CASPAR: Breaking Serialization in Lock-Free Multicore Synchronization *

Tanmay Gangwani
University of Illinois at Urbana–Champaign
gangwan2@illinois.edu

Adam Morrison
Technion—Israel Institute of Technology
mad@cs.technion.ac.il

Josep Torrellas
University of Illinois at Urbana–Champaign
torrella@illinois.edu

Abstract

In multicores, performance-critical synchronization is increasingly performed in a *lock-free* manner using atomic instructions such as CAS or LL/SC. However, when many processors synchronize on the same variable, performance can still degrade significantly. Contending writes get serialized, creating a non-scalable condition. Past proposals that build hardware queues of synchronizing processors do not fundamentally solve this problem—at best, they help to *efficiently serialize* the contending writes.

This paper proposes a novel architecture that *breaks* the serialization of hardware queues and enables the queued processors to perform lock-free synchronization *in parallel*. The architecture, called CASPAR, is able to (1) execute the CASes in the queued-up processors in parallel through eager forwarding of expected values, and (2) validate the CASes in parallel and dequeue groups of processors at a time. The result is highly-scalable synchronization. We evaluate CASPAR with simulations of a 64-core chip. Compared to existing proposals with hardware queues, CASPAR improves the throughput of kernels by 32% on average, and reduces the execution time of the sections considered in lock-free versions of applications by 47% on average. This makes these sections 2.5x faster than in the original applications.

Keywords lock-free synchronization; serialization

1. Introduction

The arrival of large multicores such as Intel's Xeon Phi [27] provides renewed impetus to develop highly-threaded applications that share data in a fine-grained manner. Examples of such applications can be found in the traditional domains of numerical and database [15, 63] computing, as well as in the runtimes of emerging programming frameworks, such as Galois [50] for graph analytics, and those of Google's Go [1] and Mozilla's Rust [45] languages.

Fine-grained applications require efficient synchronization to manage shared data structures. For highest performance, they often employ *lock-free* synchronization [25], which avoids the overheads of using locks [13, 15, 19, 35, 47, 59, 61, 63]. Lock-free synchronization forgoes locking by directly manipulating the data structures using atomic instructions such as compare-and-swap (CAS) or load-linked/store-conditional (LL/SC). There are several popular lock-free versions of basic data structures, such as queues [26, 40], stacks [23], LRU caches [9], and priority queues [6]. They provide fast operations on these structures.

Regrettably, while lock-free synchronization provides fast sequential access to shared structures, many-thread synchronization can still lead to substantial contention—e.g., when all the threads attempt to perform a CAS on the head of a queue to update it atomically. For this reason, developers of highly-threaded codes today turn to algorithms that *distribute* synchronization [5, 22, 33, 34, 50, 62].

Unfortunately, these algorithms are difficult to design and debug. Even more importantly, they often provide only *weak* and unintuitive data structure semantics [5, 22, 34, 62]—e.g., items in a distributed queue are removed in an order that is different than their insertion order. Such weak semantics make these distributed algorithms more error-prone and difficult to use, and render them inappropriate for programs that require familiar semantics.

To reduce synchronization bottlenecks, past designs have proposed to queue-up the synchronizing processors in a hardware queue [20, 32, 43, 55, 56, 58, 68]. Other designs have also proposed to forward or prefetch [56, 65] the data accessed in the critical section when a lock is transferred. These proposals attempt to *efficiently serialize* concurrent synchronizations. However, they do not solve the serialization problem itself: serialized writes inherently become slower as the number of synchronizing processors grows. To make synchronization truly scalable, we need to break the serialization of the queue and allow the queued processors to synchronize *in parallel*.

This paper proposes a novel architecture design that *breaks* the serialization of hardware queues and enables the queued processors to perform lock-free synchronization *in*

*This work was supported in part by NSF under grants CCF-1012759, CNS-1116237, and CCF-1536795; by the Israel Science Foundation under grants 1227/10 and 1749/14; and by the Yad HaNadiv foundation.

ASPLOS '16, April 2–6, 2016, Atlanta, Georgia, USA.

ACM 978-1-4503-4091-5/16/04…$15.00.
http://dx.doi.org/10.1145/2872362.2872400

parallel. The scheme, called CASPAR, is applicable to some common lock-free synchronization patterns. The result is low-overhead, highly-scalable synchronization. With CASPAR, it is possible to have non-distributed scalable versions of lock-free data structures.

We present CASPAR in the context of the CAS instruction, although it also applies to other synchronization instructions such as LL/SC. Recall that a CAS takes three operands: a memory address *addr*, an *old* value, and a *new* value. A CAS changes the value in *addr* to *new*, provided the current value in *addr* is *old*. We observe that, in common synchronization patterns, the *new* value that a processor stores on a variable with a CAS does not depend on the *old* value previously read from the variable. Instead, the *new* value is generated locally by the processor. For example, as we will see, this occurs when pushing nodes into a stack.

This observation motivates CASPAR, which exploits it with two new ideas: (i) parallel execution of the CASes in the queued-up processors through eager forwarding of the expected *new* values, and (ii) parallel validation of the CASes and group dequeue of the processors.

The first idea uses the fact that a queued processor may know early-on the *new* value that it will set the shared variable to. Hence, it eagerly forwards it to its immediate successor in the queue, so that the successor processor can use it as its *old* value. As a result, queued processors can perform their CASes early-on and in parallel, and continue execution past the CAS. However, since the value passed is a hint, the successors' execution becomes speculative.

The second idea involves the group validation of CASes and, therefore, the group commit and dequeue of processors. It leverages the fact that the directory knows the values passed between processors and can interrogate a chain of dependent processors in parallel. If all processors validate, a two-phase commit operation dequeues them all in one step.

CASPAR effectiveness. We evaluate CASPAR by augmenting a simulated 64-core multicore with a synchronization hardware queue (representing prior work) and then extending it with the complete CASPAR design. We run five kernels and several applications—including some from the Galois system [50]. The applications are modified to use lock-free data structures. Compared to the design with only the hardware queue, CASPAR: (i) improves the throughput of the kernels by 32% on average, and (ii) reduces the execution time of the sections considered in the lock-free applications by 47% on average. This makes such sections 2.5x faster than in the original, tuned versions. Also, compared to a design with only conventional CAS synchronization, CASPAR improves the kernel throughput by 83% and reduces the execution time of the application sections by 58% on average.

Contributions. This paper makes three contributions:

- The CASPAR architecture, which provides scalable and efficient lock-free synchronization by parallelizing the operation of hardware-queued processors.
- A design for automatically triggering hardware queueing on the unmarked loads of contended CAS variables.
- Simulation-based evaluation of CASPAR using highly-threaded kernels and applications.

2. Motivation

Lock-free synchronization (also referred to as *nonblocking* synchronization [24]) directly manipulates shared data using atomic instructions such as CAS and LL/SC, or transactional memory (TM) instead of using locks (see Section 3 for an example). Performance-critical multi-threaded codes such as operating systems [2, 13, 35], databases [4, 15, 63], language runtimes [19], memory allocators [47, 59, 61], or trading frameworks [3] often utilize lock-free synchronization to avoid the overheads of locking.

Lock-free synchronization has two main performance advantages compared to lock-based solutions. First, it is more efficient, since it has no lock acquire and release operations on the critical path. Indeed, lock-free versions of many basic data structures have fast synchronization operations that write to at most a few variables—queues [26, 36, 48], stacks [66], and priority queues [18, 44]. Second, lock-free algorithms guarantee system-wide progress, and thus eliminate problems such as deadlocks and preemption of the holder of a lock [24].

Performance issues. Lock-free synchronization still has two performance issues when many processors contend to perform a write to the same synchronization variable. First, sometimes CAS or LL/SC atomic instructions fail, and transactions abort. This can place useless work on the critical path. Second, contending writes are *serialized*, as each processor must obtain exclusive access to the contended variable's cache line to update it. This serialization causes the latency of each write to grow with the amount of concurrency, making contending writes non-scalable and slow.

Software approaches. To avoid these issues, developers turn to algorithms that distribute synchronization [5, 22, 33, 34, 50, 62]. However, these algorithms are often complex and provide only weak and unintuitive data structure semantics [5, 22, 34, 62]. For example, consider a FIFO queue algorithm. Instead of synchronizing all operations on a single queue head, we distribute the synchronization. We maintain a queue per producer thread and require a remove operation to iterate over each of these queues until it finds an item to remove [22]. Doing so, however, no longer maintains the cause and effect relation in the program: If thread T_1 adds x_1 to its queue after thread T_0 adds x_0 to its queue, a remove can return x_1 before x_0. We *weakened* the guarantees provided by the queue from global FIFO to per-thread FIFO.

Hardware approaches. Alternatively, there are hardware proposals to make synchronization efficient (Section 8). The most relevant to our work are those that build hardware queues of processors synchronizing on a variable. Specif-

```
1  // Stack consists of linked list of nodes.
2  // The following defines a stack node:
3  struct Node {
4      struct Node* next;
5      void* value;
6  }
7  // Pointer to top of the stack, initially
8  // NULL as the stack is empty:
9  Node* stack = NULL;
```

(a) Definitions

```
10 void push(void* v) {
11     Node *old_top, *new_top = malloc();
12     new_top->value = v;
13     while (true) {
14         old_top = stack;
15         new_top->next = old_top;
16         if (CAS(&stack, old_top, new_top))
17             return;
18 }  }
```

(b) Pushing a value

```
19 void* pop() {
20     Node *old_top, *new_top;
21     while (true) {
22         old_top = stack;
23         if (old_top == NULL) return NULL;
24         new_top = old_top->next;
25         if (CAS(&stack, old_top, new_top))
26             return old_top->value;
27 }  }
```

(c) Popping a value

Figure 1: Treiber's lock-free LIFO stack [66].

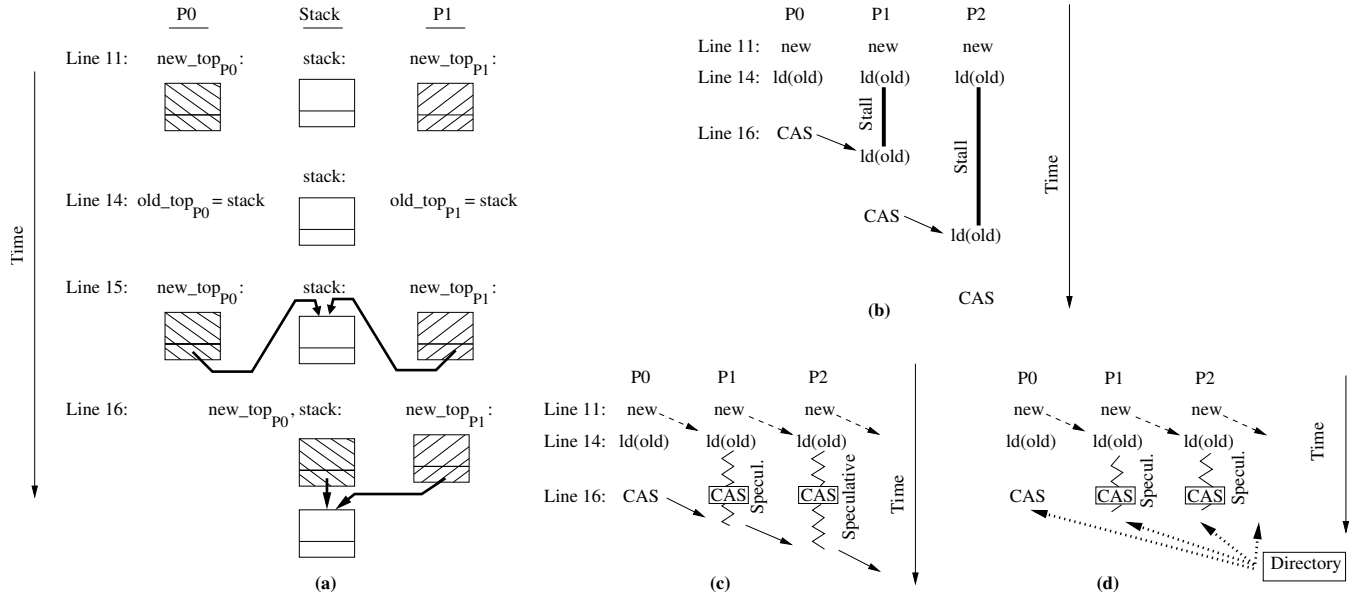


Figure 2: Ideas behind CASPAR.

ically, Goodman et al. [20, 32] proposed QOSB/QOLB, which forms a hardware queue linking the caches of the synchronizing processors. Hardware queues have also been implemented in the directory of DASH [43], and proposed by other researchers [55, 56, 58, 68]. In one of these designs [55], coherence actions are delayed to prevent repeated failure of lock acquire or atomic instructions. Specifically, a processor delays responding to requests on the synchronization variable while the processor is executing the critical section or the code between LL and SC. Finally, other designs proposed to forward or prefetch [56, 65] the data accessed in a critical section while the lock is transferred.

These proposals efficiently serialize concurrent synchronizations. However, they do not solve the serialization problem itself: executing the critical section or performing the lock-free write takes on average proportionally longer as the number of synchronizing processors grows.

Our approach and observation. To make synchronization truly scalable, we need hardware that *breaks* the queue serialization and allows the queued processors to synchronize *in parallel*. To design this hardware, we observe that, in some common synchronization patterns that we describe in the next section, the *new* value that a CAS writes to a variable does not depend on the variable's previous value. We exploit this fact next.

3. Overview of CASPAR

3.1 A LIFO Stack Example

Figure 1 shows the C code of Treiber's lock-free LIFO stack [66]. As shown in Figure 1(a), the stack is a linked list of nodes, each of which holds a value. The top of the stack is the first node in the list. The *push()* (Figure 1(b)) and *pop()* (Figure 1(c)) operations use a CAS.

We focus on the *push()* operation. It allocates a new node, *new_top* (Line 11), whose *next* field is set to point to the current top of the stack, *old_top* (Lines 14–15). It then uses a CAS to set the top of the stack to this *new_top* (Line 16). A CAS failure implies that another thread has modified the top of the stack, and so the *push()* operation retries the CAS using a freshly-read value of *stack*.

Figure 2(a) illustrates the above process. It shows the execution of two processors (P0 and P1) and the state of the stack. The top row corresponds to Line 11, where P0 allocates the new_top_{P0} node and P1 allocates the new_top_{P1} node. The second row is for Line 14, where the processors set their old_top_{P0} and old_top_{P1} to *stack*. The third row is

for Line 15, where the processors point $new\_top_{P0} \rightarrow next$ and $new\_top_{P1} \rightarrow next$ to *stack*. Finally, the last row is for Line 16, where both P0 and P1 attempt the CAS but only P0 succeeds. The new_top_{P0} node is inserted at the top of the stack, and P1 has to retry.

The ABA problem. An ABA problem [49] occurs when a thread reads the same value (e.g., *A*) from the *stack* location in Lines 14 and 16 and, in between the two reads, other threads update the location to a different value (e.g., *B*) and then back to *A*. The CAS in Line 16 succeeds, even though the atomicity of the load-to-CAS execution was violated. As in other works, in this paper, we assume implementations that avoid this problem by not recycling a node as long as some thread holds a reference to it [18, 30, 46].

3.2 CASPAR Ideas

In the *push()* operation, the CAS of the successful processor (P0) writes new_top_{P0} to *stack*. Such a value does not depend on the value that P0 did read from *stack* into *old_top*. Instead, it is obtained locally by P0, early-on, with a malloc(). Interestingly, this is the value that the failing processor (P1) will need to read into its own *old_top* when it wants to perform the next successful CAS. To summarize, each processor generates the *new* value for its CAS locally and early-on, and this is the value that its immediate successor will need to read as its *old* value to perform its own CAS. This provides an opportunity for parallelism.

Unfortunately, even the most aggressive proposals for hardware synchronization queues fail to take advantage of this opportunity. Indeed, assume a design based on any of the hardware queues described in Section 2, where requests are queued-up in hardware in the directory. Further, assume that, for highest efficiency, the load-to-CAS execution is designed to be atomic—i.e., a processor reads *stack*, prepares the new value, and then writes the new value with the CAS without any conflicting access allowed to interleave in the middle.

In this design, Figure 2(b) shows the execution timeline of three processors (P0, P1, and P2). Assume that all three generate their *new* values at approximately the same time, and that they attempt to load *old* and queue-up in the P0-P1-P2 order. P0 gets the *old* value, while the others stall (thick lines). Only after P0 completes its CAS can P1's load complete and return the current *old* value—even though it was available as the *new* value that P0 produced long ago. P2 suffers an even longer stall. The operation of the processors is completely serialized.

First idea: parallel CAS execution. CASPAR's first idea is parallel CAS execution, as shown in Figure 2(c). When a processor generates its *new* value locally and early-on, we propose that it *eagerly forwards* the *new* value right away to its immediate successor in the queue (dashed arrows). A processor uses the received value as the response to its load for the *old* value, and has all the information to perform the CAS. Hence, *the CASes are performed early-on and in parallel*. Since the values forwarded may be incorrect under certain conditions, execution past the load becomes speculative (zig-zag lines), and can only commit after a validation step in the background (solid arrows). Such a step would require waiting until the processor reaches the head of the queue, and then verifying that the actual content of the variable matches the value of the earlier hint. In cases when the CAS pattern is not amenable to eager forwarding, CASPAR reverts to the serialization of prior queue designs.

Second idea: parallel CAS validation. CASPAR's second idea is to validate groups of CASes in parallel and, therefore, commit groups of processors at a time. This technique reduces the amount of work done speculatively and, hence, reduces the risk of squashes. It is shown in Figure 2(d). The idea is based on the observation that the directory knows the value that each processor forwarded early-on to its successor. Hence, the directory can later interrogate a chain of queued processors in parallel (dotted lines), to see if the value that a processor's CAS ended-up generating is indeed equal to the value that the processor forwarded early-on to its successor. If this is true for a group of processors that begins with the one at the queue head, the group is committed and dequeued in one shot.

3.3 CASPAR Effectiveness

CASPAR is effective when queued processors can generate their *new* value early-on, independently of the *old* value that they read. This pattern appears in several cases. The most common one is when inserting elements into a shared data structure, as in the *push()* operation of Figure 1(b). This pattern also appears when using atomic swap instructions (like x86's XCHG), setting variables to fixed values, resetting variables (e.g., a counter), and detaching a list by swapping the head pointer with null.

Insertion-heavy scenarios occur in many workloads. For example, they arise in runtimes using a shared work queue for load-balancing task-based parallelism [51], when the task queue is populated, either initially or as part of a bulk-synchronous execution [50, 67]; in update-heavy OS data structures such as the reverse page map or pathname lookup cache [12]; in memory allocators when accessing the main heap [47, 59, 61]; in high-speed networking when enqueuing packets [13]; and in other cases.

We also believe that the CASPAR ideas apply more broadly than lock-free code based on CAS or LL/SC, and can be used to break the serialization in TM (Section 5). We defer exploration of this idea to future work.

CASPAR is not effective when the *new* value created by a queued processor depends on the *old* value that the processor reads. This pattern occurs most commonly when removing elements from a shared data structure. An example is the *pop()* operation of Figure 1(c), where the new value of the stack is obtained by reading the node currently at the top of the stack and accessing its *next* field. It also occurs when inserting elements to a structure using ABA-tagging [49]—i.e., devoting some bits in pointers for a counter that each

operation increments, to reduce the chance of an ABA problem. Incrementing such a counter creates a dependency. In all of these cases, CASPAR reverts to the serialization present in prior queue designs.

4. CASPAR Architecture

CASPAR is composed of three modules, which (1) identify contended CASes, (2) efficiently queue-up concurrent CASes operating on a location, and (3) enable parallel operation of the queued-up CASes (i.e., the process that we called *breaking the serialization*). Table 1 lists the modules and where they reside in the architecture. Since module (2) is reminiscent of previously-proposed designs of hardware queues for synchronization (e.g., [20, 32, 43, 56]), we do not consider it a main contribution of this work. Hence, we only outline it briefly.

Module Function	Location
(1): Identify contended CAS locations	Processor core
(2): Efficiently queue-up concurrent CASes	
Enforce load-to-CAS atomicity	Processor core
Queue requests in the directory	Directory module
(3): Parallel operation of queued-up CASes	
Parallel CAS execution with Eager Forwarding	Mostly core + directory
Parallel CAS validation using Group Commits	Mostly directory

Table 1: Components of CASPAR.

For ease of explanation, we divide module (2) into two parts: enforcing load-to-CAS atomicity and enqueuing requests in the directory. Module (3) is also composed of two parts: *Parallel CAS Execution* through eagerly forwarding, and *Parallel CAS Validation* using group commits. The following discussion assumes a generic chip multiprocessor (CMP) with a distributed directory for coherence.

4.1 Identifying Contended CAS Locations

4.1.1 Intuitive Idea

CASPAR dynamically identifies contended CAS locations in hardware, without the need to modify the executable. To understand how it works, consider a CAS such as the one in Line 16 of Figure 1(b). When it has failed a few times in a row, the CASPAR hardware saves the address it contends on (i.e., *stack*) in a table. In addition, every load issued by the processor is dynamically checked against the entries in that table. When a load hits (such as the one in Line 14 of Figure 1(b)), the load becomes a *Triggering Load (TL)*, which exercises the CASPAR hardware. The CASPAR hardware remains active until the corresponding CAS completes, at which point the hardware actions typically complete. In a processor, only a single load at a time can be exercising the CASPAR hardware.

4.1.2 Detailed Design

The two per-processor hardware structures used in this process are shown in Figures 3(a)-(b). One is the Triggering Addresses Table (TAT), which has the addresses identified as "under CAS contention" by this processor. It is a 4-8 entry fully-associative table. Its entries are regularly aged out. The second structure is the Active CAS (AC), which can only contain one of the addresses from the TAT: the one currently exercising the CASPAR hardware.

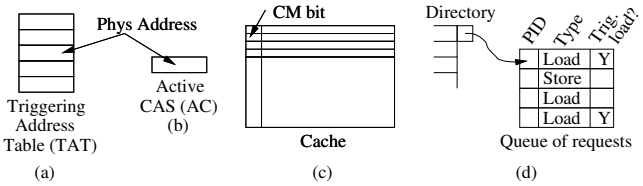


Figure 3: Basic hardware structures in CASPAR.

The address read by each load is compared to the addresses in the TAT. If a load hits and the AC is currently null, then the address read from is stored in the AC and the load becomes a TL. Normally, the AC will retain its value until the corresponding CAS completes. Then, the AC is cleared. If a second load hits in the TAT while the AC is full, that load executes as an plain load (i.e., it is not a TL).

CASPAR does not always need multiple CAS failures to insert an address in the TAT. We will see that a CAS that has encountered a queue in the directory returns a hint that can be used to insert the address accessed in the TAT.

4.2 Efficiently Queueing-up Concurrent CASes

4.2.1 Enforcing Load-to-CAS Atomicity

To support hardware queueing of concurrent CASes directed to the same address, CASPAR enforces Load-to-CAS atomicity. This is shown in Figure 4. A TL requests the memory line in Exclusive state. When the line arrives at the cache, the hardware sets a CAS Mode (CM) bit in the line's cache tag (Figure 3(c)). The cache is now in *CAS Mode*, and rejects any incoming coherence requests for the line. The cache remains in CAS Mode until the corresponding CAS completes. If no CAS to the line executes within a timeout period (e.g., due to a bug), the CAS Mode expires. If an exception occurs, CM is cleared.

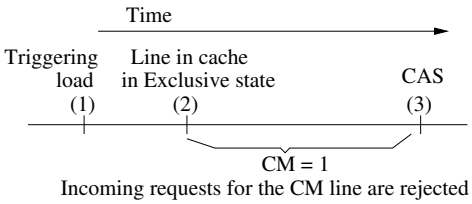


Figure 4: Timeline to enforce Load-to-CAS atomicity.

It is possible that two or more processors end-up waiting on each other—e.g., if two processors execute load-to-CASes to different addresses, but, in between the load and the CAS, they attempt to access the address of the other's CAS location (or false-share it). The timeout mechanism avoids deadlock. A similar timeout mechanism has been proposed for LL/SC [55].

4.2.2 Queueing Requests in the Directory

All the concurrent requesters to a given CAS location are queued-up in hardware by CASPAR in a directory module. As shown in Figure 5, when a TL request from processor P_i arrives at the directory, it is placed at the tail of the hardware queue. The directory continuously issues requests to the current owner of the line on behalf of the processor at the head of the queue. Such requests keep being rejected while the owner has its CM bit set in its cache line tag. When the owner clears the CM bit, the directory succeeds at stealing the line in Exclusive state and sends it to the processor at the head of the queue—as a response to its initial TL. Immediately after that, the directory dequeues the head entry from the directory queue and starts requesting the line from the new owner on behalf of the new entry at the head of the queue.

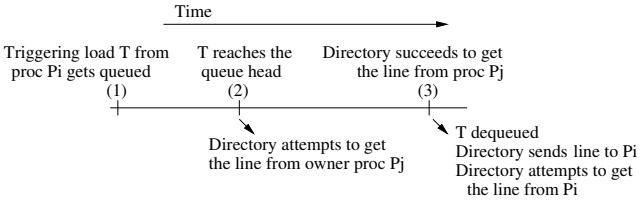


Figure 5: Timeline to queue/dequeue requests.

Figure 3(d) shows the hardware queue. Logically, the directory entry for the contended CAS location has a pointer to the queue. Each queue entry records the requesting processor's ID, the type of request (request to read or to write), and if the read is a TL. A directory module can have multiple queues for different addresses.

A queue entry of Store type is typically due to a CAS from a processor that is unaware that this location suffers contention. Hence, as indicated before, when the directory processes such an entry, it augments its reply to the requesting processor with a hint bit. If the access was indeed a CAS, this bit prompts the requesting processor to save the address in its TAT (if it is not already there).

4.3 Parallel CAS Execution Through Eager Forwards

Prior proposals that build a synchronization queue in hardware process the queue elements sequentially. CASPAR processes them *in parallel* thanks to two ideas: (1) parallel CAS execution through Eager Forwarding and (2) parallel CAS validation using Group Commits. This section describes the first idea; the next one the other.

4.3.1 Intuitive Idea

CASPAR targets codes where the *new* value to store in the CASed location does not depend on the *old* value in that location. With CASPAR, a processor P_i, before obtaining the line with the *old* value from memory, can *forward* its *new* value to the directory; the directory can send it to the successor processor P_{i+1} in the queue. P_{i+1} receives the value as a response to its TL request for its *old* value, and uses this highly-accurate hint of *old* in its CAS. Note that the forwarded value is coupled with its offset in the cache line, so P_{i+1} uses it only if it matches its TL address. With this scheme, both processors can execute the CAS *in parallel*.

Later, when P_{i+1} reaches the head of the queue, the coherence protocol supplies the line to P_{i+1}, as the true response to P_{i+1}'s TL. On reception of the line, the hardware in P_{i+1} compares the value in the line to the *old* value that was received (and used) earlier on. If the validation succeeds, the hardware merges the *new* value produced by P_{i+1} into the line, and allows the protocol to transfer the line to the next processor in the queue.

Typically, the *old* value received from the predecessor will be correct. However, there are events such as branch mispredictions in the predecessor that may cause divergence between the value forwarded and the line received later. In this case, the *old* value used was incorrect and execution needs to be squashed and restarted from the TL issue. To support this, in a TL, the processor performs a checkpoint and enters speculative (i.e., transactional) execution.

CASPAR speeds-up execution because CASes are executed early and in parallel. For now, the validation step that allows processors to exit speculation is serialized. (We relax this property in Section 4.4.) However, processors can continue speculative execution past the CAS, until the *old* value is validated. Then, they commit all the work performed since the TL issue.

A value is forwarded by writing it back to the directory in a fine-grain writeback-like transaction. The directory stores the value in the queue entry of the sender processor and, if there is a successor processor, passes the value to the successor. Note that the successor receives the value as a speculative response to its TL. This means that the successor is expecting a value, which makes our approach different than classical unsolicited forwarding (e.g., [37, 43]).

In practice, the directory does not pass the value to the successor unconditionally. It tries to avoid passing a value to a processor whose *new* value depends on the *old* value. To see why, consider the *pop()* operation in Figure 1(c), where the *new* value depends on the *old* one (Line 24). If a processor executing *pop()* receives a forwarded value, it will dereference it in Line 24, attempting to read data written by the sender of the forward. This will lead to the squashing of the predecessor, if it is still executing speculatively.

Therefore, when the directory receives a forwarded value from P_i, it will only pass it to P_{i+1} if and when P_{i+1} also sends a forwarded value to the directory. The latter forward is a hint that P_{i+1} will not squash the predecessor because its *new* value does not depend on the *old* one. Intuitively, in a queue with push and pop requests, forwarding will only occur between consecutive pushes. Alternatively, we could implement CASPAR on top of a TM design that allows some coherence operations between executing transactions, such as OmniOrder [54]. This would allow a processor to deref-

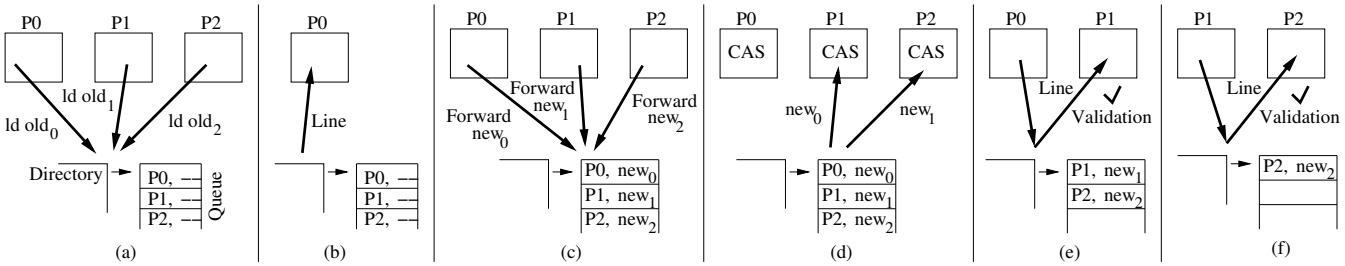


Figure 6: Operation of Eager Forwarding.

erence the received value without squashing the predecessor. We defer this extension to future work.

Figure 6 shows the operation of CASPAR for three processors. In Figure 6(a), all processors issue TLs, and get queued as P_0 first, then P_1, and then P_2. The queue contains no *new* values. In Figure 6(b), the directory provides the line to P_0, which is at the head of the queue. Soon after the TLs reach the directory, the forwarded *new* values also arrive at the directory, and are stored in the queue (Figure 6(c)). The directory immediately sends the *new* values to the successor processors, and all processors now have the data they need to perform the CAS in parallel (Figure 6(d)). As described before, the directory attempts to get the line from P_0. When it succeeds (Figure 6(e)), it pops the first entry from the queue and replies to the next entry's TL by sending the line to P_1. P_1 validates the speculative execution and commits it. The directory repeats the same process for P_2, which is the new head of the queue (Figure 6(f)).

4.3.2 Architectural Components Required

Eager forwarding requires architectural components to: (1) transfer the *new* value to the successor, (2) accept an early *old* value from the predecessor and later validate it, and (3) support speculative execution from the TL until the execution is validated. We describe each one in turn and outline the hardware structures.

A. Transfer the new value. In a conventional pipeline, a CAS instruction performs the read-modify-write of a cache line when it is at the head of the Reorder Buffer (ROB). With CASPAR, a CAS instruction whose address hits in the Active CAS (AC) (Figure 3(b)) has a two-step execution. First, as soon as its *new* value is known, it forwards *new* to the directory. Second, when it reaches the ROB head and we know its *new* value and its *old* value (perhaps speculatively), it performs the read-modify-write as in a conventional system.

Since forwarding the *new* value is on the critical path of the parallel execution, it is performed as soon as *new* is known, bypassing all the other loads and stores by the processor. This is safe because *new* is observed only by the next processor in the directory queue (as it is deposited in the queue rather than in memory), where it is used only as a hint that is validated upon committing (i.e., as a value prediction). As detailed in Section 4.3.4, standard speculative execution conflict checks guarantee that using a forwarded value does not cause memory consistency errors.

B. Old value use and validation. After a processor issues a TL to memory, it may receive a speculative *old* value from the directory. Such a value is stored in the AC structure, and cannot be used until the TL has reached the ROB head and checkpointed. At that point, execution turns speculative, and the thread can use the received *old* value. In particular, the CAS operation may use it, and store the speculative CAS result in the cache.

Eventually, the processor will receive from the directory the line requested by the TL. Then, the hardware compares the value in the incoming line to the speculative *old* value. If the values are different, execution is rolled back to the checkpoint and the value in the line is used rather than the speculative *old* value. Otherwise, the work done so far is useful and correct, and is committed. Note that this value-based validation does not introduce an ABA problem [49]. Since the directory manages the hardware queue, a processor can only see updates from its immediate predecessor; no other processor's updates can interleave between the two.

C. Speculative execution from TL to validation. When a TL reaches the ROB head and the requested line is not in Exclusive (or Dirty) state in the cache, the hardware performs a checkpoint and the processor enters speculative execution. If, instead, the line is already in one of these states, there is no need to become speculative because the CAS will execute with safe data very soon.

As in conventional TM, during speculative execution, data conflicts with incoming coherence transactions cause an abort. If speculative data is about to overflow the cache, the execution can stall rather than abort. There is no danger of deadlock because there is always a non-speculative thread—the one at the head of the queue.

At some point during speculative execution, the requested line is provided by the memory system. If the processor had used a speculative *old* value, then the hardware performs the above validation step, and the transaction commits or aborts. This may occur past the CAS execution.

D. Hardware structures. Figure 7(a) shows the two main hardware structure extensions required for eager forwarding. First, in the processor, the Active CAS (AC) is extended to include the speculative *old* value received (SpecOld)

and a set of bits (Trans?, LineArrived?, CASDone? and NewSent?). The speculative *old* value is kept in the AC to compare it to the line's value in the validation step. The bit fields are used by a state machine to track execution states.

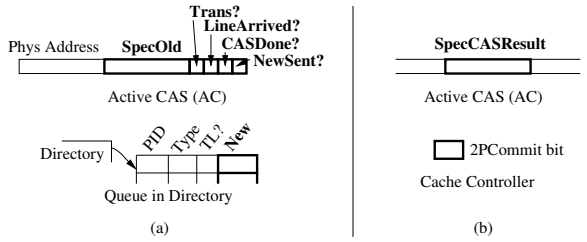


Figure 7: Additional hardware for eager forwarding (a) and parallel CAS validation (b).

In the directory queue, we add one extra field per entry. For the entry of a given processor, the field contains the *new* value forwarded by the processor to the directory.

4.3.3 Timeline

Figure 8 shows a typical timeline of eager forwarding. The events are shown above the horizontal line, while the actions are shown below the line.

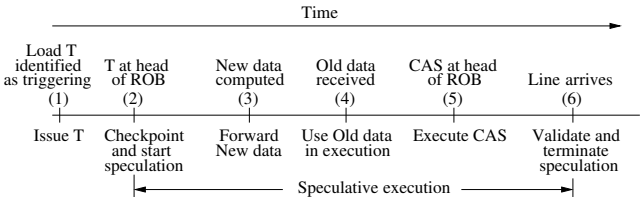


Figure 8: Timeline to perform eager forwarding.

The first event occurs when a load is identified as TL (Section 4.1). Event (2) is when the TL reaches the ROB head. Unless the line is already in Exclusive (or Dirty) state in the cache, the hardware checkpoints and starts speculative execution. Event (3) occurs when a CAS instruction in the pipeline finds that its address matches the one in the AC, and that its register operand with the *new* value to write is already full. In this case, the hardware forwards the *new* value to the directory. Typically, the CAS is not at the ROB head.

When an *old* value is received for a line with the CM bit set, it is saved in the SpecOld field of the AC for later validation. If the processor is in speculative mode, then Event (4) occurs, and the received *old* value is used in the execution.

Event (5) occurs when the CAS instruction reaches the ROB head, the register operands with its *new* value and its *old* (possibly speculative) value are full, and all prior accesses have completed. The CAS then executes, either speculatively (reading from SpecOld) or not (reading from the cache). If the CAS succeeds, the cache is updated. Irrespective of whether the CAS succeeds, execution continues (possibly speculatively).

Finally, Event (6) occurs when the requested line finally arrives for a cache entry marked with the CM bit. The hardware validates the SpecOld field of the AC against the line. If the validation fails, the processor rolls back to the checkpoint; otherwise the speculative execution commits. In all cases, the AC and the CM bit get cleared.

These events may be ordered in slightly different manners. In all cases, it can be shown that the algorithm works.

A processor may forward a *new* value to the directory twice. This may occur in a branch misprediction where *new* is forwarded on both sides of the branch. The directory only takes the first forwarded value. This case may cause the successor processor to fail the validation. However, correctness is guaranteed.

4.3.4 Memory Consistency Issues in Forwarding Data

In CASPAR, the *new* value of a processor (P_i) finds its way to the immediate successor in the queue (P_{i+1}) before P_i performs the CAS. In addition, *new* can bypass all the other outgoing accesses from P_i. This operation causes no memory consistency errors for the following reasons.

First, the forwarded value does not update memory; it is saved in the directory queue and sent to P_{i+1}. Second, if *new*'s value is wrong, the worst that can happen is that P_{i+1} executed past the CAS, gets squashed when its validation fails, and restarts from the TL using the correct value.

Since *new* can bypass earlier accesses in P_i's outgoing buffers, P_{i+1} may observe *new* before it observes other P_i accesses that precede the corresponding CAS in P_i's program order. Figure 9 shows an example, where P_i performs a TL and CAS on location *Stack*. In between the two, the *new* value of *Stack* is forwarded before P_i updates variable *X*. It is possible that P_{i+1} reads the forwarded value of *Stack* and then X. Hence, it observes the new value of *Stack* and the old value of *X*. This will not cause a consistency violation because P_{i+1} turns speculative when it reads *Stack*. Hence, when P_i writes X, it will send an invalidation to P_{i+1} and squash P_{i+1}'s execution.

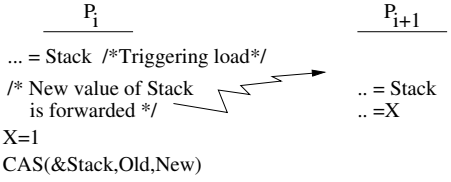


Figure 9: Forwarding causes no consistency violation.

In fact, P_{i+1}'s transaction can commit only if it uses the correct state. The transaction cannot commit until P_i performs the CAS, and P_{i+1} receives the line and validates the *new* value that it used. By this time, all of P_i's accesses that precede the CAS have been completed, including all writes. Such writes would have squashed P_{i+1}'s transaction if it had read an incorrect value. Moreover, P_i cannot read any state generated by P_{i+1} before *new* is validated because P_{i+1}'s execution is speculative.

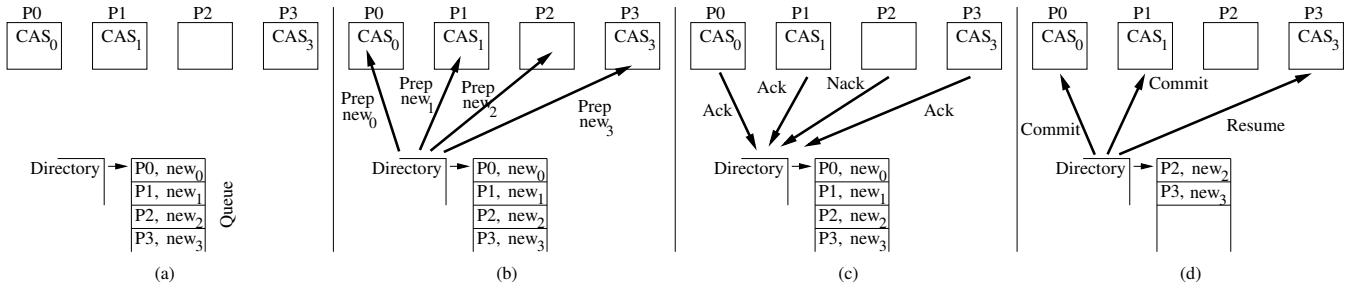


Figure 10: Parallel CAS validation.

4.4 Parallel CAS Validation Using Group Commits

With eager forwarding, the queued processors still perform the CAS validation step sequentially (Figures 6(e)-(f)). In theory, this should not hurt performance because processors do not stall after a CAS waiting for validation: they continue executing speculatively. In practice, however, it is desirable to validate sooner for two reasons. The first one is to reduce the time a processor remains speculative and, hence, exposed to aborts. The second reason is to avoid stalls in codes where a processor repeatedly executes CASes to the same address—a common pattern in codes with fine-grain synchronization.

Indeed, assume that a processor has issued a TL and completed a CAS, but has not received the line yet. It remains speculative. Suppose that it then executes the load for a second load-to-CAS to the same address. Since the AC is still full, the load is not designated a TL but a plain load. The load goes to the cache and stalls, waiting for the line requested by the TL. The pipeline will likely stall soon after. In essence, the processor has overlapped as much speculative execution as it could with the TL, and it now stalls. It will not resume execution until the processor receives the line.

If we have a long queue, it will take on average a long time for the line to reach a processor. As a result, if a processor repeatedly executes CASes to the same address, it will likely stall for long periods.

To solve these two problems, we propose to accelerate the CAS validation by performing it *in parallel*.

4.4.1 Intuitive Idea

We augment the design of Section 4.3 to validate the CASes in groups of queued processors at a time. The idea is to augment the protocol so that the directory orchestrates group-commits in a manner modeled after the *two-phase commit* (2PC) in transactional processing.

Consider a queue of processors, where many have forwarded their *new* values to the directory—which in turn has saved the values and sent them to their successors. When the directory finally obtains the line, rather than sending it to the next processor in the queue for validation, it attempts to *group-validate* a group of processors. To do this, as it dequeues the processor that supplied the line, it checks that the value that the processor forwarded in the past matches the current value in the line. If so, it proceeds to group-validate the next set of contiguous processors in the hardware queue that have provided their *new* values. It does so in three steps.

First, it sends a *Prep* message (for "prepare-to-commit") to all of these processors in parallel. In each message (say to processor P_i), the directory includes the *new* value that the processor had earlier forwarded to the directory (new_i). The goal is for P_i to validate it against the outcome of P_i's CAS. Recall that new_i has already been used by the successor processor P_{i+1}.

Second, when P_i receives the *Prep* message, it compares *Prep*'s value (new_i) to the result of its CAS. If the values match, the validation succeeds. P_i then responds to the directory with an *Ack* message and temporarily sets the pipeline in a quiescent, stalled state to enable a correct 2PC (see Section 4.4.2). If, instead, the values do not match, or P_i has not performed its CAS yet, or P_i has other pending accesses waiting for the same cache line, then P_i responds to the directory with a *Nack* message and does not stop execution.

Third, after the directory gets all the responses, it identifies the set of contiguous processors (starting from the head of the queue) that responded with *Ack*. To these, it sends a *Commit* message and removes them from the queue; they commit their speculative execution and then resume. To the others that sent *Acks*, the directory sends a *Resume* message; they resume executing speculatively. Finally, to the ones that sent *Nacks*, it does not respond.

If the first processor in the queue sent a *Nack*, the group commit fails. We will see that this processor then falls back to the sequential CAS validation of Section 4.3.

Figure 10 shows an example of parallel CAS validation. In Figure 10(a), four processors are queued up and have forwarded their *new* values to the directory (which has passed them to their successors). In addition, P_0, P_1, and P_3 have completed their CAS. Assume also that the directory/memory has attained the line—hence, the line is not in any cache.

In Figure 10(b), the directory initiates a group commit. It sends a *Prep* to the four processors, together with the corresponding *new* values that the processors had forwarded. On reception, processors P_0, P_1, and P_3 compare the *new* value to the outcome of their CAS. Assume that the values match. Hence, as shown in Figure 10(c), they send *Acks* to the directory. P_2 has not completed its CAS yet. Hence it sends a

Nack. In Figure 10(d), the directory finds that, starting from P_0, the set of contiguous processors that responded with *Ack* are P_0 and P_1. So, these two can commit in a group. The directory sends a *Commit* to them and removes their entries from the queue. It sends a *Resume* to P_3.

4.4.2 Architectural Components Required

Parallel CAS validation requires components to: (1) quiesce a processor pipeline in a two-phase commit, (2) commit a load-to-CAS section without the processor ever obtaining the memory line with the CAS data, and (3) seamlessly revert to sequential CAS validation if group validation fails.

A. Quiesce a pipeline in a two-phase commit. We extend the Active CAS (AC) structure to save the result of a speculative CAS execution. Then, if the processor receives a *Prep*, it compares the message's *new* value to the result of the CAS. If they are the same, the processor prepares for a two-phase commit (2PC).

For correctness, the 2PC requires that the processor be able to commit if it is instructed to do so. Thus, its speculative execution must never get squashed after sending an *Ack*. Quiescing the pipeline achieves this: The processor stops issuing new instructions and flushes the pipeline, discarding all the unretired instructions. It also sets a new bit in the cache controller called *2PCommit*. This bit will reject all incoming coherence requests that could cause a squash of the thread—i.e., incoming reads to speculatively written lines, and incoming writes to speculatively accessed lines. Finally, when the write buffer is drained, the processor disables interrupts (like in the x86 CLI instruction), bringing the pipeline to the quiescent state.

Once in quiescent state, the processor sends the *Ack* to the directory. It remains quiescent until the arrival of a *Commit* or *Resume*. Then (after committing the thread if the message was a *Commit*) the *2PCommit* bit is cleared to accept all coherence requests. Interrupts are re-enabled (like in x86's STI instruction) and the processor re-starts issuing instructions.

It is possible that a pipeline cannot get into a quiescent state because the writes in its write buffer end up getting rejected by another processor with its *2PCommit* bit set. This case is detected because the response to the rejected writes indicates that the destination processor does not accept requests. In this case, the processor refuses to participate in the 2PC: it sends a *Nack* to the directory and continues executing. Its CAS will be validated later, either in a group or sequentially with the default algorithm.

B. Commit without ever getting the cache line. A successful parallel CAS validation is fast because the memory line with the CAS data does not need to be transferred between the caches of the processors involved. Instead, these processors commit their load-to-CAS code without ever obtaining the line in their caches.

To see how it works, consider a processor that is executing a load-to-CAS section. The TL caused a cache miss, which triggered the allocation of an MSHR entry and of space for a line in the cache. When the *old* value is received from the predecessor, it is stored in the MSHR and used speculatively. Later, the CAS is performed speculatively and its result is stored in the AC. Suppose that a *Prep* now arrives and its value matches the CAS value in the AC. If and when the *Commit* is eventually received, the hardware simply commits the execution. In addition, it discards the MSHR entry and frees-up the empty cache line.

C. Seamlessly revert to sequential CAS validation. Whenever a group commit fails, our algorithm performs a sequential CAS validation like the algorithm of Section 4.3. Specifically, a failure occurs when the first processor in the queue (P_0) responds to the directory's *Prep* with a *Nack*. This may be because either P_0 has not performed its CAS yet or P_0's CAS fails the validation—i.e., the CAS produces a value different than the one P_0 forwarded to the directory (new_0). In either case, in CASPAR, the directory sends the memory line to P_0, which performs a local CAS validation as described in Section 4.3. The directory also sends *Resumes* to processors that sent *Acks*. As usual, the directory will then try to obtain the cache line from P_0. Once it gets it, it attempts the next parallel CAS validation.

D. Hardware structures. Figure 7(b) shows the two main hardware structure extensions required. First, in the processor, the Active CAS (AC) is extended to include the result of the speculative CAS operation (*SpecCASResult*). Second, in the cache controller, we have the *2PCommit* bit, which rejects incoming coherence requests that could cause a squash of the thread during the two-phase commit.

5. Supporting Other Primitives

CASPAR applies straightforwardly to other atomic instructions, such as LL/SC [29]. LL/SC has an explicit triggering load (i.e., the LL), and so is simple to serialize with hardware queueing. We can then apply parallel execution and validation, which are agnostic to the atomic instruction used.

We believe CASPAR can also be extended to TM, which is increasingly being used for lock-free synchronization [16]. In typical TM designs, programmer-defined transactions that execute read-modify-write (RMW) access sequences to shared variables (e.g., a queue head) will abort on conflict and be serialized. Such transactions can benefit from the CASPAR ideas. Specifically, aborts can be used as the signal to identify the contended variables, similarly to how TLs are identified. Once a RMW sequence of accesses is identified, the written value can be eagerly forwarded to another transaction, allowing the concurrent execution of multiple transactions. Compared to the current CASPAR design, supporting TM presents new challenges, such as the possibility of multiple RMW sequences in a transaction. We defer exploration of this CASPAR extension to future work.

6. Evaluation Environment

We evaluate CASPAR with simulations of a 64-core chip using the Sniper simulator [14]. Table 2 shows the baseline

architecture modeled. The core and L1/L2 cache parameters are taken from Nehalem [64]. We implement three designs which incrementally build on top of this baseline (B). They are Queue (Q), EagerForwarding (EF) and CASPAR (C). *Queue* implements basic hardware queueing in each module of the distributed L3 tag directory, similar to past proposals, as per Section 4.2. *EagerForwarding* adds parallel CAS execution with eager forwarding as per Section 4.3. CASPAR further adds support for parallel CAS validation using group commits as per Section 4.4, and is our complete design.

Parameter	Value
Architecture	64 cores on chip
Core	2.66 GHz, 4-wide out-of-order
ROB, Res. Stations	128 entries, 36 entries (unified)
Private L1	32KB WB, 8-way, 4 cycles round trip
Private L2	256KB WB, 8-way, 9 cycles round trip
Shared, NUCA L3	16MB WB, 16-way, 12 cycles (near access)
Cache line size	64B
Coherence	MESI, full-mapped tag directory
Network	2-D torus, 2-cycle hop latency, 64 bits/cycle link
Main memory	120 cycles round trip
Entering quiescence	$\approx$ Time to drain write buffer

Table 2: Architecture simulated.

We use two sets of programs for our evaluation (Table 3): five kernels and four applications. The kernels consist of four computational kernels (*FIFO*, *LPO*, *MBrot*, and *LIFO*) and one standard memory allocation kernel (*Larson*). In the computational kernels, each thread executes a loop where, in each iteration, the thread performs some computation and then synchronizes with a lock-free operation. The memory allocation kernel runs Michael's memory allocator [47], which internally uses lock-free algorithms.

Program	Description
Kernels:	
FIFO	Add/remove from Michael and Scott's lock-free queue [48].
LIFO-push-only (LPO)	Push into a lock-free stack, modeling bulk synchronization [67] or initial population of a work list.
MBrot	Mandelbrot set computation. Computing threads pass results to rendering thread via a multi-producer/single-consumer queue.
Larson	Threads allocate/deallocate objects, while transferring some objects to be freed by other threads [41].
LIFO	Push/Pop into a Treiber's lock-free stack [66].
Applications:	
FFT	1D FFT of a vector of complex values from BOTS.
CC	Connected components computation based on a concurrent union-find algorithm from Galois. Input is USA road network.
IS	Maximal independent set computation from Galois. Input is USA road network.
DT	Delaunay triangulation from a given a set of points from Galois. Input is 5 million 2D points.

Table 3: Programs evaluated.

The applications are FFT from the Barcelona OpenMP Tasks Suite (BOTS) [17] and three graph analytics programs from the Galois system [39, 50]. BOTS includes several task parallel applications from various domains; we evaluate FFT, which uses fine-grained tasks and stresses the task scheduler. We run FFT using the open-source Qthreads parallel runtime [51, 69], which supports multiple scheduling options. We compare schedulers that use lock-free LIFO [66] and FIFO [48] queues to the default lock-based scheduler. Galois [39] provides a domain-specific language and runtime for graph algorithms. The runtime parallelizes graph analytics loops using a work list data structure where threads add/remove work. We evaluate three Galois programs where work list synchronization accounts for a sizable fraction of the execution time. We add lock-free LIFO and FIFO work lists to the Galois 2.2.1 runtime [1], and compare the execution time to the default lock-based work list.

7. Evaluation

7.1 Kernels

Since the threads in the kernels repeatedly perform work and then synchronize using a lock-free algorithm, we use *CAS throughput*—the number of successful CAS operations per unit time—as the performance metric. We measure throughput over 5 ms (13.3 million cycles) of kernel execution time. Figure 11 shows the results, normalized to the throughput of the baseline (B) design. On average, EF and C improve the CAS throughput by 53% and 83%, respectively, over the baseline multicore (B), and by 10% and 32%, respectively, over hardware queues only (Q).

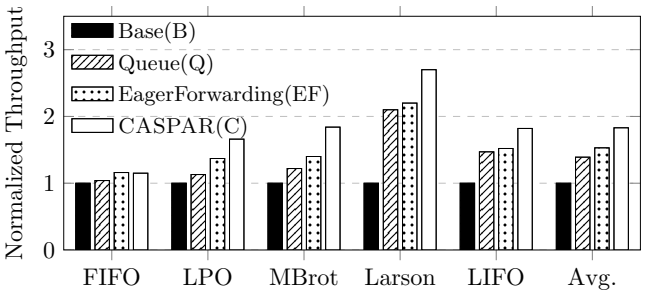


Figure 11: Kernel throughput for the different designs.

The gains vary depending on the kernel characteristics. Hardware queueing (Q) provides benefits in most of the kernels. The benefits are especially large in *Larson* and *LIFO*, where B's CASes frequently fail. Q eliminates CAS failures by enforcing load-to-CAS atomicity.

EF provides additional throughput boost for most kernels. The improvements are largest in the kernels with mostly enqueue operations, namely *LPO* and *MBrot*. The other kernels have both enqueue and dequeue operations; the latter have CAS dependencies as shown in Figure 1(c), which reduce the frequency of eager forwarding.

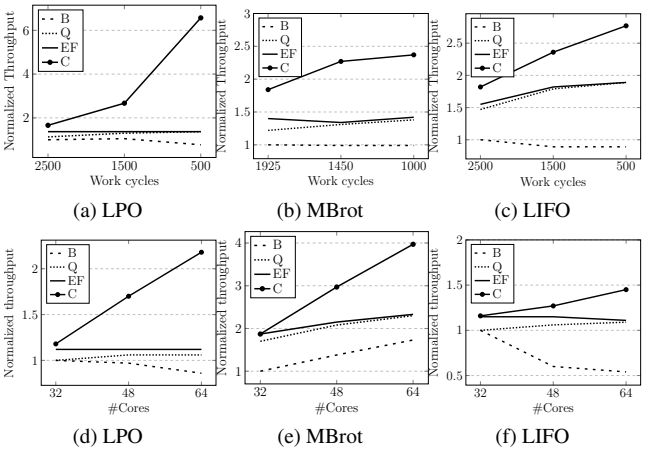
C improves over EF in all kernels except *FIFO*. To see why, recall that a processor in EF can execute speculatively past a CAS but stalls upon attempting to execute another load to the location that was CASed until the pending cache line arrives (Section 4.4). The wait time for the cache line is proportional to the length of the queue in the directory. On average, the queue size increases by 2.2x from Q to EF, since speculative execution increases the rate at which a

[1] The code is available at `http://git.io/galoisLF`.

core issues TLs. This results in stall cycles for kernels where the work between successive executions of the load-to-CAS section is too small to absorb the wait time for the cache line. This is the case for all kernels under EF except *FIFO*. On the other hand, C uses group commit to dequeue groups of processors at a time. With C, the average queue size is ≈33% of Q's. This reduces the stall and improves the throughput of C over EF in these kernels. In *FIFO*, EF had few stall cycles, and so C and EF perform comparably.

7.1.1 Impact of the Amount of Work

We now measure the change in throughput as we change the amount of work performed between synchronizations. We start with the amount of work in the experiments of Figure 11 and progressively reduce the work. Figures 12(a)-(c) show the throughput of *LPO*, *MBrot* and *LIFO* in each of the architectures. The plots are normalized to the B design for the amount of work in Figure 11.



(a) LPO (b) MBrot (c) LIFO

(d) LPO (e) MBrot (f) LIFO

Figure 12: Impact of work size (a-c) & scalability (d-f).

Reducing the work should increase CAS throughput, but it also increases CAS contention. In B, it increases the CAS failure rate. This results in a largely flat or decreasing throughput, which is the number of *successful* CASes per unit time. In Q, hardware queueing improves synchronization efficiency and eliminates CAS failures. Hence, decreasing work increases the CAS throughput in two of the three kernels.

In EF, the throughput is initially higher than in Q because processors perform part of the work speculatively. However, the throughput gap between the two narrows as the available work to speculate on decreases. EF is unable to exploit the reduction in work to improve throughput in two kernels—the processors eventually stall. Finally, C keeps increasing its throughput as the amount of work decreases. This is due to C's parallel validation, which eliminates EF's stalls.

To better compare EF and C, Figure 13 breaks down the normalized execution cycles in EF and C as they run *MBrot* and *FIFO*. We show two variants per kernel: high work and low work between CASes. The cycles are broken down into

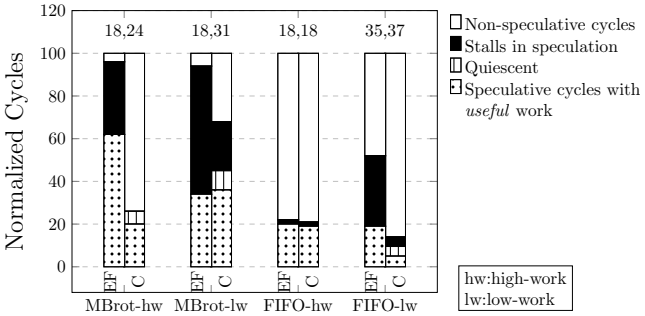


Figure 13: Cycle breakdown for EF and C. The numbers at the top of the bars are the CAS throughput.

non-speculative execution, processor stalls while speculative, quiescent pipeline (C only), and speculative execution of useful work. On top of the bars, we have the CAS throughput in successful CASes per 1,000 cycles.

Consider *MBrot* first. When the work is low, EF stalls frequently. C converts a good fraction of these cycles to non-speculative cycles using group commit. While C suffers from some quiescent pipeline cycles, the result is a large improvement in throughput. When the work is high, EF has fewer stall cycles. In this case, C converts both the remaining stall cycles and a portion of speculative cycles into non-speculative. However, the CAS throughput is not as high because improvements come only from the elimination of the stall cycles.

Consider *FIFO* now. When the work is low, EF has stall cycles and C eliminates most of them, increasing the throughput. When the work is high, however, EF has few stall or speculative cycles. The hardware queue is short and contains both push and pop requests. As a result, C is unable to perform much parallel CAS validation, and does not reduce either type of cycles. As a result, as shown in Figure 11, the EF and C throughputs are similar.

7.1.2 Scalability

We now measure the throughput as we change the number of processors for a fixed amount of work (i.e., the intermediate work amount from Figures 12(a)-(c)). This is shown in Figures 12(d)-(f), which are normalized to B with 32 cores. We note that *LPO* and *LIFO* have high CAS contention. Hence, they scale poorly in B because, with more cores, we have more CAS failures. Q and EF maintain performance for these kernels even at a high core count. C breaks the sequential validation in EF, and scales well. On the other hand, *MBrot* has low CAS contention. Hence, B's throughput improves with additional cores due to parallelization. Q and EF scale better, and C scales linearly.

7.2 Applications

Figure 14 compares the execution time of the applications (both LIFO and FIFO variants) on different architectures. The B bar is now replaced by two bars: L is the *original* lock-based version, and LF is the *lock-free* one. Q, EF, and C use

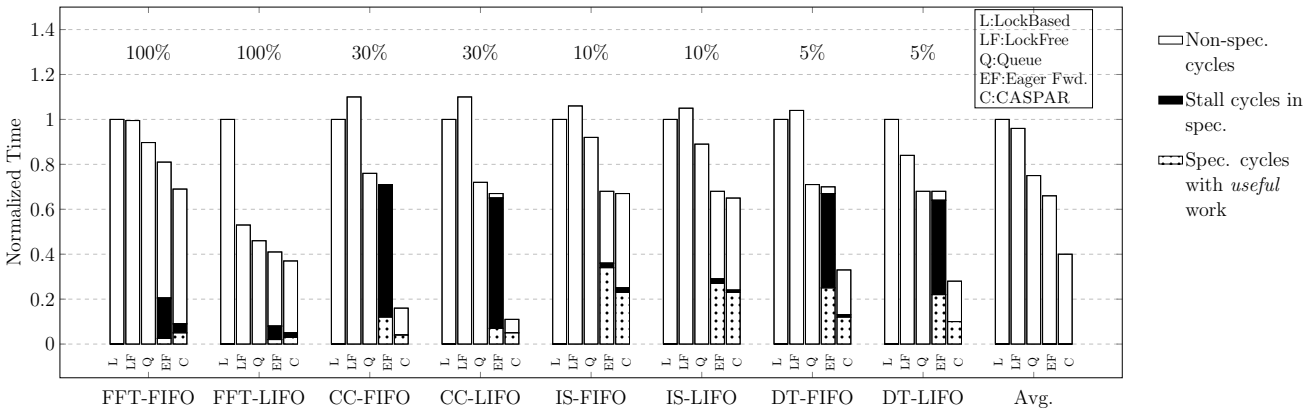


Figure 14: Execution time of the applications.

the latter. The time is normalized to L and broken down into the categories of Figure 13—though the Quiescent cycles are too few to see. The average bars are not broken down. The number above each program is the fraction of the program time that we simulate. In Galois, this is the phase in which the work list is populated.

By taking L and re-writing the synchronizations in a lock-free manner in LF, the execution time decreases by an average of only 4%—in fact, in some programs, the time goes up. As we go from LF to Q, the efficient queue-based synchronization reduces the execution time by 22% on average. Among the applications, *CC* and *DT* have the highest reductions. This is because they have the least work between successive calls to the load-to-CAS region and, as a result, suffered a high CAS failure rate. Going from Q to EF, we see that adding eager forwarding reduces the execution time by 12% on average. The reductions are substantial in *IS*, where there are long queues and data is forwarded effectively. The other programs have smaller gains in EF: in *CC* and *DT*, the processors forward data but soon stall as they re-reference the CAS location; in *FFT*, the hardware queue has as many requests for enqueue as for dequeue operations, hampering data forwarding. Finally, going from EF to C, we see that adding group commits reduces the execution time by 40% on average. The largest reductions occur in *CC* and *DT*, where the processors were stalled, and group commit allows them to make progress. In the other programs, C helps processors commit sooner, transforming speculative cycles into non-speculative ones. However, this does not translate into lower execution time.

Overall, C is a very robust design. On average, it reduces the execution time of these sections of applications by 47% relative to Q, and 58% relative to LF. It makes these lock-free sections 2.5x faster on average than the original, lock-based versions (L).

8. Related Work

TM. Most conflict-serializable TM designs [8, 11, 53, 54, 57] abort when multiple transactions read the same variable and then write to it. In contrast, CASPAR preemptively serializes such regions. Other designs [10, 28] similarly preemptively stall loads to contended locations. These designs place special software routines on the transactions' critical path. DATM [57] and OmniOrder [54] transfer speculative data from one transaction to another. They do so to prevent squashes between transactions that commit serially. In contrast, CASPAR transfers speculative data early so that serialized operations can execute in parallel.

Hardware support for scalable synchronization. The proposals most relevant to our work are those that build hardware synchronization queues [20, 32, 43, 55, 56, 58, 68]. In Section 2, we showed how they relate to our work. Some experimental or commercial machines have implemented other scalable synchronization primitives. They include Full/Empty bits in the HEP [31] and Tera multiprocessors [7]; Fetch&Add with request combining in the NYU Ultracomputer [21]; Fetch&Φ operations in the IBM RP3 [52], the SGI Origin [42], and the Cray T3E [60]; and a versatile Synchronization Processor in Cedar [38].

9. Conclusions and Future Work

This paper proposed CASPAR, an architecture that *breaks* the serialization of hardware queues and enables the queued processors to perform lock-free synchronization *in parallel*. CASPAR executes the CASes in the queued-up processors in parallel through eager forwarding, and validates them in parallel through group commit. Compared to existing proposals with hardware queues, CASPAR improves the throughput of kernels by 32% on average, and reduces the execution time of the sections considered in lock-free versions of applications by 47% on average. This makes such sections 2.5x faster than in the original applications.

CASPAR can be improved in several ways. First, we can dynamically rearrange the placement of processors in the queue so that processors that forward values to the queue are placed next to each other. This would increase eager forwarding effectiveness. Also, CASPAR can leverage TM designs such as OmniOrder [54] that allow passing data between transactions. Finally, we can extend CASPAR to allow breaking serialization in TM.

References

[1] The Go Programming Language. `http://golang.org`, 2014.

[2] NetBSD producer/consumer queue. `ftp://ftp.netbsd.org/pub/NetBSD/NetBSD-current/src/sys/kern/subr_pcq.c`, 2014.

[3] fix8: High Performance C++ FIX Framework. `http://fix8.org`, 2014.

[4] MySQL Concurrent Allocator. `https://github.com/twitter/mysql/blob/master/mysys/lf_alloc-pin.c`, 2014.

[5] Y. Afek, G. Korland, and E. Yanovsky. Quasi-Linearizability: Relaxed Consistency for Improved Concurrency. In *Proceedings of the 14th International Conference On Principles Of Distributed Systems (OPODIS 2010)*, volume 6490 of *LNCS*, pages 395–410. 2010.

[6] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit. The SprayList: A Scalable Relaxed Priority Queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 11–20, 2015.

[7] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proceedings of the 4th International Conference on Supercomputing*, ICS '90, pages 1–6, 1990.

[8] U. Aydonat and T. S. Abdelrahman. Hardware Support for Relaxed Concurrency Control in Transactional Memory. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '10, pages 15–26, 2010.

[9] S. Bansal and D. S. Modha. CAR: Clock with Adaptive Replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 187–200, 2004.

[10] G. Blake, R. G. Dreslinski, and T. Mudge. Proactive Transaction Scheduling for Contention Management. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '09, pages 156–167, 2009.

[11] C. Blundell, A. Raghavan, and M. M. Martin. RETCON: Transactional Repair Without Replay. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 258–269, 2010.

[12] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. OpLog: a library for scaling update-heavy data structures. Technical Report MIT-CSAIL-TR-2014-019, MIT Computer Science and Artificial Intelligence Laboratory, September 2014.

[13] J. D. Brouer. Qdisc lockless FIFO. In *Netfilter Workshop*, 2014.

[14] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 52:1–52:12, 2011.

[15] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1243–1254, 2013.

[16] N. Diegues, P. Romano, and L. Rodrigues. Virtues and Limitations of Commodity Hardware Transactional Memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 3–14, 2014.

[17] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 124–131, 2009.

[18] K. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, Computer Laboratory, University of Cambridge, Computer Laboratory, February 2004.

[19] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 229–240, 2013.

[20] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient Synchronization Primitives for Large-scale Cache-coherent Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '89, pages 64–75, 1989.

[21] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – Designing an MIMD, Shared Memory Parallel Machine. In *IEEE Transactions on Computers*, February 1983.

[22] A. Haas, M. Lippautz, T. A. Henzinger, H. Payer, A. Sokolova, C. M. Kirsch, and A. Sezgin. Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 17:1–17:9, 2013.

[23] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA 2004, pages 206–215, 2004.

[24] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13:124–149, January 1991.

[25] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[26] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In *Proceedings of the 11th International Conference on Principles of Distributed Systems*, OPODIS'07, pages 401–414, 2007.

[27] Intel. Intel Xeon Phi Coprocessor. `https://software.intel.com/en-us/mic-developer`, 2013.

[28] S. A. R. Jafri, G. Voskuilen, and T. N. Vijaykumar. Wait-n-GoTM: Improving HTM Performance by Serializing Cyclic Dependencies. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 521–534, 2013.

[29] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A New Approach to Exclusive Data Access in Shared Memory Multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, 1987.

[30] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[31] H. Jordan. Performance Measurements on HEP A Pipelined MIMD Computer. In *International Symposium on Computer Architecture (ISCA)*, June 1983.

[32] A. Kägi, D. Burger, and J. 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.

[33] C. Kirsch, H. Payer, H. Röck, and A. Sokolova. Performance, Scalability, and Semantics of Concurrent FIFO Queues. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP '12)*, volume 7439 of *LNCS*, pages 273–287. 2012.

[34] C. M. Kirsch, M. Lippautz, and H. Payer. Fast and Scalable, Lock-Free k-FIFO Queues. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 241–252, 2013.

[35] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. OSv—Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference*, ATC '14, pages 61–72, June 2014.

[36] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 223–234, 2011.

[37] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-memory Multiprocessors. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, pages 255–264, 1995.

[38] D. Kuck, E. Davidson, D. Lawrie, A. Sameh, C. Q. Zhu, A. Veidenbaum, J. Konicek, P. Yew, K. Gallivan, W. Jalby, H. Wijshoff, R. Bramley, U. M. Yang, P. Emrath, D. Padua, R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, T. Murphy, and J. Andrews. The Cedar System and an Initial Performance Study. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 213–223, 1993.

[39] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 211–222, 2007.

[40] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free FIFO queues. In *Proceedings of the 18th International Symposium on Distributed Computing (DISC 2004)*, volume 3274 of *LNCS*, pages 117–131, 2004.

[41] P.-A. Larson and M. Krishnan. Memory Allocation for Long-running Server Applications. In *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, pages 176–185, 1998.

[42] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 241–251, 1997.

[43] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. In *IEEE Computer*, March 1992.

[44] I. Lotan and N. Shavit. Skiplist-Based Concurrent Priority Queues. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, IPDPS '00, pages 263–268, 2000.

[45] N. D. Matsakis and F. S. Klock, II. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, 2014.

[46] M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed System*, 15(6):491–504, June 2004.

[47] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '04, pages 35–46, 2004.

[48] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, 1996.

[49] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51:1–26, May 1998.

[50] D. Nguyen, A. Lenharth, and K. Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, 2013.

[51] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins. OpenMP Task Scheduling Strategies for Multicore NUMA Systems. *International Journal of High Performance Computing Applications*, 26(2):110–124, May 2012.

[52] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, ICPP '85, pages 764–771, 1985.

[53] X. Qian, B. Sahelices, and J. Torrellas. BulkSMT: Designing SMT Processors for Atomic-block Execution. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, 2012.

[54] X. Qian, B. Sahelices, and J. Torrellas. OmniOrder: Directory-based Conflict Serialization of Transactions. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 421–432, 2014.

[55] R. Rajwar, A. Kägi, and J. R. Goodman. Improving the throughput of synchronization by insertion of delays. In

Proceedings of the 6th International Symposium on High-Performance Computer Architecture, HPCA '00, pages 168–179, January 2000.

[56] R. Rajwar, A. Kägi, and J. R. Goodman. Inferential Queueing and Speculative Push for Reducing Critical Communication Latencies. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, pages 273–284, 2003.

[57] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-aware Transactional Memory for Increased Concurrency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '08, pages 246–257, 2008.

[58] A. Ros and S. Kaxiras. Complexity-effective Multicore Coherence. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 241–252, 2012.

[59] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable Locality-conscious Multithreaded Memory Allocation. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 84–94, 2006.

[60] S. L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '96, pages 26–36, 1996.

[61] S. Seo, J. Kim, and J. Lee. SFMalloc: A Lock-Free and Mostly Synchronization-Free Dynamic Memory Allocator for Many-cores. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 253–263, 2011.

[62] N. Shavit. Data Structures in the Multicore Age. *Communications of the ACM*, 54(3):76–84, Mar. 2011.

[63] M. Stonebraker, A. Pavlo, R. Taft, and M. L. Brodie. Enterprise Database Applications and the Cloud: A Difficult Road Ahead. In *Proceedings of the 2014 IEEE International Conference on Cloud Engineering*, IC2E '14, pages 1–6, 2014.

[64] M. E. Thomadakis. The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms. *Resource*, 3:2, 2011.

[65] P. Trancoso and J. Torrellas. The Impact of Speeding Up Critical Sections with Data Prefetching and Forwarding. In *Proceedings of the 1996 International Conference on Parallel Processing*, ICPP '96, pages 79–86.

[66] R. K. Treiber. Systems Programming: Coping with Parallelism. Technical Report RJ5118, IBM Almaden Research Center, 2006.

[67] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

[68] E. Vallejo, R. Beivide, A. Cristal, T. Harris, F. Vallejo, O. Unsal, and M. Valero. Architectural Support for Fair Reader-Writer Locking. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '10, pages 275–286, 2010.

[69] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the 22nd International Parallel and Distributed Processing Symposium*, IPDPS '08, pages 1–8, 2008.