A Case Against (Most) Context Switches

Jack Tigar Humphries*
Stanford University

Kostis Kaffes*
Stanford University

David Mazières
Stanford University

Christos Kozyrakis
Stanford University

Abstract

Multiplexing software threads onto hardware threads and serving interrupts, VM-exits, and system calls require frequent context switches, causing high overheads and significant kernel and application complexity. We argue that context switching is an idea whose time has come and gone, and propose eliminating it through a radically different hardware threading model targeted to solve software rather than hardware problems. The new model adds a large number of hardware threads to each physical core – making thread multiplexing unnecessary – and lets software manage them. The only state change directly triggered in hardware by system calls, exceptions, and asynchronous hardware events will be blocking and unblocking hardware threads. We also present ISA extensions to allow kernel and user software to exploit this new threading model. Developers can use these extensions to eliminate interrupts and implement fast I/O without polling, exception-less system and hypervisor calls, practical microkernels, simple distributed programming models, and untrusted but fast hypervisors. Finally, we suggest practical hardware implementations and discuss the hardware and software challenges toward realizing this novel approach.

ACM Reference Format:
Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. 2021. A Case Against (Most) Context Switches. In *Workshop on Hot Topics in Operating Systems (HotOS '21), May 31–June 2, 2021, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3458336.3465274

1 Introduction

Context switches complicate both operating system and application design and add unnecessary performance overheads. Frequent switches – due to I/O interrupts, system calls, and VM-exits – disrupt the regular execution stream, require expensive state saving and restoration, and lead to poor caching behavior. For example, waking up a thread blocked on network I/O requires handling an interrupt from the network interface (NIC), running the kernel scheduler, potentially sending an inter-processor interrupt (IPI) to another core, and suffering many cache misses along the way. In systems with modern SSDs and NICs, such context switches occur too frequently, severely impacting the latency and throughput of system and application code [40, 41, 49]. In turn, system and application designers opt to avoid interrupts, wake-ups, and scheduling altogether with polling, which sacrifices one or more CPU cores and abandons the more intuitive blocking programming model [24, 37, 46, 48, 55, 63, 65, 68]. Similarly, significant overheads plague the transitions between CPU protection modes, inflating the cost of system calls and virtual machine mode switches [20, 69]. Even switching between software threads in the same protection level incurs hundreds of cycles of overhead as registers are saved/restored and caches are warmed [25, 46].

Such problems persist over decades due to shortcomings of the hardware abstractions for threading. Threading support in modern hardware has been designed to solve hardware problems, i.e., mainly improve utilization of wide, out-of-order cores. Existing multithreaded processors, such as Intel Hyperthreaded CPUs, AMD SMT CPUs, and IBM Power CPUs, implement multiple hardware threads per physical core (2–8), each with its own architectural state and instruction stream. These instruction streams are multiplexed in the processor pipeline in a fine-grained manner to best utilize all available hardware resources such as arithmetic and load/store units [75, 76]. If one thread has low instruction-level parallelism (ILP), instructions from the other thread(s) can utilize idling pipeline resources. While there is a large body of work on scheduling software threads to hardware threads [5, 12, 26, 31–33, 44, 59, 63, 64, 70, 71, 79], there is little innovation in the interface exposed to system and user software. The OS views each hardware thread as a standalone

[*] Both authors contributed equally to this research.

HotOS '21, May 31–June 2, 2021, Ann Arbor, MI, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8438-4/21/05.
https://doi.org/10.1145/3458336.3465274

logical core to independently halt, interrupt, and assign a software thread for execution. Logically, the availability of multiple threads per core is no different from the availability of multiple cores per CPU.

We argue it is time to *redesign the hardware abstractions for threading with software in mind.* We propose two radical changes that can drastically improve the efficiency and reduce the complexity of systems software. First, we propose that *hardware should directly support a much larger number of hardware threads per core.* There should be potentially 10s, 100s, or even 1000s of threads per core, where even 10 threads would be a meaningful step forward but we believe far more will eventually be practical. Resuming execution for one of these threads should be sufficiently fast (nanosecond scale), even if just a subset of threads share the pipeline at any given time. Second, *software should have direct control over hardware threads.* It should be able to `start` and `stop` them, i.e., determine which ones can be scheduled for execution on the pipeline, and modify registers belonging to different hardware threads.

Having a large number of software-controlled hardware threads per core renders many ideas practical that improve systems efficiency or simplicity. Interrupts and the design complexities associated with them can be eliminated. Hardware threads can wait on I/O queues when there are no pending I/O events and immediately wake up when an event arrives (e.g., a packet received by the NIC) without needing an expensive transition to a hard IRQ context. Expensive VM-exits [20] and system calls can simply make a specialized root-mode hardware thread runnable rather than waste hundreds of nanoseconds context-switching to root-mode in the same hardware thread. Microkernel components can run on dedicated hardware threads, reducing the communication cost and avoiding scheduling delays. Similarly, isolation necessary for hypervisors and sandboxes will be offered at low cost through the use of different hardware threads. Finally, due to the large number of processor-resident threads, blocking I/O will be cheaper, simplifying distributed systems' design.

Fundamentally, these optimizations are based on the insight that *it can be faster and simpler to start and stop a large number of hardware threads rather than to frequently multiplex a large number of software contexts on top of a single hardware thread [53, 60, 69].* In addition to requiring mode changes and state swapping, the latter often invokes the general OS scheduler with unpredictable performance results. The cost of starting and stopping the execution of hardware threads can be kept low, and its impact on overall performance can be predictable.

The rest of the paper is organized as follows. §2 provides examples of long-standing system design problems that our approach can address. §3 proposes the new hardware interface for threading. Finally, §4 reviews hardware implementations and their tradeoffs.

2 Faster and Better Kernels

We argue that giving software the ability to control a large number of hardware threads will lead to drastically different and more efficient systems design:

No More Interrupts: Our design can remove all preparatory work for interrupt processing. Instead of registering interrupt handlers in the interrupt descriptor table (IDT), the kernel can designate a hardware thread per core per interrupt type. Each of these threads initially blocks on a memory address using an `mwait` instruction. In place of sending an interrupt, an event trigger – such as the timer in the local APIC – writes to the memory address that its target hardware thread is waiting on. The hardware thread becomes runnable and handles the event without the need to jump into an IRQ context and the associated overheads. In §4 we discuss how we can use hardware thread priorities to eliminate delays for time-critical interrupts.

Fast I/O without Inefficient Polling: Having minimized the cost of notifying a waiting thread, there is no need for polling anymore. To improve performance under heavy I/O load, kernel developers use polling threads rather than interrupts. However, polling threads waste one or more cores and complicate core allocation under varying I/O load [55, 63]. Given a large number of threads per core, polling is unnecessary; we can implement I/O fast and efficiently by having threads wait on I/O events (`mwait` on the I/O queue tail in the x86 case), letting other threads run until there is I/O activity. Upon a new I/O event arrival, a waiting thread can quickly start running to process the event. I/O-heavy applications can thus achieve high throughput and low latency while other threads can efficiently consume spare cycles. If the number of hardware threads is sufficiently high, we can avoid the software and hardware complexities associated with having threads each busy poll multiple memory locations [57].

Exception-less System Calls and No VM-Exits: We can use the same approach to optimize system calls and VM-exits. Currently, kernel developers have to make an unnecessary trade-off. When a system call or VM-exit event (e.g., a `vmcall` for x86, an interrupt, a privileged instruction such as `wrmsr`, etc.) occurs, kernel code can either run in the same thread as the application or in a separate dedicated kernel thread. In the first case, the state management necessary when switching privilege levels within a hardware thread can take hundreds of cycles [46, 69]. The second option, used for serving both system calls (FlexSC [69]) and VM-exits (SplitX [53]), requires complex asynchronous APIs and scheduler fine-tuning so that kernel threads do not suffer excessive delays. Due to its simplicity, the single-thread approach has been dominating both commercial (Linux) and research (Dune [23], IX [24], ZygOS [65]) operating systems. Our proposed threading model enables the best of both worlds. System calls and VM-exits can be served in dedicated hardware threads, avoiding the mode switching overheads. Application threads can directly start kernel threads and use the API in §3 to pass parameters, eliminating the need for asynchronous API and scheduling complications. AWS Nitro [2] catches VM-exits and ships the work to an offload chip (Nitro controller), in essence, shipping them to another "thread" similar to our proposal.

Access to All Registers in the Kernel: Kernels traditionally refrain from using floating-point and vector operations (e.g., for machine learning or cryptography). This makes context switches faster as the FP and vector registers do not need to be saved and restored. With software-managed hardware threads, kernel code can run in one hardware thread and application code in a different hardware thread. Kernels can then use FP and vector operations and link with libraries that use them without affecting the system call invocation latency.

Faster Microkernels and Container Proxies: Microkernel OSes are becoming increasingly popular [3, 4, 7, 11, 14, 49, 54, 55] due to their modularity and the protection from bugs and vulnerabilities they provide through isolation. However, they suffer from the same problem as dedicated kernel threads, i.e., potentially excessive scheduling delays. With a large number of software-managed hardware threads, when an application wishes to communicate with a microkernel service such as the file system or the network stack, it can directly start the service's hardware thread achieving the same result as XPC [30] while using a simpler hardware mechanism. There is no need to move into kernel space and invoke the scheduler. This improves performance for I/O-intensive services, which have so far resorted to using dedicated cores (TAS [48], Snap [55]). Even in monolithic kernels, features such as kernel subsystem isolation can be implemented easily with very low overhead. Previous solutions such as LXDs [60] had to jump through hoops to avoid the prohibitive cost of synchronous cross-domain invocations. Such invocations will now come cheaply through software-controlled hardware threads simplifying the code and eliminating the need for helper threads that handle asynchronous interactions. We can use similar functionality to accelerate container proxies, such as Istio [15]. Container proxies would benefit from the direct transfer of control between the container and the proxy hardware threads.

Untrusted Hypervisors: Hypervisors [6, 23] on today's CPUs need privileged access to the host operating system for good performance. For example, the KVM driver [6] used for hardware virtualization in hypervisors such as QEMU [10] needs to run in the kernel. A VM-exit from a virtual machine switches to the hypervisor in the kernel and the hypervisor uses its kernel access to service I/O requests and handle page faults. Like a microkernel design, isolating the hypervisor in a less privileged mode (e.g., ring 3 in root mode for x86) would require expensive scheduling and context switch overheads. With many hardware threads per core, a hypervisor could be isolated in its own unprivileged hardware thread. VM-exits would stop the virtual machine's hardware thread and start the hypervisor's hardware thread. If the hypervisor needs to handle an I/O request or a page fault, it could, in turn, start the kernel's hardware thread. Thus, hypervisors still provide the same functionality with the same performance without privileged access to the kernel or the hardware.

Similar to hypervisors, other system components can be isolated in a less privileged mode, such as binary translators [1, 16] and eBPF [13] code. For eBPF, we could even relax some code restrictions if it ran in its own privilege domain. Quick hand-offs between hardware threads allow isolation without loss of performance.

Simpler Distributed Programming: With a limited number of hardware threads per physical core, distributed applications either resort to event-based models or rely on the scheduler to multiplex software threads on top of hardware threads. Event-based models are more difficult to work with than threading models due to their confusing control flow [78]. Distributed systems prefer the simpler threading abstraction and create large numbers of threads to hide internode communication latencies with other work [61]. Even though threading is a simpler abstraction for distributed systems, multiplexing a large number of software threads onto a small number of hardware threads is expensive. Multiplexing requires frequent scheduler interaction and good application performance is dependent on the scheduling algorithm, which is not necessarily well-suited to the application [58, 63, 77]. Given a large number of hardware threads, developers can assign one hardware thread per request and use simple blocking I/O semantics without suffering from significant thread scheduling overheads.

3 Hardware/Software Interface

We now summarize the ISA extensions that support large numbers of software-controlled hardware threads. A CPU core supports a large but fixed number of physical hardware threads named by identifiers called *ptid*s. To facilitate virtualization, instruction operands specify virtual thread identifiers, or *vtid*s, transparently mapped to ptids.

At any point, a given ptid can be in one of three states: *runnable*, *waiting*, or *disabled*. Runnable ptids can execute instructions on the CPU core. The CPU may be executing instructions only for a subset of runnable ptids on any clock cycle. A ptid can voluntarily enter the waiting state through the x86-inspired instructions *monitor*/*mwait* described below. Finally, a disabled ptid does not execute instructions until it is restarted by another ptid. Events such as page faults that trigger exceptions in today's CPUs simply write an exception descriptor to memory and disable the current ptid. A different ptid monitors the exception descriptor to detect and handle the exception.

3.1 Proposed instructions

- `monitor <memory_address>`
  `mwait`
  Like the x86 ISA [8, 9], these instructions pause the current ptid until a write occurs to `<memory_address>`. A hardware thread can `monitor` multiple memory locations. Prior work on hardware transactional memory [36] and RX queue notification mechanisms [57] has shown that such monitoring is possible with relatively small overhead. The difference in our design is that these instructions monitor any write (including DMA) to any address, may be used from any privilege level, and hence serve both external event processing and inter-thread communication. For example, each core's APIC timer can increment a counter every time a timer interrupt is triggered. In turn, the hardware thread hosting the kernel scheduler can `monitor/mwait` on that memory location. Similarly, a network thread can wait on the RX queue tail until packet arrival. Unlike x86, one can monitor uncachable addresses such as device memory or memory-mapped I/O registers; §4 sketches a possible implementation.
- `start <vtid>`: Enable ptid mapped to `vtid`.
  `stop <vtid>`: Disable ptid mapped to `vtid`.
- `rpull <vtid>, <local-reg>, <remote-reg>`
  `rpush <vtid>, <remote-reg>, <local-reg>`
  Used to read and write the registers of a disabled ptid, so as to swap software threads in and out of hardware threads. Note that in addition to normal registers, `remote-reg` can be the program counter or various control registers including a few novel ones: the exception descriptor pointer register specifies where to write an exception descriptor when the ptid becomes disabled, while a thread-descriptor-table register specifies the location of a table mapping vtids to ptids.
- `invtid <vtid>, <remote-vtid>`
  Any update to a ptid's TDT (described below) must be followed by an `invtid`. Requiring explicit invalidation facilitates hardware caching and virtualization.

3.2 Security Model

The security model is designed to facilitate process isolation, kernel/hypervisor security, and virtualization. A ptid can be in either user or supervisor mode. Access to certain registers from user mode always disables the current ptid and writes an exception descriptor to memory that a supervisor ptid can use to emulate privileged instructions for guests running in user ptids. In some cases, multiple ptids will need to report their exceptions to the same hypervisor ptid, requiring a software-based queuing design.

Consecutive Exceptions. What happens if thread *B* causes an exception while handing a previous exception in thread *A*? For example, *A* could divide by 0, and *B* could experience a page fault in the exception handler, or could even divide by 0 itself. Nothing prevents arbitrarily nested exceptions, so long as another thread *C* handles *B*'s exceptions. However, any handler chain must end somewhere, at a lowest-level kernel thread that does not have an exception handler. Triggering an exception in a thread without a handler for that exception type indicates a serious kernel bug akin to a triple-fault, and can be handled by halting or resetting the CPU.

Thread Descriptor Table. One particularly important privileged register is the thread descriptor table pointer, or TDT, which maps vtids to ptids and permissions. A ptid must be in supervisor mode to set this register in its own context or any other vtid. However, the TDT table may allow a user ptid to start, stop, and modify other less privileged registers in other ptids. Table 1 shows an example TDT.

vtid	ptid	Permissions
0x0	0x01	0b1000
0x1	0x00	0b0000 (invalid)
0x2	0x10	0b1111
0x3	0x11	0b1110

Table 1: Example Thread Descriptor Table. The 4 permission bits allow the caller to `start` - `stop` - *modify some registers* - *modify most registers* of the callee.

This TDT design allows for flexible, non-hierarchical privilege levels that facilitate the sandbox use cases discussed in §2. For example, thread B might have permission to `stop` thread A, and thread C might have permission to `stop` thread B, but thread C does not necessarily have any permission over thread A. Such a configuration is impossible in existing protection-ring-based designs.

An alternative to the TDT could be a secret-key-based design. Threads that perform thread management would need to provide the target thread's secret key if they are not running in privileged mode. Each thread would set its own key and share it with other threads using existing software mechanisms, e.g., shared memory or pipes.

3.3 Microarchitectural Security

The Spectre [50] attack demonstrates that sibling threads sharing microarchitectural resources may spy on each other.

While there is ongoing work to fix this vulnerability [21, 22, 34, 35, 51, 62], it is unclear whether Spectre can ever be fully mitigated. From initial glance, it may seem that our proposal futher aggravates microarchitectural security vulnerabilities.

We argue that these vulnerabilities are orthogonal to our proposal. While we do propose that there should be many more hardware threads, they are *not* SMT threads – they are hardware threads multiplexed onto a small number of SMT threads. Thus, with regard to Spectre, our proposal is no different from software-thread context-switching today: We still just multiplex on top of SMT threads. These SMT threads may spy on each other, but that is not a consequence of having many hardware threads. Our changes to the threading API will allow developers to build reasonable security policies, but beyond this, security policy for microarchitectural vulnerabilities is not our focus.

4 The Space of Hardware Designs

We now discuss the key requirements and implementation options for hardware that supports a large number of software-controlled threads. While there are important issues to address in future research, we believe that the required hardware is feasible and practical.

Storage for Thread State: Our proposal relies on hardware to store state for a large number of threads so that `start` and `stop` are fast (nanosecond scale). The state includes all general-purpose and control registers. For x86-64, a thread has 272 bytes of register state that goes up to 784 bytes if SSE3 vector extensions are used.

A first option is to implement hardware threads as SMT hyperthreads in modern CPUs [75, 76]. Hyperthreads execute concurrently by sharing pipeline resources in a fine-grain manner. Their implementation is expensive as all pipeline buffers must be tagged, partitioned, or replicated. This is why most popular CPUs support up to 2 hyperthreads and few designs have gone up to 4 or 8 hyperthreads [74]. We believe that *the two concerns should be separated*: use a small number of hyperthreads to best utilize the complex pipeline (likely 2-4) and multiplex additional runnable hardware threads on the available hyperthreads in hardware.

The state for additional hardware threads can be stored in large register files, similar to early multithreaded CPUs for coarse-grain thread interleaving [17–19]. The overhead of starting execution of a thread stored in these register files would be proportional to the length of the pipeline, roughly 20 clock cycles in modern processors. Modern GPUs have demonstrated the practicality of implementing multi-ported register files that store 10s of KBytes and support 10s of hardware threads. For example, the 64KByte register file in the sub-core of a Nvidia Tesla V100 GPU can store the state for 83 to 224 x86-64 threads [27]. GPUs only support a restricted execution model designed for bulk parallelism across a large number of threads. We believe that our design will allow for concurrent general-purpose CPU execution of a smaller number of hardware threads while enabling fast switching to other register-file-resident threads. For a CPU with 100 cores, the cost is 6.4MB in register file space, which is non-trivial but well within what is possible given that upcoming CPU chips feature L3 caches with a capacity of 256MB or higher [72].

A further option is to utilize larger caches, such as the private (per-core) L2 caches or the shared L3 caches, to also store state for the additional hardware threads, similar to Duplexity [56]. A fraction of a 512KB private L2 cache can store the state of tens of threads, while a few MB of an L3 cache can support hundreds of threads. Modern CPUs use wide interconnect links (32-byte or wider) and scalable topologies to connect the cache hierarchy levels. Hence, the additional cost of a bulk transfer of register state from the L2 or L3 cache is limited to 10 to 50 clock cycles (i.e., 3ns to 16ns for a 3GHz CPU). While software can save thread contexts itself, delegating this responsibility to hardware ensures that the contexts are pinned to caches and reduces software complexity. Combining these three options can support hundreds to thousands of threads per core in a cost-effective manner. There are several optimizations to consider: selecting which threads are stored closer to the core based on criticality, tracking used/modified registers to avoid redundant transfers, and hardware prefetching of the state of recently woken up threads closer to the processor core.

Managing Non-register State: As is the case with ordinary context switching [69], managing register state may only be a fraction of the cost of supporting a large number of hardware threads. Misses in caches and TLBs can lead to significant performance loss and even thrashing as numerous hardware threads start and stop. However, for many use cases, the working set size and the process count will not increase, so our proposal should not add significant additional burden to caches and the TLB. Moreover, there are two complementary techniques to alleviate caching problems for large thread counts.

First, we can pin the most critical instructions/data/translations (few KBytes) for performance-sensitive threads in caches, using fine-grain cache partitioning techniques that allow hundreds of small partitions without loss of associativity [66]. Second, we can use prefetching techniques that warm up caches of all types as soon as threads become runnable. For that, it is critical to capture threads’ working sets [38, 43, 45], including their instructions and their data such as the cache line they perform an `mwait` on and memory regions written to by I/O devices.

Neither prefetching nor caching have to be perfect. Modern out-of-order processors feature techniques that can hide a reasonable amount of latency due to cache and TLB misses. If these misses are served on-chip (e.g., by an L2 or L3 cache) and there is sufficient instruction-level parallelism (ILP), performance loss is minimal [47]. However, L3 misses served by off-chip memory lead to severe performance losses. The first goal of the hardware design should be to keep all critical state for threads on-chip. The on-chip capacity will serve as the upper bound on the number of threads a CPU can support across all cores.

Generalized `monitor-mwait`: In our proposal, hardware threads use `monitor-mwait` to express blocking and wakeup conditions. Hence, these two instructions must support more than the use cases in current x86 implementations [42]. Hardware should monitor updates to any address by any source. Current CPUs monitor writes by CPU cores that map to DRAM main memory. Future hardware should also monitor updates by I/O devices or specialized accelerators. For instance, this will enable monitoring addresses updated by a DMA engine when a new packet arrives in a network interface. The switch of external interfaces from legacy I/O specifications like PCIe to memory-based and partially-coherent specifications like CXL [28] will greatly simplify monitoring memory and caches in both the CPU and devices/accelerators. Lastly, since future hardware should be compatible with legacy devices, hardware must translate external interrupts to memory writes (similar to PCIe MSI-x functionality).

Support for Thread Scheduling: Hardware must also play an important role in thread scheduling. At a minimum, it must ensure that critical threads associated with the kernel – handlers for exceptions and external events, or the kernel scheduler – get to execute instructions when they are runnable. A simple way to meet this requirement is to execute runnable hardware threads in a fine-grain, round-robin (RR) manner, which emulates processor sharing (PS) and allows all runnable threads to make progress without the need for interrupts. The combination of PS scheduling with thread-per-request will actually provide superior performance for server workloads with high execution-time variability [46, 80]. In addition to RR scheduling, we can introduce hardware support for thread priorities (e.g., threads used for serving time-sensitive interrupts receive more cycles [56]) or even hardware-based (but software-managed) thread queuing, load balancing, priorities, and scheduling [29, 52, 67]. Hardware support will be needed for fine-grain tracking of threads’ resource consumption for cloud billing or software decisions. Part of scheduling, such as associating hardware threads with I/O events, could also be transparently offloaded to peripheral devices such as smartNICs [39, 73].

The role of the OS scheduler will also change. Its main task today is to decide which software thread runs on each hardware thread. With a large number of hardware threads, the scheduler will rarely need to swap a software thread in and out of a hardware thread. This operation should become as uncommon as swapping memory pages to disk. The OS scheduler will enforce software policies by starting and stopping hardware threads and setting their priorities. It will also manage the mapping of threads to cores in order to improve locality. Since starting and stopping threads incurs low overhead, the scheduler will run in much tighter loops, drastically improving application performance (reduced queuing time, more time for higher-quality management decisions [46, 63]).

5 Conclusion

Context switches and their associated overheads and limitations should be eliminated altogether through the adoption of a new hardware threading model that gives software control over a large number of hardware threads per core. While there are software and hardware research questions to address, we argue that the proposed threading model will lead to significant simplification and performance gains for both systems and application code.

6 Acknowledgments

We thank Adam Belay, Frans Kaashoek, Paul Turner, Xi Wang, Neel Natu, Thomas Wenisch, Partha Ranganathan, Hudson Ayers, Deepti Raghavan, Gina Yuan, and the anonymous HotOS reviewers for their helpful feedback. This work is partially supported by Stanford Platform Lab sponsors and Facebook.

References

[1] About the rosetta translation environment. https://developer.apple.com/documentation/apple_silicon/about_the_rosetta_translation_environment. Last accessed: 2021-02-01.

[2] Aws nitro system. https://aws.amazon.com/ec2/nitro/. Last accessed: 2020-11-29.

[3] Data plane development kit. https://www.dpdk.org. Last accessed: 2021-01-29.

[4] Driverkit | apple developer documentation. https://developer.apple.com/documentation/driverkit. Last accessed: 2021-01-29.

[5] "fully HT-aware scheduler" support, 2.5.31-BK-curr. https://lwn.net/Articles/8553/. Last accessed: 2021-01-18.

[6] Kernel virtual machine. https://www.linux-kvm.org/page/Main_Page. Last accessed: 2021-01-31.

[7] libfuse. https://github.com/libfuse/libfuse. Last accessed: 2021-01-29.

[8] Monitor x86 instruction. https://www.felixcloutier.com/x86/monitor. Last accessed: 2021-02-01.

[9] Mwait x86 instruction. https://www.felixcloutier.com/x86/mwait. Last accessed: 2021-02-01.

[10] Qemu. https://www.qemu.org. Last accessed: 2021-05-11.

[11] Storage performance development kit. https://spdk.io. Last accessed: 2021-01-29.

[12] The scheduler and hyperthreading. https://lwn.net/Articles/8720/. Last accessed: 2021-01-18.

[13] A thorough introduction to ebpf. https://lwn.net/Articles/740157/. Last accessed: 2020-12-10.

[14] The userspace i/o howto. https://www.kernel.org/doc/html/v4.14/driver-api/uio-howto.html. Last accessed: 2021-01-29.

[15] What is istio? https://istio.io/latest/docs/concepts/what-is-istio. Last accessed: 2021-01-30.

[16] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 2–13, New York, NY, USA, 2006. Association for Computing Machinery.

[17] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The mit alewife machine: A large-scale distributed-memory multiprocessor. Technical report, USA, 1991.

[18] A. Agarwal, J. Kubiatowicz, D. Kranz, B. H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: an evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3):48–61, 1993.

[19] A. Agarwal, B. . Lim, D. Kranz, and J. Kubiatowicz. April: a processor architecture for multiprocessing. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 104–114, 1990.

[20] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software Techniques for Avoiding Hardware Virtualization Exits. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 373–385, Boston, MA, June 2012. USENIX Association.

[21] G. Barthe, S. Cauligi, B. Gregoire, A. Koutsos, K. Liao, T. Oliveira, S. Priya, T. Rezk, and P. Schwabe. High-assurance cryptography in the spectre era. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 788–805, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[22] Jonathan Behrens, Anton Cao, Cel Skeggs, Adam Belay, M. Frans Kaashoek, and Nickolai Zeldovich. Efficiently mitigating transient execution attacks using the unmapped speculation contract. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1139–1154. USENIX Association, November 2020.

[23] 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.

[24] 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.

[25] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Lightweight preemptible functions. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 465–477. USENIX Association, July 2020.

[26] James R. Bulpin and Ian A. Pratt. Hyper-Threading Aware Process Scheduling Heuristics. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 27, USA, 2005. USENIX Association.

[27] Jack Choquette. NVIDIA's Volta GPU: Programmability and Performance for GPU Computing. In *Proceedings of the 2017 Hot Chips Technical Conference*, 2017.

[28] Compute Express Link Specification 2.0. https://www.computeexpresslink.org, 2020.

[29] Intel data streaming accelerator preliminary architecture specification. https://software.intel.com/sites/default/files/341204-inteldata-streaming-accelerator-spec.pdf, 2019.

[30] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. Xpc: Architectural support for secure and efficient cross process call. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 671–684, Phoenix, Arizona, 2019. Association for Computing Machinery.

[31] S. Eyerman, P. Michaud, and W. Rogiest. Revisiting symbiotic job scheduling. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 124–134, 2015.

[32] Alexandra Fedorova, Christopher Small, Daniel Nussbaum, and Margo Seltzer. Chip Multithreading Systems Need a New Operating System Scheduler. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, EW 11, page 9–es, New York, NY, USA, 2004. Association for Computing Machinery.

[33] 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.

[34] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2020.

[35] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation, 2020.

[36] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. *SIGARCH Comput. Archit. News*, 32(2):102, March 2004.

[37] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[38] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns, 2018.

[39] 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, Princeton, NJ, USA, 2019. Association for Computing Machinery.

[40] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP ≈ RDMA: Cpu-efficient remote storage access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 127–140, Santa Clara, CA, February 2020. USENIX Association.

[41] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Nick McKeown, and Changhoon Kim. The nanopu: Redesigning the cpu-network interface to minimize rpc tail latency, 2020.

[42] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 11 2020.

[43] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, page 247–259, Davis, California, 2013. Association for Computing Machinery.

[44] 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.

[45] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In

Proceedings of the 17th Annual International Symposium on Computer Architecture, ISCA '90, page 364–373, Seattle, Washington, USA, 1990. Association for Computing Machinery.

[46] 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.

[47] Tejas Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *In Workshop on Memory Performance Issues*, 2002.

[48] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, Dresden, Germany, 2019. Association for Computing Machinery.

[49] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash ≈ local flash. *SIGARCH Comput. Archit. News*, 45(1):345–359, April 2017.

[50] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[51] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Speccfi: Mitigating spectre attacks using cfi informed speculation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 39–53, 2020.

[52] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, page 162–173, San Diego, California, USA, 2007. Association for Computing Machinery.

[53] Alex Landau, Muli Ben-Yehuda, and Abel Gordon. SplitX: Split Guest/Hypervisor Execution on Multi-Core. In *Proceedings of the 3rd Conference on I/O Virtualization*, WIOV'11, page 1, Portland, OR, 2011. USENIX Association.

[54] 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.

[55] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, Huntsville, Ontario, Canada, 2019. Association for Computing Machinery.

[56] A. Mirhosseini, A. Sriraman, and T. F. Wenisch. Enhancing server efficiency in the face of killer microseconds. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 185–198, 2019.

[57] Amirhossein Mirhosseini, H. Golestani, and T. Wenisch. HyperPlane: A Scalable Low-Latency Notification Accelerator for Software Data Planes. *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 852–867, 2020.

[58] 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.

[59] Jun Nakajima and Venkatesh Pallipadi. Enhancements for hyperthreading technology in the operating system: Seeking the optimal scheduling. In *Proceedings of the 2nd Conference on Industrial Experiences with Systems Software - Volume 2*, WIESS'02, page 3, USA, 2002. USENIX Association.

[60] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. Lxds: Towards isolation of kernel subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 269–284, Renton, WA, July 2019. USENIX Association.

[61] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, page 291–305, USA, 2015. USENIX Association.

[62] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1481–1498. USENIX Association, August 2020.

[63] 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.

[64] Peng Jianzhang, Gu Naijie, Li Yehua, and Zhang Xu. Tuning linux's load balancing algorithm for cmt system. In *IEEE Conference Anthology*, pages 1–4, 2013.

[65] 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, Shanghai, China, 2017. Association for Computing Machinery.

[66] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *Proceedings of the 38th annual International Symposium in Computer Architecture (ISCA-38)*, June 2011.

[67] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. *SIGPLAN Not.*, 45(3):311–322, March 2010.

[68] Leah Shalev, Julian Satran, Eran Borovik, and Muli Ben-Yehuda. IsoStack: Highly Efficient Network Processing on Dedicated Cores. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 5, Boston, MA, 2010. USENIX Association.

[69] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 33–46, USA, 2010. USENIX Association.

[70] Read Sprabery, Konstantin Evchenko, Abhilash Raj, Rakesh B. Bobba, Sibin Mohan, and Roy H. Campbell. A Novel Scheduling Framework Leveraging Hardware Cache Partitioning for Cache-Side-Channel Elimination in Clouds. *CoRR*, abs/1708.09538, 2017.

[71] S. Srikanthan, S. Dwarkadas, and K. Shen. Coherence Stalls or Latency Tolerance: Informed CPU Scheduling for Socket and Core Sharing. In *USENIX Annual Technical Conference*, 2016.

[72] David Suggs and Dan Bouvier. AMD Zen2 Processors. In *Proceedings of the 2019 Hot Chips Technical Conference*, 2019.

[73] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandres Daglis. *The NeBuLa RPC-Optimized Architecture*, page 199–212. IEEE Press, 2020.

[74] Brian Thompto. POWER9 Processor for the Cognitive Era. In *Proceedings of the 2016 Hot Chips Technical Conference*, 2016.

[75] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, page 191–202, Philadelphia, Pennsylvania, USA, 1996. Association for Computing Machinery.

[76] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, page 392–403, New York, NY, USA, 1995. Association for Computing Machinery.

[77] 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.

[78] J. Robert von Behren, Jeremy Condit, and Eric A. Brewer. Why events are a bad idea (for high-concurrency servers). In Michael B. Jones, editor, *Proceedings of HotOS'03: 9th Workshop on Hot Topics in Operating Systems, May 18-21, 2003, Lihue (Kauai), Hawaii, USA*, pages 19–24. USENIX, 2003.

[79] Yaohua Wang, Rongze Li, Zhentao Huang, and Xu Zhou. An In-Depth Analysis of System-Level Techniques for Simultaneous Multi-Threaded Processors in Clouds. In *Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications*, HP3C 2020, page 145–149, New York, NY, USA, 2020. Association for Computing Machinery.

[80] 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.