eBPF-based Extensible Paravirtualization

Luigi Leonardi, Giuseppe Lettieri, and Giacomo Pellicci

University of Pisa, Pisa, Italy
luigi.leonardi@phd.unipi.it, giuseppe.lettieri@unipi.it,
pellicci.giac@gmail.com

Abstract. High performance applications usually need to give many hints to the OS kernel regarding their needs. For example, CPU affinity is commonly used to pin processes to cores and avoid the cost of CPU migration, isolate performance critical tasks, bring related tasks together, and so on. However, when running inside a Virtual Machine, the (guest) OS kernel can only assign virtual resources, e.g., pinning a guest process to a virtual CPU thread (vCPU); the vCPU thread, however, is still freely scheduled by the host hypervisor, which is unaware of the guest application requests. This *semantic gap* is usually overcome by statically allocating virtual resources to their hardware counterparts, which is costly and inflexible, or via *paravirtualization*, i.e., by modifying the guest kernel to pass the hints to the host, which is cumbersome and difficult to extend. We propose to use host-injected eBPF programs as a way for the host to obtain this kind of information from the guest in an extensible way, without modifying the guest (Linux) kernel, and without statically allocating resources. We apply this idea to the example of CPU affinity and run some experiments to show its effect on several microbenchmarks. Finally, we discuss the implications for confidential computing.

Keywords: eBPF · Paravirtualization · Virtualization · CPU Pinning

1 Introduction

Performance-critical applications often cannot rely on the resource scheduling decisions of general purpose kernels. Kernels have evolved to meet the special requirements of these applications by either accepting resource-usage hints (e.g., via the `madvice()` system call), providing means to let the applications override the kernel's scheduling decisions (e.g., in CPU pinning [4, 6] or IRQ affinity [5]) or implementing ways to completely bypass some of the kernel's abstractions [10]. The unifying idea of these advanced kernel APIs is that applications may benefit from direct access to hardware resources, with as little kernel intervention as possible. However, inside a Virtual Machine, the guest kernel itself has no direct access to the host hardware, and the traditional implementation of these APIs may not achieve the intended effect.

The issue is particularly clear for CPU pinning. Applications pin their threads to specific CPU cores for a number of performance-related reasons: avoid the

cost of CPU migration, isolate performance critical tasks, bring related tasks together, or place them closer to critical hardware resources. When running in a virtual machine, however, the guest kernel is only able to pin application threads to *virtual* CPUs, which are usually implemented as software threads scheduled by the VM hypervisor. The hypervisor is still free to migrate the CPU threads among available hardware cores or to time-share several threads on a single core. Moreover, virtual resources that look "close" to the guest (e.g. hyperthreads of the same virtual core) may still be scheduled on "far" hardware resources (e.g., on two separate cores). This behaviour effectively negates most or all the performance gains that the guest applications where trying to achieve.

The problem is that the relevant information is split between two distinct subsystems: the application needs are only known by the guest kernel, while the hardware resource state is only known to the hypervisor. The traditional way to address this problem is to partly remove the distinction by virtual and hardware resources, e.g., by statically pinning each virtual CPU thread to an isolated hardware CPU thread. This solution, while effective, may be unnecessarily costly, if the application workload is subject to changes. A different, more dynamic approach is suggested by Lee and Eom [7]: here, suitable hypercalls are inserted in the guest kernel to pass the scheduling decisions down to the hypervisor, which will then apply them on the host system. This is an example of *paravirtualization* [2], where the guest system is made aware of running inside a virtual machine.

The typical way to achieve paravirtualization, as also exemplified here, is to modify the guest kernel in an *ad hoc* way to solve the particular problem of interest. This is unsatisfying for several reasons: modifying the kernel is a complex undertaking; new modifications must be devised whenever new needs arise; the choices of guest software may be subject to limitations (e.g., particular kernel versions where the modification is available).

In recent years, eBPF [9] has emerged as a generic solution to extract information, or customize the behaviour of unmodified Linux kernels, and work to support it in Windows is also underway [3]. Using the eBPF framework, userspace applications can inject programs in the running kernel and attach them to specific tracepoints. The programs, run by the kernel whenever the tracepoint is reached, have controlled access to kernel data structures and can pass informations to interested userspace programs. In this paper we propose to reuse eBPF technology for paravirtualization purposes, i.e., by having the virtual machine monitor inject eBPF programs in the guest kernel, in order to extract any information that may be relevant for the efficient, dynamic allocation of hardware resources based on guest application requests.

2 eBPF

eBPF is a flexible and efficient technology, available in the Linux kernel, that is composed of an instruction set, storage objects, and helper functions. Its

instructions are executed by a Linux kernel BPF runtime, which includes an interpreter and a JIT[1] compiler for efficiency.

This technology enables dynamic tracing: the ability to insert tracing points into live software, and costs zero overhead when not in use, as software runs unmodified.

An eBPF program is "attached" to a designated code path in the kernel. When the code path is traversed, any attached eBPF programs are executed. Code paths can be of various kind, allowing programs to be attached to tracepoints, kprobes, and perf events. Since eBPF programs can access kernel data structures, developers can write and test new debugging code without having to recompile the kernel.

One really important aspect of eBPF is that it is verifiable: There are inherent security and stability risks with allowing user-space code to run inside the kernel. For this reason, there are some limitations to the instruction set, like loops. The verifier has an important role and is asked to detect the eBPF program type, to restrict which kernel functions can be called from eBPF programs and which data structures can be accessed.

3 Extensible Paravirtualization

This work aims to create a generic mechanism that allows to have a certain degree of paravirtualization in a system that uses hardware-assisted virtualization. This must be done transparently, meaning that the guest operating system should not be modified.

The approach that has been followed is the following: A remote client injects a payload into the virtual device on the host side, and the guest daemon, through the device driver regulating the access to the device, obtains such message and consumes it accordingly to its content. Note that the payload content might be whatever, from a simple command to a more complex payload, for instance, an eBPF program containing a kprobe ready to be loaded into the guest kernel. Any information obtained by this eBPF program might be used inside or outside the guest system and possibly being propagated up to host hypervisor or even to the client that initiated the communication.

The proposed architecture, shown in Figure 1, is simple and linear. This choice was made to favor integration with existing technologies such as QEMU and the Linux kernel. Note that no specifications on the communication channel are given, meaning that can be either simplex or duplex. In the general case, it is assumed that this communication is duplex. Although a QEMU guest agent exists, it has not been used in this work and has been re-implemented for simplicity and to better fit this project's needs.

To summarize a system that allows communication between host and guest has been created. This communication happens through messages structured as header and payload, whose meaning is associated with their message type.

[1] Just in Time

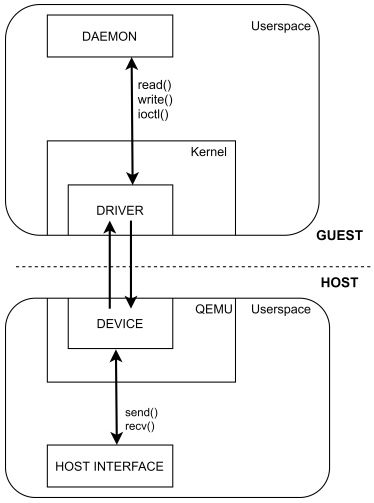

Fig. 1. Generic extensible paravirtualization mechanism architecture

4 Virtual to Physical CPUs Affinity

CPU affinity, also known as CPU pinning, is a technique that allows to set the affinity of a process or thread with a set of CPUs. By doing so the process or thread will only run on the designated CPU(s), ignoring the others. The main advantage of pinning a process or a thread to one single CPU is that it cannot move to other CPUs, never losing the content of its cache as it happens when a process or thread is moved to another CPU. There are several reasons why a program might want to control this aspect of the system:

- Considering a pair of coupled processes or threads like in a typical producer-consumer system: If the communication protocol is non-blocking, there are some performance benefits when two processes or threads are running on different CPUs. Note that, in the default way in which Linux handles it, no guarantees are given about the two sharing the same CPU, while the other CPUs are idle. Even though this is a rare event but may happen, on the other hand with CPU affinity is possible to ensure that two processes or threads are never scheduled on the same CPU.
- In NUMA[2] machines accessing resources like RAM or I/O has different costs from different CPUs. Forcing a process or thread on the CPU that has "local" access to the most used memory zones can have beneficial effects on performance. This last case is not covered in this work.

On Linux, the CPU affinity of a process can be altered with the *taskset* program, while the *sched_setaffinity* system call can be used to modify the CPU affinity of a process or thread.

[2] Non-Uniform Memory Architecture

Note that the usage of CPU affinity to boost performance depends heavily on the program structure. Different benefits can be achieved depending on whether the program is CPU bound or if it makes extensive use of cache.

In this work, the environment is composed of a host system and one or more virtual machine(s) that are called guest(s). Inside the guest, an operating system will manage the underlying virtualized hardware and provide system calls to the guest userspace applications. The latter system can be started with one or more CPUs that are virtual, or vCPUs for short, and in reality are host userspace threads which in this case are created by QEMU. Setting the CPU affinity inside the guest operating system, that binds a guest userspace thread to a guest CPU, means nothing because the latter is just a representation of a physical CPU. In other words it is binding the guest thread to the host thread that represent the guest virtual CPU.

It would be useful in some applications to be able to apply CPU affinity also outside the guest system, resulting in a real CPU pinning.

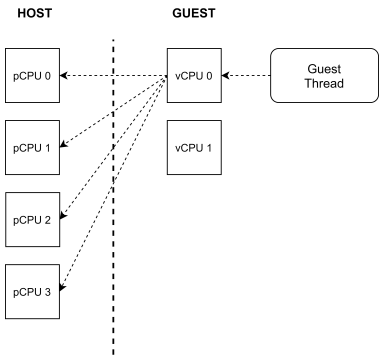

Fig. 2. Dashed arrows represent affinity. Virtual CPUs are threads in the host system and therefore they can have their affinity with host physical CPUs

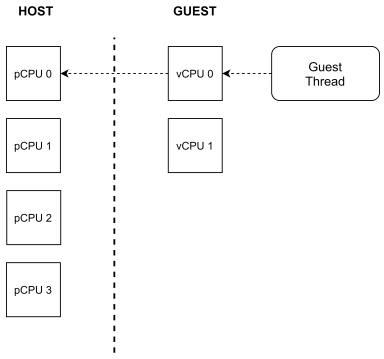

Fig. 3. Dashed arrow represent affinity. A guest thread requires to be executed only on vCPU 0 and the request is correctly applied also in the host system, on pCPU 0

4.1 Implementation

The general idea is to have an eBPF program that will track calls to the *sched_setaffinity* system calls and notify the guest agent. This agent will then send to the QEMU virtual device the requested affinity masks through the device driver. Once the affinity masks arrive in the QEMU virtual device, they can be used to call *sched_setaffinity* on the host side. Affinity mask remapping can be enabled from the host side depending on whether Hyper-threading is supported, sending a specific message which has been defined for this purpose.

The aforementioned eBPF program, which contains a Kprobe to be installed on *sched_setaffinity()*, is loaded into the kernel. Then, whenever the probed function is called, the eBPF program will fire and the affinity mask, which is provided as an argument, is written in an eBPF map.

In addition if the device is enabled for affinity mask remapping, a remapping is then performed, otherwise, the unmodified affinity mask is applied on the QEMU threads that represent the virtual CPUs that the guest uses to run its code.

To summarize: the eBPF program, which uses the *"kprobe/sched_setaffinity"* hook, writes the affinity masks inside an eBPF map every time that syscall is invoked. The guest agent constantly checks this map for changes, if any, and uses the *ioctl* system call to notify the guest driver when new affinity masks are available so that it can start the transfer. In the host system, these masks can be used to invoke *sched_setaffinity*, according to some host-defined allocation policy. In this work for simplicity, no particular policy has been enforced and the host's vCPU(s) process(es), inside the affinity mask, are pinned to their relative pCPU(s).

Hyper Threading Hyper-threading has different CPU ordering depending on whether the system is a native one or if it is started from QEMU. To fully support HT a remapping function is needed. Every time an affinity mask is received and has to be applied to the host system, it will be remapped if the flag is set. Note that it is not necessary to modify the eBPF program itself.

4.2 Tests

The testing phase aims at understanding the difference in terms of performance between a virtual machine running in a hardware-assisted virtualization context and one using the extensible paravirtualization mechanism, on top of the hardware-assisted facilities.

The application used as a benchmark is a simplified version of the system presented in the article "Cache-aware design of general-purpose Single-Producer-Single-Consumer" queues [8]. In particular, Lamport queues have been picked. In the system, there are two threads, one producer and one consumer, that are pinned to different CPUs, so that they can run in parallel.

Performance in the system has been evaluated and the Mpps[3] metric has been chosen. The host system is subjected to different types of loads, from fully unloaded CPUs to multiple processes running on each CPU. To simulate the workload in the host system the simple Linux command *yes* is used.

Producer-consumer systems like SPSCQ benefit from parallelization and interaction in a non-blocking way. The Linux scheduler does nothing to prevent two threads from being scheduled one after the other on the same CPU. The latter is not a frequent scenario because the scheduler will try to balance the

[3] Mega packets per second

workload among the available CPUs; However, in a system under load, there is the possibility that the two threads: producer and consumer, are scheduled one after the other on the same processor, introducing a serialization condition, which drastically degrades the performance of the entire system.

By introducing this mechanism that allows applying CPU affinity requests on the host system as well, there is the guarantee that the producer and consumer threads will never be serialized on the same CPU, avoiding this kind of slowdown.

Two types of tests have been carried out to evaluate the benefits of this extension:

- **Virtual CPU pinning** In this testing scenario, the performance difference between a hardware-assisted virtualization system with and without the extensible paravirtualization extension is evaluated. The SPSCQ system throughput is shown when these two parameters vary: the host system load and the fraction of test time in which the two threads, producer and consumer, are serialized. For this test an Intel Core i5-6600 CPU @ 3.30GHz has been used, which does not implement Hyper-threading.
- **Virtual Hyper-thread pinning**. During this test, the performance differences are evaluated between using standard *sched_setaffinity()* behaviour and Hyper-thread pinning extensions, in guest machines. Moreover, differences in throughput between running an SPSCQ application directly on the host system and running SPSCQ within a guest system are also evaluated. For this test, an Intel Core i7-6700K CPU @ 4.00GHz has been used, which implements Hyper-threading.

5 Results

Results are obtained through repeated experiments that are carried out by a Python script. They are plotted with their 90% confidence intervals.

5.1 Virtual CPU pinning

In this test, the throughput in a standard guest system is compared with one that uses the extension made for this purpose. This comparison is performed by varying two elements: the load on the host system and the fraction of time during which the two threads, producer and consumer, are scheduled on the same CPU. Note that the host load refers to how many *yes* processes are being executed in the background on the host system.

Inside the guest machine, two threads are created: one producer P and one consumer C. Those threads will request to be pinned to different CPUs: P on vCPU0 and C on vCPU 1. In the 'no load' scenario, without vCPU pinning what happens in the host system is that the QEMU threads associated with their vCPUs are freely scheduled on all available CPUs. The host system, being idle, will result in those threads running on any host CPU with a low probability of being moved to another CPU, because no other process or thread is likely to

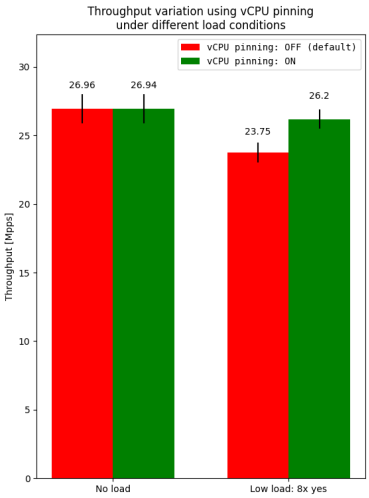

Fig. 4. Throughput comparison between standard system and vCPU pinning. No load and low load conditions are compared

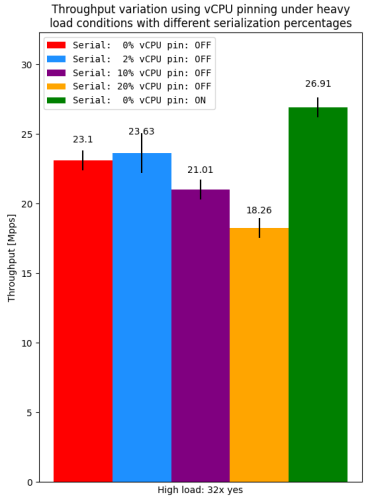

Fig. 5. Throughput comparison between standard system and vCPU pinning. High load conditions with various serialization percentage are compared

preempt them. Instead, pinning vCPU 0 to pCPU 0 and vCPU 1 to pCPU 1 will guarantee that the QEMU threads responsible for vCPU 0 and 1 will not be moved to any other host physical CPU. And as is possible to see in Figure 4. The advantage is negligible as shown in the 'no load' section of the graph.

As the load starts to intensify in the host system, 'low load' in Figure 4, different behaviour is shown. The host system is forced to execute 8x yes processes in addition to the P and C threads. So, for this reason, without vCPU pinning performance is reduced with regard to the 'no load' case. With vCPU pinning instead, the performance remains similar as P and C, which are scheduled on vCPU 0 and vCPU 1, can only be assigned on pCPU 0 and pCPU 1, while the yes processes will share the other physical CPUs.

Another scenario worthy of mention is what can be called the serialization one. The SPSCQ system benefits from parallelization and in particular, having the two threads P and C scheduled one after the other on the same CPU is not beneficial. This situation is represented, on the plot, by the serialization variable, whose range starts from no serialization (0%) and goes up to high serialization (20%). The use of vCPU pinning guarantees that P and C will never be scheduled on the same physical CPU as long as they are pinned on two different vCPUs that are assigned to two different pCPUs.

This behaviour can be seen in Figure 5 in which the host system is under high load conditions (32x yes processes running on the host). Overall, serialization significantly degrades performance in standard systems, while the others that use vCPU pinning, are not affected by this problem.

5.2 Virtual Hyper-thread pinning

This test is a comparison of the throughput obtained by a standard guest system with one that uses the Hyper-thread pinning technique, which is a further extension of the vCPU pinning mechanism. For this test, an Intel Core i7-6700K was used; It implements Hyper-threading and has 8 logical processors. For this reason, host load indicators are now multiplied by 8.

The SPSCQ test program creates two threads, P and C, and performs high-speed operations on a lockless queue while varying the load in the host system. Then all performances with Hyper-thread pinning and the standard behaviour are compared. Finally, the performance penalty is analyzed, which is introduced by running the SPSCQ program within a virtual machine, with and without Hyper-thread pinning, with regard to executing it directly on the host machine.

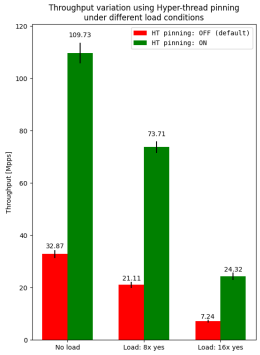

Fig. 6. Throughput comparison between standard system and Hyper-thread pinning. No load and lower load conditions are compared

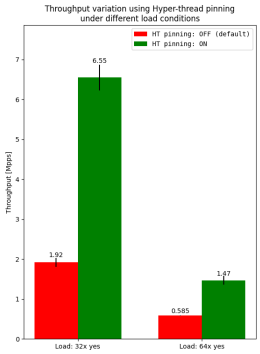

Fig. 7. Throughput comparison between standard system and vCPU pinning. Higher load conditions are compared

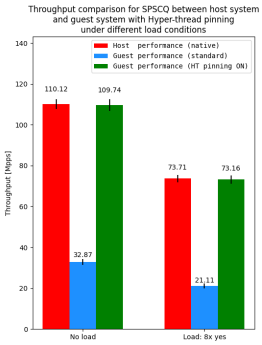

Fig. 8. Throughput comparison between host system and guest system with Hyper-thread pinning

In Figures 6 and 7 is plotted the comparison between standard system throughput and on a system using Hyper-thread pinning technique. Results are interesting: using HT pinning throughput is about 3.4 times higher than what is achieved in a traditional guest system. There is a small drop, in terms of throughput gain, when the load on the host system is very high i.e. 64x yes processes. Note that unlike many programs, SPSC benefit a lot from this technology, because running on the same core, processes can easily share resources such as caches.

In Figure 8 a comparison between native and virtualized performance is shown. What is clear, is that SPSCQ performance on a standard guest system, that does not use Hyper-threading correctly is much lower than what you get by running SPSCQ directly on the host. This shows that the theoretical 'near-native' speed is not achieved by the guest system. On the other hand, using

Hyper-thread pinning, the guest system can pin P and C threads on the same host physical core, thus obtaining near-native performance as hardware-assisted virtualization claims, but cannot be guaranteed in this special case, without modifications.

6 Confidential Computing

Confidential Computing techniques allow Virtual Machines (or processes) to run on untrusted hypervisors (or kernels). The VM can run in an "enclave" where most memory is hardware encrypted, with a decryption key that is available only while the guest software controls the CPU. Since the hypervisor has no access to the decryption key, the guest can protect its confidential data even if all of the software running outside the VM is compromised.

Confidential computing creates difficulties for paravirtualization solutions, which have always assumed that the host had free access to all of the guest memory. For example, consider *hyperupcalls* as proposed in [1]. The motivation for hyperupcalls is similar to our own: improve performance by granting the host access to some guest kernel state. Hyperupcalls achieve this aim by having the *guest* download programs in the *host*. The host runs these programs when needed, to correctly interpret the data structures living in the guest memory. In a confidential computing setting this is not easily done, since these programs would run in a context that has no access to the decryption key. In our proposed solution, instead, the programs are injected in the opposite direction and are run in the *guest*, where they have access to the decrypted guest memory.

We think that our solution, with respect to the alternatives, is also more compatible with the spirit of confidential computing. In systems, where confidential computing is used, security is a major concern. For this reason, using an ad-hoc modified version of the kernel, to introduce the paravirtualization hypercalls, can be considered a possible weak link in the root-of-trust. On the other hand, using our proposed method, the guest can formally verify the injected code, and according to some policies, decide whether to load that code or not.

7 Conclusions and future work

In conclusion, in this work is realized an extensible paravirtualization mechanism that makes use of eBPF programs. It allows programmers to add specific capabilities to host-guest systems. Overcoming the limitation introduced by running the guest system in a virtualized environment and realizing CPU affinity. This showed substantial performance gains, especially when Hyper-threading is available and a program that can benefit from is used. Future work should consider implementing a whitelist of eBPF's hook inside the guest user agent to enforce security policies and other potential applications of this extensible mechanism.

References

1. Amit, N., and Wei, M.: The Design and Implementation of Hyperupcalls. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18), pp. 97–112. USENIX Association, Boston, MA (2018)
2. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A.: Xen and the art of virtualization. ACM SIGOPS operating systems review 37(5), 164–177 (2003)
3. eBPF for Windows. `https://github.com/microsoft/ebpf-for-windows` (2022)
4. Ghatrehsamani, D., Denninnart, C., Bacik, J., and Amini Salehi, M.: The Art of CPU-Pinning: Evaluating and Improving the Performance of Virtualization and Containerization Platforms. In: 49th International Conference on Parallel Processing - ICPP. ICPP '20. Association for Computing Machinery, Edmonton, AB, Canada (2020). DOI: `10.1145/3404397.3404442`
5. Gutiérrez, C.S.V., Juan, L.U.S., Ugarte, I.Z., and Vilches, V.M.: Real-time Linux communications: an evaluation of the Linux communication stack for real-time robotic applications. arXiv preprint arXiv:1808.10821 (2018)
6. Krzywda, J., Ali-Eldin, A., Carlson, T.E., Östberg, P.-O., and Elmroth, E.: Power-performance tradeoffs in data center servers: DVFS, CPU pinning, horizontal, and vertical scaling. Future Generation Computer Systems 81, 114–128 (2018)
7. Lee, T., and Eom, Y.I.: VCPU Prioritization Interface for Improving the Performance of Latency-Critical Tasks. In: 2020 14th International Conference on Ubiquitous Information Management and Communication (IMCOM), pp. 1–4 (2020). DOI: `10.1109/IMCOM48794.2020.9001717`
8. Maffione, V., Lettieri, G., and Rizzo, L.: Cache-aware design of general-purpose Single-Producer-Single-Consumer queues. Software: Practice and Experience 49 (2018). DOI: `10.1002/spe.2675`
9. Vieira, M.A., Castanho, M.S., Pacífico, R.D., Santos, E.R., Júnior, E.P.C., and Vieira, L.F.: Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. ACM Computing Surveys (CSUR) 53(1), 1–36 (2020)
10. Zhang, I., Liu, J., Austin, A., Roberts, M.L., and Badam, A.: I'm not dead yet! the role of the operating system in a kernel-bypass era. In: Proceedings of the Workshop on Hot Topics in Operating Systems, pp. 73–80 (2019)