ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling

Jack Tigar Humphries[1], Neel Natu[1], Ashwin Chaugule[1], Ofir Weisse[1], Barret Rhoden[1], Josh Don[1], Luigi Rizzo[1], Oleg Rombakh[1], Paul Turner[1], Christos Kozyrakis[2]
[1] Google, Inc. [2] Stanford University

Abstract

We present ghOSt, our infrastructure for delegating kernel scheduling decisions to userspace code. ghOSt is designed to support the rapidly evolving needs of our data center workloads and platforms.

Improving scheduling decisions can drastically improve the throughput, tail latency, scalability, and security of important workloads. However, kernel schedulers are difficult to implement, test, and deploy efficiently across a large fleet. Recent research suggests bespoke scheduling policies, within custom data plane operating systems, can provide compelling performance results in a data center setting. However, these gains have proved difficult to realize as it is impractical to deploy a custom OS image(s) at an application granularity, particularly in a multi-tenant environment, limiting the practical applications of these new techniques.

ghOSt provides general-purpose delegation of scheduling policies to userspace processes in a Linux environment. ghOSt provides state encapsulation, communication, and action mechanisms that allow complex expression of scheduling policies within a userspace agent, while assisting in synchronization. Programmers use any language to develop and optimize policies, which are modified without a host reboot. ghOSt supports a wide range of scheduling models, from per-CPU to centralized, run-to-completion to preemptive, and incurs low overheads for scheduling actions. We demonstrate ghOSt's performance on both academic and real-world workloads, including Google Snap and Google Search. We show that by using ghOSt instead of the kernel scheduler, we can quickly achieve comparable throughput and latency while enabling policy optimization, non-disruptive upgrades, and fault isolation for our data center workloads. We open-source our implementation to enable future research and development based on ghOSt.

CCS Concepts • **Software and its engineering**;

Keywords Operating systems, thread scheduling

ACM Reference Format:
Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–28, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3477132.3483542

ACM ISBN 978-1-4503-8709-5/21/10.
https://doi.org/10.1145/3477132.3483542

1 Introduction

CPU scheduling plays an important role in application performance and security. Tailoring policies for specific workload types can substantially improve key metrics such as latency, throughput, hard/soft real-time characteristics, energy efficiency, cache interference, and security [1–26]. For example, the Shinjuku request scheduler [25] optimized highly dispersive workloads – workloads with a mix of short and long requests – improving request tail latency and throughput by an order of magnitude. The Tableau scheduler for virtual machine workloads [23] demonstrated improved throughput by 1.6× and latency by 17× under multi-tenant scenarios. The Caladan scheduler [21] focused on resource interference between foreground low-latency apps and background best effort apps, improving network request tail latency by as much as 11,000×. To mitigate recent hardware vulnerabilities [27–32], cloud platforms running multi-tenant hosts had to rapidly deploy new core-isolation policies, isolating shared processor state between applications.

它这里举了很多针对特定workload设计调度算法的例子，也许可以用在我们的扩展题里面！

Designing, implementing, and deploying new scheduling policies across a large fleet is an exacting task. It requires developers to design policies capable of balancing the specific performance requirements of many applications. The implementation must conform with a complex kernel architecture, and errors will, in many cases, crash the entire system or otherwise severely impede performance due to unintended side effects [33]. Even when successful, the disruptive nature of an upgrade carries its own opportunity cost in host and application downtime. This creates a challenging conflict between risk-minimization and progress.

这段的意思就是比如说CFS这样的通用框架要考虑的性能均衡太多了，隐含着点它不够适合特定场景的意思

Prior attempts to improve performance and reduce complexity in the kernel by designing userspace solutions have significant shortcomings: they require substantial modification of application implementation [1, 10, 21, 25, 34, 35],

以前试图通过设计用户空间解决方案来提高性能和降低内核复杂性的尝试有明显的缺点：
1.需要修改用户程序实现
2.为了高度响应，需要耗费
3.需要大量修改内核

这个分界线的实现，应该就是ebpf的功劳所在了

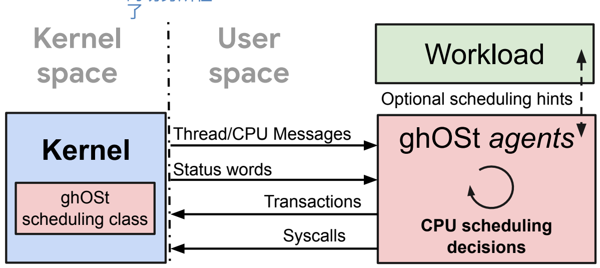


Figure 1. Overview of ghOSt.

require dedicated resources in order to be highly responsive [1, 21, 25], or otherwise require application-specific kernel modifications [1, 2, 10, 21, 25, 34–37].

Although Linux supports multiple scheduling implementations; tailoring, maintaining, and deploying a different mechanism for each application is impractical to manage on a large fleet. One of our goals is to formalize which kernel changes are required to enable a plethora of optimizations (and experimentation) in userspace, on a *stable* kernel ABI.

我们会努力最小化内核的修改

Further, the hardware landscape is ever changing, stressing existing scheduling abstraction models [38] originally designed to manage simpler systems. Schedulers must evolve to support these rapid changes: increasing core counts, new sharing properties (e.g., SMT), and heterogeneous systems (e.g., big.LITTLE [39]). NUMA properties of even a single-socket platform continue to increase (e.g., chiplets [40]). Compute offloads such as AWS Nitro [41] and DPUs [42] as well as domain-specific accelerators such as GPUs and TPUs [43] are new types of tightly coupled compute devices that exist entirely beyond the domain of a classical scheduler. We need a paradigm to more efficiently support the evolving problem domain than is possible today with monolithic kernel implementations.

硬件不断提升，因而我们的调度策略也可以随之发展

比如说增加cpu核数

big.little是大小核心技术，详情可见https://news.mydrivers.com/1/429/429012.htm

这里介绍到的硬件技术，是不是也能算创新点？

除了硬件发展的传统调度外，还有很多新型计算设备的调度（GPU、TPU也值得我们考虑）

聚焦于宏内核实现

In this paper, we present the design of ghOSt and its evaluation on production and academic use-cases. ghOSt is a scheduler for native OS threads that delegates policy decisions to userspace. *The goal of ghOSt is to fundamentally change how scheduling policies are designed, implemented, and deployed. ghOSt provides the agility of userspace development and ease of deployment, while still enabling μs-scale scheduling.* ghOSt provides abstraction and interface for userspace software to define complex scheduling policies, from the per-CPU model to a system-wide (centralized) model. Importantly, ghOSt decouples the kernel scheduling mechanism from policy definition. The mechanism resides in the kernel and rarely changes. The policy definition resides in userspace and rapidly changes. We open-source our implementation to enable future research and development using ghOSt [44, 45].

woc，既有原理又有应用场景，无敌了

ghost的调度对象依然是os threads，只不过把调度策略托给了用户态实现

确实，它给用户态提供了很多接口来帮助实现调度策略

ghOSt将内核调度机制与策略定义解耦。这个感觉也有点类似CFS，CFS应该就是将策略，以及内核的具体调度那段代码解耦了。这个解耦就是cmu15445那个LRU-K和bpm的解耦

By scheduling native threads, ghOSt supports existing applications *without any changes*. In ghOSt (Fig. 1), the scheduling policy logic runs within normal Linux processes, denoted as *agents*, which interact with the kernel via the ghOSt API. The kernel notifies the *agent(s)* of state changes for all managed threads – e.g., thread creation/block/wakeup – via message queues (§3.1) in an asynchronous path. The

调度策略runs as 普通的linux process

agent=调度策略

这也就是thread的生命周期

消息队列

异步地

所以说，agents（调度策略）通过ghOSt的API与内核互动。
内核通过消息队列异步地通知agents线程的生命周期等信息；agents收到信息后同步地通过基于事务的API来做出调度决策。
这里就能体现出上面说到的“调度”和“策略”二者的解耦了。具体调度还是由内核实现，这里只是把具体的策略推给了用户态来做，也就是用户态对内核调度发号施令。
这下明白为什么是“基于CFS”了。

而ghost要做的，就是支持并发执行多个策略、隔离故障、将CPU资源分配到不同的用户程序

基于事务的API？ 同步的

agent(s) then use a transaction-based API in a synchronous path to commit scheduling decisions for the threads (§3.2). ghOSt supports concurrent execution of multiple policies, fault isolation, and allocation of CPU resources to different applications. Importantly, to enable practical transition of existing systems in the fleet to use ghOSt, it co-exists with other schedulers running in the kernel, such as CFS (§3.4).

ghost与CFS共存

We demonstrate that ghOSt enables us to define various policies leading to performance comparable or better than existing schedulers on both academic and production workloads (§4). We characterize the overheads of key ghOSt operations (§4.1). We show that ghOSt's overheads are small and range from 265 ns for message delivery, several hundred nanoseconds to context-switch into an agent, and 888 ns to schedule a thread, making ghOSt scheduling overheads only slightly higher than in existing kernel schedulers. With amortization, these overheads allow just a single ghOSt agent to schedule over 2 million threads per second (Fig. 5). We evaluate ghOSt by implementing centralized and preemptive policies for μs-scale workloads that lead to high throughput and low tail latency in the presence of high request dispersion [10, 25] and antagonists [1, 21] (§4.2). We compare ghOSt to Shinjuku [25], a specialized modern data plane, to show ghOSt's minimal overhead and competitive μs-scale performance (within 5% of Shinjuku) while supporting a broader set of workloads, including multi-tenancy.

看来策略切换也是需要上下文转换的

我们通过对μs规模的工作负载实施集中式和抢占式的调度策略来评估ghOSt

这里它提供了几种性能调优的思路：
1.测试时间开销
2.采用特定场景workload来测试
3.与别的调度框架做对比

We also implement a policy for Snap [2], our production packet-switching framework used in our data centers, leading to comparable and in some cases 5-30% better tail latency than MicroQuanta, our soft real-time scheduler used today (§4.3). We then implement a ghOSt policy for machines running Google Search (§4.4). By customizing the policy to the machine's topology, in <1000 total lines of code, we demonstrate that ghOSt matches the throughput provided by the existing scheduler, and often outperforms the latency by 40-50%. Lastly, we implement a policy for virtual machines (§4.5) that is secure against recently discovered microarchitectural vulnerabilities [27–32] and show that ghOSt's performance is competitive with a pure in-kernel policy implementation. With ghOSt, scheduling strategies — previously requiring extensive kernel modification — can be implemented in just 10s or 100s of lines of code.

2 Background & Design Goals

这个部分对于写文档应该很有帮助

Large cloud providers and users (e.g., cloud clients) are motivated to deploy new scheduling policies to optimize performance for key workloads running on increasingly complex hardware topologies, and provide protection against new hardware vulnerabilities such as Spectre [29–32] and L1TF/MDS [27, 31, 46] by isolating untrusted threads on separate physical cores.

Linux支持通过调度类来实现多个策略

Scheduling in Linux. Linux supports implementing multiple policies via scheduling *classes* [7]. Classes are ordered by their priority: a thread scheduled with a higher priority

并且linux不同策略类是有优先级的。
每个线程调度应该只能依据一个策略。具有高优先级调度策略的线程会比具有低优先级调度策略的进程优先调度
也就是说，进程调度的时候，影响其优先级的有，它的调度策略的优先级，以及它在调度策略中的优先级。

为了优化特定应用程序的性能，开发人员可以修改调度类，以更好地适应应用程序的需求，例如，对应用程序的线程进行优先排序，以减少网络延迟[21]，或者使用实时策略调度应用程序的线程，以满足数据库查询的截止日期[47]。

class will preempt a thread scheduled with a lower priority class. To optimize performance of specific applications, developers can modify a scheduling class to better fit the application's needs, e.g., prioritizing the application's threads to reduce network latency [21] or scheduling the application's threads using real-time policies to meet deadlines [47] for database queries. In principle, a cloud provider or an application developer can also create an entirely new class/policy (rather than modify an existing one) optimized for a specific service, such as cloud virtual machines [24] or a reinforcement learning framework [20]. However, the complexity of implementing and maintaining these policies in the kernel leads many developers to instead use existing generic policies, such as the Linux Completely Fair Scheduler (CFS [7]). Therefore, existing *classes* in Linux are designed to support as many use-cases as possible. It is then challenging to use these overly-generic classes to optimize high-performance applications to use the hardware at maximum efficiency.

开发人员既可以修改已有调度类，也可以创建一个新的调度类

Implementing schedulers is hard. When developers do embark on designing a new kernel scheduler, they find these schedulers hard to implement, test, and debug. Schedulers are typically written in low-level languages (C and assembly), cannot leverage useful libraries (e.g., Facebook Folly [48] and Google Abseil [49]), and cannot be introspected with popular debugging tools. Lastly, schedulers depend on and interact with complicated synchronization primitives including atomic operations, RCU [50], task preemption, and interrupts, making development and debugging even harder. Long-term maintenance is also a challenge. Linux rarely merges new scheduling classes, and would be especially unlikely to accept a highly-tuned non-generic scheduler. Thus, custom schedulers are maintained out-of-tree with consistent merge churn as upstream Linux evolves.

部署对调度策略的更改需要跨一个大型机群部署一个新的内核。

Deploying schedulers is even harder. Deploying changes to scheduling policy requires deploying a new kernel across a large fleet. This is extremely challenging for cloud providers. So much so that, in our experience, kernel rollouts are not well-tolerated below an *O*(month) granularity. To install a new kernel on a machine, cloud providers must migrate or terminate the machine's assigned work, quiesce the machine, install the new system software, allow system daemons to re-initialize, and finally, wait for newly assigned applications to become ready to serve again. Initial deployment of the new scheduler is only the beginning. After the first release candidate is deployed, the cloud provider will make frequent changes to fix bugs and tune performance, repeating the expensive process described above. At Google, for instance, there was a scheduler bug on our disk servers that led to millions of dollars of lost revenue until the bug was noticed and fixed [51]. ghOSt enables scheduler update, testing, and tuning without having to update the kernel and/or reboot machines and applications.

O(月)草

ghOSt支持调度程序的更新、测试和调优，而不必更新内核和/或重新启动机器和应用程序。
这也正是ebpf的卖点之一吧

User-level threading is not enough. ghOSt is a kernel scheduler running in userspace, scheduling native OS threads. In contrast, user-level threading runtimes schedule *user* threads. These runtimes [22, 48, 52–60] multiplex M *user* threads on N *native* threads. This is inherently unpredictable: although the userspace runtime may control which user thread runs on a given native thread, it cannot control when that *native* thread is scheduled to actually run or which CPU it runs on. Even worse, the kernel can deschedule a thread holding a user-level lock. To overcome this limitation, developers have two options. *(1)* Dedicate CPUs to the native threads running the user-threads, thus guaranteeing implicit control. However, this option wastes resources at low workload utilization, because the dedicated CPUs cannot be shared with another application (see §4.2), and requires extensive coordination around scaling capacity. Alternatively, developers can *(2)* stay at the mercy of the native thread scheduler, allowing CPUs to be shared, but ultimately losing the control over response time that they turned to a user-level runtime for. ghOSt enables the best of both worlds by guaranteeing control over response time while allowing flexible sharing of CPU resources.

这里澄清了一件事，就是ghost是内核级的调度器，只不过run在用户态。它管理的是内核级线程。
它不是管理用户级线程的用户态调度器。

ghOSt通过保证对响应时间的控制，同时允许灵活地共享CPU资源，从而实现了两方面的优点。

Custom scheduler/data plane OS per workload is impractical. Previous work, such as Shinjuku, Shenango, and others [1, 10, 21, 25, 34], implemented highly specialized data plane operating systems with custom scheduling policies and network stacks for specific network workloads. Although these systems provide good performance for their targets, their kernel implementation cannot be easily changed and they provide poor performance for other workloads. Shinjuku [25] is 2,535 lines of code and cannot co-exist with other applications in the system (§4.2). Shenango [1] is 8,399 lines of code and can only implement a single thread scheduling policy for network workloads. Adding an additional policy requires significant code modification. Further, both systems are unable to run on machines without particular NICs and Shenango cannot schedule non-network workloads.

通过BPF进行自定义调度是不够的

遇到关键词自动注意！怎么突然说到了BPF

Custom scheduling via BPF is insufficient. An attractive way to customize kernel scheduling is to inject BPF [61] programs into the kernel scheduler, as has been done for other kernel subsystems [62]. It is indeed possible to implement a Linux scheduler class whose function pointers call a BPF program to decide which thread to run next. Unfortunately, BPF is limited in its expressiveness and which kernel data structures it can access. For example, the BPF-verifier must be able to determine that loops will exit, and BPF programs cannot use floats.

对对对！这正是我本来的思路，难道要否了吗2333

BPF验证器必须能够确定循环将退出，并且BPF程序不能使用浮点数
确实这对调度器来说很致命，因为调度器就是要死循环，而且一些优先级算法需要进行浮点运算

More importantly, BPF programs run *synchronously*, meaning they must react quickly to scheduling events, blocking the CPU until they complete. An *asynchronous scheduler*, in contrast, can receive scheduling events and react to these events at a later time. Therefore, an asynchronous model, such as the global scheduler described in §3.3, can make scheduling decisions based on a wider perspective of the system, comprised from multiple scheduling events. That being

但确实还是要用到BPF的，按他的意思，只不过不是像我以前简单理解的那样那么用。具体怎么用，就看后面怎么说了。

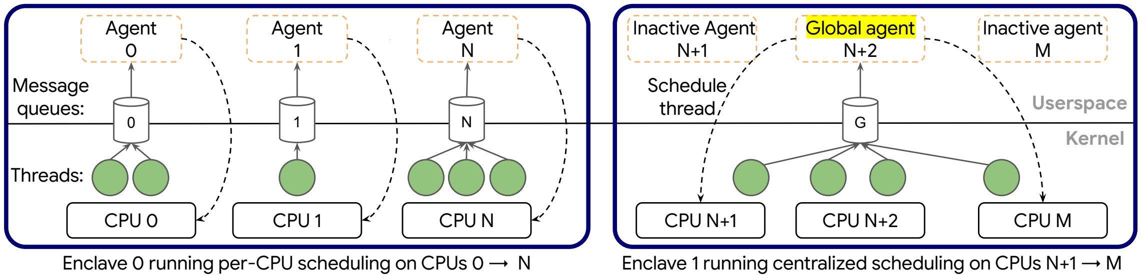


Figure 2. ghOSt policies manage CPUs assigned to enclaves. Each enclave can be managed by a different ghOSt policy.

said, BPF plays a large role in ghOSt to *accelerate* fast-path operations and provide customization at performance-critical spots in the kernel, as explained in §3.2.

2.1 Design Goals

ghOSt's goal is to introduce a new paradigm for designing, optimizing, and deploying schedulers. We design ghOSt with the following requirements in mind:

Policies should be easy to implement and test. As new workloads, kernels, and heterogeneous hardware enter the data center, implementing new scheduling policies should not mandate yet another kernel patch. As DashFS [63] demonstrated, implementing systems in userspace rather than kernel space simplifies development and enables faster iteration as popular languages and tools can be used.

在用户空间而不是内核空间中实现系统简化了开发，并支持更快的迭代，因为可以使用流行的语言和工具。

Scheduling expressiveness and efficiency. Cloud providers and users need to express scheduling policies for a wide variety of optimization targets, including throughput-oriented workloads, μs-scale latency-critical workloads, soft real-time, and energy efficiency requirements.

Enabling scheduling decisions beyond the per-CPU model. The Linux scheduler ecosystem implements scheduling algorithms that make per-CPU scheduling decisions, i.e., they use a *per-CPU model*. Recent work demonstrates compelling tail latency improvements for μs-scale workloads via centralized, dedicated polling scheduling threads [1, 21, 25], i.e., they use a *centralized model*. Other systems such as ESXi NUMA [64] and Minos [15] have also demonstrated the benefits of coordination at a granularity coarser than a single CPU, such as per-NUMA-node. The centralized model is difficult to support within Linux's existing "Pluggable Scheduler" framework, as the framework is implemented with the assumption of a per-CPU model. ghOSt should support delegating scheduling decisions to remote CPUs to support all scheduling models from per-CPU to centralized and everything in between.

Linux调度程序生态系统实现了调度每个cpu调度决策的调度算法，也就是说，它们使用per-CPU model

集中的专用轮询调度线程

在比单个CPU更粗的粒度上进行协调的好处

ghOSt应该支持将调度决策委托给远程cpu，以支持从每个cpu到集中式以及介于两者之间的所有调度模型

也就是说ghost不仅支持per-CPU，也支持集中式？

Supporting multiple concurrent policies. Machine capacity is continuing to horizontally scale with the addition of hundreds of cores and new accelerators, supporting multiple loads and tenants on a single server. ghOSt must support partitioning and sharing the machine so that multiple scheduling policies may execute in parallel.

并行执行多个调度策略

Non-disruptive updates and fault isolation. OS upgrades on a large fleet incur expensive downtime. The work assigned to each host must be migrated or restarted and the host itself must reboot. This time-consuming process reduces compute capacity and hinders the data center's fault-tolerance until the update completes. This is even a challenge for users rolling out their own policies for similar reasons: they must maintain service uptime while manually taking down and upgrading nodes. Therefore, the scheduling policy should be decoupled from the host kernel and ghOSt must allow new policies to be deployed, updated, rolled-back, or even crash without incurring the machine-reboot costs.

无中断的更新和故障隔离

调度策略应该与主机内核解耦，ghOSt必须允许部署、更新、回滚甚至崩溃，而不产生机器重新启动成本

设计目标

+ 容易实现和测试

+ 效率高，易表达

+ 不局限于per-CPU模型

+ 支持多种并发策略

+ 非中断更新（不需要重启）和错误隔离

3 Design

We now discuss ghOSt's design and implementation, and explain how they achieve our requirements listed in §2.

ghOSt overview. Fig. 1 summarizes the ghOSt design. Userspace agents make scheduling decisions and instruct the kernel how to schedule native threads on CPUs. ghOSt's kernel side is implemented as a scheduling class, akin to the commonly used CFS class. This scheduling class provides userspace code with a rich API to define arbitrary scheduling policies. To help the agents make scheduling decisions, the kernel exposes thread state to the agents via *messages* and *status words* (§3.1). The agents then instruct the kernel on scheduling decisions via *transactions* and system calls (§3.2).

在内核中，具体实现为一个调度类，类似于CFSclass。这个调度类提供了一个带有丰富API的用户空间代码来定义任意的调度策略。那么它是怎么控制内核中执行用户代码的呢？还是用ebpf吗？

We will use two motivating examples throughout this section: *per-CPU* scheduling and *centralized* scheduling. Classical kernel scheduling policies, such as CFS, are per-CPU schedulers. Although these policies typically employ load-balancing and work-stealing to even out the load across the system, they still operate from a per-CPU perspective. The centralized scheduling example is similar to the models proposed in Shinjuku [25], Shenango [1], and Caladan [21]. In this case, there is a single global entity constantly observing the entire system and making scheduling decisions for all threads and CPUs under its purview.

接下来会分别针对两个模型分析，来说明ghost的具体实现

尽管这些策略通常使用负载平衡和偷工作窃取来平衡整个系统的负载，但它们仍然从每个cpu的角度来操作。

在这种情况下，有一个全局实体不断地观察整个系统，并在其权限范围内的所有线程和cpu做出调度决策
这也就是前面说到的轮询线程了

Threads and CPUs. ghOSt schedules native threads on CPUs. All threads mentioned in this section are native threads, in contrast to user-level threads mentioned in §2. We refer to logical execution units as CPUs. For example, we consider a machine with 56 *physical* cores and 112 *logical* cores (hyperthreads) to have 112 CPUs.

之后说的cpu都是上面介绍NUMA架构时提到的逻辑CPU。

Partitioning the machine. ghOSt supports multiple concurrent policies on a single machine using *enclaves*. A system

对于per-CPU model，每个逻辑CPU都能有自己的一个调度策略；
但对于集中model，应该是会对逻辑CPU进行分组，每组有一个活跃的global调度策略，和一个暂时disable的调度策略

can be partitioned into multiple independent enclaves, at CPU granularity, each of which runs its own policy, as depicted in Fig. 2. From a scheduling perspective, the enclaves are isolated. Partitioning makes sense especially when running different workloads on a single machine. It is often useful to set the granularity of these enclaves by machine topology, such as per-NUMA-socket or per-AMD-CCX [40]. Enclaves also help in isolating faults, limiting the damage of an agent-crash to the enclave it belongs to (see §3.4).

ghOSt userspace agents. To achieve many of our design goals, the scheduling policy logic is implemented in userspace *agents*. The agents can be written in any language and debugged by standard tools, making them easier to implement and test. To achieve fault tolerance and isolation, if one or several of the agents crash, the system will fall back to the default scheduler, such as CFS. The machine is then still fully functional while a new ghOSt userspace agent is launched — either the last known stable release or a newer revision with a fix.

为了实现容错和隔离，如果其中一个或几个代理崩溃，系统将返回到默认的调度程序，如CFS。

Thanks to the crash resilience property, updating a scheduling policy amounts to relaunching the userspace agents, *without having to reboot the machine*. This property enables experimentation and rapid policy customization for a wide variety of hardware and workloads. Developers can make a policy tweak and simply relaunch the agents. Dynamic update of a ghOSt policy is discussed in §3.4.

更新调度策略相当于重新启动用户空间代理，而不必重新启动计算机。

Regardless of the scheduling model — per-CPU or centralized — each *CPU* managed by ghOSt has a local *agent*, as shown in Fig. 2. In the per-CPU case, each agent is responsible for thread scheduling decisions for its own CPU. In the centralized case, a single global agent is responsible for scheduling all CPUs in the enclave. All other local agents are *inactive*. Each agent is implemented in a Linux pthread and all agents belong to the same userspace process.

要点：
1.采用消息队列
2.一个agent一个消息队列
3.支持负载均衡的窃取
4.使用seq num保证通信同步，seq存储在shared-memory

要开始讲内核到用户的通信了。我记得是通过异步消息队列吧

3.1 Kernel-to-Agent Communication

Exposing thread state to the agents. For agents to make scheduling decisions for threads under their purview, the kernel must expose thread state to the agents. One approach is to memory-map existing kernel data structures into userspace, such as `task_structs`, so agents can inspect them to infer the thread state. However, the availability and format of these data structures varies between kernels and kernel versions, tightly coupling *userspace* policy implementation with *kernel* version. Another approach is to expose thread state via sysfs files, in a `/proc/pid/...` fashion. However, file system APIs are inefficient for fastpath operations, making it difficult to support μs-scale policies: `open/read/fseek`, originally designed for block devices, are too slow and complex (e.g., require error handling and data-parsing).

这里否决了两种方法：
1.将内核内部的结构体内存映射给userspace
2.将结构体信息存储到user可读的文件中
最后采用了消息队列的方法

Ultimately, we need a kernel-userspace API that is both fast and does not depend on the underlying kernel implementation of threads. Inspired by distributed systems, we use *messages* as an efficient and simple solution.

灵感来自分布式系统的消息队列

kernel会给agent传递时钟中断的信息，以及发送某个进程当前的状态

ghOSt messages. In ghOSt, the kernel uses the *messages* listed in Table 1 to notify the userspace agents of thread state changes. For example, if a thread was blocked and now ready to run, the kernel posts a `THREAD_WAKEUP` message. Additionally, the kernel informs the agents of timer ticks with a `TIMER_TICK` message. To help agents verify they are making decisions based on the most up-to-date state, messages also have sequence numbers, as will be explained later.

这是不是跟网络传输蛮像的

Message queues. Messages are delivered to agents via *message queues*. Each thread scheduled under ghOSt is assigned a single queue and all messages about that thread's state changes are delivered to that queue. In the per-CPU example, each thread is assigned to a queue corresponding to the CPU it is intended to run on (Fig. 2, left). In the centralized example, all threads are assigned to the global queue (Fig. 2, right). Messages for CPU events, such as `TIMER_TICK`, are routed to the queue of the agent thread associated with the CPU.

每个调度族都有一条消息队列。也就是per-cpu的每个逻辑cpu都有一个消息队列，集中model的一个global agent有一个消息队列。
也就是说一个agent有一个消息队列。

Although there are many ways to implement queues, we opted to use custom queues in shared memory to efficiently handle agent wakeups (explained below). We deemed existing queue mechanisms to be insufficient for ghOSt as they only exist in specific kernel versions. For instance, the BPF system passes BPF events to userspace via BPF ring buffers [65] and recent versions of Linux also pass asynchronous I/O messages to userspace via `io_urings` [66]. These are both fast lockless ring buffers that synchronize consumer/producer access. However, older Linux kernels and other operating systems do not support them.

Thread-to-queue association. After ghOSt's enclave initialization, there is a single default queue in the enclave. The agent process can create/destroy queues using the `CREATE/DESTROY_QUEUE()` API. Threads added to ghOSt are implicitly assigned to post messages to the default queue. That assignment can be changed by the agent via `ASSOCIATE_QUEUE()`.

Queue-to-agent association. A queue may be optionally configured to wake up one or more agents when messages are produced into the queue. The agent can configure the wakeup behavior via `CONFIG_QUEUE_WAKEUP()`. In the per-CPU example, each queue is associated with exactly one CPU and configured to wake up the corresponding agent. In the centralized example, the queue is continuously polled by the global agent so a wakeup is redundant and therefore not configured. The latency of producing a message into a queue and observing it in the agent is discussed in §4.1.

per-cpu中，需要唤醒agent；密集型中，队列是会一直被global agent轮询的

Agent wakeup uses the standard kernel mechanism to wake up a blocked thread. This involves identifying the agent thread to be woken up, marking it as runnable, optionally sending an interrupt to the target CPU to trigger a reschedule, and performing a context switch to the agent thread.

是的，前面说了agent本质也是一个线程，因而可以用kernel的唤醒机制

Moving threads between queues/CPUs. In our per-CPU example, to enable load-balancing and work-stealing between CPUs, agents can change the routing of messages

在per-cpu模型中，我们还支持负载均衡的任务窃取（当然具体算法由agent自身决定）

其实“agent也是线程”这个观念让我觉得非常有意思。
之前在xv6的时候，看到它将scheduler称为“调度线程”，我还觉得很惊艳，因为确实，别的线程/进程yield之后它就被唤醒了，它还会切换上下文切到另一个线程，它自身的这些行为不就表明它自己也可以被视为一个线程嘛！
同理这边也是一个道理。agent会sleep，直到被消息队列有消息这个事件wakeup；agent也会做出决策，切换到另一个线程中。

Messages	Syscalls
THREAD_CREATED	AGENT_INIT()
THREAD_BLOCKED	START_GHOST()
THREAD_PREEMPTED	TXN_CREATE()
THREAD_YIELD	TXNS_COMMIT()
THREAD_DEAD	TXNS_RECALL()
THREAD_WAKEUP	CREATE_QUEUE()
THREAD_AFFINITY	DESTROY_QUEUE()
TIMER_TICK	ASSOCIATE_QUEUE()
	CONFIG_QUEUE_WAKEUP()

更改某个线程的消息队列，更改时队列必须空

Table 1. ghOSt messages and system calls.

？？？但是一个agent不是只对应一个cpu，一个消息队列吗？它怎么知道要把mesg rout到哪去？

from threads to queues via ASSOCIATE_QUEUE(). It is up to the agent implementation (in userspace) to properly coordinate the message routing across queues to agents. If a thread has its association change from one queue to another while there are pending messages in the original queue, the association operation will fail. In that case, the agent must drain the original queue before re-issuing ASSOCIATE_QUEUE().

新消息到来可能导致决定更改。我们通过seq num解决这个问题。【这里跟计网很相似！也确实，网络通信怎么说也算是一种消息队列】

Synchronizing agents with the kernel. Agents operate on the system's state as observed via messages. However, while the agent is making a scheduling decision, new messages may arrive into the queue which could change that decision. This challenge is slightly different for the per-CPU example versus the centralized scheduling example (see §3.2 and §3.3). Either way, we address this challenge with *agent/thread sequence numbers*: Every agent has a sequence-number, A_{seq}, which is incremented whenever a message is posted to a queue associated with that agent. We explain our use of A_{seq} for the per-CPU example in §3.2. Every thread T has a sequence-number, T_{seq}, which is incremented whenever that thread posts a new state change message, M_T. When an agent pops the queue it receives both a message and its corresponding sequence number: (M_T, T_{seq}). We explain how we use T_{seq} for the centralized scheduling example in §3.3.

也就是说，进程状态信息采用消息队列而不是memory-map，但是维护消息队列所需的seq num采用了mmap
每次kernel update的时候，也会同步update mmap部分

Exposing sequence numbers via shared memory. ghOSt allows agents to efficiently poll auxiliary information about thread and CPU state through *status words*, mapped into the agent's address space. For brevity, we only discuss our use of status words to expose sequence numbers, A_{seq}

看来确实把状态塞进队列、递增seq num都是在kernel做的，这是否就用到了ebpf？

and T_{seq}, to the agents. When the kernel updates a thread's or agent's sequence number, it also updates the corresponding status word. Agents can then read the sequence numbers from the status words in the shared mapping.

要点：
1.使用事务提交调度策略，允许group commit

2.使用seq num保证通信同步

3.使用bpf在未决定调度策略时，注入对随便某个进程的调度

其实我觉得12点也得用bpf来修改内核代码的，还是说它直接修改了内核代码emmm

到时候可以问问赛题指导老师，能不能修改内核实现，这个问题。

3.2 Agent-to-Kernel Communication

We now describe how the agents instruct the kernel which thread to schedule next.

Sending scheduling decisions via transactions. Agents send scheduling decisions to the kernel by committing *transactions*. Agents must be able to schedule both their local CPU (per-CPU case) as well as other remote CPUs (centralized case). The commit mechanism must be fast to support μs-scale policies and scale to hundreds of cores. For the per-CPU example, a syscall interface, in theory, would suffice. For the centralized case, the agent needs to efficiently send scheduling requests to multiple CPUs and

```
void Agent::PerCpuSchedule() {
    DrainMessageQueue(); // Read messages from queue
    Thread *next = runqueue_.Dequeue();
    if (next == nullptr) return; // Runqueue empty.
    // Schedule thread:
    Transaction *txn = TXN_CREATE(next->tid, my_cpu);
    TXNS_COMMIT({txn});
    if (txn->status != TXN_COMMITTED) {
        // Txn failed. Move thread to end of runqueue.
        runqueue_.Enqueue(next);
        return;
    }
    // The schedule has succeeded for `next`.
}
```

Figure 3. Scheduling a thread in per-CPU agent code.

then inspect whether those requests succeeded or not. A shared memory interface is therefore more suitable. As a side note, using transactions in shared memory as the scheduling interface would allow, in the future, to offload scheduling decisions to an external device with access to that memory.

这个所谓的shared什么什么interface，指的就是txn这个结构体吧

Inspired by transactional memory [67] and database [68] systems, we designed our own transaction API, implemented via shared memory. These systems support fast, distributed commit operations with atomic semantics, and there could be multiple commits that simultaneously target the same remote node. ghOSt agents require similar properties. Agents open a new transaction in shared memory with the TXN_CREATE() helper function. The agent writes both the TID of the thread to schedule along with the ID of the CPU to schedule the thread on. In the per-CPU example, each agent only schedules its own CPU. When the transaction is filled in, the agent commits it to the kernel via the TXNS_COMMIT() syscall, which kicks off the commit procedure and triggers the kernel to initiate a context switch. A simplified example is shown in Fig. 3.

代理写入要调度的线程的TID和要调度线程的CPU的ID

这也是ebpf吧！感觉到目前为止，ebpf就只用做两个固定的事：
1.hook在进程状态change的那个点，在那个点进行进程状态信息的提交以及对seq num的递增
2.hook在TXNS_COMMIT这个syscall执行的点，控制进行上下文的切换

Group commits. In the centralized scheduling example, to allow ghOSt to scale to hundreds of CPUs and hundreds of thousands of transactions per second, we must mitigate the

这个思想很值得学习！通过group commit来摊还单次commit的开销

expensive cost of system calls. We amortize the cost of transactions by introducing group commits. Group commits also reduce the number of interrupts to be sent to other CPUs, similar to Caladan [21]. An agent commits multiple transactions by passing all of them to the TXNS_COMMIT() syscall. This syscall amortizes the expensive overheads over several transactions. Most importantly, it amortizes the overhead of

批处理中断

sending interrupts by using the batch interrupt functionality present in most processors. Instead of sending multiple interrupts (one per transaction), the kernel sends a single batch interrupt to the remote CPUs, saving significant overhead.

Sequence numbers and transactions. In the per-CPU example, the agent committing a transaction is giving up its CPU to the target thread it is scheduling. Messages posted to the queue while the agent is running *do not* cause a wakeup, since the agent is already running. However, the new message in the queue might be from a higher-priority thread,

bingo！这句话就精髓地写出了我前面说的，agent也是线程，这个观点

agent想要立刻接收到消息队列的信息的话，必须处于sleep状态，这样它才能被信息wakeup。
而如果当agent正在run，它就无法接收到最新消息。只有在当他make decision之后，将当前CPU所有权转让给下一个需要被调度的线程之后，再次进入沉睡，才能再次被唤醒，从而得到最新消息，但这太晚了。
to address this，我们引入了seq num。

接下来这部分会说明seq num是怎么在per-CPU model中解决此困难的。

```
void GlobalAgent::CentralizedSchedule() {
    DrainMessageQueue();
    map<Cpu, Thread*> assignments;
    // GetIdleCPUs() will return all available CPUs.
    for (const Cpu& cpu : GetIdleCPUs()) {
        Thread *next = runqueue_.Dequeue();
        if (next == nullptr) break; // Runqueue empty.
        assignments[cpu] = next; // Run `next` on `cpu`.
    }
    // Now send transactions for all assignments:
    vector<Transaction*> txns = Schedule(assignments);
    for (const Transaction *txn : txns) {
        // Check if `txn` committed successfully.
        if (txn->status != TXN_COMMITTED) {
            Thread *next = GetThreadFromTID(txn->tid);
            // Transaction failed. Re-enqueue.
            runqueue_.Enqueue(next);
            continue;
} } }
vector<Transaction*> GlobalAgent::Schedule(
  const map<Cpu, Thread*>& assignments) {
  vector<Transaction*> txns;
  for (const auto& [cpu, next] : assignments) {
    Transaction *txn = TXN_CREATE(next->tid, cpu);
    txns.push_back(txn);
  }
  TXNS_COMMIT(txns);
  return txns;
}
```

对cpu的指派

Figure 4. A simplified example of a global agent.

and would affect the scheduling decision if the agent were aware of it. The agent will only get a chance to inspect that message on the next wakeup, which is too late. We now explain how to address this challenge via sequence numbers for the per-CPU example. We explain the slightly different case for centralized scheduling in §3.3.

We resolve this challenge using the agent sequence number, A_{seq}. An agent polls for its A_{seq} by inspecting the agent-thread's *status word*. Recall that A_{seq} is incremented when a new message is posted to the queue associated with the agent. The order of operations is: 1) Read A_{seq}; 2) Read messages from queue; 3) Make a scheduling decision; and 4) Send A_{seq} alongside the transaction to TXNS_COMMIT(). If the A_{seq} sent with the transaction is older than the current A_{seq} observed by the kernel (i.e., a new message was posted to the agent's queue), the transaction is considered "stale" and will fail with an ESTALE error. The agent then drains its queue to retrieve the newer messages and repeats the process.

通过mmap轮询自己的seq

所以对seq num是否陈旧的检查是在agent发送之后才在内核代码中确认的啊

这里终于提到了ebpf

Accelerating scheduling with BPF. The user-level flexibility provided by ghOSt is not free: message delivery and group scheduling incur up to 5 μs (see Table 3 in §4.1); in the centralized scheduling model, a thread might wait an entire centralized-scheduling loop until a scheduling decision is committed on its behalf (30 μs in §4.4).

ghOSt allows recovering that lost CPU time via a custom BPF program, attached by the agent to the kernel's pick_next_task() function. When a CPU becomes idle and the agent has not already issued a transaction, the BPF program issues its own transaction, picking a thread to run on that CPU. The BPF program communicates with the agent via a shared-memory window into the agent's address space. The specifics of how the agent uses the BPF infrastructure to schedule threads on CPUs is part of the scheduling policy. The ghOSt BPF program is essentially an extension of the agent itself, and hence the BPF bytecode is embedded in the agent binary, using *libbpf* [69].

当CPU变得空闲而代理尚未发出事务时，BPF程序会发出自己的事务，选择一个在该CPU上运行的线程。
这是利用了ebpf的即时性：你随便选一个就行。这部分改动其实也不是不能在agent做或者多弄一个线程做。但是ebpf胜就胜在其方便高效

要点：
1.一个agent一个队列
2.由于global agent采用轮询方式查询消息队列，因而要将其优先级设为所有进程中的最高级；同时还得引入idle机制维护kernel stability
3.引入thread seq保证通信同步【这点现在没弄懂】

3.3 The Centralized Scheduler

We now explain additional implementation details required for constructing a centralized scheduling ghOSt policy.

One global agent with a single queue. For centralized scheduling, there is a single global agent polling a single message queue and making scheduling decisions for all CPUs managed under ghOSt. If a designated CPU already runs a ghOSt thread, the transaction will preempt that previous thread in favor of the new one. A simplified example of the scheduler's code is depicted in Fig. 4. Intuitively, the centralized policy may seem incapable of supporting μs-scale scheduling, though we show in §4 that ghOSt has comparable or better overall performance on our production workloads.

在per-cpu中，agent与消息队列的交互采用wakeup机制；但在centralized中，使用轮询机制

如果一个指定的CPU已经运行了一个ghOSt线程，则该事务将优先于之前的线程，而不是新的线程。【为啥？？？？】

Avoiding preemption of the global agent. To support μs-scale scheduling, the global agent must continuously run, as any preemption will directly lead to scheduling delays. To prevent global agent preemption triggered by a higher priority kernel scheduling class, ghOSt assigns *all* agents a high kernel priority, similar to real-time scheduling. In other words, no other thread in the machine, whether ghOSt or non-ghOSt, can preempt agent-threads. This priority assignment, however, will destabilize the system unless handled carefully. For example, most systems have per-CPU daemon worker threads that must run on their designated CPUs.

要让调度线程一直跑下去

然而，除非仔细处理，否则这种优先级分配将使系统不稳定。例如，大多数系统都有每个cpu守护进程的工作线程，它们必须在它们指定的cpu上运行。

ghOSt maintains the system's stability in the following way. All inactive agents immediately yield, vacating their CPUs. Whenever a non-ghOSt thread needs to run on the global agent's CPU, CPU_{global}, the global agent performs a "hot handoff" to an inactive agent on another CPU, CPU_{idle}. For example, if the kernel CFS scheduler tries to schedule a thread on CPU_{global}, the global agent will first find an idle CPU (CPU_{idle}) and then wake up the inactive agent on CPU_{idle} to serve as the new global agent. Once CPU_{idle} runs the global agent, the old global agent yields, allowing the CFS thread to run on CPU_{global}.

当non-ghost进程要用agent的cpu时，agent就马上yield，并且让其中一个占据了别的cpu的idle agent进行工作

Sequence numbers and centralized scheduling. At some point, the global agent may have an inconsistent view of a thread's state. For example, a thread T might post a THREAD_WAKEUP message. The global agent receives this message and decides to schedule T on CPU_f. In the meantime, some entity in the system invoked sched_setaffinity(), leading to a THREAD_AFFINITY message, *forbidding* T from running on CPU_f. We need a mechanism to ensure that the transaction that schedules T on CPU_f will fail.

THREAD_AFFINITY（线程绑核），这里意思就是说别的线程独占了某个CPU

In principle, we can use agent sequence numbers, as described above for the per-CPU example. However, the global

per-CPU中，每个agent只对应一个CPU，因而内存区只有一个；但是在这里，有多个内存区域，因此应该很难保证seq的一致性？

Linux CFS (kernel/sched/fair.c)	6,217 LOC
Shinjuku [25] (NSDI '19)	3,900 LOC
Shenango [1] (NSDI '19)	13,161 LOC
ghOSt Kernel Scheduling Class	3,777 LOC
ghOSt Userspace Support Library	3,115 LOC
Shinjuku Policy (§4.2)	710 LOC
Shinjuku + Shenango Policy (§4.2)	727 LOC
Google Snap Policy (§4.3)	855 LOC
Google Search Policy (§4.4)	929 LOC
Secure VM Kernel Policy (§4.5)	7,164 LOC
Secure VM ghOSt Policy (§4.5)	4,702 LOC

Table 2. Lines of code for ghOSt and compared systems.

agent has to support many thousands of threads that continuously post messages to the global queue, making it time consuming to drain the queue. Unlike the local agent in the per-CPU example, the global agent is not giving up its own CPU. The global agent must only verify that it is up-to-date with respect to the thread T being scheduled right now.

We solve this issue with *thread* sequence numbers. Recall that every queued message M_T is tagged with the thread sequence number T_{seq} as (M_T, T_{seq}). When the agent commits a transaction for thread T, it sends the transaction along with the most recent sequence number for T it is aware of: T_{seq}. When the kernel receives the transaction, it verifies that T_{seq} is up to date with respect to the thread in the transaction. Otherwise, the transaction fails with an ESTALE error.

要点：
1.agent崩溃后旗下线程转CFS调度
2.通过替换agent（优先）和摧毁族进行热更新和rollback
3.杀死指定时间仍然有线程饥饿的agent族

它这意思是说有一个thread的seq不一致，该次事务的group commit全部无效吗？

3.4 Fault Isolation and Dynamic Upgrades

错误隔离和热更新

与其他kernel 调度类的互动

Interaction with other kernel scheduling classes. One of ghOSt's design goals is enabling easy adoption on existing systems. So even if a ghOSt policy is faulty, we still want ghOSt-managed threads to interact well with other threads in the system. We want to avoid ghOSt threads causing unintended consequences for other threads, such as starvation, priority inversion, deadlock, etc.

如果policy错误，ghost管理的线程应该仍能与os正确互动。这应该就是前面说到过的转为CFS调度吧

We achieve this goal by assigning ghOSt's kernel scheduler *class* a lower priority (§2) than the default scheduler class — typically CFS — in the kernel's scheduling class hierarchy. The result is that most threads in the system will preempt ghOSt threads. The preemption of a ghOSt thread leads to the creation of a THREAD_PREEMPT message, triggering the relevant agent (which is running in a different high priority scheduling class) to make a scheduling decision. The agent further decides how to handle the preemption.

我们要实现这个目标，可以在内核的调度类层次结构中为ghOSt的内核调度器类分配一个比调度器类默认优先级更低的优先级(2)——通常是CFS。

为啥？不应该比默认的高吗？？？这段话说真的我没明白

Dynamic upgrades and rollbacks. ghOSt enables rapid deployment, since updating the scheduling policy (i.e., the agents) does not require restarting the kernel or *applications*. Many production services can take minutes to hours to start, particularly to populate in-memory caches. Similarly, we want to minimize interruptions for client virtual machines. These long-running applications continue to run correctly during a planned agent update or an unplanned agent crash. ghOSt achieves dynamic upgrades by either *(a)* replacing the

保证方法：
1.更换agent
2.摧毁族，将其下面的thread都转为CFS调度

agents while keeping the enclave infrastructure intact, or by *(b)* destroying the enclave and starting from scratch.

Replacing agents and destroying enclaves. ghOSt supports updating an agent "in-place" without destroying the enclave. Userspace code can query, and epoll on, whether an agent is attached to an enclave. To upgrade an agent, we run both old and new agents concurrently; the new agent blocks until the old agent crashes or exits and is no longer attached. The new agent extracts the state of all threads in the enclave from the kernel and resumes scheduling. If this process fails, either the kernel or userspace code can destroy the enclave. Destroying the enclave kills all the agents in that enclave, keeping other enclaves in the system intact, and automatically moves all threads in the destroyed enclave back to CFS. At this point, the threads are still functioning normally but are scheduled by CFS instead of ghOSt.

用户空间代码可以查询并打开代理是否附加到飞地。

双内核是吧2333

如果此进程【应该指更换agent的步骤】失败，内核或用户空间代码都可以破坏该飞地。

我有一个问题，就是它又不知道agen
什么时候crash，难
道是一直都有两个
agent，一个用于替
补，一直在block另
一个是真正的agen
吗

看门狗是吧

ghOSt watchdog. Scheduling bugs in ghOSt or in any other kernel scheduler have system-wide consequences. For example, a ghOSt thread may be preempted while holding a *kernel* mutex, and if it is not scheduled for too long, it could transitively stall other threads including those in CFS or other ghOSt enclaves. Similarly, the machine will grind to a halt if critical threads such as garbage collectors and I/O pollers are not scheduled. As a safety mechanism, ghOSt automatically destroys enclaves with misbehaving agents. For example, the kernel will destroy an enclave when it detects an agent has not scheduled a runnable thread within a user-configurable number of milliseconds.

这不是很正常吗？还是说其实CFS对这种情况做了优化，而我们的agent不一定会优化，所以要由ghost负责优化？

看来确实是这样

引发线程饥饿时摧毁线程

4 Evaluation

Our evaluation of ghOSt focuses on three questions: *(a)* What are the overheads of ghOSt-specific operations, which are not present in classical schedulers (§4.1); *(b)* How do scheduling policies implemented with ghOSt perform in comparison to prior work, such as Shinjuku [25] (§4.2); and *(c)* Is ghOSt a viable solution for large-scale and low-latency production workloads, including Google Snap (§4.3), Google Search (§4.4), and virtual machines (§4.5)?

4.1 Analysis of ghOSt Overheads and Scaling

Lines of code: Table 2 presents the lines of code (LOC) for ghOSt and, for reference, related work such as the Linux CFS scheduler. ghOSt is production-ready and flexibly supports a range of scheduling policies for our production workloads with 40% less kernel code than CFS. The policies in this section can be short (few 100s LOC) as they utilize common functions from a userspace library. This reflects an advantage of working within higher-level languages for policy definition: more flexible abstractions, enabling complexity to be focused on the scheduling decisions.

Experimental Setup: Unless otherwise noted, experiments run on Linux 4.15 with our ghOSt patches applied. We run microbenchmarks on a 2-socket Intel Xeon Platinum 8173M @ 2GHz, 28 cores per socket, 2 logical cores each.

并应用了我们的ghOSt补丁。看来是要修改内核了

优先级翻转：优先级翻转是当一个高优先级任务通过信号量机制访问共享资源时，该信号量已被一低优先级任务占有，因此造成高优先级任务被许多具有较低优先级任务阻塞，实时性难以得到保证。

处理器间中断（Inter-Processor Interrupt，IPI）是一种特殊类型的中断，即在多处理器系统中，如果中断处理器需要来自其它处理器的动作，一个处理器向另一个处理器发出的中断行为。可能要求采取的行动包括：刷新其它处理器的内存管理单元缓存，如转译后备缓冲器，当一个处理器更改内存映射时；停机，当系统被一个处理器关闭时。
https://blog.csdn.net/fishmai/article/details/99593954

1. Message Delivery to Local Agent	725 ns
2. Message Delivery to Global Agent	265 ns
3. Local Schedule (1 txn)	888 ns
Remote Schedule (1 txn for 1 CPU)	
4. Agent Overhead	668 ns
5. Target CPU Overhead	1064 ns
6. End-to-End Latency	1772 ns
Group Remote Schedule (10 txns for 10 CPUs)	
7. Agent Overhead	3964 ns
8. Target CPU Overhead	1821 ns
9. End-to-End Latency	5688 ns
10. Syscall Overhead	72 ns
11. pthread Minimal Context Switch Overhead	410 ns
12. CFS Context Switch Overhead	599 ns

Table 3. ghOSt microbenchmarks. End-to-end latency is not equal to the sum of agent and target overheads as the two sides do some work in parallel and the IPI propagates through the system bus.

Table 3 summarizes the overhead of basic operations that are unique to ghOSt. We also report the overhead of equivalent thread operations under CFS.

这几段事实上总结了每个部分要做的工作

Message delivery overhead (lines 1-2). In the per-CPU example, delivery to the local agent consists of adding the message to a queue, context switching to the local agent, and dequeuing the message. The overhead (725 ns) is dominated by the context switch (410 ns). In the centralized example, delivery to the global agent (265 ns) consists of adding the message to the queue and dequeuing the message within the global agent, which is always spinning.

由于per-CPU中，agent被视为可切换的线程，故而需要有一个切换到agent的上下文切换开销。但集中型不用

Local scheduling (line 3). In the per-CPU model, this is the overhead of committing a transaction and performing a context switch on the local CPU, until the target thread is running. The overhead (888 ns) is slightly higher than CFS context switch overhead (599 ns) due to the transaction commit, but still competitive.

Remote scheduling (lines 4-9). In the centralized scheduling model, the agent-side commits the transaction and sends an inter-processor interrupt (IPI). The target CPU handles the IPI and performs the context switch. The agent's overhead (668 ns) sets a theoretical maximum throughput per agent at $10^9/668 = 1.5$M scheduled threads per second. Grouping 10 transactions for different CPUs improves the theoretical maximum to $10 * 10^9/3964 = 2.52$M scheduled threads per second, by amortizing the IPI overhead.

也确实，毕竟local agent和它想调度的cpu是不同的

Given these numbers, a single agent can theoretically schedule roughly 25,200 threads per CPU per second for a 100 CPU server. The agent can keep 100 CPUs busy if threads are 40μs long. Policy developers should keep this per-agent scalability limit in mind as they design ghOSt policies. This limit is improved relatively linearly with more agents.

Global agent scalability (Fig. 5). To show how a global agent scales, we analyze a simple round-robin policy. The policy manages all threads in a FIFO runqueue, scheduling them on CPUs as soon as CPUs become idle. The agent groups as many transactions as possible per commit. We ran

这是在测试global agent的范围能力

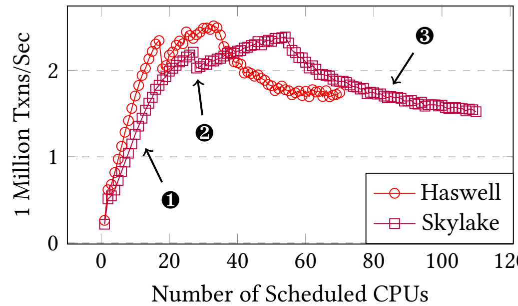


Figure 5. The scalability of a global agent.

the experiment on the default microbenchmarks machine, which uses *Skylake* processors, as well as a 2-socket machine with Haswell processors (18 physical cores per socket, two logical cores each, 2.3 GHz).

The results are shown in Fig. 5. Both lines follow a similar pattern and we annotate the Skylake line in the figure: The steep ramp-up ❶ shows that the global agent schedules more transactions per second as more CPUs are available to schedule work on. The drop ❷ occurs when we co-locate the global agent on the same physical core as a ghOSt thread that executes work. The hyperthreads are contending for resources in the physical core's pipeline, which degrades the global agent's performance. Finally, the degradation ❸ comes from the global agent scheduling CPUs in the remote socket. Scheduling these CPUs requires memory operations and sending interrupts across NUMA sockets which incur higher overheads.

要开始拷打别人以前做的实验了

4.2 Comparison to Custom Centralized Schedulers

We now compare ghOSt, a generic scheduling framework, to a highly specialized scheduling system from recent academic research that uses centralized policies to schedule demanding μs-scale workloads. The experiments run on a single socket from a 2-socket Intel Xeon CPU E5-2658 (12 cores per socket, 24 logical cores per socket, 2.2 GHz).

Systems under comparison. We compared three implementations of the scheduling approach in Shinjuku [25], all serving a RocksDB workload [70]. We use one physical core for load generation with all systems. The Shinjuku system runs on Linux 4.4 as its Dune [35] driver fails to compile for newer versions. The other systems under comparison (ghOSt-Shinjuku and CFS-Shinjuku) run on Linux 4.15 with our ghOSt patches applied.

Dune: Safe user-level access to privileged CP features

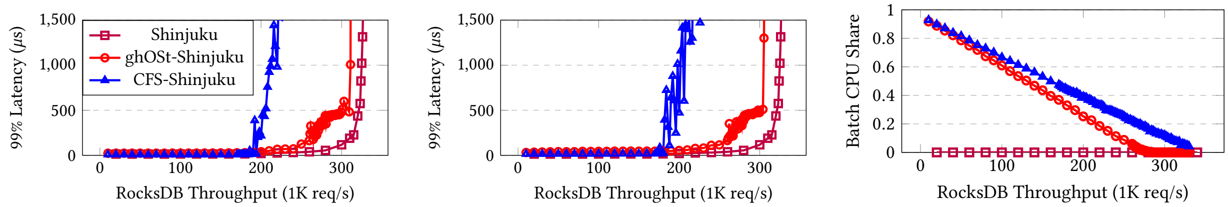
(1) We used the original **Shinjuku** system [25]. It uses 20 spinning worker threads pinned to 20 different hyperthreads and a spinning dispatcher thread, running on a dedicated physical core. The spinning threads prevent *any other thread* from running on their CPUs (Fig. 6c). The dispatcher manages arriving requests in a FIFO and assigns them to worker threads. Each request runs up to a limited runtime, before it is preempted and added to the back of the FIFO. FIFO+抢占式

所以就是分发线程负责把workload发给各个worker是吧

(2) We implemented the Shinjuku scheduling policy in ghOSt using the centralized model in 710 lines of userspace

事实上shinjuku这个策略其实就是我们实现的集中型fifo策略吧！！

throughput越多，latency越大



分散的

(a) Tail latency for dispersive loads. **(b)** RocksDB co-located with a batch app. **(c)** CPU share of the batch app.

Figure 6. ghOSt implements modern μs-scale preemptive policies and shares CPU resources without harming tail latency.

code. The **ghOSt-Shinjuku** global agent starts out on its own physical core but is free to move (§3.3). We maintain a pool of 200 worker threads that the load generator assigns requests to. The global agent maintains a FIFO queue of runnable worker threads and schedules them to the remaining 20 CPUs. Note that ghOSt allows other loads in the system to use any idling CPUs, shown in Fig. 6b-c.

(3) For reference, we also implemented a non-preemptive version of Shinjuku that runs on Linux CFS. This **CFS-Shinjuku** version does not benefit from Shinjuku's specialized data plane features, such as the use of virtualization features and posted interrupts for preemption.

Single Workload Comparison. As in the Shinjuku [25] paper, we generate a workload in which each request includes a GET query to an in-memory RocksDB key-value store [70] (about 6 μs) and performs a small amount of processing. We assigned the following processing times: 99.5% of requests - 4 μs, 0.5% of requests - 10 ms. The allotted time-slice per worker thread, before forcing a preemption and returning back to the FIFO, is 30 μs. CFS-Shinjuku is non-preemptive, so all requests run to completion.

ghost在μs范围内的workload比起shinjuku来说非常competive

Our results are depicted in Fig. 6a. ghOSt is competitive with Shinjuku for μs-scale tail workloads, even though its Shinjuku policy is implemented in 82% fewer lines of code than the custom Shinjuku data plane system. ghOSt has slightly higher tail latencies than Shinjuku at high loads and is within 5% of Shinjuku's saturation throughput. The difference reflects the extra overhead ghOSt has for scheduling a thread for every request, whereas Shinjuku passes request descriptors between spinning threads. CFS-Shinjuku saturates about 30% sooner than the other two systems due to its lack of preemption.

但在高负载情况下，ghost的tail latency就比较高了

Multiple Workloads Comparison. In a production scenario, when RocksDB load is low, it is appealing to use the idling compute resources to serve low-priority batch applications [1, 21, 71] or run serverless functions. The original Shinjuku system schedules requests and cannot manage any other *native* threads. Fig. 6c shows that when we co-locate a batch application with a RocksDB workload managed by Shinjuku, the batch application cannot get *any* CPU resources even when the RockDB load is low.

To enable the safe co-location of low-latency and batch workloads, one might consider using a centralized scheduling system that is thread-oriented, such as Shenango [1]. Shenango's centralized scheduler monitors the load of a network application, and when the app is under light load, the scheduler gives the spare CPU cycles to a batch app. However, Shenango is not suitable for requests with varying execution times, and so the RocksDB workload would have far worse tail latencies than with Shinjuku.

We extended our ghOSt-Shinjuku policy to implement Shenango-style scheduling with merely 17 more lines of code, bringing the policy to 727 lines in total (Shinjuku + Shenango Policy in Table 2). The policy monitors the load to RocksDB and gives spare cycles to the batch app. Fig. 6b shows that our modified ghOSt policy produces the same tail latencies as our original ghOSt policy. The major benefit is shown in Fig. 6c. While keeping the RocksDB tail latencies *intact*, ghOSt now shares spare CPU cycles with the batch app. The amount of compute the batch app can utilize is similar to what it can achieve under CFS when running with a `nice` value of 19, while RocksDB has a `nice` value of -20. A few lines of code in ghOSt combined the best of Shinjuku [25] and Shenango [1], without any application changes.

4.3 Google Snap

We now evaluate ghOSt as a replacement for our soft real-time kernel scheduler MicroQuanta [2] which manages the worker threads for Snap [2], our userspace packet-processing framework.

Similar to DPDK [72], Snap maintains polling (worker) threads that are responsible for interactions with NIC hardware and for running custom networking and security protocols on behalf of important services. Snap may decide to spawn/join worker threads as networking load changes.

与DPDK [72]类似，Snap维护轮询（工作）线程，这些线程负责与网卡硬件的交互，并代表重要服务运行自定义网络和安全协议。当网络负载更改时，Snap可能会决定生成/加入工作线程。

How are worker threads scheduled today? Snap maintains at least one worker thread constantly polling. As bursts of networking load arrive, Snap may wake up and subsequently put to sleep additional worker threads. These frequent wakeups/sleeps require swift scheduler intervention to avoid added latency. Trying to guarantee low latency via existing real-time schedulers, such as `SCHED_FIFO`, destabilizes the system, as it may starve other applications on the same machine. Therefore, we deploy in production MicroQuanta, a custom, soft real-time scheduler that guarantees that for any *period*, e.g., 1 ms, at most a quanta of time, e.g., 0.9 ms, is given to each packet processing worker. This policy ensures worker threads receive runtime while not starving other threads. However, it also leads to networking blackouts of up to 0.1 ms.

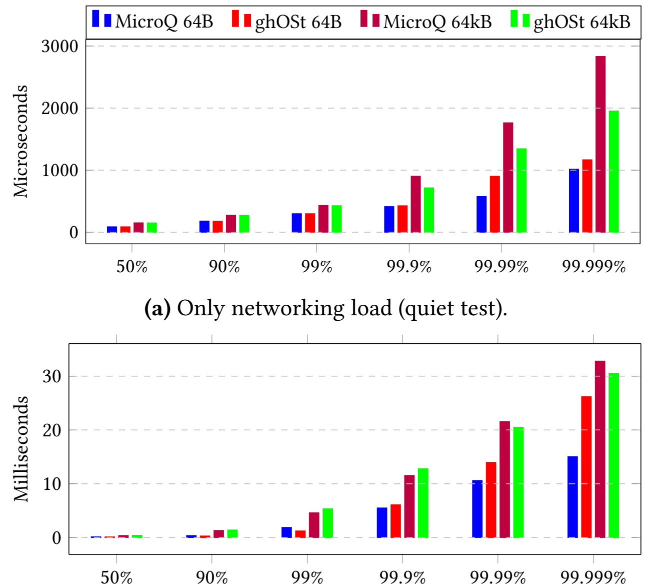
Experimental setup. We used two machines, each with two Intel Xeon Platinum 8173M processors (28 physical cores per socket, 2 logical cores each, 2GHz), 383 GB of DRAM and a 100Gbps NIC and a matching switch. Our tests used a single socket, i.e., 56 logical CPUs per machine.

The test workload. Our test workload is comprised of six client threads, sending 10k messages/second to six server threads on the other machine and receiving a symmetrically sized reply. The test is designed to stress thread scheduling and not the NIC. One client thread sends 64-byte messages, which represents a worst-case scenario rather than a realistic load. Each of the other five client threads sends 64kB messages. The total bandwidth achieved in all cases is 51.86Gbps. In all experiments, the client and server threads are scheduled by Linux CFS. In our ghOSt experiments, the worker threads are scheduled with ghOSt instead of MicroQuanta.

We run tests in two modes. In the *quiet* mode, the client/server threads are the only explicit workload on their machines. In the *loaded* mode, the machines are also running 40 additional antagonist threads of a batch workload that attempt to use idling CPU resources, unused by server or Snap threads, when network traffic is low.

ghOSt policy. We replace MicroQuanta with the following ghOSt policy. The policy manages the worker threads of Snap and the antagonist threads. It is a simple, yet effective centralized FIFO policy. The global agent tries to find an idle CPU to schedule its threads, giving Snap worker threads strict priority over antagonist threads. In quiet mode tests, the worker threads are frequently preempted by the client/server threads and other native threads scheduled by CFS (e.g., important but periodic daemons). In loaded mode tests, Snap worker threads will preempt antagonist threads, but cannot preempt threads managed by CFS. Antagonist threads only run when there are spare resources from both CFS threads and Snap. We *did not* use any dedicated cores.

Tail latency comparison. Fig. 7 compares the round-trip tail latencies for client requests given the two schedulers in the machine. We present tail latencies separately for 64B and 64kB messages. For 64B messages, ghOSt performs similar or 10% better than the baseline when we consider up to 99.9th percentile latency. For 99.99% and above, ghOSt latencies are up to 1.7× worse. The 64B messages require little compute for packet processing. Hence, when traffic is bursty with many 64B messages, ghOSt's overhead of scheduling events is noticeable . For 64kB messages, ghOSt performs similarly to the baseline for up to the 99th percentile latency (within 15% in either direction). For 99.9th percentile and above, ghOSt leads to tail latencies that are 5% to 30% lower. The 64kB messages require more processing (for copying data) and therefore lead to fewer scheduling events. The reason ghOSt performs better than MicroQuanta in some cases is that it can relocate a worker thread when a CPU becomes busy due to a server process thread. MicroQuanta, on the other hand, has to wait for a blackout period.



(a) Only networking load (quiet test).

(b) With additional load (loaded test).

Figure 7. Google Snap latencies for 64B and 64kB messages.

These results are very encouraging. A very simple ghOSt policy performs similarly to a custom kernel scheduler without needing to modify Snap. ghOSt allows rapid experimentation and performance optimization, which is substantially harder with kernel schedulers. We expect that additional improvements, such as including scheduling hints from the worker threads, can further optimize performance.

4.4 Google Search

We now evaluate ghOSt as a replacement for the CFS scheduler on machines that serve Google Search queries [73].

The test workload. The benchmark includes a query generator running on a separate machine (without ghOSt), sending three different query types, denoted `A`, `B`, and `C`. Query type `A` is a CPU and memory-intensive query serviced by worker threads which are woken up as needed. Query type `B` needs little computation but does require access to the SSD, and is serviced by a collection of short-lived workers, also woken up as needed. Query type `C` is a CPU-intensive load serviced by long-living worker threads.

At packet ingress, all queries are first processed by one of several server threads, which create sub-queries to be processed by the worker threads referenced above. Some subqueries must be processed by specific worker threads tied to a NUMA node to take advantage of data locality. Since this workload is memory and CPU bound, queries are serviced much faster if they are handled by worker threads running on the same socket where the data they access reside.

Experimental setup. The machines we evaluated ghOSt on have two AMD Zen Rome processors with 256 CPUs in total (2 sockets, 64 physical cores per socket, 2 logical cores each). AMD's architecture brings new challenges because it clusters groups of 4 physical cores (8 logical cores) into CCXs (CPU Core Complexes), where each CCX has its own L3 cache.

ghOSt policy. We implement a policy using ghOSt's centralized model with a single global agent to schedule all 256 CPUs. At startup, the global agent first generates a model of the system topology, using `sysfs`. The global agent then uses the topology information to schedule threads based on their NUMA preferences and, when possible, prefers running the threads on a CPU belonging to the last CCX the threads ran on. The global agent maintains a min-heap ordered by thread runtime, where threads with the least elapsed runtime are picked for execution before others. Threads run to completion or until preempted by a CFS thread.

The NUMA- and CCX-aware heuristics for picking the next idle CPU are only 57 lines of code due to ghOSt's flexible transaction API and use of the C++ standard library. When a new worker thread is spawned, its cpumask is set (via `sched_setaffinity()`) to the set of CPUs in the socket where its query data is located. This cpumask is included as part of the `THREAD_CREATED` message that is sent to the global agent. When the ghOSt global agent wants to run the next thread at the front of its runqueue, it intersects the thread's cpumask with the set of idle CPUs. If the intersection is empty, the agent skips the thread and schedules the next thread in the runqueue, revisiting the skipped thread in the next iteration of its scheduling loop.

缓存预热
就是系统启动前，提前将相关的缓存数据直接加载到缓存系统。避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

Search query performance improves when each thread runs on CPUs with warmed-up L1, L2, or L3 caches. For each thread-scheduling event, our ghOSt policy assigns the thread to an idle CPU target that is closest to where the thread last ran. The policy first searches for available CPUs within the same L1 and L2 cache domain of the CPU where the thread last ran. If no idle CPUs are found, the policy extends the search to the CCX (L3 cache) domain. If this fails, too, then it does a fan-out search for the nearest neighbor of the CCX where the thread last ran, to avoid expensive thread migration costs due to high inter-CCX communication latencies.

CFS vs. ghOSt. Fig. 8 compares normalized query latency and throughput for the search benchmark using CFS and the ghOSt policy over a period of 60 seconds. Fig. 8*a*-*c* shows that ghOSt offers comparable throughput to CFS. Both CFS and ghOSt consider NUMA socket and CCX placement. The NUMA and CCX optimizations were critical in achieving parity with CFS as they delivered 27% and 10% throughput improvements, respectively. Iteratively optimizing for NUMA and CCX placement in a short ghOSt policy is much easier than experimenting with changes in the kernel CFS code. Every modification to the ghOSt agent requires merely restarting the agent's process, whereas any modification to CFS would mandate a kernel installation and a reboot.

Tail Latency. Fig. 8*d*-*f* shows that ghOSt leads to about 40-45% reduction in tail latency for query types A and B, compared to CFS, and comparable tail latency for query type C. Prior to socket- and CCX-aware optimizations, the ghOSt policy led to nearly 2x worse latency for query type A and was on par with CFS for query type B and C (i.e., within 10%). Query type A is memory-bound and benefited the most from topology optimizations. Query type B accesses both memory and SSD, while query type C is mostly compute-bound. For B and C, the ghOSt policy had an advantage to begin with as the global agent spins and reacts quickly to changes in capacity in the whole system, rebalancing threads across CPUs on the order of *microseconds*. CFS on the other hand only rebalances threads across CPUs at periodic intervals on the order of *milliseconds*, harming query tail latencies.

We can improve query C's latency on ghOSt by further refining our policy. The workload assigns `nice` values to threads to express relative priority ordering to CFS, which is important for ensuring that worker threads run with higher priority than low-priority background threads (e.g., for garbage collection). CFS makes more optimal decisions by using these `nice` values, and initial experiments show that incorporating them into ghOSt's policy will allow ghOSt to beat CFS for query C's tail latency.

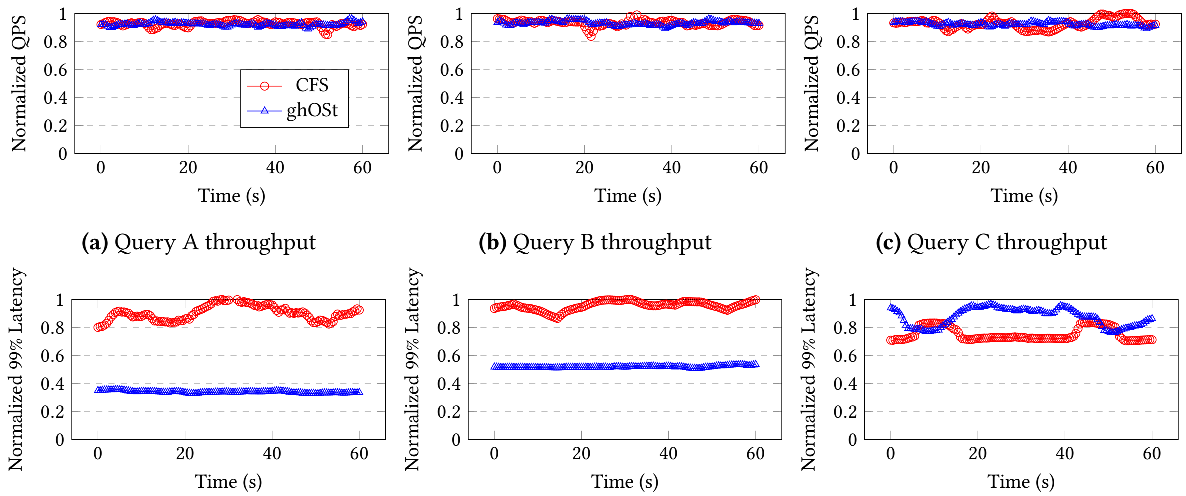
Our experience with rapid experimentation. ghOSt is a viable solution for our production fleet, capable of scheduling for large machines and realistic workloads, while enabling rapid development and roll-out of scheduling policies.

When developing a kernel scheduler, the write-test-write cycle includes *(a)* compiling a kernel (up to 15 minutes), *(b)* deploying the kernel (10-20 minutes), and *(c)* running the test (1 hour due to database initialization following a reboot). As a result, the enthusiastic *kernel developer* experiments with 5 variants per day. With ghOSt, compiling, deploying and launching the new agent is comfortably done within *one minute*. In fact, due to ghOSt's ability to upgrade without a reboot, the test continues to run uninterrupted. This also has an important secondary effect, as the scale and complexity of this test can contribute non-trivial run-to-run variance across a reboot, confounding optimization development.

This ease of experimentation allowed us to experiment with bespoke optimizations, which would be extremely difficult to discover otherwise. For example, due to high variance in intra-CCX and inter-CCX latencies in the Rome architecture, we found that if a thread's preferred CCX cluster is unavailable, it is more efficient to temporarily keep the thread pending for 100 μs rather than migrate it to another CCX immediately.

4.5 Protecting VMs from L1TF/MDS Attacks

We now evaluate a ghOSt policy protecting virtual machines from cross-hyperthread speculative execution attacks, such as L1TF and MDS [27–32]. In these attacks, a malicious VM exploits microarchitectural flaws to steal data from *a different VM* running on a sibling hyperthread. The attacks are mitigated by ensuring every physical core only runs virtual



(a) Query A throughput **(b)** Query B throughput **(c)** Query C throughput

(d) Query A 99% tail latency **(e)** Query B 99% tail latency **(f)** Query C 99% tail latency

Figure 8. Google Search benchmark results with CFS and ghOSt scheduling.

CPUs (vCPUs) from *the same VM*. Microarchitectural buffers are flushed when a new VM is scheduled on a physical core.

Mitigating cross-hyperthread attacks requires scheduling physical cores, with two logical CPUs each. Implementing core scheduling in a per-CPU model is challenging since the scheduler code can only run threads on the CPU it is currently executing on. In contrast, a ghOSt agent can easily schedule an entire core by performing a synchronized group commit for each physical core, i.e., issuing commits for both CPUs of a core which must either all succeed or all fail.

Our per-core scheduling model is illustrated in Fig. 9. In every core, both sibling CPUs post messages to the same queue. At any time, only a single agent is active on a given *physical* core. The CPU that generates the message wakes up its corresponding agent to be the active agent; the other agent becomes inactive if necessary. The active agent makes a scheduling decision for both CPUs and commits a group transaction in a way such that there are no pending messages in the queue, as explained in §3.2. In this per-physical-core scheduling model, the ghOSt policy ensures the threads running on the physical core belong to the same VM. The VM's threads may occupy (a) both siblings or (b) only one sibling while the other sibling runs an idle thread.

Secure VM Core Scheduling Policy. We implement a VM-scheduling policy similar to Tableau [23] in both the kernel and in ghOSt. We designed this policy to ensure forward progress, bound tail latency, and provide good average latency among all VMs. The policy ensures the former two by scheduling each runnable VM thread for c units of time every period p. Specifically, we use a partitioned EDF scheme wherein each physical core dedicates guaranteed time for each thread to run, bounding tail latency. Any excess time is shared fairly among runnable threads, improving average latency. The runqueues span a single NUMA node; when a physical core goes idle and looks for a new thread to run, it prefers to select a thread in its NUMA-local runqueue. However, under high system load the policy allows spilling across runqueues, providing NUMA preference without the hard boundary imposed by a CPU affinity mask.

Evaluation. The experimental setup is identical to §4.3. We scheduled 32 vCPUs on 25 physical cores with 50 logical CPUs. We ran the *bwaves* benchmark from SPECCPU 2006, with three scheduling policies: 1) CFS, providing no security against speculative execution attacks; 2) In-kernel secure VM core scheduling; and 3) ghOSt secure VM core scheduling. The results are depicted in Table 4. CFS provides better overall performance, but no security. The ghOSt policy mitigates cross-hyperthread attacks and performs similar to the in-kernel version of core scheduling, even in light of the additional context switching overheads in ghOSt.

5 Future Work

Accelerating scheduling with BPF. Accelerating ghOSt by delegating some of the agent responsibilities to synchronous BPF callbacks is an open research area. The global agent scheduling loop in §4.4 takes 30 μs, creating potential scheduling gaps. Indeed, some of the threads in our system run for only 5-30 μs before they block, leaving CPUs idle during these gaps. We can mitigate these scheduling gaps using an integrated BPF program, described in §3.2.

The BPF program communicates with userspace via shared memory with several multi-producer, multi-consumer ring buffers. The agent inserts runnable threads into the buffers and BPF tries to run them. The agent may revoke a thread before BPF can schedule the thread. For example, the global agent can use one ring buffer per NUMA node; the global agent can then track each thread's preferred NUMA node and load-balance the threads between the two rings. 有一说一确实

Tick-less scheduling. When ghOSt is in centralized mode, timer ticks can be disabled across CPUs to avoid expensive VM-exits in VM workloads. In a classic per-CPU

QS，global agent线程CPU的时钟中断可以直接关了

Scheduling Policy	*bwaves* Rate	Total Time
CFS (no security)	489	888 seconds
In-kernel Core Scheduling	464	937 seconds
ghOSt Core Scheduling	468	929 seconds

Table 4. Secure VM Core Scheduling performance. SPEC-CPU 2006 bwaves, scheduling 32 vCPUs on 50 real/logical CPUs. *Rate* - higher is better. *Time* - lower is better.

scheduler, the ticks trigger the scheduler every millisecond to ensure round-robin preemption across all VMs. Unfortunately, these ticks cause a VM-exit to host kernel context.

这边关于为什么ghost可以关中断、CFS不能关中断我没太看懂

Since the global agent is continuously spinning and making scheduling decisions, there is no need for these ticks. Eliminating these ticks across all CPUs will substantially reduce guest jitter. This type of optimization is not possible with CFS. The closest option in CFS is to enable `CONFIG_NO_HZ_FULL`, but that will only disable ticks when there is no more than one runnable thread on a given CPU, which is typically not the case under high utilization.

6 Related Work

We briefly discuss additional prior work related to ghOSt.

这个东西提供了API，API可以让用户程序控制它运行的CPU的调度

Scheduler Activations and user-level threading. Scheduler Activations [74] provides an API for an individual application to coordinate the scheduling of CPUs assigned to it by the kernel. In contrast to ghOSt, Scheduler Activations allows an application to react to the assignment or removal of a CPU, but it does not allow the assignment of CPUs to applications. One may use Activations to synchronize an application with ghOSt scheduling decisions. Similarly, userspace thread libraries and schedulers [13, 22, 48, 52–57, 59, 60, 75] multiplex userspace contexts on top of kernel threads but do not control when or where kernel threads run.

SmartNIC scheduling. Recent work explores the right policies and mechanisms to offload scheduling and applications from the host to a SmartNIC [19, 76–78]. ghOSt's shared-memory queue and transaction APIs were designed to work seamlessly with new coherent interconnect technologies such as CXL [79], allowing ghOSt to be offloaded in part or in full to a SmartNIC.

Return of microkernels. An emerging trend is offloading kernel components to userspace, similar to microkernels [80]. DPDK [72], IX [34], and Snap [2] offload network drivers and stacks to userspace. SPDK [81], ReFlex [82], FUSE [83], and DashFS [63] offload storage and file system operations. Linux Userspace I/O (UIO) [84] facilitates offload of kernel drivers. A new CPU design has even been proposed to accelerate microkernels [85]. ghOSt continues this trend.

Prior work also suggested moving the scheduler to userspace. Stoess developed a hierarchical, user-level scheduler for the L4 microkernel [86]. However, unlike ghOSt, Stoess requires modifying applications to implement the custom scheduling policies.

CPU Inheritance Scheduling. Ford and Susarla's *CPU Inheritance Scheduling* [87] is a user-level scheduling system

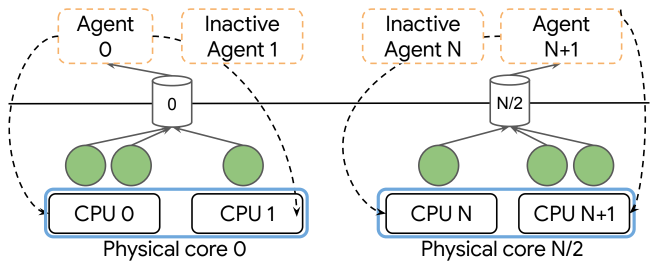


Figure 9. ghOSt per-core scheduling. Each couple of sibling-CPUs shares a single message queue. For each core, only a single agent is active at a time.

where scheduling is done by threads *donating* their runtime. The donation model is less efficient and less expressive than ghOSt's transactions. Each CPU's *root scheduler* would donate its runtime to another thread. That thread could also be a scheduler, allowing for a hierarchy of schedulers, or an application thread — elegantly solving priority inversion. These schedulers, like ghOSt agents, received messages about thread and system events. The system's *dispatcher*, analogous to ghOSt kernel code, handled mechanism: executing donations and sending messages. A donation is akin to a ghOSt transaction to run the local CPU, but it cannot schedule remote CPUs quickly. To do so, it must wake a scheduler on the remote CPU and have it donate. Further, unlike ghOSt, each scheduler can only schedule a single CPU at a time.

7 Conclusion

We presented ghOSt, a new platform for evaluating and implementing thread scheduling policies for the modern data center. ghOSt transforms scheduler development from the realm of monolithic kernel implementations to a more flexible userspace setting, allowing a wide range of programming languages and libraries to be applied. While intuitively, moving scheduling decisions to userspace might suggest excessive coordination overhead, we have minimized synchronous costs in our APIs and our characterization shows most operations to be circa μs. What previously required extensive effort to optimize and deploy can now be often realized in less than a thousand lines of code. ghOSt allowed us to quickly develop policies for our production software — rapidly testing and iterating — leading to competitive performance compared to the status quo schedulers. We open-source ghOSt to serve as the basis of future lines of research for the new era of user-driven resource management.

Acknowledgments

We thank Eric Brewer, David Culler, Hank Levy, Amin Vahdat, Kostis Kaffes, Adam Belay, Josh Fried, David Mazières, our shepherd Irene Zhang, Google Systems Infrastructure, Stanford Platform Lab, and the anonymous SOSP reviewers for their helpful feedback. Christos Kozyrakis was supported by Stanford Platform Lab.

References

[1] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[2] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *In ACM SIGOPS 27th Symposium on Operating Systems Principles*, New York, NY, USA, 2019.

[3] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. Elastic rss: Co-scheduling packets and cores using programmable nics. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, APNet '19, page 71–77, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Amy Ousterhout, Adam Belay, and Irene Zhang. Just in time delivery: Leveraging operating systems knowledge for better datacenter congestion control. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[5] Esmail Asyabi, Azer Bestavros, Renato Mancuso, Richard West, and Erfan Sharafzadeh. Akita: A cpu scheduler for virtualized clouds, 2020.

[6] Esmail Asyabi, Azer Bestavros, Erfan Sharafzadeh, and Timothy Zhu. Peafowl: In-application cpu scheduling to reduce power consumption of in-memory key-value stores. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 150–164, New York, NY, USA, 2020. Association for Computing Machinery.

[7] *SCHED(7) Linux Programmer's Manual*, September 2020.

[8] Weiwei Jia, Jianchen Shan, Tsz On Li, Xiaowei Shang, Heming Cui, and Xiaoning Ding. vsmt-io: Improving i/o performance and efficiency on SMT processors in virtualized clouds. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 449–463. USENIX Association, July 2020.

[9] Tim Harris, Martin Maas, and Virendra J. Marathe. Callisto: Co-scheduling parallel runtime systems. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[10] 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, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[11] Kostis Kaffes, Dragos Sbirlea, Yiyan Lin, David Lo, and Christos Kozyrakis. Leveraging application classes to save power in highly-utilized data centers. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 134–149, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Trans. Comput. Syst.*, 34(4), December 2016.

[13] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, Carlsbad, CA, October 2018. USENIX Association.

[14] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, April 2014. USENIX Association.

[15] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 79–94, Boston, MA, February 2019. USENIX Association.

[16] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 158–164, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1225–1240. USENIX Association, November 2020.

[18] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, Renton, WA, July 2019. USENIX Association.

[19] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. Mind the gap: A case for informed request scheduling at the nic. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 60–68, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[21] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[22] Barret Rhoden, Kevin Klues, David Zhu, and Eric Brewer. Improving per-node efficiency in the datacenter with new os abstractions. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[23] Manohar Vanga, Arpan Gujarati, and Björn B. Brandenburg. Tableau: A high-throughput and predictable vm scheduler for high-density workloads. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[24] Performance-driven dynamic resource management in e2 vms. https://cloud.google.com/blog/products/compute/understanding-dynamic-resource-management-in-e2-vms. Last accessed: 2020-11-11.

[25] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association.

[26] Amirhossein Mirhosseini, Brendan L. West, Geoffrey W. Blake, and Thomas F. Wenisch. Q-zilla: A scheduling framework and core microarchitecture for tail-tolerant microservices. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 207–219, 2020.

[27] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical report*, 2018. See also USENIX Security paper Foreshadow [28].

[28] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD,

August 2018. USENIX Association.
[29] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 753–768, New York, NY, USA, 2019. Association for Computing Machinery.
[30] Zombieload: Cross privilege-boundary data leakage. https://www.cyberus-technology.de/posts/2019-05-14-zombieload.html. Last accessed: 2020-09-02.
[31] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105, 2019.
[32] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, and Yuval Yarom. Fallout: Reading kernel writes from user space. *CoRR*, abs/1905.12701, 2019.
[33] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: A decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.
[34] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, 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, Broomfield, CO, October 2014. USENIX Association.
[35] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, October 2012. USENIX Association.
[36] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. Mittos: Supporting millisecond tail tolerance with fast rejecting slo-aware os interface. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 168–183, New York, NY, USA, 2017. Association for Computing Machinery.
[37] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Redline: First class support for interactivity in commodity operating systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 73–86, USA, 2008. USENIX Association.
[38] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, October 2018. USENIX Association.
[39] big.little - arm. https://www.arm.com/why-arm/technologies/big-little. Last accessed: 2020-11-27.
[40] Kevin Lepak. The next generation amd enterprise server product architecture. In *Proceedings of the IEEE Annual Symposium on Hot Chips*, August 2017.
[41] Aws nitro system. https://aws.amazon.com/ec2/nitro/. Last accessed: 2020-11-29.
[42] What's a dpu? https://blogs.nvidia.com/blog/2020/05/20/whats-a-dpu-data-processing-unit/. Last accessed: 2020-11-29.
[43] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, and Jonathan Ross. In-datacenter performance analysis of a tensor processing unit. 2017.
[44] ghOSt kernel code. https://github.com/google/ghost-kernel.
[45] ghOSt userspace code. https://github.com/google/ghost-userspace.
[46] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.
[47] Dario Faggioli, Michael Trimarchi, Fabio Checconi, Marko Bertogna, and Antonio Mancina. An implementation of the earliest deadline first algorithm in linux. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, page 1984–1989, New York, NY, USA, 2009. Association for Computing Machinery.
[48] Folly: Facebook open-source library. https://github.com/facebook/folly. Last accessed: 2020-11-10.
[49] Abseil. https://abseil.io/. Last accessed: 2020-11-29.
[50] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf [Viewed October 15, 2004].
[51] Dick Sites. Data center computers: Modern challenges in cpu design. https://www.youtube.com/watch?v=QBu2Ae8-8LM. Last accessed: 2020-11-10.
[52] P. Gadepalli, R. Pan, and G. Parmer. Slite: Os support for near zero-cost, configurable scheduling *. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 160–173, Los Alamitos, CA, USA, apr 2020. IEEE Computer Society.
[53] Martin Karsten and Saman Barghi. User-level threading: Have your cake and eat it too. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(1), May 2020.
[54] The go programming language. https://golang.org/. Last accessed: 2020-11-10.
[55] Boost fiber. https://www.boost.org/doc/libs/1_68_0/libs/fiber/doc/html/fiber/overview.html. Last accessed: 2020-11-10.
[56] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for internet services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 268–281, New York, NY, USA, 2003. Association for Computing Machinery.
[57] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, page 207–216, New York, NY, USA, 1995. Association for Computing Machinery.
[58] Kevin Alan Klues. *OS and Runtime Support for Efficiently Managing Cores in Parallel Applications*. PhD thesis, University of California, Berkeley, USA, 2015.
[59] Lithe. http://lithe.eecs.berkeley.edu/. Last accessed: 2020-11-10.
[60] Seastar. https://github.com/scylladb/seastar. Last accessed: 2020-11-10.
[61] A thorough introduction to ebpf. https://lwn.net/Articles/740157/. Last accessed: 2020-12-10.
[62] Yu Jian Wu, Hongyi Wang, Yuhong Zhong, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. Bpf for storage: An exokernel-inspired

approach. In *Proceedings of the 18th ACM Workshop on Hot Topics in Operating Systems*, HotOS '21. Association for Computing Machinery, 2021.

[63] Jing Liu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Sudarsun Kannan. File systems as processes. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[64] How esxi numa scheduling works. https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.resmgmt.doc/GUID-BD4A462D-5CDC-4483-968B-1DCF103C4208.html. Last accessed: 2020-11-02.

[65] Bpf ring buffer. https://nakryiko.com/posts/bpf-ringbuf. Last accessed: 2021-04-26.

[66] The rapid growth of io_uring. https://lwn.net/Articles/810414/. Last accessed: 2021-04-26.

[67] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, page 204–213, New York, NY, USA, 1995. Association for Computing Machinery.

[68] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[69] libbpf. https://github.com/libbpf/libbpf. Last accessed: 2020-08-25.

[70] Rocksdb. https://rocksdb.org. Last accessed: 2020-11-27.

[71] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In Deborah T. Marr and David H. Albonesi, editors, *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pages 450–462. ACM, 2015.

[72] Data plane development kit. http://www.dpdk.org/. Last accessed: 2019-06-26.

[73] Luiz Andre Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23:22–28, 2003.

[74] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 95–109, New York, NY, USA, 1991. Association for Computing Machinery.

[75] Upthread. http://akaros.cs.berkeley.edu/parlib/upthread/. Last accessed: 2020-11-10.

[76] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for nic-accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, Carlsbad, CA, October 2018. USENIX Association.

[77] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[78] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis. The nebula rpc-optimized architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 199–212, 2020.

[79] Compute Express Link https://docs.wixstatic.com/ugd/0c1418_d9878707bbb7427786b70c3c91d5fbd1.pdf. Last accessed: 2019-06-26.

[80] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 251–266, New York, NY, USA, 1995. Association for Computing Machinery.

[81] Storage performance development kit. https://spdk.io/. Last accessed: 2020-08-20.

[82] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash ≈ local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 345–359, New York, NY, USA, 2017. Association for Computing Machinery.

[83] libfuse. https://github.com/libfuse/libfuse. Last accessed: 2020-08-15.

[84] The userspace i/o howto. https://www.kernel.org/doc/html/v4.14/driver-api/uio-howto.html. Last accessed: 2021-01-11.

[85] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. *A Case against (Most) Context Switches*, page 17–25. Association for Computing Machinery, New York, NY, USA, 2021.

[86] Jan Stoess. Towards effective user-controlled scheduling for microkernel-based systems. *ACM SIGOPS Oper. Syst. Rev.*, 41(4):59–68, 2007.

[87] Bryan Ford and Sai Susarla. CPU inheritance scheduling. In Karin Petersen and Willy Zwaenepoel, editors, *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, Washington, USA, October 28-31, 1996*, pages 91–105. ACM, 1996.