Fast Privileged Function Calls

Till Miemietz[*]
Barkhausen Institut

Maksym Planeta[*]
TU Dresden

Viktor Reusch[*][†]
Barkhausen Institut

Jan Bierbaum
TU Dresden

Michael Roitzsch
Barkhausen Institut

Hermann Härtig
TU Dresden

ABSTRACT

We are approaching a world, where the CPU merely orchestrates a plethora of specialized devices such as accelerators, RDMA NICs, or non-volatile memory (NVM). Such devices operate by mapping their internal memory directly into an application's address space for fast, low-latency access. With the latency of modern I/O devices low enough to make traditional system calls a performance bottleneck, kernel interaction has no place on the data path of microsecond-scale systems. However, kernel bypass prevents the OS from controlling and supervising access to the hardware.

This paper tries to make a step back by bringing the OS to the critical path again, but with a reduced performance penalty. We pick up on previous ideas for reducing the cost of kernel interaction and propose the *fastcall space*, a new layer in the traditional OS architecture that hosts specialized and quickly accessible OS functions called *fastcalls*. Fastcalls can stay on the critical path of a microsecond-scale application because the invocation of fastcall-space functionality is up to 15 times faster than calling kernel functions from user space. We present and evaluate a prototype implementation of the fastcall framework and thereby show how much the overhead of calling into privileged mode can be reduced while using standard CPU features.

1 INTRODUCTION

Today, the operating system kernel is isolated from user applications by the tight barrier of CPU privilege levels, shielding the kernel from the user for security and safety reasons. In a traditional system, user applications have to cross this barrier by invoking *system calls*, e.g. for sending a network packet. However, modern I/O devices are so fast that the transition between user and kernel mode makes up for a significant share of the overall I/O latency [27, 48].

Accessing devices through the system call layer is slow for two reasons: First, performing a system call incurs a considerable performance penalty for the privilege transition, aggravated by mitigations against speculation-based side-channel attacks. This aspect alone adds up to 300 ns (see Section 4) to the *datacenter tax* [19, 37] for each system call invocation. The latency of a high-end InfiniBand NIC can be as low as 600 ns [36], making system calls prohibitively expensive on the critical path of applications [4, 37, 50]. Second, the hot path in OS kernels like Linux [28] can be long and include several latency-inflating asynchronous calls that require multiple intra-kernel context switches.

Kernel-bypass architectures remove these performance pitfalls by mapping devices directly into user applications [4, 16, 27, 37, 48]. But these architectures also remove devices from the control of the OS and thus rely on the hardware to offer an extended feature set.

[*]These authors contributed equally to this work.
[†]Also with TU Dresden.

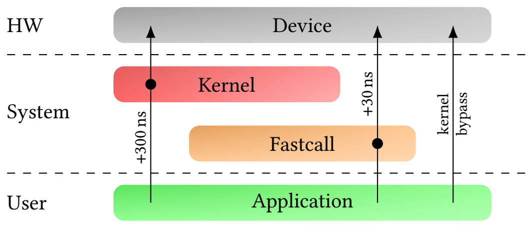

Figure 1: Fastcall system layer. Accessing a device through a system call adds up to 300 ns to the end-to-end latency, compared to kernel bypass. A fastcall adds ≈ 30 ns.

For instance with kernel-less networking stacks like RDMA, the NIC needs to provide features like multi-tenancy, QoS guarantees [20], connection tracking [14], or live migration [39] to provide the same functionality as OS kernels do. For devices lacking such support, kernel bypass requires the application to be trusted with full device access, which is viable only for a subset of infrastructure software. Moreover, the implementation of complex features in hardware increases both the cost of the device as well as the time-to-market for new features [1, 6].

To reconcile the goals of performance and OS control, we introduce the *fastcall space*, a layer within the OS that hosts *fastcalls*; functions for fast and supervised device access by user applications. Fastcalls are small code snippets tailored to specific use cases. They represent shortcuts of privileged operations and offer user programs efficient access to the hardware while keeping the OS in control. Fastcalls implement user logic, but are verified and trusted by the OS to adhere to the OS-enforced policies. In this aspect, fastcalls are similar to concepts like UDFs in the Exokernel architecture [7].

The fastcall space (see Figure 1) is a domain within a privileged CPU mode. In contrast to the kernel space, the fastcall space is not protected by software-based side channel mitigations [44], making the invocation of fastcalls faster than the invocation of system calls. However, this imposes some limitations on fastcalls, like not being able to access sensitive kernel information for security reasons. Our prototype implementation of fastcalls shows the minimal overhead for calling privileged functions on several CPU types, demonstrating the applicability of our concept. We conclude with an outlook on possible use cases for fastcalls.

2 THE FASTCALL ARCHITECTURE

Fastcalls provide a low-latency alternative to system calls by minimizing the transition overhead between user space and fastcall space. Moreover, each fastcall implements only the fast path of an application-specific use case, like sending a network packet of a pre-defined protocol type and with a fixed destination address.

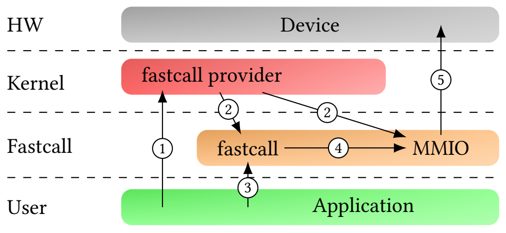

Figure 2: Fastcall usage. An application requests ① the fastcall provider to register a fastcall. The provider installs ② the fastcall, including the MMIO mapping, into the application's fastcall space. Now, the application can access the device by invoking ③ the fastcall. The fastcall uses ④ the MMIO region of the device to trigger ⑤ the corresponding I/O operation. Fastcalls and the MMIO region are inaccessible to the application.

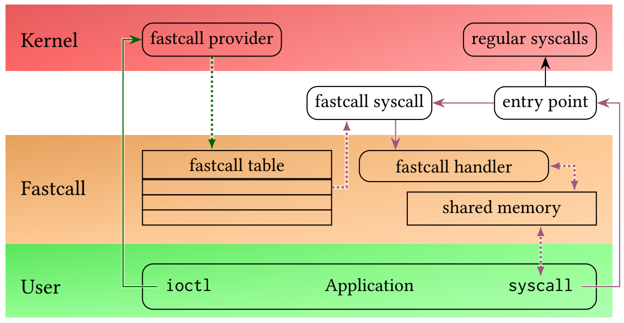

Figure 3: Registration (left) and invocation (right) of fastcalls in our `x86-64` implementation. Solid lines show the control flow, dotted lines the data flow. An application requests a fastcall from a fastcall provider using `ioctl`. This provider sets up the fastcall handler and inserts it into the application's fastcall table. Directly at the kernel entry point, each `syscall` invocation is routed either to kernel or fastcall space.

Fastcalls trade off the generality as well as some of the security of system calls for a faster implementation of privileged operations. This section describes the overall design principles of fastcalls.

The Fastcall Space. The main design goal for fastcalls is to incur minimal latency overhead when invoking them. To this extent, the code of a fastcall function is very simple and highly application-specific. Hence, similar to user spaces, each process in the system has its private fastcall space.

A fastcall has to securely enforce OS policies, so the user application must not be able to manipulate a fastcall's code or data. Using modern CPU models, this isolation is achieved by making the fastcall code accessible from within a privileged CPU mode only. The hardware forces the user to enter this privileged mode through fixed entry points defined by the OS. Unlike the standard code path for entering an OS kernel, a transition to the fastcall space always omits software-based side-channel mitigations to lower the overhead of fastcalls.

For the current implementation of the fastcall framework, the fastcall space lives in the standard kernel CPU mode. However, this is not by design. For instance, Apple's recent M1 CPU architecture introduces GXF; an alternative set of privilege levels orthogonal to traditional CPU modes [38]. We currently explore whether such features allow for the implementation of the fastcall space as well.

The Life Cycle of a Fastcall. When a process spawns, its fastcall space is empty; the application has no access to any fastcall. As shown in Figure 2, applications may issue requests to a *fastcall provider* for getting access to fastcall functions (step 1). A fastcall provider is a kernel component that generates the code of fastcall functions ad hoc and maps it into the fastcall space of the requesting process (step 2).

For example, in a scenario with high performance demands, an application could use fastcalls for issuing network packets to a NIC. To this extent, the user process applies for a suitable fastcall function at the fastcall provider using an interface specific for the respective use case. The code generated by the provider can contain OS-defined policies like a check that an application-generated network packet header contains only a permitted destination IP address.

Together with the fastcall code, the provider also maps resources, which the fastcall needs for its work, into the fastcall space of the target application. In our example, such a resource might be an MMIO window to communicate directly with the NIC.

Since device resources that are associated with fastcalls are only accessible from within the fastcall space, all operations of an application that affect these resources have to pass through the trusted fastcall code before reaching a device (steps 3 – 5). If an application invokes a fastcall using parameters that do not comply with the OS policies defined in this particular fastcall (like sending a network packet to an arbitrary address), the fastcall function denies the operation and returns to user space. The application can then either modify its request or invoke the standard OS stack by issuing a regular system call.

Security Considerations. As stated before, the entry procedure to the fastcall space omits software-based side channel mitigations to speed up the transition from user space. In particular, the fastcall space resides in the same virtual address space as the application and is not protected by measures like KPTI. Thus, while applications cannot change any data inside the fastcall space, in the presence of side-channel vulnerabilities like Meltdown [29], they can effectively read the data. So the fastcall space must not contain any information that should be kept secret from a user application.

Fastcall functions do not interact with the OS kernel in any way. A fastcall provider must verify that the code of a fastcall neither accesses kernel data nor calls any kernel functions, On systems that use a separate address space for their kernel, there is a strong isolation between fastcall space and kernel space. So fastcalls must be self-contained as, for security reasons, calling user functions is forbidden as well.

```
1  entry_SYSCALL_64: // kernel entry point
2    cmpq $NR_fastcall, %rax
3    je fastcall // branch to fastcall system call
4    /* original kernel entry sequence [...] */
5
6  fastcall: // fastcall dispatcher
7    cmpq $NR_TABLE_ENTRIES, %rdi
8    jae error // table index out of bounds
9    movq $TABLE_ADDR, %rax
10   imulq $TABLE_ENTRY_SIZE, %rdi
11   addq %rdi, %rax
12   jmpq *(%rax) // &some_fastcall_function
13
14 some_fastcall_function: // example fastcall
15   /* fastcall function body [...] */
16   movq <RETVAL>, %rax
17   sysretq // return to user space
```

Figure 4: Fastcall invocation on `x86-64`. System calls enter the kernel at label `entry_SYSCALL_64`. Lines 2 and 3 test for the fastcall system call number and branch off to the fastcall dispatcher, if needed. Register `%rdi` holds the number of the invoked fastcall function. Lines 7 to 11 compute the fastcall function's address using the fastcall number as an index into the fastcall table. Line 12 jumps to the fastcall function. Finally, the fastcall function returns to the application via `sysret`.

3 IMPLEMENTATION

This section describes the implementation of fastcalls for the Linux kernel [28] on the `x86-64` architecture[1]. Figure 3 presents a high-level view of this particular implementation. The per-process fastcall space hosts all facilities relevant for executing fastcalls: For each fastcall registered with a process, the *fastcall table* contains an entry with the metadata needed for executing the respective fastcall function. In addition to a pointer to the code of a fastcall function, a fastcall table entry comprises auxiliary data like configuration parameters or pointers to memory regions associated with a fastcall.

Besides a fastcall table, the fastcall space hosts a memory region shared with the user application for exchanging data that is too large to fit into CPU registers. If required by the fastcall function, the fastcall space can also hold device memory mappings and fastcall-private memory pages (e.g. for locks and counters). Fastcall functions written in C also require a stack. All operations in fastcall space run in the privileged mode of the CPU (ring 0), so fastcalls have privileges similar to the kernel and fastcall space memory is not accessible from user mode. To avoid concurrency issues, there is one dedicated stack per CPU and interrupts remain disabled during the fastcall execution.

The left side of Figure 3 shows the creation of a fastcall: An application requests access to fastcalls from a *fastcall provider*. In our implementation, fastcall providers are loadable kernel modules that interact with the fastcall infrastructure built into the modified kernel. If the application is authorized to use the requested fastcall, the fastcall provider adds it to the fastcall table of the calling process.

The right side of Figure 3 shows the invocation of a fastcall and Figure 4 provides a low-level view of the entry procedure. We use the standard system call interface of `x86-64` (`syscall`) and assign a new system call number to fastcalls. To minimize latency, fastcalls are distinguished from standard system calls right at the kernel entry point. If the entry routine detects a fastcall, control flow is forwarded to the *fastcall dispatcher*. The dispatcher locates the requested fastcall function via the fastcall table and executes it.

Fastcalls do not include the overhead of typical operations that the kernel performs on system call entry and exit. On `x86-64`, fastcalls avoid setting up and tearing down the kernel environment (e.g. `swapgs`) and mitigating side-channel attacks like Spectre [22], Meltdown, and Microarchitectural Data Sampling [45]. Fastcalls also avoid storing and restoring general purpose registers, the system call table lookup, and consistency checks which are unnecessary for the fastcall environment.

In essence, the presented design enables fastcalls to perform privileged operations without fully entering the kernel, which results in reduced latency compared to system calls. Even in the presence of currently known side-channel attacks, an application cannot interfere with or actively manipulate memory in the fastcall space. Therefore fastcalls are safe even when facing malicious user applications.

4 EVALUATION

To demonstrate that a fastcall space can be implemented efficiently using available CPU features, we conduct two sets of microbenchmarks with the fastcall mechanism described in Section 3. First, we compare the overhead of fastcalls for executing privileged functions to the overhead of alternative implementations. Second, we measure the overhead our fastcall implementation imposes on existing Linux infrastructure, namely process creation.

The fastest way to implement a kernel-supplied function is the vDSO library that is mapped into each user application [9]. Effectively being normal function calls without any mode transitions, vDSO functions serve as a lower bound for the overhead introduced by any of the mechanisms used in our experiments. Special-purpose system calls or `ioctl`-based handlers do run in privileged mode, so they serve as the main comparison point for fastcalls.

Table 1 lists the CPU models we conducted our experiments on. The selection covers the ISAs prevalent in today's data centers; Intel/AMD `x86-64` and ARM `aarch64`. For the benchmarks on ARM we ported our fastcall framework to that architecture. We used a Linux kernel of version 5.11 for all experiments and, if needed, modified it to support fastcalls. However, to prevent the in-kernel fastcall implementation from influencing the results of vDSO, system calls, and `ioctl`s, we used a vanilla kernel for these measurements. We also configured the CPUs to ensure consistent hardware conditions: Simultaneous multithreading ("Hyperthreading") and dynamic overclocking ("Turbo Boost") were switched off. CPU frequency and voltage scaling were disabled by selecting the "performance" CPU governor. The source code used for running the experiments described hereinafter is available online[2].

[1]The source code is available at https://github.com/vilaureu/linux/tree/fastcall.

[2]https://github.com/tmiemietz/fastcall-spma

Table 1: Median latency for invoking an empty function with side channel mitigations disabled.

CPU Model	Clock Rate [GHz]	vDSO [ns / cycles]	Fastcall [ns / cycles]	System Call [ns / cycles]	`ioctl` [ns / cycles]
AMD Ryzen 3700X	3.6	1 / 3	26 / 87	46 / 159	61 / 207
Intel Xeon Platinum 8375C	2.9	2 / 4	47 / 161	58 / 275	74 / 343
Intel Core i7-4790	3.6	2 / —[1]	24 / 100	47 / 172	73 / 306
AWS Graviton2 (Neoverse-N1)	2.5	3 / —[1]	35 / 105	97 / 250	120 / 304

Microbenchmarks. Table 1 shows the latency of an empty kernel-supplied function for different implementations and CPU types. Since no real work is performed in the functions, the numbers represent the overhead of the respective mechanisms. We repeated each experiment at least 10 000 times and present the median latency, both in nanoseconds (from `clock_gettime`) and CPU cycles (from `rdpmc`). To achieve the best performance possible for implementations inside kernel space (system call and `ioctl`), we fully disabled side-channel mitigations.

As expected, calling a vDSO function introduces nigh to no overhead. On recent Intel Xeon CPUs, the fastcall mechanism is able to yield a latency improvement of 19% compared to system calls and 36% compared to `ioctl` handlers, respectively. When considering older CPU models like the Intel Core i7-4790, the advantage of fastcalls increases. For the ARM Neoverse-N1, the performance gain of fastcalls is more pronounced then on `x86-64`, yielding speedups of 64% compared to system calls and 70% compared to `ioctl` handlers.

In conclusion, fastcalls significantly reduce the overhead of invoking privileged functions on all contemporary ISAs, making fastcalls the ideal mechanism for latency-sensitive use cases; see Section 5.

As part of our research on whether the GXF feature of Apple's M1 cores can be used for implementing fastcalls, we also measured the overhead of the associated mode transition instruction. Our early measurements find the roundtrip latency for using the alternative privilege modes of GXF is only 69 cycles, compared to 87 cycles required for `svc`/`eret`, ARM's equivalent of the `syscall`/`sysret` instructions. Thus, we believe that the performance of fastcalls on the ARM architecture could be reduced by another 18 cycles, if Graviton2 systems supported GXF.

Impact of Side-Channel Mitigations. As shown in Figure 5, the side-channel mitigation configuration of the kernel has a significant impact on the efficiency of the methods for implementing privileged OS functions. Particularly with older CPU models like the Intel Core i7-4790, enabling mitigations incurs a significant performance penalty for entering and leaving the kernel due to, for example, KPTI which causes additional address space switches in this situation. Here, fastcalls have a distinct latency advantage whenever the content of the privileged functions does not need to be secret.

For newer CPU models that have built-in mitigations for some side-channel attacks, the latency advantage of the fastcall mechanism with mitigations is less pronounced than on older CPU models.

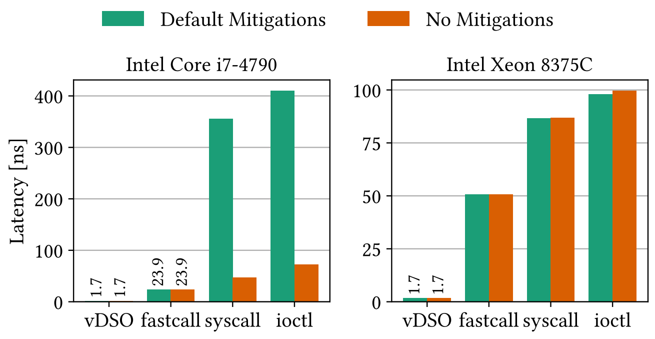

Figure 5: Performance comparison of empty system calls with different mitigation configurations. Note, that the performance of vDSO calls does not change with different mitigation settings and is equal to the results presented in Table 1.

However, as Figure 5 shows, when all side-channel countermeasures are in effect, the latency of an empty fastcall is still 33% lower than that of an equivalent system call.

Note, that while post-Meltdown processors do not show a significant performance degradation when enabling side-channel mitigations in the kernel, crossing the `syscall` barrier incurs a higher latency penalty in general. We believe CPU vendors should additionally offer a privilege transition instruction without side-channel mitigations. Programmers can then choose a trade off between security and latency that is appropriate for a specific use case.

Overhead of the Fastcall Mechanism. The implementation of fastcalls demands for several changes in other kernel subsystems like the memory management or the `fork` handler. Hence, we need to make sure that introducing fastcalls does not adversely affect the performance of other features, e.g. by requiring additional memory manipulation during `fork`. Figure 6 shows the impact of fastcalls on the performance of `fork`. If a process has no fastcalls registered, the time for `fork` is almost unchanged by introducing the fastcall feature to the kernel ("Without Registrations"). When there are fastcalls registered, the overhead for `fork` increases ("With Registrations") because the memory mappings involved in the fastcall mechanism have to be reset when spawning a new process.

The green bars in Figure 6 show the overhead caused by calling `fork` from a process with 100 fastcalls registered, each using two private memory pages. Also note, that the mitigation settings have no significant impact on the `fork` performance.

[1] Our measurement infrastructure could not produce precise results in these cases.

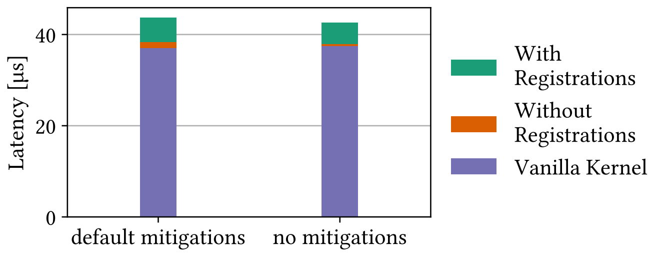

Figure 6: Overhead of the fastcall framework on the `fork` operation (CPU: AMD Ryzen 3700X)

Applicability Considerations. Fastcalls are designed to improve performance in comparison to system calls. If the overhead of a system call is negligible due to the long runtime of the operation performed, there is no need for fastcalls. For example, assume an overhead $O \leq 5\%$ to be negligible, a system call overhead with default mitigations $o_s = 355\,\text{ns}$, and a fastcall overhead $o_f = 24\,\text{ns}$ (corresponding to the Intel Core i7 in Figure 5). The total runtime of an invocation is $T = t+o$, where t is the time spent on the operation itself and the overhead o is either o_s or o_f. A fastcall is beneficial if the overhead of a system call is not negligible:

$$\frac{o_s}{T} = \frac{o_s}{o_s + t} > O$$

Solving for t, we get that the operation must not take more than 6.745 ns for fastcalls to offer a tangible benefit. If we additionally require the fastcall overhead to be negligible, the operation should take at least 456 ns. This lower limit is not a hard limit, but rather an indication that another solution, possibly a hardware-based one, can offer significantly better performance. To put these numbers into perspective, a high-end InfiniBand NIC provides back-to-back latency as low as 600 ns [36]. We believe this small example shows that few other microsecond-scale techniques can offer performance advantages over fastcalls.

5 USE CASES FOR FASTCALLS

We envision a framework for low-overhead execution of tailored OS functions that will cater to a variety of use cases. This section outlines the most promising ones.

Enforcing Network Policies. Consider a situation where the OS passes an Ethernet device to an application but wants to limit the egress traffic to a specific IP address. Normally, the OS must rely on hardware capabilities to enforce such policies, which often are not fine-grained enough. We propose to map the device's MMIO region into fastcall space and wrap the corresponding send function (`ibv_post_send` for RDMA and `rte_eth_tx_burst` for DPDK) with a fastcall. This way, the application can only send a packet by invoking the fastcall, and the corresponding wrapper will impose the checks required by the OS.

Fine-grained Access Control for NVM. Current CPUs offer memory protection only at the level of memory pages that are typically several kilobyte in size. Thus, applications cannot be granted direct access to a page without exposing all of its content. With NVM, however, this may become a problem. On the one hand, data structures protected by the OS should be packed densely to save persistent storage space. On the other hand, funneling all application accesses to such in-memory data structures through the system call interface lowers performance [32]. We envision fastcalls as a means for implementing secure and efficient access to fine-grained data structures located in NVM.

Fast Event Notification. The fastest way to receive event notifications is to busy poll on the corresponding event notification queue. Unfortunately, this method wastes energy and CPU time. Therefore, in many cases, the application requests the kernel to deliver event notifications; a mechanism that incurs significant latency overhead.

Fastcalls are able to improve wake-up times without resorting to busy polling by using the privileged `monitor`/`mwait` instructions. For an initial experiment, we adapted OpenMPI [10] to use fastcalls wrapping these instructions for node-local synchronisation. The performance is comparable to the regular polling mode:Across the NPB benchmark suite [34] we found a median overhead of 0.6% for the fastcall-based `monitor`/`mwait` mechanism.

This experiment strengthens our conviction that fastcalls are applicable in real-world scenarios and that they are easier to deploy than custom system calls. We plan to integrate this functionality into *libevent* [35], which will allow us to study the impact on a multitude of applications.

Augmenting Specialization. Fastcalls are installed on demand and can be parameterized for each specific instantiation. This property allows them to serve as an efficient implementation vehicle for OS specialization frameworks such as the Synthetix operating system [40]. By reducing the overhead for entering and exiting privileged CPU modes, fastcalls could also further speed up kernel software layer bypassing solutions like netmap [41]. For specialization, we envision fastcalls to be used together with modern approaches like JIT-compilation and a flexible infrastructure that allows for exposing arbitrary privileged operations through fastcalls.

6 RELATED WORK

This section puts the fastcall framework in the context of other works that try to reduce the cost of isolation mechanisms residing on the critical path of high-performance applications. These approaches either try to make transitions between isolation domains cheaper or to remove isolation boundaries altogether. Previous works already attempted to solve problems similar to ours. Unfortunately, they either turned out to be slow [25], were insecure [46], or relied on not-yet-existent hardware extensions [32]. We believe that fastcalls are able to solve all of the aforementioned shortcomings.

Removing Isolation Boundaries. An I/O-bound application can improve its performance by having direct access to the underlying device, thus removing the need to switch between kernel and user space. Referred to as *kernel-bypassing*, such architectures exist for networking [16, 27] as well as for storage devices [21, 48].

To achieve the best performance attainable, it is not enough just to pass a device to the application, one also needs an efficient API to access that device. This observation has been reflected in dataplane OS [4, 37, 49] and LibOS [7, 26, 30] architectures. Single-address-space systems completely avoid CPU-enforced isolation. Instead, these systems rely on language features and runtime checks to achieve isolation between different applications [23, 33] or software modules [47]. Our approach brings novelty to the existing architectures by maintaining the isolation boundary but making it as thin as it is technically possible.

Fast Transition Mechanisms. The reduction of mode transition latency can be addressed at a CPU-microarchitectural level, e.g. by providing faster instructions [13, 17, 31]. For example, Intel's discontinued Itanium architecture provided *promotion pages*, that are executable-only for user applications. Code inside these page can trigger a fast transition to privileged mode [18, Section 4]. In the most extreme case, the transition cost can be reduced to almost zero, as in case with vDSO [9]. As a disadvantage, vDSO offers only very limited functionality. Asynchronous system call mechanisms [3, 11, 43] and *io_uring* in asynchronous mode [5] also allow to reduce the transition latency but, in exchange, induce higher CPU utilization.

Simurgh [32] proposes an ISA extension for the x86 architecture that allows to mark kernel-level pages as *execute protected.* Invoking a special instruction (`jmpp`) causes the CPU to jump to an execute-protected page while simultaneously switching to privileged CPU mode. This way, the OS exposes a set of privileged functions to a user process through an execute-protected page. In contrast to Simurgh, the fastcall mechanism is available on standard CPUs today.

Critical path interposition. Traditionally, an external I/O-request (e.g. some request received over the network) will first be processed inside the kernel and then be passed to the user-space application, causing a context switch. The response will require a similar path. Multiple methods employed *eBPF* [8] to offload simple request processing routines from the user application into the storage [50], network [12], or scheduling [15] subsystems of the Linux kernel. This way, many context switches can be completely eliminated.

Both fastcalls and eBPF functions insert code snippets into the kernel, their purpose is vastly different however. Whereas eBPF programs effectively shift application code into the kernel, fastcalls are primarily designed for leveraging the efficient and secure deployment of kernel components in user space, thus reducing the amount of code running in privileged mode. Fastcalls thus aim at fostering an OS design closer to that of microkernels and exokernels.

In the context of kernel-bypass architectures the OS may want to achieve similar interposition, without impairing application performance. For that, the OS can employ SmartNICs [6, 42], programmable SSDs [24], or programmable switches [2] to filter, schedule, or process user application requests transparently. In contrast to interposition methods using smart hardware, fastcalls do not require custom extensions of I/O devices.

7 CONCLUSION

We built this paper on the observation that current mechanisms for the transition from applications to kernel code can become a performance bottleneck in modern systems. We propose the *fastcall* framework for enabling operating systems to protect security-critical resources like kernel or device memory from arbitrary access by applications. Fastcalls reduce the overhead for calling privileged functions to the bare minimum and thus yield better performance properties than traditional system calls. We implemented a prototype version of the fastcall framework and conducted microbenchmarks that showed latency improvements of up to 15× compared to system calls of a vanilla Linux kernel. Even though fastcalls offer a limited execution environment that only allows for running simple code snippets, they may be used to implement secure, fast, and CPU-efficient device accesses as well as to give applications protected access to privileged CPU instructions.

8 ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments. We furthermore thank Christian von Elm for conducting measurements on the Apple M1 CPU. This research was co-financed from the European Union's Horizon 2020 research program under grant agreement No. 957216, by the SAB (Development Bank of Saxony) under frameworks from both the ERDF (European Regional Development Fund) as well as the ESF (European Social Fund), and by public funding from the state of Saxony/Germany.

REFERENCES

[1] networking:toe [Wiki]. URL https://wiki.linuxfoundation.org/networking/toe.
[2] Bedrock: Programmable network support for secure RDMA systems. URL https://www.usenix.org/conference/usenixsecurity22/presentation/xing.
[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L Stillwell, David Goltzsche, David Eyers, Rudiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. page 17.
[4] Adam Belay, George Prekas, Christos Kozyrakis, Ana Klimovic, Samuel Grossman, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 49–65. ISBN 978-1-931971-16-4.
[5] Jonathan Corbet. Ringing in a new asynchronous i/o API. URL https://lwn.net/Articles/776703/.
[6] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Vivek Bhanu, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, NSDI'18. ISBN 978-1-931971-43-0.
[7] Dawson R Engler, M Frans Kaashoek, James O'Toole, and M I T Laboratory. Exokernel: An Operating System Architecture for Application-Level Resource Management. page 16.
[8] Matt Fleming. A thorough introduction to eBPF. URL https://lwn.net/Articles/740157/.
[9] Mike Frysinger. vdso(7). URL https://man7.org/linux/man-pages/man7/vdso.7.html.
[10] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104, Berlin, Heidelberg, September 2004. Springer. ISBN 978-3-540-30218-6. doi:10.1007/978-3-540-30218-6_19.
[11] Luis Gerhorst, Benedict Herzog, Stefan Reif, Wolfgang Schröder-Preikschat, and Timo Hönig. Anycall: Fast and flexible system-call aggregation. In *PLOS '21: Proceedings of the 11th Workshop on Programming Languages and Operating Systems, Virtual Event, Germany, October 25, 2021*, pages 1–8. ACM, 2021. doi:10.1145/3477113.3487267. URL https://doi.org/10.1145/3477113.3487267.
[12] Yoann Ghigoff, Julien Sopena, Gilles Muller, Kahina Lazri, and Antoine Blin. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. page 16.
[13] Charles Gray, Matthew Chapman, Peter Chubb, David Mosberger, and Gernot Heiser. Itanium - A system implementor's tale(awarded general track best student paper award!). In *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 265–278. USENIX, 2005. URL http://www.usenix.org/events/usenix05/tech/general/gray.html.

[14] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. MasQ: RDMA for Virtual Private Cloud. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '20, pages 1–14. Association for Computing Machinery. ISBN 978-1-4503-7955-7. doi:10/gg9rjq.

[15] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 588–604. Association for Computing Machinery. ISBN 978-1-4503-8709-5. doi:10/gm8ntm.

[16] InfiniBand Trade Association. *InfiniBand Architecture Specification*, volume 1. InfiniBand Trade Association, 1.3 edition. URL https://cw.infinibandta.org/document/dl/8567.

[17] Intel Corporation. Flexible return and event delivery (FRED). URL https://software.intel.com/content/dam/develop/external/us/en/documents-tps/346446-flexible-return-and-event-delivery.pdf.

[18] Intel Corporation. Intel Itanium architecture software developer's manual, May 2010. URL https://www.intel.de/content/dam/www/public/us/en/documents/manuals/itanium-architecture-software-developer-rev-2-3-vol-2-manual.pdf.

[19] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 158–169. Association for Computing Machinery. ISBN 978-1-4503-3402-0. doi:10/ghmjs6.

[20] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI, pages 113–125, . ISBN 978-1-931971-49-2. doi:10.5555/3323234.3323245.

[21] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A user-space i/o framework for application-specific optimization on NVMe SSDs. . URL https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/kim.

[22] Paul Kocher, Jann Horn, Anders Fogh, {and} Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[23] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. Singularity: Scientific containers for mobility of compute. 12(5):e0177459. ISSN 1932-6203. doi:10/f969fz.

[24] Dongup Kwon, Junehyuk Boo, Dongryeong Kim, and Jangwoo Kim. FVM: fpga-assisted virtual device emulation for fast, scalable, and flexible storage virtualization. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 955–971. USENIX Association, 2020. URL https://www.usenix.org/conference/osdi20/presentation/kwon.

[25] Hojoon Lee, Chihyun Song, and Brent ByungHoon Kang. Lord of the x86 rings: A portable user mode privilege separation architecture on x86. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1441–1454. ACM, 2018. doi:10.1145/3243734.3243748. URL https://doi.org/10.1145/3243734.3243748.

[26] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. FlexOS: Towards Flexible OS Isolation. URL http://arxiv.org/abs/2112.06566.

[27] LF Projects, LLC. Data plane development kit. URL https://www.dpdk.org/.

[28] Linux Kernel Organization, Inc. The linux kernel archives. URL https://www.kernel.org/.

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[30] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: library operating systems for the cloud. 48:461. ISSN 03621340. doi:10.1145/2499368.2451167.

[31] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 1–15. Association for Computing Machinery. ISBN 978-1-4503-6281-8. doi:10.1145/3302424.3303946. URL https://doi.org/10.1145/3302424.3303946.

[32] Nafiseh Moti, Frederic Schimmelpfennig, Reza Salkhordeh, David Klopp, Toni Cortes, Ulrich Rückert, and André Brinkmann. Simurgh: a fully decentralized and secure NVMM user space file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, pages 1–14. Association for Computing Machinery. ISBN 978-1-4503-8442-1. doi:10/gn4bjw.

[33] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and Communication in a Safe Operating System. pages 21–39. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram.

[34] NASA Advanced Supercomputing Division. NAS Parallel Benchmarks. URL https://nas.nasa.gov/Software/NPB/.

[35] Nick Mathewson, Azat Khuzhin, and Niels Provos. libevent – an event notification library. URL https://libevent.org/.

[36] NVIDIA. ConnectX-6 VPI Card Product Brief. URL https://www.mellanox.com/files/doc-2020/pb-connectx-6-vpi-card.pdf.

[37] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. 33(4):11:1–11:30. ISSN 0734-2071. doi:10.1145/2812806. URL https://doi.org/10.1145/2812806.

[38] Sven Peter. Apple silicon hardware secrets: Sprr and guarded exception levels (gxf). URL https://blog.svenpeter.dev/posts/m1_sprr_gxf/.

[39] Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoefler, and Hermann Härtig. MigrOS: Transparent Operating Systems Live Migration Support for Containerised RDMA-applications. In *USENIX ATC 2021*, pages 47–63. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/planeta.

[40] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: streamlining a commercial operating system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 314–321. Association for Computing Machinery. ISBN 978-0-89791-715-5. doi:10/c8w2pt.

[41] Luigi Rizzo. Netmap: a novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, page 9. USENIX Association.

[42] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, and Justine Sherry. We need kernel interposition over the network dataplane. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 152–158. Association for Computing Machinery. ISBN 978-1-4503-8438-4. doi:10.1145/3458336.3465281. URL https://doi.org/10.1145/3458336.3465281.

[43] Livio Soares and Michael Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 33–46. USENIX Association.

[44] The kernel development community. Page table isolation (PTI), . URL https://www.kernel.org/doc/html/v5.11/x86/pti.html.

[45] The kernel development community. Microarchitectural data sampling (MDS) mitigation, . URL https://www.kernel.org/doc/html/v5.11/x86/mds.html.

[46] Amit Vasudevan, Ramesh Yerraballi, and Ashish Chawla. A high performance kernel-less operating system architecture. In *Proceedings of the Twenty-eighth Australasian conference on Computer Science - Volume 38*, ACSC '05, pages 287–296. Australian Computer Society, Inc. ISBN 978-1-920682-20-0.

[47] Robert Wahbe. Efficient data breakpoints. In Barry Flahive and Richard L. Wexelblat, editors, *ASPLOS-V Proceedings - Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, USA, October 12-15, 1992*, pages 200–212. ACM Press, 1992. doi:10.1145/143365.143518. URL https://doi.org/10.1145/143365.143518.

[48] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. doi:10.1109/CloudCom.2017.14. ISSN: 2330-2186.

[49] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath OS architecture for microsecond-scale datacenter systems. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 195–211. ACM, 2021. doi:10.1145/3477132.3483569. URL https://doi.org/10.1145/3477132.3483569.

[50] Yuhong Zhong, Hongyi Wang, Yu Jian Wu, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. BPF for storage: an exokernel-inspired approach. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 128–135. Association for Computing Machinery. ISBN 978-1-4503-8438-4. doi:10.1145/3458336.3465290. URL https://doi.org/10.1145/3458336.3465290.