Understanding the Overheads of Hardware and Language-Based IPC Mechanisms

Zhaofeng Li
University of California, Irvine
Irvine, California, USA

Tianjiao Huang
University of California, Irvine
Irvine, California, USA

Vikram Narayanan
University of California, Irvine
Irvine, California, USA

Anton Burtsev
University of California, Irvine
Irvine, California, USA

Abstract

A recent surge of security attacks has triggered a renewed interest in hardware support for isolation. Extended page table switching with VMFUNC, memory protection keys (MPK), and memory tagging extensions (MTE) are just a few of the hardware isolation mechanisms that promise support for low-overhead isolation in recent CPUs. Along with the restored interest in lightweight hardware isolation mechanisms, safe programming languages like Rust has made a leap towards practical, zero-overhead safety implemented without garbage collection.

Both lightweight hardware mechanisms and zero-overhead language safety can be leveraged to enforce the isolation of subsystems, e.g., browser plugins, device drivers and kernel extensions, user-defined database and network functions, etc. However, as both technologies are still young, their relative advantages are still unknown. In this work, we study the overheads of hardware and software isolation mechanisms with the goal to understand their relative advantages and disadvantages for fine-grained isolation of subsystems with tight performance budgets. We ask two questions: What is the overhead of hardware isolation in an ideal scenario where the hardware isolation mechanism takes zero cycles? And if the safety of the Rust language can lower the overhead of cross-subsystem invocations, can the language on its own introduce overheads that might outweigh isolation advantages? To answer these questions, we develop and compare two carefully optimized versions of inter-process communication (IPC) mechanisms (one in safe Rust and one in a carefully-optimized assembly), and two identical (to the degree possible) DPDK-based network packet processing frameworks (one in C++ and one in Rust). Our analysis shows that for systems requiring frequent boundary crossings, a safe language is still beneficial even if the overheads of hardware isolation mechanisms drop to zero.

PLOS '21, October 25, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8707-1/21/10.
https://doi.org/10.1145/3477113.3487275

1 Introduction

Despite significant academic interest in the performance of hardware isolation primitives [17, 69, 71, 72], for decades they remained a low priority in commodity CPUs. On x86 machines, segmentation was deprecated as part of the transition from 32-bit to 64-bit addressing mode, leaving page tables as the only available isolation mechanism. Today, a carefully-optimized, page-based inter-process communication (IPC) mechanism requires 814 cycles (on Intel) and 783 cycles (on ARM) to perform a cross-address-space function call invocation [1].

A recent surge of security attacks, however, triggered a renewed interest in hardware support for isolation. Memory Protection Keys (MPKs) and Extended Page-Table (EPT) switching with VM functions [43] provide support for memory isolation with overheads gradually approaching [34, 52, 56, 68] the overhead of a function call. The newest ARM CPUs introduce support for 16-byte-granularity isolation with the Memory Tagging Extension (MTE) [4, 7], which is critical for enabling low-overhead software-fault isolation (SFI) implementations [46] and zero-copy exchange of data across isolated subsystems.

Along with the restored interest in lightweight hardware isolation mechanisms, safe languages have made a leap towards practical, zero-overhead safety. Rust is a new systems programming language that offers language safety without garbage collection [44]. Rust enforces type and memory safety through a restricted ownership model, assigning a unique owner to each live object in memory. This allows for static tracking of object lifetimes and thus static deallocation without a garbage collector. The runtime overhead of the language is limited to bounds checking, which in many cases can be concealed by modern out-of-order CPUs that can predict and execute the correct path around the check [21].

Both safe languages and hardware isolation mechanisms can be used for fine-grained isolation of small untrusted subsystems [6, 11, 31, 34, 40, 47, 55, 56, 68]. As both approaches

mature, questions of their practicality and relative advantages for isolation arise. On one hand, the overhead of switching the isolation boundary with a hardware instruction is becoming progressively lower, e.g., writing a `pkru` register that changes the tag of the page table introduces an overhead of only 20-26 cycles [34, 58]. However, it remains unclear what the total cost of a secure IPC invocation will be even in an ideal scenario when the switch of an isolation boundary takes zero cycles. On the other hand, while providing low-overhead cross-subsystem invocations with an overhead of a function call, a safe language itself can introduce overheads (due to said safety) that might outweigh the benefits of lightweight language-based isolation itself.

In this work, we explore the advantages and disadvantages of safe and unsafe isolation approaches, especially for systems in which fine-grained isolation, and hence frequent cross-subsystem invocations, are critical; e.g., browser plugins [61, 74], user-defined database functions [15, 66], network functions [2, 38, 41, 51, 59, 62], device drivers [27, 29, 67], storage systems [13], and kernel modules [12, 24, 28, 33, 35–37].

To understand the performance benefits of the two approaches (hardware-based isolation and language safety), we develop a carefully optimized implementation of a hardware-based IPC mechanism that assumes a zero-cycle cost of switching the isolation boundary. We then perform a detailed analysis of the overheads involved in implementing a secure cross-subsystem function invocation in both unsafe C++ and safe Rust.

We find that the overhead of implementing a secure IPC with a zero-cost hardware isolation primitive remains high. Two inherent reasons are the need to save and restore callee-saved general and extended registers and the requirement to switch to a new stack. Even if the cost of the hardware switch instruction drops to zero, a cross-subsystem invocation takes 111-406 cycles (Section 3).

In contrast to hardware-based IPC, safe languages provide isolation with the overhead of a function call [5, 9, 11, 30, 40, 55, 70]. The execution can continue on the same stack (safety ensures isolation of objects on the stack) and does not require saving and restoring general and extended registers (calling conventions save and restore registers between the caller and callee, and exception handling, i.e., *unwind*, mechanisms allow recovery from a fault in an untrusted subsystem).

To understand whether safety itself introduces prohibitive overhead in real applications, we develop two implementations (one in C++ and one in Rust) of a network function processing framework similar to Netbricks [57]. We analyze the performance difference between C++ and Rust by developing identical (to the degree possible) implementations of several representative network functions. We find that while high-level Rust abstractions can introduce significant overhead, a carefully planned (with respect to performance) Rust implementation remains fast and executes within 4-8% of unsafe C++.

Our analysis shows that for systems requiring frequent boundary crossings a safe language is still beneficial even if the performance of hardware isolation mechanisms drops to zero. Moreover, high-level safety and type guarantees allow safe languages to provide high-level isolation invariants, e.g., *fault isolation* and clean termination of crashing subsystems [6, 11, 40, 55].

Of course, safety comes at the price of restricting development to a specific language, and a larger trusted computing base—the language itself, and a trusted compilation environment required to ensure the safety of third-party extensions [11, 47, 55]. We, therefore, hope that our analysis of hardware-based IPC mechanisms will help hardware designers to improve the performance of hardware isolation primitives even further. Specifically, we argue that one of the two main sources of overheads—saving and restoring general and extended registers—should be optimized in hardware.

2 Anatomy of Safe and Unsafe IPCs

Before diving into the analysis of language and hardware-based IPC implementations, we discuss the internal organization of both mechanisms. The IPC mechanisms which arguably were one of the hottest areas of system research accumulate a long history of innovation aimed at improving security and performance through a broad range of abstractions and implementation designs that spanned microkernels [1, 3, 8, 20, 25, 31, 33, 39, 45, 48, 52, 63], operating systems [10, 34, 54, 56, 64, 68], hypervisors [26], language-based operating systems [5, 30, 40, 55, 70], software fault-isolation (SFI) frameworks [14, 23, 46, 50], and interface definition language compilers [18, 32].

Unsafe IPC path Traditionally, IPC mechanisms require assistance from a privileged, ring 0, kernel code to switch the address space between the callee and the caller. The kernel was responsible for saving the state of the caller, enforcing security checks, switching the address space, and finally, switching the execution from the caller to the callee [45]. Most recent hardware isolation primitives, e.g., MPK [43] and VMFUNC [31], provide support for an *exitless* IPC path, i.e., it is possible to implement a secure IPC mechanism that avoids entering the kernel and transitions between the caller and the callee through a small trusted trampoline directly in ring 3 [34, 52, 68]. The option to avoid exiting into the kernel on the critical path significantly reduces the cost of the IPC (on modern x86 hardware, the system call required to enter and leave the kernel takes 96-140 cycles without KPTI mitigations [34, 56]).

As an alternative to synchronous address space switches, on a multi-core machine, an IPC can be implemented as a cross-core invocation that relies on cache-coherence to transfer the message between the cores [8, 39, 54, 64]. While faster than address space switches, the cross-core invocations are

still expensive. A minimal call/reply invocation requires four cache-line transactions and takes 448-1988 cycles depending on whether the line is transferred between the cores of the same socket or over a cross-socket link [54].

To analyze the overheads of a minimal IPC path, we describe a minimalistic implementation of an exitless IPC (Listing 1). We assume the future hardware mechanism similar to MPK and VMFUNC that provides a way to instantly switch the address space between the caller and the callee with a non-privileged instruction (in our implementation we substitute such future instruction with a `nop`). Our IPC implements a *migrating threads* model of invocation [25], in which the caller thread enters the address space of the callee without the context switch. Specifically, the caller saves its state on the stack, switches into the address space of the callee with a hardware mechanism similar to `wrpkru` or `vmfunc`, picks a new stack inside the callee address space, and continues execution calling a callee function. To minimize the invocation cost, we pass a fixed number of arguments in registers following the C calling convention. While it is possible to implement a general calling convention that uses memory to pass messages of arbitrary length similar to seL4 [45], to keep our experiments concise, we instead implement a simple ABI that passes several arguments in registers. On the caller side, we first save extended registers with the `fxsave` instruction (lines 2–6) that takes a pointer to the memory location in the `rax` register (we allocate 512 bytes on the caller's stack which have to be 16 bytes aligned). We then save callee saved registers on the stack (lines 9–15) and zero out all general registers not used to pass the arguments and all extended registers (18–23). After that, we switch domain boundary executing a `nop` instruction under the assumption that the future hardware mechanism will have a one cycle overhead (line 31).

Inside the callee domain, we try to allocate a new stack from a pool of available stacks. We maintain a lock-free stack data structure from which we dequeue elements with a single `cmpxchg` instruction. We first check if there is at least one free stack on the list by checking if the head of the list is empty (33–35) and then perform an attempt to dequeue one element (38–43). Here we assume that the global variable `RT_FREE_LIST` that maintains the head of the free list is accessible in the callee domain.

```
1     ; fxsave target must be 16-byte aligned
2     mov rax, rsp
3     sub rsp, 512
4     and rsp, -16
5     fxsave [rsp]
6     push rax
7
8     ; save callee-saved registers
9     push rbp
10    push rbx
11    push r12
12    push r13
13    push r14
14    push r15
15    pushfq
16
17    ; zero out registers (rax, rbx, rbp, rsp, r10-r15)
18    xor rax, rax
19    xor rbx, rbx
20    ...
21    xor r15, r15
22    ; zero out extended registers
23    vzeroall
24  %ifdef UNWIND
25    ; switch to kernel
26    nop
27    ; handle continuation stack
28    ...
29  %endif
30    ; switch to callee
31    nop
32  .try_getting_stack:
33    mov rax, [RT_FREE_LIST]
34    cmp rax, 0
35    jne .stack_available
36    ...
37  .stack_available:
38    mov r10, rax
39    add r10, CALL_NEXT
40
41    ; try to remove from free list
42    mov r11, [r10] ; .next
43    lock cmpxchg [RT_FREE_LIST], r11
44    jnz .try_getting_stack
45
46    ; set callee rsp
47    mov rsp, r10
48    ...
49    ; we are ready to call callee
50       ...
```

Listing 1. IPC path (Intel x86-64 ASM)

Safe IPC Path A safe language like Rust can provide isolation through the safety of a programming language. The language provides mechanisms to control access to the state of the program at module and class boundaries by specifying fields of individual objects as public or private. Isolated parts of the program have access to the state transitively reachable through public global variables and explicitly passed arguments. Control over references and communication channels allows isolating the state of the program on function and module boundaries enforcing confidentiality and integrity, and, more generally, constructing a broad range of least-privilege systems through a collection of object-capability patterns [53]. Recent systems develop support for dynamic loading of Rust extensions [11, 47, 55] hence enabling process-like development and execution environment in a safe language.

In a safe language, a minimal isolation boundary can be implemented as a private class or module [6, 47, 57]. In such a system the IPC mechanism is simply a function call invocation of a method exported by a protected subsystem. Naturally, this eliminates multiple overheads of unsafe IPC. First, the safety guarantees protect the general and extended registers between the callee and the caller, i.e., the calling convention ensures that original values of the callee registers are saved and restored, hence there is no need to save and restore the registers upon entering an untrusted callee subsystem. Second, the execution can continue on the same

stack, therefore eliminating the overhead of picking a new stack in the callee domain.

2.1 Unwind and Error Handling

The minimal IPC implementations presented above enforce confidentiality and integrity across isolated subsystems, i.e., one subsystem cannot read and modify data of other subsystems. These mechanisms, however, provide no way of isolating faults across subsystems—a crash in any of the isolated subsystems halts the entire system requiring a restart. To isolate a fault, the IPC subsystem should provide a mechanism to unwind the execution of a thread from a crashing subsystem returning an error to the caller.

Unsafe unwind To unwind execution of a thread from a crashing subsystem, our IPC code records the state of the thread right before entering the callee subsystem. Before switching into the callee, we first switch into the address space that can be accessed by the kernel (line 26). We assume that each thread has a region of thread-local memory accessible from inside the kernel, for example, relative to the `gs` register. Inside this portion of the kernel-accessible address space we implement the following logic on each invocation: 1) The IPC code checks if the subsystem is alive before performing the invocation. If the subsystem is alive, the IPC records the fact that the thread moves between subsystems. We use this information to unwind all threads that happen to execute inside the crashing subsystem. 2) For each invocation, the IPC code creates a lightweight *continuation* that captures the state of the thread right before the cross-subsystem invocation. Specifically, inside the kernel subsystem we save the caller's stack pointer—the register state of the caller is already saved on the caller's stack, therefore it is sufficient to save only the stack address. The continuation allows us to unwind the execution of the thread, and return an error to the caller. If we have to unwind the thread, the kernel restores the stack to the state captured by the continuation and returns an error to the caller.

Safe unwind In a safe language, unwinding is possible with native language support for catching exceptions. The benefit of exception handling is that the state of all registers can be restored by iterating over the stack frames that contain the saved value for each register. This eliminates the need for explicit register saving and restoring on each invocation and instead incurs the costs of unwinding only if the thread panics.

Rust implements support for catching panics modeled after the Itanium C++ ABI [42]. To restore register state of the caller to the state at the cross-subsystem invocation, a common *personality routine* is invoked repeatedly for every stack frame, hence *unwinding* the stack up to the point at which the exception can be handled and an error can be returned to the user. Unwinding relies on the debugging information embedded into the binary to restore register state between stack frames.

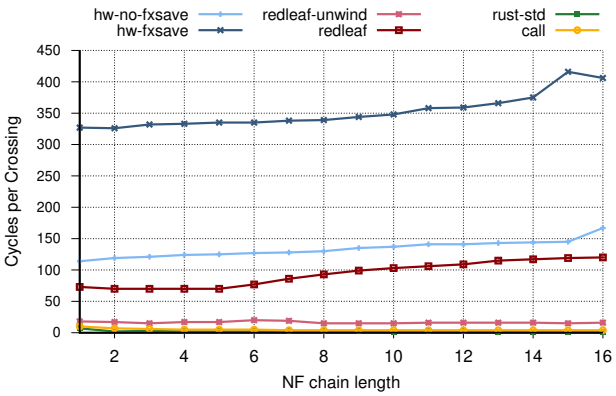

Figure 1. Overhead of Safe and Unsafe IPC

3 Performance Analysis

We run our experiments on CloudLab [60] c220g2 servers configured with two Intel E5-2660 v3 10-core Haswell CPUs running at 2.60 GHz, 160 GB RAM, and a dual-port Intel X520 10GbE NIC. Linux machines run 64-bit Ubuntu 20.04 with a 5.4.0 kernel configured without any speculative execution attack mitigations (`mitigations=off`) reflecting the trend of recent Intel CPUs addressing a range of speculative execution attacks in hardware. In all the experiments, we disable hyper-threading, turbo boost, CPU idle states, and frequency scaling to reduce the variance in benchmarking.

3.1 Safe vs Unsafe IPC

We first analyze the overheads of safe and unsafe IPC implementations discussed in Section 2. Specifically, we compare four IPC variants. First, we benchmark a minimal unsafe IPC which we described in Section 2: we assume a one cycle hardware primitive that switches the isolation boundary. We evaluate two different configurations of this IPC: with (`hw-fxsave`) and without (`hw-no-fxsave`) saving extended registers. Second, we implement a minimal safe IPC in Rust that utilizes the standard library and its unwind implementation to unwind from crashing subsystems, but provides no fault isolation mechanisms (`rust-std`). Finally, we implement our safe IPC ideas in the RedLeaf operating system [55]. RedLeaf supports fault isolation through a combination of heap isolation and cross-subsystem invocation proxying. Our experiments are aimed at evaluating the overhead of fault isolation. Specifically, we run two configurations: 1) an original RedLeaf implementation [55] that saves and restores all registers similar to our unsafe IPC (`redleaf`), and 2) our new implementation that relies on our custom unwind library that we implemented for RedLeaf (`redleaf-unwind`).

Our experiments measure IPC overheads on a chain of cross-subsystem invocations (Figure 1). We vary the length of the chain from 1 to 16. Each invocation simply invokes the next subsystem in a chain and then returns. In all experiments, we measure the total time to execute ten million iterations. An unsafe IPC implementation needs 111-164 cycles to

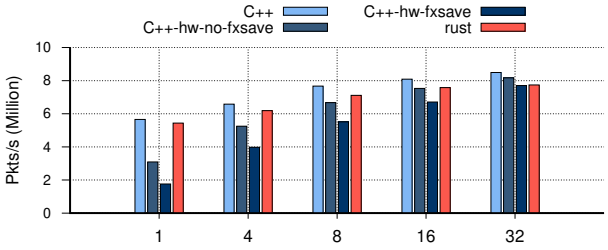

Figure 2. Isolation overheads on varying batch sizes

perform a null cross-subsystem invocation (`hw-no-fxsave`). The overhead is increasing as the invocation chain is growing. To explain performance degradation, we utilized the Intel VTune profiler to collect a range of hardware performance events related to cache and memory utilization. As the chain grows, saving and restoring general registers touches different memory pages that hold different stacks of the thread in each subsystem. The execution becomes memory-bound due to higher pressure on the L1 cache and data TLB. Saving and restoring general registers takes 35-71 cycles. The atomic exchange operation which is needed to dequeue a free stack in the callee domain takes 28-59 cycles. The atomic exchange instruction is the major contributor to the stack switching, without it picking a stack takes only 10 cycles. Saving and restoring extended registers introduces an overhead of additional 216-242 cycles at the total cost of 327-406 cycles for a cross-subsystem invocation (`hw-fxsave`). Zeroing all general registers introduces an overhead of only 3 cycles. With saving and restoring extended registers, the performance impact of the invocation chain is even more profound—the execution becomes memory-bound. Moreover, the need to save extended registers creates enough pressure that `fxrstor` instruction experiences L3 latency in 71% of accesses.

Safety guarantees of Rust allow us to perform cross-subsystem invocations with an overhead of a function call, 1-7 cycles (`rust-std`). As the execution continues on the same stack, Rust avoids cache and data-TLB bottlenecks by staying on the same continuous stack page. Our unwind-based RedLeaf IPC approaches the cost of a function call with only 16-18 cycles per domain crossing. To ensure safety and fault isolation, in RedLeaf the execution crosses through a proxy that checks if the callee subsystem is alive and moves the ownership of all `RRef<T>` arguments (we pass one `RRef<T>` in this test). This experiment highlights the importance of the unwind mechanism. Without unwind, an original version of RedLeaf IPC that creates a continuation takes 73-120 cycles (`redleaf`). Note, that RedLeaf disables the use of extended registers, hence they are not saved. It is still faster than an unsafe IPC as it can continue on the same stack.

In general, we observe that a safe, unwind-based invocation approach the performance of a non-isolated unsafe system.

3.2 Application Benchmarks

To understand the impact of IPC performance in real-world applications, we implement a network function virtualization framework similar to Netbricks [57]. Today, a wide range of *network functions* (NFs) handle the most complex network tasks such as intrusion detection, packet filtering, load balancing, etc. NFs are typically deployed as a part of a *service chain* that together processes a stream of packets.

In a modern network, NFs are often built as independent software by third-party vendors and have a set of unique requirements centered around performance, isolation, and reliability. NFs often have conflicting reliability and security goals and require isolation [49, 51, 65, 73, 75]. Isolation of NFs remains a challenging problem due to stringent performance requirements of packet processing applications [2, 38, 41, 51, 59, 62]. Traditional mechanisms that can enforce isolation boundaries — hardware primitives, software fault isolation (SFI), and language safety — impose overheads that are too high for systems that execute at line rate.

To understand the overheads of isolation, we implement the same network functions in C++ and Rust. Both implementations use the DPDK network processing framework [16] to provide low-overhead access to the network interface. Both the C++ and Rust versions operate on a batch of packets (we form the batch using the C DPDK functions which are a trusted part of the system, and hence require no isolation). Then we either call a C++ version of each network function (no isolation is provided) or enter the Rust environment that enforces isolation across individual functions of the network chain.

We implement four network functions: (1) **TTL** which decrements the time-to-live field in a packet's IPv4 header, (2) **NAT** which rewrites the source IP and port of a packet according to a mapping, (3) **ACL Firewall** which allows or drops a packet based on a list of pre-defined rules, and (4) **Maglev** which is a load balancer developed by Google to evenly distribute incoming client flows among a set of backend servers [19]. For each new flow, Maglev selects one of the available backends by performing a lookup in a hash table, the size of which is proportional to the number of backend servers (65,537 in our experiments). Consistent hashing allows even distribution of flows across all servers. Maglev then records the chosen backend in a hash table, a *flow tracking table*, that is used to redirect packets from the same flow to the same backend server. The size of the flow tracking table is proportional to the number of flows (we choose 1 M flows for our experiments). Processing a packet requires a lookup in the flow tracking table if it is an existing flow, or a lookup of a backend server and an insertion into the flow tracking table to record the new flow.

3.2.1 Overheads of Language Safety We first analyze the performance impact of a safe language—after all if safety

	TTL	NAT	ACL	Maglev
	C++/Rust			
Instructions	120/171	235/318	355/351	267/302
Cycles	57/70	149/139	110/142	197/216
Branches	13/22	25/33	44/37	20/33
Branch mispr.	0.04/0.06	0.04/0.05	0.12/0.06	0.06/0.09

Table 1. Microarchitectural comparison of C++ vs Rust

on its own introduces an overhead that is higher than hardware isolation in an unsafe language using safety for isolation does not make sense. We compare performance and several microarchitectural characteristics for the C++ and Rust implementations of our network functions (Table 1). Specifically, we collect the number of instructions generated by the compiler and the number of cycles required to execute each network function.

In general, for simple network functions like TTL, NAT, and ACL, the Rust and C++ code have similar characteristics, i.e., the Rust code stays within 13-35% of the number of generated CPU instructions for all but the trivial TTL function, and within 7-29% of cycles which are required to execute the function (taking fewer cycles than C++ on NAT). In all functions but ACL, Rust uses a higher number of branch instructions to implement bounds checks, and encounters a slightly higher rate of branch mispredictions.

To understand the performance impact of safety on real-world applications, we compare performance of the network function chain implemented in C++ and Rust on varying batch sizes (Figure 2). In our tests, we send 64-byte packets and measure the performance on several batch sizes ranging from 1 to 32 packets. We use a packet generator based on ixy [22]. The generator generates 64-byte IPv4 UDP packets at line rate and cycles through 1 M different source IP addresses to simulate the presence of multiple flows.

Overall, Rust is 4-8% slower. The major difference is in the use of high-level abstractions which differ between the two languages. For example, Rust relies on the notion of *interior mutability* to break strict ownership rules through a collection of trusted, standard types, e.g., mutexes (`Mutex<T>`), reference-counted pointers (`Rc<T>` and `Arc<T>`), etc., that enforce ownership at run-time. Naturally, run-time checks, and specifically additional pointer dereferences which create a higher cache pressure might negatively affect performance. Similar, high-level language abstractions, like option (`Option<T>`) types, add additional bytes to the data structure they wrap and hence break cache-line alignment [55].

3.2.2 Overheads of Isolation To understand the performance impact of various isolation mechanisms on real-world applications, we isolate network functions in C++ and Rust (Figure 2). We use Rust to enforce the confidentiality and integrity of each network function (`rust`). This is a default guarantee provided by Rust through its safety—each network function can access the state of the program that is reachable through public variables and fields (a similar isolation scheme was implemented by Netbricks [57]). We compare the Rust implementation against the C++ version that relies on hardware-based isolation. To understand the impact of saving and restoring extended registers, we evaluate two configurations: one that uses extended registers and hence has to save them as part of the IPC (`C++-hw-fxsave`) and one that disables the use of extended registers (`C++-hw-no-fxsave`). Our goal is to evaluate whether the use of extended registers can outweigh the cost of saving and restoring them on cross-subsystem invocations.

On small batch sizes, the cost of hardware-based IPC isolation impacts the performance of the network function chain. Without extended SIMD registers, on a batch of one, an isolated C++ chain (`C++-hw-no-fxsave`) achieves only 54% performance of non-isolated code. Saving and restoring extended SIMD registers adds significant overhead, allowing the configuration that uses them (`C++-hw-fxsave`) to achieve only 31% performance of non-isolated code. The overheads of hardware IPC become amortized on larger batch sizes, allowing both SIMD and non-SIMD versions to match and even outperform by 5% Rust equivalents. Overall, while extended registers provide tremendous optimization opportunities, in simple network functions the performance of SIMD and non-SIMD code is nearly identical (we measure it to be within 1%). Naturally, with the current cost of saving and restoring extended registers, their benefit can be realized only in carefully optimized vectorized code.

4 Conclusions

After decades of relatively slow adoption, we finally see a renewed interest in hardware isolation mechanisms. Fortunately, this interest coincides with rapid progress in the domain of practical language safety designed to support the development of low-level systems code and hence provide an alternative way to implement isolation. We study the overheads of hardware isolation in an ideal scenario—the address space switch takes zero cycles—and compare the performance of hardware mechanisms with isolation enforced through the safety of Rust. Our analysis shows that even in this ideal scenario the cost of hardware isolation remains high due to the need to save general and extended register state and the requirement to switch stacks between isolated subsystems. Rust avoids these two overheads implementing isolation with an overhead of a function call. Moreover, we observe that on realistic workloads Rust incurs only minor overhead of 4-8% compared to unsafe C++.

Acknowledgments

We would like to thank PLOS'21 reviewers for various insights helping us to improve this work. This research is supported in part by the National Science Foundation under Grant Numbers 1837127 and 1840197, and VMWare.

References

[1] seL4 Performance. https://sel4.systems/About/Performance/.

[2] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. XOMB: Extensible open middleboxes with commodity servers. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS'12, pages 49–60, New York, NY, USA, 2012.

[3] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, R Wisniewski, and Jimi Xenidis. Utilizing Linux kernel components in K42. Technical report, IBM Watson Research, 2002.

[4] Arm. Armv8.5-A Memory Tagging Extension white paper. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[5] Godmar Back and Wilson C Hsieh. The KaffeOS Java Runtime System. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):583–630, 2005.

[6] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS'17)*, pages 156–161, 2017.

[7] Steve Bannister. Memory Tagging extension: Enhancing memory safety through architecture, August 2019. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety.

[8] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP'09, pages 29–44. ACM, 2009.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 267–283, 1995.

[10] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1):37–55, February 1990.

[11] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1–19, November 2020.

[12] Bromium. Bromium micro-virtualization, 2010. http://www.bromium.com/misc/BromiumMicrovirtualization.pdf.

[13] Anton Burtsev, Kiran Srinivasan, Prashanth Radhakrishnan, Lakshmi N Bairavasundaram, Kaladhar Voruganti, and Garth R Goodson. Fido: Fast inter-virtual-machine communication for enterprise appliances. In *Proceedings of the 2009 USENIX Annual Technical Conference (USENIX ATC'09)*, pages 25–25, 2009.

[14] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP'09, pages 45–58. ACM, 2009.

[15] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM Trans. Comput. Syst.*, 6(1):28–50, February 1988.

[16] Intel Corporation. DPDK: Data Plane Development Kit. http://dpdk.org/.

[17] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. XPC: Architectural support for secure and efficient cross process call. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA'19, pages 671–684, New York, NY, USA, 2019.

[18] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing IDL compiler. In *ACM SIGPLAN Notices*, volume 32, pages 44–56. ACM, 1997.

[19] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, pages 523–535, March 2016.

[20] Kevin Elphinstone and Gernot Heiser. From L3 to SeL4 What Have We Learnt in 20 Years of L4 Microkernels? In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*, pages 133–150, 2013.

[21] Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, et al. The Case for Writing Network Drivers in High-Level Programming Languages. In *Proceedings of the 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–13. IEEE, 2019.

[22] Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, et al. The Case for Writing Network Drivers in High-Level Programming Languages. In *Proceedings of the 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–13. IEEE, 2019.

[23] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI'06, pages 75–88, 2006.

[24] Feske, N. and Helmuth, C. Design of the Bastei OS architecture. 2007.

[25] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a Migrating Thread Model. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference (WTEC '94)*, pages 97–114, 1994.

[26] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, Mark Williamson, et al. Safe hardware access with the xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on Demand IT InfraStructure (OASIS)*, pages 1–1, 2004.

[27] Vinod Ganapathy, Matthew J Renzelmann, Arini Balakrishnan, Michael M Swift, and Somesh Jha. The design and implementation of microdrivers. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 168–178. ACM, 2008.

[28] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 193–206, 2003.

[29] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J Elphinstone, Volkmar Uhlig, Jonathon E Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, pages 109–114. ACM, 2000.

[30] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX Operating System. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (USENIX ATC'02)*, pages 45–58, 2002.

[31] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *2020 USENIX Annual Technical Conference (USENIX ATC'20)*, pages 401–417, July 2020.

[32] Andreas Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig. Stub-code performance is becoming important. In *Proceedings of the 1st Workshop on Industrial Experiences with Systems Software*, October 22 2000.

[33] Hardy, N. KeyKOS architecture. *ACM SIGOPS Operating Systems Review*, 19(4):8–25, 1985.

[34] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 489–504, July 2019.

[35] Heiser, G. and Elphinstone, K. and Kuz, I. and Klein, G. and Petters, S.M. Towards trustworthy computing systems: taking microkernels to the next level. *ACM SIGOPS Operating Systems Review*, 41(4):3–11, 2007.

[36] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[37] Hohmuth, M. and Peter, M. and Härtig, H. and Shapiro, J.S. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *Proceedings of the 11th ACM SIGOPS European Workshop*, page 22. ACM, 2004.

[38] Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. MSwitch: A Highly-Scalable, Modular Software Switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR'15, New York, NY, USA, 2015.

[39] Tomas Hruby, Herbert Bos, and Andrew S. Tanenbaum. When slower is faster: On heterogeneous multicores for reliable systems. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 255–266, San Jose, CA, June 2013.

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

[41] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 445–458, Seattle, WA, April 2014.

[42] Intel Corporation. Itanium C++ ABI: Exception Handling, May 2005. `https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html`.

[43] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2020. `https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html`.

[44] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2019.

[45] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 207–220, 2009.

[46] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the 12th European Conference on Computer Systems*, pages 437–452, 2017.

[47] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 627–643, Carlsbad, CA, October 2018.

[48] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two years of experience with a μ-kernel based OS. *SIGOPS Oper. Syst. Rev.*, 25(2):51–62, April 1991.

[49] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, pages 218–233, 2017.

[50] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software Fault Isolation with API Integrity and Multi-Principal Modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, pages 115–128, 2011.

[51] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 459–473, Seattle, WA, April 2014.

[52] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19)*, 2019.

[53] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, May 2006.

[54] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs: Towards Isolation of Kernel Subsystems. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 269–284, July 2019.

[55] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 21–39, November 2020.

[56] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight Kernel Isolation with Virtualization and VM Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'20)*, pages 157–171, 2020.

[57] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 203–216, Savannah, GA, November 2016.

[58] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, July 2019.

[59] Kaushik Kumar Ram, Alan L. Cox, Mehul Chadha, and Scott Rixner. Hyper-Switch: A Scalable Software Virtual Switching Architecture. In *2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pages 13–24, San Jose, CA, June 2013.

[60] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), December 2014.

[61] David Sehr, Robert Muth, Cliff L. Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th USENIX Security Symposium*, pages 1–11, 2010.

[62] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, pages 323–336, San Jose, CA, April 2012.

[63] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. EROS: a fast capability system. In *Proceedings of the seventeenth ACM symposium on Operating Systems Principles (SOSP'99)*, pages 170–185, 1999.

[64] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010.

[65] Radu Stoenescu, Vladimir Olteanu, Matei Popovici, Mohamed Ahmed, Joao Martins, Roberto Bifulco, Filipe Manco, Felipe Huici, Georgios Smaragdakis, Mark Handley, and Costin Raiciu. In-Net: In-network processing for the masses. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015.

[66] Mark Sullivan and Michael Stonebraker. Using Write Protected Data Structures To Improve Software Fault Tolerance in Highly Available Database Management Systems. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB'91, pages 171–180, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[67] Michael M Swift, Steven Martin, Henry M Levy, and Susan J Eggers. Nooks: An Architecture for Reliable Device Drivers. In *Proceedings of the 10th workshop on ACM SIGOPS European Workshop*, pages 102–107, 2002.

[68] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*, pages 1221–1238, August 2019.

[69] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. CODOMs: Protecting Software with Code-centric Memory Domains. In *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 469–480, June 2014.

[70] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-Kernel: A Capability-Based Operating System for Java. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, pages 369–393. 1999.

[71] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 31–44, 2005.

[72] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.

[73] Kenichi Yasukata, Felipe Huici, Vincenzo Maffione, Giuseppe Lettieri, and Michio Honda. HyperNF: Building a high performance, high utilization and fair NFV platform. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 157–169, New York, NY, USA, 2017.

[74] Bennet Yee, David Sehr, Greg Dardyk, Brad Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy (Oakland'09)*, IEEE, 3 Park Avenue, 17th Floor, New York, NY 10016, 2009.

[75] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K.K. Ramakrishnan, and Timothy Wood. Flurries: Countless fine-grained NFs for flexible per-flow customization. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 3–17, New York, NY, USA, 2016.