Contextual Concurrency Control

Sujin Park[†] Irina Calciu[*] Taesoo Kim[†] Sanidhya Kashyap[‡]
[†]*Georgia Tech*, [*]*VMware Research*, [‡]*EPFL*

ABSTRACT

Kernel synchronization primitives are of paramount importance to achieving good performance and scalability for applications. However, they are usually invisible and out of the reach of application developers. Instead, kernel developers and synchronization experts make all the decisions regarding kernel lock design.

In this paper, we propose *contextual concurrency control* (C^3), a new paradigm that enables userspace applications to tune concurrency control in the kernel. C^3 allows developers to change the behavior and parameters of kernel locks, to switch between different lock implementations and to dynamically profile one or multiple locks for a specific scenario of interest.

To showcase this idea, we designed and implemented Concord, a framework that allows a privileged userspace process to modify kernel locks on the fly without re-compiling the existing code base. We performed a preliminary evaluation on two locks showing that Concord allows userspace tuning of kernel locks without incurring significant overhead.

CCS CONCEPTS

• **Computer systems organization** → Multicore architectures; • **Software and its engineering** → **Mutual exclusion**; **Concurrency control**; *Scheduling*.

KEYWORDS

concurrency control, eBPF, Linux.

ACM Reference Format:
Sujin Park[†] Irina Calciu[*] Taesoo Kim[†] Sanidhya Kashyap[‡] . 2021. Contextual Concurrency Control. In *Workshop on Hot Topics in Operating Systems (HotOS '21), May 31-June 2, 2021, Ann Arbor, MI, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3458336.3465279

ACM ISBN 978-1-4503-8438-4/21/05…$15.00
https://doi.org/10.1145/3458336.3465279

1 INTRODUCTION

The end of Moore's law and Dennard scaling have left applications scrambling for more performance and better optimizations. Developers can no longer rely on hardware to deliver significant improvements every two years. Instead, they must resort to novel ways to improve application performance. In particular, recent work in the industry and academia has focused on eliminating inefficiencies in the system software stack [29, 39, 49] or circumventing high overhead system operations. Kernel bypass, for example, realizes this goal by moving several operations in userspace [29, 32, 49]. Other work restructures the underlying operating system (OS) for certain classes of applications [42, 48].

In many domains, specialization seems to be the answer in the pursuit of more performance. Across the board, specialization bridges the semantic gaps between applications and the kernel [26, 48, 50], or even between different kernel sub-systems [37]. In particular, specialization can establish the *context* under which an application is requesting certain functionality from the system. While application specialization and kernel bypass have been extensively studied for storage [29], networking [28, 49], and accelerators [15], we focus our attention on the part of the kernel that has been traditionally not exposed to applications: concurrency control.

The kernel locks are of paramount importance to achieving good performance and scalability for applications [12, 13, 33, 35, 43]. However, kernel synchronization primitives are usually invisible and out of the reach of application developers. Instead, kernel developers and synchronization experts make all the decisions regarding kernel synchronization and lock design.

This approach to kernel lock design raises two major concerns. First, designing locking algorithms and verifying their correctness is already challenging. Increasing hardware heterogeneity makes it exponentially more challenging because lock designers try to optimize locks for each new platform to decrease cache traffic and minimize contention [16, 22, 41]. Thus, synchronization experts need to develop many more algorithms and platform-dependent optimizations [4, 6, 45, 47]. Second, the locks and their developers lack awareness of the context in which the lock is operating, leading to pathological cases, such as priority inversion [35, 37, 43], scheduler subversion [46], and lock-holder preemption [36]. Prior work addressed these problems but provided only point-solutions without addressing the underlying lack-of-context issue.

In this paper, we propose *contextual concurrency control* (C^3), a new paradigm in which userspace applications can tune concurrency control in the kernel. For example, users can prioritize a certain task or system calls that hold a set of locks. Moreover, users can enforce hardware-specific policies, such as asymmetric multiprocessing-aware locking, and they can decide to prioritize either readers or writers, based on the given workload. C^3 allows developers to tune various locks in the kernel, change their parameters and behavior, even change between different lock implementations, and dynamically profile the observed lock(s) for a specific interest (*e.g.*, platform, workload, data distribution, etc.).

To showcase the C^3 idea, we designed and implemented Concord, a framework that allows a privileged userspace process to modify kernel locks on the fly without recompiling the existing code base. Concord exposes a set of well-defined lock-based APIs and uses eBPF [26] and live-kernel patching [31] to update locks specified by a userspace application dynamically. The APIs provided by Concord are designed not to break mutual-exclusion invariants for maintaining the correctness of the mutual-exclusion lock property while implementing the user-defined policies. Also, Concord enables better profiling for applications affected by a lock in the kernel. For example, using Concord an application developer can choose and profile a single contending lock, unlike with current tools, which only allow developers to profile all locks at the same time [56]. After profiling, the developer can specialize the locking primitive to further improve the application performance within a given context. Developers can customize the lock based on a specific hardware platform as well as a specific known application and workload.

We used Concord on two existing locks, ShflLocks [33] and BRAVO [20]. ShflLock allows *lock developers* to specify lock *policies*, which can influence the acquisition order of the lock. Concord allows *application developers* to define new policies for the locks without modifying the kernel code. Our initial investigation and preliminary results are promising, showing that Concord does not incur significant overhead when implementing known policies, such as NUMA-awareness, or prioritizing readers over writers.

Realizing C^3 is challenging on various levels, which opens up interesting research directions. First, application developers require an in-depth understanding of the kernel to know the problematic locks. Second, designing and composing policies for the locks is not straightforward. Moreover, conflicting policies can sometimes lead to worse performance and unexpected behavior. Finally, code generated from a policy can have its own overhead, which can degrade application performance.

C^3 not only creates the opportunity to explore kernel concurrency specialization in userspace but also opens up possibilities to explore hardware support. Moreover, we can further extend this paradigm to other forms of concurrency control mechanisms, such as optimistic locking, and relaxed synchronization mechanisms.

2 BACKGROUND

We first discuss various kernel extensions and kernel bypass techniques for improving application performance and later discuss locks evolution.

2.1 Exposing Application Semantics

Software stack specialization is now the new norm for improving application performance. The idea of co-designing the entire software stack, including the kernel, started with the Exokernel [23], which proposed pushing code to the kernel for performance purposes. Corey [11] proposed three new interfaces, such as shares, address ranges, and kernel cores, to improve applications scalability by avoiding kernel bottlenecks for increasing core count. Over time, even monolithic kernels, such as Linux, have started to allow userspace applications to customize the kernel behavior. Developers can now use eBPF [26] to customize the kernel for tracing, security, and even for performance purposes. For example, eXpress Data Path (XDP) [50] provides a programmable network data path to modify packets without compilation.

Moreover, there is an eBPF-enabled framework for accelerating some FUSE file system operations [8]. Besides eBPF, Linux developers are using io_uring [7], a shared-memory ring-buffer between userspace and the kernel, to expedite asynchronous IO operations. In addition, applications today can handle on-demand paging entirely in userspace with the help of userfaultfd [18]. Following a similar ideology, we take a step further in applying application specialization to control concurrency mechanisms of the underlying kernel, which opens up various opportunities for both lock designers and application developers.

2.2 Locks: Past, Present, and Future?

Hardware is the dominant factor in determining the scalability of locks, thereby impacting applications' scalability. For instance, queue-based locks [41] minimize excessive cache-line traffic when multiple threads acquire the lock at the same time. Meanwhile, hierarchical locks [16, 22] use batching to minimize the issue of cache-line bouncing in today's machines. CNA [21] and ShflLock [33] address the problem of hierarchical locks: memory overhead and degraded performance for a smaller number of cores. ShflLock presents a new ideology of designing lock algorithms by decoupling lock-policy from implementation. Instead of reinventing new locks, it treats both hardware characteristics, such as NUMA-awareness, or software behavior, such as parking threads, as policies. They introduce the notion of a shuffler that reorders the queue or modifies waiting threads' states, mostly off the critical path. Although ShflLocks provides an approach to enforce policies, they try to focus on generic policies on a set of simple lock acquire/release APIs. To cater to applications' requirements, we expose a set of methods that allow application developers to define their policies in a controlled and safe manner.

3 USE CASES

C^3 enables multiple use cases that were not possible before. For example, application developers can use C^3 to profile the specific kernel locks that impact a given workload and to update lock acquisition policies on the fly. We list some of the use cases that expose context to the underlying locking primitives for both performance and profiling. Although our approach is applicable to any lock design, we apply C^3 to ShflLock [33] and we use the shuffler for enforcing policies.

3.1 Scheduling Threads Waiting for a Lock

The first category of use cases that we consider is re-ordering the list of threads waiting to acquire the lock according to a user-defined policy to improve performance under a certain workload. Threads waiting for the lock can be scheduled in two different manners: acquisition-aware scheduling, based on the acquisition order of the lock, and occupancy-aware scheduling, based on the time threads spend inside the critical section.

3.1.1 Acquisition-aware scheduling.

Lock switching. C^3 enables developers to switch among various lock algorithms. There are three prominent scenarios. (i) Switch from a neutral readers-writer lock design to a per-CPU or NUMA-based readers-intensive readers-writer design for a read-intensive workload [16]. Examples include page-faulting [35] and enumerating files in a directory [43]. The other scenario is switching from a neutral readers-writer lock to a pure writers lock for a write-intensive workload; an example is creating multiple files in a directory. (ii) Switch from a NUMA-based lock design to a NUMA-aware combining approach, in which the lock holder executes the operation on behalf of a waiting thread [24]. This approach leads to better performance because it removes at least one cache-line transfer. (iii) Switch between blocking and non-blocking locks or vice versa, *e.g.*, switch blocking readers-writer lock (`rwsem`) to non-blocking readers-writer lock (`rwlock`) by switching off the parking/wakeup policy from the ShflLock 's shuffler function. This approach brings two benefits: First, developers can remove the ad-hoc synchronization, such as using a non-blocking lock and implementing a parking/wakeup strategy with wait events, which is commonly used in the Btrfs file system. Second, C^3 allows developers to unify the locking design by multiplexing multiple policies on the fly.

Lock inheritance. A process might acquire multiple locks to perform an operation. For example, a process in Linux can acquire up to 12 locks (*e.g.*, `rename` operation), or an average of four locks to do memory or file metadata management operations. Unfortunately, this locking pattern raises a pathological case for queue-based locks, which we have observed in our experiments—some threads have to wait for a longer duration to acquire the top-level lock, which is held by another thread waiting for another lock. For example, suppose thread $\texttt{t}_1$ wants to acquire two locks—L_1 and then L_2 for an operation, while $\texttt{t}_2$ only wants to acquire L_1 for its operation. Since these locking protocols are FIFO-based, $\texttt{t}_1$ might be at the end of the queue to acquire L_2, while $\texttt{t}_2$ is waiting to acquire L_1. Thanks to C^3, a developer can provide more context to the kernel: either $\texttt{t}_1$ acquires all locks together, or $\texttt{t}_1$ declares which locks it already holds, so that the shuffler can give it a higher priority for acquiring the next lock (L_2 in this case).

Lock priority boosting. An application might want to prioritize either a system call path or a set of tasks over others for better performance. With C^3, a developer can encode the task priority context by annotating it and pass this information to the affected locks. For a system call, the developer can share information about a set of locks and the prioritized threads on the critical path. The shuffler will then prioritize these threads over other threads waiting for the locks for the specified application.

Exposing scheduler semantics. Often, over-subscribing hardware resources, such as CPU or memory, can lead to better resource utilization, for both userspace runtime systems [17], as well as virtual machines [55]. Although over-subscription improves hardware utilization, it introduces the issue of a double scheduling problem [25, 52]. The hypervisor may schedule out a `vCPU` being the lock holder or the very next lock waiter in a VM. With C^3, the hypervisor can expose the `vCPU` scheduling information to the shuffler to prioritize waiters based on their running time quota.

Adaptable parking/wake-up strategy. All blocking locks follow spin-then-park strategy, in which they park themselves after spinning for a while [44]. This spin time is mostly ad-hoc [35], *i.e.*, waiters either spin for a certain duration based on a time quota, or keep on spinning if no tasks are present to execute. Now, application developers can expose the timing context after either analyzing the length of the critical section to minimize energy consumption and wake-up information to schedule the next waiters on time to minimize the wake-up latency. Moreover, developers can further encode sleeping information to wake up waiters right before the lock is about to be acquired to minimize long wake-up latency. This approach is also applicable for paravirtualized spinlocks [34] to avoid the convoy effect [10].

3.1.2 *Occupancy-aware Scheduling.*

Priority inheritance. Priority inversion occurs when a low priority task holding a lock is scheduled out by a normal priority task, waiting for the same lock. Kim *et al.* [37] illustrated this issue in the Linux IO stack: when scheduling IO requests, a normal task that wants to acquire a lock can schedule out a lower priority background task that is holding the same lock. The scheduling out of the lock holder, *i.e.*, the background task, leads to degraded IO performance. C^3 enables application developers to allow inheriting the priority of the normal task (called priority inheritance) and avoid such anomalies.

Task-fair co-operative scheduling. Patel *et al.* [46] introduce a new class of problem, called the scheduler subversion problem, in which two tasks acquire the lock for varying time. The task with longer holding time subverts the scheduling goal of the OS. They address this problem by keeping track of the critical section length and penalizing tasks that hold the lock for a longer duration. Although this solution addresses the problem, it enforces scheduling fairness even for applications that might not benefit from it. C^3 allows application developers to encode this information by injecting code at exposed lock APIs and overcome the problem of scheduler subversion only when needed.

Task-fair locks on AMP machines. Asymmetric multicore processors (AMP) have varying computation capability cores in one processor [4, 5, 19]. The basic locking primitives used on such an architecture suffer from a type of scheduler subversion problem, and applications throughput can collapse because of the slower computing capability of weaker cores. For faster progress, developers can either assign critical locks on faster cores or reorder the queue of threads waiting to acquire the lock in such a way that improves the lock throughput.

Realtime scheduling. Similar to scheduling in realtime systems, application developers can create lock policies that always schedule threads to guarantee SLOs. Here, lock developers can design an algorithm based on the phase-fair property [14]. This approach allows us to also eliminate

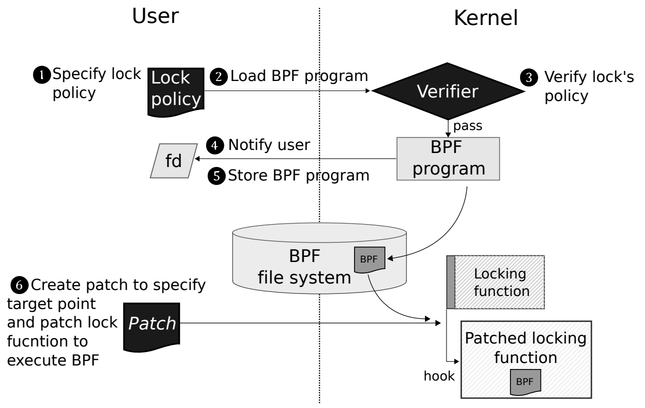

Figure 1: An overview of CONCORD's workflow.

jitters and guarantees an upper bound on tail latency for latency-critical applications.

3.2 Dynamic Lock Profiling

With C^3, application developers can profile information about any kernel lock, unlike current tools [56], in which all locks are profiled together. The ability to select which locks to profile enables developers to profile locks at different levels of granularity. For example, they can profile all spinlocks running in the kernel, locks in a specific function, code path or namespace, or even a single lock instance. This approach benefits application developers by enabling them to understand better the underlying synchronization by profiling only the interesting parts.

In the future, a developer could also reason about performance contracts [30] by encoding performance contracts that affect an application's performance due to various shuffling policies and even reason about some of the guarantees provided by a set of policies.

4 THE CONCORD FRAMEWORK

To demonstrate the C^3 potential to improve application performance and simplify profiling kernel bottlenecks, we implemented the CONCORD framework. CONCORD modularizes decisions and behaviors used by locking functions and exposes them as APIs. By replacing these exposed APIs with user-defined code, users can customize locking functions for their needs. For example, whether to spin before joining a waiting queue can be an API so that users can make the decision. A user first writes her own code to modify lock protocols in the kernel based on her use case (§3). Then CONCORD takes this code and replaces the annotated lock functions inside the kernel. This replacement can range from one lock instance to every lock in the kernel.

APIs	Description	Hazard
`bool cmp_node(lock, shuffler_node, curr_node)`	Decide whether to move current node forward	Fairness
`bool skip_shuffle(lock, shuffler_node)`	Skip shuffling on this shuffler and hand over shuffler	Fairness
`bool schedule_waiter(lock, curr_node)`	Waking/parking/priority for a lock	Performance
`void lock_acquire(lock)`	Invoked when trying to acquire a lock	Increase critical section
`void lock_contended(lock)`	Invoked when trylock failed and need to wait	Increase critical section
`void lock_acquired(lock)`	Invoked when actually acquired a lock	Increase critical section
`void lock_release(lock)`	Invoked when release a lock	Increase critical section

Table 1: APIs and their potential hazards provided by Concord with ShflLocks.

4.1 Overview

Figure 1 shows the workflow of Concord. A user specifies a lock policy (❶), which the eBPF verifier validates after compilation, considering both eBPF restrictions and mutual exclusion safety properties (❷, ❸). Then the verifier notifies the user of the verification outcome (❹) and, in case of success, stores the compiled eBPF code in the file system (❺). Finally, Concord uses the livepatch module [31] to replace the annotated functions for the specified locks(❻).

4.2 Safety and APIs

Although the onus lies on the user, Concord provides various APIs to support flexible implementation of locking policy while ensuring safety. The underlying implementation of Concord relies on eBPF to modify kernel locks. By using eBPF and the lock APIs (Table 1), users implement their required policies for a set of lock instances in the kernel. A user can encode multiple policies [33] in a C-style code, which is translated into native code and is checked by an eBPF verifier for safety. The verifier performs symbolic execution before loading the native code into the kernel, such as memory access control or allowing only whitelisted helper functions.

In addition to the eBPF verifier, Concord provides more safety properties with respect to locks. Table 1 presents an example of Concord APIs to manipulate ShflLocks and their potential side effects on pathological policy. ShflLocks is a suitable target to introduce our framework as it has separate lock acquiring phase and a phase for reordering waiting queue.

The first two APIs customize shuffling behavior for a waiting queue. `cmp_node()` function returns a boolean result that is used as a comparison function when reordering waiting nodes of a lock. A user relies on this function to compare the current node and the shuffler node to whether reorder the current node. For example, a user can define a NUMA-aware policy by grouping waiters from the same socket. On the other hand, one can design scheduler cooperative lock [46] by giving less priority to nodes by prioritizing nodes with small critical section length. Although users can incorrectly implement policy that may break fairness guarantees, our runtime checks ensure the correct mutual exclusion properties. Moreover, the kernel does not have any liveness of deadlock issues because our APIs, such as `cmp_node()` does not modify the locking behavior but only returns the decision for moving a node. Our second API, *i.e.*, `skip_shuffle()`, is called right before the shuffling phase that decides whether to skip shuffling for current shuffler.

The last four APIs are designed for lock profiling. They enable developers to profile their locks in fine-grained manner by implementing the desired behavior for each invocation. Although they do not alter the behavior of the locking function, heavy profiling policies can increase the length of critical sections, thereby leading to performance degradation.

In addition to Concord APIs, we can further introduce additional invariants to guarantee further safety of lock algorithms. For example, statically bounding the number of shuffling rounds minimizes starvation, while comparing the length of the waiting node before and after the shuffling phase ensures that the linked list is correct.

Although our APIs allow developers to compose multiple policies for various use cases (§3), we rely on eBPF because of its maturity and well defined interfaces that allow us to not only implement such policies easily but also are already a standard that are regularly used by developers. For example, we use eBPF helper functions [1], such as CPU ID, NUMA ID and time along with its map data structure to store information at runtime. Moreover, eBPF exposes the functionality of chaining multiple eBPF programs, which users can use for composing policies. Finally, we also rely on livepatching shadow data structures [2] for modifying data structures that are used by locking primitives. For example, we can extend the node data structure of the queue based lock with extra information for encoding information for specific use cases.

5 PRELIMINARY EVALUATION

We evaluate the overhead of the Concord framework by comparing the overhead of dynamically changing lock designs using Concord with the pre-compiled versions of the same locks. We evaluate on an eight-socket, 80-core machine, and modify two lock algorithms: `BRAVO` [20] and

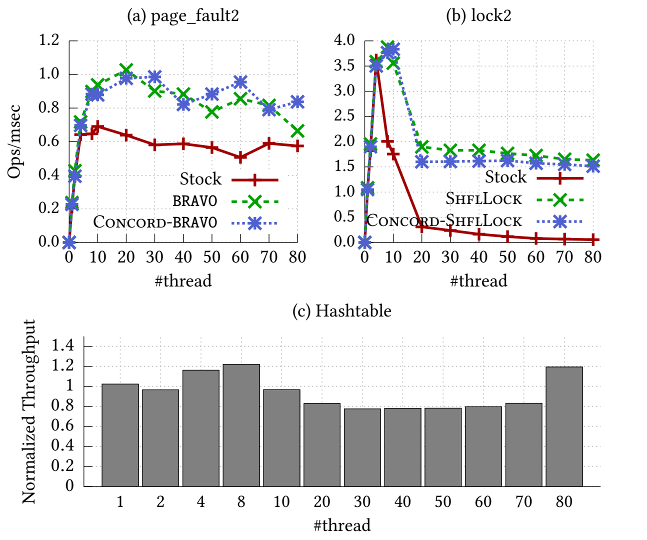

Figure 2: Impact of Concord on BRAVO and ShflLock.

ShflLock [33]. For the BRAVO, we explicitly switch between a neutral readers-writer lock to a distributed version for readers. In the case of ShflLock, we implement the NUMA-awareness policy.

In Figure 2 (a) and (b), we use two micro-benchmarks [9] that severely stress readers for BRAVO and writers for the ShflLock, respectively. We observe that Concord has almost negligible overhead, while providing the flexibility to modify the behavior of the lock on the fly. Figure 2 (c) shows the worst-case slowdown of Concord. We use a benchmark [54] that uses a global lock to protect the hash table. The benchmark shows the normalized throughput of ShflLock and Concord-ShflLock. We observe that dynamically modifying lock algorithms can incur up to 20% overhead in the worst-case scenario when no userspace code is executed.

6 DISCUSSION

C^3 opens up a new research direction for tuning concurrency control: applications can have even more control of the software stack, by tuning the kernel concurrency control. However, there are multiple challenges to comprehensively realize C^3.

Composing policies. Application developers provide a set of policies that locks need to apply. Composing multiple policies is a difficult task, especially when some of the policies could be conflicting. We would like to automate this process by leveraging program synthesis [38, 51], that can move the verification of the safety properties entirely in userspace and also provide a *safe* way to compose conflicting policies.

Overhead in applying policies. A user cannot add too many policies, as their execution may fall on the critical path. Moreover, eBPF also has some overhead, which we have noticed while designing the lock profiler. In the future, we would like to address such challenges by revisiting the eBPF design in the context of lightweight profiling.

Safety. Currently, our model allows a privileged user to modify kernel locks. This model is only applicable to one user using the whole system. However, to handle multi-tenancy in cloud environments, we need a tenant-aware policy composer that does not violate isolation among users. We will try to address this issue with two approaches: synthesizing policies in the userspace to avoid such conflicts and also add runtime checks in the lock algorithms, which are only used if a policy can affect a particular behavior.

Other synchronization mechanisms. Besides locks, there are other synchronization mechanisms, that are heavily used in the kernel, such as RCU [40], seqlocks [27], wait events [53], etc. Extending Concord to support them will further allow applications to improve their performance. Our approach also opens up new ways to expose various system-based higher level synchronization mechanisms that are important for overall correctness [3].

Modifying locks in userspace applications. In addition to kernel locks, userspace applications have their own locks that are generic in nature. We plan to extend Concord for userspace applications that provides more control of the concurrency control in a dynamic manner, while the application is running. In contrast, existing techniques, such as library interposition, allow only a one time change to a different lock implementation when the application starts its execution.

7 CONCLUSION

Kernel synchronization primitives have a huge impact on some applications' performance and scalability. However, controlling kernel synchronization primitives is out of reach for application developers. This paper proposes a new paradigm, called *contextual concurrency control* (C^3) that allows userspace applications to fine tune the kernel concurrency primitives. C^3 opens up a new way to think about specializing the software stack and accelerates innovation in the field of kernel synchronization.

8 ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful feedback. This research was supported, in part, by the NSF award CNS-1563848, CNS-1704701, CRI-1629851 and CNS-1749711 ONR under grant N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA AIMEE grant under Agreement No. HR00112090034 and ETRI IITP/KEIT[2014-3-00035], and gifts from Facebook, Mozilla, Intel, VMware and Google.

REFERENCES

[1] 2020. Linux eBPF helper functions (version: 5.4). (2020). https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/uapi/linux/bpf.h?h=v5.4.

[2] 2020. Shadow Variables. (2020). https://www.kernel.org/doc/html/latest/livepatch/shadow-vars.html.

[3] Nadav Amit, Amy Tai, and Michael Wei. 2020. Don't Shoot down TLB Shootdowns!. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*. Virtual, 14.

[4] Apple. 2020. Small chip. Giant leap. (2020). https://www.apple.com/mac/m1/.

[5] ARM. [n. d.]. Processing Architecture for Power Efficiency and Performance. ([n. d.]).

[6] ARM. 2016. The ARMv8-A Architecture Reference Manual. (2016). http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html.

[7] Jens Axboe. 2019. Ringing in a new asynchronous I/O API. (2019). https://lwn.net/Articles/776703/.

[8] Ashish Bijlani and Umakishore Ramachandran. 2019. Extension Framework for File Systems in User space. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. Renton, WA, 121–134.

[9] Anton Blanchard. 2013. will-it-scale. (2013). https://github.com/antonblanchard/will-it-scale.

[10] Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price. 1979. The Convoy Phenomenon. *SIGOPS Oper. Syst. Rev.* 13, 2 (April 1979), 20–25.

[11] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M. Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. 2008. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, CA.

[12] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Vancouver, Canada, 1–16.

[13] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2012. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*. Ottawa, Canada.

[14] B. B. Brandenburg and J. H. Anderson. 2011. Spin-based reader-writer synchronization for multiprocessor real-time systems. In *Real Time Systems*. 184–193. https://doi.org/10.1109/ECRTS.2009.14

[15] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2020. hXDP: Efficient Software Packet Processing on FPGA NICs. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Virtual, 973–990.

[16] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. 2013. NUMA-aware Reader-writer Locks. In *Proceedings of the 18th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. Shenzhen, China, 157–166.

[17] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking Software Runtimes for Disaggregated Memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[18] Jonathan Corbet. 2015. User-space page fault handling. (2015). https://lwn.net/Articles/636226/.

[19] Ian Cutress. 2020. Intel Alder Lake: Confirmed x86 Hybrid with Golden Cove and Gracemont for 2021. (2020). https://www.anandtech.com/show/15979/intel-alder-lake-confirmed-x86-hybrid-with-golden-cove-and-gracemont-for-2021.

[20] Dave Dice and Alex Kogan. 2019. BRAVO: Biased Locking for Reader-Writer Locks. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. USENIX Association, Renton, WA, 315–328.

[21] Dave Dice and Alex Kogan. 2019. Compact NUMA-aware Locks. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, New York, NY, USA, Article 12, 15 pages.

[22] David Dice, Virendra J. Marathe, and Nir Shavit. 2012. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. New Orleans, LA, 247–256.

[23] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*. Copper Mountain, CO.

[24] Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. New Orleans, LA, 257–266.

[25] Thomas Friebel. 2008. *How to Deal with Lock-Holder Preemption*. Technical Report. Xen Summit.

[26] Brendan Greggs. 2019. BPF: A New Type of Software. (2019). http://www.brendangregg.com/blog/2019-12-02/bpf-a-new-type-of-software.html.

[27] Gregory Haskins. 2008. seqlock: serialize against writers. (2008). https://lwn.net/Articles/296209/.

[28] IETF. 2007. A Remote Direct Memory Access Protocol Specification. (2007). https://tools.ietf.org/html/rfc5040.

[29] Intel. 2016. Introduction to the Storage Performance Development Kit (SPDK). (2016). https://software.intel.com/content/www/us/en/develop/articles/introduction-to-the-storage-performance-development-kit-spdk.html.

[30] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. 2019. Performance Contracts for Software Network Functions. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, MA, 517–530.

[31] Seth Jennings. 2014. Kernel Live Patching. (2014). https://lwn.net/Articles/619390/.

[32] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. Ontario, Canada.

[33] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. 2019. Scalable and Practical Locking With Shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. Ontario, Canada.

[34] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2015. Scalability In The Clouds! A Myth Or Reality?. In *Proceedings of the 6th Asia-Pacific Workshop on Systems (APSys)*. Tokyo, Japan.

[35] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Scalable NUMA-aware Blocking Synchronization Primitives. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*. Santa Clara, CA.

[36] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2018. Scaling Guest OS Critical Sections with eCS. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*. Boston, MA.

[37] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. 2017. Enlightening the I/O Path: A Holistic Approach for Application Performance. In *15th USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, CA, 345–358.

[38] Calvin Loncaric, Michael D. Ernst, and Emina Torlak. 2018. Generalized Data Structure Synthesis. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. Gothenburg, Sweden, 958–968.

[39] Madhavapeddy, Anil and Scott, David J. 2014. Unikernels: The Rise of the Virtual Library Operating System. *Commun. ACM* (Jan. 2014), 61–69.

[40] Paul E. McKenney, Jonathan Appavoo, Andy Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. 2002. Read-Copy Update. In *Ottawa Linux Symposium (OLS)*.

[41] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (Feb. 1991), 21–65.

[42] Changwoo Min, Woon-Hak Kang, Mohan Kumar, Sanidhya Kashyap, Steffen Maass, Heeseung Jo, and Taesoo Kim. 2018. SOLROS: A Data-Centric Operating System Architecture for Heterogeneous Computing. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*. Porto, Portugal.

[43] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. 2016. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*. Denver, CO.

[44] Ingo Molnar and Davidlohr Bueso. 2017. Generic Mutex Subsystem. (2017). https://www.kernel.org/doc/Documentation/locking/mutex-design.txt.

[45] David Mulnix. 2017. Intel Xeon Processor Scalable Family Technical Overview. (2017). https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview.

[46] Yuvraj Patel, Leon Yang, Leo Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. 2020. Avoiding Scheduler Subversion Using Scheduler-Cooperative Locks. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*. Virtual, 17.

[47] Andy Patrizio. 2017. HPE refreshes its Superdome servers with SGI technology. (2017). https://www.networkworld.com/article/3236789/hpe-refreshes-its-superdome-servers-with-sgi-technology.html.

[48] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado.

[49] The Linux Foundation Projects. 2021. DPDK. (2021). https://www.dpdk.org/.

[50] RedHat. 2021. Achieving high-performance, low-latency networking with XDP. (2021). https://developers.redhat.com/blog/2018/12/06/achieving-high-performance-low-latency-networking-with-xdp-part-1/.

[51] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. USA. Advisor(s) Bodik, Rastislav. AAI3353225.

[52] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. 2013. Schedule Processes, Not VCPUs. In *Proceedings of the 4th Asia-Pacific Workshop on Systems (APSys)*. 7.

[53] Linus Torvalds. 2005. Linux Wait Queues. (2005). http://www.tldp.org/LDP/tlk/kernel/kernel.html.

[54] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. 2011. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*. Portland, OR, 11–11.

[55] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. 2004. Towards Scalable Multiprocessor Virtual Machines. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3 (VM)*. 14.

[56] Peter Zijlstra. 2003. lockstat: documentation. (2003). https://lwn.net/Articles/252835/.