SpecLock: Speculative Lock Forwarding

Pooria M. Yaghini
University of California, Irvine
pooriam@uci.edu

George Michelogiannakis
Lawrence Berkeley National Laboratory
mihelog@lbl.gov

Paul V. Gratz
Texas A&M University
pgratz@tamu.edu

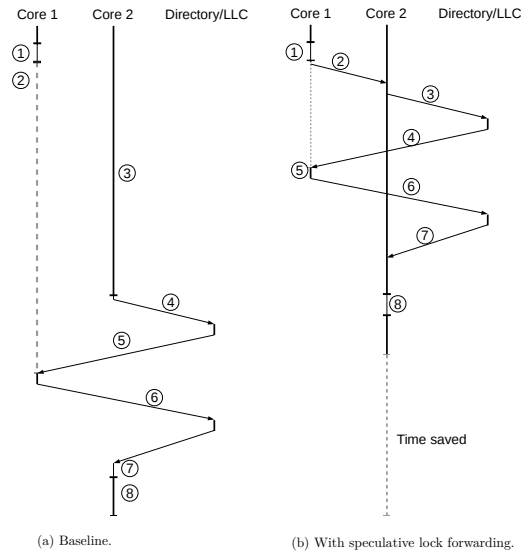
***Abstract*—As core counts increase, lock acquisition and release become even more critical because they lie on the critical path of shared memory applications. In this paper, we show that many applications exhibit regular and repeating lock sharing patterns. Based on this observation, we introduce SpecLock, an efficient hardware mechanism which speculates on the lock acquisition pattern between cores. Upon the release of a lock, the cache line containing the lock is speculatively forwarded to the next consumer of the lock. This forwarding action is performed via a specialized prefetch request and does not require coherence protocol modification. Further, the lock is not speculatively acquired, only the cache line containing the lock variable is placed in the private cache of the predicted consumer. Speculative forwarding serves to hide the remote core's lock acquisition latency. SpecLock is distributed and all predictions are made locally at each core. We show that SpecLock captures 87% of predictable lock patterns correctly and improves performance by an average of 10% with 64 cores. SpecLock incurs a negligible overhead, with a 75% area reduction compared to past work. Compared to two state of the art methods, SpecLock provides a speedup of 8% and 4% respectively.**

I. Introduction

As core counts increase [60], [7], [14], [5], [28], [24], communication distances and contention lead to higher latencies between levels of the memory system. This causes thread-level load imbalance which, by Amdahl's law, directly limit application scalability. This is exacerbated by applications that share large amounts of data between threads [55], [54].

Prior work focuses on hiding latency using multithreading [47], prefetching [6], [16], data value prediction [56], and a plethora of other techniques [35]. While these techniques reduce the impact of private data memory accesses, they do little to address the latency associated with coordinating shared memory accesses between threads, sometimes even slowing these accesses down, exactly the form of latency that becomes more critical as multithreaded applications scale. One such example is the latency associated with lock acquisition and release [55], [54]. Lock acquisition and release, by definition, constitute the serial (critical) path of the application. Parallel performance is fundamentally limited by synchronization through critical sections [15].

A large body of prior work in accelerating lock acquisition focuses on reducing the acquisition latency of highly contended locks [49], [20], [33], [18]. We find that often locks are not in direct contention, but rather much of the latency in critical sections comes from the movement of the cache line containing a lock to the current lock requester driven by the cache coherence protocol. This action can consume hundreds of cycles [55], [54], right at a time when latency is most critical. Unlike this large body of past work, we aim to hide the coherence latency to fetch a lock-containing cache line from a remote core. Thus, our proposed mechanism benefits non-contended and contended locks alike. We show that prior simplistic lock acquisition predictors for non-contended locks [38], [35], [21], [27], [44] may often harm performance because they do not accurately model local lock reacquisition patterns. Further, we find that dynamic lock behavior is highly data dependent and lock acquisition is typically initiated via an OS or system software call. Thus, software prefetching techniques are unsuitable to the problem.



(a) Baseline. (b) With speculative lock forwarding.

Fig. 1: Lock sharing and directory based cache coherence.

To illustrate the problem, Fig. 1(a) shows two threads being synchronized via a lock. Core 1 first enters its critical section by acquiring the lock from its own private cache (①). Some time later, core 2 enters its critical section and issues an LL (load linked) for the cache line containing the lock. A request is made to the directory for an exclusive copy, ④. At ⑤, this request leads to an invalidation request to core 1. At ⑥, core 1's private cache issues a write-back to the LLC of the line containing the lock. Finally, at ⑦, this line is then forwarded to core 2 and core 2's critical section can proceed (⑧). Note that in this case, as is often true, core 2's critical section is not actually waiting on the completion of core 1's critical section. Simply the action of cache coherence pulling the lock from one private cache to another is impeding critical section execution.

978-1-5386-6648-7/19/$31.00 ©2019 IEEE
DOI 10.1109/ICCD46524.2019.00041

If the lock sharing pattern could be predicted to allow locks to be preemptively forwarded to the next core that will use them, significant performance improvements could be achieved [38]. As shown in Fig. 1(b), immediately after the release of the lock at the end of core 1's critical section (①), core 2 receives a message (②) informing core 2 to preemptively request (prefetch) the lock cache line. Core 2 then initiates the request for the lock cache line (③-⑦) far ahead of its demand fetch during core 2's critical section (⑧). Thus, when core 2 is ready to acquire the lock, the cache line containing the lock will be in its local private cache. This is the primary goal of our proposed technique. *Note that forwarding the lock cache line in no way means that the lock is speculatively acquired by core 2*, it simply means that should core 2 try to acquire the lock it will take less time.

We show that many applications exhibit regular and repeatable lock[1] sharing patterns. Thus, we introduce SpecLock, an efficient hardware mechanism to predict which thread will request each lock upon its release. This prediction is made locally at each core using a small lookup table in the private caches that records past lock sharing patterns. When a lock is released, this table is accessed to determine if the cache line containing the lock should be preemptively forwarded to the core predicted to be the next acquirer, without waiting for an explicit request from that core. Speculative forwarding serves to hide the latency for the remote core to acquire the lock. In the event a misprediction occurs and the cache line containing the lock has been forwarded to the wrong core, the demand request for the lock from the correct core causes the coherence mechanism to forward that lock to the correct core. Mispredictions do not typically increase lock acquisition latency because, even if the lock is sent to a core that does not use it, the core that ends up actually requesting the lock would nevertheless have to fetch the lock from a remote core.

SpecLock has a distributed implementation, with no direct global communication. It applies to barriers and other synchronization constructs in addition to locks, does not affect program semantics, and natively supports nested locks. Importantly, SpecLock does not modify the coherence protocol, as that would require significant validation efforts [32], [9].

While simple, SpecLock captures 87% of the sharing patterns in our benchmarks correctly. By reducing lock acquisition latency, SpecLock provides a 7% average speedup for 16 cores and 10% for 64 cores. Compared to delayed lock release [49] and most-frequent acquirer [38] schemes which are popular alternatives, for 16 cores, these numbers become 4% and 8% respectively. SpecLock outperforms traditional prefetchers because it is initiated by the holder of the lock, only when that core is done using the lock and thus can better predict *when* to transfer the lock. SpecLock incurs a small power overhead of 0.14% and area of 0.2% compared to on-chip caches, 75% less than prior work. SpecLock does not penalize performance even for our less predictable benchmarks.

[1]we use the term "lock" broadly to imply any synchronization construct which leverages hardware synchronization constructs such as LLSC.

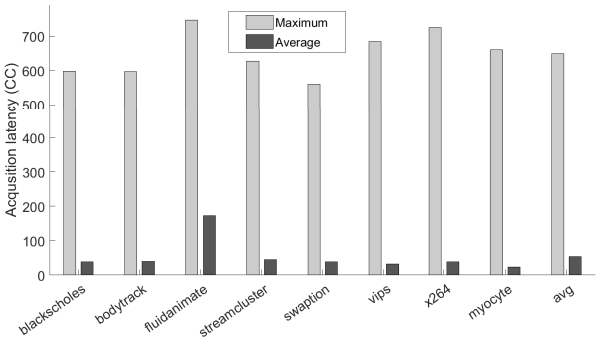


Fig. 2: Lock acquisition latency breakdown (16 cores).

II. Motivation

Fig. 2 shows average and maximum non-local lock acquisition latencies per benchmark. Here we define a non-local lock as one which resides in either a remote private cache or only in the shared LLC. Across all benchmarks, the average latency to acquire a non-local lock is ∼35 cycles, a considerable amount of time, while the average maximum is ∼655 cycles. Further, both the average and maximum latencies increase with core count due to increased distances and the possibility of contention. By contrast, the latency to acquire a lock that resides in the local L1 cache is the same as any other L1 cache hit, 3 cycles in this particular case.

Unfortunately, acquiring a lock from the local L1 cache is not the common case. In these experiments 77% of locks resided in a remote L1 at the time the lock is requested. These remote L1 locks incur long latencies because the local core must request the lock-containing cache line via the coherence mechanism from the remote code, facing possible contention and long distance of communication. Fetching a lock-containing cache line from the shared LLC (L2) incurs a somewhat better latency than that of a remote L1 request. However, only 2% reside in the shared LLC. Just 21% of locks are found in a core's local L1 cache once the core attempts to acquire the lock. The fraction of locks found in "main memory" is vanishingly small and barely visible in the figure. Thus, lock acquisition latency is more often than not that of a remote acquisition. As shown in Fig. 2, the average latency of remote lock acquisition is ∼10× that of a local acquisition.

A. Lock Sharing Patterns

Fig. 3 shows the various lock sharing patterns typically seen in some PARSEC benchmarks. In the figure, each node represents a core which acquires a given lock. Edges represent lock acquisition requests that can be local (loopback edges) or to remote cores. We see that different patterns have varying degrees of predictability. For example the "Weighted ping-pong" and "Long sequence" patterns (Fig. 3a and 3b), are perfectly predictable for each of the involved cores. We label this class of lock sharing patterns *Perfectly predictable*. The "Two way alternating ping-pong" and "One way alternating ping-pong" patterns (Fig. 3c and 3d) represent cases with highly-predictable behavior, where most of the cores involved can be accurately predicted most of the time. We classify

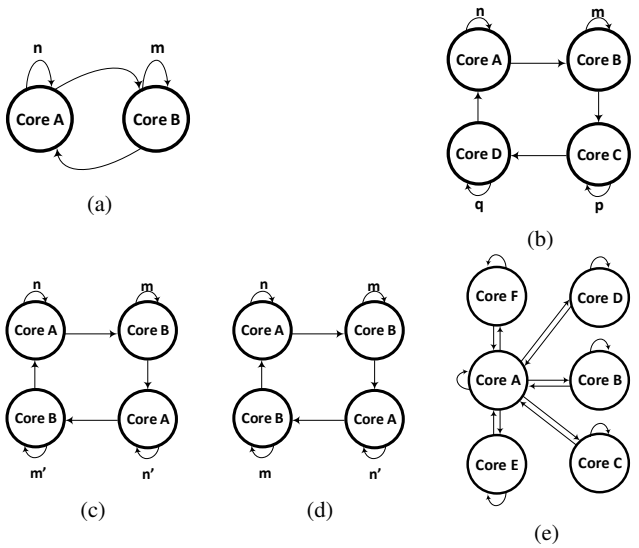


Fig. 3: Common lock acquisition patterns. (a) Weighted ping-pong - The lock is handed back and forth between two cores with multiple local iterations between handoffs. (b) Long sequence - A repeated, long sequence of lock sharers. (c) Two way alternating ping-pong - The lock is handed back and forth between two cores with multiple local iterations between handoffs, the iteration number alternates for both cores. (d) One way alternating ping-pong - The lock is handed back and forth between two cores with multiple local iterations between handoffs, the iteration number alternates for one of the cores. (e) Star sequence - The lock is passed from one primary core followed by one of a set of other cores.

these patterns as *Highly predictable*. The "Star sequence" and the "Alternating sequence" patterns (Fig. 3e represents cases were at least some of the cores show predictable behavior (*e.g.* Cores B - F in Fig. 3e), although some are not easily predictable (*e.g.* Core A). We classify this set of patterns as *Predictable*. We classify lock acquisition patterns that appear random as *Unpredictable*.

Fig. 4, shows a breakdown of lock acquisition patterns based upon the aforementioned classifications. On average, 26% of the acquisitions are predictable. Because many of these patterns show multiple local re-acquisitions prior to a remote acquisition, a predictor must not only predict the next requester but also accurately predict how many times the lock will be locally re-acquired prior to a remote request. Further, the predictor should be conservative to not forward the lock to a predicted acquirer unless it is certain the lock will not be re-acquired locally, as this condition could hurt performance.

III. SpecLock: Speculative Lock Forwarding

SpecLock speculates on the next core to request a given lock once it is released, and forwards that lock to the predicted core, hiding the latency of remote lock cache line acquisition. SpecLock uses a counting algorithm which relies on previously recorded local lock reacquisition counts to predict when the lock should be forwarded to the next core. SpecLock's

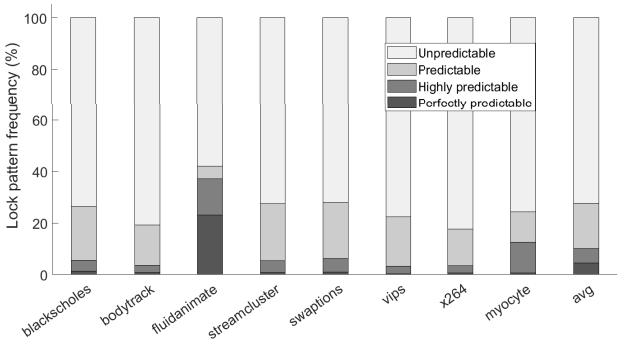


Fig. 4: Lock pattern breakdown. Only a forwarding action (not hold) is considered.

predictor is distributed with each core's private caches speculating on the next acquirer of local locks once those locks are released. Thus, SpecLock can accurately predict any lock whose count of local re-acquisitions and next acquiring core is static (*i.e.* all cores in Fig. 3a, and 3b, as well as some of the cores in Fig. 3c, 3d, and 3e. Speclock identifies locks by the memory addresses synchronization instructions, such as LLSC, operate on.

SpecLock's operation can be divided into two phases: *training* and *prediction*. Each lock goes through the *training* and *prediction* phases independently of other locks. The *training* phase learns both the number of consecutive local re-acquisitions by the same core (the loopback edge weights in Fig. 3, labeled *weight* from here on), as well as the most likely next acquiring core. Once SpecLock's predictor becomes confident in its predictions for a given lock, the predictor entry for that lock enters the *prediction* phase. In the prediction phase, each time a given lock is re-acquired from another core, SpecLock will begin counting local re-acquisitions until the number of local re-acquisitions matches the expected number. Thereafter, the cache line containing the lock will be forwarded to the predicted next requesting core.

A. Operation

Fig. 5 and 6 detail SpecLock's operation. To facilitate SpecLock's lock prediction each core's private caches are augmented with a small *Lock Prediction Table* (LPT), shown in Fig. 5. The purpose of this table is to track the state of locks recently acquired by the local core. Each LPT entry contains the following components: a valid bit (V); the lock's effective address (Tag); a *weighted* FIFO, containing the local reacquisition counts from the last n times this lock has been acquired by the given core[2]; the core ID of the last remote acquisition (LA) (*i.e.,* the last core to force an invalidation of the lock containing line away from this core); a Premature Forward Counter (PFC) to record the total number of times the lock was forwarded only to be subsequently reacquired before it was used at the remote core (we call this false-forward condition a "premature forward"); and a current acquisition

[2]Note, in the following example we will define the FIFO depth of n=2, thus W0 and W1 in Fig. 6

Entry Valid	TAG	Weight FIFO N	Weight FIFO N-1	Last Acquirer	Premature Forward Count	CAC
1	0x15040	15	24	2	0	1
1	0xF0A80	43	43	15	3	7
	⋮					
0	-	255	255	-	0	0

Fig. 5: Each core's private L1 cache has a lock prediction table (LPT) which is a set-associative table, indexed by the lock's physical address (TAG). The LPT is organized as a 4-way set associative cache.

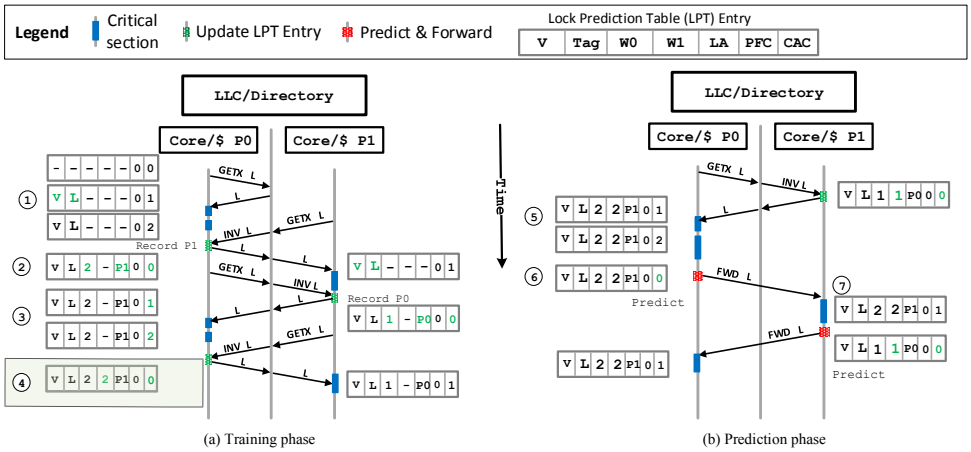


Fig. 6: Lock prediction using SpecLock. The lock pattern is "Weighted Ping-Pong" (Fig. 3a) where P0 acquires the lock twice before P1 acquires the lock once, then the pattern repeats.

counter (CAC), which counts the number of times the most recently accessed lock has been reacquired by this core.

Fig. 6 follows the life cycle of a single, per-lock, LPT entry, first through the *training* phase and then through the *prediction* phase. In the *training* phase (Fig. 6.a), when a previously unseen lock is first **acquired** (① in the figure), a new entry in the LPT is allocated and cleared (the LPT organization and replacement is discussed in Section III-C). SpecLock only takes into account successful lock acquisitions. The *V* bit and the lock's *Tag* are recorded. The *CAC* is then incremented for each time the lock is released and reacquired (blue critical sections on the Core/$ P0 line) prior to the lock cache line being **invalidated** by being requested by another core. When another core later **invalidates** the cache line containing the lock (②), the core causing the invalidation is noted in the *LA* field (P1 in this example), and the number of reacquisitions is pushed into the *weights* FIFO (number 2 is placed in *W0* here). We note, the *training* phase works similarly regardless of whether the lock is currently held at the time an invalidation arrives, to make SpecLock functional for both contended and non-contended locks.

To reduce the probability of a premature forward, we bias the predictor towards predicting to hold the lock by choosing the maximum of the reacquisition counts in the weights FIFO (W0 and W1 in Fig. 6). We make this choice because premature forwards actually introduce a latency penalty compared to a baseline system where the lock would remain and simply be reacquired by the releasing core; by contrast, *conservatively not forwarding locks matches the behavior of the baseline system*. Later, when core P0 acquires the lock from P1 (③), the LPT entry for this lock is found. Since there is already an entry for this lock, the LPT entry is not changed, but the *CAC* is reset to start counting the number of reacquisitions of the given lock. Because the FIFO is not full (*n* in this example is 2), no prediction will be made yet. When core P1 invalidates the lock cache line (④), the CAC has been incremented to 2, thus the number 2 is pushed into the *weights* FIFO. Now the LPT entry is full and ready for prediction (*i.e.* the entry enters the *Prediction* phase). Note that a *weights* FIFO depth of $n = 2$ represents a relatively aggressive predictor which will begin prediction after only two iterations of training. Other FIFO depths are possible, *e.g.* $n = 1$ would extremely aggressively forward locks after only seeing a given pattern once, while $n = 4$ would more conservatively require 4 consistent iterations before prediction would occur.

The *prediction* phase is shown in Fig. 6.b. In the figure, prediction first starts with the predicting core **acquiring** a lock for which an LPT entry has already been trained (⑤). An LPT entry is considered trained when all n *weights* in the *weights* FIFO are full and the premature forward counter (PFC) is below a given threshold (g). In this event, as in training, the *CAC* is reset and each local reacquisition causes an increment. When the *CAC* is equal to the greatest of the weights in the *weights* FIFO (⑥), then the cache line containing the lock is speculatively forwarded to the predicted core (P1 in this case). In the figure, we also see that P1's LPT entry for this lock has also been trained. Thus, once P1's *CAC* hits its target, the lock is speculatively forwarded back to P0 (⑦).

1) Outcomes: SpecLock operates only on cache lines containing locks. Moving a cache line containing a lock to a remote core does not imply the thread in that core has acquired the lock. Therefore, program semantics remain intact and no rollback or other handling is required in the event of a misprediction. At worst an incorrect prediction would only lead to excess movement of the cache line. In addition, since predictor state is maintained separately per lock address, nested locks are orthogonal and supported. In the event that a single cache line contains multiple locks, all locks are part of the same predictions. Furthermore, if a thread if migrates to another core, the corresponding LPT lines are cleared to avoid mispredictions for the next thread.

B. SpecLock Messages

Modifications to the coherence protocol are not to be taken lightly, as the effort required in validation of additional durable

or transient states can be quite high [32], [9]. Thus, SpecLock does not modify[3] the coherence protocol, but instead adds two non-coherence "hint" messages.

The first and most important message is the "lock forward" (*LOCK_FWD*) message. Upon predicting that a lock should be transferred to another core, the releasing core sends this control message to the recipient core, (⑥ in Fig. 6). This message is a prefetch hint to the receiving core that it should issue a read exclusive to its local L1 cache for this address. The receiving core then invokes the unmodified coherence protocol (e.g., MESI [53]) to invalidate remote copies, ⑥ and claim ownership of the lock in its local L1 cache, ⑦.

Although this method introduces one extra round-trip of communication, we make this choice because cache coherency protocols such as MESI do not support receiving cache lines that they did not request. By making the cache line request originate from the receiving core, the cache coherency protocol is invoked the same as if the recipient core requested the cache line. This way, the *LOCK_FWD* message only needs to reach the cache coherence controller in the recipient core's L1 cache and not the core itself. Also, no modifications are needed to the cache coherency protocol.

In the event that a given lock is either forwarded to an *incorrect destination* or *prematurely forwarded* (case 3 or case 4 in Section III-A1), the sending core would never receive feedback as to the accuracy of the predicted next acquirer of the lock. Thus, SpecLock uses an additional message, the *negative acknowledgment (NACK)*, to gain feedback on the quality of its predictions. This feedback message is sent from the recipient core back to the predicting core. If the lock was acquired by the recipient core, the prediction was correct and no message is sent back. Otherwise, if the lock was acquired by any other core before the recipient lock acquires it, a *NACK* is sent back to indicate that the prediction was incorrect. In the event that the *NACK* was caused by a premature forward, the releasing core will increment the *PFC* field of the associated LPT entry. If the total number of premature forwards surpasses a threshold, the LPT entry for that lock is erased to prevent future premature forwards[4]. Note there is no dependency between the NACKs and $LOCK\_FWD$ messages, thus they need not be placed in different virtual channels (VCs) [11]. both *LOCK_FWD* and *NACK* messages can be dropped with no impact on correctness, but a possible performance degradation.

C. Lock Prediction Table (LPT) Organization

To keep SpecLock scalable, each core maintains its own LPT to record history, as shown in Fig. 5. Entries in the LPT are allocated upon new lock acquisition as described in the previous sections. The LPT is organized as a 4-way set associative cache. If the LPT entries are all full, a replacement candidate must be chosen. Through experiments, we observed that many locks do not have any sharers. Thus, we first try to replace any entry which has a *Last acquirer* field empty. If no such entry exists, the entry with the highest total number of premature forwards will be replaced.

[3]It is assume the requester core-id exists in the request message.

[4]We also examined forwarding the lock to the LLC in the case of too many inaccurate predictions. However, we found this to have no significant impact on performance and dropped it from the final technique.

D. Implementation

The size of the LPTs and the depth of each FIFO poses minimal overhead, shown in Fig. 5. Each LPT entry is 12 bytes assuming the following: 64-bit addresses, a FIFO of depth two (each 8 bits), 6 bits for core identifiers (64 cores), 4 bits for premature forward count, and 8 bits for CAC. We empirically determined that the number of live, shared locks rarely exceeded 100 across the benchmarks we examined. Thus, we implemented 128 entries in the LPT per core. Therefore, the LPT is $\sim$2KB per core. Assuming 64MB total on-chip caching, the additional storage for the LPTs is negligible (0.2%). We quantify the power overhead of accessing the LPTs in Section V-F. In order to be able to send ACK/NACK response, a small 16 entry 9-byte table is implemented in each core to buffer 64-bit addresses and 6 bits for core identifiers.

Given the small size, 128 entries and 4-way set associativity of the LPT, we estimate two cycles is all that is required to read or write the appropriate row. A further cycle is required for any modifications prior to writing the row back. This delay has little impact because lock forwarding speculation need only be as fast as the time between lock release and the next acquisition for that lock. That time is rarely small enough that an additional few cycles to make the prediction would have an impact. This observation lets us pipeline the lock prediction structures as needed to increase clock frequency. Therefore, the additional lock speculation logic has no impact to the chip multiprocessor (CMP)'s clock frequency. Further, the LPT is only accessed upon a write to a lock cache line, thus it has no impact on L1 cache access time.

IV. Methodology

We perform full system simulations using the GEM5 simulator with the Ruby memory system and the Garnet network on chip model [4]. We use a CMP with 16/64 Alpha cores [40], 32KB 2-way L1 cache with 3 cycle-access, a shared distributed (2MB per core) 8-way set L2 cache with 15 cycle-access, a 128 entries 4-way LPT with 2 cycle-access, and 4-hop MESI coherence protocol. Also, four Micron MT41J512M8 memory controller (one at each corner), a DDR3-1600 x64 channel, and 4×4, 8×8 mesh NoC is considered. We show results below for both in-order cores and out-of-order cores. Unless otherwise specified, there is no prefetching. We discuss the impact of prefetching in Section V-E. PFC threshold (g) is set to 2 in order to minimize premature forwards and the weighted FIFO size is set to 2. Also, there is one thread per core with no migration.

We use applications from the PARSEC [3] and Rodinia [8] benchmark suites that represent a variety of lock sharing patterns with their pre-defined medium-size inputs by default. We use Cacti 6.5 with a 32nm process [37] to estimate the power and area of caches and LPTs. For power, we also use the

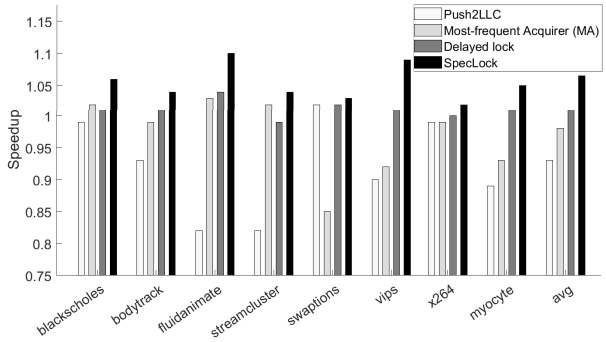


Fig. 7: Speedup (execution time decrease) for different benchmarks normalized to the baseline with 16-cores.

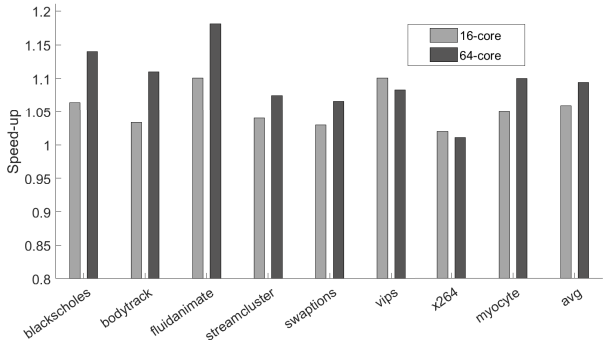


Fig. 8: SpecLock speed-up (execution time decrease) comparison for 16- and 64-core CMPs normalized to the baseline.

activity factors of each structure of SpecLock. We configure SpecLock as described in Section III-D.

We evaluate SpecLock against the following schemes:

- *Baseline* no lock speculation.
- *Push2LLC* where locks are always always pushed to the LLC when released.
- All four schemes proposed by Lucia *et al.* [38] (though not evaluated in their paper) including: First-to-acquire (FA), Last-to-acquire (LA), Most-frequent-transition (MT), and Most-frequent-acquire (MA). We find that MA is their most accurate scheme and so it is used exclusively in several graphs to be more succinct.
- *Delayed Lock* a delay scheme, Rajwar *et al.* [49].

In *baseline*, to maintain inclusivity, the current holder of a lock first forwards it to the directory, which then forwards it to the requester. In *MA*, each core forwards locks to a remote core that most often requests it. In *delayed lock*, once a request arrives from a remote core for a lock, in the event that the lock is currently held, the response is delayed by a configurable number of cycles to provide time for that lock to be released and avoid unnecessary coherence events. We configure the delay response to be 128 cycles, derived from observing average lock re-use times and round-trip latencies.

V. Evaluation

A. Execution Time

Fig. 7 shows execution time results for SpecLock compared to Push2LLC, MA, and Delayed Lock on the 16-core system. As shown in Figure 7, SpecLock provides an average 7% speedup for 16 cores (results improve to 10% for 64 cores as discussed in Section V-B), compared to the baseline.

Looking at individual benchmarks, the largest speedups are for *fluidanimate*, followed by *streamcluster* and *blackscholes*. These benchmarks tend to be more well balanced [17], which means that the lock acquisition latency is more likely to dominate the critical path. *Fluidanimate* in particular exhibits many ping-pong style lock sharing patterns, which are more easily predictable. In contrast, benchmarks less well balanced (*e.g.* those which are pipeline parallel such as *x264*) see less impact as any latency reduction is overshadowed by other overheads.

Compared to *MA* [38] and *Delay Lock* [49], SpecLock sees an 8% and 4% speedup improvement, respectively. Strikingly, *MA* actually decreases performance versus baseline on average. This is because MA does not take into account repeated re-acquisitions locks by the local core, thus for many benchmarks *MA* leads to many "Premature Forwards" (see Section III-A1) which incur extra latency as the lock cache line is shuttled back and forth for benchmarks with many re-acquisitions (*bodytrack*, *swaptions* and *vips*). In fact none of the four predictors defined by Lucia *et al.* [38] can appropriately predict the repeated reuse of locks by the same core that we see in real workloads, leading to high levels of premature-forwarding. Predicting the repeated acquisition pattern seen in real workloads is a principal advantage of SpecLock. Similarly, we see that *Push2LLC* serves to hide acquisition latency for some benchmarks without the possibility of forwarding to the wrong core. However, as with MA, Push2LLC fails to consider re-acquisition by the same core, as we later elaborate.

We also note, Delayed Lock and SpecLock are orthogonal in that they address different components of critical section latency (contended lock delay versus coherence delay in non-contended locks). Implementing both techniques together provides further, small ($\sim$1%) speedup average above SpecLock by itself, across these largely non-contended benchmarks.

B. Scalability

Fig. 8 shows the performance scalability for SpecLock going from 16 cores to 64 cores. For this number of cores, SpecLock scales well, with an average performance increase from 7% to 10%. Generally, going to 64 cores leads to more coherence delays as more locks are transferred between cores, thus most benchmarks experience some gain. In *vips* and *x264*, performance degrades slightly going to 64 cores. In *vips*, lock contention becomes more common as the benchmark scales. *x264* is a pipeline parallel benchmark, thus it exhibits poor load balance, which worsens as it scales to 64 cores.

C. Accuracy

Fig. 9 shows the accuracy breakdown for lock forward speculation per benchmark for 16 cores, baseline versus four lock prediction schemes proposed in [38] and SpecLock (Spec). Speculation is considered accurate if it correctly predicts that

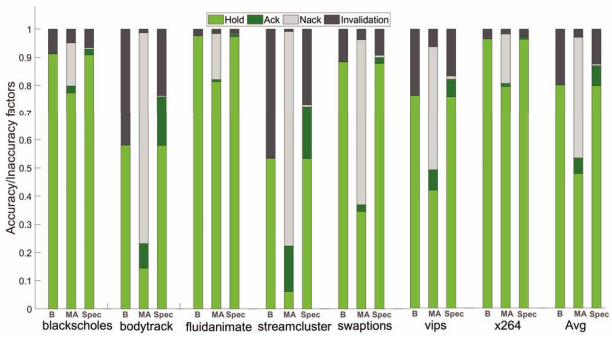


Fig. 9: Accuracy comparison for Baseline (B), Most-frequent-acquire (MA), and SpecLock (Spec), from left to right. The green colors (Hold and Ack) reflects the accuracy obtained through lock prediction. The inaccuracy is represented by gray colors (Nack and Invalidation).

the lock will be reused and does not forward it (*Hold*) or if it accurately predicts the recipient core, forwards the lock to that core, and that core uses the lock before eviction (*Ack*). A prediction is considered inaccurate if the lock is invalidated while being held (*Invalidation*) or if it is sent to another core which does not use it prior to eviction (*Nack*). On average, SpecLock's total prediction accuracy (*Hold* plus *Acks*) is **87%** across benchmarks, with *streamcluster* being the highest at 96% and *bodytrack* the lowest at 46%.

To illustrate the native number of *holds* and invalidates for these benchmarks, Baseline (no prediction) is included in the figure. We notice that the number of *holds* is comparable between SpecLock and baseline. This indicates that SpecLock avoids premature forwards, which leads to less *invalidations* and improve performance. This is in sharp contrast to the other four predictors where premature forwards occur often and show low accuracy as evident by the number of *Nacks*. When SpecLock does forward a lock, it shows good accuracy as evident by the number of *Acks*.

Interestingly, we see that although the best performing benchmark, *fluidanimate*, has a high prediction accuracy, it only shows a small fraction of actual forwards (*Acks* plus *Nacks*) relative to *Holds*. In *fluidanimate*, although remote acquisitions are greatly outnumbered by local acquisitions, there remain enough remote acquisitions to impact performance when accurately predicted. We see that *MA* significantly outperforms the other techniques proposed in [38]. Nevertheless, all are much less accurate than SpecLock with the same number of prediction table entries, with an average of only ∼50% across our benchmarks. This is largely because of these techniques' inability to predict local re-acquisitions.

D. Out-of-Order vs. In-Order

We also examine the impact of SpecLock on out-of-order (OOO) cores. Fig. 10 presents speedup results of SpecLock on an OOO machine normalized to an OOO baseline for a 16-core CMP. The figure also reproduces the results for SpecLock on an in-order machine from Fig. 7 normalized against an in-order baseline. The figure shows slightly lower speedup with

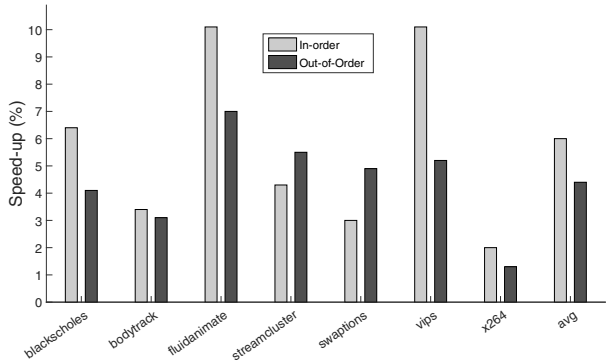


Fig. 10: Speed-up of SpecLock running on out-of-order cores versus in-order.

out-of-order cores versus in-order cores. This is because OOO execution can cover only a small amount of the latency of synchronization instructions (e.g., LLSC) due to coherence. Generally, OOO cores must serialize to ensure correctness when entering a critical section. An OOO core cannot execute around locks to improve performance as with typical memory instructions and thus, an OOO core behaves like an in-order core when executing those instructions.

E. Prefetching

To study the effects of prefetching, we enable a stream prefetcher with a typical configuration in each core's private L1 cache [25]. Each core's prefetcher uses 4 streams, 4 cache misses before creating a stride (training misses), 8 non-unit stride filters, and 8 unit stride filters. we see that on average, enabling the prefetcher actually hurts performance slightly (without SpecLock), as the same is reported in [26], [2]. This is because in some benchmarks, naive prefetching can cause a premature forwarding of shared data and/or locks between threads, leading to greater memory latency and lost performance as this data is ping-ponged around. Unlike SpecLock, prefetchers are poor at predicting the *time* locks need to be prefetched, because prefetchers are typically receiver-initiated but the optimal time depends on when the sender releases lock.

F. Area and Power Overhead

Considering an 128-entry LPT with 12-byte entries and four-way set associativity, the power consumption and area is just less than 2 mW and 2291 um^2 . In contrast, the CMP's collection of 32KB L1 and 2MB L2 cache space per core (128MB in total for 64 cores) consumes 69 mW and 1454 mW, respectively [37]. The area occupied by these caches is 68841 um^2 and 1160000 um^2 respectively. The per-core LPTs represent a negligible power and area increase relative to the on-chip caches of our CMP. In fact, for single-threaded applications, the LPT structure can be power gated to eliminate static power as well. In addition, SpecLock's LPT structure is 75% smaller than the equivalent structures for MA with the same number of entries.

In addition, SpecLock actually decreases bytes transferred over the network by approximately 2% on average. Here, the reduced sharing latency actually leads to shorter critical

section times, which in turn reduces the chance that another core will cause a lock invalidation during the critical section. This reduces the number of invalidation messages.

VI. Discussion

Lock speculation is best left to hardware as most locking primitives are implemented in the OS through posix system calls. As a result, it would be very difficult for the application programmer or compiler to insert appropriate prefetch instructions ahead of a given system call. System calls themselves are quite short so inserting the prefetch instructions in the OS at compile time would do little to hide the lock access latency. Further, lock acquisition patterns would be difficult to predict at compile time since they are often data dependent.

Here, SpecLock is implemented in a system using the Alpha ISA with LLSC as the underlying synchronization instruction. Nevertheless, SpecLock is easily implementable with minor modifications in systems employing single, atomic, read-modify-write instructions such as the x86 XCHG instruction.

VII. Related Work

Push protocols [44] add prediction information to directories in order to predict future readers of cache lines. However, such protocols attempt exclusive push predictions only when the sharing pattern is observed to be between just two nodes. Alternative sharer prediction protocols [29] are tailored to cache lines with multiple readers, and thus do not directly apply to exclusive acquisitions, such as for locks.

Further past work motivates lock prediction by describing use cases and designing simple predictors [38]. We show that those methods compare poorly to SpecLock. The predictors described in [38] cannot appropriately predict the repeated reuse of locks by the same core that we see in real workloads, leading to high levels of premature-forwarding. The principle novelty of our approach lies in counting local reacquisitions to avoid premature forwards and reduce the latency of acquisition of non-contended locks. In addition, [38] does not implement those predictors in a system.

Biased locks [58], [10], [51] identify a dominant thread for each lock that is most likely to request it at any one time, and always forward locks to the dominant thread. This differs from SpecLock in that all cores make the same prediction for each lock. Taking into account the releasing thread has been shown to outperform approaches that do not [38].

SpecLock is orthogonal to dead block prediction [31] because this prediction simply concerns itself with identifying blocks (cache lines) that are likely to be dead, not where they will be requested from next. This is much like dynamic self-invalidation [34]. In addition, this research shows that counting algorithms do not do well in dead block prediction, whereas we show they work well for lock prediction.

Hardware and software support for efficient locks has been well researched in multiprocessor systems [21], [27]. Hardware support for fairness sequences threads in FIFO order according to the time they attempted to acquire the lock [33], [57], [52], [20], [36], [18]. Alternative approaches record and maintain the state of frequently synchronized data with hardware support [62]. Fairness can also be achieved entirely in software [43].

Tournament locks and barriers also increase the scalability of locks by using a communication pattern of a binary tree [19]. Real-time nested protocol (RNLP) locks are an alternative fine-grain locking mechanism [59]. Other work implements FIFO ordering in software instead [45].

Further work proposed techniques that speculatively execute critical sections [50], [39], [50], [46]. Such techniques usually focus on performance at the expense of complex mechanisms to abort execution and revert to a previous state and to detect conflicts. Speculative lock elision (SPE) [50] prefetches data in a critical section, not the lock itself. Thus, SPE is orthogonal and complementary to SpecLock.

To address messaging caused by spinning threads, prior work uses different but related cache lines for threads spinning on the same lock [41], [13]. Another approach associates a private mutually exclusive variable (mutex) for each cache line at each core [23]. Alternatively, spinning can be performed in the memory controller with dedicated hardware support [42]. The memory controller can also be extended to perform atomic operations in order to reduce the amount of communication [61]. Dedicated hardware can also accelerate message passing and task scheduling to avoid modifying the memory controller [48]. Alternative approaches propose synchronization without the use of locks such as tagged memory [1], [22], [30], [12].

SpecLock is synergistic to techniques that make the lock acquisition pattern more predictable and speculative critical section execution [50], [39] because it helps with cases where locks are not forwarded to the correct core. In addition, SpecLock reduces the cases where speculative critical section execution is necessary.

VIII. Conclusion

In this paper, we introduce SpecLock. SpecLock predicts lock sharing patterns and speculatively forwards locks as they are being released to the core that is most likely to request them next. This hides the latency of the next core to acquire the lock. Cores make this prediction independently and thus in a scalable manner and without modifying the cache coherence protocol. SpecLock captures 87% of the sharing patterns correctly. By reducing lock acquisition latency, SpecLock provides a 7% average speedup for 16 cores and 10% for 64 cores, shows good scalability, and outperforms competition with a smaller area (75% reduction) and power overhead. SpecLock advances the state of the art by predicting *both the acquisition pattern and time* of locks.

Acknowledgments

This work was supported by ARPA-E ENLITENED Program (project award DE-AR00000843) and the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This work was also supported by the National Science Foundation through grants FoMR-1823403 and I/UCRC-1439722.

REFERENCES

[1] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.H. Lim, K. Mackenzie, and D. Yeung. The MIT alewife machine: architecture and performance. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 2–13, June 1995.

[2] L. M. AlBarakat, P. V. Gratz, and D. A. Jimnez. Mtb-fetch: Multithreading aware hardware prefetching for chip multiprocessors. *IEEE Computer Architecture Letters*, 17(2):175–178, July 2018.

[3] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[5] Shekhar Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, 2007.

[6] Surendra Byna, Yong Chen, and Xian-He Sun. A taxonomy of data prefetching mechanisms. In *Proceedings of the The International Symposium on Parallel Architectures, Algorithms, and Networks*, ISPAN '08, pages 19–24, 2008.

[7] L. Chang, D.J. Frank, R.K. Montoye, S.J. Koester, B.L. Ji, P.W. Coteus, R.H. Dennard, and W. Haensch. Practical strategies for power-efficient computing technologies. *Proceedings of the IEEE*, 98(2):215–236, Feb 2010.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[9] Xiaofang Chen, Yu Yang, G. Gopalakrishnan, and Ching-Tsun Chou. Reducing verification complexity of a multicore coherence protocol using assume/guarantee. In *Formal Methods in Computer Aided Design, 2006. FMCAD '06*, pages 81–88, 2006.

[10] Jimmy Cleary, Owen Callanan, Mark Purcell, and David Gregg. Fast asymmetric thread synchronization. *ACM Trans. Archit. Code Optim.*, 9(4):27:1–27:22, January 2013.

[11] William J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2), 1992.

[12] William J. Dally, J. A. Stuart Fiske, John S. Keen, Richard A. Letbin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gregory A. Fyler. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE MICRO*, 12:23–39, 1992.

[13] A. Dave, N. Islam, and R.H. Campbell. A low-latency scalable locking algorithm for shared memory multiprocessors. In *Parallel and Distributed Processing, 1994. Proceedings. Sixth IEEE Symposium on*, pages 10–17, Oct 1994.

[14] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, 2011.

[15] Stijn Eyerman and Lieven Eeckhout. Modeling critical sections in amdahl's law and its implications for multicore design. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 362–370, 2010.

[16] Babak Falsafi and Thomas F Wenisch. A primer on hardware prefetching. *Synthesis Lectures on Computer Architecture*, 9(1):1–67, 2014.

[17] Ehsan Fatehi and Paul Gratz. ILP and TLP in shared memory applications: A limit study. In *the 23rd International Conference on Parallel Architectures and Compilation (PACT)*, pages 113–126, 2014.

[18] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS III, pages 64–75, 1989.

[19] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23(6):60–69, June 1990.

[20] Bijun He, William N. Scherer, and Michael L. Scott. Preemption adaptivity in time-published queue-based spin locks. In *Proceedings of the 12th International Conference on High Performance Computing*, HiPC'05, pages 7–18, 2005.

[21] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[22] Yatin Hoskote, Sriram Vangal, Arvind Singh, Nitin Borkar, and Shekhar Borkar. A 5-ghz mesh interconnect for a teraflops processor. *IEEE Micro*, 27(5):51–61, 2007.

[23] W.C. Hsieh and W.W. Weihl. Scalable reader-writer locks for parallel systems. In *Parallel Processing Symposium, 1992. Proceedings., Sixth International*, pages 656–659, Mar 1992.

[24] Wei Huang, K. Rajamani, M.R. Stan, and K. Skadron. Scaling with design constraints: Predicting the future of big chips. *Micro, IEEE*, 31(4):16–29, July 2011.

[25] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 397–408, Dec 2006.

[26] N. D. E. Jerger, E. L. Hill, and M. H. Lipasti. Friendly fire: understanding the effects of multiprocessor prefetches. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 177–188, March 2006.

[27] Alain Kägi, Doug Burger, and James R. Goodman. Efficient synchronization: Let them eat qolb. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 170–180, 1997.

[28] Miray Kas. Toward on-chip datacenters: a perspective on general trends and on-chip particulars. *The Journal of Supercomputing*, 62(1):214–226, 2012.

[29] S. Kaxiras and C. Young. Coherence communication prediction in shared-memory multiprocessors. In *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, pages 156–167, 2000.

[30] Stephen W. Keckler, William J. Dally, Daniel Maskit, Nicholas P. Carter, Andrew Chang, and Whay S. Lee. Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, pages 306–317, Washington, DC, USA, 1998. IEEE Computer Society.

[31] S. M. Khan, Y. Tian, and D. A. Jimenez. Sampling dead block prediction for last-level caches. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 175–186, Dec 2010.

[32] Rakesh Komuravelli, Sarita V. Adve, and Ching-Tsun Chou. Revisiting the complexity of hardware cache coherence and some implications. *ACM Trans. Archit. Code Optim.*, 11(4):37:1–37:22, December 2014.

[33] O. Krieger, M. Stumm, Ron Unrau, and Jonathan Hanna. A fair fast scalable reader-writer lock. In *Parallel Processing, 1993. ICPP 1993. International Conference on*, volume 2, pages 201–204, Aug 1993.

[34] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 48–59, June 1995.

[35] Daniel E. Lenoski and Wolf-Dietrich Weber. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[36] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable reader-writer locks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 101–110, 2009.

[37] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. CACTI-P: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '11, pages 694–701, 2011.

[38] Brandon Lucia, Joseph Devietti, Tom Bergan, Luis Ceze, and Dan Grossman. Lock Prediction. In *USENIX Hot Topics on Parallelism (HotPar)*, 6 2010.

[39] Jose F. Martinez and Josep Torrellas. Speculative locks for concurrent execution of critical sections in shared-memory multiprocessors. Technical report, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2001.

[40] Edward McLellan. The alpha axp architecture and 21064 processor. *IEEE Micro*, 13(3):36–47, May 1993.

[41] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[42] M. Monchiero, G. Palermo, C. Silvano, and O. Villa. Efficient synchronization for embedded on-chip multiprocessors. *Very Large Scale*

Integration (VLSI) Systems, IEEE Transactions on, 14(10):1049–1062, Oct 2006.

[43] H. Mushtaq, Z. Al-Ars, and K. Bertels. Detlock: Portable and efficient deterministic execution for shared memory multicore systems. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 721–730, Nov 2012.

[44] Malek Musleh and Vijay S. Pai. Automatic sharing classification and timely push for cache-coherent systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 13:1–13:12, 2015.

[45] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 97–108, 2009.

[46] Dimitra Papagiannopoulou, Giuseppe Capodanno, Tali Moreshet, Maurice Herlihy, and R. Iris Bahar. Energy-efficient and high-performance lock speculation hardware for embedded multicore systems. *ACM Trans. Embed. Comput. Syst.*, 14(3):51:1–51:27, May 2015.

[47] Joan-Manuel Parcerisa and Antonio Gonzalez. Improving latency tolerance of multithreading through decoupling. *IEEE Trans. Comput.*, 50(10):1084–1094, October 2001.

[48] P.G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, and G. Nicolescu. Parallel programming models for a multi-processor soc platform applied to high-speed traffic management. In *Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on*, pages 48–53, Sept 2004.

[49] R. Rajwar, A. Kagi, and J.R. Goodman. Improving the throughput of synchronization by insertion of delays. In *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, pages 168–179, 2000.

[50] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 294–305, 2001.

[51] Kenneth Russell and David Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 263–272, 2006.

[52] B. Saglam and V. Mooney, III. System-on-a-chip processor synchronization support in hardware. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '01, pages 633–641, 2001.

[53] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

[54] Daniel J. Sorin, Jonathan L. Lemon, Derek L. Eager, and Mary K. Vernon. Analytic evaluation of shared-memory architectures. *IEEE Trans. Parallel Distrib. Syst.*, 14(2):166–180, February 2003.

[55] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 269–280, 2010.

[56] N. Tuck and D.M. Tullsen. Multithreaded value prediction. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 5–15, Feb 2005.

[57] Enrique Vallejo, Ramon Beivide, Adrian Cristal, Tim Harris, Fernando Vallejo, Osman Unsal, and Mateo Valero. Architectural support for fair reader-writer locking. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 275–286, Washington, DC, USA, 2010. IEEE Computer Society.

[58] Nalini Vasudevan, Kedar S. Namjoshi, and Stephen A. Edwards. Simple and fast biased locks. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 65–74, 2010.

[59] Bryan C. Ward and James H. Anderson. Multi-resource real-time reader/writer locks for multiprocessors. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 177–186, 2014.

[60] R. Yung, S. Rusu, and K. Shoemaker. Future trend of microprocessor design. In *Solid-State Circuits Conference, 2002. ESSCIRC 2002. Proceedings of the 28th European*, pages 43–46, Sept 2002.

[61] Lixin Zhang, Zhen Fang, and John B. Carter. Highly efficient synchronization based on active memory operations. In *In International Parallel and Distributed Processing Symposium*, 2004.

[62] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R. Gao. Synchronization state buffer: Supporting efficient fine-grain synchronization on many-core architectures. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 35–45, 2007.