Temporally Bounding TSO for Fence-Free Asymmetric Synchronization

Adam Morrison
Computer Science Department
Technion—Israel Institute of Technology

Yehuda Afek
Blavatnik School of Computer Science
Tel Aviv University

Abstract

This paper introduces a *temporally bounded total store ordering* (TBTSO) memory model, and shows that it enables *nonblocking fence-free* solutions to *asymmetric synchronization* problems, such as those arising in memory reclamation and biased locking.

TBTSO strengthens the TSO memory model by *bounding* the time it takes a store to drain from the store buffer into memory. This bound enables devising fence-free algorithms for asymmetric problems, which require a performance-critical *fast path* to synchronize with an infrequently executed *slow path*. We demonstrate this by constructing (1) a fence-free version of the *hazard pointers* memory reclamation scheme, and (2) a fence-free *biased lock* algorithm which is compatible with unmanaged environments as it does not rely on safe points or similar mechanisms.

We further argue that TBTSO can be implemented in hardware with modest modifications to existing TSO architectures. However, our design makes assumptions about proprietary implementation details of commercial hardware; it thus best serves as a starting point for a discussion on the feasibility of hardware TBTSO implementation. We also show how minimal OS support enables the adaptation of TBTSO algorithms to x86 systems.

Categories and Subject Descriptors C.1.4 [*Computer Systems Organization*]: Processor Architectures—Parallel Architectures; D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords TSO; bounded TSO; hazard pointers; biased locks; memory fences

Copyright © Owner/Authors, 2015. This is the authors' version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in ASPLOS '15, March 14–18, 2015, Istanbul, Turkey, http://dx.doi.org/10.1145/2694344.2694374.

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey..
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2835-7/15/03. . . $15.00.
http://dx.doi.org/10.1145/2694344.2694374

1. Introduction

Modern multicore architectures implement *relaxed* memory consistency models, which—unlike sequential consistency [21]—allow memory operations to execute out of (program) order. Preventing such reordering, which is crucial for maintaining correctness of synchronization code, requires issuing costly *memory fence* instructions [18].

We focus on the *total store ordering* (TSO) memory model followed by the x86 and SPARC architectures [1, 34], which only allows store/load reordering—i.e., reordering of a store with a later load from a different address. Unfortunately, such a store/load sequence forms the core of the *flag principle* [18], a common synchronization idiom used in algorithms for mutual exclusion [12, 31], memory reclamation [19, 28], and biased locking [33, 40] to name a few. These performance-critical codes must thus issue expensive memory fences that hurt their performance. However, several of these codes are *asymmetric*—they involve synchronization between performance-critical *fast path* code and infrequently executed *slow path* code [11, 28, 33], and only the memory fences on the fast path affect overall performance.

In this paper, we introduce *temporally bounded* TSO (TBTSO)—a strengthening of the TSO memory model that enables asymmetric synchronization with a *fence-free* fast path and a *nonblocking* slow path which can make progress even when the fast path code gets scheduled out. TBTSO facilitates this by guaranteeing a bound Δ on the amount of time it takes a store instruction to become globally visible in memory—i.e., to propagate to memory from the abstract *store buffer* [1, 34] that models store/load reordering in TSO. (TSO store/load reordering occurs when a store remains buffered in the store buffer while a later load from a different address is satisfied from memory.)

1.1 Fence-free asymmetric synchronization

We demonstrate how TBTSO facilitates nonblocking fence-free asymmetric synchronization by developing a fence-free version of the flag principle for TBTSO (§ 3) and applying it to two important problems.

Safe memory reclamation (§ 4) The *safe memory reclamation* (SMR) problem [28] arises in concurrent data structures (particularly *nonblocking* [17] ones) implemented with C/C++, in which multiple operations (e.g., hash table `lookup` and `delete` [27, 38]) access the data structure concurrently. The SMR problem occurs because a thread removing an object O from a data structure cannot immediately reclaim O's memory, since other threads might be reading from the data structure and internally holding a reference to O. Therefore, memory reclamation of an object (slow path) must synchronize with the threads concurrently reading from the data structure (fast path).

We build on *hazard pointers* [28], a nonblocking SMR technique in which each thread maintains a set of *hazard pointers* pointing to objects it may access. After pointing a hazard pointer at an object O, a thread verifies that O has not been removed from the data structure before accessing O for the first time. In standard hazard pointers, this requires issuing a memory fence to order the write of the hazard pointer before the read from the data structure.

TBTSO enables removing this fence. Instead, it is safe to reclaim O's memory if no hazard pointer visibly points to it once Δ time units pass since O is removed from the data structure—even when hazard pointers are written without the subsequent fence. The reason is that any thread holding a reference to O either wrote its hazard pointer at least Δ time units ago, and so its write is globally visible, or it has not yet performed the write and so its (future) validation check will fail since the removal is now globally visible. The hazard pointers method reclaims memory periodically [28], and so Δ time units naturally pass between these periods and no additional waiting is introduced.

Evaluating fence-free hazard pointers (FFHP) on Intel x86 processors shows that they eliminate all overhead associated with hazard pointers, obtaining read-only performance matching that of RCU [26], a reclamation method which imposes no fast path overhead but, unlike FFHP, does not guarantee bounded memory consumption by removed objects.

Biased locking (§ 5) Prior work has shown that many locks in multi-threaded programs are uncontended, and moreover are acquired mostly (or only) by the same thread [20, 30, 33]. This observation lead to the development of *biased lock* algorithms, in which lock acquisition by a designated *owner* thread is cheap—does not use atomic operations—but acquisition by other threads is even more expensive than in standard locks. Existing biased locks either issue fences in the owner acquisition code [30] or block non-owners until the owner reaches a *safe point* outside the critical section [33].

We construct (owner) fence-free biased locks using our TBTSO flag principle. We further adapt the *echo method* [29] to make non-owner lock acquisition speed comparable to standard locks, assuming the owner acquires the lock frequently. Because our lock does not rely on blocking safe points, it (1) can be used in C/C++ programs, which do not naturally define safe points, and (2) enables non-owner acquisition even if the owner is scheduled out or delayed.

Our evaluation shows that the fence-free biased lock (FFBL) outperforms `pthread` locks and performs comparably to safe point based biased locks [33] except for when the owner is delayed outside of the critical section, in which case our FFBL greatly outperform the biased locks.

1.2 Implementing TBTSO

The implementability of TBTSO is an intriguing question. Existing TSO processors do not guarantee bounded time until store visibility, because they do not *fairly arbitrate* [36] the hardware resources (memory units, buses, etc.) involved in propagating stores to memory. In practice, however, stores become visible after only a short delay. Therefore, we propose a hardware design (§ 6) in which the processor *bails out* a store that remains buffered for a long time by *quiescing* the system (this can be done with existing mechanisms [39]).

Unfortunately, we cannot *prove* the feasibility of this design or calculate a worst-case Δ bound, as that entails reasoning about all possible internal interactions that can delay the propagation of a store—many of which are proprietary and undocumented. We thus outline the design and justify—at a high-level—why we believe it is feasible. We also estimate, by extrapolating from measurements on x86 hardware, that this TBTSO design can achieve sub-millisecond Δ bounds. We hope this paper starts a discussion on the implementability of TBTSO and the achievable bounds.

We additionally show that TBTSO algorithms can be adapted to existing x86 systems, provided minimal changes are made in the OS (§ 6). While such adapted TBTSO algorithms experience coarser Δ bounds (1–10 ms) and incur extra work in the slow path, our evaluation shows they perform well.

2. Temporally bounded TSO (TBTSO[Δ])

Here we define TBTSO[Δ], a strengthening of TSO in which a store can remain buffered in a store buffer at most Δ time units. We define TBTSO[Δ] operationally, via an abstract machine whose executions explain observable TBTSO[Δ] behaviors. That is, an execution on a real TBTSO[Δ] machine produces the same read values and final memory state as *some* execution of the abstract TBTSO[Δ] machine.

Technically, the TBTSO[Δ] machine extends Sewell et al.'s x86-TSO abstract machine [34] with a notion of global time. We impose the store buffering bound using this global time by admitting only executions in which the bound holds. Below we describe the TBTSO[Δ] machine informally; the reader can refer to Sewell et al.'s work [34] for the full x86-TSO definitions from which the formal TBTSO[Δ] machine can be derived.

Abstract machine The machine consists of a set of threads, each corresponding to an in-order stream of instructions, that

interact through a memory subsystem. The memory subsystem contains one FIFO store buffer for each thread and is protected by a global fair lock. The memory subsystem lock is used to model atomic read-modify-write operations as being performed by a thread holding the lock. (For simplicity, we use atomic operations directly throughout this paper.) The machine also has a global clock (initially 0) readable by the threads.

The execution of the machine proceeds in *time units*. In each time unit the global clock increases by one. Then, at most one of the following actions can be executed for each thread T if it is valid to do so under the rules below. (This does not mean that a valid action must be executed for T—a scheduler decides the actions in each time unit. Thus, despite the presence of a global clock, the execution is asynchronous.)

The following actions are possible only when the memory subsystem lock is unlocked or held by thread T:

1. The memory subsystem can dequeue T's oldest entry from T's store buffer and write it to memory.
2. T may read: If T reads from an address for which a matching write exists in its store buffer, the read returns the newest corresponding value stored in the buffer. Otherwise, the read returns the value from memory.
3. T may acquire the memory lock if it does not hold it.
4. T may release the memory lock if it holds the lock and its store buffer is empty (if T wishes to release the lock when its store buffer is not empty, the memory subsystem must first empty T's store buffer with #1 actions).

The following are allowed at any time:

5. T can execute a `fence` if its store buffer is empty (similarly to #4, the memory subsystem must act to empty T's store buffer first).
6. T may write, enqueuing an entry to its store buffer.
7. T may read the global clock.

Bounding store buffering time In the TBTSO[Δ] model (where $\Delta \geq 1$), we consider only the abstract machine executions in which the following property holds:

> A write enqueued to a thread's store buffer (action #6) at global time t_0 is written to memory (action #1) at global time $t_1 \leq t_0 + \Delta$.

3. TBTSO flag principle

Fence use in TSO often occurs when applying the *flag principle* [18]. The flag principle says that when two threads, T_0 and T_1, each "raise a flag"—writing to a variable in memory—and then "look" at the other's flag by reading it from memory, then at least one will see the other's flag raised [18]. Of course, correctly ordering "raising the flag" to be globally visible before "looking at the other flag" requires a memory fence on TSO (and TBTSO):

	T_0	T_1
	`flag0 := 1`	`flag1 := 1`
Flag	`fence`	`fence`
principle	`if (flag1)`	`if (flag0)`
	`print "saw` T_1`"`	`print "saw` T_0`"`

This section shows a TBTSO variant of the flag principle that is asymmetric: it removes the fence from T_0's code and shifts the responsibility of maintaining correct ordering of T_0's reads and writes to T_1, which does so using the TBTSO Δ bound. We subsequently apply this asymmetric TBTSO flag principle to remove the fence from the fast path of hazard pointers (§ 4) and biased locks (§ 5).

To devise the TBTSO flag principle, we first use TBTSO's global time to rephrase the original flag principle: If, when T_j reads $\texttt{flag}_\texttt{i}$ at time t_j, T_i's write to $\texttt{flag}_\texttt{i}$ does not appear in memory (i.e., is not yet globally visible), then T_i will necessarily see $\texttt{flag}_\texttt{j}$ raised when it reads $\texttt{flag}_\texttt{j}$ at time $t_i > t_j$. This holds because if this happens, then T_i did not yet execute its fence at time t_j, whereas T_j's write is already globally visible at time t_j because of its fence. TBTSO allows us to break this symmetry by removing the fence from T_0 and placing the responsibility of guaranteeing the above property on T_1, which will *wait* Δ *time units* before reading T_0's flag:

	T_0	T_1
	`flag0 := 1`	`flag1 := 1`
TBTSO		`fence`
flag		`wait` Δ `time units`
principle	`if (flag1)`	`if (flag0)`
	`print "saw` T_1`"`	`print "saw` T_0`"`

Now, if T_0 reads $\texttt{flag}_1$ at time t_0 but T_1's flag write is not yet globally visible, T_0 is still guaranteed that T_1 will see $\texttt{flag}_0$ raised—because in this case T_1 has not yet issued its fence at time t_0 and thus will read $\texttt{flag}_0$ at least Δ time units later, by which time T_0's write is globally visible. In the opposite case, if T_1 reads $\texttt{flag}_0$ at time t_1 but T_0's write is not yet globally visible, then T_0 has not written to its flag before $t_1 - \Delta$. However, T_1's write is globally visible at $t_1 - \Delta$ since T_1 issues a fence, and so T_0 must observe $\texttt{flag}_1$ flag raised.

4. Fence-free hazard pointers (FFHP)

This section describes *fence-free hazard pointers* (FFHP), a nonblocking fence-free SMR algorithm for TBTSO. (Although we build on hazard pointers [28], the ideas described here apply equally well to Herlihy et al.'s *guards* [19]—an SMR method that differs from hazard pointers only in how removed objects are stored before being reclaimed.)

4.1 Standard hazard pointers

In the hazard pointers method, each thread maintains several *hazard pointers*, $\texttt{hp}_0, \texttt{hp}_1, \ldots, \texttt{hp}_\texttt{k}$, which it uses to announce objects it is about to access. Applying hazard pointers in

```
1  Data types:
2  struct MarkPtr {
3    next : pointer to Node
4    mark : boolean (stored in LSB of next)
5  }
6  struct Node {
7    key : keyType
8    nextPtr : struct MarkPtr
9  }
10 Shared variables:
11   head : MarkPtr (list head)
12 Per-thread local variables:
13   prev : pointer to MarkPtr
14   cur, next : pointer to Node
15   hp0, hp1, hp2 : hazard pointers (to Node)
16
17 lookup(key : keyType) {
18   return find(&head, key)
19 }
20
21 delete(key : keyType) {
22   while (true) {
23     if (!find(key)) return false
24     // above find() call initialized cur, next & prev
25     if (!CAS(&cur.nextPtr, <next,0>, <next,1>)) continue
26     if (CAS(prev, <cur,0>, <next,0>))
27       retire(cur)
28     return true
29 } }
```

```
30 find(head : pointer to MarkPtr, key : keyType) {
31   prev := head
32 retry:
     ┌──────────────────────────────────────┐
     │ <cur,mark> := *prev                  │
33   │ hp1 := cur                           │
     │ fence                                │
     │ if (*prev ≠ <cur,0>) goto retry      │
     └──────────────────────────────────────┘
34   while (true) {
35     if (cur = null) return false
       ┌──────────────────────────────────────────────┐
       │ <next,mark> := cur.nextPtr                   │
36     │ hp0 := next                                  │
       │ fence                                        │
       │ if (cur.nextPtr ≠ <next,mark>) goto retry    │
       └──────────────────────────────────────────────┘
37     ckey := ckey.key
38     if (*prev ≠ <cur,0>) goto retry
39     if (!mark) {
40       if (ckey ≥ key) return (ckey = key)
41       prev := &cur.nextPtr
42       hp2 := cur // copy hp1, no need for fence
43                  // Section 4.1 explains why
44     } else {
45       if (CAS(prev, <cur,0>, <next,0>))
46         retire(cur)
47       else
48         goto retry
49     }
50     cur := next
51     hp1 := next // copy hp0, no need for fence
52 } }
```

Figure 1: Hazard pointers usage example in Michael's nonblocking sorted linked list [27] (the `insert` operation is omitted). Boxed code segments show setting of a hazard pointer and verifying that the object has not been removed in the meantime.

a concurrent data structure requires manually changing the algorithm in two ways: First, removed objects may be freed only via the `retire()` method of the hazard pointers. The `retire()` method guarantees that an object is not reclaimed as long as some thread's hazard pointer points to it.

Second, every access to an object must be *protected* by a hazard pointer, that is, one of the thread's hazard pointers must point to the object continuously from a time at which the object was definitely in the data structure (i.e., not `retire()`ed). This entails the following sequence of operations in order to access an object: (1) writing to the hazard pointer, (2) issuing a fence to ensure that the write is globally visible, (3) verifying (in a data structure specific way) that the object is still part of the data structure and has not been retired. Success of this validation implies that the object was part of the data structure throughout steps (1)–(3), so it is successfully protected. If the validation fails, the algorithm typically retries the entire operation, since the state of the data structure has changed under its feet.

Hazard pointers example Figure 1 shows Michael's nonblocking linked list algorithm [27], which uses hazard pointers to safely traverse the list in the presence of concurrent deletions. (Code is presented in C-like pseudo code. Throughout the paper, we ignore the issue of compiler code reordering and assume that memory operations are executed according to the order shown in the pseudo code.) Deletions are done in two steps: first the node is *logically deleted* by setting the least significant bit of its `next` pointer, and only then it is physically removed from the list and passed to `retire()` (Lines 25–27). The `find()` method traverses the list using three pointers, `prev`, `cur` and `next`, and always protects a node with a hazard pointer before reading from it (the boxes at Lines 33 and 36). The hazard pointer validation step consists of verifying that the node being protected is still pointed to by its predecessor. Notice that the shared variable `head` which points to the first element of the list is an immutable sentinel, and therefore `find()` does not need to protect `head` with a hazard pointer.

Copying hazard pointers Figure 1 demonstrates an additional way of setting hazard pointers. In Lines 42 and 51 the value of a hazard pointer is *copied* from one pointer $\mathtt{hp_i}$ to another $\mathtt{hp_j}$. As we explain below, this does not compromise the protection of the hazard pointer, provided that $j > i$.

```
1  Shared variables:
2    hplist : list of all hazard pointers in the system
3  Per-thread local variables:
4    rlist : list of object pointers (removed objects)
5    plist : list of object pointers (copies of hazard pointers)
6    rcount : int
7
8  retire (objp : pointer to object) {
9    rlist .add(objp)
10   rcount += 1
11   if (rcount ≥ R)
12     reclaim ()
13 }
14 reclaim () {
15   // find all non−null hazard pointers in the system
16   plist := ∅
17   foreach (hp ∈ hplist, in ascending index order) {
18     if (hp ≠ null)
19       plist .add( object pointed by hp)
20   }
21   // find all objects which can be removed safely
22   foreach (objp ∈ rlist) {
23     if (objp ∉ plist) {
24       rcount −= 1
25       rlist .remove(objp)
26       free (objp)
27 } } }
28
```

(a) Standard hazard pointers [28]

```
29 Shared variables:
30   hplist : list of all hazard pointers in the system
31 Per-thread local variables:
32 □ rlist : list of <object pointer,time> pairs (removed objects)
33   plist : list of object pointers (copies of hazard pointers)
34   rcount : int
35
36 retire (objp : pointer to object) {
37 □ rlist .add(<objp, global_clock ()>)
38   rcount += 1
39 □ while (rcount ≥ R) // executes at most Δ times
40     reclaim ()
41 }
42 reclaim () {
43   // find all non−null hazard pointers in the system
44   plist := ∅
45 □ now := global_clock ()
46   foreach (hp ∈ hplist, in ascending index order) {
47     if (hp ≠ null)
48       plist .add( object pointed by hp)
49   }
50   // find all objects which can be removed safely
51 □ foreach (<objp,time> ∈ rlist with time < now − Δ) {
52     if (objp ∉ plist) {
53       rcount −= 1
54 □     rlist .remove(<objp,time>)
55       free (objp)
56 } } }
```

(b) Fence-free hazard pointers (FFHP)

Figure 2: Reclamation in hazard pointers and fence-free hazard pointers. Implementation differences are marked by □.

Handling reclamation (Figure 2a) Each thread maintains a list, `rlist`, of all objects it has retired. When the size of `rlist` exceeds a prespecified bound R, the thread calls the `reclaim()` routine to free memory (Lines 9–12). This routine copies the hazard pointers of every thread into a private list, `plist` (Lines 15–20). Next, `reclaim()` frees any previously retired node which is not in `plist`, i.e., not protected by a hazard pointer (Lines 21–27). (Notice that each thread's hazard pointers are scanned in ascending index order, to guarantee that `reclaim()` does not miss a hazard pointer being copied: If thread T copies value v from $\mathtt{hp_i}$ to $\mathtt{hp_j}$ ($j > i$), overwrites $\mathtt{hp_i}$, and the reclaiming thread observes $\mathtt{hp_i}$'s new value, then it will necessarily observe v in $\mathtt{hp_j}$ because stores are ordered under TSO.)

Progress guarantee The `reclaim()` subroutine is *wait-free* [17], as it completes within a bounded number of steps. As a result, hazard pointers can be used by nonblocking algorithms without comprising their progress guarantee.

Complexity analysis Let N be the number of threads in the system, and $H \geq N$ the total number of hazard pointers. Recall that R is the maximum number of nodes a thread may retire and that have not yet been reclaimed, which is the amount of "waste" memory the thread may hold. If the set `plist` is implemented as a hash table and the set `rlist` as a simple linked list, then a `reclaim()` does $O(H+R)$ work to reclaim at least $R-H$ nodes. Therefore, if $R = H + \Omega(H)$, `reclaim()` does an expected amortized $O(1)$ steps per retired node. In practice, however, it is simpler to implement `plist` as an array, sort it and perform lookups using binary search [28]. Doing this provides a worst-case amortized bound of $O(\log H)$.

4.2 Fence-free hazard pointers implementation

This section presents the new fence-free reclamation method, which we obtain by observing that hazard pointers apply the flag principle: Here, a thread T_0 about to reference an object O interacts with T_1, the thread reclaiming O's memory after removing it from the data structure. T_0 pointing a hazard pointer at O constitutes "raising its flag," and the subsequent validation check "looks" at T_1's flag. Conversely, the removal of O from the data structure forms T_1's "flag raising," and it "looks" at T_0's flag when scanning T_0's hazard pointers. (Note that this implies that the remove of O must be globally visible when O is retired. This holds if the removal

performs an atomic operation, which flushes the store buffer. Otherwise, `retire()` must issue a fence.)

In fence-free hazard pointers, we thus apply the TBTSO flag principle as follows: (1) We omit the fence from the hazard pointer validation code (e.g., from the boxes at Lines 33 and 36 of Figure 1). (2) We defer trying to reclaim O's memory for Δ clock ticks since its removal (Figure 2b). Importantly, this additional waiting time does not lie on the critical path of the threads and so does not negatively impact their performance. The Δ clock ticks naturally overlap the time period between consecutive memory reclamation attempts, since a thread accumulates multiple retired objects before trying to reclaim memory.

Retirement For each retired node, the thread records the time (the value of the global clock, which we assume never wraps around) at which the node was retired (Line 37). Reclaiming memory is now done in a loop until some node is freed, because (as explained below) a single invocation of `reclaim()` is not guaranteed to free a node (Line 39). However, as explained below (in Progress guarantee), this loop is still guaranteed to be wait-free.

Reclamation For every retired object O, the reclaiming thread checks whether some hazard pointer protects O only if, at the time that the `reclaim()` has started, more than Δ clock ticks have passed since O was retired (Lines 45 and 51). We implement `rlist` as a linked list to which `retire()` appends, so scanning `rlist` from oldest to newest retired objects is trivial and costs $O(1)$ per object. Notice that if no object O is old enough, `reclaim()` exits without freeing any memory.

Progress guarantee The `while` loop which calls `reclaim()` *is wait-free* [17], provided that $R > H$. Once Δ time units pass since the latest retirement, all R objects in the thread's `rlist` are checked, and thus at least one object is reclaimed. Because Δ is a fixed constant, the loop executes a bounded number of times, overall—there is no indefinite waiting.

4.2.1 Space/time tradeoff

The complexity of the reclamation depends on how large is R relative to Δ. There are two cases. The first is when $R \geq c\Delta \geq H + \Omega(H)$ for some $c > 1$. In this case, a `reclaim()` can find at least $(1 - 1/c)R - H$ objects to reclaim, because in our model every retired node receives a unique time stamp, and so all but the last Δ retired nodes are safe to check against the hazard pointers. Therefore, `reclaim()` has the same amortized complexity as in hazard pointers. In the remaining case, $\Delta > R > H$. This constrained case can be detected in advance (as Δ, R, and H are known) and requires changing the `reclaim()` routine slightly, so that it does not do any work until Δ ticks have passed since the retirement time of the oldest $H+1$ objects in the `rlist`. With this modification, the worst-case running time of reclaim becomes $O(\Delta)$.

To give a practical sense to these parameters, we note that in experiments exercising Michael's nonblocking linked list algorithm [27] on an x86 machine with 80 hardware threads we observe a maximal retirement rate of 1300 nodes per millisecond per thread (see § 7). In § 6 we argue that such a machine can implement TBTSO with Δ on the order of 0.5 to 10 milliseconds. Therefore, setting R to $1300 \times 10 \times 2 = 26000$ nodes—which may take up 2 megabytes—guarantees that a `reclaim()` can always free at least $R/2$ nodes.

5. Fence-free biased locking (FFBL)

This section describes our *fence-free biased lock* (FFBL). A *biased lock* is a lock in which a designated *owner* thread can acquire the lock quickly if there is no contention, possibly at the expense of the other *non-owner* threads whose lock acquisition is slower [20, 30, 33].

5.1 FFBL algorithm

We present a mutual exclusion algorithm based on the flag principle, which we then convert into our FFBL using the TBTSO flag principle. The flag principle can be used to implement two-thread mutual exclusion by having each thread (1) raise its flag and check the other's flag, (2) enter the critical section if the other's flag is not up, otherwise lower its flag and try again, and (3) lower its flag when it exits the critical section. However, such a scheme can *livelock* if the two threads always see each other's flag up and thus never enter the critical section. Thus, mutual exclusion algorithms based on the flag principle, such as Peterson's [31] and Dekker's [12], introduce a mechanism to break such livelocks. These mechanisms are quite intricate, since the classic algorithms use only read and write operations. In contrast, we can use atomic operations, so long as they do not degrade the owner's *fast path*—accessing an uncontended lock. Next, we describe this mechanism.

Baseline biased lock (top row of Figure 3) Because the flag principle works for two threads, we introduce a standard lock, `L`, to prevent multiple non-owners from participating in the flag protocol. Therefore, from here on we shall speak only of T_0—the owner—and T_1, which is some non-owner that has acquired `L`. We also use the new lock `L` to break the flag protocol's symmetry and prevent livelock. If T_0 sees that $\texttt{flag}_1$ is up, it lowers $\texttt{flag}_0$ and tries to acquire `L`. For its part, if T_1 sees $\texttt{flag}_0$ raised after raising its own $\texttt{flag}_1$, then it waits for T_0 to lower $\texttt{flag}_0$. This maintains the property that a thread that does not see the other's flag raised enters the critical section immediately, but if both see each other's flag T_1 will enter first. Nevertheless, because biased locks are targeted at scenarios in which T_1 participates rarely, *in the common case T_0 immediately enters the critical section.*

Fence-free biased lock (bottom row of Figure 3) Converting the baseline into an FFBL merely requires using the TBTSO flag principle in place of the standard one. However, if the owner T_0 acquires the lock very frequently—as expected from biased locks, otherwise biasing does not make a difference—then it will get delayed for Δ time units

```
1  //
2  // Basic algorithm
3  //
4
5  Shared variables:
6    flag0, flag1 : bits
7    L : standard lock
8
9
10
```

(a) Variables

```
11 lock() {
12   flag0 := 1
13   fence
14
15   if (flag1) {
16     flag0 := 0
17     L.lock()
18   }
19 }
20
```

(b) Owner lock

```
21 unlock() {
22   if (flag0) {
23     flag0 := 0
24   } else {
25     L.unlock()
26   }
27 }
28
29
30
```

(c) Owner unlock

```
31 lock() {
32   L.lock()
33   flag1 := 1
34   fence
35   await (flag0 = 0)
36 }
37 unlock() {
38   flag1 := 0
39   L.unlock()
40 }
```

(d) Non-Owner

```
41 //
42 // Fence-free
43 // algorithm
44 //
45
46 Shared variables:
47   flag0, flag1 :
48     struct {v:63 bits, f:1 bit}
49   L : standard lock
50
51
52
53
```

(e) Variables

```
54 lock() {
55   flag0 := <0,1>
56   // no fence
57
58   while (flag1.f) {
59     do
60       flag0 := <flag1.v,0>
61     while (L.is_locked())
62     if (L.try_lock())
63       break
64   }
65 }
66
```

(f) Owner lock

```
67 unlock() {
68   if (flag0.f) {
69     flag0 := <0,0>
70   } else {
71     flag0 := <0,0>
72     L.unlock()
73   }
74 }
75
76
77
78
79
```

(g) Owner unlock

```
80 lock() {
81   L.lock()
82   flag1 := <flag1.v+1,1>
83   fence
84   now = global_clock()
85   await ((global_clock()>now+Δ)
86          or (flag0.v=flag1.v))
87   await (flag0.f = 0)
88 }
89 unlock() {
90   flag1 := <flag1.v+1,0>
91   L.unlock()
92 }
```

(h) Non-Owner

Figure 3: Fence-free biased locking (FFBL) construction. Top: Basic (not fence-free) algorithm. Bottom: FFBL algorithm.

whenever T_1 attempts to acquire the lock, which is suboptimal. This happens because once the owner, T_0, sees T_1's flag raised, it lowers its flag and tries to acquire `L`. But `L` is held by T_1 which is waiting for Δ time units.

We turn this problem around, exploiting the fact that T_0 frequently acquires the lock to speed up T_1's lock acquisition, which in turn speeds up T_0 as well. We do this by having T_0 notify T_1 whenever it sees $\texttt{flag}_1$ raised (which causes T_0 to try to acquire the `L` lock). This signals to T_1 that T_0 is waiting to acquire `L`, so T_1 can stop the Δ delay and enter the critical section. To implement this notification we use *echoing* [29]: We expand the flags to 64-bits, 63 of which are used as *version numbers* that uniquely identify each write—whenever T_1 writes to $\texttt{flag}_1$, it increases $\texttt{flag}_1$'s version. T_0 uses this version to notify T_1 that it is spinning while trying to acquire `L`, by writing—or *echoing*—what it reads from $\texttt{flag}_1$ into $\texttt{flag}_0$ (Lines 59– 63). (For simplicity, the code implements a simple `trylock` loop to do this, but one can incorporate echoing into the spin loop of any lock, e.g., to guarantee that T_0 acquires `L` by using a fair lock implementation.) Similarly, T_1 looks for T_0's echoes as it spins waiting for Δ time units to pass, and stops waiting once it observes an echo (Figure 3h). In practice the echoes reach memory much faster than Δ time units (§ 6.1.2), and so the gains from this optimization can be substantial (§ 7.2).

6. Implementing TBTSO

This section discusses guaranteeing bounded-time store buffering. (Modern x86 systems already support an *invariant timestamp counter* that can be read cheaply and used as a global clock [2, 32], thus satisfying the other requirement for implementing TBTSO.)

We present a simple hardware design for implementing TBTSO[Δ] on x86 systems (§ 6.1). We estimate the design can achieve a Δ bound as low as $\approx 6P\,\mu$s on a system with P hardware threads; as we discuss shortly, we unfortunately cannot prove feasibility of the design and the Δ bound, as that requires proprietary implementation knowledge. We further show how minimal OS support enables adaptation of TBTSO algorithms to existing x86 systems (§ 6.2).

6.1 Enforcing TBTSO with hardware quiescence

x86 processors may delay propagating a store to the memory subsystem indefinitely if a resource required to do so—e.g., a cache port—gets constantly used by another unit. In practice, however, most stores propagate quickly. Our idea is thus to detect a starving store that remains buffered for a long time, and *bail it out* by invoking the system's existing *quiescence* mechanism [39]. This pauses all competing units, which enables the starving store to propagate to the memory subsystem.

We cannot, unfortunately, provide a complete design and calculate an exact worst-case Δ bound. Doing that requires accounting for any possible interaction between hardware mechanisms that can delay the propagation of a store—many of which are proprietary, undocumented, and/or obscure. Instead, we (1) present a high-level design while pointing out—and justifying—its assumptions on internal mechanisms (§ 6.1.1), and (2) estimate its Δ bound by extrapolating from measurements performed on x86 hardware (§ 6.1.2).

6.1.1 TBTSO design

x86 processors retire a store instruction from the reorder buffer without waiting for its value to propagate to the memory subsystem (caches and memory). The processor instead holds a retired store's data in a *store buffer* and propagates it to memory asynchronously [2, 3].

Propagating a store to the memory subsystem—i.e., to the L1 cache—uses hardware resources with limited capacity: for example, (1) the number of *ports* through which an instruction can access the L1 cache [3], (2) the number of *line fill buffers* that are required to hold a missed cache line's data as it comes in [3], (3) the capacity of the communication links and queues through which cache coherence messages go through [16], and so on.

The system *arbitrates* access to these resources to decide, in each cycle, which unit is granted access to each resource. If arbitration is *unfair*, some unit may be prevented from accessing a resource in the face of competition [36, Section 9.3.3]. For example, consider a policy favoring line fill buffer allocation for loads over (buffered) stores (e.g., because a load cannot complete without handling the miss). Then a store followed by a sequence of loads, all of which miss in the L1, will be prevented from being written to the L1 cache. (We have found evidence of unfair policies in performance counter documentation [3, Section B.6.9.3].)

However, the hardware can force a store out of the store buffer by forcing *quiescence*—pausing all competition in order to allow the store to drain out of the buffer. (Without such a mechanism, the system would be vulnerable to absurd situations, e.g., failing to retire a fence instruction because a store cannot be drained.) Because a store may be starved by system-wide competition—on interconnect links or memory controllers—we must obtain *system-wide quiescence*, which is already supported on modern x86 hardware [39].

Hardware design We propose to force quiescence whenever a store remains buffered past a certain amount of time after its retirement (discussed later). Then, once the system is quiescent, propagate every buffered store to the memory subsystem. For this approach to achieve *bounded* store buffering time requires that both forcing quiescence and then propagating a store to the memory subsystem complete in bounded time. Below, we justify these assumptions:

Bounded quiescence time (Assumption 1) We believe quiescence is obtained within bounded time based on the high-level description of the x86 quiescence implementation [39] (however, this property is not explicitly guaranteed). Essentially, a quiescence request is signalled on a sideband channel, instructing cores and memory controllers to block new memory operations (i.e., instructions or DMAs) [39]. Once this happens, we are left with a finite number of pending memory operations—and thus finite levels of competition for resources—so the pending operations drain out within bounded time, achieving quiescence.

Bounded quiescent propagation time (Assumption 2) Propagating a store to the memory subsystem should complete in bounded time on a quiescent system, because the required cache coherence transactions and memory writebacks face no competition on the communication links and queues. Additional delays may occur due to the memory (RAM) itself, but to our knowledge such delays are also bounded—e.g., DRAM refresh events, or recoverable ECC errors that generate exceptions which are handled asynchronously [2, Section 15.5].

The only exception we are aware of is a store to the IO or configuration space of a PCI device. Such a store waits for a completion response from the device, which can delay arbitrarily before responding. (In contrast, writes to general device memory are *posted* and do not wait for an acknowledgment [7, Chapter 2], and thus the former reasoning applies.) Device IO/config stores do not compromise our design, as we explain next.

Putting it together In our design the processor forces system-wide quiescence if a store remains buffered for more than τ cycles since its retirement, where τ is such that most stores propagate to the memory subsystem in $< \tau$ cycles (in § 6.1.2, we estimate τ can be $10\,\mu s$). Stores propagate from the store buffer in retirement order (due to TSO), and so the timeout always expires first for the oldest store in the store buffer, leading to quiescence and subsequent propagation of all buffered stores. Crucially, x86 guarantees that a store to device IO/config space drains the store buffer (including itself) when it retires [2, Section 11.3]. There can thus never be buffered (retired) stores *behind* such an IO/config write, and so the (arbitrary) delays of IO/config writes do not prevent achieving a bound on standard store visibility.

6.1.2 Estimating achievable Δ bound

To get a sense of what a feasible Δ bound might be, we measure the time to force quiescence on real x86 hardware and extrapolate from that. We also measure store visibility times on this system, to estimate what the timeout that triggers quiescence can be—for good performance, we need a timeout that expires rarely but that does not make the Δ bound exceedingly large.

Our measurements use a system with four Xeon E7-4870 (Westmere EX) processors, each of which has 10 2.40 GHz cores that multiplex 2 hardware threads, for a total of 80 hardware threads overall. Based on the measurements below,

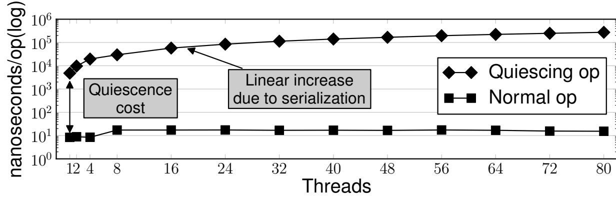

Figure 4: Time to reach system-wide quiescence (log scale) as the number of quiescing threads increases.

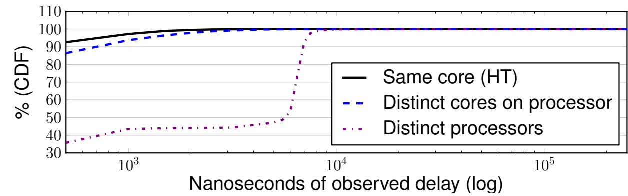

Figure 5: Cumulative distribution of observed store buffering times (log scale) for various writer/reader thread placements.

we estimate that similar TBTSO hardware could be implemented with $\Delta = 500\,\mu$s (or $\approx 6\,\mu$s per hardware thread).

Quiescence time We run a microbenchmark in which all threads repeatedly perform an atomic operation that crosses cache line boundary to a thread-private location, which forces system-wide quiescence on our system [39]. We measure the average latency of an operation. For comparison, we rerun the benchmark with each thread performing a standard atomic operation to a thread-private location.

Figure 4 shows the results: Quiescing the system takes $\approx 5\,\mu$s, about 600× the cost of the normal operation. Because the system must serialize quiescence operations, the time it takes a thread to enforce quiescence grows almost linearly with the number of threads. The results do not change if cores not participating heavily load the system (we omit plots due to space constraints), and thus we believe this measurement represents worst-case time to achieve quiescence.

Store buffering time in practice We use a microbenchmark with two threads: (1) a *writer* that writes the value of the global timestamp counter to a shared variable v (initially 0), and then issues an endless sequence of non-store instructions, and (2) a *reader* which reads v until it sees a non-zero value. The reader then reports the difference between the current time and the time read from v. We vary writers' non-store sequence, to see if different instructions have different effect. For example, to see if the processor delays a store in the store buffer so that it could be read more quickly, we use a benchmark in which the writer repeatedly reads v. To maximally exercise different scenarios, we (1) run the benchmark with different placement of threads—hardware threads of the same core, cores of the same processor, and on different processors, (2) run the benchmarks alone, concurrently, and with the memory-intensive STREAM benchmark [24] in the background. In all, we run $2 \cdot 10^{10}$ executions.

Figure 5 shows the cumulative distribution of the observed delays—99.9% of stores become visible after at most $10\,\mu$s. Notice that this benchmark overestimates the store buffer delay—e.g., it also measures the time it takes the reader to acquire v's cache line.

Estimating Δ: Because quiescence forcing is serialized, we estimate its worst-case time for this system at $80 \times 5\mu\text{s} = 400\,\mu$s (the maximal time measured was $300\,\mu$s). We assume that quiescence time dwarves the time to subsequently drain the store buffer. Still, as a safety margin, we extrapolate that a TBTSO hardware implementation can obtain a Δ of $500\,\mu$s on a system similar to our test machine.

6.2 Adapting TBTSO algorithms to x86 with OS help

On the x86 architecture, a user/kernel transition—i.e., an interrupt or context switch—drains the contents of the store buffer to memory [2, Section 11.10]. By programming the hardware to generate a timer interrupt on every core every Δ time units, the OS can thus force periodic store buffer flushes. (In fact, operating systems already use periodic per-core timer interrupts to trigger expiration of process time quanta and other housekeeping code [9].) Unfortunately, this does not suffice to enforce TBTSO[Δ]: First, timer interrupt precision is not guaranteed—a timer interrupt might fire late. Second, the interrupt only triggers a store buffer flush—current hardware does not bound the time to complete the flush (and hence the delivery of the interrupt).

In practice, however, timer interrupts rarely get delayed arbitrarily, which means that a system with periodic interrupts behaves similarly to a TBTSO system. We now show how simple OS support allows adapting TBTSO algorithms to benefit from this similarity.

OS support The OS maintains an array A in which each core writes the current time when it receives a timer interrupt or enters the kernel. This array is mapped read-only into the address space of every process. (Similarly to how OSes expose their time-keeping variables to implement `gettimeofday()` without trapping into the kernel [8].)

Adapting TBTSO algorithms To establish that every store retired by time t_0 is globally visible, we can adapt the TBTSO policy of "wait Δ ticks" to wait instead for every entry in A to indicate a time $> t_0$. As in TBTSO, this only adds work when we need to wait, and not at time t_0. Thus, in hazard pointers for example, we only add work to the reclamation path and not to every object retirement. (In contrast, the bounded TSO[S] model [29], which does not have a global clock, would require reading A when retiring an object.)

Adapted TBTSO algorithms have two disadvantages. First, the Δ granularity must be *coarse* to minimize the overhead of timer interrupt processing—e.g., current systems use a timer interrupt period of 1 to 10 milliseconds [9].) Second, adapted algorithms incur extra work in the slow path. However, our evaluation shows this overhead is small.

7. Evaluation

This section evaluates our FFHP and FFBL algorithms. We emulate a TBTSO system by using an x86 machine on which we configure the algorithms to use $\Delta = 0.5$ ms. (This corresponds to the estimated obtainable Δ bound for such a machine from § 6.1.2.) We also evaluate the algorithms adapted to run safely on x86 hardware, as described in § 6.2. We emulate the required OS support in user space using POSIX timers to interrupt each thread every $\Delta = 4$ ms (a typical OS timer interrupt period). When describing FFHP and FFBL results, we use the Δ values to distinguish between the TBTSO and the adapted variants that rely on period timers.

Platforms We use two test systems: (1) a server with four Intel Xeon E7-4870 processors, each with 10 2.40 GHz cores that multiplex 2 hardware threads, for a total of 80 hardware threads, and (2) a Core i7-4770 (Haswell) processor, which has 4 3.4 GHz cores, each with 2 hardware threads.

7.1 Safe memory reclamation

We compare FFHP to several SMR algorithms in terms of overhead imposed and memory consumption. We test the following SMR methods: (1) hazard pointers (HP) [28], (2) RCU [26], a widely used scheme [25] that imposes zero overhead on the fast path, (3) drop the anchor (DTA) [6], a quiescence-based method that guarantees bounded memory consumption but requires the fast path to periodically issue an atomic operation, and (4) StackTrack [4], an SMR method that uses hardware transactional memory in the fast path.

Benchmark We use the same methodology as in prior work on SMR [4, 6, 19, 25, 28]— a microbenchmark focusing on a widely used data structure with a read-dominated operation mix, in which memory fences are a significant penalty. Our benchmark, based on McKenney's benchmark [25], is written in C. It consists of n threads performing `lookup()`s, `insert()`s and `remove()`s on a 1024-bucket concurrent hash table with chaining. (Threads are *pinned* to the processors in a round-robin manner, so that every run has threads on all processors.) We vary the size of the universe U from which the key range is picked, allowing us to control the average length L of each chain. We test chains of average length $L = 4$ (short, as in real hash tables), $L = 20$ and $L = 80$ (medium), and $L = 256$ (long). Due to space constraints, we focus on short and long chains, as the results for medium chains are qualitatively similar to the long chains. We test both read-only workloads, and workloads in which readers and updaters work concurrently, as detailed below. Each execution starts with $U/2$ random keys in the hash tables. The chains are implemented with Michael's nonblocking linked lists [27], which support concurrent reading and updating. We report median results from 10 runs, each lasting 10 seconds. Results' variance is negligible (we use a dedicated test machine). The tests use the `jemalloc` memory allocator to prevent the memory allocator from being a bottleneck.

Implementations To obtain the best possible performance, we use the OS in tickless mode. We use McKenney's implementation of hazard pointers [25]. All hazard pointers implementations use $R = 32000$ (which translates to about 2 megabytes). We use the original authors' implementation of RCU [10] and DTA [6], and implement StackTrack following [4]. To avoid irrelevant cache-related effects, the hash table nodes are equally sized in all implementations.

7.1.1 Performance and overheads

Read-only workload (left column of Figure 6) This test evaluates the overhead each SMR method imposes on read-only code, modeling scenarios in which updates are rare. All threads perform random `lookup()`s over the hash table. Since no reclamation is done, the value of Δ has no impact on FFHP performance. FFHP and RCU perform similarly ($\approx 5\%$ difference) in all tests, outperforming HP by 30% on the Westmere-EX machine, and by 60% (short operations) to 15% (long operations) on the Haswell. (We do not include Haswell plots due to space constraints.) FFHP's advantage is solely due to the omission of memory fences compared to HP, as otherwise their lookup code is identical.

DTA and StackTrack do not perform as well as FFHP and RCU. In DTA, every `lookup()` operation updates a per-thread *timestamp* of when it begins and ends (including issuing a fence), and sets a per-thread *anchor* variable using an atomic `compare-and-swap` at least once. This constitutes significant overhead in short operations, causing DTA's throughput to be 30% worse than that of FFHP. Similarly, StackTrack imposes overhead on the fast path by using transactions, leading to 10% less throughput than FFHP for short operations. In long operations, StackTrack starts to experience transaction *capacity aborts* [2], forcing it to *split* each operation into multiple transactions [4]. As a result, StackTrack's throughput is $0.3\times$ that of FFHP.

Read/write workloads (last two columns of Figure 6) In this test, we split the n threads into $\frac{3}{4}n$ *reader* threads and $\frac{1}{4}n$ *updater* threads. A reader performs only `lookup()`s over the entire universe, while an updater alternates between `insert()`ing and `remove()`ing each item in an equally-sized subset of the universe owned by it.

The middle column of Figure 6 shows the throughout obtained by the readers. For all the SMR methods but DTA, the throughput is about $60\% - 70\%$ of the throughput obtained in the read-only case, but with essentially the same throughput ratios between these methods. This is caused by increased reader cache miss rates due to the updaters modifying the hash table. DTA suffers from additional cache misses: updating a reader's timestamp, which is read by updaters, now frequently causes a cache miss. Consequently, DTA's read throughput does not scale past 64 threads. The periodic timer interrupts in the (adapted) FFHP[4 ms] executions have no performance impact, due to their low frequency.

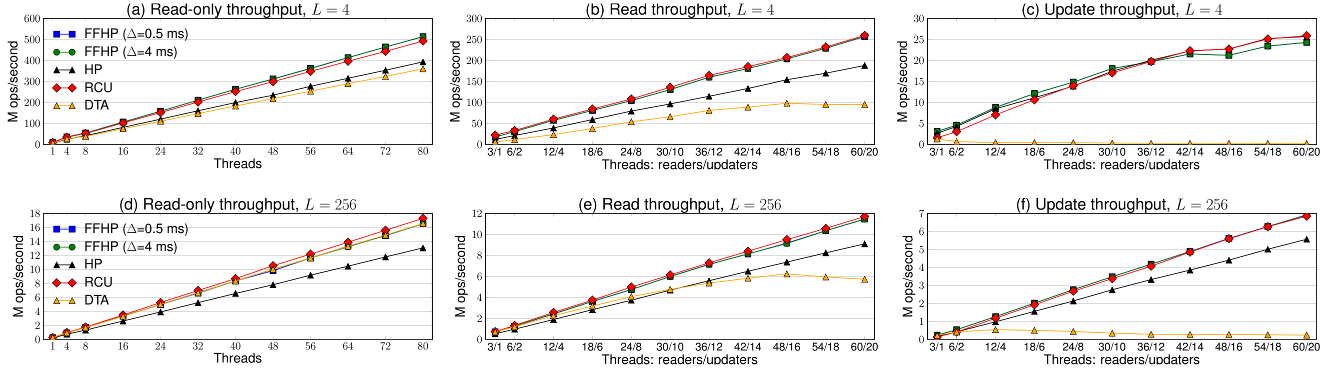

Figure 6: Quad Westmere-EX hash table throughput, for various average bucket list sizes (L).

The right column of Figure 6 shows the updaters' throughout. There is little impact by the Δ parameter, because R provides enough headroom so that a `reclaim()` finds enough nodes to free without waiting. We also see that the extra work in FFHP[4 ms] (scanning the per-core time array, see § 6.2) adds no noticeable overhead compared to the original (TBTSO[0.5 ms]) version. On the Westmere-EX system, FFHP and RCU perform the same ($< 1\%$ difference) on high thread counts, obtaining the same throughput as HP in short operations and improving on HP by 25% in long operations. The reason is that as the chains grow longer, the relative cost of the atomic operations an updater does diminishes, making the traversal cost more significant. On the Haswell system, FFHP outperforms HP by 30% (short operations) to $\approx 50\%$ (long operations). FFHP also outperforms RCU by $\approx 60\%$ and StackTrack by 10% in short operations. Due to aborts and transaction splitting FFHP outperforms StackTrack by 80% in long operations. DTA updates perform $> 100\times$ worse than other methods because an updater reads each thread's timestamp after removing a node [6], an act that can cost more than the original hash chain traversal.

7.1.2 Retired nodes memory consumption

Here we quantify the impact that thread stalls—e.g., due to context switches—have on the SMR schemes. We modify the read/write test done above so that one reader stalls for s milliseconds in one of its `lookup()`s, as would occur on a context switch. We measure the peak memory consumption of the benchmark (as reported by the OS) for various stall times. Figure 7 depicts the results from the Westmere-EX server, in which three trends are apparent.

First, FFHP keeps more removed nodes unreclaimed than HP, because FFHP's `reclaim` frees only the subset that was removed Δ milliseconds ago. In this test, this amounts to at most 7% greater memory consumption than in HP. Second, memory consumption with RCU is 40% greater than when FFHP or HP are used even when there is zero stalling. The reason is that the RCU library handles reclamation more slowly than hazard pointers, since it uses background threads which periodically wake up and free memory. Finally, memory consumption with RCU keeps growing as the stall time increases, because RCU cannot free memory as long as a thread is stalled inside an operation. At maximum stall time, the benchmark using RCU consumes $6\times$ (short operations) to $2\times$ (long operations) more memory than the FFHP.

7.2 Biased locks

We evaluate the FFBL on a benchmark C program that synthesizes various access patterns to the lock. The benchmark consists of two threads, the owner and non-owner, repeatedly acquiring the lock for a period of 10 seconds, with a random *interarrival* delay (simulating application work) between lock acquisitions. We test FFBL versions with and without the echoing optimization, comparing them to the default Linux `pthreads` lock and to our implementation of biased locking using safe points [33], in which we assume that the owner reaches a safe point immediately when it exits the critical section.

Figure 8 shows the throughput—lock acquisitions per seconds—of both owner and non-owner, normalized to the throughput obtained with `pthreads`. The first access pattern, in which the owner arrives frequently and the non-owner rarely (once a millisecond on average) simulates a workload favorable to biased locking. Here the owner path of the FFBL and safe point locks outperforms `pthreads` by 5%–10%, and the non-owner is comparable, except for FFBL[4 ms] without echoing. Here, due to the long delay time, both owner and non-owner perform worse than with `pthreads`.

The next two patterns challenge the locks by increasing the frequency of non-owner arrivals until it equals the owner's frequency. This crystallizes the benefit of echoing, as without it the throughput of the FFBL versions collapses.

The last pattern captures the case in which the owner stalls for a long time—due to a context switch or long computation. While all biased locks collapse compared to pthreads in this case, we see the benefit of bounded delay: the FFBL outperform the safe point lock by $50\times$ (with echoes) and by $7\times$ (without echoes).

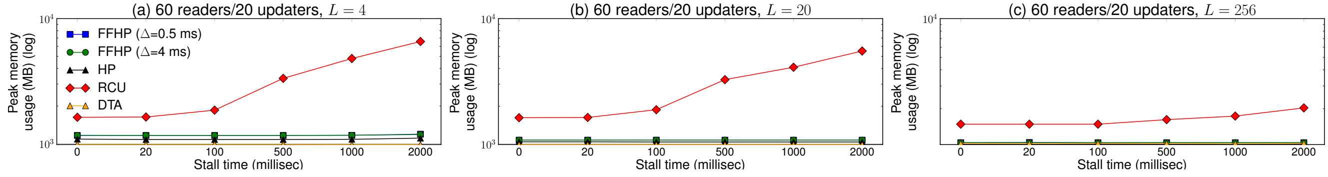

Figure 7: Read/write test memory consumption (log scale) when one reader stalls in its operation.

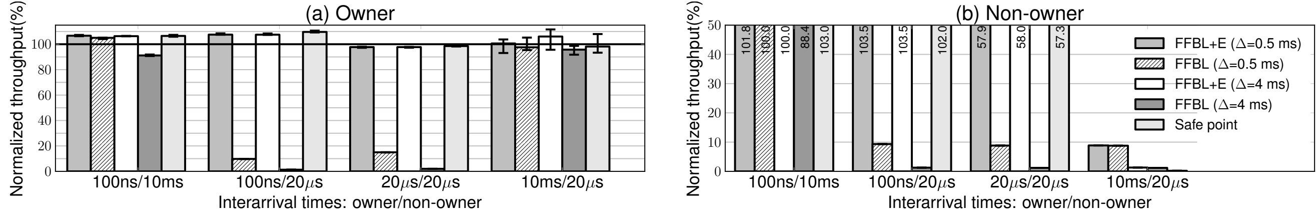

Figure 8: Biased locks throughput, normalized to standard `pthreads` lock.

8. Related work

Bounded reordering TSO[S] [29] is a *spatially* bounded strengthening of TSO in which the store buffer has bounded capacity, S. In TSO[S], a thread's store can remain buffered for an unbounded amount of time if the thread does not issue further stores. This creates uncertainty about the thread's state, making TSO[S] unsuitable for nonblocking fence-free algorithms—indeed, fence-free work stealing algorithms based on TSO[S] require either relaxed semantics or blocking [29]. In contrast, TBTSO's *temporal* reordering bound facilitates nonblocking synchronization without relaxing semantics, making it more broadly applicable.

Liu et al. [23] empirically observe that x86 processors usually propagate stores to memory quickly, and exploit this in designing a fence-free read/write lock. They do not explore how to guarantee bounded store buffering, and instead use inter-processor interrupts to handle cases in which store propagation is delayed. Dice et al. [11] consider methods for an asymmetric fence-free flag principle and mutual exclusion. One of their methods assumes a maximum store buffer delay. However, they do not discuss how to obtain or enforce this delay. They do not use echoing and thus may perform poorly in some workloads (§ 7). Finally, they do not consider safe memory reclamation.

In the context of distributed systems, the *timed consistency* model of Torres-Rojas et al. [37] requires that a write issued by a node become visible to all nodes after at most Δ time units. Torres-Rojas et al. do not apply timed consistency to shared memory systems as we do.

Safe memory reclamation Most quiescence-based memory reclamation methods, such as epoch-based reclamation [15] and RCU [26], cannot be both nonblocking and guarantee bounded memory consumption, in contrast to our FFHP method. Braginsky et al.'s *drop the anchor* (DTA) quiescence-based method [6] achieves both properties, but introduces other limitations not present in FFHP: (1) DTA requires intrusive changes to the data structure to support DTA's *freezing* operation, and (2) it performs poorly on short operations such as in hash tables (see § 7). SMR methods using hardware transactional memory (HTM) [4, 13] are also nonblocking with bounded memory consumption, but using transactions in the fast path imposes overhead (§ 7).

Biased locking FFBLs have a fence-free owner path and allow a non-owner to enter the critical section in bounded time (when the lock is not held). Prior work either blocks the non-owner [33, 40] or uses fences in the owner path [30].

Eliminating fence penalty Several architectural proposals aim at eliminating the penalty of memory fences [5, 14, 22, 35, 41]. However, TBTSO algorithms can be adapted to work on *existing* machines with simple OS changes. In addition, the modifications required to implement TBTSO in hardware are modest compared to these designs, since they rely on triggering existing quiescence mechanisms.

9. Conclusion

We have proposed TBTSO, a temporally bounded TSO memory model that bounds the amount of time it takes for a store to become globally visible, and shown that TBTSO enables nonblocking fence-free asymmetric synchronization.

The TBTSO concept raises several research questions: Are there additional applications for the TBTSO model, perhaps in real-time systems? On the hardware side, we have argued that to our understanding, a TBTSO implementation is feasible. But are there obscure or undocumented internal mechanisms that invalidate our understanding? Can we calculate and verify a TBTSO bound for a real processor?

Acknowledgments

We thank the ASPLOS reviewers for their invaluable comments, which helped us reshape and improve the paper.

This work was supported by the Israel Science Foundation (grants 1227/10, 1749/14 and 1386/11) and by Yad-HaNadiv foundation. Adam Morrison is supported in part at the Technion by an Aly Kaufman Fellowship.

References

[1] *The SPARC Architecture Manual Version 8*. Prentice Hall, 1992.

[2] Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3: System Programming Guide. `http://download.intel.com/products/processor/manual/325384.pdf`, June 2013.

[3] Intel 64 and IA-32 Architectures Optimization Reference Manual. `https://www-ssl.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html`, July 2013.

[4] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit. StackTrack: An Automated Transactional Approach to Concurrent Memory Reclamation. In *Proceedings of the 9th European Conference on Computer Systems*, EuroSys '14, pages 25:1–25:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. .

[5] C. Blundell, M. M. Martin, and T. F. Wenisch. Invisifence: Performance-transparent memory ordering in conventional multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 233–244, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0. .

[6] A. Braginsky, A. Kogan, and E. Petrank. Drop the anchor: lightweight memory management for non-blocking data structures. In *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 33–42, New York, NY, USA, 2013. ACM.

[7] R. Budruk, D. Anderson, and E. Solari. *PCI Express System Architecture*. Pearson Education, 2003. ISBN 0321156307.

[8] J. Corbet. On vsyscalls and the vdso. `http://lwn.net/Articles/446528/`, 2011. Linux World News.

[9] J. Corbet. (Nearly) full tickless operation in 3.10. `http://lwn.net/Articles/549580/`, 2013. Linux World News.

[10] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, 2012.

[11] D. Dice, H. Huang, and M. Yang. Asymmetric Dekker Synchronization. `http://home.comcast.net/~pjbishop/Dave/Asymmetric-Dekker-Synchronization.txt`, 2001.

[12] E. W. Dijkstra. Cooperating sequential processes. `http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF`, 1968.

[13] A. Dragojević, M. Herlihy, Y. Lev, and M. Moir. On the Power of Hardware Transactional Memory to Simplify Memory Management. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, pages 99–108, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0719-2. .

[14] Y. Duan, A. Muzahid, and J. Torrellas. WeeFence: toward making fences free in TSO. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 213–224, New York, NY, USA, 2013. ACM. .

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

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ISBN 0123704901.

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

[18] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123705916, 9780123705914.

[19] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems (TOCS)*, 23(2):146–196, May 2005.

[20] K. Kawachiya, A. Koseki, and T. Onodera. Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 130–141, New York, NY, USA, 2002. ACM. ISBN 1-58113-471-1. .

[21] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, Sept. 1979. ISSN 0018-9340. .

[22] C. Lin, V. Nagarajan, and R. Gupta. Address-aware fences. In *Proceedings of the 27th International Conference on Supercomputing*, ICS '13, pages 313–324, New York, NY, USA, 2013. ACM. .

[23] R. Liu, H. Zhang, and H. Chen. Scalable Read-mostly Synchronization Using Passive Reader-Writer Locks. In *Proceedings of the 2014 USENIX Annual Technical Conference*, USENIX ATC '14, pages 219–230, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2.

[24] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[25] P. E. McKenney. Structured deferral: synchronization via procrastination. *Communications of the ACM*, 56(7):40–49, July 2013. .

[26] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems*, IASTED '98, pages 508–518. ACTA Press, 1998.

[27] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM.

[28] 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.

[29] A. Morrison and Y. Afek. Fence-free Work Stealing on Bounded TSO Processors. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 413–426, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. .

[30] T. Onodera, K. Kawachiya, and A. Koseki. Lock Reservation for Java Reconsidered. In M. Odersky, editor, *ECOOP 2004 – Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 559–583. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22159-3.

[31] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981. ISSN 0020-0190.

[32] W. Ruan, Y. Liu, and M. Spear. Boosting Timestamp-based Transactional Memory by Exploiting Hardware Cycle Counters. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):40:1–40:21, Dec. 2013. ISSN 1544-3566. .

[33] K. Russell and D. 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, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. .

[34] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[35] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. End-to-end sequential consistency. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 524–535, Washington, DC, USA, 2012. IEEE Computer Society.

[36] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011. ISBN 1608455645, 9781608455645.

[37] F. J. Torres-Rojas, M. Ahamad, and M. Raynal. Timed Consistency for Shared Distributed Objects. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, PODC '99, pages 163–172, New York, NY, USA, 1999. ACM. ISBN 1-58113-099-6. .

[38] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proceedings of the 2011 USENIX Annual Technical Conference*, USENIX ATC'11, pages 145–158, Berkeley, CA, USA, 2011. USENIX Association.

[39] J. R. Vash, B. Jung, and R. Tan. System-wide quiescence and per-thread transaction fence in a distributed caching agent. `http://www.google.com/patents/US8443148`, 2013. US Patent 8443148 B2.

[40] N. Vasudevan, K. S. Namjoshi, and S. 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, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. .

[41] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 266–277, New York, NY, USA, 2007. ACM. .