CLoF: A Compositional Lock Framework for Multi-level NUMA Systems

Rafael Lourenco de Lima Chehab[*] [†]
Huawei Dresden Research Center
Technische Universität Dresden
Germany
rafael.chehab@huawei.com

Antonio Paolillo[*] [†]
Huawei Dresden Research Center
Germany
antonio.paolillo@huawei.com

Diogo Behrens[†]
Huawei Dresden Research Center
Germany
diogo.behrens@huawei.com

Ming Fu[†] [‡]
Huawei Dresden Research Center
Germany
ming.fu@huawei.com

Hermann Härtig
Technische Universität Dresden
Germany
hermann.haertig@tu-dresden.de

Haibo Chen
Huawei OS Kernel Lab
Shanghai Jiao Tong University
China
hb.chen@huawei.com

Abstract

Efficient locking mechanisms are extremely important to support large-scale concurrency and exploit the performance promises of many-core servers. Implementing an efficient, generic, and correct lock is very challenging due to the differences between various NUMA architectures. The performance impact of architectural/NUMA hierarchy differences between x86 and Armv8 are not yet fully explored, leading to unexpected performance when simply porting NUMA-aware locks from x86 to Armv8. Moreover, due to the Armv8 Weak Memory Model (WMM), correctly implementing complicated NUMA-aware locks is very difficult.

We propose a Compositional Lock Framework (CLoF) for multi-level NUMA systems. CLoF composes NUMA-oblivious locks in a hierarchy matching the target platform, leading to hundreds of correct by construction NUMA-aware locks. CLoF can automatically select the best lock among them. To show the correctness of CLoF on WMMs, we provide an inductive argument with base and induction steps verified with model checkers. In our evaluation, CLoF locks outperform state-of-the-art NUMA-aware locks in most scenarios, *e.g.*, in a highly contended LevelDB benchmark, our best CLoF locks yield twice the throughput achieved with CNA lock and ShflLock on large x86 and Armv8 servers.

[*]The first two authors contributed equally to the paper.
[†]Also with Huawei OS Kernel Lab.
[‡]Corresponding author.

CCS Concepts: • **Computer systems organization → Multicore architectures**; • **Software and its engineering → Mutual exclusion**; *Synchronization*; *Correctness*; • **Theory of computation** → *Concurrency*.

Keywords: multicore concurrency, non-uniform access memory, weak memory models

ACM Reference Format:
Rafael Lourenco de Lima Chehab, Antonio Paolillo, Diogo Behrens, Ming Fu, Hermann Härtig, and Haibo Chen. 2021. CLoF: A Compositional Lock Framework for Multi-level NUMA Systems. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3477132.3483557

1 Introduction

Many-core servers such as those powered by x86-based AMD EPYC [3] and Armv8-based Huawei Kunpeng 920 [22] processors have been widely deployed in industry. In these modern servers, CPUs are organized into clusters known as NUMA (Non-Uniform Memory Access) nodes, which are further grouped into packages. Developing efficient locking mechanisms for such systems is important for the scalability and performance of multi-threaded applications, but also very challenging. For example, threads experience a wide spectrum of memory-access latencies depending on the memory-hierarchy level in which the data is located, *e.g.*, in L1/L2/L3 caches or in the main memory of another NUMA node.

The design of high-performant locks for such multi-level NUMA systems should take the following four key aspects into account.

A1 (multi-level): Locks must support the whole memory hierarchy of the target system, not only NUMA nodes, but also packages, cache levels, and cache coloring/tagging policies. By exploiting locality in each level, lock implementations can reduce memory traffic and boost performance.

A2 (heterogeneity): Each level groups cores together, *e.g.*, 2 to 6 cores may share the same L3 partition, while dozens of threads may share the same NUMA-node memory bank. Lock algorithms can be tuned according to the contention characteristics of each level — simpler algorithms tend to be faster under low contention, but suffer under higher contention.

A3 (architecture-optimized): Different architectures allow for different optimizations, *e.g.*, coherence-traffic reductions specific to x86 but not to Armv8. Lock algorithms can be tuned according to the architectural peculiarities.

A4 (correctness): Architectures such as Armv8 implement weak memory models (WMMs), which allow aggressive reorderings of memory accesses to improve performance. To ensure correctness on WMMs, one has to carefully use memory barriers to enforce the necessary order, which is an error-prone process.

Existing NUMA-aware locks do not support multiple of these aspects (see Table 1). CNA lock [11] and ShflLock [24] only support 2-level memory hierarchies (**A1**, **A2**) and, originally, only target x86 (**A3**). The rather strong x86 TSO memory model already guarantees the necessary order of memory accesses for these algorithms without the need for barriers. On the Armv8 WMM, however, barriers are required to enforce the correct order of memory accesses (**A4**). Running them on Armv8 quickly causes hangs or mutual exclusion violations. In contrast, HMCS [6] does support multi-level NUMA-systems and provides memory barriers for WMMs. Nevertheless, it uses one specific lock implementation (*i.e.*, MCS lock [31]) for each hierarchy level (**A2**, **A3**); and the proposed memory barriers are incorrect (**A4**), causing applications to hang on Armv8 — our prior work on HMCS-WMM studies and fixes these memory barrier issues [33]. Even though HMCS supports arbitrary hierarchies, the published HMCS configurations do not exploit the potential of the cache levels. Finally, lock cohorting [15] does support *heterogeneous* locks in each level, but only works with two levels (**A1**) and does not offer guidance on memory barrier placement (**A4**).

In this work, we propose a Compositional Lock Framework (CLoF) for multi-level NUMA systems. Given a set of simple, NUMA-oblivious spinlocks verified on WMMs, CLoF generates hundreds of *heterogeneous, multi-level NUMA-aware* spinlocks (*i.e.*, CLoF locks) for a target platform, supporting the user in selecting the best one. While developing CLoF, we faced four main challenges: first, how to make a multi-level lock which allows for heterogeneity; second, how to compose the user-provided NUMA-oblivious spinlocks without requiring any adaption to them; third, how to select the best lock among the generated locks; fourth, and lastly, how to ensure the correctness of CLoF locks on WMMs.

To support heterogeneity and combine different spinlock implementations, we use compile-time *syntactic recursion*. To compose the user-provided spinlock implementations, we devise a *context abstraction* technique, which standardizes the use of spinlock interfaces. To select the best lock, we perform a weighted average of the benchmark results, biasing the weights to focus either on low or high contention scenarios. Finally, to show correctness, we exploit the recursive nature of our lock generator. Based on a set of simple but verified spinlocks for WMMs (such as those in our prior work [32]), we argue about correctness inductively by model checking one induction step of CLoF with TLA$^+$ [28] and GenMC [26].

Table 1. Key aspects coverage of recent NUMA-aware locks.

Algorithms	**A1**	**A2**	**A3**	**A4**
CNA lock [11]	✖	✖	✖	✖
ShflLock [24]	✖	✖	✖	✖
HMCS [6]	✔	✖	✖	✖
HMCS-WMM [33]	✔	✖	✖	✔
lock cohorting [15]	✖	✔	✔	✖
CLoF	✔	✔	✔	✔

✔: covered ✖: lacking

CLoF composes NUMA-oblivious spinlocks verified on WMMs in an arbitrary hierarchy, generating many correct by construction NUMA-aware locks (**A4**) for x86 and Armv8 multi-core platforms. First, CLoF precisely identify the multi-level hierarchy of the target NUMA systems through micro-benchmarks (**A1**). Next, provided a set of NUMA-oblivious spinlocks — such as Ticketlock [19], MCS lock, and CLH lock [19] — CLoF exhaustively generates hundreds of multi-level NUMA-aware locks (**A2**). Finally, CLoF benchmarks the generated locks, supporting the user in finding the best lock for a target system. Once a new NUMA-oblivious lock is designed to take advantage of an architecture-specific optimization, *e.g.*, the coherence-aware optimization of *Hemlock* [13] on x86, the process can be repeated (**A3**).

Our evaluation shows that our best CLoF-locks outperform state-of-the-art NUMA-aware locks in most contention levels: with LevelDB [9] on Armv8, between 109% to 105% better than CNA lock and ShflLock at mid and high contention; between 4% and 15% better than an equivalently configured HMCS for all contention levels. We obtain similar results on x86 and cross-validate our locks with Kyoto Cabinet [27].

Contributions. The contributions of this paper are:

- A technique to capture the multiple levels of the memory hierarchy (§3);
- Two techniques to derive the lock generator: syntactic recursive generator and context abstraction (§4.1);
- A correctness argument for CLoF (§4.2);
- An approach to select the best generated lock (§4.3).

2 Background

Spinlocks are popular synchronization mechanisms that protect access to shared data, and are widely used in operating systems and applications. In this section, we briefly introduce a set of spinlocks and concepts used in this work.

2.1 NUMA-oblivious Spinlocks

Ticketlock [19] is a simple and practical spinlock, usually consisting of two fields: ticket and grant. To acquire the lock, a thread atomically increments the ticket and, subsequently, waits for grant to equal its ticket value. To release the lock, the lock owner increments the grant field. The key benefit of Ticketlocks is that they are fair, *i.e.*, threads trying to acquire the lock do not starve — in this work, we only consider fair locks. The problem with Ticketlocks is that all threads spin on a single memory location, namely, the grant field. In systems with many cores, this *global spinning* can pressure the memory subsystem and degrade the performance. Due to that, locks with *local spinning* are preferred.

MCS lock [31] is a popular local-spinning lock (it is the base of Linux qspinlock [8]). In MCS each thread has its own node. To acquire the lock, a thread appends its node to the tail of a global queue. The head of the queue is the lock owner, while the other threads wait on memory locations of their own nodes. On release, the owner updates the node of its successor to pass the lock. Since each thread spins on its own node, the memory location is cached, and the pressure on the memory subsystem is relieved.

CLH lock [19] is another local-spinning lock, used, for example, as the big kernel lock in the seL4 microkernel [34]. It creates an indirect list where a thread spins on the node of its predecessor. On release, the owner writes a message on its own node and takes its predecessor's node along for subsequent lock acquisitions.

Other similar locks exist, and some of them try to profit from special architectural details to improve performance. *Hemlock* [13] is such a special lock, which has local spinning behavior most of the time. Similar to CLH, it uses an indirect queue, but it alters the release function such that after the owner writes on its node, the successor has to reply by resetting the value. Hemlock can employ a Coherence-Traffic Reduction (CTR) technique specific to the x86 architecture. Counterintuitively, the CTR optimization replaces a load from `x` with `fetch_add (&x, 0)` and replaces a store with a `cmpxchg`. The optimization avoids upgrades from shared to modified state in MESI and MESIF protocols, improving the lock handover performance.

All these locks have one characteristic in common: they are NUMA-oblivious and, consequently, do not scale well beyond one NUMA node. When cores from multiple NUMA nodes contend for the lock, the strict FIFO order guaranteed by these locks causes cache lines to jump back-and-forth between NUMA nodes.

2.2 NUMA-aware Locks

NUMA-awareness in locks has been studied for more than a decade [30]. In NUMA-aware locks, instead of releasing the lock in a strict FIFO order, the lock owner may prefer passing the lock to a thread whose core is in its NUMA node. This approach can reduce the cache misses of the data accessed inside the critical section, potentially improving performance. For example, CNA lock [11] is a modified MCS lock, in which the lock owner scans the MCS queue and reorders the waiting threads such that the lock is passed first to the threads from the same NUMA node. Another lock with a similar flavor is ShflLock [24], which additionally provides a configurable reordering policy (called *shuffling*).

Without major modifications, most NUMA-aware locks only support 2-level NUMA hierarchies [11, 24, 30]: system level and NUMA-node level. In contrast, *HMCS lock* [6] is a multi-level NUMA-aware lock, which creates a tree of MCS locks mirroring the configured hierarchy. For example, on an x86 platform with hyperthreading, the authors suggest a hierarchy of system level, NUMA-node level, and core level (*i.e.*, hyperthread pairs). To enter the critical section, the thread needs to acquire the ownership of the locks on its path from the leaf till the root (system level), *e.g.*, core, NUMA node, and system. To exploit locality, the lock is passed to the waiting thread, which shares most levels (*e.g.*, in the same core or in the same NUMA node), as long as the threshold at that level is not reached.

Interestingly, the authors of CNA lock and ShflLock compare their locks to HMCS lock configured with 2 levels only. With this configuration, CNA and ShflLock perform on par with HMCS$\langle 2 \rangle$. In Section 3, we show that a properly configured HMCS can greatly outperform them.

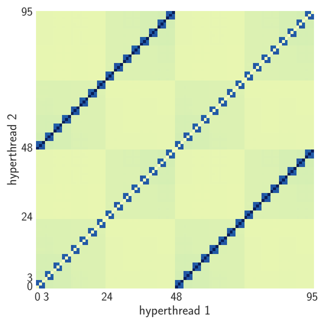
2.3 Heterogeneous NUMA-aware Locks

In HMCS lock, each level contains a set of MCS locks. In contrast, Lock Cohorting [15] is a technique that allows combining different NUMA-oblivious locks in a *2-level hierarchy*. We say locks created with a hierarchy of different locks are *level-heterogeneous*, whereas locks created with a hierarchy of identical locks are *level-homogeneous* — *e.g.*, HMCS is *level-homogeneous* as it only supports the MCS lock at each level.

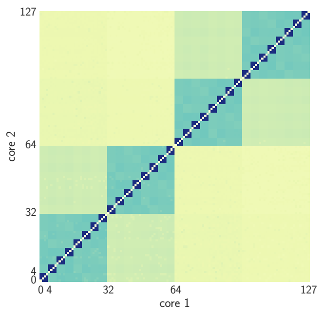
The Lock Cohorting work shows that C-BO-MCS, a level-heterogeneous lock composing backoff lock [1] and MCS, has better throughput than C-MCS-MCS, a level-homogeneous lock. Because C-BO-MCS is unfair, the authors tone down the usefulness of heterogeneity. Nevertheless, in the next section, we show that great performance improvements can be achieved by applying level-heterogeneity to a multi-level hierarchy of fair locks.

3 A Case for Multi-Heterogeneous Levels

In this section, we motivate the development of CLoF extending the four key aspects (**A1**-**A4**) discussed in the introduction. We conclude the section summarizing our results.



(a) x86 server



(b) Armv8 server

Figure 1. Throughput heatmap of two threads assigned to different CPUs. Threads atomically increment a shared counter for 1s. Absolute values are not relevant; darker tiles indicate higher throughput. In the heatmaps, one can identify the levels of the memory hierarchy of each platform.

3.1 Deep NUMA Hierarchies

Modern NUMA architectures have deep hierarchies formed by packages, NUMA nodes, cache partitions, *etc.* Unfortunately, vendors often do not expose details about cache organization to the operating system — sometimes even omit such details in the documentation. Consequently, tools such as `lscpu` in Linux only capture some hierarchy levels (*e.g.*, hyperthreads, NUMA nodes, sockets), and miss the level introduced by L3 cache tagging/partitioning.

To avoid these limitations, we propose a method to experimentally discover the full NUMA hierarchy. We build a simple benchmark that exposes the actual underlying hierarchy on any multi-core platform. For a fixed time duration (*e.g.*, 1s), two threads take turns incrementing a shared counter: Thread 1 waits the counter to be even before incrementing it; Thread 2 waits it to be odd before incrementing it. We assign threads to every possible pair of CPUs and measure the throughput (increments per second).

We run this experiment on the following two platforms: an x86 server with two 24-core packages (1 NUMA node per package), with hyperthreading enabled (2 CPUs per core). and an Armv8 server with two 64-core packages (2 NUMA nodes per package), with no hyperthreading (1 CPU per core) — see Section 5 for details.

Figure 1 shows throughput heatmaps of both servers: x-axis is the core to which Thread 1 is assigned; y-axis the core to which Thread 2 is assigned; the darker the heatmap tile, the higher the throughput — the absolute throughput value is not relevant for our investigation.

The main diagonal (both threads in same CPU) has minimal throughput in both heatmaps.[1] Around the diagonal, one can see groups of darker tiles (with much higher throughput): three cores in x86 and four cores in Armv8. These cores share the L3 cache; we call them *cache groups*. On the x86 heatmap, we can also observe two secondary diagonals (both threads in same core, different hyperthread). The throughput in this case is the highest (since they also share L1 cache). Note that the cache groups on x86 are formed by 3 cores/6 hyperthreads, *e.g.*, hyperthreads 0, 1, 2, 48, 49, and 50. Other levels can also be observed as larger, lighter boxes: the NUMA nodes around several cache groups and, in Armv8 the packages around two NUMA nodes.

[1]Since the counter is incremented alternately between threads, it only happens on a reschedule and most of the time is spent on userspace spinning, keeping the other thread from making progress.

Table 2. Throughput speedups of two threads sharing the atomic counter in the same cohort over the system cohort.

cohort	x86	Armv8
system	1.00	1.00
package	1.54	1.76
NUMA node	1.54	2.98
cache group	9.07	7.04
core	12.18	–

We observe the following:

1. On both servers, cache groups show a greater potential for performance improvement than the NUMA nodes themselves.
2. On x86, hyperthreads in the same core display the highest throughput.
3. On Armv8, the difference between NUMA node and package is substantially larger than between system and package.
4. On x86, the difference when crossing the package is the least noticeable.

A *cohort* is a group in a specific level of the NUMA hierarchy, *e.g.*, a single NUMA node is a cohort in the NUMA-node level, a single cache group is a cohort in the cache-group level. Table 2 summarizes the results obtained from the heatmaps showing the average speedup relative to the system cohort (*i.e.*, with both threads running on cores of two different packages). Naturally, the system cohort speedup is 1. For other cohorts, the closer the threads, the bigger the speedup. Note that there is no difference between package and NUMA node cohorts on x86, as there is only a single NUMA node per package. No value is provided for the core cohort on Armv8, because there is no hyperthreading on this platform.

Based on these observations, we consider the following five levels for these servers: core, cache-group, NUMA-node, package, and system.

To illustrate the performance improvements of locks exploiting these levels (aspect **A1**), we compare in Figure 2 the throughput of LevelDB [9] with several locks on the x86 server. Besides MCS lock, we evaluate the HMCS algorithm with 3 configurations:

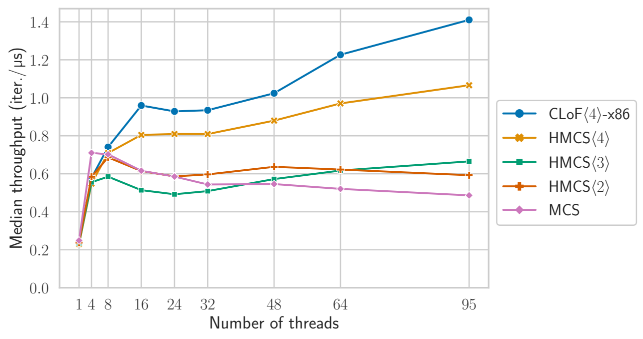


Figure 2. LevelDB with increasing contention, comparing different HMCS configurations and CLoF on x86.

- HMCS⟨2⟩: with 2-levels, NUMA-node and system — configuration used in the CNA and ShflLock [11, 24];
- HMCS⟨3⟩: with 3-levels, core, NUMA-node and system — configuration used in the original HMCS work [6];
- HMCS⟨4⟩: with 4-levels, core, cache-group, NUMA-node and system — a new configuration, to the best of our knowledge, not proposed in prior work.

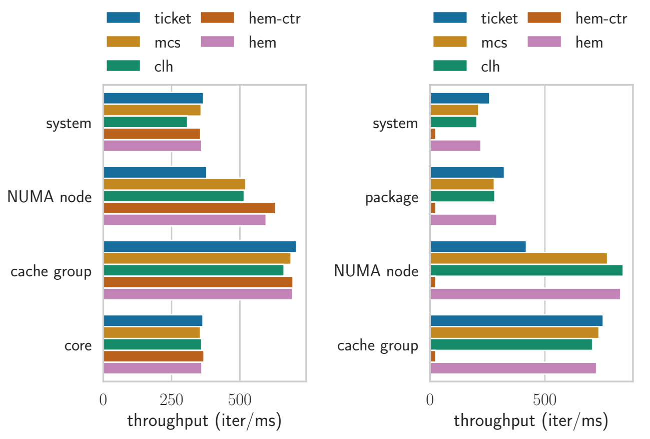
We observe that after crossing the NUMA-node level with 24 threads, HMCS⟨2⟩ outperforms MCS. HMCS⟨3⟩ initially performs worse than HMCS⟨2⟩ because, with less than 48 threads, only one hyperthread per core is used — the overhead of the additional core level is not yet amortized. Finally, by adding the cache-group level, HMCS⟨4⟩ achieves up to 60% better throughput than HMCS⟨3⟩. This showcases the importance of the cache-group level — although not specified by the OS (*e.g.*, lscpu) nor considered in prior work.

3.2 Potential of Heterogeneous Locks

The best performing lock differs between levels and architectures. To show that, we select cohorts from our heatmaps and run LevelDB on them with a set of NUMA-oblivious locks — see Section 5 for setup details. For x86, the cohorts are core, cache group, NUMA node, and system; for Armv8, we use cache group, NUMA node, package, and system.

Figure 3 shows the performance of LevelDB on these cohorts with different locks. First, note that the best lock differs between levels (**A2**). On both platforms, Ticketlock is the best lock (even if with only a small margin) at the system level. At this level, only two threads run, one in each NUMA node in x86 or in each package in Armv8. In contrast, Ticketlock performs poorly on the NUMA level. At this level, eight threads run, one in each cache group. Moreover, the best lock differs between architectures (**A3**). At the NUMA-node level, the best lock is Hemlock on x86 and CLH lock on Armv8.

Note that the difference between the two versions of Hemlock in the figure: `hem` has the x86-specific CTR optimization (described in Section 2.1) disabled, while `hem-ctr` has it enabled. As expected, CTR can yield better performance on x86; however, it dramatically degrades performance on Armv8



(a) x86 server **(b)** Armv8 server

Figure 3. LevelDB throughput with different NUMA-oblivious locks on different cohorts at maximum contention. The best lock differs between platforms (and underlying architecture) and cohorts.

(the throughput is close to 0). On Armv8, both operations involved in the optimization are implemented with load-exclusive/store-exclusive pairs[2]. Since they access the same variable, the `fetch_add` operation will cause the `cmpxchg` to repeatedly fail, hindering the thread releasing the lock from completing its operation. In the remainder of this paper, `hem` on x86 denotes Hemlock with CTR *enabled*, whereas `hem` on Armv8 denotes Hemlock with CTR *disabled.*

3.3 Correctness of Complex Locks on WMMs

To improve performance, WMM architectures such as Armv8 can optimize the sequential execution by aggressively reordering memory accesses. Guaranteeing correctness of lock algorithms on WMMs requires careful use of memory barriers to disable harmful reorderings, while still allowing as many hardware optimizations as possible. As previously discussed, existing work on locks, such as CNA and ShflLock, tends to ignore WMM issues as they target x86.

In this work, we used VSync [32] to mechanically verify the implementations of ShflLock, CNA, and HMCS and to maximally relax their barriers, while maintaining their correctness on Armv8 (**A4**). However, this approach does not scale to arbitrary hierarchy depths. The deeper the hierarchy is, the more threads/cores are required for verification; and the verification time is generally super-exponential in the number of threads. For example, the verification of HMCS⟨3⟩ requires 4 threads and takes 10 seconds; the verification of HMCS⟨4⟩ requires 5 threads and lasts more than a day.

[2]Armv8.1 introduced LSE instructions [23] that can mitigate the negative impact of CTR in Kunpeng 920 processors. However, the overall performance with LSE is sometimes worse than with load-exclusive/store-exclusive. Similar conclusions were found in experiments with MySQL [4].

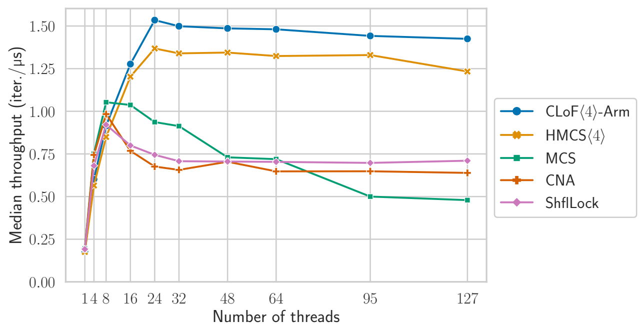


Figure 4. LevelDB with increasing contention, comparing different state-of-art locks and CLoF on Armv8.

3.4 In Search for a Composable Approach

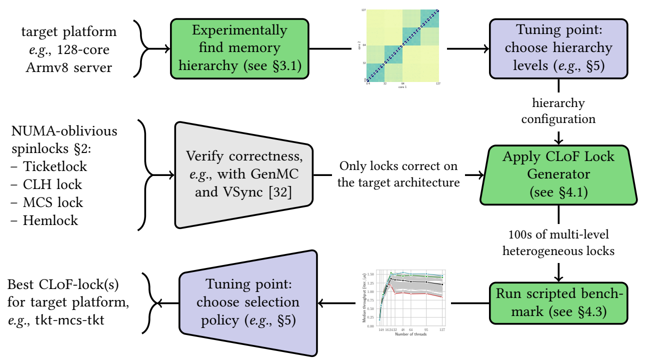
Given that modern NUMA architectures have deep hierarchies (Section 3.1) and the best lock differs among architectures and levels (Section 3.2), we propose CLoF in Section 4, a framework for building multi-level locks with level-heterogeneity. CLoF generates NUMA-aware locks that are correct by construction and tailored to the target platform (and underlying architecture).

We now demonstrate the potential of CLoF. In Figure 4, we compare the throughput of LevelDB with the best CLoF-lock on Armv8, several state-of-the-art NUMA-aware locks, and MCS lock. For fewer than 32 threads, CNA lock and ShflLock suffer from a shuffling overhead, degrading their performance with respect to MCS lock. Once the NUMA level is crossed (> 32 threads), CNA lock and ShflLock match and later (> 64 threads) improve over MCS lock because they support NUMA-node level. Note that CLoF and HMCS do not introduce that overhead. The support of the full hierarchy allows HMCS⟨4⟩ to greatly outperform these locks. Introducing heterogeneity gives CLoF additional 10% to 15% higher performance from 8 to 128 threads. Similarly, the heterogeneity aspect is also beneficial for x86. In Figure 2, CLoF⟨4⟩ outperforms HMCS⟨4⟩ for most contention levels, *e.g.*, by 5% with 8 threads and 33% with 96 threads.

As we are going to see in Section 5, the best CLoF-lock on x86 is not composed of the same NUMA-oblivious locks as the best CLoF-lock on Armv8.

4 The Compositional Lock Framework

Figure 5 describes the CLoF user's perspective. First, CLoF requires a *hierarchy configuration*, *i.e.*, a file describing the memory hierarchy levels of target platform. As in Section 3.1, CLoF produces a heatmap of the target platform, from which the user can identify these levels by grouping tiles colored with similar intensity. The resulting configuration can be tuned to select only those levels most relevant to the user (see example in Section 5.2). Note that, although not currently implemented, identifying levels in a heatmap can be easily automated. Next, the user selects a set of NUMA-oblivious



target platform
e.g., 128-core
Armv8 server
Experimentally find memory hierarchy (see §3.1)
Tuning point: choose hierarchy levels (e.g., §5)
hierarchy configuration
NUMA-oblivious spinlocks §2:
– Ticketlock
– CLH lock
– MCS lock
– Hemlock
Verify correctness, e.g., with GenMC and VSync [32]
Only locks correct on the target architecture
Apply CLoF Lock Generator (see §4.1)
100s of multi-level heterogeneous locks
Best CLoF-lock(s) for target platform, e.g., tkt-mcs-tkt
Tuning point: choose selection policy (e.g., §5)
Run scripted benchmark (see §4.3)

Figure 5. Workflow of CLoF: input is a target platform and a set of NUMA-oblivious locks; output is a correct best performing multi-level heterogeneous NUMA-aware lock. Green boxes are the contributions of this work; blue boxes are optional tuning points for the user; and gray boxes are outside the scope of this work.

spinlocks, *e.g.*, those described in Section 2. We call these the *basic locks*. We verify and optimize these locks using the VSync framework [32].

The core of CLoF is the lock generator described in Section 4.1. Based on the set of correct basic locks and the hierarchy configuration, the CLoF lock generator outputs hundreds of multi-level heterogeneous NUMA-aware locks. In Section 4.2, we use an induction argument with model checking to show that CLoF locks are correct by construction.

Finally, the scripted benchmark described in Section 4.3 selects the best CLoF lock for the target platform. The user can change the default *selection policy* to prioritize low contention performance over high contention (we evaluate both policies in Section 5).

The CLoF workflow (Figure 5) is fully automated with the exception of the tuning points: (1) creating the hierarchy configuration, and (2) choosing the selection policy.

4.1 The Lock Generator

We now present our lock generator in two steps: first, we address the issue of supporting multiple levels of heterogeneous locks; second, we show how to abstract the spinlocks to isolate the framework from the lock implementations.

4.1.1 Syntactic Recursion. CLoF employs syntactic recursion to support different locks on each level of the hierarchy. Since syntactic recursion unfolds at compile-time (*e.g.*, via C++ templates or C macros), it does not have the overhead of virtual function pointers. We describe the recursion in the lock generator with a simple domain specific language shown in Figure 6.

Lock sets. Let l be a lock in *BasicLocks*, *i.e.*, the set of all NUMA-oblivious locks. Let L be a lock in *ClofLocks*, the set of all CLoF-generated locks. By construction, the composed

(*NoCtxLockType*)	τ_n	::=	ttas | ticket | ...
(*CtxLockType*)	τ_x	::=	mcs | clh | hem | ...
(*LockType*)	τ	::=	$\tau_n \mid \tau_x$
(*LockId*)	o	$\in$	Nat
(*CtxId*)	c	$\in$	Nat
(*MetaData*)	d	::=	...
(*BasicLocks*)	l	::=	(τ, o, d)
(*ClofLocks*)	L	::=	$l \mid$ CLoF(l, L)
(*ClientCode*)	C	::=	acq(L, c) | rel(L, c) | ...

Figure 6. The grammar of the lock generator language defining the recursive structure of *ClofLocks*.

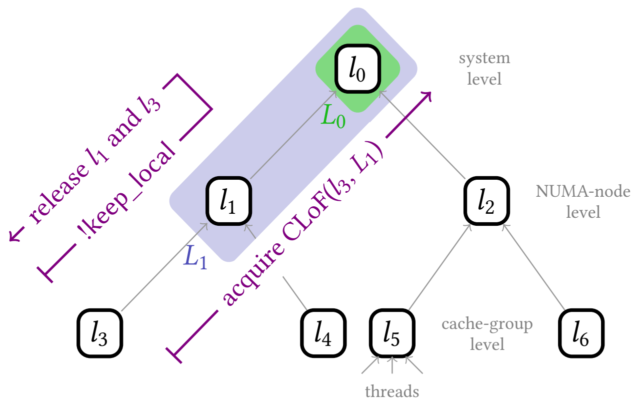


Figure 7. CLoF unfolded hierarchy of locks with three levels (system, NUMA-node, and cache-group). Threads acquire locks from the lowest cohorts up to the root. keep_local controls up to which level locks may be released. In the example, keep_local returns false twice, so l_3 and l_1 are released, but not l_0, which will be passed to the next owner of l_1.

lock CLoF(l, L) is also part of *ClofLocks*. We call l the *low lock* and L the *high lock* of CLoF(l, L). The low lock l belongs to a hierarchy level immediately lower than the high lock L. Consider the example in Figure 7 with an unfolded hierarchy of three levels: cache-group, NUMA-node, and system. Low lock l_3 protects a cohort of the cache-group level, whereas high lock L_1 protects a NUMA-node and the system cohort.

A low lock l consists of the lock type τ, the unique identifier o, and the metadata d used to link with the high lock and to pass locks among different levels. Some locks require a context c to operate. Contexts are represented as unique identifiers *CtxId*. We discuss contexts in detail in Section 4.1.3.

Typically, a single lock is used to protect a critical section *CS*. In CLoF, each thread may acquire a different CLoF-lock when accessing *CS* as long as these locks (1) have the same sequence of levels; and (2) have the same system-level lock l, *e.g.*, l_0 in Figure 7. Requirement (1) guarantees a locking hierarchy to avoid circular waits and deadlocks [18]. Requirement (2) guarantees mutual exclusion of *CS*. In the example of Figure 7, a thread that belongs to the left-most cache-group cohort uses CLoF(l_3, CLoF(l_1, l_0)) to access *CS*, whereas another thread that belongs to the right-most cohort uses CLoF(l_6, CLoF(l_2, l_0)). Note that, for convenience, our lock generator implementation groups all CLoF-locks rooted at the same system lock l_0 into a single lock interface, abstracting from the user which low-level lock the thread has to acquire first.

Base cases. The client code C provided by the user is processed to unfold the recursive definition CLoF(l, L). In particular, lock-acquire and lock-release calls, *i.e.*, acq(L, c) and rel(L, c), are unfolded with **lockgen** (see Figure 8); other statements are kept unmodified. The base cases of **lockgen** directly unfold to acquire and release calls of l, *e.g.*, mcs_acq and mcs_rel when τ = mcs.

Inductive cases. These cases unfold recursively, adding code to control the multiple levels while acquiring or releasing a lock. In the unfolded **lockgen**(acq(CLoF(l, L), c)), a thread first acquires l before acquiring L, *e.g.*, l_3 before L_1 in Figure 7. When acquiring L, the thread climbs up the level hierarchy (see acquire direction in Figure 7). Alternatively, the thread may skip acquiring L if the lock is passed within the cohort of l. In Section 4.1.2, we will discuss the lock-passing mechanism in detail; essentially, we say that a thread *passes* L, if the thread does not release L but gives L to the next thread acquiring l, effectively keeping L local to the cohort protected by l. We can see the lock-passing mechanism in the unfolded **lockgen**(rel(CLoF(l, L), c)). A thread either releases l and passes L to its successor, or releases both, again climbing up the hierarchy. For example, in Figure 7, a thread may release l_3 and l_1 but pass L_0.

Note that we do not specify how the links between locks of different levels are maintained. Each CLoF(l, L) extends the low lock l with metadata. Our lock generator implementation keeps a pointer to the high lock L in l's metadata.

4.1.2 Lock-Passing Mechanism.

When a thread T releases CLoF(l, L), two conditions control the decision of whether to pass L to another thread in the same cohort or release L to a thread from another cohort.

First, T can only pass L if there is some thread T' in the same cohort waiting to acquire l. Otherwise, it releases to another cohort. CLoF employs a strategy similar to the *read indicator* by Calciu *et al.* [5]. Before acquiring l, each thread T' calls inc_waiters, which atomically increments a counter in l's metadata. T' also calls dec_waiters immediately after acquiring l. This way, thread T can detect waiting threads in the current cohort by calling has_waiters. Note that in some lock algorithms, the lock owner T can easily detect whether another thread is waiting to acquire a lock without having to rely on such additional counter variable. For example, in MCS lock it suffices to check whether the next pointer is set, and

$$\mathbf{lockgen}(\mathrm{acq}(l, c)) = \begin{cases} \tau_n\_\mathrm{acq}(o) & \textbf{if } l = (\tau_n, o, d) \\ \tau_x\_\mathrm{acq}(o, c) & \textbf{if } l = (\tau_x, o, d) \end{cases}$$

$$\mathbf{lockgen}(\mathrm{rel}(l, c)) = \begin{cases} \tau_n\_\mathrm{rel}(o) & \textbf{if } l = (\tau_n, o, d) \\ \tau_x\_\mathrm{rel}(o, c) & \textbf{if } l = (\tau_x, o, d) \end{cases}$$

```
lockgen(acq(CLoF(l, L), c)) =
        inc_waiters(l.d);
        lockgen(acq(l, c));
        dec_waiters(l.d);
        if (¬has_high_lock(l.d)) {
                lockgen(acq(L, high_ctx(l.d)));
        }

lockgen(rel(CLoF(l, L), c)) =
        if (has_waiters(l.d) && keep_local(l.d)) {
                pass_high_lock(l.d);
                lockgen(rel(l, c));
        } else {
                clear_high_lock(l.d);
                lockgen(rel(L, high_ctx(l.d)));  ①
                lockgen(rel(l, c));  ②
        }
```

Figure 8. The lock generator of CLoF. Defining **lockgen** allows unfolding the code at compile-time for *ClofLocks*.

in Ticketlock to check if the difference between grant and ticket is larger than 1. CLoF supports custom has_waiters as an optional parameter, which also removes inc_waiters and dec_waiters from the generated lock.

Second, T has to eventually let L be acquired by threads in other cohorts even if a thread T' in the same cohort is waiting, otherwise the other cohorts would starve. To avoid that situation, the keep_local function periodically forces T to let L go to a different cohort. When either has_waiters or keep_local returns false, T clears a flag in l's metadata with clear_high_lock and releases both L and l. When has_waiters and keep_local both return true, T calls pass_high_lock, setting a flag in l's metadata to indicate to the next thread that L is already acquired. Now, when another thread acquires CLoF(l, L), it can check whether the flag is set by calling has_high_lock, and decide whether to acquire L accordingly.

Note that our keep_local implementation follows a similar strategy as HMCS: it increments a counter and returns false if a threshold H is reached, resetting the counter. By default, CLoF uses $H = 128$ for each level. Excessively high H values might affect short-term fairness and should be avoided (see the original work for a thorough analysis [6]). Although different strategies are possible, they are outside the scope of this paper.

4.1.3 Context Abstraction.

To fully isolate the lock generator from the implementations of basic locks, we must treat the basic locks as black boxes, abstracting their inner workings. That boils down to abstracting the *context* of diverse lock implementations to a common denominator.

There are two common patterns of the acquire/release interfaces that we consider in this work. Locks without a context (*NoCtxLockType* in Figure 8) usually spin globally, *e.g.*, Ticketlock. Their acquire/release interfaces only take one argument: the lock address. Context-based locks (*CtxLockType* in Figure 8) are common in local spinning algorithms such as MCS and CLH locks. In addition to the lock address, their acquire/release interfaces[3] also expect the address of a context. The *context* is an object typically used by the thread to enqueue itself in the lock-queue and, therefore, is exclusive to each thread. Among other things, the context contains a memory location on which a thread spins. To support both lock types, the lock generator initially assumes all locks require a context and eventually removes the context in the recursive unfolding of **lockgen** if the lock requires no context. From this point onwards, we focus on the context-based locks only.

Context invariant. Although some locks may allow for a thread to use the same context in multiple concurrent lock acquisitions (*e.g.*, Hemlock), most context-based locks do not allow that (*e.g.*, CLH and MCS lock). To illustrate why this is important, consider the MCS lock: when acquiring an MCS lock l_1, a thread appends its context c to a queue, whose tail is kept in l_1. The context c is now visible to other threads trying to acquire l_1; c's next pointer may be modified to point to the successor thread. If c is reused to acquire another lock l_2 while it is linked in l_1, then c may be modified again, corrupting l_1's queue. In such cases, threads acquiring l_1 may hang. Similar issues can happen in other context-based locks, causing deadlocks or mutual exclusion violations.

CLoF aims at generality and support of any context-based lock. Therefore, CLoF has to guarantee a *context invariant*: no context is ever used to concurrently acquire/release multiple locks. With traditional locks such as MCS lock, the user typically enforces this invariant by allocating a context per thread, or keeping the context in the stack. In CLoF(l, L), to acquire the lowest lock l, threads also use the typical per-thread context. To acquire the high lock L, however, threads need a different context — call it c. The context c is necessary because L is either a basic lock l' or a composed lock CLoF(l', L'); and l' requires c to be acquired or released. To transparently provide contexts to the high locks, our lock generator extends l's metadata with such a context c. In order to guarantee the context invariant, CLoF enforces that only the owner of l can use c to acquire or release L.

[3]Locks with more than two parameters can have their interface rewritten to group all shared variables together and all local ones into two parameters.

The release order of CLoF(l, L) is crucial to guarantee the context invariant (see ① and ② in Figure 8). A deadlock could easily happen if one would inadvertently invert the order in which locks are released, *i.e.*, first releasing l and then L. If a thread T_1 releases l before releasing L, then a thread T_2 can acquire l between both releases and immediately try to acquire L. In such case, T_2 retrieves c from l's metadata (with high_ctx) and tries to acquire L, while T_1 uses the same c to release L. This concurrent use of the context violates the invariant causing the issues mentioned above.

Thread-obliviousness. Remember when a thread passes L, it does not release it. Consequently, L is sometimes acquired by one thread and released by another. Hence, the lock implementations used in high locks have to be *thread-oblivious* — lock cohorting has a similar requirement [15]. Most locks are thread-oblivious provided that the thread releases lock L using the same context c that it previously used to acquire L. Hemlock is an exception: it defines a thread-local context and implicitly accesses this context whenever necessary — which is why the authors claim that the lock is context-free, despite of implicitly using a context. By defining the context explicitly and passing it to the normal acquire/release interface, Hemlock becomes thread-oblivious.

4.2 CLoF Correctness

We now argue that locks generated with CLoF are correct. For that we sketch an induction proof and mechanically check both base and induction steps with model checkers.

4.2.1 Properties. A lock is correct if it satisfies *mutual exclusion* and *progress*. Two progress properties concern us: fairness and spinloop termination. *Fairness* guarantees that any thread trying to acquire the lock will eventually succeed — this property is also called starvation freedom. Not all locks are fair, *e.g.*, test-and-test-and-set (TTAS) [19]. Unfair locks should at least satisfy *spinloop termination.* Spinlock termination guarantees that each thread acquiring or releasing the lock eventually succeeds provided the client code is finite. Note that fair locks also satisfy spinloop termination.

Theorem 4.1 (CLoF correctness)**.** *For all L in ClofLocks, L is correct,* i.e.*, L satisfies mutual exclusion and spinloop termination. Moreover, L satisfies fairness if L is a fair basic lock or composed from fair basic locks.*

4.2.2 Proof Sketch. We start with the induction step.

Induction step: L = CLoF(l, L') is correct, where l is in *BasicLocks* and L' in *ClofLocks.* The induction hypothesis is that L' is correct. For now, we assume the basic lock l to be correct and revisit this assumption in the base step. We show the induction step is correct with two model checkers.

First, we model CLoF(l, L') in TLA$^+$ and utilize the accompanying TLC model checker [28] to show that *mutual exclusion* and *fairness* properties hold as well as the context invariant (Section 4.1.3). Since l is correct by assumption and L' is correct by the induction hypothesis, we can replace them with abstract fair locks. Their state is modeled by queues (sequences in TLA$^+$) and acquire/release functions are modeled as single steps. The remainder of the model is almost a one-to-one mapping of **lockgen** in Figure 8. Unfortunately, TLA$^+$ can only model sequentially consistent (SC) memory accesses and alone cannot guarantee CLoF(l, L') with relaxed atomics[4] is correct on WMMs.

Second, we show CLoF(l, L') is also correct on WMMs by model checking the induction step again with GenMC [26], while maximally relaxing the memory barriers to improve performance with VSync [32]. As in the TLA$^+$ model, we represent l and L' with an abstract fair lock, *i.e.*, a verified Ticketlock implementation. With that we can show *mutual exclusion* and *spinloop termination* hold on WMMs. Note that although spinloop termination is necessary to guarantee fairness, it does not imply fairness. Currently, there is no model checking approach capable of verifying fairness of non-trivial locks on WMMs. Therefore, we can only gauge fairness with our experiments on real hardware.

Base step: $L = l$ is correct, where l is in *BasicLocks.* As described in Figure 5, we take as input only correct NUMA-oblivious spinlocks. Our implementations were verified on WMMs with GenMC and — as CLoF itself — optimized with VSync to maximally relax their memory barriers.

4.2.3 Discussion. To conclude the correctness argument, we highlight three important aspects of CLoF.

Memory barriers on WMMs. With VSync, we identified that all memory accesses introduced by the highlighted auxiliary functions in **lockgen** (Figure 8) can be relaxed, *i.e.*, require no additional memory barriers, as long as the memory barriers already present in the basic locks are maintained.

Fairness. A CLoF-lock provides fairness only if its basic locks are also fair. This can be easily shown with our TLA$^+$ model: the fairness property is violated if we replace any of the abstract fair locks with an unfair one. For example, consider the hierarchy in Figure 7, if l_0 is a TTAS lock, *i.e.*, an unfair lock, then one of the NUMA-node cohorts can starve, being unable to successfully acquire l_0 during long periods. For most applications, the use of fair locks is preferable, and we only consider fair locks in the remainder of this paper.

Model checking deep hierarchies. Even though existing model checkers cannot fully verify fairness on WMMs, verifying mutual exclusion and spinloop termination is essential for locks optimized with relaxed memory barriers. A single missing barrier can easily cause the application to crash, hang, or corrupt data. Unfortunately, even an efficient model

[4]On real hardware with WMM, SC atomics are known to be rather expensive because they introduce several strong memory barriers; whenever possible, practitioners tend to employ more relaxed atomics instead.

checker such as GenMC does not scale to verify a complete NUMA-aware lock with support to a hierarchy with more than 3 levels. For example, a 2-level CLoF-lock with simple Ticketlocks takes about a second to be model checked; a 3-level takes almost 3 minutes; and a 4-level times out after 12 hours. This exponential increase of time is due to the greater number of threads — 3, 4, and 5 threads, respectively — that are necessary to verify the lock while exercising all possible memory-access reorderings. If more complex NUMA-oblivious locks are used (*e.g.*, MCS lock), the model checking time may be orders of magnitude higher.

In spite of that, the composability of CLoF allows us to scale the model checking to arbitrary hierarchy depths. Our correctness argument employs the model checker in the induction and base steps separately. The induction step merely requires a 2-level CLoF-lock with Ticketlocks, which in turn requires 3 threads for verification. The NUMA-oblivious locks of the base step tend to have similar requirements. In fact, the 10 NUMA-oblivious spinlocks in [32], including the complex Linux qspinlock, require 3 threads. Hence, CLoF enables the exploration of lock designs in deep NUMA hierarchies with elaborate NUMA-oblivious locks.

4.3 Finding the Best CLoF Lock

Given a hierarchy configuration and a set of basic locks, the lock generator combines basic locks, one per hierarchy level, producing hundreds of NUMA-aware locks.

Modern processors are complex and their behavior hard to predict, especially with concurrent software. Manually deciding which combination of basic lock provides the best performance is an error-prone process that highly depends on the target platform. To find the best lock, we instead propose an automatic approach: we exhaustively generate and evaluate all possible combinations of the basic locks. This results in hundreds of CLoF locks: With a set of N basic locks and M levels, there are N^M combinations. In practice, this number stays relatively low, allowing this exhaustive evaluation approach to be tractable[5].

Scripted benchmark. After generating the locks, we run a quick performance evaluation using a selected benchmark. The selected benchmark must support changing the lock implementation and the number of threads trying to acquire the lock simultaneously. Note that selecting a benchmark matching the target workload may improve the later lock selection for this specific workload. Each lock is evaluated for a few different contention levels, resulting in a set of lines, one for each CLoF lock, mapping contention levels to throughput (as depicted in Figure 9, Section 5). We call this procedure the *scripted benchmark*.

[5]In a scenario with a high number of combinations, one can use pre-selection heuristics (possibly based on the results reported in Figure 3) to reduce the size of the search space before performing the actual lock generation.

Selection policy. Once the benchmark is complete, we need to rank the locks and select the best. The efficiency of CLoF locks differ for different number of threads. We observe that for a fixed number of levels, no CLoF lock combination outperforms all the others in all contention levels. Therefore, we define two *selection policies* (the last step in Figure 5): (1) rank with the weighted average throughput among all contention levels, favoring locks performing well at high contention; and (2) rank with the inverse weighted average throughput among all contention levels, favoring locks performing well at low contention. Taking the first for (1) gives us the CLoF HC-best lock (for High Contention), while taking the first for (2) gives us the CLoF LC-best (for Low Contention). Additionally, for informative purpose, we also select the last for (1), giving us the CLoF worst lock.

In the next section, our evaluation shows that in general, both HC-best and LC-best locks outperforms state-of-the-art locks such as HMCS. In some cases, the HC-best lock trades a moderate performance loss at low contention for a strong performance gain at high contention. On the other hand, LC-best has moderate performance gains across all contention scenarios. The scripted benchmark outputs both HC-best and LC-best locks together with their respective performance, and it is up to the CLoF user to manually choose between them according to its use case. The selection policy can be further customized by the user if necessary.

5 Evaluation

In this section, we evaluate CLoF on two different architectures to show the following: the CLoF workflow (Figure 5) is practical and is able to find best CLoF locks for different platforms; the selection policies are effective in optimizing CLoF for different contention scenarios; level tuning allows improving performance; in each platform, the best CLoF lock is composed of different basic locks; and the best CLoF locks outperform the state-of-the-art NUMA-aware locks for most contention scenarios.

5.1 Experimental Setup

5.1.1 Hardware and Software.

We evaluate CLoF on the following platforms:

x86: a GIGABYTE **R182-Z91-00** server [16] equipped with 2 **EPYC 7352** processors, each has **24** cores [2], totaling 48 cores (96 hyperthreads).

Armv8: a Huawei **TaiShan 200** (Model 2280) server [21] equipped with 2 **Kunpeng 920–6426** processors, each has **64** cores [20], totaling 128 cores (no hyperthread).

On these servers, we installed Ubuntu 18.04.5 LTS, with the `5.4.0` Linux kernel. We reproduced similar results on OpenEuler 20.03 LTS. To ensure stable results, we (1) pin threads to CPUs, (2) set the operating CPU frequency to a constant value, (3) disable IRQ balancing, (4) disable NUMA balancing, and (5) isolate the CPUs running the benchmarks (only a single CPU, responsible to run all the other tasks of the system, is not marked as "isolated" in the kernel boot arguments; this CPU is then not used by the benchmarks).

5.1.2 Benchmarks. Our evaluation is based on two well-known concurrent benchmarks: LevelDB and Kyoto Cabinet. LevelDB [9] is a storage library written by Google that supports concurrent accesses. We use its "readrandom" benchmark, which is often used in the literature to evaluate the performance of locks [11, 13, 24]. Kyoto Cabinet [27] is a library of parallel database routines also used in several prior works [11, 14, 32]. Importantly, we use LevelDB within the scripted benchmark (Section 4.3) to find the best CLoF locks, and we use Kyoto Cabinet as a control benchmark to cross-validate the performance of the best CLoF locks.

Both benchmarks can be configured for a fixed *duration* (in seconds), *number of threads*, and *number of runs* (#runs). The benchmarks return the throughput (executed operations per second) of each run.

We interpose benchmark calls to `pthread` functions with `LD_PRELOAD` in order to replace the lock implementation without having to recompile the benchmarks — as similarly done in other works [17].

5.2 Instantiating the CLoF Workflow

We now apply the approach described in Section 4.3, running the scripted benchmark (using LevelDB) to find the best lock among all possible combinations of NUMA-oblivious locks. The scripted benchmark evaluates all the generated locks (#runs = 1 and duration = 1s). As baseline, we use HMCS configured with the same hierarchy as CLoF.

5.2.1 Tuning Points. We explore both tuning points outlined by Figure 5, varying the number of levels in the hierarchy configuration and trying both selection policies (HC/LC). We group the results by number of levels.

Selecting the hierarchy levels is a trade-off between overhead and locality: skipping a level reduces the number of basic locks to acquire/release at the cost of missing locality opportunities of this level. The user may tune the discovered hierarchy accordingly (see Figure 5). We use the following notation for identifying a specific CLoF lock. Considering the core, cache-group, NUMA-node, package and system levels — in this order — we denote a CLoF lock with *n* levels by a sequence of *n* abbreviations, where `tkt`, `mcs`, `clh` and `hem` respectively denote Ticketlock, MCS lock, CLH lock and Hemlock. The sequence only considers the *n* levels in the lock's hierarchy configuration. For example, `hem-hem-mcs-clh` refers, in the x86 4-level hierarchy below, to the 4-level CLoF lock composed of Hemlocks at the core and cache-group levels, MCS at the NUMA-node level and CLH at the system level; there is no package level in that configuration.

4-level locks. We start by finding the best CLoF locks with the following 4-level hierarchies:

- x86: core, cache, numa, system;
- Armv8: cache, numa, package, system.

On x86, we skip the package level, as its processor model has only one NUMA node per package. On Armv8, we skip the core level as there is no hyperthreading, *i.e.*, there is 1 CPU per core. Considering all $N = 4$ basic locks (*i.e.*, Ticketlock, CLH, MCS and Hemlock) and $M = 4$ levels, the number of combinations is $N^M = 4^4 = 256$ on both platforms.

Figures 9a and 9b respectively report the performance of these 256 locks for x86 and Armv8. Due to the large number of locks, we highlight HMCS (as baseline), the HC-best, the LC-best and the worst (all found with the selection policy described in Section 4.3). The remaining locks are grouped in the Others category, which forms a *beam* of gray lines.

In relation to HMCS on x86, HC-best trades a moderate performance loss at low contention (*e.g.*, -5% with 8 threads) for a strong performance gain at high contention (*e.g.*, 47% with 95 threads). On the other hand, LC-best has moderate performance gains across all contention scenarios, ranging on Armv8 from 3% (1 and 4 threads) to 15% (127 threads).

Finally, note that the best CLoF locks on each architecture are composed of different basic locks. For example, on x86, the HC-best is composed of Hemlock, CLH and MCS, whereas on Armv8, it is composed of Ticketlock and CLH.

3-level locks. Next, we find the best CLoF locks for each platform considering the following 3-level hierarchies:

- x86: cache, package, system;
- Armv8: cache, numa, system.

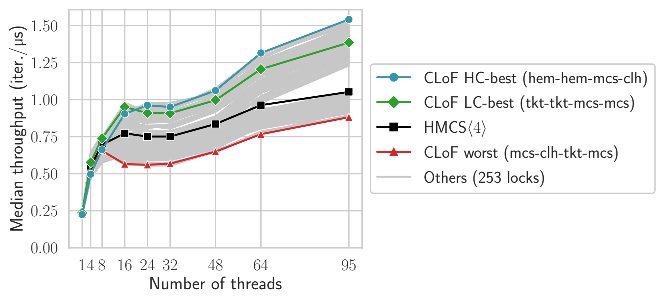
We skip the package level on Armv8 since the difference in performance between system and package is thin (see Figure 1b and Table 2). On x86, we skip the core level as many applications disable the usage of hyperthreads altogether.

We repeat the same workflow as in the 4-level case. With $M = 3$ levels the number of combinations is $N^M = 4^3 = 64$. Figures 9c and 9d show the results obtained on each platform. Overall the performance gains observed are similar to the 4-level case. Nevertheless, one interesting result is that, on Armv8, both selection policies find the same best lock.

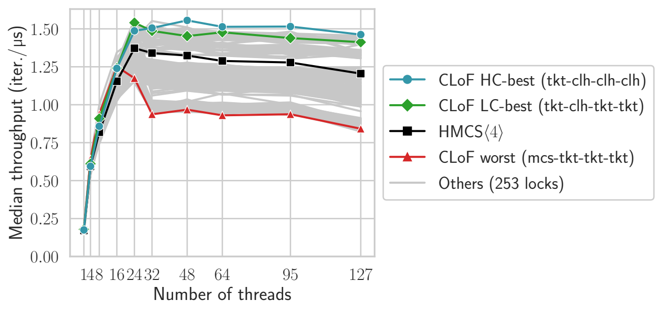
The basic lock composing the best CLoF locks are different from the 4-level case on both platforms. For example, on x86, the HC-best lock is now composed of Hemlock, Ticketlock and MCS, whereas on Armv8, the system level is Ticketlock instead of CLH.

5.2.2 Composition Analysis of CLoF Locks. Each best CLoF lock found in Figure 9 has its own distinct composition. Every component has an impact on the overall performance of the CLoF lock. We now discuss two cases, using the performance reported in Figure 9.

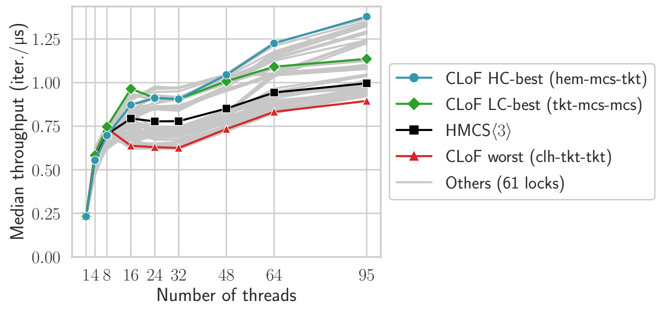
Ticketlock in the NUMA level on Armv8. We observe that the worst CLoF lock uses the Ticketlock at the NUMA level with both 3-level and 4-level. Recall that, when run in isolation in the NUMA cohort, Ticketlock yields almost half



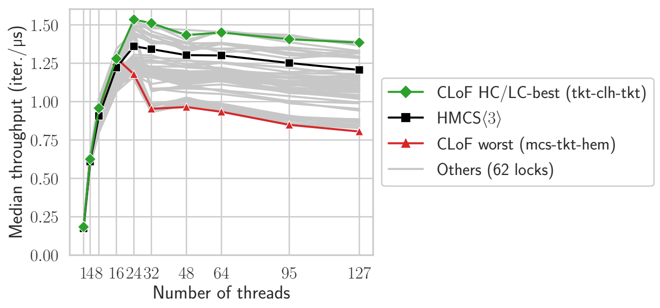
(a) x86, 4-level hierarchy: `core-cache-numa-system`.



(b) Armv8, 4-level hierarchy: `cache-numa-package-system`.



(c) x86, 3-level hierarchy: `cache-numa-system`.



(d) Armv8, 3-level hierarchy: `cache-numa-system`.

Figure 9. LevelDB evaluation on x86 and Armv8 comparing all CLoF-generated locks with 3 and 4 levels against HMCS implementations with equivalent hierarchy. CLoF locks using hem use the CTR optimization only on x86.

the performance of other locks (see Figure 3). In fact, if we replace the NUMA level of any CLoF lock with Ticketlock, the performance dramatically drops at 32 threads — similarly to the worst lock.

Core level in x86. The 4-level results (Figure 9a) show that the beam of gray lines splits at 64 threads into two groups, one above HMCS and another below. The reason is related to core-level lock: CLoF locks with Hemlock or Ticketlock have higher performance than HMCS, whereas those with MCS or CLH have lower performance. Figure 3 shows that, in isolation, Hemlock and Ticketlock yield a slightly better performance than MCS and CLH. Between Hemlock and Ticketlock, however, the difference is less clear.

Other cases. The reason why a lock is part of a best CLoF lock is not always clear. Because of the interplay between different levels and locks used, the results obtained with basic locks in isolation (Figure 3) can be diminished or exacerbated when the locks are a part of a CLoF lock.

5.2.3 Fairness. We evaluated the fairness of all the locks with a slightly modified LevelDB benchmark to report per-thread results. As expected, the fairness of all CLoF-locks closely matches the fairness obtained with HMCS in all cases since CLoF employs the same keep_local strategy as HMCS. For the sake of space, we do not detail these results further.

5.3 The Best CLoF Locks in Action

In this section, we pick the best CLoF locks (for 3/4 levels, x86/Armv8) and evaluate them thoroughly (#runs = 3 and duration = 10s) on both platforms; this time with both LevelDB and Kyoto Cabinet benchmarks. We only consider locks selected with the LC policy, as they yield more balanced gains than HC ones across all contention levels. We use the notation CLoF⟨n⟩-x86 and CLoF⟨n⟩-Arm to refer to the LC-best CLoF lock with n levels on x86 and Armv8 respectively — CLoF⟨n⟩ refers to the CLoF lock of both platforms. We start by comparing the best CLoF locks against each other. Then, we compare them against state-of-the-art NUMA-aware locks. Results are showed together in Figure 10. Since the results have only a small variance, we report the median throughput.

5.3.1 Comparing Best CLoF Locks.

3-level against 4-level — x86. CLoF⟨3⟩-x86 matches the performance of CLoF⟨4⟩-x86 when only one hyperthread per core is used (*i.e.*, with at most 48 threads). With more than 48 threads, CLoF⟨4⟩-x86 experiences a great performance boost of about 23% on both LevelDB and Kyoto Cabinet.

3 levels against 4 levels — Armv8. The Kyoto Cabinet benchmark shows that CLoF⟨3⟩-Arm is better than CLoF⟨4⟩-Arm from low to mid contention by around 8%. At high contention, CLoF⟨4⟩-Arm matches CLoF⟨3⟩-Arm performance.

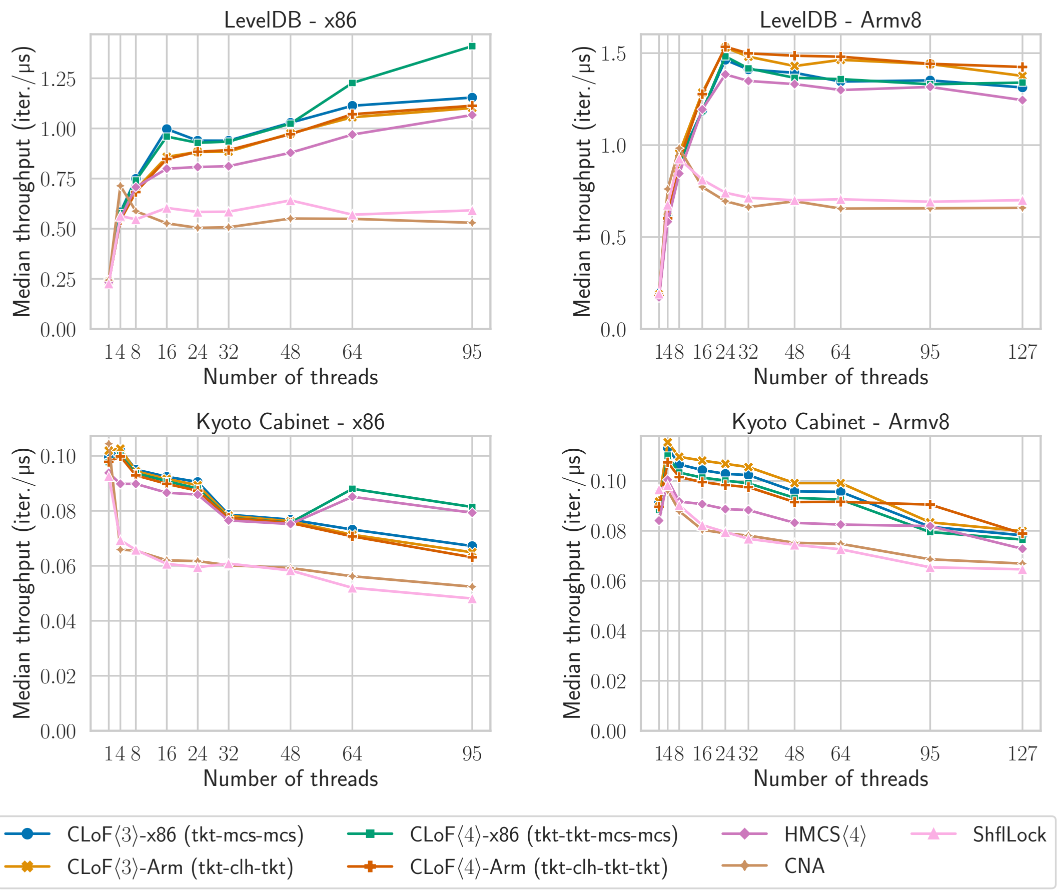


Figure 10. Evaluation on LevelDB and Kyoto Cabinet on x86 and Armv8. LC-best CLoF locks of each platform and hierarchy with 3 and 4 levels are compared to state-of-the-art locks, *i.e.*, HMCS⟨4⟩ (with equivalent hierarchy), CNA and ShflLock.

We conclude that the fourth level does not always improve the overall performance, possibly due to the rather small latency difference between intra-package NUMA nodes and extra-packages NUMA nodes (Figure 1b, Table 2).

Best lock cross platform. Figure 10 shows how locks picked as best for one platform perform in another. When a CLoF lock for Armv8 is used in x86 (or vice versa), it typically deteriorates performance, yielding results closer to HMCS. We conclude that every platform needs a tailored lock to exploit all its performance potential.

5.3.2 Comparing Against the State of the Art.

HMCS. In most scenarios showed by Figure 10, CLoF⟨4⟩ clearly outperforms HMCS⟨4⟩; note they are configured with the same hierarchy levels. For example, on x86 with LevelDB, CLoF⟨4⟩-x86 outperforms HMCS⟨4⟩ with 8, 32 and 95 threads by 5%, 15% and 33% respectively.

Likewise, on Armv8 with Kyoto Cabinet, CLoF⟨4⟩-Arm outperforms HMCS⟨4⟩ with 8, 32 and 127 threads by 11%, 10% and 8% respectively. In the highly contended Kyoto Cabinet on x86, CLoF⟨4⟩-x86 and HMCS⟨4⟩ display a performance increase with more than 48 threads. That is caused by the activation of the core level lock, which is only present in HMCS⟨4⟩ and CLoF⟨4⟩-x86.

CNA and ShflLock. In general, ShflLock performs comparably to CNA. Both locks do not scale with increasing contention and are greatly outperformed by our CLoF⟨4⟩ locks: up to 139% on x86 and 109% on Armv8. This is because CNA and ShflLock are not aware of levels other than NUMA level (and system level). Whether multi-level extensions of these locks are possible is an open question.

6 Related Work

In Section 2, we already covered a few basic and NUMA-aware spinlocks. Herlihy and Shavit's textbook [19] is a good reference for further basic spinlocks. In this section, we focus on the differences between CLoF and previous NUMA-aware locks and discuss possible extensions.

NUMA-aware locks. NUMA-aware locks try to keep lock handovers local to the NUMA node of the lock owner, making better use of the cache hierarchy [5–7, 10–12, 15, 24, 25, 30, 35]. Recently, Dice and Kogan [11] presented the CNA lock, which is a modification of MCS lock that supports 2-level NUMA-architectures. A key aspect of CNA is its small memory footprint: instead of keeping a hierarchy of locks, the lock owner scans the MCS queue and reorders waiting threads such that threads in the same NUMA node are favored when passing the lock. Since January 2019, there is an ongoing effort to replace MCS lock with CNA in Linux qspinlock[6]. CNA neither supports multi-level NUMAs nor exploits level-heterogeneity, thus missing out on large performance gains in modern architectures.

Kashyap *et al.* [24] propose a family of spinlocks and mutexes based on a strategy similar to CNA: the MCS queue is shuffled according to a policy, which among other things, can prioritize NUMA-node proximity. Unfortunately, their shuffling policy does not account for deep NUMA hierarchies, nor they provide any guidelines how to implement such a multi-level policy.

The works most related to CLoF are the HMCS lock by Chabbi *et al.* [6] and the lock cohorting technique introduced by Dice *et al.* [15] — see Section 2. The HMCS lock achieves building the MCS lock hierarchy with a semantic recursion, whereas CLoF employs a syntactic recursion. The cohorting technique cannot be easily extended to support multi-level NUMA systems and requires the implementations of basic locks to be modified/adapted.

Correctness on weak memory models. With the exception of the HMCS lock, previous work does not consider the correctness on WMMs, leaving the task of placing barriers as an exercise to the readers. Placing barriers can influence the performance of applications [29] and, more importantly, can break the correctness [32]. For example, Oberhauser *et al.* [33] have recently shown that the barriers presented in the HMCS paper are insufficient for Arm.

In this work, we propose using a modular way to show correctness: simpler spinlocks are easier to be verified on WMMs and model checkers can be used to correctly place barriers on them [32]. By composing simpler but correct locks with CLoF, the resulting NUMA-aware locks are correct by construction.

Fast-path extensions. Since often only a single thread tries to acquire a spinlock, slow path optimizations should minimally affect the critical path for a single thread. Dice and Kogan [12], for example, study low contention scenarios on x86 and show how the addition of a simple test-and-set lock (TAS) as fast path can increase performance compared to previous NUMA-aware locks. Kashyap *et al.* [24] also provide a similar fast path for their ShflLock. AHMCS [7] proposes a more elaborate fast path that accounts for low and mid contention scenarios by allowing threads to bypass parts of the tree and directly acquire locks from higher levels. Fast-path extensions are a topic orthogonal to our work. Extending CLoF with the same TAS approach as ShflLock is rather simple. The investigation of bypassing strategies for CLoF is left as our future work.

[6]https://lwn.net/Articles/852138/

7 Conclusion

Efficient NUMA-aware locks must be tailored to target multi-level NUMA systems; they must fully leverage the deep NUMA hierarchy as well as platform-specific optimizations. Designing complex multi-level heterogeneous NUMA-aware locks and guaranteeing they are correct on WMMs is very challenging. CLoF tackles this challenge in a divide-and-conquer fashion. With detailed hierarchy information, CLoF composes NUMA-oblivious spinlocks — which are simpler to be verified on WMMs and can more easily exploit platform-specific optimizations — into many correct by construction NUMA-aware locks. With a user-provided policy, CLoF then selects the best performing lock for the target NUMA system.

The non-uniform access latencies observed in large NUMA systems can also be observed in modern big.LITTLE architectures, widely employed in handheld devices. Such systems combine slow but power efficient cores with fast but less efficient cores. These two groups of cores form cohorts with different communication trade-offs. Besides the already mentioned future work, we plan to to investigate the applicability of CLoF in such systems.

Acknowledgments

We sincerely thank our shepherd Christopher Rossbach and our anonymous reviewers for their insightful comments and suggestions. We thank Dave Dice for making us aware of the Hemlock algorithm. We also thank Lilith Oberhauser for implementing and verifying the HMCS lock used in this work. Finally, we thank Viktor Vafeiadis and our colleagues of Huawei DRC and OS Kernel lab for reviewing this work.

References

[1] A. Agarwal and M. Cherian. 1989. Adaptive Backoff Synchronization Techniques. *SIGARCH Comput. Archit. News* 17, 3 (April 1989), 396–406. https://doi.org/10.1145/74926.74970

[2] AMD. [n.d.]. 2nd Gen AMD EPYC™ 7352 | Server Processor | AMD. https://www.amd.com/en/products/cpu/amd-epyc-7352. Accessed: 2021-05-07.

[3] AMD. 2021. The 2nd Gen AMD EPYC 7002 Series Processors. https://www.amd.com/en/processors/epyc-7002-series.

[4] Krunal Bauskar. 2020. ARM's LSE (for atomics) and MySQL. https://aws.amazon.com/ec2/graviton.

[5] 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 SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 157–166. https://doi.org/10.1145/2442516.2442532

[6] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. 2015. High Performance Locks for Multi-Level NUMA Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel*

Programming (San Francisco, CA, USA) *(PPoPP 2015)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2688500.2688503

[7] Milind Chabbi and John Mellor-Crummey. 2016. Contention-Conscious, Locality-Preserving Locks. *SIGPLAN Not.* 51, 8, Article 22 (Feb. 2016), 14 pages. https://doi.org/10.1145/3016078.2851166

[8] Jonathan Corbet. 2014. MCS locks and qspinlocks. https://lwn.net/Articles/590243/.

[9] Jeffrey Dean and Sanjay Ghemawat. 2021. LevelDB. https://github.com/google/leveldb.

[10] Dave Dice. 2017. Malthusian Locks. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 314–327. https://doi.org/10.1145/3064176.3064203

[11] Dave Dice and Alex Kogan. 2019. Compact NUMA-Aware Locks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 12, 15 pages. https://doi.org/10.1145/3302424.3303984

[12] Dave Dice and Alex Kogan. 2021. Fissile Locks. In *Networked Systems*, Chryssis Georgiou and Rupak Majumdar (Eds.). Springer International Publishing, Cham, 192–208.

[13] Dave Dice and Alex Kogan. 2021. Hemlock: Compact and Scalable Mutual Exclusion. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '21)*. Association for Computing Machinery, New York, NY, USA, 173–183. https://doi.org/10.1145/3409964.3461805

[14] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. 2014. Adaptive Integration of Hardware and Software Lock Elision Techniques. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures* (Prague, Czech Republic) *(SPAA '14)*. Association for Computing Machinery, New York, NY, USA, 188–197. https://doi.org/10.1145/2612669.2612696

[15] Dave Dice, Virendra J Marathe, and Nir Shavit. 2012. Lock cohorting: a general technique for designing NUMA locks. *ACM SIGPLAN Notices* 47, 8 (2012), 247–256.

[16] GIGABYTE. [n.d.]. R182-Z91 (rev. 100) | Rack Servers - GIGABYTE Global. https://www.gigabyte.com/Rack-Server/R182-Z91-rev-100. Accessed: 2021-05-07.

[17] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. 2016. Multicore Locks: The Case is Not Closed Yet. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Denver, CO, USA) *(USENIX ATC '16)*. USENIX Association, USA, 649–662.

[18] Marc Hamilton. 1999. *Software development: building reliable systems.* Prentice Hall Professional, USA.

[19] Maurice Herlihy and Nir Shavit. 2011. *The art of multiprocessor programming.* Morgan Kaufmann, USA.

[20] HiSilicon. [n.d.]. Kunpeng 920-6426 - HiSilicon - WikiChip. https://en.wikichip.org/wiki/hisilicon/kunpeng/920-6426. Accessed: 2021-05-07.

[21] Huawei. [n.d.]. 2280 Balanced Model - Huawei Enterprise. https://e.huawei.com/uk/products/servers/taishan-server/taishan-2280-v2. Accessed: 2021-05-07.

[22] Huawei. 2019. Huawei Unveils Industry's Highest-Performance ARM-based CPU. https://www.huawei.com/en/news/2019/1/huawei-unveils-highest-performance-arm-based-cpu.

[23] Jeff Defilippi. 2017. Introducing AMBA 5 CHI protocol enhancements: Specification now available. https://community.arm.com/developer/ip-products/system/b/soc-design-blog/posts/introducing-new-amba-5-chi-protocol-enhancements.

[24] 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* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 586–599. https://doi.org/10.1145/3341301.3359629

[25] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Scalable NUMA-aware Blocking Synchronization Primitives. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 603–615. https://www.usenix.org/conference/atc17/technical-sessions/presentation/kashyap

[26] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model Checking for Weakly Consistent Libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 96–110. https://doi.org/10.1145/3314221.3314609

[27] FAL Labs. 2011. Kyoto Cabinet: A straightforward implementation of DBM. http://fallabs.com/kyotocabinet.

[28] Leslie Lamport. 2002. *Specifying systems: the TLA+ language and tools for hardware and software engineers.* Addison-Wesley Longman Publishing Co., Inc., USA.

[29] Nian Liu, Binyu Zang, and Haibo Chen. 2020. No Barrier in the Road: A Comprehensive Study and Optimization of ARM Barriers. 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, 348–361. https://doi.org/10.1145/3332466.3374535

[30] Victor Luchangco, Dan Nussbaum, and Nir Shavit. 2006. A Hierarchical CLH Queue Lock. In *Euro-Par 2006 Parallel Processing*, Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 801–810.

[31] 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. https://doi.org/10.1145/103727.103729

[32] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. 2021. VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 530–545. https://doi.org/10.1145/3445814.3446748

[33] Jonas Oberhauser, Lilith Oberhauser, Antonio Paolillo, Diogo Behrens, Ming Fu, and Viktor Vafeiadis. 2021. Verifying and Optimizing the HMCS Lock for Arm Servers. In *Networked Systems*. Springer International Publishing.

[34] Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. 2015. For a Microkernel, a Big Lock Is Fine. In *Proceedings of the 6th Asia-Pacific Workshop on Systems* (Tokyo, Japan) *(APSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 3, 7 pages. https://doi.org/10.1145/2797022.2797042

[35] Zoran Radovic and Erik Hagersten. 2003. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA '03)*. IEEE Computer Society, USA, 241.