结合协程调度的中断控制器设计与应用

姓名 赵方亮　　指导老师 向勇　　院系 计算机系

一、 选题背景及其意义

在当今时代，云应用程序（例如网络搜索、社交网络、电子商务等）在日常生活中发挥着越来越更重要的作用。为了保证良好的人机交互体验，这些应用程序需要在几十毫秒的时间尺度上对用户的操作作出响应，而它们通常将单个请求分散到数据中心的数百台计算机上运行的数千个通信服务。其中耗时最长的服务成为了影响端到端响应时间的主要因素，这要求每个参与的服务的尾部延迟在几百微秒的范围内，且这些需求将会逐渐变得更加苛刻。因此，如何构建出低时延的服务已经成为一个重要的研究领域。

近年来，为了提高计算效率，许多现代编程语言（C++、Go、Javascript、Python 和 Rust 等）先后引入了协程的概念，提供了异步函数的抽象，允许程序员通过关键字创建异步函数，通过允许异步函数的执行重叠来更好的利用计算资源并提高程序执行效率。与传统的线程相比，协程更加轻量，具有非常小的上下文切换开销，并且它非常适合 I/O 栈通常需要的基于状态机的异步事件处理机制。然而，协程只能以协作的方式进行调度，下一个就绪的协程只能等待上一个协程运行结束才能够运行，这不利于构建低时延服务。

与协程需要的协作式调度相反，中断作为一种 CPU 抢占机制，通过软件和硬件之间相互配合，能够打断 CPU 当前的正在执行的任务，转而执行优先级更高的任务。因此当某项任务长期占用 CPU 时，可以通过中断让 CPU 能够处理其他需要快速响应的任务，有利于构建低时延服务。但是，中断会产生许多直接开销（寄存器保存/恢复、调度器、以及处理器流水线刷新等）和间接开销（缓存丢失等），这些开销随着高速设备和高速网络的发展而逐渐变得不能接受。尽管目前有很多先进的技术用轮询来消除中断产生的开销，从而也能实现低时延的目的，但代价却是功耗和 CPU 占用非常高。

结合上述分析，将中断机制与协程调度结合起来，充分利用两者的优点，设计出以协程为单位的结合中断机制的任务调度器对于构建低时延服务具有非常重要的意义。一方面能够利用协程上下文切换开销小的特点，减小目前主流的先进技术使用线程作为任务单元的任务切换开销。另一方面，结合了中断机制的任务调度器能够高效的进行任务调度，在一定程度上消除由于中断抢占带来的开销。

二、 国内外研究现状

1. 调度策略

Wierman 和 Zwart 在他们 2012 年发表的论文中证明，没有单一的调度策略可以最小化所有可能工作负载的尾部延迟 [20]。在任务的尾部延迟较小时，first come first served (FCFS) 策略的尾部延迟是渐进最优的；在任务的尾部延迟较大时，则 processor sharing (PS) 策略的尾部延迟是渐进最优的。并且这些策略体现出明显的二分性，即在轻尾工作负载下表现良好的策略在重尾负载下表现较差，反之亦然。在此基础上，Prekas、Kogias 和 Bugnion 等人在开展 ZygOS 的研究工作时，对

表 1: 低时延服务研究

研究工作	调度策略	中断	其他延迟优化手段
IX	D-FCFS[1]	*[3]	Kernel-bypass networking, Batched Syscalls
ZygOS	D-FCFS + Work stealing	*	Kernel-bypass networking, Batched Syscalls
Shenango	D-FCFS + Work stealing	×	Kernel-bypass networking, Lightweight Threading
Shinjuku	C-FCFS[2]	√[4]	Kernel-bypass networking, Fast Preemption
Concord	C-FCFS	×[5]	Kernel-bypass networking, Compiler-Enforced Cooperation
Demikernel	C-FCFS	×	Kernel-bypass networking, Coroutine, zero-copy

[1] D-FCFS: 分布式队列（多队列），FCFS 调度策略。
[2] C-FCFS: 中心化队列（单队列），FCFS 调度策略。
[3] *: 使用中断进行资源控制。
[4] √: 使用中断进行任务抢占。
[5] ×: 没有中断。

单队列和多队列进行了分析，证明了无论是 FCFS 还是 PS 调度策略，单队列的尾部延迟均优于多队列 [15]。近年来，有很多研究工作围绕着构建低时延服务进行。在上述的理论前提下，他们结合了操作系统本身的一些优化手段，成功构建了一些低时延服务 [1, 9, 10, 14, 15, 22]。

IX[1] 给每个线程分配固定的网卡队列实现了多队列，并且通过批处理系统调用以及零拷贝技术实现了较低的服务延迟，但由于多队列造成了负载不均衡、以及队头阻塞问题，导致在负载较重的情况下，延迟较高。ZysOS[15] 针对负载不均衡的问题，在 IX 的基础上增加了工作窃取模块用以保证负载均衡和解决队头阻塞问题，但这些优化均需要保持固定数量的处理器核心用于满足突发高负载情况下的低时延要求，导致了 CPU 使用效率不高。Shenango[14] 以非常精细的时间尺度在应用程序之间重新分配处理器核心，使得在服务延迟相当的情况下能够取得更高的 CPU 效率。以上这些研究工作采用多队列，通过工作窃取等手段保证负载均衡，但工作窃取同样存在开销，尤其对缓存不友好。Shinjuku[10]、Concord[9] 则使用一个专用调度线程来轮询单队列，并且由这个调度线程将网络请求均衡的发送给各个工作线程使用的局部队列中保证了负载均衡。Demikernel[22] 同样采用了 C-FCFS 调度策略。

2. 中断

中断作为一种提高计算机工作效率的技术，最初是用于解决处理器忙等外部设备导致的效率低下问题 [8]。在构建低时延服务时，使用中断能够帮助解决队头阻塞问题，让短作业能够及时抢占 CPU，不需要等到长作业运行结束。在负载较重时，使用中断实现的 PS 调度策略能让尾部延迟达到渐进最优。

IX[1] 在控制平面使用时钟中断来提供重新获取 CPU 控制权的手段，而 ZygOS[15] 则使用核间中断（IPI）来保证在同一个核上发送和接收来自给定流的数据包，从而保证了较好的缓存亲和性。这两者都没有将中断直接用于任务抢占，Shinjuku[10] 利用了 Intel 用户态中断技术 [21]，调度线程直接发送 IPI 给处于其他核上的工作线程，实现了快速任务抢占，不需要切换特权级。Concord[9]

则使用其他的手段达到了近似于 Shinjuku 的低尾部延迟效果，它通过编译器插桩进行强制协作，定期检查缓存行中的标志从而使得 CPU 能及时让权给短作业，减小了由于中断抢占带来的开销。

随着许多高速设备出现，中断系统越来越复杂，且中断发生的频率很高，容易发生中断丢失或中断过载的现象，导致一些服务受到影响 [16]。Linux 系统将中断处理分为上、下半段，在上半段完成简单的、有时限的处理工作，保证在下次中断产生之前将上一次中断处理完毕。尽管在软件中处理中断能够提供灵活性，但其开销逐渐变得不能接受。Demikernel[22] 以及一些其他的在用户态实现的网络协议栈则通过轮询的方式来达到低时延的目的，消除了中断带来的开销。此外，也出现了许多硬件处理中断的方案。

Humphries 和 Kaffes 等人提出一种新的模型，使用专用的处理器核心处理中断 [7]。Erwin 和 Jensen 使用 queued CAM（content-addressable memories）硬件来缓存中断，减少中断丢失 [5]。Regehr 和 Duongsaa 使用硬件中断调度器来降低给 CPU 发送中断信号的频率，从而防止中断过载 [16]。但这些方案中，实际的中断处理还是由 CPU 来完成。而 PCP[17] 则真正在硬件上实现了中断处理程序。当 PCP 收到中断信号时，它将根据中断信号，激活在内存中的阻塞任务，并修改其状态，这个处理过程不会干扰 CPU 上正在执行的任务，达到了并行的效果。但由于 PCP 和 CPU 同时访问内存中的任务队列，需要额外的机制保证互斥访问，尽管最终实现了防止由于中断带来的优先级反置现象，但也导致了中断处理延迟比原本的软件中断处理高。

PCP 将中断处理从 CPU 上卸载下来，但由于中断处理过程与 CPU 中的任务调度过程相互耦合，存在同时访问任务队列的情况，导致没有达到理想的效果。近年来，很多研究工作围绕着在硬件在中维护优先级队列展开 [2, 11, 13]。这些硬件优先级队列能够在几个时钟周期内完成一次入队/出队操作，并且具有较好的扩展性，能够满足任务调度的需求。

目前较先进的低时延服务技术大多数建立在传统的线程模型上。近年来，以协程为任务单元的研究开始引起学术界和工业界的关注，Demikernel[22] 使用协程作任务单元，能够在十几个时钟周期内完成任务切换；基于 Rust 协程实现的为嵌入式设备上运行的异步驱动生成框架 Embassy[4] 在处理设备中断方面取得了令人激动的效果，在中断耗时和中断延迟方面远胜于用 C 语言实现的 FreeRTOS[3]。由于协程具有异步执行的特点，它与中断这种异步通知机制具有天然的联系。然而，由于协程极其轻量的特点，将协程调度与目前的软件中断处理方式结合起来，存在着一定的不适配性。在软件中断处理程序中唤醒被阻塞协程的开销与中断带来的上下文切换和缓存缺失开销相比，显得微不足道 [19]。经过调研，目前很少有研究将中断处理与协程调度同时从 CPU 上卸载下来，且使用硬件完成中断处理与任务调度是具有可行性的。

三、 主要研究内容

1. 基于优先级的协程调度机制

基于目前对 Rust 协程的应用以及低时延服务的发展现状来看，将 Rust 协程用于构建低时延服务是可行的。尽管 Rust 语言为协程以及异步编程提供了良好的支持 [6]，但 Rust 协程必须通过运行时的调度才能运行，因此需要高效的运行时是研究工作的第一项内容。

一方面，仅仅依靠 Rust 语言提供的支持无法达到精准控制协程的目的，只能使用粗暴的轮询方式推动协程执行，无法与异步 I/O 机制结合起来，发挥出协程的优势。因此，需要在协程控制块中增加额外三个控制字段：1）cid 字段用于标识协程控制块；2）ctype 字段指示协程所属的任务类型，在协程无法继续推进时，运行时将根据协程的类型进行不同的处理；3）priority 字段表示任务的优先级顺序，用于保证在每次调度时，让优先级最高的协程能够最先运行。我们通过这些附加控

制字段，实现对协程的精确控制。此外，我们借鉴了 Concord 的思路，假设协程每个阶段运行的时间较短，不存在某个协程长期占据 CPU 的情况，一旦协程无法继续推进，运行时将调度下一个优先级最高的就绪协程。因此，发生协程调度的频率将非常高，借助优先级机制，我们可以通过协作式调度达到近似于抢占式调度的效果。

另一方面，协程通常在用户态使用，操作系统内核无法感知用户态协程。为此，我们将就绪协程的优先级队列维护在硬件中，操作系统通过访问硬件接口，能够间接感知到用户态协程，从而保证内核与用户进程之间的协调调度。并且由于硬件维护的优先级队列能够在几个时钟周期内完成一次入队/出队操作，协程调度需要的开销也将被大幅度缩减。

2. 结合中断处理的协程唤醒机制

协程具有异步的特性，能与基于状态机的异步事件处理机制紧密结合，与中断这种异步通知机制具有天然的联系。使用协程处理 I/O 请求，若协程需要等待设备处理结束才能继续运行，那么协程将把 CPU 让给其他的就绪协程，一旦设备处理完毕后，向 CPU 发起中断，CPU 则在中断处理程序中唤醒相关的被阻塞协程。然而中断与协程之间存在这一定的不适配性。由于中断是一种抢占方式，它能够在任意指令后强制转移控制流，需要保存和恢复大量的寄存器。并且第一项研究工作 $1. 的基础上，在中断处理程序中唤醒被阻塞协程需要的开销将大幅缩减，与中断造成的上下文切换开销以及缓存缺失开销等相比，微不足道。为了避免由于中断带来的开销，部分研究工作使用专门的线程/协程轮询设备，由该线程/协程唤醒其他的协程，尽管这种方式消除了中断带来的开销，但这种方式导致 CPU 占用率和功耗较高。因此，将中断处理与协程唤醒[1]紧密结合是研究工作中的第二项内容。

由于第一项研究工作 $1. 在将就绪协程的优先级队列维护在了硬件中，因此，由硬件并行处理中断且唤醒阻塞的协程是可行的。针对各种外部设备，我们将在硬件中维护对应的阻塞协程队列，一旦某个协程由于相应的外部设备而阻塞时，它将被运行时写入到硬件中维护的相应阻塞队列中。一旦硬件收到来自外部设备的中断信号，它将根据中断向量号从对应的阻塞队列中获取到某个阻塞协程，并根据优先级将其添加至相应的就绪协程队列中，等待运行时的调度。以此种方式，将中断处理与协程唤醒从 CPU 上卸载下来。

3. 基于协程调度与中断处理的 IPC 加速机制

第三项研究内容受到第二项研究工作 $2. 的启发，既然可以使用硬件将外部中断的处理与协程唤醒结合起来，那么同样可以使用硬件加速使用 IPI 机制实现的 IPC 机制。

一方面是对系统调用的优化。由于熔断漏洞和幽灵漏洞的影响，内核页表隔离机制（Kernel page-table isolation，KPTI）被提出，内核拥有独立的地址空间，可被视为一个“进程”。用户进程每次执行系统调用都需要切换到独立的内核地址空间中，导致系统调用的开销增加。因此，我们将系统调用视为一种特殊的 IPC，它的接收方固定为内核，发送方为用户进程，由发送方通过 ecall 指令发起。已有很多研究工作针对系统调用进行了优化 [12, 18, 23]。我们借鉴了这些研究工作，尤其是 FlexSC[18]。我们在硬件中维护了一个系统调用处理协程阻塞队列，内核在初始化时，向其中添加阻塞的系统调用处理协程，并在内存中申请一块用于存放系统调用消息的空间。内核系统调用处理协程通过访问硬件接口从内存中获取需要处理的系统调用消息。而用户进程内的系统调用协程则通过操作硬件接口，生成系统调用消息，并由硬件唤醒对应的内核系统调用处理协程。一旦内核系统调用处理协程完毕，则通过操作硬件接口，唤醒对应阻塞的用户进程系统调用协程。

[1] 协程唤醒：将被阻塞的协程添加至就绪任务队列中。

表 2: 研究成果

研究成果	具体形式	特征
结合任务调度的新型中断控制器	ATS-INTC[1]	加速任务调度；并行中断处理
异步操作系统	异步 Arceos（unikernel、宏内核）	异步驱动；并行异步批处理系统调用；加速基于 IPI 实现的 IPC

[1] ATS-INTC: Asynchronous task scheduler and interrupt controller。

另一方面是对其他基于 IPI 机制实现的 IPC 的优化。与系统调用不同，这种 IPC 的收发双方均为用户进程，由发送方通过 send_ipi 指令发起 [21]。这种方式远远优于 Linux 系统中的信号机制，我们借鉴了其研究成果，并针对其接收方的中断处理进行优化。发送方通过操作硬件接口向接收方发起 IPC，硬件将根据 IPC 消息的内容唤醒接收方对应的处理协程，从而消除了接收方中断的开销，完成加速。

四、 预期成果和创新点

在借鉴前人研究成果以及对协程调度、协程唤醒与中断处理机制的分析基础上，将协程调度机制与中断处理机制相结合，从 CPU 卸载到并行的硬件设备上，弥补协作式调度高响应延迟的缺陷，得到近似于抢占式调度的响应时延，并且利用协程减小上下文切换的开销的特点，以及 IPC 加速功能构建出微秒级的低时延服务。预计研究成果如表2所示。

五、 工作计划

工作的总体时间安排如下：

1. 2023 年 11 月至 2024 年 1 月，查找相关资料，了解国内外的发展动态，细化设计方案，理清实施思路，完成开题工作。
2. 2024 年 1 月至 2024 年 2 月，在 QEMU 模拟器中实现基于优先级的协程调度器。
3. 2024 年 2 月至 2024 年 3 月，在 QEMU 模拟器中实现结合了协程调度的中断控制器（只考虑网卡中断处理）。
4. 2024 年 3 月至 2024 年 5 月，在 FPGA 中实现上一阶段的结合协程调度的中断控制器。在 Arceos unikernel 环境下，测试网卡的性能，并改造 redis 等上层应用，完成相关的性能对比测试，争取发表一篇论文。
5. 2024 年 5 月至 2024 年 6 月，在第三阶段3基础上，将系统调用进行并行异步批处理优化。
6. 2024 年 6 月至 2024 年 8 月，将上一阶段成果移植至 FPGA 中，并基于 Arceos 宏内核 Starry，对上层应用进行改造，完成系统调用性能对比测试，争取发表一篇论文。
7. 2024 年 8 月至 2024 年 9 月，在第五阶段5基础上，将基于 IPI 的 IPC 进行优化。
8. 2024 年 9 月至 2024 年 11 月，将上一阶段成果移植至 FPGA 中，并基于 Arceos 宏内核 Starry，对上层应用进行改造，完成 IPC 性能对比测试，争取发表一篇论文。
9. 2024 年 11 月至 2025 年 1 月，完成学位论文写作。

References

[1] A. Belay et al. “IX: A Protected Dataplane Operating System for High Throughput and Low Latency”. In: USENIX Symposium on Operating Systems Design and Implementation. Oct. 6, 2014. URL: https://www.semanticscholar.org/paper/IX%3A-A-Protected-Dataplane-Operating-System-for-High-Belay-Prekas/567f633bc74ac40364a1961fad1b7fe80605b815.

[2] Imad Benacer et al. “A fast systolic priority queue architecture for a flow-based Traffic Manager”. In: *2016 14th IEEE International New Circuits and Systems Conference (NEWCAS)*. 2016 14th IEEE International New Circuits and Systems Conference (NEWCAS). June 2016, pp. 1–4. DOI: 10.1109/NEWCAS.2016.7604761. URL: https://ieeexplore.ieee.org/document/7604761.

[3] Dion. *Async Rust vs RTOS showdown! - Blog - Tweede golf*. https://tweedegolf.nl/en/blog/65/async-rust-vs-rtos-showdown. Jan. 31, 2022.

[4] *embassy-rs/embassy: Modern embedded framework, using Rust and async*. https://github.com/embassy-rs/embassy. 2023.

[5] Jerry D. Erwin and E. Douglas Jensen. “Interrupt processing with queued content-addressable memories”. In: *Proceedings of the November 17-19, 1970, fall joint computer conference*. AFIPS ’70 (Fall). New York, NY, USA: Association for Computing Machinery, Nov. 17, 1970, pp. 621–627. ISBN: 978-1-4503-7904-5. DOI: 10.1145/1478462.1478553. URL: https://dl.acm.org/doi/10.1145/1478462.1478553.

[6] *Getting Started - Asynchronous Programming in Rust*. https://rust-lang.github.io/async-book/.

[7] Jack Tigar Humphries et al. “A case against (most) context switches”. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS ’21: Workshop on Hot Topics in Operating Systems. Ann Arbor Michigan: ACM, June 2021, pp. 17–25. ISBN: 978-1-4503-8438-4. DOI: 10.1145/3458336.3465274. URL: https://dl.acm.org/doi/10.1145/3458336.3465274 (visited on 08/27/2023).

[8] *interrupt*. https://en.wikipedia.org/wiki/interrupt.

[9] Rishabh Iyer et al. “Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling”. In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP ’23. New York, NY, USA: Association for Computing Machinery, Oct. 23, 2023, pp. 466–481. ISBN: 9798400702297. DOI: 10.1145/3600006.3613136. URL: https://dl.acm.org/doi/10.1145/3600006.3613136.

[10] Kostis Kaffes et al. “Shinjuku: Preemptive Scheduling for second-scale Tail Latency”. In: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). 2019, pp. 345–360. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/conference/nsdi19/presentation/kaffes.

[11] Lukáš Kohútka. “Efficiency of Priority Queue Architectures in FPGA”. In: *Journal of Low Power Electronics and Applications* 12.3 (Sept. 2022). Number: 3 Publisher: Multidisciplinary Digital Publishing Institute, p. 39. ISSN: 2079-9268. DOI: 10.3390/jlpea12030039. URL: https://www.mdpi.com/2079-9268/12/3/39.

[12] Dmitry Kuznetsov and Adam Morrison. “Privbox: Faster System Calls Through Sandboxed Privileged Execution”. In: 2022 USENIX Annual Technical Conference (USENIX ATC 22). July 11, 2022. ISBN: 978-1-939133-29-8. URL: https://www.usenix.org/conference/atc22/presentation/kuznetsov.

[13] Sung-Whan Moon, J. Rexford, and K.G. Shin. “Scalable hardware priority queue architectures for high-speed packet switches”. In: *IEEE Transactions on Computers* 49.11 (Nov. 2000). Conference Name: IEEE Transactions on Computers, pp. 1215–1227. ISSN: 1557-9956. DOI: 10.1109/12.895938. URL: https://ieeexplore.ieee.org/abstract/document/895938.

[14] Amy Ousterhout et al. “Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads”. In: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). 2019, pp. 361–378. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/conference/nsdi19/presentation/ousterhout.

[15] George Prekas, Marios Kogias, and Edouard Bugnion. “ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17: ACM SIGOPS 26th Symposium on Operating Systems Principles. Shanghai China: ACM, Oct. 14, 2017, pp. 325–341. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132780. URL: https://dl.acm.org/doi/10.1145/3132747.3132780.

[16] John Regehr and Usit Duongsaa. “Preventing interrupt overload”. In: *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. LCTES ’05. New York, NY, USA: Association for Computing Machinery, June 15, 2005, pp. 50–58. ISBN: 978-1-59593-018-7. DOI: 10.1145/1065910.1065918. URL: https://dl.acm.org/doi/10.1145/1065910.1065918.

[17] Fabian Scheler et al. “Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system”. In: *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*. CASES ’09. New York, NY, USA: Association for Computing Machinery, Oct. 11, 2009, pp. 167–174. ISBN: 978-1-60558-626-7. DOI: 10.1145/1629395.1629419. URL: https://dl.acm.org/doi/10.1145/1629395.1629419.

[18] 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. USA: USENIX Association, Oct. 4, 2010, pp. 33–46.

[19] Dan Tsafrir. “The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)”. In: *Proceedings of the 2007 workshop on Experimental computer science*. ExpCS ’07. New York, NY, USA: Association for Computing Machinery, June 13, 2007, 4–es. ISBN: 978-1-59593-751-3. DOI: 10.1145/1281700.1281704. URL: https://dl.acm.org/doi/10.1145/1281700.1281704.

[20] Adam Wierman and Bert Zwart. “Is Tail-Optimal Scheduling Possible?” In: *Operations Research* 60.5 (Sept. 1, 2012), pp. 1249–1257. ISSN: 0030-364X. DOI: 10.1287/opre.1120.1086. URL: https://doi.org/10.1287/opre.1120.1086.

[21] *x86 User Interrupts support [LWN.net]*. https://lwn.net/Articles/869140/.

[22] Irene Zhang et al. “The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems”. In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP ’21. New York, NY, USA: Association for Computing Machinery, Oct. 26, 2021, pp. 195–211. ISBN: 978-1-4503-8709-5. DOI: 10.1145/3477132.3483569. URL: https://dl.acm.org/doi/10.1145/3477132.3483569.

[23] Zhe Zhou et al. “Userspace Bypass: Accelerating Syscall-intensive Applications”. In: 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). 2023, pp. 33–49. ISBN: 978-1-939133-34-2. URL: https://www.usenix.org/conference/osdi23/presentation/zhou-zhe.