Parsec: Fast, Scalable, and Secure Design with Wait-Free Parallelism

Ruslan Nikolaev
Pennsylvania State University
University Park, USA
rnikola@psu.edu

Abstract

In this paper, we look at the problem of concurrent progress from an unconventional perspective and find new benefits for practical wait-freedom by considering security and reliability implications of concurrent data structures. Although lock-free data structures are increasingly used in many system applications, they can be prone to denial-of-service (DoS) attacks and pose some dilemmas for a user. For example, when can we safely conclude that one side of a communication channel is buggy or malicious? What should we do about temporary slowdowns due to scheduling peculiarities?

We observe that when using wait-free approaches, these questions can be fully resolved due to strict theoretical upper bounds of wait-free algorithms; e.g., all loops are finite and the worst-case number of iterations is known beforehand. This discussion is crucial since other existing mechanisms, such as read-copy-update (RCU), are also quite problematic from the security standpoint. Moreover, even some well-known hardware primitives are less resilient to DoS attacks. We show that with recent advancements in lock- and wait-free algorithm design, many past challenges can now be fully overcome, potentially making such data structures more appealing and easier to use than present RCU equivalents.

We present real-life examples that show benefits of wait-free synchronization using Go-like channels. Our overheads with respect to lock-free synchronization remain fairly low while advantages with respect to more traditional fine-grained locking are visible even under fairly moderate contention due to a reduction of synchronization system calls.

CCS Concepts

• **Security and privacy** → *Denial-of-service attacks*; Operating systems security; • **Theory of computation** → **Concurrent algorithms**.

APSys '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1572-3/25/10
https://doi.org/10.1145/3725783.3764410

Keywords

wait-free, lock-free, RCU, denial-of-service attacks, Linux

ACM Reference Format:
Ruslan Nikolaev. 2025. Parsec: Fast, Scalable, and Secure Design with Wait-Free Parallelism. In *16th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '25), October 12–13, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3725783.3764410

1 Introduction

In the past, blocking synchronization in system applications based on mutual exclusion sufficed. But even basic systems today are becoming increasingly multicore, necessitating better OS parallelism support. Thus, non-blocking synchronization, which allows manipulation of the same data structure without mutual exclusion, becomes increasingly popular.

With blocking synchronization, two well-known problems need to be addressed: deadlocks and starvation. Starvation-freedom also implies deadlock-freedom, as the former provides much stronger guarantees. In the same vein, non-blocking synchronization defines various progress guarantees such as *lock-freedom* and *wait-freedom*. In simple terms, wait-freedom, effectively, implies starvation-freedom but within the context of non-blocking algorithms.

Historically, wait-free concurrent algorithms were much slower than their (much simpler) lock-free counterparts, making them impractical in most use cases. However, fast-path-slow-path methods [16, 24, 32], which combine wait-free progress guarantees with great performance characteristics of lock-free algorithms by effectively using lock-free algorithms most of the time, somewhat changed that perception. That said, ordinary lock-free algorithms are still far more common due to their smaller overall complexity.

Wait-free synchronization is widely noted for its great latency characteristics that withstand adverse hardware and software (e.g., an unfair OS scheduler) conditions by strictly bounding the number of iterations for *any* thread under any circumstances [13]. In this paper, we go a step further and make an additional crucial insight, which did not receive a proper attention in the literature previously. We argue that wait-freedom is *instrumental from the security standpoint* and should be adopted more widely in system applications.

In this paper, we also analyze commonly used approaches such as read-copy-update (RCU) [18] and hardware instructions with respect to their resiliency to DoS attacks. We find that typical RCU usage in the Linux kernel is very prone to DoS attacks. Moreover, we argue against the use of load-link (LL) and store-conditional (SC) instructions, implemented in some RISC architectures, to provide strict theoretical guarantees against DoS attacks.

2 Background and Related Work

Various in-memory data structures (queues, lists, hash tables, skip lists, trees, etc) are used widely in system applications, e.g., OS kernels, file systems, and network applications.

2.1 Read-Copy-Update (RCU)

The Linux kernel widely uses RCU [18]: it allows unhindered reader parallelism but typically requires mutual exclusion for writers (unless writer operations are trivial). RCU is especially beneficial in read-mostly data structures, where it avoids locking on the read path. However, preemption increases memory footprints in RCU data structures dramatically [26], which is a problem when running an OS in a virtual machine, especially in a multi-tenant cloud environment. This problem may occur even on a bare metal hardware, e.g., in the presence of buggy code or an adversary.

2.2 Non-blocking Approaches

Fine-grained locking and RCU are still *blocking* approaches. Alternative, *non-blocking*, approaches can be categorized into three types. *Obstruction-freedom* ensures that a thread always makes progress when executing without interference from other threads. *Lock-freedom* permits thread interference and guarantees that *at least* one thread always makes progress after a finite number of steps. Finally, *wait-freedom*, the strongest non-blocking guarantee, ensures that *all* threads complete their operations after a finite number of steps, meaning no thread will starve. Historically, wait-free concurrent algorithms presented only *theoretical* interest due to their complexity and lackluster *practical* performance.

While progress properties mainly focus on CPU resources, an algorithm may also become blocking when memory runs out: the process is stalled until the OS reclaims memory. Thus, RCU is blocking *even* if writers avoid mutual exclusion as RCU's underlying mechanism does not bound memory usage (even though Linux implements some practical mitigations).

2.3 Atomic Instructions

Compare-and-swap (CAS), known as `cmpxchg` in x86-64, an instruction which *atomically* reads a memory word, compares it with the expected value, and exchanges it with the desired value, is used almost universally in non-blocking synchronization. An alternative pair of instructions, load-link (LL) / store-conditional (SC), which splits the loading and writing phases but still guarantees atomicity, is preferred by RISC-V [28], MIPS [4], and POWER [9]. Some CPUs such as x86-64, RISC-V, and ARM64 [5] implement fetch-and-add (FAA), an instruction that atomically increments a shared memory value and returns a previous value, which is less powerful than either CAS or LL/SC but scales better [21].

While LL/SC is considered to be superior to CAS in the literature [13] due to its flexibility, it is often used to simply implement CAS, FAA, and other atomic primitives in assembly, as LL/SC cannot be exposed to high-level languages [1].

C11 [3] and C++11 [2] expose CAS, FAA, and other operations. Since some CPUs implement CAS via LL/SC, and LL/SC can fail due to external events such as interrupts, there are two versions of CAS: *weak CAS* which can sporadically fail, and *strong CAS* which can only fail if the memory word does not match the expected value. Strong CAS wraps around weak CAS in a loop which repeats the same operation until CAS either succeeds or fails due to the value mismatch. We argue that architectures *must* support (strong) CAS natively and avoid LL/SC altogether to prevent DoS attacks.

2.4 Practical Wait-Free Methods

A number of approaches exist for building wait-free data structures. Notably, Kogan & Petrank's *fast-path-slow-path* method [16] uses a lock-free procedure for the fast path, taken most of the time, and falls back to a wait-free procedure if the fast path does not succeed after a small number of tries. The original method is quite general but can only be used with CAS, and consequently does not always scale well. A recent paper [24] successfully created a custom method for FAA, an instruction commonly used in non-blocking queues [20, 24, 29, 32] for better scalability.

Although some challenges for wait-free data structures remain, both the original and custom fast-path-slow-path methods demonstrate that efficient wait-free data structures *can* be created. We anticipate that even more wait-free data structures will appear in the coming years, making it crucial to consider them in the context of system applications.

3 Security Analysis

In this section, we analyze resilience of different blocking and non-blocking approaches to DoS attacks.

3.1 RCU Challenges

The Linux kernel defines several major API functions for RCU presented in Figure 1. Each reader calls (non-blocking) `rcu_read_lock` and `rcu_read_unlock`. RCU needs to keep track of active readers because allocated memory has to

```
// Reader methods
rcu_read_lock();    // before critical section
rcu_read_unlock(); // after critical section
// Writer methods
synchronize_rcu();     // a blocking call
call_rcu();            // a non-blocking call
rcu_assign_pointer(); // advertise a pointer
// Common methods
rcu_dereference();     // retrieve a pointer
```

Figure 1: Linux's RCU API.

be released to the OS when a writer decides to deallocate memory that pertain to the data structure. However, memory cannot be freed instantaneously as concurrent readers may still access the same memory via stale pointers. RCU's idea is that memory has to be reclaimed with some delay, i.e., when *all* in-flight readers that *potentially* access previously allocated memory are no longer in the critical section.

The main challenge for writers is to decide how to free memory. Sometimes, writers can just wait (block) until all readers are done, and a special call, `synchronize_rcu`, guarantees that memory can be safely reclaimed. Waiting, however, is not always acceptable, and writers may prefer to use `call_rcu`, which frees memory via a special callback function. With `call_rcu`, there is no delay, and writers are not blocked. However, no memory will actually be reclaimed when just one reader is indefinitely stalled [19]. When memory is fully exhausted, the OS will crash.

RCU documentation [8] recognizes `call_rcu`'s problem and states that `synchronize_rcu` is more resilient to DoS attacks as it naturally slow downs data structure updates. However, this is a double-edged sword: limited update frequency is achieved by a high latency of `synchronize_rcu` (at least 1 jiffy), which causes massive slowdowns of several milliseconds [12], unacceptable in many cases. Thus, Linux also supports `synchronize_rcu_expedited`, a special version of the operation for which latency is reduced to less than 1 microsecond [12]. However, the reduced latency automatically increases the chance of DoS attacks. Moreover, an adversary can first mount a control-flow attack by delaying or removing `rcu_read_unlock`, as shown in Figure 2, causing a subsequent DoS attack. With `synchronize_rcu`, writers will simply hang, whereas with `call_rcu`, writers will potentially exhaust all system memory.

We analyzed Linux 6.4.11 and found 495 of `call_rcu`, 19 of `synchronize_rcu_expedited`, and 403 of `synchronize_rcu` occurrences. Thus, > 50% of the code is DoS-susceptible.

The sheer complexity of the RCU API in Linux is also a concern: `synchronize_rcu` and `synchronize_rcu_expedited` are just some examples of highly-specialized calls. Moreover, there is also a "sleepable" RCU version (srcu) which mitigates (but does not fully solve) the problem with stalled threads.

```
struct foo { struct rcu_head rh; };
struct foo *g;

void reader() {
  rcu_read_lock();
  cur_mem = rcu_dereference(g);
  ... // control-flow attack: unlock is
  rcu_read_unlock(); // skipped or delayed
}

void writer_block() {
  new_mem = malloc(sizeof(struct foo));
  old_mem = rcu_dereference(g);
  rcu_assign_pointer(g, new_mem);
  synchronize_rcu(); // Blocks indefinitely!
  kfree(old_mem);     // Not reachable
}

void writer_nonblock() {
  for (i = 0; i < count; i++) {
    new_mem = malloc(sizeof(struct foo));
    old_mem = rcu_dereference(g);
    rcu_assign_pointer(g, new_mem);
    call_rcu(&old_mem->rh, callback_kfree);
    // Exhausts memory because it allocates
    // new memory without releasing anything!
} }
```

Figure 2: DoS attack for RCU.

The burden of knowing intricate RCU details falls largely on the programmer. Unsurprisingly, RCU bugs and vulnerabilities are quite common, e.g., CVE-2024–27394 [27] is a recently discovered race condition bug in the TCP stack.

3.2 Non-blocking Data Structures

The main problem with RCU is that it is blocking and does not bound memory usage, which opens various venues for DoS attacks. Non-blocking approaches that bound memory usage seem to be a great alternative to prevent DoS attacks.

Let us consider a common [15, 17, 22] scenario when a non-blocking data structure is used as a communication channel between any two entities that need to be isolated from each other (two user-space processes, a user-space process and the OS kernel, the OS kernel's core and a driver, etc.) as shown in Figure 3. In this scenario, the data structure resides in the shared memory, but both entities are not trusting each other: they verify the consistency of the data structure at each step (validity of pointers, array indices, etc). However, this is insufficient, e.g., one entity can create cyclic lists with valid pointers such that a second-entity operation will be unable to complete in a finite number of steps.

Clearly, *obstruction-free* algorithms are vulnerable to such DoS attacks since one entity can deliberately interfere with the other one even without damaging the data structure. Common *lock-free* data structures such as queues are typically operation-wise lock-free [20] (e.g., at least one thread enqueues and at least one thread dequeues simultaneously).

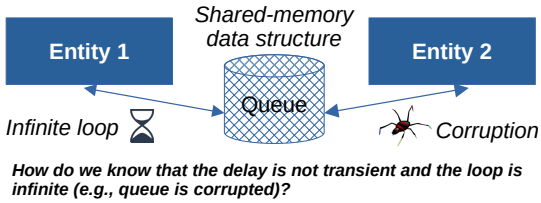

Figure 3: Challenge: Slow-down vs. Damage.

However, such guarantees are only present as long as the data structure is not corrupted. Thus, lock-free data structures will have to explicitly limit the number of iterations to detect DoS attacks. However, it is unclear where to draw the line here. First, even in non-adversarial scenarios, a *given* thread may *theoretically* never complete due to starvation. Due to randomness, this scenario is unlikely in practice but is still possible for a poorly-designed lock-free algorithm. However, an attacker may deliberately invalidate L1 cache lines for the corresponding shared-memory region to artificially slow down CAS operations of one thread to reduce the overall randomness and make such attacks realistic even for well-designed lock-free algorithms. Second, tail latency of lock-free algorithms can be quite substantial, making the detection process tricky and slow. Loops should not be terminated prematurely to avoid false-positives, and CPUs can easily be hogged for several seconds even when the number of iterations is in the range of 100,000-1,000,000. That will, effectively, slow down the system and thus still achieve the overall goal of a DoS attack.

Wait-free data structures stipulate finite number of steps which require inter-thread cooperation to avoid CPU monopolization. In Figure 3, the entities will perform rigorous safety checks at every step, especially because threads (from either of the two entities) now collaborate and use shared state. The challenge here is to implement an efficient algorithm, which provides comparable performance to state-of-the-art lock-free counterparts. As discussed in Section 2, this is attained with fast-path-slow-path methods. A theoretical bound for the worst-case number of (fast- and slow-path) iterations is known. If that bound is exceeded, an entity can safely conclude that the data structure is corrupted by the other side. Unlike lock-free counterparts, this condition is detected *much* sooner as thresholds for the fast-path methods do not need to consider the worst-case time for slower threads.

To the best of our knowledge, wait-free synchronization was not explicitly considered in the OS context for security reasons. Most non-blocking data structures in use are lock-free at best. Moreover, the concurrency literature focuses on benefits of wait-free algorithms for performance reasons only (i.e., reduced tail latency) and overlooks security.

We note that some well-known systems already use DoS-safe data structures, though their scalability is limited. For example, Xen [11] uses a single-producer and single-consumer I/O ring buffer for backend and frontend drivers. This ring buffer uses two independent locks for producers and consumers: one producer and one consumer can run in parallel and independently. Though this ring buffer is not scalable, it is still better than lock-free approaches for security reasons. Adopting a wait-free ring buffer would have been even better: it solves performance issues without sacrificing security.

3.3 LL/SC Vulnerability

LL/SC goes in line with the RISC design philosophy: it simplifies the instruction set and provides greater flexibility when implementing atomic operations. A CPU core atomically reserves a word and loads from memory with the LL instruction. Then, a user performs desired operations. Finally, the same memory word from the original reservation is overwritten by SC. If another core modifies the word during this time, the reservation is lost, and SC will fail, leaving the memory unchanged. When SC fails, the operation has to be repeated. LL/SC can be used to implement CAS, FAA, and many other atomic operations.

The main problem with LL/SC is that it is not *one* instruction with finite execution time. With LL/SC, *sporadic* failures are possible, i.e., when no CPU core can successfully modify the same memory word. First, since the reservation granule is typically an L1 cache line, the same cache line modification can already cause sporadic failures even if the actual memory word is not modified. Second, any external event (e.g., an OS interrupt) can cause SC to fail. An attacker may deliberately invalidate the same cache line using a tight loop with memory writes in a dedicated thread, which is much shorter than the corresponding LL/SC loop and will cause SC to incessantly fail until the offending thread is preempted. An attacker may also trigger fake OS events to make SC fail.

Unfortunately, even when there is no adversarial intent, it is hard to predict sporadic failures in many cases. For example, suboptimal code can use false sharing, which is not that uncommon in practice. Also, although interrupts can be disabled in kernel space, this is much harder to control in user space, and there is still no architectural guarantee that sporadic failures are fully prevented even when interrupts are disabled and there is no false sharing.

For lock-free algorithms, a common approach is to use weak CAS and strong CAS built from LL/SC, as in C11 [3] and C++11 [2]. Since a particular thread can potentially starve, strong CAS, which may have an *infinite loop* in case of incessant sporadic failures does not *generally* contradict lock-free progress guarantees, as at least one other thread still makes progress. It may, however, come as an unpleasant surprise

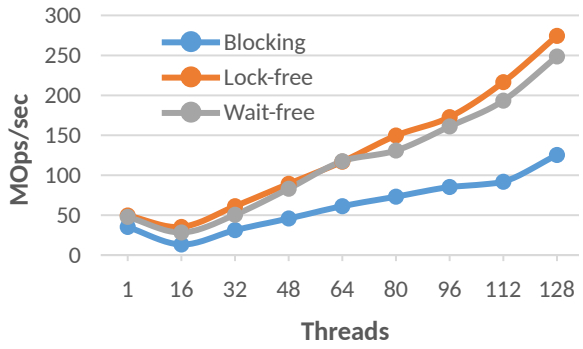

Figure 4: Throughput of Go-like channels.

```
1 bool StrongCAS(long *mem, long *expect, long new)
2     do // DANGER: Potentially infinite loop!
3         old = LL(mem);
4         if ( old != *expect )
5             *expect = old;
6             return false;
7     while !SC(mem, new);
8     return true;
```

Figure 5: Strong CAS vulnerability.

to a programmer who reasonably expects that all CAS operations take finite time (either succeed or fail). An attacker may take advantage of this misconception and mount a DoS attack (Figure 5).

The above is also generally applicable to wait-free algorithms. But we also must maintain a strict theoretical bound which is now diluted by sporadic weak CAS failures, none of which can easily be predicted. Thus, strictly speaking, we cannot implement wait-free algorithms via LL/SC.

We thus argue that LL/SC is harmful and must be completely avoided irrespective of the CPU design philosophy (RISC vs. CISC). Instead, modern architectures must provide native CAS similar to x86-64 and recent revisions of ARM64. Only native CAS *simultaneously* guarantees finite execution time and lack of sporadic failures, obviating the need for weak CAS and strong CAS.[1] Although LL/SC provides greater flexibility that allows efficient implementation of FAA and similar operations, the same security problem exists for them. In fact, FAA implemented via LL/SC is always dangerous from the security perspective similar to strong CAS, and there is no "weak FAA" in C11/C++11. Modern architectures can provide native FAA which is similar to that of x86-64 and recent revisions of ARM64, though native FAA is not *fundamentally* required to build correct wait-free algorithms: it is only used for better performance.

Sometimes, it is argued [13] that LL/SC prevents the ABA problem. In practice, this advantage is rarely used since LL/SC is not exposed to high-level languages directly [1]. With CAS, we can also solve this problem by placing an adjacent tag which is monotonically incremented by one whenever the corresponding pointer changes. This solves false-positive CAS matches but requires CAS2, a special instruction that compares and exchanges two *adjacent* words simultaneously. CAS2, which is supported by the latest revisions of x86-64 and ARM64, must also be implemented for completeness by modern architectures.

We believe that the above arguments will be important to hardware designers, especially because LL/SC's simplicity and ABA-safety were primary motivating factors for RISC-V's initial adoption of LL/SC [28], a decision which has already been partially re-evaluated by introducing both CAS and CAS2 in RISC-V's "Zacas" extension [28]. Note that the compiler preference of native CAS instructions for ARM64 is still sometimes debated [14], and we hope that our paper resolves this debate by looking also from the security angle.

4 Preliminary Evaluation

We implemented buffered channels in C, a popular concurrency primitive in Go [6]. Our approach is typical and uses semaphores to synchronize senders with receivers, as well as a mutex for the (internal) buffer. We then integrated a lock-free buffer [21] to avoid buffer locking. Finally, we integrated a more advanced wait-free buffer [24]. It is worth noting that in the latter two cases, we safely remove mutual exclusion (for the buffer) on the fast path. As a result, since semaphores are implemented via futuxes [7], system calls are only needed when a sender or receiver needs to sleep.

We ran the tests on AMD EPYC 9554 64-core machine (up to 3.75 GHz), which has 128 hardware threads and 384 GiB of RAM. Every test point runs for 10 seconds (5 times). We allow up to 512 messages in channels. Every thread sends data to or receives data from its own channel as well as from the neighboring channel for the next thread. Thus, at most *two threads* access every channel irrespective of the total number of threads, which is quite favorable for a lock-based version. Despite low contention, in Figure 4, both the lock- and wait-free implementations already show a performance boost of 2x-3x, which we attribute to the reduced number of system calls for synchronization. (The gap would have been even higher if more than two threads accessed the same

[1] This is another problem since a user reasonably expects that CAS takes finite time, which is not guaranteed for strong CAS implemented via LL/SC.

channel.) Moreover, the overhead of the wait-free approach is low, and the gap between lock-/wait-free and blocking versions grows with the increased number of threads.

5 Future Work

The proposed wait-free synchronization can benefit existing software such as the Linux kernel, Xen [11], DPDK [31], and SPDK [30], which use concurrent data structures widely.

Although RCU is far from being ideal, its use in Linux is no accident: RCU has a simple and fully integrated *safe memory reclamation* methodology, one of the key problem that concurrent data structures must solve. Unfortunately, the reclamation method used in RCU is blocking: it exhausts system memory when one thread is indefinitely stalled [19].

Aside from using wait-free synchronization in Go-like channels, shared-memory communication mechanisms that span multiple independent entities, it would also be advisable to adopt wait-free synchronization for ordinary in-kernel data structures in lieu of RCU. A critical piece for wait-free data structures is wait-free memory reclamation that guarantees bounded memory usage. This problem was already addressed in [23, 25], removing the remaining obstacle for wait-free synchronization. Moreover, wait-free reclamation can also help to build *a new generation* of RCU to properly address its present issues.

API Complexity and Flexibility. Linux offers a zero-copy API for managing lists through "list.h", which uses locks, and "rculist.h", which relies on RCU; both are extensively utilized across various device drivers. In contrast, "freelist.h", labeled as "lockless" provides a much more restricted list implementation that deviates significantly from the API used by the other two. However, recently, it has been demonstrated [10], that zero-copying is feasible for lock-free algorithms. The new API is even closer to "list.h" than "rculist.h": RCU still puts many restrictions on how things can be moved around. By adapting this approach to wait-free algorithms, we may simplify the usage complexity of RCU-based algorithms.

Challenges. One challenge with wait-free synchronization is the need to provision additional per-thread memory for every data structure when using fast-path-slow-path methods. Although the amount of this memory is negligible, this is not required with more traditional RCU and lock-free data structures. The OS kernel must be extended with the infrastructure to allocate, free, and access per-thread regions on demand. This, in turn, must also be done in a wait-free manner.

6 Conclusions

In this paper, we analyzed various blocking and non-blocking approaches with respect to their resiliency to DoS attacks. We advise better vigilance when using RCU. Despite that the possibility of DoS attacks is well-known for RCU, we find little evidence that programmers take any tangible steps to prevent them. Furthermore, we make a case for a much wider adoption of wait-free synchronization, which will improve both performance (latency) and DoS resiliency of shared data structures. We also analyzed different hardware primitives, and concluded that LL/SC must be entirely avoided irrespective of any perceived benefit for RISC architectures.

To the best of our knowledge, this paper is the first one to analyze these issues in a more or less systematic way, which can help future systems researchers to fully address or mitigate security challenges in future CPU architectures, OS designs, and applications.

Availability

Code relevant to this paper is available at https://github.com/rusnikola/parsec.

Acknowledgments

The author thanks Prof. Timothy Zhu, a colleague at Penn State, for inspiring the idea of implementing *lock-based* Go-like channels in C (a course project assigned to students). The author developed channels in C using lock-free and wait-free synchronization methods, that avoid locks on the fast path, presented as an example in this paper.

References

[1] 2006. Linux Mailing List: LL/SC Discussion. https://yarchive.net/comp/linux/cmpxchg_ll_sc_portability.html.
[2] 2011. ISO/IEC 14882:2011, Information technology - Programming languages - C++. https://www.iso.org/standard/50372.html.
[3] 2011. ISO/IEC 9899:2011, Information technology - Programming languages - C. https://www.iso.org/standard/57853.html.
[4] 2016. MIPS32/MIPS64 Revision 6.06. https://www.mips.com/products/architectures/.
[5] 2024. ARM Architecture Reference Manual ARMv8.1. https://developer.arm.com/.
[6] 2024. Channels in Go. https://go101.org/article/channel.html.
[7] 2024. futex(2) – Linux manual page. https://man7.org/linux/man-pages/man2/futex.2.html.
[8] 2024. Linux Documentation: What is RCU? https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt.
[9] 2024. PowerPC Architecture Book, Version 2.02. Book I: PowerPC User Instruction Set Architecture. https://www.ibm.com/developerworks/.
[10] Md Amit Hasan Arovi and Ruslan Nikolaev. 2025. RRR-SMR: Reduce, Reuse, Recycle: Better Methods for Practical Lock-Free Data Structures. In *Proceedings of the 2025 ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Seoul, South Korea) *(PLDI '25)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3729337
[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP'03)*. 164–177.

[12] Joel Fernandes, Uladzislau Rezki, and Rushikesh Kadam. 2022. Make RCU do less (& later). In *Linux Plumbers Conference 2022*. https://lpc.events/event/16/contributions/1204/attachments/985/1937/Make%20RCU%20do%20less%20(and%20later)!%20(1).pdf

[13] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2020. *The Art of Multiprocessor Programming* (2nd ed.).

[14] Ricardo Jesus and Michèle Weiland. 2023. POSTER: AArch64 Atomics: Might They Be Harming Your Performance?. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 419–421. doi:10.1145/3572848.3579838

[15] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. 2014. OSv - Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'14)*. 61.

[16] Alex Kogan and Erez Petrank. 2012. A Methodology for Creating Fast Wait-free Data Structures. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) *(PPoPP '12)*. ACM, New York, NY, USA, 141–150. http://doi.acm.org/10.1145/2145816.2145835

[17] 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. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP'19)*. 399–413.

[18] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, O. Krieger, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. 2001. Read-Copy Update. In *Ottawa Linux Symposium*. 338–367.

[19] Maged M. Michael, Michael Wong, Paul McKenney, Arthur O'Dwyer, David Hollman, Geoffrey Romer, and Andrew Hunter. 2017. Hazard Pointers: Safe Resource Reclamation for Optimistic Concurrency, Project: Programming Language C++, SG14/SG1 Concurrency, LEWG, P0233R4. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0233r4.pdf.

[20] Adam Morrison and Yehuda Afek. 2013. Fast Concurrent Queues for x86 Processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. ACM, New York, NY, USA, 103–112.

[21] Ruslan Nikolaev. 2019. A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue. In *Proceedings of the 33rd International Symposium on Distributed Computing, DISC 2019 (LIPIcs, Vol. 146)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:16.

[22] Ruslan Nikolaev and Godmar Back. 2013. VirtuOS: An Operating System with Kernel Virtualization. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. 116–132.

[23] Ruslan Nikolaev and Binoy Ravindran. 2020. Universal Wait-Free Memory Reclamation. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 130–143. doi:10.1145/3332466.3374540

[24] Ruslan Nikolaev and Binoy Ravindran. 2022. WCQ: A Fast Wait-Free Queue with Bounded Memory Usage. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures* (Philadelphia, PA, USA) *(SPAA '22)*. Association for Computing Machinery, New York, NY, USA, 307–319. doi:10.1145/3490148.3538572

[25] Ruslan Nikolaev and Binoy Ravindran. 2024. A Family of Fast and Memory Efficient Lock- and Wait-Free Reclamation. In *Proceedings of the 2024 ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Copenhagen, Denmark) *(PLDI '24)*. Association for Computing Machinery, New York, NY, USA, 235:1–235:25. doi:10.1145/3658851

[26] Aravinda Prasad, K. Gopinath, and Paul E. McKenney. 2017. The RCU-Reader Preemption Problem in VMs. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, USA, 265–270.

[27] Theori Vulnerability Research. 2024. Deep Dive into RCU Race Condition: Analysis of TCP-AO UAF (CVE-2024–27394). https://blog.theori.io/deep-dive-into-rcu-race-condition-analysis-of-tcp-ao-uaf-cve-2024-27394-f40508b84c42.

[28] RISC-V International. 2024. Ratified Specification. https://riscv.org/specifications/ratified/.

[29] Raed Romanov and Nikita Koval. 2023. The State-of-the-Art LCRQ Concurrent Queue Algorithm Does NOT Require CAS2. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 14–26. doi:10.1145/3572848.3577485

[30] SPDK Contributors. 2024. Storage Performance Development Kit (SPDK). https://spdk.io/.

[31] The Linux Foundation. 2024. Data Plane Development Kit (DPDK). https://dpdk.org/.

[32] Chaoran Yang and John Mellor-Crummey. 2016. A Wait-free Queue As Fast As Fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. ACM, New York, NY, USA, Article 16, 13 pages. doi:10.1145/2851141.2851168