Pegasus: Transparent and Unified Kernel-Bypass Networking for Fast Local and Remote Communication

Dinglan Peng
Purdue University
West Lafayette, IN, USA

Congyu Liu
Purdue University
West Lafayette, IN, USA

Tapti Palit
Purdue University
West Lafayette, IN, USA

Anjo Vahldiek-Oberwagner
Intel Labs
Berlin, Germany

Mona Vij
Intel Labs
Hillsboro, OR, USA

Pedro Fonseca
Purdue University
West Lafayette, IN, USA

Abstract

Modern software architectures in cloud computing are highly reliant on interconnected local and remote services. Popular architectures, such as the service mesh, rely on the use of independent services or sidecars for a single application. While such modular approaches simplify application development and deployment, they also introduce significant communication overhead since now even local communication that is handled by the kernel becomes a performance bottleneck. This problem has been identified and partially solved for remote communication over fast NICs through the use of kernel-bypass data plane systems. However, existing kernel-bypass mechanisms challenge their practical deployment by either requiring code modification or supporting only a small subset of the network interface.

In this paper, we propose Pegasus, a framework for transparent kernel bypass for local and remote communication. By transparently fusing multiple applications into a single process, Pegasus provides an in-process *fast path* to bypass the kernel for local communication. To accelerate remote communication over fast NICs, Pegasus uses DPDK to directly access the NIC. Pegasus supports transparent kernel bypass for unmodified binaries by implementing core OS services in user space, such as scheduling and memory management, thus removing the kernel from the critical path. Our experiments on a range of real-world applications show that, compared with Linux, Pegasus improves the throughput by 19% to 33% for local communication and 178% to 442% for remote communication, without application changes. Furthermore, Pegasus achieves 222% higher throughput than Linux for co-located, IO-intensive applications that require both local and remote communication, with each communication optimization contributing significantly.

CCS Concepts: • **Software and its engineering → Operating systems**.

Keywords: operating systems, kernel-bypass networking

1 Introduction

Cloud software design and deployments increasingly depend on fast communication between loosely coupled services [12, 31, 34]. These services run in independent processes, containers, or VMs within and across nodes, and communicate extensively with each other via standardized protocols [34] (e.g., HTTP or gRPC). In contrast to traditional monolithic applications, architectures based on loosely coupled services help developers build modular, scalable, and fault-tolerant applications and have been adopted by major data center operators, including Google [36] and Meta [61].

The benefits of breaking applications into multiple services come at a cost: the fine-grained communication between services, including *local communication* that happens within a node, and *remote communication* that happens across nodes, causes significant application overhead. Profiling in data centers by Google [36] shows that up to 27% of cycles are spent on communication in user space such as serialization and data copy, and nearly another 20% of cycles are spent in the kernel, many of which are I/O-related.

Kernel-bypass techniques allow applications to directly access the network hardware, making them an effective technique to reduce the remote communication overhead. The significant overhead caused by the kernel network stack has motivated a thriving line of work on kernel-bypass networking libraries and operating systems that generally rely on custom APIs [6, 20, 21, 53, 69, 78].

An alternative approach co-locates applications on the same node, making communication that would otherwise be remote local. When co-location is possible, this approach offers the advantage of bypassing the network entirely and allows fast local communication mechanisms like shared memory [35, 43, 79]. As many communication patterns are synchronous, such as remote procedure calls (RPC), these fast local communication mechanisms usually require process coordination. For example, shared memory is often used with futexes in Linux for coordination, which can be expensive. Our experiment (Table 1) shows that it can take 1.37 μs to wake up a thread using a futex. Prior work [43] also shows similar latency – e.g., more than 2.8 μs for a process wakeup, which is even significantly higher than the latency of a network RDMA operation (1.6 μs), thus prohibitive for applications that are I/O intensive.

To minimize the overhead of thread coordination, recent works [9, 22, 42, 65] implement microservice runtimes that support running multiple services written in specific languages such as Golang and Python in the same process, so that they can achieve fast user-space scheduling and synchronization without using slow OS mechanisms. However, these systems are language-specific and do not support general, unmodified applications.

Although both kernel bypass and co-location can effectively reduce communication overhead, the existing frameworks have important limitations that hinder their practical deployment in real-world applications. First, existing frameworks typically have limited compatibility with existing cloud infrastructure and applications. In particular, they do not support the dominant interfaces of real-world deployments, such as the POSIX API, and some require significant code changes and application refactoring [21, 78] or are limited to certain languages [22, 42]. Second, existing systems do not provide a unified interface for both local and remote communication. As a result, developers have to decide which interface to use for each pair of interacting services. This forces developers to consider the deployment and orchestration of the services during application development and limits how the services are scheduled on the nodes as the services that use the interface specific to local communication must be co-located on the same node.

We argue that addressing the challenges posed by current and emerging data center paradigms requires a *practical* and *unified* approach for fast I/O. In particular, addressing the performance challenges of emerging applications requires (a) transparently supporting current data center deployments and (b) removing the kernel from the critical path for both local and remote communication.

Our approach co-locates and *fuses* multiple Linux processes into the same address space. This approach is particularly suited for *symbiotic processes*, which are increasingly prevalent in data center workloads. A group of processes is symbiotic if the processes (1) extensively and frequently communicate with each other and (2) are part of the same application. As an example, symbiotic processes are inherent to the widely used service mesh architecture [4] where a process known as a *sidecar proxy* is attached to each service process to transparently implement advanced features, such as security- and logging-related features. For such symbiotic processes, our approach co-locates and fuses them when possible, to leverage optimized local *fast path*, and applies kernel-bypass networking for remote communication made by these processes. While bypassing the kernel with optimized fast paths, our approach ensures that this is transparent to applications by maintaining compatibility with existing interfaces.

Several challenges need to be addressed to realize this approach. First, we need to transparently virtualize process resources, despite the large and complex Linux interface, so that multiple programs correctly execute while co-existing in the same Linux process. Second, we need to ensure isolation between symbiotic processes to preserve logical independence so that failures do not propagate across them. Finally, we need to introduce minimal overhead and significantly improve applications' performance.

To fulfill this vision, we built Pegasus, a local and remote kernel-bypass system that supports real-world applications without modifications. Pegasus implements a protected user space monitor that transparently loads and runs multiple applications in a single Linux process and implements process- and I/O-related OS services, including scheduling and networking, in user space so that applications can communicate without kernel invocations. Pegasus supports existing applications without modifications by operating at the Linux ABI level. Furthermore, Pegasus implements efficient in-process isolation, leveraging Intel® MPK-based memory isolation and fast implicit kernel context switches [56], to maintain functional isolation. Hence, despite running several applications in the same address space, Pegasus ensures that a faulty application does not affect others. Using this approach, Pegasus optimizes both local and remote communication using a practical and unified approach that bypasses the kernel network stack.

Pegasus is designed to be compatible with complex real-world applications and deployable with zero effort, supporting existing Linux applications at the binary level. Furthermore, Pegasus supports the Open Container Initiative (OCI) runtime specification [70] for the integration with existing container management and orchestration platforms, such as Docker and Kubernetes. To the best of our knowledge, Pegasus is the first kernel-bypass framework to optimize both local and remote communication efficiently, securely, and practically.

Our evaluation shows that Pegasus can significantly improve application performance for widely-used, complex applications, including Redis, Nginx, Memcached, Nodejs, Envoy, and Caddy, without any application modification. Compared with Linux, Pegasus achieves throughput improvement that ranges from 19% to 33% for local communication and 178% to 442% for remote communication. Furthermore, Pegasus achieves 222% higher throughput than Linux for the proxied Web server that requires both local and remote communication, with both fast paths contributing significant improvement. Pegasus has comparable or better performance than existing kernel-bypass networking systems that require application changes. For example, when applied to Redis, the throughput with Pegasus is 153% higher than Demikernel and only 1.5% lower than F-Stack, both of which require significant application changes.

2 Emerging Communication Paradigms

The kernel has long been identified as a communication bottleneck [6, 21, 43, 53, 78], especially in the context of remote communication across different nodes over fast NICs. Recently, due to the rise of communication paradigms such as the *service mesh*, a new manifestation of this problem has arisen. Emerging communication paradigms, such as the service mesh and the actor model, intensively use local communication using traditional kernel primitives, such as pipes and local sockets, where the peer process is located in the same pod or host. The communication overhead between such *symbiotic* processes – i.e., processes that are physically co-located and frequently communicate with each other – is substantially affected by the overhead introduced by the kernel. As shown in SocketDirect [43], Linux IPC mechanisms, such as pipes and sockets cause 8 to 11 μs round-trip latency, which introduces significant overhead to symbiotic processes. Such latency results from the complex kernel components, such as the network stack and the scheduler, as well as hardware overhead, such as ring changes, page table switches, and increased TLB and other cache misses.

Existing research [9, 22, 25, 32, 35, 42, 43, 61, 65, 66, 79] proposes techniques to accelerate the communication between symbiotic processes running on the same host. They typically use fast local communication mechanisms that bypass the slow kernel network stack. We discuss the benefits and limitations of these mechanisms in the following paragraph.

2.1 Removing the Kernel from the Critical Path

Achieving fast local and remote communication requires removing the kernel from the critical path. In the case of local communication, this is typically achieved by using shared memory primitives. However, this is far from a complete solution. While this optimizes the data path, the control path still poses a challenge. Shared memory abstractions require synchronization between the communicating applications, which requires using kernel synchronization primitives, such as *futexes*. Therefore, this approach does not completely eliminate the kernel from the critical path—the kernel overheads of scheduling and waking up a task are still present. In fact, as we show in §6.2, it takes 1.37 μs to wake up a thread using a futex, and can potentially negate much of the benefits of using shared memory primitives. To address the problem of slow thread coordination, recent works [9, 22, 42, 65] propose to apply user-space scheduling and synchronization to microservice runtimes that support running multiple services in the same process, so as to avoid slow OS mechanisms. However, they are language-specific and do not apply to general, unmodified applications.

In the case of kernel bypass for remote communication, most existing solutions [21, 33, 53, 69, 78] require the program to be rewritten using custom APIs that differ from traditional POSIX APIs. This is necessary because traditional POSIX APIs do not provide support for the optimizations necessary to fully benefit from kernel bypass, such as zero-copy transfers and lightweight user-space threading. To this end, Demikernel [78] proposes the PDPIX API, and Shenango [53] and Caladan [21] provide a custom threading and networking API. To further improve performance, Cornflakes [58] removes serialization buffers and communicates with the NIC with zero copy. However, such approaches require the developers to rewrite the programs and this is a significant obstacle in the wide adoption of such kernel-bypass techniques. For example, rewriting the Redis server to use Demikernel required the addition of 2478 lines of code, which involved a significant effort.

A few systems, such as libVMA [48] and TAS [37], use `LD_PRELOAD` to intercept the C library calls to support unmodified binaries. However, this approach has several limitations. First, `LD_PRELOAD` cannot guarantee full coverage of the system call interception because applications may invoke system calls directly, such as the Golang programs do. Second, and more importantly, it is challenging for such systems to implement correct semantics of system calls efficiently without user-space scheduling, in particular, when handling blocking system calls. For example, libVMA only allows one epoll file per process and cannot support applications that use `accept()` and `connect()` at the same time, which significantly limits its compatibility with real-world applications, such as proxy servers. A concurrent work, Junction [20], also recently proposed achieving Linux ABI compatibility for kernel-bypass networking. However, Junction lacks in-process isolation, which is a critical prerequisite for securely fusing symbiotic processes. Also, to maximize the performance, Junction does not protect the library OS, allowing the applications to directly access the NICs, which increases the attack surfaces (e.g., a faulty application can sniff and send raw packets). Finally, Junction has limited support for some kernel functions such as file systems.

2.2 Practical, Unified Kernel Bypass

The dual challenge of *seamlessly* adopting kernel-bypass techniques for remote communication, and the necessity of adopting kernel bypass for local communication for symbiotic processes leads us to propose a *unified* and *transparent* kernel-bypass mechanism for *all* communication, local or remote. To this end, we propose Pegasus, a kernel-bypass mechanism for providing a fast path for both local and remote communication, which retains full Linux ABI compatibility to support unmodified applications.

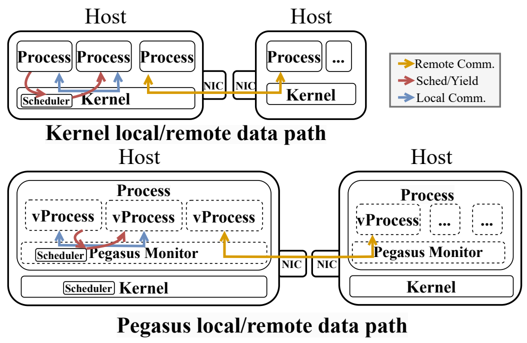


Figure 1. Pegasus architecture of local and remote data paths compared to the Linux kernel.

Inspired by ServiceWeaver [22] which decouples service physical boundaries from their logical boundaries by running multiple Golang microservices in the same process, we extend the idea to fusing multiple Linux processes into the same address space. Pegasus fuses symbiotic programs into the same address space and uses a local ring buffer for communication. To prevent the need for any kernel involvement, Pegasus implements critical OS functions, such as scheduling and memory management, in user space. Effectively, Pegasus implements virtual processes in user space. With such optimization, the complex kernel network stack is replaced with a simple user-space fast path, and context switches can be eliminated during communication with the help of user-space scheduling. For remote communication, Pegasus uses DPDK to directly access NICs. To achieve Linux ABI compatibility, Pegasus intercepts system calls made from programs, delegating them to fast paths if they are communication-related. Performing scheduling in user space allows Pegasus to transparently implement features essential for kernel bypass, such as lightweight threading, without requiring any modifications to the programs.

While fusing programs to optimize local communication can lead to performance improvements, this performance improvement should not come at the cost of reduced fault tolerance and isolation guarantees. To provide isolation guarantees, Pegasus relies on Intel® MPK to isolate the memory of different programs in the same address space. To prevent a single program from crashing the process, Pegasus intercepts and gracefully handles critical signals, such as `SIGSEGV`. A fault in one program only affects itself.

3 Pegasus Design

Pegasus's goal is to provide a *transparent* approach that allows *fast* local and remote communication. Achieving the dual goals of both transparency and performance requires Pegasus to assume control of the OS functions that are critical to communication and implement them in user space.

3.1 System Model and Threat Model

System Model. In Pegasus, a process and its virtual address space are split into multiple unprivileged program domains and a privileged monitor intercepting interactions with the host kernel, such as signals or memory management. Pegasus assumes an oracle (e.g., the user or a framework) that specifies which applications are symbiotic, and should be loaded into the same process.

Each loaded program is represented as a `vProcess`. A `vProcess` consists of one or more `vThreads`, each representing a thread of the process, that is mapped to a user-space task managed by Pegasus's scheduler. A `vProcess` can access OS functions through the Linux ABI (e.g., system calls). A subset of the OS functions is directly handled by the OS kernel and the remaining are intercepted by Pegasus to the monitor for safety and handled in the monitor by different *backends* to improve performance. Pegasus controls the process management, scheduling, program loading, memory management, signal handling, and network I/O. Figure 1 shows the architecture of Pegasus.

Threat Model. Pegasus assumes that the monitor, the underlying Linux kernel, and the hardware are trusted, while the loaded programs may contain bugs or vulnerabilities. Hence, programs may contain bugs that lead to arbitrary memory access, control flow hijacking [55], and arbitrary system call invocations [1]. Pegasus ensures that such bugs in one `vProcess` will not affect other `vProcesses` [46].

To prevent fault propagation between programs, their memory and kernel context are isolated from each other. We apply memory protection with Intel® MPK and kernel context isolation [56], and enforce control flow integrity between the monitor and each `vProcess` with protected mode switch gates, which are the only way a `vProcess` can invoke the monitor. Thus, Pegasus does not assume fine-grained control flow integrity for applications [55]. Pegasus follows the techniques from prior works [26, 56, 71], such as binary inspection and secure switch gates, to securely apply Intel® MPK, and shares similar limitations, including a maximum of 16 memory isolation domains.

Each program can only access its own resources, i.e., memory and kernel context such as files and namespaces, while the monitor can access the resources of all the programs. Intel® MPK ensures that unauthorized memory accesses by programs will cause page faults, which can be caught by the Pegasus monitor by handling `SIGSEGV` signals. For fault isolation, Pegasus guarantees that if a program crashes due to signals such as `SIGSEGV`, the monitor and other programs will not be affected. While Pegasus considers fault isolation, our current prototype is mostly focused on functional isolation, not performance isolation. Finally, we assume Pegasus fuses programs from the same tenant, as its in-process isolation design is not resistant to side-channel attacks, and we consider side-channel attack protection out of scope. This assumption generally holds true for popular paradigms of symbiotic processes, such as the service mesh. Furthermore, existing techniques for side-channel protection can be applied or generalized to the Pegasus context [3].

Programs Assumptions. Pegasus makes the following assumptions about the loaded programs:

- the program is compiled and linked into a position-independent executable (PIE);
- the program does not need to allocate pages at hard-coded/fixed addresses;
- the program does not need to duplicate its address space (e.g., by forking).

These assumptions typically hold for networked programs like web servers, databases, or high-level language runtimes like Python. For example, PIE is often used as a security enhancement feature since it is a prerequisite to harden programs with ASLR. Some Linux distributions (e.g., Debian [14]) have enabled PIE for system packages by default. Furthermore, the few programs that use fork in this context, can typically be configured to not use it (§5.3,§6.1).

3.2 Process Management and Scheduling

To achieve faster communication, which consists of synchronization in addition to data copy, efficient scheduling is critical. To support fast context switches between programs in the same address space, Pegasus implements core process management and scheduling mechanisms in user space. By performing context switches in user space updating only a few registers, a Pegasus context switch is faster compared to a kernel context switch, which may involve saving and restoring all registers and switching the page table.

`VThread` and `VProcess` Creation. To facilitate the creation of the `vProcess` and its main `vThread`, Pegasus implements its own ELF loader in user space. We discuss the details in §3.3. Symbiotic processes can create additional `vProcesses` and `vThreads` by using the `clone` system call.

User-Space Scheduler. The `vThreads` spawned by the `vProcess`'s are scheduled using a user-space scheduler running in the Pegasus domain. Pegasus keeps one worker thread for each logical CPU core and pins it to the core. Any runnable `vThread` can be scheduled on a worker thread. Pegasus dynamically migrates `vThread` between worker threads to balance the load [19].

Pegasus maintains a run queue for each worker thread, which consists of active tasks on that worker thread. Pegasus also provides wait queues that store the tasks that are waiting for I/O, futex, or any other events to occur. A task will be detached from the run queue and put into a wait queue when it turns to the sleeping state. Then, when the task becomes runnable again, it is added back to the run queue and scheduled to run.

Privilege Modes. To provide fault isolation among the monitor and different applications, Pegasus supports two privilege modes, a privileged *monitor mode* where the monitor executes, and an unprivileged *application mode* where applications execute. During execution, each worker thread can be in either mode. The transitions in and out of monitor mode are guarded by *mode switch gates*. Applications can only invoke the monitor through these protected gates. All communication between applications is handled through the monitor, and direct invocations between applications are not allowed. The details on mode switch gates are discussed in §4.

Cooperative and Preemptive Scheduling. Pegasus is optimized to handle symbiotic processes that synchronously communicate with each other. However, it does not require the applications to *always* communicate synchronously with each other and proactively yield to the monitor (for example, by making a system call). Therefore, Pegasus supports both cooperative and preemptive scheduling. To achieve fairness and good performance, our implementation uses a Completely Fair Scheduler [54] algorithm, the details of which are discussed in §5.1.

To implement cooperative scheduling, the privileged monitor intercepts all system calls such as `futex`, `read`, and `clone`, which may wait for events or wake other tasks up. Every time these scheduling points are executed, the context of the currently executing task is saved and the scheduler is invoked. As this context includes only the task stack and a few registers, this process is fast and efficient. To implement preemptive scheduling, Pegasus sets a timer for each worker thread that periodically generates `SIGALARM` signals to force the tasks to switch to the monitor mode and yield to the scheduler. These signals also trigger load balancing between the worker threads. For cross-CPU preemption, the `SIGURG` signals are delivered through `tgkill` to interrupt the worker threads in the application mode.

3.3 Binary Loader and Memory Management

Loading multiple programs in the same address space allows Pegasus to schedule these programs from user space, thus eliminating expensive transitions to kernel space. However, in order to load the programs in the same address space, Pegasus must also handle memory management operations associated with launching a new `vProcess` or `vThread`. Similarly, to provide reliable isolation between different programs, Pegasus must apply the correct memory protections to the memory regions associated with different programs.

ELF Loader. Pegasus provides a user-space ELF file loader to load different programs to the same address space. When Pegasus starts a program, this ELF loader first creates a `vProcess` and its main `vThread`, then loads its binary. Any dynamic linker specified in its ELF header is also loaded to the memory of that program domain. Then, Pegasus sets up the stack for the program and retrieves the entry point from the ELF header, which is used to launch the program later. The ELF loader depends on the memory management subsystem to allocate memory and set the correct memory protections for each allocated page for the ELF binary's code, data, and stack regions. The ELF loader then creates a task that starts the execution in the application mode from the entry point, and wakes up the task to let it be scheduled to run by the user-space scheduler (§3.2). The ELF loader requires the binaries to be position-independent as the address range of the program domain is not fixed.

MPK-Based Isolation. Pegasus uses Intel® MPK [11] to isolate efficiently the monitor and different program domains, running in the same address space. The privileged monitor intercepts all system calls related to memory management, such as `mmap`, `brk`, and `munmap`, and handles them in the monitor. During initialization, for each program domain, the monitor reserves a large contiguous memory area using `mmap` with the `PROT_NONE` protection flag. When the program requests a memory region, the monitor computes the required number of memory pages and sets their MPK domain as well as the requested permissions (R/W/X) for the pages. Moreover, only the monitor domain is permitted to perform certain system calls that are essential for fault isolation. The details of the system call interception are in §3.4.

To maintain the correct memory permission semantics, and maintain compatibility with the Linux kernel's memory management system, the monitor must maintain the permissions (R/W/X) and attributes (anonymous/private/etc.) of all the pages allocated by the program domains. To achieve that, Pegasus records the information of all the virtual memory areas and synchronizes it with the host kernel via system calls. As a result, the memory view is consistent between the monitor and the kernel. The monitor uses this information for multiple operations, including checking the permission of a memory region or finding empty gaps between VMAs to allocate unused regions.

3.4 Monitoring Interactions Between VProcesses and the Kernel

3.4.1 System Call Interception

Pegasus applies per-domain Seccomp filters to intercept the scheduling, memory management, and networking system calls. The Seccomp filter of each program domain allows only a small number of system calls. If the program invokes a system call that is not on the allowlist, it raises a `SIGSYS` signal. The monitor domain registers a handler for the `SIGSYS` signal, and can thus handle the system call and provide the requested functionality to the program domain. Raising a signal every time a system call is executed is expensive. Therefore, Pegasus optimizes signal interception by using `LD_PRELOAD` hooks and binary rewriting whenever possible. The details of this optimization are in §4.

3.4.2 Signal Handling and Fault Isolation

Pegasus distinguishes between real signals and virtual signals, depending on whether they originate from within Pegasus or from the underlying operating system.

Virtual Signals. Virtual signals are either raised by communicating `vProcesses` and `vThreads`, or by the Pegasus monitor for timers and other operations that can raise signals according to the POSIX standard. Virtual signals are delivered to the tasks. Each `vThread` and each `vProcess` has its own signal queue. The signals to be delivered are appended into these queues, and every time the task returns to the application mode, it will poll the queues to find any pending signals, and invoke the corresponding signal handlers in the application mode.

Real Signals. Real signals are raised by the Linux kernel, including the ones caused by hardware exceptions such as page faults, the signals from the kernel, such as `SIGSYS` signals generated by the Seccomp filter, and the ones from other threads through the `kill` or `tgkill` system calls. Pegasus achieves fault isolation by handling all real signals in the privileged monitor. This ensures that the signals and exceptions by one program will not crash the whole process.

The monitor handles the following real signals transparently without informing the programs: the `SIGSYS` signals which are used for intercepting system calls, the `SIGALRM` and `SIGURG` signals which are used for preemptive scheduling, and the `SIGTRAP` signals which are used for breakpoint handling. The monitor is also capable of passing them to the programs. For example, when the monitor fails to transparently handle the `SIGSEGV` signals, it will either enter the application mode to invoke the signal handler registered by the program or terminate the program if no handler is registered. The monitor and other programs in the same process will not be affected.

3.4.3 Kernel Managed Functions

Pegasus also delegates many OS functions that are not related to communication optimization, such as disk I/O, to the underlying Linux kernel. To delegate a function to the Linux kernel, Pegasus simply passes them through to the host kernel. For example, each program has its own file descriptor table and mount namespace, which is achieved by μSwitch [56]. Thus, most of the kernel functions that use a file interface, such as disk I/O, timerfd, and eventfd, are automatically virtualized and isolated by the kernel. We directly delegate these functions to the kernel as the kernel-managed components. Pegasus still needs to intercept the system calls that are both related to sockets and other files, such as `read` or `write`, and multiplex them according to the file descriptor. If the file descriptor is a socket, the system call will be handled by the monitor according to the selected network backend. Otherwise, it will be passed through the kernel.

Delegating functions to the underlying Linux kernel is slower than handling them in user space. Therefore, in our current prototype, we delegate only the slower components to the Linux kernel. For example, we fall back to the Linux kernel when handling disk I/O. This has a negligible performance impact, as none of our use cases are disk I/O intensive.

Similarly, functionalities that are used less frequently and are thus not critical to performance, such as `eventfd` and `timerfd`, are also delegated to Linux.

3.5 Io_uring-Based Event Polling

Pegasus uses io_uring to handle blocking system calls, including the ones related to files (such as sockets or pipes), timeout, and I/O multiplexing, such as `read`, `nanosleep`, and `epoll_wait`. For such calls, Pegasus polls events such as a file getting readable or a timeout happening, using io_uring, and wakes up the task waiting for the event to the worker threads accordingly. Io_uring is also used as a default network backend if kernel-bypass networking is not available.

Io_uring uses two ring buffers, a submission queue and a completion queue, for submitting the operations to the kernel and retrieving its results. This allows Pegasus to check the completion queue every time the scheduler is invoked with very low overhead. When the worker thread becomes idle, that is, there is no runnable task, it either goes to sleep with `io_uring_enter` system call and waits for ready I/O events in the *sleeping mode*, or uses busy-waiting to repeatedly check for ready events or any runnable tasks in the *polling mode*. The polling mode avoids the overhead caused by worker threads' sleeping and waking up but consumes more CPU. Therefore, by default, the sleeping mode is enabled for CPU efficiency but the polling mode can optionally be used if performance is critical, for example, in our evaluation for remote communication optimization.

3.6 TCP Fast Path for Local Communication

To improve the performance of local communication, Pegasus provides a *local TCP fast path* for communications between symbiotic processes. Since symbiotic processes are fused into a single OS process running inside Pegasus, any TCP connection uses a fast path via Pegasus's shared ring buffers instead of invoking system calls and context switching into the OS. When a process tries to connect to a local TCP IP, Pegasus transparently switches from the OS-provided network stack to Pegasus's fast path. As a result, such TCP connections convert sending a message into copying to a shared memory buffer. To further improve the performance, Pegasus controls the task scheduling in user space. In particular, it controls the sleeping and blocking of tasks until the arrival of the data and the waking up of the sleeping tasks. This allows Pegasus to immediately switch the executing process when a network message is sent to a symbiotic process running in the same Pegasus instance. Thus, the execution follows the critical path of a message. Without the scheduler implemented in user space, Pegasus would need to rely on slow kernel synchronization techniques (e.g., `futex`).

3.7 Kernel Bypass for Remote Communication

To improve the performance of remote communication, Pegasus supports running unmodified Linux binaries with kernel-bypass network mechanisms. Traditionally, when using kernel-bypass systems such as Shenango, Demikernel, and F-Stack, programs must be rewritten to use custom APIs for threading and socket operations, to achieve microsecond-scale scheduling and maximize performance. Instead, Pegasus intercepts the regular socket system calls to transparently forward them to the kernel-bypass backend.

Pegasus, furthermore, virtualizes the kernel-bypass network interface across all symbiotic processes, allowing all of them to use the interface. To avoid coherence and synchronization issues, Pegasus relies on fast synchronization between processes when multiple processes send and receive messages at the same time.

4 Secure User-Space Process Virtualization

As discussed in §3.4, Pegasus uses Seccomp filters to intercept disallowed system calls invoked by the program domains. These system calls must then be delegated to the monitor, which *quickly* and *securely* emulates them on behalf of the program domain. Pegasus bases the required memory and kernel resource isolation using implicit context switching to avoid expensive transitions to the kernel [56]. This consists of splitting a process into multiple *domains*, each of which contains isolated resources, such as an Intel® MPK memory domain and a set of kernel resources (e.g., files, namespaces, and Seccomp filters). Pegasus provides a shared-descriptor for each thread that specifies the currently active domain for that thread, in the shared memory between kernel and user space. The thread can then select the kernel resources of a specific domain to use for system calls by writing to the shared-descriptor. Then, at the entry point of the next system call the thread invokes, the kernel will update the kernel resources according to the domain specified by the shared-descriptor if it has changed since last invocation. Pegasus uses this mechanism to access kernel resources of different `vProcesses` from the same worker thread. Unlike µSWITCH [56], which is designed for library isolation, Pegasus supports complete programs and a privileged monitor domain.

Memory Virtualization. Pegasus reserves Intel® MPK memory domain 0 for its monitor and creates one domain for each `vProcess`. Also, Pegasus reserves domain 1 to store the data that is writable by the monitor but read-only by `vProcesses`. When switching between the monitor mode and the application mode at mode switch gates, a worker thread sets the PKRU register, which controls the memory read and write permissions, according to the current domain. For the monitor domain, PKRU is set to 0 which grants access to all the memory domains. For other domains, PKRU is set to only granting access to the memory of the domain itself and read-only access to domain 1.

Fast System Call Interception. Disabled system calls raise a `SIGSYS` signal in the program domain, since the Seccomp filter disallows it. To avoid raising a signal for every system call invocation, Pegasus redirects and rewrites system call invocations using two methods. First, for the system calls that are performed via Libc wrappers (for example, using the Libc function `read` to invoke the system call `SYS_read`), Pegasus uses `LD_PRELOAD` to hook and redirect these function calls directly to the monitor. Second, for the system calls in the executable binary using a `syscall` assembly instruction, Pegasus rewrites the binary on the first execution of the `syscall` instruction. The first time the `syscall` instruction is invoked, it triggers the `SIGSYS` signal and invokes the signal handler registered by the monitor. When triggered, the signal handler can retrieve the address of the `syscall` instruction that triggered the `SIGSYS` signal by inspecting the signal frame. This allows Pegasus to correctly identify the instruction boundary of the `syscall` instruction, which is otherwise difficult to determine in x86-64 binaries through static analysis.

File Descriptors. Pegasus maintains a user file descriptor table for each `vProcess`. The user file descriptor table maps file descriptors to the corresponding Pegasus file objects, and is synchronized with the kernel file descriptor table in the domain of the `vProcess`. Pegasus updates both file descriptor tables when handling the system calls that open or close files. For a file object that does not have a corresponding kernel file descriptor such as a local fast path socket file, Pegasus allocates a placeholder kernel file descriptor by duplicating an opened null device file.

System Call Delegation. The Pegasus monitor performs all security-critical system calls *on behalf of* the program domain. For example, if a program requests file-backed memory, by invoking the `mmap` system call with a file descriptor, the monitor must allocate memory and map it to the application's requested file descriptor. µSwitch does not support calling system calls on behalf of another domain. We extend µSwitch to allow the monitor to perform system calls on behalf of `vProcess`'s. Pegasus modifies µSwitch, adding a *seccomp-descriptor* shared variable to the µSwitch kernel structure, to allow executing a system call in the program domain's context but with the monitor domain's Seccomp context that does not block any system call. Using this shared variable, the monitor can temporarily relax the program domain's Seccomp profile, switch to the program domain, and invoke the system call with the correct kernel context. To safely perform these steps, the shared variable is protected by Intel® MPK to ensure that only the monitor can access it.

Mode Switch Gates. The entry/exit points into and out of the monitor are guarded by mode switch gates. These gates control the switching between the monitor mode and application mode (§3.2) and have a similar design to µSwitch's *privcall* and *sandboxcall*. The mode switch gates update the PKRU register, the program and signal stack, and the shared-descriptor, to switch the memory and kernel context domain. The mode switch gates use the `WRPKRU` instructions to update the PKRU register. To prevent a `vThread` from directly jumping to a `WRPKRU` instruction to tamper with the PKRU register and escaping isolation, the `WRPKRU` instruction is always followed by a piece of assembly code that ensures the current PKRU register is expected or terminates the `vThread` otherwise.

In the gates switching from the monitor mode to the application mode, we compare PKRU to the expected value stored at per-worker-thread `%GS:0` in memory domain 1, which is read-only in the application mode. Also, the state of the monitor, including the instruction pointer, the stack pointer, and other registers, is saved to the protected memory in memory domain 0 before jumping to the application mode. The saved state is also per worker thread addressed relative to the `GS` segment. To prevent a `vThread` from tampering with the base address of the `GS` segment by using a `WRGSBASE` instruction, we apply binary inspection to avoid the occurrences of `WRGSBASE` in the executable memory, which is similar to µSwitch's approach to eliminate `WRPKRU` and `WRFSBASE`. Also, in our implementation of the `arch_prctl` system call, applications are not allowed to use `ARCH_SET_GS` to set the base address of the `GS` segment.

In the gates that switch from the application mode to the monitor mode, we simply check that PKRU is 0 by a `TEST` instruction followed by the code that loads the saved state of the monitor from the protected memory and jumps to the saved instruction pointer. Thus, we ensure that if the worker thread reaches the `WRPKRU` instruction, it must execute the tamper-proof code to enter the monitor with the correct state and resume the execution in the monitor mode.

In this way, Pegasus's mode switch gates ensure that the unprivileged programs cannot escape isolation.

5 Implementation

Pegasus is implemented with 25,946 lines of C++ and x86 assembly code, which covers the core components discussed in §3, including the local TCP fast path (1,826 LoC) and the kernel-bypass networking stack (1,985 LoC). In addition, Pegasus uses several libraries: Intel® XED [68] for breakpoint handling, libseccomp [44] for setting Seccomp filters, mapletree from the Linux kernel for memory management, and DPDK [69] for kernel-bypass networking. In this section, we discuss the Pegasus implementation.

5.1 Process Management and Scheduling

`vProcess` and `vThread` Creation. A program running in Pegasus can use the `clone` system call to create a `vProcess` or `vThread`. Pegasus supports the commonly used flags of the `clone` system call. In particular, we support the flags used by the pthread library. This lets Pegasus transparently support binaries that use the pthread library. Pegasus also supports the flags used for `vfork`. A program can also use `vfork`, followed by an `exec`, to dynamically load new binaries into the same address space without program changes.

Scheduler Internals. Pegasus uses a simplified version of Linux Completely Fair Scheduler (CFS). To avoid CPU starvation or overloading, Pegasus performs periodic load balancing between the worker threads, at an interval of 10 millisecond. Load balancing is also automatically triggered when a worker becomes idle. Preemptive scheduling is implemented via a timer that raises the `SIGALRM` signal at a frequency of 100 Hz. The `SIGALRM` signal handler forces the current task to yield to the scheduler. Similarly, `SIGURG` is used for cross-CPU preemption, which may occur when a task is scheduled to run on another core and preempts the core's currently active task. To avoid the high overhead of signals, Pegasus additionally supports a high-performance polling mode, allowing the users to optimize for CPU efficiency or high performance.

5.2 Signal Handling

Pegasus carefully handles signals (§3.4.2) to prevent the programs from escaping isolation. When a real signal occurs, the kernel writes the signal frame to the signal stack and returns to user space to a signal handler registered by Pegasus, which will check the current mode of the worker thread. If it is in monitor mode, the monitor handles the signal and returns from the handler to resume execution. If it is in the application mode, the execution will be forced to enter the monitor mode to let the monitor handle the signal. Finally, Pegasus will copy the signal frame to protected memory, validate it, and invoke `rt_sigreturn` to resume from it.

When a virtual signal (§3.4.2) occurs, Pegasus will write a virtual signal frame that is compatible with the kernel and enter the application mode to invoke the signal handler registered by the program. Then, Pegasus uses `rt_sigreturn` to resume from the virtual signal frame after validation, which is similar to real signals.

5.3 Platform Integration

Docker and Kubernetes. To integrate with container management platforms such as Docker, we implement a container runtime CLI program, `runpc`, following the Open Container Initiative (OCI) runtime specification. `runpc` accepts a container bundle directory including the configuration file and the root file system, and dynamically launches the container in a Pegasus instance. `runpc` is compatible with the default Docker runtime, `runc`. As a result, we use Docker for managing the images and containers when deploying Pegasus. In Kubernetes, all containers in the same pod are always colocated and typically host inter-dependent programs. Therefore, when being used with Kubernetes, `runpc` will automatically create a new instance for each Kubernetes pod, and run all containers of this pod in the same Pegasus instance.

Kernel-Bypass Networking. We integrate Pegasus with F-Stack [67], a robust and production-ready DPDK-based [69] user-space network stack library, into Pegasus with less than 2,000 lines of (application-agnostic) code. Compared with prior works [6, 21, 53, 78], Pegasus supports kernel-bypass networking with unmodified Linux binaries, which simplifies deployment significantly.

F-Stack uses a main loop that interacts with the DPDK poll mode driver and handles sending and receiving packets. To integrate with F-Stack, Pegasus creates a dedicated thread that runs the F-Stack main loop. This thread polls all the sockets for the socket events and wakes up the tasks waiting for them. As Pegasus implements the scheduler in user space, it wakes up efficiently the tasks, which is important to achieve low latency and high throughput for network-intensive applications.

Currently, some features in Linux, such as netlink, netfilter, and virtual network interfaces, are not supported by F-Stack. Thus, as Pegasus depends on F-Stack for kernel-bypass networking, some network management tools, such as iproute2 and iptables, cannot be used with Pegasus when kernel-bypass networking is enabled due to F-Stack's lack of the required features. Support for such tools can be achieved by switching to other kernel-bypass network stacks that support more Linux features. Pegasus's contributions are orthogonal to the kernel bypass network stack used.

Application Support. As discussed in §3.1, Pegasus makes some assumptions about applications. Thus, applications that need certain features, including mapping pages at static addresses or forking, are not supported by our implementation. While our evaluation experiments show that Pegasus supports many real-world applications, some applications, such as Apache or Bash that heavily depend on `fork()`, are not supported for this reason. Support for `fork()` may be addressed by extending Pegasus to multiple coordinated processes, as a fallback, to support clones of address spaces. This process can be significantly accelerated using fast fork implementation, such as on-demand fork [80], which applies copy-on-write to page tables and application pages. We leave these improvements for future work.

6 Evaluation

In this section, we evaluate Pegasus on its performance benefits to local and remote communication as well as its ease of deployment, using microbenchmarks and real-world applications without any code modification.

6.1 Setup

Testbed. We ran all the experiments on two r6525 servers from Cloudlab [16], each with two 2.8 GHz AMD EPYC 7543 CPUs, 256 GiB RAM, and a Mellanox ConnextX-6 100 Gbps NIC, running Ubuntu 22.04.

Applications and Benchmarks. We use the following versions of applications: Redis 6.2.6, Nginx 1.16.1, Memcached 1.6.22, Caddy 1.2.6, Istio 1.20.0, and Node.js 18.13.0. The Demikernel Redis port is based on Redis version 4.0.9. We do not change the source code of any application. For Golang programs, we build them with the option "`-buildmode=pie`" to generate PIEs (position-independent executables). We use

Table 1. Latency of synchronization primitives and protocol operations between two applications.

Operation	**Baseline** (μs)	**Pegasus** (μs)	**Reduction**
Futex Wake Up	1.37	0.49	64%
Condition Variable Wake Up	1.51	0.56	63%
TCP Echo	7.8	1.2	85%
Redis Set	11.0	4.8	56%
Memcached Set	10.3	3.7	64%
HTTP Echo	45.8	36.7	20%

the default configurations for all applications except for the following changes: for Nginx, daemonizing and worker process spawning are disabled because Pegasus does not support fork, as discussed in §5.3; for Redis, background persistence is disabled for the same reason. For each data point, we execute 10 runs and report the median.

6.2 Local Communication

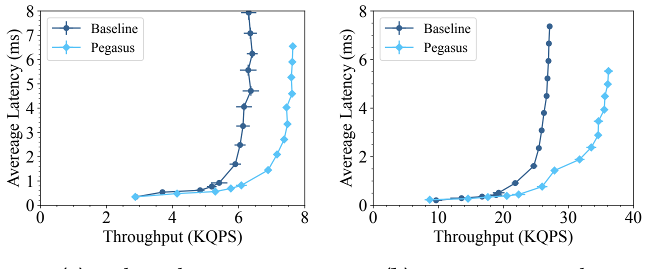
We first evaluate the performance provided by Pegasus's local communication optimization. For the experiments in this section that require remote communication, we do not enable kernel-bypass networking, i.e., remote communication is handled by the Linux kernel.

Microbenchmarks. We designed several microbenchmark applications to evaluate the communication overhead reduction between two co-located applications running with Pegasus. For each application, we measure the latency of different synchronization primitives or protocol operations, which are selected for their commonness in co-located applications. The results are shown in Table 1.

Linux Synchronization. We first evaluate how Pegasus improves the latency of Linux synchronization primitives to show that user-space scheduling is necessary. We create two threads that are pinned on the same core. Thread 1 sleeps with the `FUTEX_WAIT` operation of the `futex` system call while thread 2 uses `FUTEX_WAKE` to wake up thread 1. We repeat that 1,000,000 times and measure the average latency between thread 1 invoking `FUTEX_WAKE` and thread 2 waking from `FUTEX_WAIT`. Pegasus achieves an average latency of 0.49 μs, which is 64% lower than the baseline. Similarly, conducting the same experiment with a pthread condition variable shows that Pegasus achieves 63% lower latency compared to the baseline.

Protocol Operations. We then evaluate how Pegasus improves the latency for different network protocols. We use four applications of different protocols, including a TCP echo server, a Redis server, a Memcached sever, and an HTTP echo server. For each protocol, we use a client to send the requests to the server 1,000,000 times and report the average latency. The results show that Pegasus achieves latency improvements ranging from 20% to 85%.

Web Application. We evaluate the performance benefits of fusing symbiotic processes in web application deployments.



(a) Web application. **(b)** Istio service mesh.

Figure 2. Web application and HTTP API with the Istio service mesh latency and throughput.

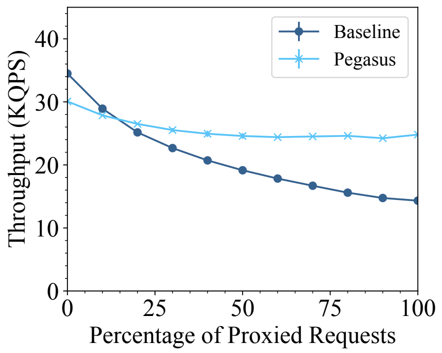


Figure 3. Throughput of the HTTP server with different percentages of proxied requests.

Our setup consists of a Nodejs backend server, a Redis database and a Nginx reverse proxy with TLS encryption enabled. This web application implements a simple pastebin, which saves the user input to the Redis database, and provides a URL to retrieve it. We run all three programs in the same Pegasus instance and use the bombardier HTTP benchmark tool to measure the throughput and average latency from another machine with a different number of concurrent clients from 1 to 50 to vary the server load. Figure 2a shows how the latency and throughput change as we increase server load. Pegasus increases the maximum throughput by 19% compared with the Linux baseline, reflecting the increased efficiency using kernel bypass on the local data path. Furthermore, Pegasus significantly decreases the average latency when the system is loaded (e.g., throughput higher than 4 KQPS), and achieves comparable latency to the baseline under low load scenarios.

Service Mesh Proxy. Pegasus also reduces the overhead caused by proxies in a service mesh. In this experiment, we deploy an OpenResty echo HTTP API, with an Istio service mesh to evaluate how Pegasus can make the service mesh approach more efficient and faster. The OpenResty container and the Envoy sidecar container run in the same Kubernetes pod, with Envoy forwarding requests to OpenResty. For Pegasus, the two containers in the pod run in the same Pegasus instance. We use another machine to measure the throughput and latency with different concurrent connections to

Table 2. Application porting effort for different systems. Inserted lines(+)/deleted lines(-)/changed files.

	Demikernel	F-Stack	Junction	Pegasus
Redis	2478+/549-/19	715+/20-/12	0+/0-/0	0+/0-/0
Nginx	N/A	1943+/124-/55	0+/0-/0	0+/0-/0
Memcached	N/A	N/A	0+/0-/0	0+/0-/0

vary server load. We compare it to the baseline without Pegasus. The results are shown in Figure 2b. Pegasus achieves 33% higher throughput for the HTTP API and improves latency when the server is loaded while keeping the latency comparable to the baseline at a lower load.

Reverse Proxy. Pegasus intercepts and monitors system calls, which introduces overhead. We evaluate this overhead and in what cases the data-path optimization provided by Pegasus compensates that overhead. We run an experiment with two HTTP servers, Caddy and Nginx, each serving a 4 KiB static file. At the same time, the Nginx server also serves as a reverse proxy for the Caddy server at a different URL path, so the client can choose whether a request is handled directly or via the reverse proxy by accessing different URL paths of the Nginx server. We vary the percentage of the requests that are handled by the reverse proxy and measure the throughput. The results are shown in Figure 3. Pegasus increases the throughput if more than 20% of the requests go through the proxy, peaking at a 74% throughput increase (from 14.3 KQPS to 24.8 KQPS) when all requests are handled through the proxy. As expected, if fewer requests go through the proxy, since our overheads are fixed, the throughput decreases. In this experiment, the decrease can reach 13% (from 34.5 KQPS to 30.1 KQPS). This shows that without remote communication optimization, Pegasus should primarily be used when the local communication is high.

6.3 Remote Communication

In this section, we evaluate Pegasus against three network kernel-bypass systems and the Linux baseline to assess how Pegasus can transparently accelerate the remote communication for unmodified applications with comparable or better performance than prior approaches.

To reduce latency, similar to other kernel-bypass networking systems, we enable polling for Pegasus in which the worker thread continuously polls for the runnable tasks instead of sleeping. For Memcached, which is multi-threaded, we use two Pegasus worker threads. For other applications that are single-threaded, we use one worker thread. For all the experiments, we use one thread for polling the NIC and disable the local communication optimization.

Application Support. Demikernel and F-Stack support the Redis server through system-specific modifications. F-Stack also supports the Nginx server through custom modifications. As shown in Table 2, both Demikernel and F-Stack require extensive application changes, ranging from 735 to 3027 lines of source code across 12 to 55 files. We do not port other

Table 3. TCP echo server latency in microseconds.

Linux	Demikernel	F-Stack	Junction	Pegasus
27.78	11.08	11.97	10.55	13.88

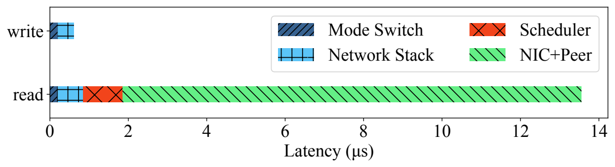


Figure 4. Latency composition of TCP echo server with Pegasus

applications to these two frameworks due to the significant effort required. However, we further demonstrate Pegasus's compatibility by also benchmarking Memcached, which is not supported by either Demikernel or F-Stack. Junction also supports all three applications without code modification in this experiment. However, as discussed in §6.4, Junction does not work with Nginx if used as a TLS proxy because Junction lacks support for socket functions required by the application.

TCP Round-Trip Latency. We first use a TCP round-trip latency experiment to analyze the performance improvement and overhead of Pegasus for remote communication. We send 1-byte data to a TCP echo server from a remote client 1,000,000 times and report the average latency in Table 3. Pegasus reduces the request latency compared with Linux by 13.90 μs. However, as expected, Pegasus introduces some additional latency, 1.91 μs, 2.80 μs, and 3.33 μs, compared with F-Stack, Demikernel, and Junction, respectively. To analyze the overhead of Pegasus, we measure the latency introduced by each component. Figure 4 shows the latency composition for `read` and `write` respectively. It shows that mode switches and scheduler overhead introduced by Pegasus account for the 1.91 μs difference from F-Stack. In practice, this benchmark is particularly demanding for Pegasus because the messages have the smallest possible size, which shows the worst-case latency overhead for Pegasus. With larger message sizes, the Pegasus overhead can be considerably amortized as the following experiments on commonly deployed applications show.

Real-World Applications. We evaluate Pegasus on three real-world applications, Nginx, Redis, and Memcached, against three kernel-bypass networking systems as well as the Linux baseline.

For Redis and Memcached, we use the memtier_benchmark tool [59] to measure the request performance from a remote machine. We vary the number of concurrent connections, ranging from 1 to 64, and measure the throughput, the average latency, and the 99% tail latency. Then, we plot these results to show how average latency and 99% tail latency relate to the different throughput achieved as we vary the

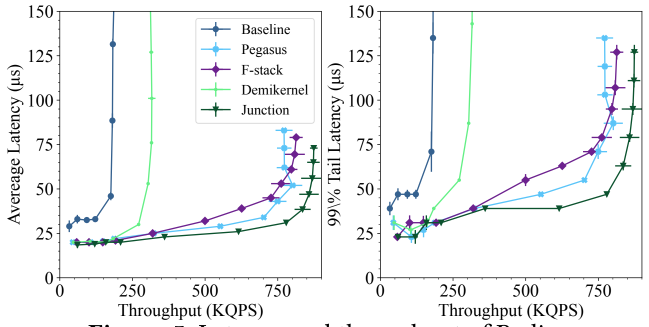


Figure 5. Latency and throughput of Redis.

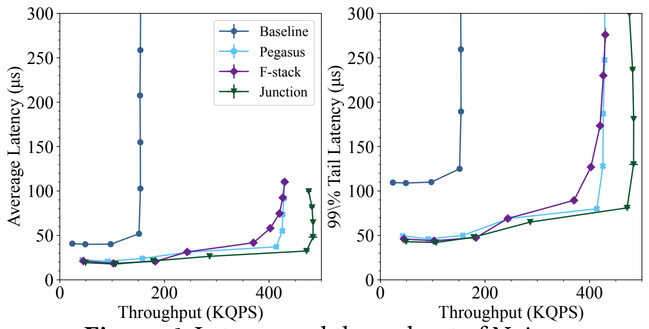


Figure 6. Latency and throughput of Nginx.

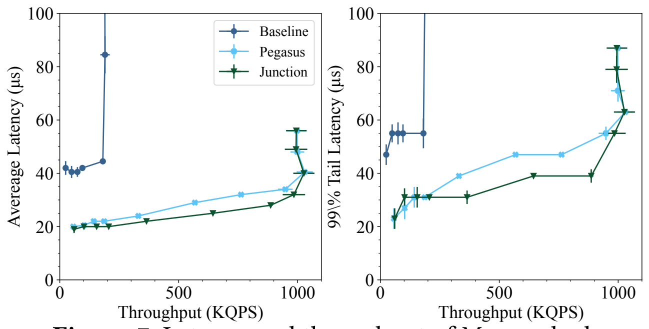


Figure 7. Latency and throughput of Memcached.

load (i.e., number of concurrent client connections). For Nginx, we use the bombardier tool with the same methodology, ranging from 1 to 48 connections.

Figure 5 shows the Redis performance results. Pegasus achieves the maximum throughput of 801 KQPS, which is 323% higher than the Linux baseline's 189 KQPS. For comparison, Demikernel, F-Stack, and Junction achieve the maximum throughput of 317 KQPS, 813 KQPS, and 874 KQPS, respectively. Pegasus decreases the throughput by 1.5% compared with F-Stack, which is a modest reduction given that it transparently supports unmodified applications. The throughput of Junction is 9.1% higher than Pegasus. Considering that Junction's throughput exceeds the one of F-Stack by 7.5%, the performance gap between Junction and Pegasus is attributable to Junction's stack improvement over F-Stack, which Pegasus leverages to implement the remote fast path and is orthogonal to Pegasus's design. Interestingly, Pegasus achieves 153% higher throughput than Demikernel, despite its uses of a custom API and significant porting effort requirements. This performance difference is primarily caused by the API of Demikernel which requires polling the clients linearly, limiting its scalability at high concurrency. Similarly, Figure 6 and Figure 7 show the performance of Nginx and Memcached, respectively. Without any code modification, Pegasus achieves 178% and 442% higher throughput than the Linux baseline respectively, and has similar performance to Junction and F-Stack.

6.4 Mixed Communication

In this section, we evaluate how Pegasus improves performance for applications that use both local and remote communication: a Caddy HTTP server that servers a 4 KiB static file with a Nginx TLS reverse proxy. This pattern of co-located reverse proxy is common in web applications, as the proxy can provide functions such as encryption [51] without changing the proxied application itself.

In addition to the Linux baseline, we run three configurations of Pegasus with different features enabled: (1) both local and remote optimizations (Pegasus), (2) only local optimization (Pegasus w/o Remote Opt.), and (3) only remote optimization (Pegasus w/o Local Opt.). These three configurations show how each fast path of Pegasus contributes to performance improvement and the benefits of unifying them. For all configurations, we use five Pegasus worker threads in the sleeping mode and one thread for polling the NIC. To run as a TLS proxy, Nginx requires support for the `getsockopt` system call and the `MSG_PEEK` flag in the `recvfrom` system call, both of which Junction does not support. Fixing the compatibility issues of Junction would require substantial refactoring in either Junction's network stack or Nginx's source code. Thus, we do not evaluate it.

We use the bombardier HTTP benchmark tool [10] to measure the performance of the proxied server from a remote machine. We vary the number of concurrent connections, ranging from 1 to 48, and measure the throughput, the average latency, and the 99% tail latency. We plot how the average latency and the 99% tail latency relate to the different throughputs achieved as we vary the load.

Figure 8 shows Pegasus achieves 39.0 KQPS throughput, which means it takes 25.6 μs to process one request on average. For comparison, the Linux baseline achieves 12.1 KQPS (82.6 μs). When only one fast path is enabled, Pegasus achieves 16.8 KQPS (59.5 μs) with only remote optimization and 20.4 KQPS (49.0 μs) with only local optimizations. Thus, the local and remote optimizations reduce the average request processing time by 33.5 μs and 23.1 μs, respectively. When combined, these optimizations result in a total time reduction of 57.0 μs and a throughput increase of 222%, showing that the two optimizations reduce the latency independently.

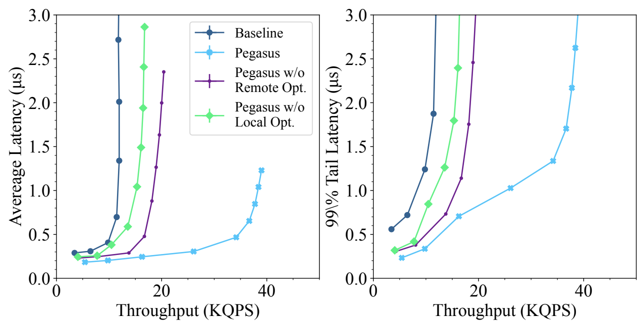


Figure 8. Latency and throughput of Caddy + Nginx.

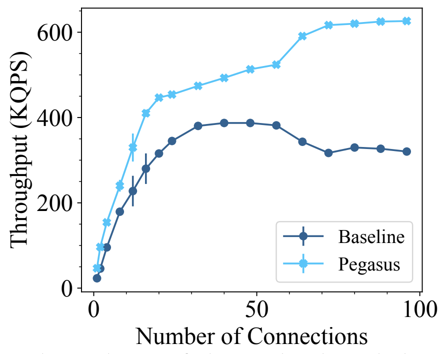


Figure 9. Throughput of the multi-threaded Web server under different concurrency.

Table 4. Latency of system calls with kernel involvement.

System Call	Baseline (μs)	Pegasus (μs)	Overhead
`write 1 B`	0.55	0.79	44%
`write 64 KiB`	5.88	6.22	6%
`mprotect`	0.09	0.25	178%
`fstat`	0.50	0.61	22%
`getuid`	0.058	0.080	38%

6.5 In-Process Virtualization

In this section, we evaluate the performance of Pegasus's in-process virtualization and assess the overhead introduced by Pegasus's `vProcess` and `vThread` abstractions.

System Call Overhead. In Pegasus, not all system calls are handled in user space. Some OS functions that are not related to communication, such as disk I/O, are still handled by the kernel. When delegating these OS functions to the kernel, Pegasus introduces overhead for in-process virtualization. System calls delegated to the kernel are handled in two different ways. Some system calls are intercepted by the Pegasus monitor and forwarded to the kernel followed by handling in user space. For example, when handling the `write` system call, the monitor needs to look up in the file descriptor table and handle it depending on the file type, for example, forwarding the write operation to the kernel for disk files. The additional latency for these system calls is incurred by mode switch gates, user-space handling, and kernel overhead from Seccomp filters and µSwitch. Other system calls are directly handled by the kernel with Seccomp filters set to allow them, resulting in only kernel overhead. We evaluate the overhead for both types of system calls.

We use four system calls of these two types to evaluate such overhead by measuring their average latency of 1,000,000 invocations. The results are shown in Table 4. Firstly, we use `write` to write to a file on disk with the buffer size of 1 B and 64 KiB respectively, and use `mprotect` to set a page to readwrite. The `write` and `mprotect` system calls require user-space handling in the monitor and thus belong to the first type. Pegasus introduces a latency of 0.24 μs for 1 B `write`, 0.34 μs for 64 KiB `write`, and 0.16 μs for `mprotect`, respectively. Then, we use `fstat` to get the information of a file, and use `getuid` to get the current user id. Both system calls are directly handled by the kernel and thus belong to the second type. Pegasus introduces a latency of 0.11 μs for `fstat` and 0.022 μs for `getuid`. This shows that currently, Pegasus introduces moderate overhead for the OS functions that are handled by the kernel. By further moving these OS functions to user space, e.g., with kernel-bypass storage, Pegasus can be extended to eliminate such overhead, which we leave as future work.

`vThread` Performance. To evaluate Pegasus on the performance of multi-threaded applications, we wrote a Web server that serves each connection with one thread using blocking I/O. We run the Web server on one machine and measure its throughput from another machine with varied numbers of connections. We use four Pegasus worker threads in the polling mode and one thread for polling the NIC, For comparison, for the Linux baseline, we set the core affinity of the server process to five cores. The results are shown in Figure 9. Pegasus achieves higher throughput than the Linux baseline. Furthermore, Pegasus scales better than the Linux baseline, whose throughput decreases as the number of connections increases after it reaches 50. This means Pegasus's in-process virtualization, including the user-space scheduler and virtualized `vThreads`, is scalable.

7 Related Work

Kernel-Bypass Networking. Existing kernel-bypass networking systems mainly focus on different aspects of remote communication optimization. IX [6] uses virtualization to protect direct hardware access. Demikernel [78] defines a flexible architecture and a new API for heterogeneous data paths. Shenango [53] provides high CPU-sharing efficiency for applications that use kernel-bypass networking. However, these systems do not provide a Linux ABI-compatible interface. While customized APIs offer potentially better performance, they also increase porting efforts.

Some kernel-bypass networking systems, such as TAS [37] and libVMA [48], provide user-space libraries that are compatible with the POSIX socket API. These systems usually depend on `LD_PRELOAD` to intercept library calls from applications for transparency. However, system calls cannot be reliably intercepted with `LD_PRELOAD`, e.g., when the application is statically linked or performs direct system calls.

Recently, NetKernel [52] proposed using a modified guest kernel in para-virtualized VMs to bypass the guest kernel network stack. This ensures transparency for applications but requires significant kernel modification and introduces higher overhead than user-space libraries.

Furthermore, none of these systems optimizes local communication, which is crucial for symbiotic processes. In contrast, Pegasus achieves transparent and unified kernel-bypass networking for both local and remote communication.

Junction [20] is a concurrent work that, similar to Pegasus, also aims at Linux ABI compatibility for practical kernel bypass and runs multiple programs in the same address space. However, Junction adopts a fate-sharing approach, lacking support for a protected monitor or isolated programs within an instance. Also, Junction does not support important OS functions, such as containers, and restricts file system operations to read-only files and in-memory virtual files. In contrast, Pegasus enforces isolation for the monitor and each application, and comprehensively supports OS functions and platform integration, including Docker and Kubernetes. Furthermore, Junction uses a dedicated core for its centralized scheduler, and forces all packets including the ones for local communication, to be processed by the NIC. In contrast, Pegasus uses a distributed scheduler that operates on every worker thread, and supports a local TCP fast path that operates inside a Pegasus instance in user space without the involvement of the kernel or the NIC, thus avoiding a centralized bottleneck that may impact scalability.

Control Plane Customization. Several techniques customize the OS control plane to improve I/O performance. eBPF [17] allows users to inject code into the kernel at runtime for various applications [23, 28, 38, 49, 81]. For example, XDP [28], which is based on eBPF, allows customized fast-path packet handling that bypasses the regular kernel network stack. However, eBPF is limited in flexibility and expressiveness. Kernel-bypass I/O [60, 69, 75] and user-space network stacks [33, 64, 67, 76] moves the networking control plane to user space for fast data paths without kernel overhead. Furthermore, a recent work, Wave [30], proposes to offload the OS control plane, including scheduling, memory management, and network stack, to SmartNICs for further optimization. All these approaches to customization typically require significant changes to applications. In contrast, Pegasus customizes the OS control plane by moving the related features such as scheduling and networking to user space while maintaining Linux ABI compatibility.

Single Address Space Isolation. Memory isolation within a single address space protects sensitive data from untrusted components such as external libraries. Beyond traditional SFI [18, 72, 77], hardware-assisted approaches [2, 5, 8, 13, 24, 26, 29, 39–42, 47, 50, 56, 57, 71] have been proposed to achieve better performance or isolation guarantees. Other works [45, 56, 74] isolate kernel resources within a single process. Pegasus builds on top of such work, including Intel® MPK and μSwitch [56], to offer in-process virtualization and isolation though the `vProcesses` and `vThreads` abstractions.

Fusing Symbiotic Applications. Hosting symbiotic programs in a single address space (similar to single address space OSs[7, 27]) opens up opportunities for performance optimization, such as the local IPC fast path provided by Pegasus. IPC between sidecar and applications incurs high overhead in service mesh [82]. Faasm [65] and Fastlane [42] chain functions in a single address space to reduce communication costs. Similarly, recent work at Google suggests fusing Go applications [22] for performance improvement. Photons [15] co-locates the same functions to achieve a low memory footprint. CAP-VMs [62] build shared memory interfaces between virtual machines to improve communication, and ORC [63] allows object reuse to improve memory density and startup times. Xia et al. [73] suggest architectural support to improve communication and synchronization between frequently communicating processes. Pegasus provides a framework to fuse general processes independent of the source language or runtime.

8 Conclusion

Modern architectures, such as the service mesh, require the kernel even for communication with services on the same host. Pegasus is a unified kernel bypass for local and remote communication. By fusing symbiotic applications into the same process, Pegasus eliminates expensive kernel transitions. Moreover, by controlling the OS functionalities critical for communication in user space, Pegasus can provide transparent kernel bypass for both fused symbiotic processes for local and remote communication using DPDK, for unmodified Linux binaries. Our evaluation demonstrates that Pegasus significantly optimizes local and remote communication workloads by improving the throughput by up to 33% and 442% respectively, and 222% for workloads that use both. Pegasus achieves these improvements without changing a single line of application code.

Acknowledgments

We would like to express our sincere gratitude to the anonymous reviewers of this and the previous submissions, and our shepherd Antoine Kaufmann, whose insightful feedback greatly improved this work. We are also grateful to Cloudlab for providing experimental machines. This work was funded in part by gifts from Intel and the NSF awards CNS-2145888 and CNS-2140305. Furthermore, it was funded in part by CNS-2127309 to the Computing Research Association for the CIFellows Project.

References

[1] Abubakar, M., Ahmad, A., Fonseca, P., and Xu, D. SHARD: Fine-grained kernel specialization with context-aware hardening. In *Proceedings of the USENIX Conference on Security Symposium (USENIX Security)* (2021).

[2] Ahmad, A., Kim, J., Seo, J., Shin, I., Fonseca, P., and Lee, B. CHANCEL: Efficient multi-client isolation under adversarial programs. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)* (2021).
[3] Ahmad, A., Schultz, A., Lee, B., and Fonseca, P. An extensible orchestration and protection framework for confidential cloud computing. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2023).
[4] AWS. What is a service mesh. https://aws.amazon.com/what-is/service-mesh/.
[5] Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2012).
[6] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).
[7] Chase, J. S., Levy, H. M., Feeley, M. J., and Lazowska, E. D. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.* (1994).
[8] Chen, Y., Reymondjohnson, S., Sun, Z., and Lu, L. Shreds: Fine-grained execution units with private memory. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2016).
[9] Cloudflare. Cloudflare workers. https://workers.cloudflare.com/, 2021.
[10] codesenberg. Fast cross-platform http benchmarking tool written in go, 2023. https://github.com/codesenberg/bombardier.
[11] Corbet, J. Memory protection keys, 2015. https://lwn.net/Articles/643797/.
[12] Datadog. The state of serverless. https://www.datadoghq.com/state-of-serverless/, 2023.
[13] Dautenhahn, N., Kasampalis, T., Dietz, W., Criswell, J., and Adve, V. Nested Kernel: An operating system architecture for intra-kernel privilege separation. *SIGARCH Comput. Archit. News* (2015).
[14] Debian. Hardening pie by default transition. https://wiki.debian.org/Hardening/PIEByDefaultTransition.
[15] Dukic, V., Bruno, R., Singla, A., and Alonso, G. Photons: Lambdas on a diet. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (2020).
[16] Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S., and Mishra, P. The design and operation of CloudLab. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (ATC)* (2019).
[17] eBPF.io authors. ebpf - introduction, tutorials & community resources, 2023. https://ebpf.io.
[18] Erlingsson, Ú., Abadi, M., Vrable, M., Budiu, M., and Necula, G. C. XFI: Software guards for system address spaces. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2006).
[19] Ford, B., and Lepreau, J. Evolving mach 3.0 to a migrating thread model. In *USENIX Winter* (1994).
[20] Fried, J., Chaudhry, G. I., Saurez, E., Choukse, E., Goiri, I., Elnikety, S., Fonseca, R., and Belay, A. Making kernel bypass practical for the cloud with junction. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2024), USENIX Association.
[21] Fried, J., Ruan, Z., Ousterhout, A., and Belay, A. Caladan: Mitigating interference at microsecond timescales. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2020).
[22] Ghemawat, S., Grandl, R., Petrovic, S., Whittaker, M., Patel, P., Posva, I., and Vahdat, A. Towards modern development of cloud applications. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)* (2023).
[23] Ghigoff, Y., Sopena, J., Lazri, K., Blin, A., and Muller, G. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2021).
[24] Ghosn, A., Kogias, M., Payer, M., Larus, J. R., and Bugnion, E. Enclosure: Language-based restriction of untrusted libraries. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2021).
[25] Gu, J., Wu, X., Li, W., Liu, N., Mi, Z., Xia, Y., and Chen, H. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (ATC)* (2020).
[26] Hedayati, M., Gravani, S., Johnson, E., Criswell, J., Scott, M. L., Shen, K., and Marty, M. Hodor: Intra-Process isolation for High-Throughput data plane libraries. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (ATC)* (2019).
[27] Heiser, G., Elphinstone, K., Vochteloo, J., Russell, S., and Liedtke, J. The mungi single-address-space operating system. *Softw. Pract. Exp.* (1998).
[28] Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., and Miller, D. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)* (2018).
[29] Hsu, T. C.-H., Hoffman, K., Eugster, P., and Payer, M. Enforcing least privilege memory views for multithreaded applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2016).
[30] Humphries, J. T., Natu, N., Kaffes, K., Novaković, S., Turner, P., Levy, H., Culler, D., and Kozyrakis, C. Wave: A split os architecture for application engines, 2024.
[31] IBM. Microservices in the enterprise, 2021: Real benefits, worth the challenges. https://www.cncf.io/blog/2021/04/19/microservices-in-the-enterprise-2021-real-benefits-worth-the-challenges/, 2021.
[32] Ivanenko, S., Stevanovic, J., Jovanovic, V., and Bruno, R. Hydra: Virtualized multi-language runtime for high-density serverless platforms. *arXiv preprint arXiv:2212.10131* (2022).
[33] Jeong, E., Wood, S., Jamshed, M., Jeong, H., Ihm, S., Han, D., and Park, K. mTCP: a highly scalable user-level TCP stack for multicore systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2014).
[34] JetBrains: Developer Tools for Professionals and Teams. Microservices - the state of developer ecosystem in 2022 infographic. https://www.jetbrains.com/lp/devecosystem-2022/microservices/, 2022.
[35] Jia, Z., and Witchel, E. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2021).
[36] Kanev, S., Darago, J. P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.-Y., and Brooks, D. Profiling a warehouse-scale computer. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)* (2015).
[37] Kaufmann, A., Stamler, T., Peter, S., Sharma, N. K., Krishnamurthy, A., and Anderson, T. TAS: Tcp acceleration as an os service. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (2019).
[38] kernel development community, T. Linux security module usage, 2023. https://www.kernel.org/doc/html/v6.1/admin-guide/LSM/index.html.
[39] Kim, T., and Zeldovich, N. Practical and effective sandboxing for

non-root users. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (ATC)* (2013).

[40] Kirth, P., Dickerson, M., Crane, S., Larsen, P., Dabrowski, A., Gens, D., Volckaert, S., and Franz, M. PKRU-safe: Automatically locking down the heap between safe and unsafe languages. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (2022).

[41] Koning, K., Chen, X., Bos, H., Giuffrida, C., and Athanasopoulos, E. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (2017).

[42] Kotni, S., Nayak, A., Ganapathy, V., and Basu, A. Faastlane: Accelerating Function-as-a-Service workflows. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (ATC)* (2021).

[43] Li, B., Cui, T., Wang, Z., Bai, W., and Zhang, L. Socksdirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)* (2019).

[44] Libsecommp Authors. Libsecommp. https://github.com/seccomp/libseccomp, 2023.

[45] Litton, J., Vahldiek-Oberwagner, A., Elnikety, E., Garg, D., Bhattacharjee, B., and Druschel, P. Light-Weight contexts: An OS abstraction for safety and performance. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2016).

[46] Liu, C., Gong, S., and Fonseca, P. KIT: Testing OS-level virtualization for functional interference bugs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2023).

[47] Liu, Y., Zhou, T., Chen, K., Chen, H., and Xia, Y. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2015).

[48] Mellanox. Linux user space library for network socket acceleration based on rdma compatible network adaptors. https://github.com/Mellanox/libvma.

[49] Meta. facebookincubator/katran: A high performance layer 4 load balancer. https://github.com/facebookincubator/katran, 2023.

[50] Narayan, S., Garfinkel, T., Taram, M., Rudek, J., Moghimi, D., Johnson, E., Fallin, C., Vahldiek-Oberwagner, A., LeMay, M., Sahita, R., Tullsen, D., and Stefan, D. Going beyond the limits of sfi: Flexible and secure hardware-assisted in-process isolation with HFI. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2023).

[51] Nginx. Securing http traffic to upstream servers. https://docs.nginx.com/nginx/admin-guide/security-controls/securing-http-traffic-upstream/, 2024.

[52] Niu, Z., Su, Q., Cheng, P., Xiong, Y., Han, D., Winstein, K., Xue, C. J., and Xu, H. NetKernel: Making network stack part of the virtualized infrastructure. *IEEE/ACM Trans. Netw.* (2021).

[53] Ousterhout, A., Fried, J., Behrens, J., Belay, A., and Balakrishnan, H. Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2019).

[54] Pabla, C. S. Completely fair scheduler. *Linux Journal 2009* (2009), 4.

[55] Palit, T., and Fonseca, P. Kaleidoscope: Precise invariant-guided pointer analysis. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2024).

[56] Peng, D., Liu, C., Palit, T., Fonseca, P., Vahldiek-Oberwagner, A., and Vij, M. uSWITCH: Fast kernel context isolation with implicit context switches. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2023).

[57] Proskurin, S., Momeu, M., Ghavamnia, S., Kemerlis, V. P., and Polychronakis, M. xMP: Selective memory protection for kernel and user space. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2020).

[58] Raghavan, D., Ravi, S., Yuan, G., Thaker, P., Srivastava, S., Murray, M., Penna, P. H., Ousterhout, A., Levis, P., Zaharia, M., et al. Cornflakes: Zero-copy serialization for microsecond-scale networking. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2023).

[59] Redis. memtier_benchmark: A high-throughput benchmarking tool for redis & memcached. https://github.com/RedisLabs/memtier_benchmark, 2023.

[60] Rizzo, L. netmap: A novel framework for fast packet I/O. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (ATC)* (2012).

[61] Saokar, H., Demetriou, S., Magerko, N., Kontorovich, M., Kirstein, J., Leibold, M., Skarlatos, D., Khandelwal, H., and Tang, C. ServiceRouter: Hyperscale and minimal cost service mesh at meta. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2023).

[62] Sartakov, V. A., Vilanova, L., Eyers, D., Shinagawa, T., and Pietzuch, P. CAP-VMs: Capability-Based isolation and sharing in the cloud. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2022).

[63] Sartakov, V. A., Vilanova, L., Geden, M., Eyers, D., Shinagawa, T., and Pietzuch, P. ORC: Increasing cloud memory density via object reuse with capabilities. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2023).

[64] ScyllaDB authors. Seastar is an advanced, open-source C++ framework for high-performance server applications on modern hardware. https://seastar.io, 2023.

[65] Shillaker, S., and Pietzuch, P. Faasm: Lightweight isolation for efficient stateful serverless computing. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (ATC)* (2020).

[66] Shillaker, S., Segarra, C., Mappoura, E., Fournial, M., Vilanova, L., and Pietzuch, P. Faabric: Fine-grained distribution of scientific workloads in the cloud. *arXiv preprint arXiv:2302.11358* (2023).

[67] Tencent Cloud. Fstack | High Performance Network Framework Based on DPDK. http://www.f-stack.org, 2023.

[68] The IntelXED Authors. Intel® x86 encoder decoder (intel® xed). https://github.com/intelxed/xed, 2023.

[69] The Linux Foundation. Data plane development kit, 2023. https://www.dpdk.org.

[70] The Open Container Initiative Authors. Open container initiative runtime specification. https://github.com/opencontainers/runtime-spec/blob/main/spec.md, 2023.

[71] Vahldiek-Oberwagner, A., Elnikety, E., Duarte, N. O., Sammler, M., Druschel, P., and Garg, D. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *Proceedings of the USENIX Conference on Security Symposium (USENIX Security)* (2019).

[72] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev.* (1993).

[73] Xia, Y., Du, D., Hua, Z., Zang, B., Chen, H., and Guan, H. Boosting inter-process communication with architectural support. *ACM Trans. Comput. Syst.* (2022).

[74] Yang, F., Im, B., Huang, W., Kaoudis, K., Vahldiek-Oberwagner, A., che Tsai, C., and Dautenhahn, N. Endokernel: A thread safe monitor for lightweight subprocess isolation. In *Proceedings of the USENIX Conference on Security Symposium (USENIX Security)* (2024).

[75] Yang, Z., Harris, J. R., Walker, B., Verkamp, D., Liu, C., Chang, C., Cao, G., Stern, J., Verma, V., and Paul, L. E. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* (2017).

[76] Yasukata, K., Honda, M., Santry, D., and Eggert, L. StackMap: Low-Latency networking with the OS stack and dedicated NICs. In *Proceedings of the USENIX Conference on USENIX Annual Technical*

Conference (ATC) (2016).

[77] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: a sandbox for portable, untrusted x86 native code. *Commun. ACM* (2010).

[78] Zhang, I., Raybuck, A., Patel, P., Olynyk, K., Nelson, J., Leija, O. S. N., Martinez, A., Liu, J., Simpson, A. K., Jayakar, S., Penna, P. H., Demoulin, M., Choudhury, P., and Badam, A. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2021).

[79] Zhang, Q., and Liu, L. Workload adaptive shared memory management for high performance network i/o in virtualized cloud. *IEEE Trans. Comput.* (2016).

[80] Zhao, K., Gong, S., and Fonseca, P. On-Demand-Fork: A microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (2021).

[81] Zhong, Y., Li, H., Wu, Y. J., Zarkadas, I., Tao, J., Mesterhazy, E., Makris, M., Yang, J., Tai, A., Stutsman, R., and Cidon, A. XRP: In-Kernel storage functions with eBPF. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2022).

[82] Zhu, X., She, G., Xue, B., Zhang, Y., Zhang, Y., Zou, X. K., Duan, X., He, P., Krishnamurthy, A., Lentz, M., Zhuo, D., and Mahajan, R. Dissecting overheads of service mesh sidecars. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)* (2023).

A Artifact Appendix

A.1 Abstract

Pegasus is a framework for transparent kernel bypass for local and remote communication.

A.2 Description & Requirements

A.2.1 How to access

The artifacts of Pegasus is published at https://github.com/rssys/pegasus-artifact. It is also archived at https://doi.org/10.5281/zenodo.13714712. The Cloudlab profile for the artifacts is available at https://www.cloudlab.us/p/6fa2ef5e5b44c20a2d45dd80e53aee0c5bd3103a.

A.2.2 Hardware dependencies

Two servers with the following hardware are required: CPU with MPK support (Intel Xeon Skylake or later, AMD EPYC Zen 3 or later), Mellanox Connext-X 5 or later, and 64 GiB or more RAM. We recommend using the r6525 physical node type on Cloudlab, as we used for all the experiments in the paper. We refer the two servers as node0 and node1, as created on Cloudlab.

A.2.3 Software dependencies

An environment created by the Cloudlab profile specified in the artifact repository is required, which includes the following major dependencies: DPDK, F-Stack, Docker, and Kubernetes.

A.2.4 Benchmarks

The artifact repository contains all the benchmarks used in the paper for the performance of local and remote communication.

A.3 Setup

It is required to set up the machines by instantiating the Cloudlab profile specified in the artifact repository.

A.4 Evaluation workflow

A.4.1 Major Claims

The major claims are as follows:

- *(C1): Pegasus transparently achieves better performance than the Linux baseline for local communication.*
- *(C2): Pegasus achieves similar performance to other kernel-bypass networking systems for remote communication with transparent support for unmodified applications.*
- *(C3): Pegasus accelerates applications with both local and remote communication, and both optimizations contribute significantly.*

A.4.2 Experiments

Experiment (E1): [5 human-minutes + 1 compute-hour]: latency of synchronization primitives and protocol operations.

Run the scripts inside the following directory on node0 according to the documentation:

- `/data/experiments/microbenchmark/`

The results will be generated as a CSV table (Table 1). In this experiment, Pegasus shows significantly lower latency than Linux for the synchronization primitives and protocol operations shown in the table (C1).

Experiment (E2): [10 human-minutes + 5 compute-hours]: local communication performance.

Run the scripts inside the following directories on node 1 according to the documentation:

- `/data/experiments/local-proxy/`
- `/data/experiments/web/`
- `/data/experiments/servicemesh/`
- `/data/experiments/results/`

The results will be generated as Figure 2 and Figure 3. In this experiment, Pegasus shows significant better performance than Linux for local communication in terms of latency and throughput (C1). In Figure 2, Pegasus achieves higher throughput and similar or lower latency than the Linux baseline at all throughput. In Figure 3, Pegasus achieves better throughput than the Linux baseline with enough proxied requests, while showing overhead at lower percentage.

Experiment (E3): [10 human-minutes + 8 compute-hours]: remote communication performance.

Run the scripts inside the following directories on node 1 according to the documentation:

- `/data/experiments/redis/`
- `/data/experiments/nginx/`
- `/data/experiments/memcached/`
- `/data/experiments/tcp/`
- `/data/experiments/results/`

The results will be generated as Figure 4, Figure 5, Figure 6, Figure 7, and Table 3. In this experiment, Pegasus shows similar performance to other kernel-bypass networking systems for remote communication with transparent support for unmodified applications (C2). In Table 3 and Figure 4, Pegasus shows similar TCP latency to other kernel-bypass networking systems, which is significantly lower than the Linux baseline. In Figure 5, Figure 6, and Figure 7, Pegasus achieves similar throughput and latency to other kernel-bypass networking systems, such as F-Stack and Junction, and is significantly better than the Linux baseline and Demikernel.

Experiment (E4): [5 human-minutes + 2 compute-hours]: local and remote communication performance.

Run the scripts inside the following directories on node 1 according to the documentation:

- `/data/experiments/proxy/`
- `/data/experiments/results/`

In this experiment, Pegasus shows acceleration for both local and remote communication with its unified kernel-bypass networking (C3). The result will be generated as Figure 8.

We recommend using the following scripts on node1 according to the documentation to automatically run all the experiments for each system:

- `/data/experiments/exp-baseline.sh`
- `/data/experiments/exp-pegasus.sh`
- `/data/experiments/exp-f-stack.sh`
- `/data/experiments/exp-demikernel.sh`
- `/data/experiments/exp-junction.sh`

After running all the experiments, use the following script to generate all the results:

- `/data/experiments/results/generate.sh`.

Please refer to the documentation included in the artifact repository for more detailed information.