New Mechanism for Fast System Calls

Till Miemietz*, Maksym Planeta**, and Viktor Laurin Reusch*\textsuperscript{1,2}

\textsuperscript{1}Barkhausen Institut, Germany \textsuperscript{2}TU Dresden, Germany

Abstract

System calls have no place on the fast path of microsecond-scale systems. However, kernel bypass prevents the OS from controlling and supervising access to the hardware. In this paper we introduce the fastcall space, a new layer in the traditional OS architecture, that hosts fastcalls. A fastcall implements the fast path of a traditional kernel operation and can stay on the fast path, because the transition to the fastcall space is $\approx 15$ faster than to the kernel space. This way the OS does not give up the control over device access, whereas the applications maintain their performance.

1 Introduction

The system call layer creates a tight barrier in the transition between the user and the kernel modes, shielding the kernel from the user for security and safety reasons. In a traditional system, user applications must cross this barrier to write a file to a disk or send a network packet. Recently however, for high-performance applications, the cost of a system call executed on a performance critical path becomes prohibitively expensive [4, 43, 60].

The kernel-bypass architectures remove the transition barrier and the corresponding system call overhead by mapping the devices directly into the user-space applications [4, 21, 27, 35, 43, 58]. Kernel bypass takes the devices away from the control of the OS and relies on the devices to implement features like QoS [30], connection tracking [17], or live migration [44]. Instead, we introduce the fastcall space, a new layer within the existing OS architectures, enabling low-overhead OS-controlled device access by user applications.

Accessing devices through system call layer is slow for two reasons. First, the switch between kernel and user modes has considerable performance penalty, aggravated by recent speculation-based side-channel-attack mitigations [32, 37]. By itself, this aspect adds around 300ns (see Section 5) to the datacenter tax [28, 43] for each system call invocation. Second, the hot path in the Linux kernel [36] can be long and have several latency-inflating asynchronous calls requiring multiple intra-kernel context switches. For example, a write to a file goes through file system and block layers, before landing with the device controller driver. If the application intends to use a modern PCIe 4 $\times$ 16 device close to its maximum request processing rate, it requires an alternative to the system-call path.

Therefore, direct device access is necessary for a high performance application but has multiple prerequisites. First, the device becomes responsible for sharing the resources between multiple user applications like, for example, in RDMA-networks [21]. Second, if such device support is not available, kernel bypass can be used only when the application is trusted to have full device access. The first model requires additional device capabilities, increasing the cost of the device and increasing the time span for deploying new features [1, 9]. The second model is viable only for a subset of infrastructure-level software, limiting the applicability of kernel bypass in practice. Fastcalls combine the features of system calls and kernel bypass to offer cost-efficient access to the hardware without limiting the applicability of kernel bypass.

The fastcall layer resides in a privileged domain but avoids
Figure 2: An address space consists of a kernel and a user space. As part of KPTI, the kernel maintains two sets of address spaces for each process and must switch between corresponding page tables, when crossing the privilege boundary.

fully entering the kernel space (see Figure 1). Therefore, invoking a fastcall is much faster than invoking a system call. This speedup that fastcalls provide comes at the cost of some limitations though, like not being able to access sensitive information inside the kernel [32, 37]. The fastcall layer hosts multiple fastcall functions, code snippets tailored to specific use cases, that represent fast paths of privileged operations. Fastcall functions are provided and trusted by the OS, and hence they are allowed to execute in the privileged CPU mode and to have direct access to hardware.

One way to look at fastcalls is CPU-onloading, i.e. moving work from the hardware accelerators to the CPU with the purpose of reducing data movement. Unnecessary data movement has been often identified as a source of performance bottlenecks [5, 26, 28], therefore doing work on a CPU may actually improve performance. Moreover, our intention is to use fastcalls for running control logic, which is traditionally executed most efficiently by general-purpose CPUs. This way fastcalls can complement CPU-offloading and become part of future multitenant environments.

2 Background

Fastcalls build upon and extend the existing Linux kernel system call mechanism. We implement fastcalls for the x86-64 architecture, and correspondingly provide details for this architecture, although fastcalls can be generalized to other CPU and OS architectures as well.

Figure 2 illustrates the specifics of the system call implementation of the Linux kernel [54]. When an application wants to perform a system call (e.g. open, read, or write), it moves the system call number into register rax and the system call parameters into other registers. Then, the application executes the syscall instruction (Figure 2, 1), resulting in the following operations: transfer of the execution flow to a preconfigured kernel entry point (Figure 2, 2), transition of the CPU into privileged mode, deactivation of interrupts, and storing the return address in the rcx register [24, pp. 2984, 1854]. Typical kernel operations can only be executed in the privileged mode, and the syscall instruction guarantees that there only exists a single entry point into the kernel code, which can be firmly secured against attacks.

Immediately after entering the kernel most of the kernel memory is not valid, because it is not mapped into the current address space for security reasons [51]. The kernel starts by setting up the kernel stack and then changes page tables by storing a new value into the cr3 register (Figure 2, 3, [24, p. 2857]). Changing page tables is a part of KPTI [50], a technique for protecting the kernel data from speculation-based side-channel attacks [37] and is not required for all CPUs. Only after changing the page tables the rest of kernel code and data becomes accessible, and the control goes to the actual system call handlers (Figure 2, 4).

The system call handler identifies the exact system call by the number in the register rax and then dispatches it for processing to the corresponding kernel subsystem. After the system call is complete, the kernel returns from the do_syscall function and performs the previously taken actions in reverse (Figure 2, 5): restores the original user-view of the address space, including the stack and registers, and sets up the correct return address. Finally, the sysret instruction returns the execution flow from the privileged CPU mode back to the user application.

It takes overall around 300 ns to reach the required kernel subsystem and return from there (see Section 5). This administrative overhead is prohibitive for microsecond-scale applications, especially if the actual functional part of the system call is short [35, 43, 59]. Unfortunately, this overhead cannot be entirely avoided, because system calls are supposed to execute privileged operations, which normally cannot be delegated to user mode.

The largest source of overhead are the page table switch and other mitigations of speculation-based attacks. We analyze the sources of the overhead in more detail in Section 5. Further sources of overhead come from complicated kernel logic, which may not always be optimized for the fast path. In the next section, we describe how fastcalls avoid most of the overhead connected to system call invocation without sacrificing security.

3 Design

Fastcalls are intended to be used similarly to system calls. Unlike a system call, a fastcall has a very specific purpose (sending a packet a specific destination or writing into a specific set of blocks) and covers only the fast path. Figure 3 depicts a new layer in the OS architecture: the fastcall space. Like the kernel, the fastcall space resides in a privileged domain, that the user application normally has no access to. When invoking a fastcall, similarly to a system call,
The flow of managing fastcalls is shown. One of these 
installations, therefore limiting the frequency of device accesses .

A system can have multiple fastcall providers residing in the kernel or running as trusted user-level processes. For our evaluation we implemented fastcall providers as kernel modules. Only after the fastcall has been mapped into the application’s fastcall space, can it be called with minimum latency and without going into the kernel.

All fastcalls can be tuned individually to ensure a specific security or resource usage policy at the moment of fastcall invocation. As an example of a security policy, the fastcall may check that the application-generated network packet header contains only a permitted destination IP address. A resource usage policy may rate-limit the frequency of fastcall invocations, therefore limiting the frequency of device accesses.

The kernel must ensure that the fastcall is self-contained in terms of memory accesses and executed code. This means that the fastcall may neither perform memory operations outside a dedicated region nor is it allowed to execute any code that is not explicitly part of it, like external functions. There are two reasons for these limitations: First, to enable the highest performance and the simplest implementation, the fastcall may not generate CPU exceptions like page faults. Second, all possible execution paths of a fastcall must be observable by the fastcall provider to enable verification of the fastcall logic. Although we do not cover fastcall verification in this paper, we expect fastcalls to be automatically verified similarly to eBPF [11] functions.

To guarantee that a fastcall causes no page faults, each fastcall receives a block of scratchpad memory to be used as a stack. Additionally, the kernel creates a pinned memory region shared between the fastcall and the application. A part of the shared region is writable by the application, another part is only readable. The application can pass data to a fastcall only through CPU registers or via the aforementioned shared memory region.

### 4 Implementation

We implemented the concept of fastcalls in the Linux kernel [36]. This implementation augments the system call handling of the kernel to allow the execution of low-latency fastcall functions. The execution of fastcall functions in a higher CPU privilege mode isolates fastcalls from applications and enables these functions to access system pages, which user mode cannot manipulate. Overall, this implementation of the fastcall design largely focuses on extending kernel facilities and reducing the latency of application–kernel transitions.

The high-level view of the fastcall implementation is depicted in Figure 4. The flow of managing fastcalls is shown on the left. Authorized applications register for fastcalls via fastcall providers, which are loadable kernel modules. They then interact with the built-in fastcall infrastructure in the kernel. Modules allow administrators to extend systems with new functionality and aid the rapid development of new fastcalls.

The right side of Figure 4 depicts the invocation of fastcalls through the normal system call interface. Figure 5 provides a more detailed view of this invocation. To minimize fastcall latency, fastcalls are detected right at kernel entry and forwarded to the fastcall dispatcher. The dispatcher locates the desired fastcall function in memory and executes it. All operations in the fastcall space run in the privileged mode, meaning that the fastcalls have kernel-level capabilities.

The fastcall space hosts all the facilities relevant for executing fastcall functions as shown in Figure 4. One of these

---

Figure 3: Fastcall invocation. First, the application requests (1) the fastcall provider (FCP) to register a fastcall (FC). The FCP installs (2) the fastcall, including the MMIO mapping, into the application’s fastcall space. Later, the application accesses the device by invoking (3) the fastcall, which accesses (4) the MMIO region of the corresponding device and triggers (5) the corresponding IO operation. Fastcalls and the MMIO region are protected against modifications by the application. The kernel is additionally protected against side-channel attacks.

Figure 4
facilities is the per-process fastcall table. Entries in this table are created by the fastcall infrastructure and they contain pointers to fastcall functions, to data regions and configuration parameters which are passed through to fastcall functions. The fastcall system call uses this table to locate the fastcall functions in fastcall space, which is depicted in Figure 5. In this sense, the fastcall table acts a table of capabilities to fastcall functions, just like the file descriptor table contains capabilities to file descriptors.

The fastcall space hosts the stacks for the fastcall functions, which is necessary to write fastcalls in C. To avoid issues with concurrently running fastcalls, the stacks are assigned per CPU and interrupts are kept disabled during the fastcall execution. The fastcall space also hosts the shared memory regions to exchange data with user applications and hosts device memory mappings and fastcall-private pages (e.g. for locks and counters). In general, these facilities enable fastcalls to perform privileged operations without fully entering the kernel, which results in reduced latency compared to system calls.

Fastcalls reduce overhead of system calls by not mitigating side-channel attacks like Meltdown [37]. Meaning that, due to KPTI, the kernel memory is not mapped when fastcalls run. As a result, fastcalls cannot access most of the kernel memory. Instead, we allocate a portion of the process address space to fastcall space. The fastcall space memory cannot be modified directly from the user mode (except shared memory, see Figure 4) and the layout cannot be altered by regular memory-management system calls. All modifications to the fastcall space layout must instead go through the in-kernel fastcall infrastructure. Else, malicious applications could, for example, exchange mapped device memory and trick fastcalls into writing to the wrong device. Therefore, the concept of fastcall space protects fastcalls from malicious applications even though memory is mapped to user space.

Moreover, the fastcall space exhibits a special behavior on forks [29]. Normally, a fork would make a one-to-one copy of memory pages, breaking potentially fastcall data structures, e.g. locks, currently in active use by fastcalls. Such inconsistencies should not occur in kernel mode, therefore we decided to reset the complete fastcall space of children upon forks.

5 Evaluation

We conduct a set of microbenchmarks to compare the latency of fastcall function invocations to alternative approaches: vDSO functions, system calls, and ioctl handlers. The source code of all benchmarks\(^2\) is publicly available online.

We have chosen vDSO [12], because it indicates the lower execution time boundary for kernel-provided functionality. However, vDSO does not allow kernel state modifications, so fastcalls remain a more powerful tool. In contrast, ioctl handlers offer full kernel functionality, but incur the highest overhead among the aforementioned approaches.

All of the evaluated techniques are compared in three different scenarios: the execution of no-operation functions, the copying of small arrays, and the invocation of non-temporal calls. In the first scenario, the called functions directly return back to the calling application. The other two scenarios

---

\(^2\)https://github.com/vilaureu/fastcall-benchmarks

\(^3\)https://github.com/vilaureu/linux/tree/fccmp
Table 1: System Configuration

| Component     | Configuration                      |
|---------------|------------------------------------|
| CPU           | Intel Core i7-4790 CPU at 3.6 GHz  |
| Memory        | 32 GiB with 4 DDR3-1600 DIMMs     |
| Kernel        | based on Linux v5.11 [53]         |
| Distribution  | Debian Testing Bullseye [48]      |
| Settings      | performance CPU governor [57],    |
|               | no HT [23], no Turbo Boost [25]   |

show how the latencies of the tested mechanisms compare when they actually perform some work. In the array-copying scenario the called functions copy a small string of 64 byte into an internal buffer, which could later on be used to manipulate kernel data. Because this scenario is very simple, the copy operations will primarily work on the CPU caches. The non-temporal copy scenario prevents this by writing 64 bytes directly to main memory using the `vmovntdq` [24, p. 1273] vector instruction. The concrete 64-byte value originates from the size of commands in NVMe submission queues [42, p. 65]. This benchmark scenario helps to answer the question of how relevant the lower latency of the fastcall mechanism is when working with PCIe devices. All in all, these microbenchmarks allow to evaluate how the fastcall mechanism compares to the other approaches under different load scenarios.

Finally, as a third variable, we change the mitigation settings for CPU vulnerabilities. All benchmarks are performed once with the default set of mitigations (mitigations=auto [49]), once without KPTI (nopti), and once with all mitigations disabled entirely (mitigations=off). This differentiation is important because countermeasures against Meltdown, Spectre [32], and Microarchitectural Data Sampling (MDS) [52] had a noticeable effect on the system call latency when tested on our benchmark system. Overall, depending on the enabled mitigations, the measured latencies of the functions under test might differ and are worth investigating.

Table 1 lists basic information about our benchmark system, which we tuned for low runtime variance. The benchmarks themselves are regular applications executing in user mode and use the microbenchmark library `benchmark` [15]. This library executes the code of interest in tight loops until the average execution times become statistically stable. We also independently verified that latency variation for all our benchmarks was negligible (IQR of 3.3 ns for an empty system call and 0 for a noop fastcall).

### 5.1 Results

The results of our benchmarks are shown in Figure 6. When comparing all four approaches in executing a no-operation function with mitigations enabled, vDSO and fastcall functions have by far the lowest execution times. The mere execution of the function calls to the vDSO takes only 1.4 ns. Performing ring transitions for executing fastcalls increases the latency to 23.9 ns. Nevertheless, this is still approximately 15 times faster than the 354.7 ns overhead of system calls. For `ioctl` handlers this value is even larger at 413.6 ns. This shows, that the flexibility of `ioctl` handlers comes at the cost of increased latency. In general, the fastcall mechanism allows to avoid a large portion of the overhead of system calls and `ioctl`.

However, this aspect changes when considering the other mitigation settings. The mitigation settings do not affect fastcalls because they skip the kernel entry sequence, which contains the relevant mitigations. This also applies to vDSO functions, which are by design executed in user mode. On the other hand, the mitigation settings heavily influence the system call and `ioctl` approaches. System calls only show 46.4 ns of latency when all mitigations are disabled. Hence, in the second scenario they come close to the 24.1 ns of the fastcall mechanism. This shows the huge cost of mitigations for microsecond-scale systems. Given that security is a necessity in a multi-tenant environment, the rest of the evaluation focuses on the fully-mitigated systems.

The second and the third plot in Figure 6 show the four approaches in more practical scenarios. The fastcall mechanism still holds an advantage under these conditions. For example, fastcall functions are about $\times 12.5$ faster in the array-copy scenario and $\times 3.4$ faster in the non-temporal-copy scenario. This demonstrates that even relative to the latency of main memory accesses, the overhead of the fastcall mechanism is relatively small. This might imply that fastcall functions could even compete with kernel-bypass techniques in regard to overall latency.

### 5.2 Auxiliary Benchmarks

This section discusses some auxiliary benchmarks, which cover latencies of the control path of the fastcall mechanism, the general process flow, and overall system performance. These benchmarks are performed by using the `clock_gettime` vDSO functionality while keeping all default mitigations enabled. The first plot in Figure 7 depicts the latencies involved with registering fastcall functions. This takes 1.4 to 2.6 $\mu$s depending on the number of additional mappings created along with the text segment. The deregistration only takes a couple of microseconds too: 2.4 to 4.8 $\mu$s. Because applications are expected to register fastcall functions very infrequent, the control-path overhead of the new mechanism is likely not adverse in any normal use case.

Another aspect of interest is the influence of the fastcall mechanism on the latency of forks because this approach requires modifications to the `fork` [29] handler of the kernel. More precisely, the modified kernel removes fastcall memory
mappings from the child processes and recreates fresh fastcall tables for them. The latter operation even applies to processes which do not use the fastcall mechanism. This might affect the overall system performance. Figure 7 only shows a slight increase in fork latency between the unchanged, stock kernel and the modified one. The influence on the overall system performance should therefore be minuscule. A more noticeable overhead only arises if the fastcall mechanism is heavily utilized. But even this overhead might be avoidable by using the related vfork system call [6], which does not trigger a reset of the fastcall table. In general, these benchmarks have shown that the control flow of the fastcall mechanism does not introduce much overhead into applications and does not compromise the responsiveness of the system as a whole.

6 Related Work

Previous work improved the traditional POSIX API by accelerating privilege transition, reducing the number of system calls, removing unnecessary memory copies, or improving application wake-up time. In this section we put fastcalls in the context of some of these techniques.

The most straightforward way to improve system call performance is to make the privilege transition faster. Currently the common way to make a system call is the syscall instruction, which replaced a slower software interrupt mechanism [24]. Call gates [24], despite initial performance advantages [16], have been hard to use and did not gain momentum. Moreover, call gates do not offer performance advantages for modern CPUs anymore [34]. Intel recently proposed FRED [22], an extension to the x86-64 architecture. FRED is designed to improve the performance or ring transitions and unify system call and interrupt invocation mechanisms. SkyBridge instead utilizes the vmfunc instruction to transition to another process without the overhead of standard context switches, but in return weakens OS security guarantees [38]. The fastcall layer also could employ any of the aforementioned methods to transition into fastcall space, but we considered our method to be the most practical.

Modern microsecond-scale systems often choose to avoid system calls entirely. Linux’ vDSO mechanism maps system calls directly into the application, eliminating the need for costly privilege transitions [12]. This mechanism offers only few system calls, that can only access non-secret read-only state of the kernel. The fastcall mechanism extends the general concept of vDSO by introducing a way to execute functions in privileged mode, allowing to manipulate kernel state in a secure way.

The io_uring interface [8] utilizes shared memory buffers to speed up the exchange of I/O requests between applications and the kernel. With io_uring, a system call is only required to update the kernel about changes of the queue state, similarly to other system call batching techniques [3, 13, 47].
In a fully asynchronous mode, a kernel thread actively polls on shared memory for new requests, thus eliminating the kernel entry and exit overhead completely. However, polling on both the application and the kernel side leads to high CPU utilization. With fastcalls instead, the application logic and the communication with I/O devices are carried out in the same thread.

To avoid system calls one also could deploy user code directly into the corresponding kernel subsystem. Multiple methods employed eBPF [11] to offload simple request processing routines from the user application into the storage [60], network [14, 20], or scheduling [19] subsystems of the Linux kernel. Additionally, it is possible to offload the request processing routines to the device [9, 18, 46], reducing data movement and CPU load. These architecture improve request response latency, but do not address the overhead of creating new requests. In this regard, fastcalls are complementary to the aforementioned offloading techniques, because they enable low-latency user-initiated interactions with the kernel.

An application can get maximum performance by getting direct access to the underlying device, bypassing the kernel. In contrast to traditional OS architectures, kernel bypassing techniques employ special APIs to avoid privilege transitions and data copying between user and kernel space. For issuing device commands, user applications write to the corresponding device registers directly. Latency of event delivery is minimized, because user applications poll on the device completion queues, instead of waiting in a blocking system call (e.g. epoll). Such approaches exist for networking [21, 35, 45] as well as for storage devices [31, 58]. However, removing the kernel from the data path often requires special support from the device [21, 39] and results in security and manageability concerns, because the OS loses the control over how the device is used. We hope that the introduction of the fastcall space into the kernel-bypass architectures may bring the OS back into the picture, without adding any significant overhead.

The concept of kernel bypass has become a first-class citizen in dataplane [4, 43, 59] and microkernel-based [2, 10] operating systems. In addition to passing the device under the control of a user process, these operating systems focus on convenient programming abstractions to reduce development effort, improve portability, and enable resource sharing. We see fastcalls as an instrument for replacing kernel bypass and for enabling low-overhead kernel interposition with the purpose of passing the device to the user application in controllable manner.

Being the most popular, the dataplane OS is not the only architecture trying to revamp system calls. Singularity [33] and RedLeaf [40] avoid CPU-enforced address space isolation by relying on language guarantees and runtime checks. Lee et al. employ call-gates to implement a new privileged user level for processing sensitive data using x86 privilege rings [34], whereas Vasudevan et al. employ a new low-latency protection domain switching mechanism to remove the kernel almost entirely [56]. Unfortunately, the latter mechanism comes with probabilistic security guarantees.

7 Discussion

We believe that the extremely low overhead that fastcalls offer may bring qualitative changes to the future OS architectures. The kernel may offload fastpath operations automatically into the fastcall space to improve common case performance. In a way such offloading will take the opposite direction of how eBPF is used nowadays. In our vision, instead of having a 30-instruction receive routine of a TCP packet [55], a fastcall will contain a similarly small send routine. The remainder of this section describes our vision of fastcalls in more detail.

7.1 Fastcall model

To understand the decisions driving the fastcall design, we outline how we reason about them. A fastcall is designed to improve performance in comparison to system calls in exchange for a more limited usage model. This means that if overhead of a long-running system call is negligible, implementing the same functionality in fastcall will not be useful. Hence, a fastcall must not run longer than it is defined by the overhead introduced by a system call.

To get a more specific estimation of how long a fastcall can run, assume any overhead less than \( O = 5\% \) of the total operation to be negligible, system call overhead to be \( o_s = 300\text{ns} \), and fastcall overhead to be \( o_f = 30\text{ns} \). The total runtime of operation is \( T = w + o \), where \( w \) is the useful work performed by the operation, and \( o \) is the overhead. Assuming the system call and fastcall implementations are equivalent, \( w \) component will remain the same. Then the fastcall is beneficial only if:

\[
\frac{o_s}{T} = \frac{o_s}{o_s + w} > O
\]

Solving for \( w \), we get that the useful work must not take more than 5.7\( \mu \)s. If we additionally require the fastcall overhead to be negligible, the useful work should take at least 570\text{ns}. These limits includes all the work required to complete a privileged operation, including the preparatory work happening outside of fastcall or kernel space. The lower limit is not a hard limit, but rather an indication that another solution, possibly a hardware-based one, can offer tangible benefits, if the overhead can be kept even lower.

To put these numbers in context, a high-end Infiniband NIC provides as low as 600\text{ns} back-to-back latency [41]. With multiple switches in between, one can expect several microseconds latency in real world applications. These numbers match well our expectations about fitting fastcall use
case. We observe similar situation with modern NVMe storage.

From the design point of view having a limit on the maximum execution time opens up many interesting opportunities. For example, we envision that in the future, fastcalls will be automatically verified to contain no bugs or known vulnerabilities before mapping them to the application. The maximum execution time of several microseconds may be sufficiently short to even employ techniques like symbolic execution [7] during the verification process.

7.2 Fastcalls Are for Fast Paths

Fastcalls have several implementation limitations that follow from our design philosophy. Specifically, a fastcall has the following limitations: It must not generate exceptions, it must not access arbitrary user or kernel memory, and it must not contain secrets.

Exceptions, like a page fault, would be detrimental for the fastcall performance and render them useless. If an exception is to be expected, one should simply make a conventional system call, and get access to full kernel functionality. Having this property as a design limitation, we plan to conduct a worst case execution time analysis for fastcalls in the future. Such analysis will simplify tail latency optimization at the OS level.

A corollary of the previous statement is that fastcalls cannot access user memory, because such an access may generate a page fault. We work around this limitation as fastcalls exchange data with the user over shared regions of pinned memory.

Finally, because we omit side channel mitigations, when entering fastcall space, fastcall space must not contain confidential information. This limitation may be reconsidered for CPU architectures which offer reliable protection of sensitive information.

8 Conclusion

In this paper, we explored the construction of a framework for lightweight system calls, which we call fastcalls. Fastcalls enable the operating system to protect security-critical resources like kernel or device memory from arbitrary access by applications, while having performance properties close to that of normal function calls. In particular, the fastcall approach improves on the latency of standard system calls by up to 15 ×, while keeping protection mechanisms against side-channel attacks in effect. 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 virtualization of devices as well as to give applications protected access to privileged CPU instructions. In the future, we plan to investigate the benefit of fastcalls for the virtualization of RDMA networks and NVMe-attached SSDs.

References

[1] networking:toe [Wiki]. URL https://wiki.linuxfoundation.org/networking/toe.

[2] L4Re – The L4 Runtime Environment. URL https://l4re.org/.

[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] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, pages 316–331. ACM. ISBN 978-1-4503-4911-6. doi:10/gnk2n3.

[6] Andries Brouwer and Michael Kerrisk. vfork(2). URL https://man7.org/linux/man-pages/man2/vfork.2.html.

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. page 16.

[8] Jonathan Corbet. Ringing in a new asynchronous i/o API. URL https://lwn.net/Articles/776703/.

[9] 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.

[10] 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.

[11] Matt Fleming. A thorough introduction to eBPF. URL https://lwn.net/Articles/740157/.

[12] Mike Frysinger. vdso(7). URL https://man7.org/linux/man-pages/man7/vdso.7.html.

[13] Luis Gerhorst, Benedict Herzog, Stefan Reif, Wolfgang Schröder-Peikschat, and Timo Hönic. 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.

[14] 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.

[15] Google Inc. benchmark. URL https://github.com/google/benchmark.

[16] 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.

[17] 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.

[18] Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ryan E. Grant, and Ron Brightwell. sPIN: high-performance streaming processing in the network. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC ’17, pages 1–16. ACM Press. ISBN 978-1-4503-5114-0. doi:10.1145/3126908.3126970.

[19] 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.

[20] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The eXpress data path: fast programmable packet processing in the operating system kernel. In Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, pages 54–66. ACM. ISBN 978-1-4503-6080-7. doi:10.1145/3281411.3281443. URL https://dl.acm.org/doi/10.1145/3281411.3281443.

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

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

[23] Intel Corporation. What is hyper-threading?. . URL https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html.

[24] Intel Corporation. Intel® 64 and IA-32 architectures software developer’s manual, combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d and 4. . URL https://software.intel.com/content/dam/develop/public/us/en/documents/325462-sdm-vol-1-2abcd-3abcd.pdf.

[25] Intel Corporation. What is intel® turbo boost technology?. . URL https://www.intel.com/content/www/us/en/gaming/resources/turbo-boost.html.
[26] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data Movement Is All You Need: A Case Study on Optimizing Transformers. page 22.

[27] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sungwhan Ihm, Dongsu Han, and YoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI’14, pages 489–502. USENIX Association. ISBN 978-1-931971-09-6.

[28] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeong 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: 10.1145/2719987.

[29] Michael Kerrisk and Drew Eckhardt. fork(2). URL https://man7.org/linux/man-pages/man2/fork.2.html.

[30] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitul Padhye, Shachar Iwin, 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.

[31] 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.

[32] 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).

[33] 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.

[34] Hojoon Kang, 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.

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

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

[37] 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).

[38] 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.

[39] Mike Jackson and Ravi Budruk. PCI Express Technology 3.0.

[40] 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.

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

[42] NVM Express, Inc. NVM express base specification, revision 1.4b. URL https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4b-2020.09.21-Ratified.pdf.

[43] 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.
[44] Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoeffer, and Hermann Härting. 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.

[45] 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.

[46] 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.

[47] 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.

[48] Software in the Public Interest, Inc. Debian "bullseye" release information. URL https://www.debian.org/releases/bullseye/index.en.html.

[49] The kernel development community. The kernel’s command-line parameters. URL https://www.kernel.org/doc/html/v5.11/admin-guide/kernel-parameters.html.

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

[51] The kernel development community. Memory management. URL https://www.kernel.org/doc/html/v5.11/x86/x86_64/mm.html.

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

[53] Linus Torvalds. Linux kernel source tree v5.11. URL https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/?h=v5.11.

[54] Linus Torvalds, Andi Kleen, and Pavel Machek. 64-bit SYSCALL instruction entry. URL https://elixir.bootlin.com/linux/v5.11/source/arch/x86/entry/entry_64.S.

[55] Van Jacobson. Fwd: TCP in 30 instructions (Was Re: Karl Auerbach: Re: Storage over Eth. URL https://www.pdl.cmu.edu/mailinglists/ips/mail/msg00133.html.

[56] 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.

[57] Rafael J. Wysocki. CPU performance scaling. URL https://www.kernel.org/doc/html/v5.11/admin-guide/pm/cpufreq.html.

[58] 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.

[59] 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.

[60] 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.