. Protected dispatch shows similar benefits: the average thread dispatch latency of seL4 is 100 cycles larger, but Janus improves by over 1400 cycles over Composite. This cost is larger in Janus and Composite as a PPC to the scheduler *and* a yield to another thread are required – a consequence of the reliable, specializable scheduling policy. These results demonstrate that the design of Janus's control-plane operations significantly reduces IPC and thread dispatching latencies.

### *B. Configurable Control Mechanisms and Policies*

Janus enables the user-level definition of specialized control policies. By avoiding kernel interactions on the fast path, even performance-sensitive systems can leverage increased

spatial and temporal isolation. To showcase Janus's ability to customize core control policies and mechanisms, we implement L4-style synchronous rendezvous between threads [12], and we evaluate Janus's benefit when used with an existing secure, component-based RTOS.

**L4-style IPC** ("L4-IPC" henceforth) has a number of trade-offs versus thread-migration-based PPC. A benefit of L4-IPC is that a server can avoid multi-threaded concurrency by using a single server thread to receive client requests.

L4-IPC is based on a set of optimizations [14]: (1) threads synchronously rendezvousing to conduct IPC; (2) a specialized API that uses `call` and `reply_and_wait` to require only two system calls; and (3) a fast-path that uses direct switch and scheduler policy bypass. We've implemented an extension to the scheduler component in 112 lines of code that integrates all of these optimizations to enable synchronous rendezvous between threads-style IPC in Janus. It requires synchronous invocations to the scheduler component (*e.g.* for `call`), dispatching between threads in the scheduler, and customized logic for direct switch and policy bypass. Thus, our L4-IPC extension uses the Janus facilities to extend the system's control facilities.

Table II depicts the overhead for round-trip IPC using Janus's L4-IPC. We show four different protection domain configurations: (1) C1: all components share the same address space, use kernel-bypass IPC, and fast dispatch, (2) C2: same as C1, but using protected dispatch, (3) C3: all three components use separate address spaces, using IPC through kernel, (4) C4: the client and server each *split* from the scheduler (§III-D), thus enabling kernel-bypass IPC, but requiring dispatch between address spaces. We compare against seL4's fast-path IPC. Since seL4 has no direct round-trip IPC equivalent in its benchmarks, we use the combined average latency of `SeL4_Call` and `SeL4_ReplyRecv` for comparison.

**Discussion.** Surprisingly, Janus's support for L4-IPC is *faster* than seL4 using PPC callgates and fast dispatch, despite enabling configurable scheduling and coordination policies and emulating (without significant optimization) the fastpath in seL4. The protected dispatch configuration requires two switches to the MPK system protection domain, thus incurring around 700 cycles of additional overheads in Janus's L4-IPC. The benefit of this approach is that faults in the scheduler don't necessary cause full system failure.

When using separate address spaces for all three testing components, IPC and thread dispatch must go through the kernel resulting in a significant slowdown with average round-trip latency over 4000 cycles and 99th percentile tail latency over 5000 cycles. The comparison between single and split address spaces demonstrates the performance difference between kernel mechanisms and the Janus user-level control-plane: a reduction in latency by over 70%.

**Patina [30].** RTOSes provide basic facilities for message passing, event management, and synchronization. Patina is an RTOS built on Composite that provides these in a multi-component system focused on isolation. Results from the Patina paper show competitive performance with Linux, but significant overheads for some operations, for example, context switches. Janus transparently replaces the control-plane operations in Patina, this enabling our assessment of how such an existing system benefits from their increased efficiency.

| System Configuration | Avg | 99%th | Max |
|---|---|---|---|
| seL4 Round-Trip | 1273 | * | * |
| *SeL4_Call* | *636* | *640* | *641* |
| *SeL4_ReplyRecv* | *637* | *640* | *641* |
| C1:Janus Single Addr Space, Fast Dispatch | 1078 | 1087 | 1102 |
| C2:Janus Single Addr Space, Protected Dispatch | 1780 | 2163 | 3042 |
| C3:Janus Separate Addr Spaces, Kernel dispatch | 4484 | 5435 | 5968 |
| C4:Janus Split Addr Spaces, Kernel dispatch | 3680 | 3766 | 4284 |

TABLE II: Round-trip latency of L4-style IPC comparing seL4 against Janus with different configurations. All results are in cycles. seL4 benchmarks do not have a direct equivalent of round-trip IPC, thus we only show average latency which is the sum of the average latency of `SeL4_call` and `SeL4_ReplyRecv`.

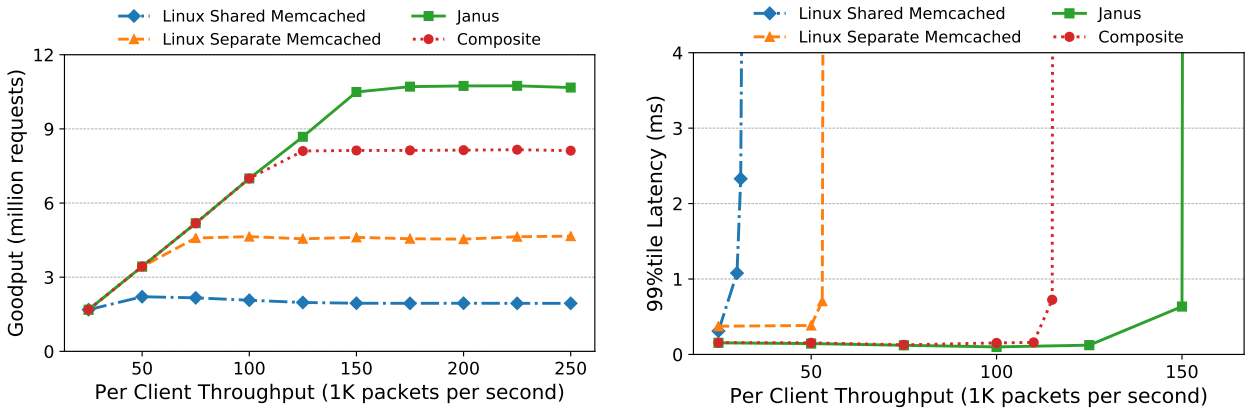
We execute Patina [30] enhanced with Janus in Table III. We compare Janus with Composite and Linux across multiple Patina benchmarks: (1) Channel communication between two threads across address spaces. In Linux, we evaluate both sockets and pipes and use pipes as they are faster. (2) Channel communication between two threads across address spaces with an event manager component notifying threads when new events have been triggered. The Linux comparison case uses `epoll` to await the event. (3) Contented semaphore. (4) Contented mutex. Both semaphore and mutex have a lower-priority lock holder that activates higher-priority contender. Results for uncontended mutex and semaphores avoid any inter-component coordination, thus results are the same between Composite and Janus.

**Discussion.** Compared to Composite, Janus significantly reduces latency across all metrics for channel communication with event management (from over 9000 cycles to 3500 cycles) and without event management (from over 5000 cycles to around 1300 cycles). These improvements demonstrate that Janus's ability to reduce overheads for control-plane operations is a strong enabler for multi-component systems with strong isolation properties. Linux pipes are nearly 7 times slower than Janus which shows that Janus with Patina has better than acceptable performance. For mutex and semaphore operations, Composite is around 10% better than the Linux equivalent, but Janus improves over Linux by almost 60%.

We also evaluate alternative approaches for user-level policy configurations, including those that rely on language runtimes such as go. Despite user-level definition of many concurrency operations in go, facilities must exist for synchronizing with the kernel (*e.g.* futexes guard the core runtime). Due to this, the cost for go to RPC between goroutines (*i.e.*user-level threads) using channels is on average 10717 cycles, nearly 10 times slower than Janus's channel communication without event management.

| | Composite Patina RTOS | | | Janus Patina RTOS | | | Linux | | |
|---|---|---|---|---|---|---|---|---|---|
| | Average | 99% tail | Max | Average | 99% tail | Max | Average | 99% tail | Max |
| Channel, Direct Switch | 5663 | 5787 | 9460 | 1330 | 1353 | 7090 | 8741 | 9379 | 218480 |
| Channel, Event Management | 9813 | 9996 | 22371 | 3536 | 3652 | 13810 | 10340 | 13026 | 186897 |
| Semaphore Contended | 5704 | 5771 | 9806 | 2640 | 2678 | 6369 | 6158 | 6364 | 11855 |
| Mutex Contended | 5628 | 5688 | 8507 | 2789 | 2842 | 4839 | 6592 | 6810 | 13356 |

TABLE III: Patina Overheads in Cycles of Janus and Composite with equivalent Linux operations.




(a) Goodput Comparison

(b) 99th Percentile Latency of Requests

Fig. 5: Evaluations on goodput and 99th percentile latency of the `memcached`. Comparing Janus against Composite, Linux with separate `memcached` instances and Linux with shared `memcached` instance.

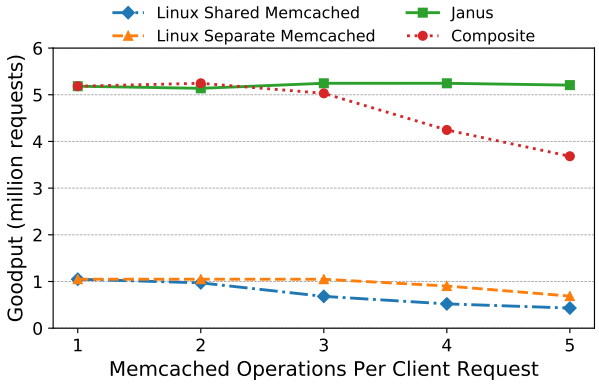
### C. In-Memory Key-Value Store

In this subsection, we investigate the benefits of creating control-plane abstractions tailored toward low-latency, high-throughput network-attached services. We further investigate the co-location of client, tenant code on the node with the service. This enables restricted hardware (*e.g.* in the edge) to densely host multi-tenant computations and services.

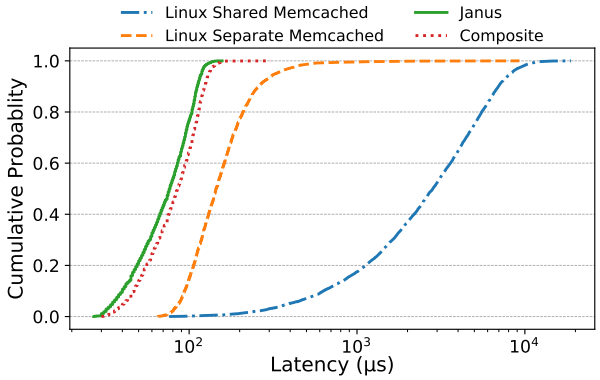
We deploy a well-known in-memory key-value store `memcached` as a component, with separate scheduler and NIC manager (for network I/O) components. Tenant components use PPC to query the `memcached` component. We use DPDK [51] to interact with the NIC, and isolate it in the NIC manager user-level component. To prevent malicious access and maximize resiliency, all of `memcached`, tenant computations, scheduler, and network I/O are isolated in separate protection domains. Consequently, each tenant must perform PPCs to transmit and receive packets via the NIC, as well as to execute `memcached` operations. In Janus, all of these components share the same address space. With Janus replacing control-plane operations, the `memcached` application benefits from both kernel-bypass IPC and user-level thread dispatch. We run in total 7 tenants across 8 cores with 1 specialized core devoted to packet receiving and demultiplexing.

We compare Janus with Composite and Linux. The Composite comparison case uses the same configuration as Janus, but instead of sharing the same address space, Composite has each component execute in separate address spaces thus relying on page-table-based isolation and thread dispatching through the kernel. We configure Linux in two different ways: (1) running `memcached` server with 8 threads across 8 cores which is a close replica of Janus's setup, and (2) running 7 separate `memcached` server instances, each with 1 thread to avoid contention and demultiplexing. The clients generate a 90% get, 10% set workload with a 16-byte key and 135-byte value as the default workload of `mcblaster`. Figure 5b shows the 99% tail latency of replies as the increase of the number of packets sent by the workload generator per second. Figure 5a depicts the changes in goodput as we increase the client sending rate.

**Discussion.** In Figure 5a, Composite achieves a goodput of around 8.1 million rps, while Janus reaches over 10.7 million, demonstrating a significant 32% increase by focusing on an efficient an predictable control-plan. This increased efficiency is also clear when focusing on 99th percentile latencies in Figure 5b. Composite starts to get overloaded when the per client throughput is greater than 120K packets per second, but Janus maintains a tight latency distribution until the per client throughput is greater than 150K packets per second. Linux with separate `memcached` instances achieves only 4.6 million goodput – about 43% of Janus's goodput. This is mainly due to overheads the Linux kernel in packet transmission. With a shared `memcached` instance, the goodput of Linux drops to around 22 million, primarily due to the cost of demultiplexing and event notification (and the associated lock




(a) Goodput Comparison on Multi-tenant `memcached`.




(b) Latency CDF with Two Operations Per Client Request

Fig. 6: Multi-tenant evaluation where each client request requires multiple `memcached` operations. We compare goodput and latency CDF results with Composite, Linux with separate `memcached` instance and Linux with shared `memcached` instance. Composite and Janus results are with per client throughput at 75K requests per second. Both Linux comparison cases process 15K requests per second per client.

contention), over 60% of the execution time is spent in `epoll`. When *underloaded* in Figure 5b, both Linux configurations have greater 99th percentile latency than Composite and Janus: 200 $\mu$s compared to around 70 $\mu$s. Linux with separate `memcached` instances begins to get overloaded when per client throughput is over 50K packets per second, while Linux with a shared `memcached` instance is overloaded when the per client throughput is greater than 30K – both hitting overload at 33% and 20% of Janus's capacity.

**Multi-tenant `memcached`.** Instead of making one get or set request per packet, real-world applications often require services that require aggregation of multiple `memcached` operations [52]. An edge mobile application might query multiple times for many proximate nodes, before replying to the client. We emulate this by allowing a single client request to be handled by isolated tenant components that aggregate data from multiple `memcached` queries. Composite and Janus implement tenant aggregation computations in separate components. For the Linux comparison cases, we implement tenants as separate processes, each binding to a specific server port to service client requests, and making requests to the `memcached` server using local sockets. We evaluate the goodput and the latency distributions of Composite, Janus, Linux with a shared `memcached` instance, and Linux with separate `memcached` instances. For Composite and Janus, each client *sends 75K requests per second*, while for Linux approaches, *each client sends only 15K requests per second*. These are the empirical values that maximize goodput. The evaluation is conducted on one million keys, to de-emphasize contention in `memcached`.

**Discussion.** Figure 6a depicts the change of goodput as increasing the number of `memcached` operations per client request. Note that the client throughput for Composite and Janus is five times larger than for Linux comparison cases. Both Linux comparison cases achieve goodput around 1 million rps while Composite and Janus achieve goodput over 5 million. The goodput of Linux with a shared `memcached` instance drops to 500K when having three `memcached` operations per client request, while Linux with separate `memcached` instances still maintains 1 million goodput. Janus maintains a high goodput of over 5 million even with 5 operations (PPCs) per request, whereas Composite drops to 3.5 million. This indicates Janus's ability to reduce control overheads.

Figure 6b presents the latency CDF when having two `memcached` operations per client request. Composite exhibits an average latency of 86 $\mu$s and 99th percentile tail latency of 149 $\mu$s. Janus decreases both: with an average of 77 $\mu$s and tail latency of 132 $\mu$s. Linux with separate `memcached` instances underperforms Janus with average latency 172 $\mu$s and 99th percentile tail around 625 $\mu$s. Linux with shared `memcached` instances is already overloaded and drops around 5% of the total requests due to demultiplexing and event notifications overheads.

## VII. Conclusions

This paper introduces Janus, which provides a fast, flexible, and secure control-plane. Janus focuses on reducing system overhead on two core control-plane operations: PPC and thread dispatch. Janus leverages hardware-based MPK to enable kernel-bypass execution of PPC and thread dispatch. It integrates MPK-based and traditional page-table-based protection domains, supporting over 15 memory protection domains.

We demonstrate Janus's ability to customize the control plane while out-performing optimized systems, providing performance improvements to existing RTOSes, and to providing effective control-plane operations for a low-latency, high-throughput, multi-tenant systems. Evaluation results show that Janus effectively reduces system overhead while maintaining strong isolation for both micro-benchmarks and real-world applications such as `memcached`. Janus achieves over 5x improvements in goodput while delivering 99th percentile latency approximately 3x lower for `memcached` services.

**Acknowledgements.** We'd like to thank our shepherd and reviewers for their time and effort that significantly improved this paper. This work is supported by NSF CPS 1837382 and ONR N000142212084. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

## REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[2] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013.

[3] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 361–378.

[4] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive scheduling for μsecond-scale tail latency," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2019.

[5] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 281–297.

[6] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "Ix: A protected dataplane operating system for high throughput and low latency," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[7] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," *ACM Trans. Comput. Syst.*, vol. 33, no. 4, Nov. 2015.

[8] K. Vipin, "Cannoc: An open-source noc architecture for ecu consolidation," in *2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2018.

[9] K. Elphinstone and G. Heiser, "From L3 to seL4 what have we learnt in 20 years of L4 microkernels?" in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 133–150.

[10] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[11] L. Soares and M. Stumm, "Flexsc: Flexible system call scheduling with exception-less system calls," in *Proceedings of the conference on Symposium on Operating Systems Design & Implementation*, 2010.

[12] J. Liedtke, "Improving IPC by kernel design," in *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 1993, pp. 175–188.

[13] J. Gu, X. Wu, W. Li, N. Liu, Z. Mi, Y. Xia, and H. Chen, "Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 401–417.

[14] J. Liedtke, "On micro-kernel construction," in *Proceedings of the 15th ACM Symposium on Operating System Principles*. ACM, December 1995.

[15] "x86 PCID Documentation. https://www.kernel.org/doc/Documentation/x86/pti.txt."

[16] A. Tatar, D. Trujillo, C. Giuffrida, and H. Bos, "TLB; DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 989–1007.

[17] "FreeRTOS: http://www.freertos.org, retrieved 5/1/13."

[18] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[19] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, S. Teodorescu, C. Raducanu *et al.*, "Unikraft: fast, specialized unikernels the easy way," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 376–394.

[20] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2019.

[21] V. A. Sartakov, L. Vilanova, and P. Pietzuch, "Cubicleos: A library os with software componentisation for practical isolation," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 546–558.

[22] "Intel 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html."

[23] J. S. Chase, M. Baker-Harvey, H. M. Levy, and E. D. Lazowska, "Opal: A single address space system for 64-bit architectures," *Operating Systems Review*, vol. 26, no. 2, p. 9, 1992. [Online]. Available: citeseer.ist.psu.edu/58003.html

[24] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke, "The Mungi single-address-space operating system," *Software Practice and Experience*, vol. 28, no. 9, pp. 901–928, 1998. [Online]. Available: citeseer.ist.psu.edu/heiser98mungi.html

[25] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1221–1238.

[26] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor:Intra-Process isolation for High-Throughput data plane libraries," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 489–504.

[27] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *ACM SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988.

[28] E. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "PKU pitfalls: Attacks on PKU-based memory isolation systems," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1409–1426.

[29] T. Close, "Acls don't," *HP Laboratories Technical Report*, 2009.

[30] S. Jero, J. Furgala, R. Pan, P. K. Gadepalli, A. Clifford, B. Ye, R. Khazan, B. C. Ward, G. Parmer, and R. Skowyra, "Practical principle of least privilege for secure embedded systems," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021, pp. 1–13.

[31] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel MPK)," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 241–254.

[32] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005.

[33] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xmp: Selective memory protection for kernel and user space," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 563–577.

[34] D. Peng, C. Liu, T. Palit, P. Fonseca, A. Vahldiek-Oberwagner, and M. Vij, "$\mu$switch: Fast kernel context isolation with implicit context switches," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2956–2973.

[35] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 437–452.

[36] J. Gu, H. Li, W. Li, Y. Xia, and H. Chen, "EPK: Scalable and efficient memory protection keys," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 609–624.

[37] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1607–1619.

[38] X. Wang, S. Yeoh, P. Olivier, and B. Ravindran, "Secure and efficient in-process monitor (and library) protection with intel mpk," in *Proceedings of the 13th European workshop on Systems Security*, 2020, pp. 7–12.

[39] M. Sung, P. Olivier, S. Lankes, and B. Ravindran, "Intra-unikernel isolation with intel memory protection keys," in *Proceedings of the 16th*

*ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020, pp. 143–156.

[40] Y. Xu, C. Ye, Y. Solihin, and X. Shen, "Hardware-based domain virtualization for intra-process isolation of persistent memory objects," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 680–692.

[41] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: a fast capability system," in *Symposium on Operating Systems Principles*, 1999, pp. 170–185. [Online]. Available: citeseer.ist.psu.edu/shapiro99eros.html

[42] J. B. Dennis and E. C. V. Horn, "Programming semantics for multiprogrammed computations," *Commun. ACM*, vol. 26, no. 1, pp. 29–35, 1983.

[43] B. Ford and J. Lepreau, "Evolving Mach 3.0 to a migrating thread model," in *WTEC*, 1994.

[44] J. Bruno, J. Brustoloni, E. Gabber, A. Silberschatz, and C. Small, "Pebble: A component-based operating system for embedded applications," in *Proc. USENIX Workshop on Embedded Systems*, 1999, pp. 55–65.

[45] G. Parmer, "The case for thread migration: Predictable IPC in a customizable and reliable OS," in *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT)*, 2010.

[46] P. K. Gadepalli, R. Pan, and G. Parmer, "Slite: OS support for near zero-cost, configurable scheduling," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020, pp. 160–173.

[47] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis, "ghost: Fast & flexible user-space delegation of linux scheduling," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 588–604.

[48] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain Keys–Efficient In-Process Isolation for RISC-V and x86," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1677–1694.

[49] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen, "Skybridge: Fast and secure inter-process communication for microkernels," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–15.

[50] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. Big Sky, MT, USA: ACM, Oct 2009.

[51] "Intel Data Plane Development Kit (DPDK). http://dpdk.org/."

[52] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman, "Splinter:{bare-metal} extensions for {multi-tenant}{low-latency} storage," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 627–643.