Fence-Free Work Stealing on Bounded TSO Processors

Adam Morrison *
Computer Science Department
Technion – Israel Institute of Technology

Yehuda Afek
Blavatnik School of Computer Science
Tel Aviv University

Abstract

Work stealing is the method of choice for load balancing in task parallel programming languages and frameworks. Yet despite considerable effort invested in optimizing work stealing task queues, existing algorithms issue a costly *memory fence* when removing a task, and these fences are believed to be necessary for correctness.

This paper refutes this belief, demonstrating work stealing algorithms in which *a worker does not issue a memory fence* for microarchitectures with a *bounded* total store ordering (TSO) memory model. Bounded TSO is a novel restriction of TSO – capturing mainstream x86 and SPARC TSO processors – that bounds the number of stores a load can be reordered with.

Our algorithms eliminate the memory fence penalty, improving the running time of a suite of parallel benchmarks on modern x86 multicore processors by $7\% - 11\%$ on average (and up to 23%), compared to the Cilk and Chase-Lev work stealing queues.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords work stealing; memory fences; TSO

1. Introduction

The *task-based* parallel programming model – which exposes parallelism by expressing a computation as a set of tasks that can be scheduled in parallel – is used in many programming languages and frameworks, as well as in (multicore) MapReduce. These implementations of task-based parallelism dominantly employ *work stealing* for dynamic load balancing of the executed tasks [3, 11, 13, 20, 27, 32, 33].

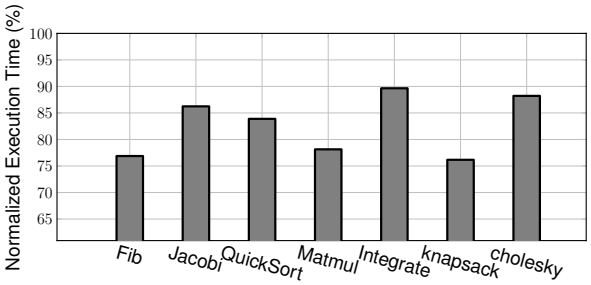


Figure 1: Single threaded execution time of several widely used CilkPlus benchmarks when not issuing a memory fence on task removal, normalized to the standard CilkPlus runtime on an Intel Haswell (Core i7-4770) 3.4 GHz processor.

In work stealing, each *worker* thread has a queue of tasks from which it continuously removes the next task to execute. While executing a task the worker might create and add new tasks to its queue. If a worker's queue empties, the worker becomes a *thief* and tries to steal a task from another worker.

Today's work stealing synchronization protocols [4, 9, 14, 20] are based on the *flag principle* [23]. The worker publishes the task it is about to take, and then checks whether a thief intends to steal the same task. If not, the worker can safely take the task because any future thief will observe its publication and avoid stealing the task. But for this reasoning to hold, the worker must issue a costly *memory fence* instruction to prevent the checking load from being *reordered* before the publishing store. As Figure 1 shows, this fence can account for up to $\approx 25\%$ of execution time.

It would thus seem that the worker's memory fence is unavoidable. In fact, Attiya et al.'s "laws of order" [10] are sometimes interpreted as saying just this [8]. In truth, however, the "laws of order" rely on certain assumptions [10] and may not hold when these underlying assumptions are invalidated – as they are in this work.

This paper demonstrates that *linearizable fence-free work stealing is possible* on mainstream multicore architectures with a total store ordering (TSO [1, 34]) memory model,

ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.

ACM 978-1-4503-2305-5/14/03…$15.00.
http://dx.doi.org/10.1145/2541940.2541987

*Work done while a PhD student at Tel Aviv University.

such as x86 and SPARC. Our insight is that these processors' TSO implementation only allows *bounded* store/load reordering (i.e., a load can be reordered with at most *S* prior stores), and that this bound can be used by a thief *instead of relying on the worker's fence* to verify that the worker could not have already removed the task it is stealing.

1.1 Bounded TSO

This paper introduces the *bounded* TSO memory model, which places a fixed bound on the size of the abstract *store buffer* [34] that models store/load reordering in TSO. This is the only type of reordering possible in TSO: a stored value gets buffered in the store buffer before reaching memory, allowing a later load from a different address to be satisfied from memory before the earlier store is written to memory.

We show that the mainstream TSO architectures – x86 and SPARC – implement bounded TSO (except for a corner case when there are consecutive stores to the same location, which can be prevented in software), and describe how to determine the reordering bound in practice.

1.2 Fence-free work stealing using bounded reordering

In a typical work stealing task queue, the worker works its way from the tail of the queue to its head and the thief works from the head towards the tail. The worker's fence removes uncertainty about the worker's position in the task queue by draining its store buffer.

Our insight is that if the worker does not issue a memory fence, a thief can use knowledge of the store buffer's capacity to detect when it can safely steal a task by bounding the number of task removals hidden in the worker's store buffer. To see this, consider the system's state after a worker works its way through tasks #10, #9 and #8 – without issuing any fences – on a processor with a 4-entry store buffer:

	stores by worker	
in memory:	about to take task #10	(earliest store)
	about to take task #9	
buffered	some store	
stores:	about to take task #8	
	another store	(latest store)

If a thief now reads from memory it will only see that the worker is about to take task #10. But the thief knows it is missing at most 4 worker announcements due to store buffering, implying that the furthest store the worker could have issued is "about to take task #6." Therefore, if the thief intends to steal task $i < \#6$, it is assured that the worker has not taken this task yet.

Based on this idea, we describe the FF-THE and FF-CL algorithms, fence-free variants of Cilk's THE algorithm [20] and of the Chase-Lev algorithm [14]. In FF-THE and FF-CL, if a thief remains *uncertain* about whether it can safely steal – say, if it intends to steal task #6 above – it refuses to steal and returns a special ABORT value instead. In doing so we are *relaxing* the work stealing semantics, but unlike previous semantic relaxations [31], our relaxation maintains the queue's *safety* and does not allow a task to be removed twice – an intolerable behavior for many applications.

We further show that work stealing can be implemented fence-free *without* relaxing its semantics. We describe the THEP algorithm, which resolves the uncertainty in FF-THE using an *echo* mechanism. When a thief cannot steal due to uncertainty, it writes a value to memory and *waits* for the worker to *echo* it back, at which point the thief is guaranteed that the worker has observed its presence and will correctly synchronize with it if necessary. The thief does not risk waiting indefinitely because in programs using work stealing the worker keeps taking tasks until the queue empties, so it will eventually notice and echo the thief's value.

Evaluation Modifying Intel's Cilk Plus [3] runtime (which is used by the C/C++ parallel extensions in Intel's compilers) to use FF-THE and THEP instead of THE improves the running time of the benchmarks from Figure 1 by 11% – 15% on average (and by up to 23%) on Intel Westmere-EX and Haswell processors. In addition, FF-CL outperforms the Chase-Lev algorithm by 17% on average on common graph problems, achieving performance comparable to that of Michael et al.'s idempotent work stealing queues [31], which are fence-free but can dequeue a task more than once.

Sidestepping the laws of order Our algorithms violate a *tightness* assumption of the "laws of order" impossibility result [10], namely that every legal sequential execution can actually occur in the algorithm. In our algorithms, the execution in which a thief running alone steals from a queue containing one item cannot occur. FF-THE and FF-CL avoid this execution by refusing to steal in such a state, whereas in THEP the thief would wait for the worker and never complete. (We discuss this in detail in § 6.)

In showing how the use of bounded reordering enables circumventing the "laws of order" theorem by violating its tightness assumption, we hope to open the door for removing memory fences in other concurrent algorithms.

1.3 Contributions

To summarize, our contributions are:

- Introducing the bounded TSO memory model, showing that it captures mainstream TSO processors, and describing how to measure reordering bounds in practice.
- The FF-THE, FF-CL and THEP work stealing algorithms, which achieve fence-freedom by exploiting the reordering bound.
- Describing how the use of bounded reordering enables violating the "laws of order" tightness assumption.
- Implementation and evaluation of our fence-free work stealing algorithms, showing they eliminate the overhead of fences and outperform existing algorithms.

2. TSO[S]: Bounded TSO memory model

This section defines TSO[S], a bounded TSO memory model in which a load can be reordered with at most S prior stores. The model is defined via an abstract machine whose execution provides an operational explanation for observable program behaviors under TSO[S]. In other words, any execution on a real TSO[S] machine should produce the same read values and final memory state as some execution of the abstract TSO[S] machine.

The abstract TSO[S] machine is essentially Sewell et al.'s x86-TSO abstract machine [34] in which the store buffers are bounded. We therefore describe the TSO[S] machine informally, and refer the reader to Sewell et al.'s work [34] for the full formal definitions.

Abstract TSO[S] machine The machine consists of a set of threads that interact through a memory subsystem. Each thread corresponds to an in-order stream of instructions. The memory subsystem contains one FIFO store buffer per thread, whose *capacity* – the number of stores it can hold – is S. The memory subsystem is protected by a global fair lock, which is used to model atomic read-modify-write instructions (e.g., `compare-and-swap`) as being performed while holding the lock. (Elsewhere in this paper we simply use atomic operations directly.) The execution of the machine is a sequence of events describing actions performed by the memory subsystem and the threads, under the following rules.

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 propagate it to memory. We assume that each memory write is eventually propagated from the relevant store buffer to the shared memory [34].
2. T can read. If T reads from an address for which a matching store exists in its store buffer, the read returns the newest corresponding value in the store buffer. Otherwise, the read returns the value from memory.
3. T can acquire the lock if it does not already hold it.
4. T can release the lock if it holds the lock and its store buffer is empty (if T wishes to release the lock while its store buffer is not empty, the memory subsystem must take steps propagating T's writes to memory until T's store buffer empties).

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 take enough steps to empty T's store buffer first).
6. T can write, enqueuing an entry to its store buffer, provided the store buffer is not full (if the store buffer is full, the memory subsystem must first dequeue and propagate at least one entry to memory, similarly to #4 and #5).

3. Work stealing

3.1 Work stealing sequential specification

A *work stealing queue* is a double-ended queue that supports three methods: `put()`, `take()` and `steal()`. A `put(`y`)` enqueues y to the tail of the queue. A `take()` applied to a non-empty queue dequeues from its tail. A `steal()` applied to a non-empty queue dequeues from its head. A `take()` or `steal()` applied to an empty queue return EMPTY.

3.2 Background: work stealing synchronization

Modern work stealing algorithms [4, 9, 14, 20] strive to reduce the overhead experienced by workers performing the computation [20], even at the cost of making steal operations more expensive. As a result, these algorithms have converged on a similar design in which the worker uses a protocol based on the *flag principle* [23] to detect if a conflict with a thief might exist. If so, the worker switches to a heavier synchronization protocol to decide whether the worker or thief gets the task.

Figure 2a shows the general design. The queue consists of a (cyclic) array of W tasks with non-wrapping head and tail indices, i.e., an index with value i points to element i mod W of the tasks array. The head, H, points to the oldest task in the queue. The tail, T, points to the first unused array element. If $T = H$ the queue is *empty*. (For simplicity, we omit details of resizing the array if it becomes full.)

A worker performs a `put()` by storing the task at the tail of the queue, and then incrementing T. The TSO model guarantees that the store of the task and the subsequent store incrementing T are not reordered.

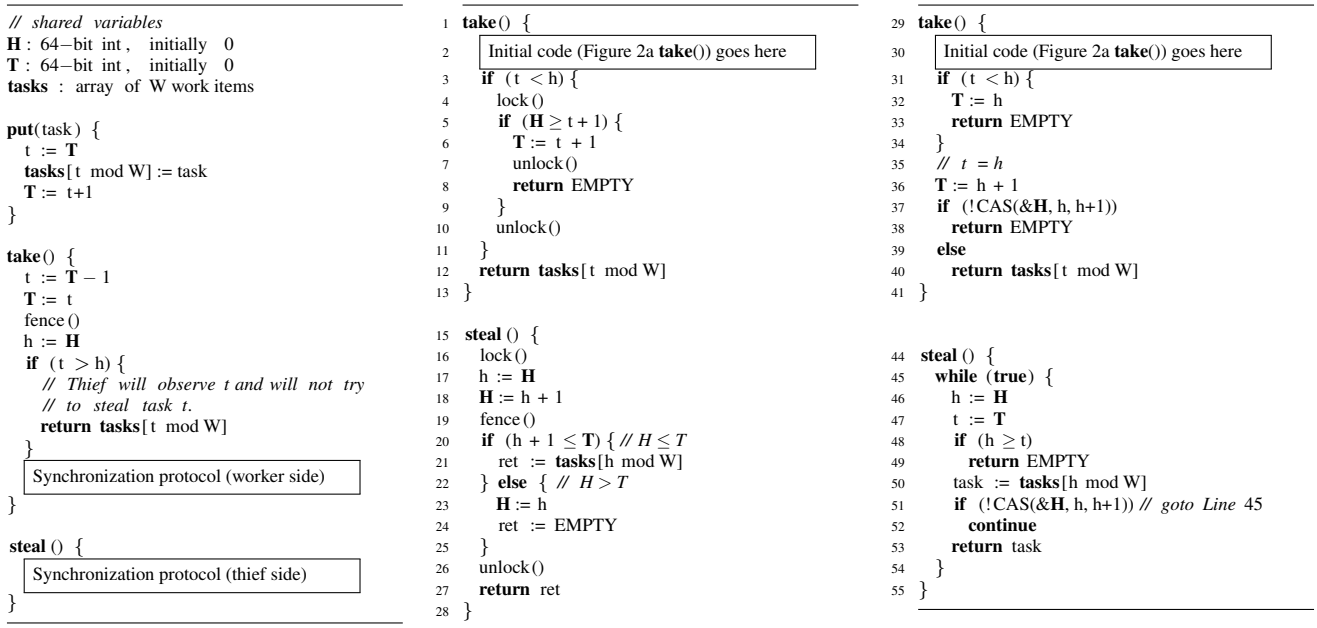
To `take()` a task, the worker "raises its flag" by decrementing T from $t+1$ to t, thereby publishing its intent to take task t (i.e., the task pointed to by index t). It then *reads* the head index H after issuing a memory fence to ensure that reading H is not reordered before decrementing T.

If the worker observes that $t > H$, it can safely remove task t from the queue, as it has verified there can be no conflict for task t: when the tail update became globally visible, thieves have announced intent to steal only tasks up to $H < t$, which means that a new steal operation will observe a queue that does not contain task t.

However, if the worker observes that $t \leq H$ there may be a conflict with a thief. The algorithms differ in the synchronization protocol used to handle such a conflict. In the following we continue with the description of each algorithm's protocol:

Cilk's THE algorithm (Figure 2b) Cilk's THE algorithm uses a per-queue lock to synchronize between a worker and a thief, and also to enforce mutual exclusion among thieves. In case of a conflict on a task, the protocol picks the worker as the winner.

A thief acquires the queue lock and then "raises its flag" by incrementing the head index H from h to $h+1$, thereby



```
// shared variables
H : 64−bit int ,  initially  0
T : 64−bit int ,  initially  0
tasks : array of W work items

put(task) {
  t := T
  tasks[t mod W] := task
  T := t+1
}

take() {
  t := T − 1
  T := t
  fence()
  h := H
  if (t > h) {
    // Thief will observe t and will not try
    // to steal task t.
    return tasks[t mod W]
  }
  Synchronization protocol (worker side)
}

steal() {
  Synchronization protocol (thief side)
}
```

(a) Algorithm outline.

```
1  take() {
2    Initial code (Figure 2a take()) goes here
3    if (t < h) {
4      lock()
5      if (H ≥ t + 1) {
6        T := t + 1
7        unlock()
8        return EMPTY
9      }
10     unlock()
11   }
12   return tasks[t mod W]
13 }

15 steal() {
16   lock()
17   h := H
18   H := h + 1
19   fence()
20   if (h + 1 ≤ T) { // H ≤ T
21     ret := tasks[h mod W]
22   } else { // H > T
23     H := h
24     ret := EMPTY
25   }
26   unlock()
27   return ret
28 }
```

(b) Cilk THE [20].

```
29 take() {
30   Initial code (Figure 2a take()) goes here
31   if (t < h) {
32     T := h
33     return EMPTY
34   }
35   // t = h
36   T := h + 1
37   if (!CAS(&H, h, h+1))
38     return EMPTY
39   else
40     return tasks[t mod W]
41 }

44 steal() {
45   while (true) {
46     h := H
47     t := T
48     if (h ≥ t)
49       return EMPTY
50     task := tasks[h mod W]
51     if (!CAS(&H, h, h+1)) // goto Line 45
52       continue
53     return task
54   }
55 }
```

(c) Chase-Lev [14].

Figure 2: Design of modern work stealing task queues. Both the Cilk THE and Chase-Lev algorithms use the flag principle to detect when the worker and thief might contend for a task. They differ in the synchronization used to manage such a conflict.

publishing its intent to steal task h. It then issues a memory fence before checking if $H \leq T$. If so, the thief knows its increment of H will be observed by any future worker and thus the thief can safely steal task h. Otherwise ($H > T$), there are two possible cases: either the queue was empty ($T = H$) when the thief arrived, or a worker has just published its intent to take the same task (e.g., initially $T = 1$ and $H = 0$, then the worker's decrement and the thief's increment cross, leading to a state in which $T = 0$ and $H = 1$). Either way, the thief restores H to its original value and aborts the steal attempt.

This behavior makes it safe for a worker which (following its decrement of T) finds that $T = H$ to take the task. The remaining case, in which a worker observes $T < H$ after its decrement, is again caused either by an initially empty queue or because of a concurrent steal attempt (which will abort). The worker therefore acquires the queue lock and returns the task or restores the queue to a consistent state if it was empty.

Chase-Lev algorithm (Figure 2c) The Chase-Lev non-blocking [21] algorithm uses an atomic `compare-and-swap` (CAS) operation to pick the winner in a conflict on a task. A thief reads the queue's head and tail, and if the queue is not empty (i.e., $T > H$) the thief tries to atomically increment H from h to $h+1$ using a CAS. If the CAS succeeds, the thief has stolen the task.

To support this simple stealing protocol (and in contrast to the THE algorithm) a worker must always increment H to remove the last task. After decrementing T, if the worker finds that $T = H$, it restores T to its original value and attempts to take this last task by incrementing H with a CAS. Otherwise ($T < H$) then the queue was initially empty or a thief has concurrently incremented H. In either case, the worker returns `EMPTY` after fixing the queue's state by setting T to H.

3.3 Linearizability of work stealing algorithms

The standard correctness condition for concurrent algorithms is *linearizability* [24], which requires that a method appears to take effect at some point in time during its execution. However, the Cilk THE and Chase-Lev work stealing algorithms are not linearizable under TSO [30]. For example, a `put()` may be delayed in the worker's store buffer and missed by a thief, causing a linearizability violation as the following Chase-Lev execution shows:

initially, $T = 0$ and $H = 0$

worker	**thief**
`put()` invoked	
buffered stores: $tasks[0] := item$, $T := 1$	
`put()` completes	
	`steal()` invoked
	read $H = 0$, $T = 0$
	return EMPTY
	`steal()` completes

In practice, such linearizability violations do not affect work stealing clients: The worker keeps dequeuing tasks until the queue empties, and so either it or a `steal()` invoked after the `put()`'s stores flush to memory will remove the task. Therefore, while adding a fence before the `put()` completes fixes these violations [30], deployed work stealing implementations do not do so.

Our fence-free algorithms have similar linearizability violations (and fix). We point this out to emphasize that these fixable linearizability violations are shared by existing work stealing algorithms, and are *not* the reason we circumvent the "laws of order" theorem and obtain fence-freedom.

4. Fence-freedom by reasoning about bounded reordering

This section derives our first technique for (worker) fence-free work stealing. We use Cilk's THE algorithm as a concrete running example, developing the FF-THE algorithm. We apply the same principles to develop FF-CL, a fence-free version of the Chase-Lev algorithm, in § 4.1.

The task queue we obtain does not comply with the original (deterministic) work stealing specification (§ 3.1), but with a *relaxed non-deterministic* specification in which a `steal()` operation can non-deterministically return `ABORT` without changing the state of the queue. We use non-determinism because the condition under which a `steal()` operation returns `ABORT` will be internal to the implementation and not part of the specification. Importantly, this relaxation does not impact the correctness of the work stealing's client (program), as it still maintains the task queue's *safety* and does not allow a task to be removed twice.

Task queue safety Once a thief makes its intent to steal task h globally visible (by incrementing H and issuing a fence), it needs *to verify that the worker is not concurrently trying to take the same task*, i.e., that $T > h$. Knowing this makes stealing task h safe: any subsequent `take()` attempt will observe the updated queue head and not try to remove task h without acquiring the lock. The standard THE protocol ensures that the thief observes T's exact value, so it can compare T to h. But the point is that *any method* for answering the question "is $T > h$?" will do.

Bounding worker position The technique we propose is to leverage the bounded store buffer capacity in the bounded TSO model to deduce how far off the worker's real position is from the position read from memory by the thief.

Let S be the store buffer's capacity. (We discuss how to determine S in practice in § 7.) A worker `take()` does one store to T which decrements it by 1. Therefore, at the time at which a thief observes that $T = t$, the last value stored by the worker must be at least $t - S$. So if $t - S > h$, the thief can safely steal the task. Otherwise, it must return the new `ABORT` value. More generally, a thief can safely steal task h whenever it observes $T > h + \delta$, where $\delta \geq 1$ is the maximum number of stores to T by `take()` operations that can exist in the store buffer – which can also contain stores executed by the client program in between task queue operations. Thus, if we know that the client always does at least x stores between consecutive `take()` operations, we have

$$\delta = \left\lceil \frac{S}{x+1} \right\rceil .$$

```
56 steal () {
57   lock()
58   h := H
59   H := h + 1
60   fence()
61   if (T − δ > h) {
62     ret := tasks[h mod W]
63   } else {
64     H := h
65     ret := ABORT
66   }
67   unlock()
68   return ret
69 }
```

Figure 3: FF-THE: fence-free THE algorithm. The code of `put()` and `take()` remains the same, but for the removal of the memory fence in `take()`. The parameter δ is the maximum number of stores to T by `take()` operations that can exist in the store buffer.

Figure 3 shows the modifications required to implement reasoning about the worker's store buffer in the THE algorithm. Notice that now the thief never knows for certain whether the queue is empty, because there is always uncertainty about the final store performed by the worker (i.e., $\delta \geq 1$). Thus, the condition for returning `ABORT` subsumes the condition for returning `EMPTY` in the original algorithm.

Determining δ To determine δ, we need to obtain a lower bound on x, the number of stores between `take()`s. We can easily get a bound by inspecting the runtime's code. For example, the CilkPlus runtime [3] updates a field in the dequeued task after removing it from the queue. Thus, we trivially have that $x \geq 1$ for any CilkPlus program.

To obtain a better bound, we can run a static analysis on the basic block control-flow graph of the program and search for a weighted shortest path from `take()` to itself, where we assign the number of stores performed in a basic block B as the weight of each edge going out of B.

Context switches The discussion thus far assumes the worker always uses the same store buffer. This does not hold if the operating system reschedules the worker thread, moving it from one core to another. However, it is easy to see that an operating system moving a thread from core C_1 to core C_2 must drain C_1's store buffer. For example, if the thread loads from a location stored to on C_1 while it runs on C_2, it must observe the value previously stored. Indeed, vendor manuals document this requirement [6]. The discussion in this section thus rightfully considers only the last core a worker runs on.

4.1 FF-CL: Fence-free version of the Chase-Lev queue

The technique of bounding the worker's position applies to the Chase-Lev algorithm, but with a somewhat different correctness argument. In the Chase-Lev algorithm a worker about to remove the last task undoes its update of T and uses CAS to advance the queue's head. Thus, a thief about to remove task h needs to verify that the worker's store writing $T := h$ cannot be in the store buffer. If this is the case, the thief is guaranteed that if the worker tries to remove task h, it will synchronize with the thief using a CAS. As before, checking that $T > h + \delta$ establishes this. Figure 4 shows the pseudo code of the modified algorithm.

```
70  steal () {
71    while (true) {
72      h := H
73      t := T
74      if (h ≥ t)
75        return EMPTY
76      if (t − δ ≤ h)
77        return ABORT
78      task := tasks[h mod W]
79      if (!CAS(&H, h, h+1)) // goto Line 71
80        continue
81      return task
82    }
83  }
```

Figure 4: FF-CL: fence-free Chase-Lev algorithm. The code of `put()` and `take()` remains the same, but for the removal of the memory fence in `take()`.

5. Fence-free work stealing without relaxed semantics

This section shows that work stealing can be implemented fence-free *without* relaxing its semantics. We achieve this by adding *worker echoes* to the FF-THE algorithm (§ 4) to obtain the THEP algorithm, a fence-free implementation of the standard work stealing specification. THEP also avoids a potential problem in FF-THE, in which a thief misses a stealing opportunity if it reads a true value of the queue's tail that happens to be within δ from the head. (Though in some cases this may not be a real problem, since it means the queue is almost empty and the worker can empty it soon.)

Echoes To safely steal, the thief needs to verify that the worker has observed its update of H. To establish this, the thief maintains a "heartbeat" counter which it increments on each `steal()`. In turn, the worker writes the value it reads from this counter to a new variable, P, allowing the thief to wait for P to reflect its counter. (Hence, the name THEP of the new algorithm.) TSO guarantees that any value the thief subsequently reads from T was written by the worker after it observed the thief's update of H. The thief thus listens for the worker's "echo," reflected in P, until it knows the worker has observed its update of H.

```
84  // shared variables
85  H : struct { s:32 bits , h:32 bits }
86  P : initially ⊥
87  // T and tasks remain unchanged

89  take() {
90    t := T − 1
91    T := t
92    <s, h> := H
93    if (t < h) {
94      lock()
95      P := ⊥
96      <s, h> := H
97      if (h ≥ t + 1) {
98        T := t + 1
99        unlock()
100       return EMPTY
101     }
102     unlock()
103   } else {
104     P := s
105   }
106   return tasks[t mod W]
107 }
```

```
108 steal () {
109   lock()
110   <s, h> := H
111   H := <s + 1, h + 1>
112   fence()
113   if (T − δ ≤ h) {
114     while (P ≠ s+1) {
115       if (h+1 > T))
116         goto Line 122
117     }
118     t := T
119     if (h + 1 ≤ t) {
120       ret := tasks[h mod W]
121     } else {
122       H := <s + 1, h>
123       ret := EMPTY
124     }
125   } else {
126     ret := tasks[h mod W]
127   }
128   unlock()
129   return ret
130 }
131
```

Figure 5: The fence-free THEP algorithm. As before, $\delta \geq 1$ is the maximum number of stores to T by `take()` operations that can exist in the store buffer.

Using this approach yields an algorithm that meets the original deterministic specification of work stealing (§ 3.1) and never needs to abort a steal attempt. The price we pay is that occasionally a thief must block and cannot make progress until the worker arrives and updates P. Fortunately, in clients using work stealing the worker keeps taking tasks until the queue is empty, because it cannot rely on the work being stolen. Thus, if the queue is not empty, the worker eventually arrives and the thief can proceed.

However, we must make certain that the thief does not have to wait when the queue is empty, because in that case the worker may never arrive to respond. Therefore, if while waiting the thief notices that $T = H$ (the queue is empty) it stops and returns EMPTY. (The thief can miss a buffered `put()` and wrongly return EMPTY, resulting in a non-linearizable execution of the same kind that already exists in the THE algorithm, as described in § 3.3.)

The THEP algorithm (Figure 5) The thief maintains its counter in the top bits of H. (The counter can also be maintained in a separate variable, at the cost of an extra load in the `take()` path.) On each `steal()` attempt, the thief increments the counter when it updates H. Then, if it is uncertain about the worker's position, it spins, reading T and P until one of the following occurs: (1) If the queue becomes empty (i.e., $T < H$ which means T was equal to H before the thief incremented H), the thief returns EMPTY (Lines 115-116). (2) If P echoes back the updated counter value, the thief reads T and proceeds as in the original THE algorithm (Lines 118- 127).

THEP algorithm correctness The safety of the THEP algorithm follows from the safety of the FF-THE variant (§ 4). The remaining issue is whether waiting for a worker to arrive while holding the queue lock can introduce deadlock. To see why this cannot happen, notice that the worker tries to acquire the queue's lock if $T < H$. A waiting thief eventually notices this and returns EMPTY, releasing the lock.

Why not use only echoes? We could still obtain a fence-free algorithm by *always* blocking the thief until it sees the worker's echo. However, this would harm the load balancing properties of the THE algorithm. For example, suppose there is one worker whose queue contains W tasks of unit length, and $W-1$ thieves. Then if thieves always block, completing all tasks would take $\approx W/2$ time units whereas the original THE takes one time unit. In contrast, reasoning about the worker's buffered stores allows a thief to steal without blocking when the queue contains $> \delta$ tasks, enabling completion of all tasks in $\approx \delta/2$ time units.

Chase-Lev algorithm Unlike the THE algorithm, the Chase-Lev algorithm is *nonblocking* [21]. In particular, a thief running alone always completes its operation. The echo method is inherently blocking as it may prevent a thief from completing until the worker takes steps. It is thus not applicable to the Chase-Lev algorithm since it would destroy its nonblocking progress property.

6. Sidestepping the laws of order

Having described our fence-free algorithms, we now pinpoint how they get around Attiya et al.'s "laws of order" impossibility result [10].

Impossibility result The "laws of order" theorem states that any linearizable [24] implementation of a *strongly non-commutative* (SNC) method must use an atomic operation or memory fence [1] in some execution. The sequential specification of the implemented data structure determines whether a method is SNC. Method M is SNC if there is another method M' (possibly the same method as M) such that applying M followed by M' from some initial state ρ yields different outputs for both M and M' than applying M' followed by M.

`take()` ***and*** `steal()` ***are SNC*** Consider the state ρ in which the work stealing task queue contains one task, x. When applying `take()` first it returns x and a subsequent `steal()` returns EMPTY. Similarly, if `steal()` is applied first it returns x and then `take()` returns EMPTY when applied. It is easy to see that ρ is the only state from which `take()` and `steal()` can influence each other in this way.

[1] The actual theorem statement does not mention memory fences, as it uses a sequentially consistent system model. Instead, the theorem states that either an atomic operation or a read-after-write (RAW) pattern must be used, where a RAW means a write to shared variable X is followed by a read to another shared variable Y without a write to Y in between. However, TSO requires issuing a memory fence after the write to X to prevent the read of Y from being reordered before it.

Tightness assumption The "laws of order" proof assumes that the concurrent implementation is *tight* – that any sequential execution which complies with the specification can occur in a sequential execution of the implementation. (This is referred to as "Assumption 1" in the paper [10].) The proof needs this assumption to argue that an execution exhibiting the strong non-commutativity of a method actually occurs in the implementation. Our algorithms break the tightness assumption.

Violating tightness by relaxing semantics The FF-THE and FF-CL algorithms refuse to steal when the queue contains one task, because there might be a `take()` of this task hidden in worker's store buffer. Instead, they return `ABORT` without changing the state of the queue – which the standard work stealing specification does not allow. Thus, reasoning about the store buffer enables implementing a relaxed specification that allows `steal()` to run first from the ρ state without affecting the return value of a later `take()`.

Violating tightness by blocking In the THEP algorithm a `steal()` invoked when the queue contains one task blocks and does not return until `take()` is invoked. This prevents the SNC execution in which `steal()` affects the return value of `take()` from occurring without relaxing the work stealing semantics, as the `steal()` would simply not terminate when running alone. THEP sidesteps the impossibility result by leveraging a property of work stealing clients (programs): that the worker *keeps taking tasks until the queue empties*. Therefore, in actual work stealing clients, indefinite blocking of a thief does not occur.

7. Bounded TSO in mainstream processors

Here we show that the mainstream TSO architectures – x86 and SPARC – implement bounded TSO, except for a corner case in which there are consecutive stores to the same location, which can be prevented in software.

We use Intel's Xeon E7-4870 (Westmere-EX) processor as a running example, since it is representative of mainstream out-of-order TSO processors. (Similar but simpler reasoning applies to in-order processors.) First, we explain how the processor's implementation of TSO leads to bounded reordering except when consecutive stores to the same location are *coalesced* (§ 7.1). Then we derive the exact bound on the amount of reordering, and show how to adjust the work stealing runtime to avoid store coalescing (§ 7.2-7.3).

7.1 Cause of bounded store/load reordering

To hide the latency of writing to the memory subsystem (which may include resolving a cache miss) the processor retires a store instruction from the reorder buffer without waiting for its value to reach the memory subsystem (henceforth simply "memory"). Instead, the processor holds the instruction's target address and data in a *store buffer entry*, from

```
for S = 1,2,...
  T0 = get_cycle_count()
  repeat K times
    store to location #1
    store to location #2
      ...
    store to location #S
    long latency instruction sequence
  T1 = get_cycle_count()
  output ⟨S, (T1 − T0)/K⟩
```

Figure 6: Code for determining store buffer's capacity.

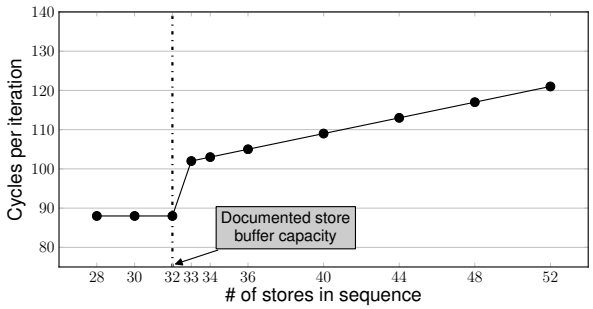


Figure 7: Measuring store buffer capacity on a Westmere-EX processor which has a documented 32-entry store buffer.

which it moves the data to memory as a background task once the store retires [2, 5, 6].

Store buffering makes store/load reordering possible because a load can retire, having read from memory, before the value of an earlier store to a different location gets written to memory. In fact, this is the only way reordering may happen. Out-of-order execution does not lead to further store/load reordering because, to maintain TSO, the processor retires a load only if the value it read remains valid at retirement time [5, 25].

The processor has a fixed number, S, of store buffer entries. However, this does *not* automatically imply that it implements a TSO[S] memory model, because the processor's store buffer is not equivalent to the store buffer of the abstract TSO[S] machine. To show that the processor implements TSO[S], we need to show that a load instruction cannot be reordered with more than S prior stores.

The reason for such bounded reordering is the *implementation* of store buffering, which assigns a store buffer entry to a store when it *enters* the pipeline and *prevents* it from entering the pipeline if the store buffer is full (i.e., all S entries have not been written to memory) [2, 5, 6]. In such a case, the entire execution stalls since later instructions also cannot enter the pipeline as pipeline entry occurs in program order.

It thus appears that a load cannot be reordered past more than S prior stores, conforming to the TSO[S] model. However, if the processor *coalesces* multiple stores to the same location into one store buffer entry, then the S store buffer entries will represent more than S stores and violate this reasoning. We ignore this issue for now and address it in § 7.3.

7.2 Measuring store buffer capacity

This section shows how to empirically determine the capacity of the processor's store buffer. The idea is to measure the time it takes to complete sequences of stores of increasing length, and find the spot at which execution starts to stall.

Measurement algorithm (Figure 6) We alternate between issuing a sequence of stores and a sequence of non-memory instructions whose execution latency is long enough to drain the store buffer. As long as the length of the store sequence does not exceed the store buffer capacity, both the execution of the stores and the flushing of their store buffer entries occur in parallel to the execution of the non-memory instructions, due to out-of-order execution. Consequently, the latency of the non-memory instruction sequence dominates the entire execution time.

However, when the length of the store sequence exceeds the store buffer capacity, the resulting stalls delay the subsequent instruction sequences from entering the pipeline and increase execution time. Importantly, stalls in the $k+1$-th iteration do not overlap the execution of the non-memory instructions in the k-th iteration, and so the stalls are not absorbed by the latency of the non-memory instructions and affect every iteration. The reason is that stores in the $k+1$-th sequence can start draining to memory only after all non-memory instructions in the k-th sequence retire, since store buffer entries are flushed post-retirement and instructions retire in program order.

Figure 7 shows the results of running the measurement algorithm on the Westmere-EX processor. Our measurement results match Intel' documented store buffer capacity for this processor [5, 6]. Measurement results on a Haswell processor are similarly accurate, correctly identifying the documented capacity of 42 [5, 6].

7.3 From store buffer capacity to a reordering bound

Knowing the capacity of the processor's store buffer does not necessarily provide a bound on store/load reordering. For example, if the processor coalesces multiple stores to the same location into a single store buffer entry, the S store buffer entries can represent more than S stores and thus allow reordering beyond S prior stores. There may also be other implementation issues that affect the reordering bound.

Therefore, we develop a *litmus test* that, given a presumed bound S on the reordering, can be used to prove that the processor *violates* the TSO[S] model. We can then gain confidence that a processor implements TSO[S] if extensive testing does not show a violation of the model, although testing will never *prove* that the processor implements TSO[S].

The test program (Figure 9) runs a worker and thief who concurrently try to empty an FF-THE queue (Figure 3) that initially contains 512 tasks. The queue uses a user-supplied

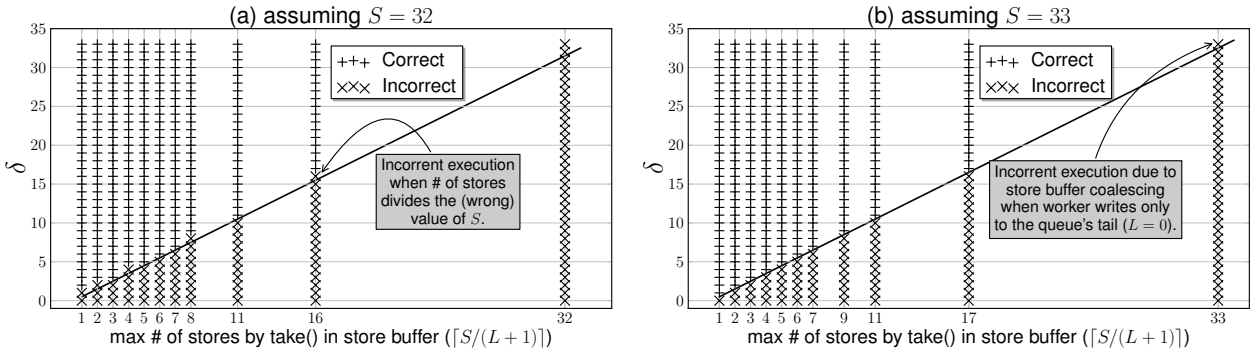


Figure 8: Testing if the processor implements TSO[S]. If S is the correct bound, executions on or above the line $y = x$ should be correct.

Initially: initialize work stealing queue with 512 items.

worker	**thief**
taken := 0	stolen := 0
while (**take**() $\neq$ EMPTY) {	**while** (**steal**() $\neq$ ABORT) {
taken := taken + 1	stolen := stolen + 1
store to location #1	}
...	
store to location #L	
}	

Finally: **if** (taken + stolen = 512) output `CORRECT`, **else** output `INCORRECT`

Figure 9: Program for finding executions not valid under a bounded TSO model. We vary the δ parameter of our work stealing algorithm and L, the number of stores performed by the worker.

value for δ (the maximum number of `take()`s not visible in memory due to store buffering). The worker performs a sequence of L stores to distinct locations between each invocation of `take()`, where L is a program parameter. The worker performs no other stores except for these and the single store in `take()`. Therefore, if the processor implements TSO[S], an execution with $\delta \geq \lceil \frac{S}{L+1} \rceil$ should always be *correct* – the total number of tasks removed by the worker and the thief equals the initial size of the queue.

We run the litmus test using various values of L and δ. For each pair (L, δ), we perform 10^7 runs with each of the following assignments of threads to cores: (1) default OS placement, (2) both threads placed on hyperthreads of the same core, and (3) each thread placed on a different core.

We interpret the litmus test results in light of a supposed ordering bound S. For each $\alpha \in \{1, \ldots, S\}$ we consider all pairs (L, δ) such that $\lceil \frac{S}{L+1} \rceil = \alpha$. If any run of such a pair is incorrect, we consider the point (α, δ) *incorrect*; otherwise, we consider it *correct*. Figure 8a depicts the results when $S = 32$, the processor's store buffer capacity. Interestingly, the processor fails to implement TSO[32], as demonstrated by the incorrect executions when the maximum number of stores between `take()` operations divides 32.

We instrumented the litmus test to understand these failures, and observed that when they occur there always appear to be 33 worker stores in flight and never more. Therefore, we analyzed the litmus test data using $S = 33$ (Figure 8b). This time there is an almost perfect match with the model, with one exception: when $L = 0$. In this case the only stores the worker performs are to the tail of the work stealing queue, T. The T variable is then always at the tail of the store buffer, and in fact is the *only* store pending in the store buffer. In such a case, the processor apparently coalesces stores.

Store buffer coalescing Coalescing under TSO can take place only for consecutive stores, otherwise – as the following example shows – the TSO guarantees may be violated:

initially $A = B = 0$

buffered stores:	$A := 1$	(earliest store)
	$B := 1$	
	$A := 2$	(latest store, entering buffer)

If next $A := 2$ is coalesced with $A := 1$ and this entry is then written to memory, another processor can now observe $A = 2$ while $B = 0$ which is illegal under TSO.

To understand the processor's coalescing implementation, we repeat the store buffer capacity benchmark (Figure 6) using sequences of stores to the *same* location – and still obtain the *same results*. This shows that coalesced stores still get assigned distinct store buffer entries, so coalescing is done at a later execution stage. We hypothesize that the oldest store gets written to memory by first moving its address and data into an additional buffer B, which frees its store buffer entry, and then writing B to the memory subsystem. If the oldest store is to the same address that B holds, B is overwritten with its data, resulting in coalescing. Otherwise, B is overwritten only after its data is written to memory.

Deriving the reordering bound of 33 Because B holds a retired store's data, it *observably behaves as an additional store buffer entry*. Therefore, if we avoid store buffer coalescing, we can assume we are running on a TSO[33] ma-

chine (since our concern is the observable store buffer capacity, i.e., the bound on the reordering). To avoid coalescing of stores by `take()` in work stealing, we need to prevent consecutive invocations of `take()` *with no store in between*. This is easy to do in practice: The CilkPlus `take()` already writes to the structure of a removed task before returning it, and thus avoids coalescing. Other runtimes can similarly perform an extra store before returning from `take()`. Performing an additional store generalizes to processors with coarser coalescing granularity: for example, if a processor coalesces stores to the same cache line, we need to write to another cache line before returning from `take()`.

8. Evaluation

This section evaluates the performance impact of applying our techniques in the THE and Chase-Lev work stealing algorithms. In addition, we compare our techniques to the idempotent work stealing queues of Michael et al. [31], which avoid the worker's memory fence at the cost of relaxing the queue's safety by allowing a task to be removed more than once. Thus, we seek to understand whether comparable performance can be obtained without compromising safety.

Platform We run our experiments on Intel Haswell and Westmere-EX processors. The Westmere-EX (Xeon E7-4870) processor has 10 2.4 GHz cores, each multiplexing 2 hyperthreads. The Haswell (Core i7-4770) processor has 4 3.4 GHz cores, each with 2 hyperthreads. We measure a reordering bound of $S = 33$ on the Westmere-EX and $S = 43$ on the Haswell (ignoring coalescing, which we avoid in software; see § 7.3).

8.1 The FF-THE and THEP algorithms

We implement our techniques in Intel's CilkPlus runtime library (Build 3365, released on May 2013) which uses the THE algorithm.

To understand the individual contribution of our methods, we evaluate both THEP and the FF-THE variant (§ 4) which refuses to steal in case of uncertainty. By default, both versions use a value of $\delta = \lceil \frac{S}{2} \rceil$ derived from the fact that the CilkPlus runtime performs an additional store after each `take()`. To measure the impact of δ, we also test $\delta = 4$, which we determine to be safe by accounting for the program stores and compiler register spills between `take()`s in the benchmark binaries. Finally, we benchmark THEP with $\delta = \infty$, i.e., with thieves always waiting for the worker.

Methodology We measure the running time of a set of 11 CilkPlus programs (Table 1), which have by now become standard benchmarks in the literature on work stealing and task parallel runtimes [7, 19, 20, 26, 28]. We use the `jemalloc` [18] memory allocator to prevent program memory allocation from being a bottleneck (the runtime uses its own memory allocator). Both runtime and programs are compiled version 13.1.1 of Intel's `icc` compiler. We run each program 10 times (except for `knapsack`, which we run 50 times) and report the median run time, normalized to the default CilkPlus run time, as well as 10-th and 90-th percentiles.

Benchmark	Description	Input size
Fib	Recursive Fibonacci	42
Jacobi	Iterative mesh relaxation	1024×1024
QuickSort	Recursive QuickSort	10^8
Matmul	Matrix multiply	1024×1024
Integrate	Recursively calculate area under a curve	10000
knapsack	Recursive branch-and-bound knapsack solver	32 items
cholesky	Cholesky factorization	4000×4000, 40000 nonzeros
Heat	Heat diffusion simulation	4096×1024
LUD	LU decomposition	1024×1024
strassen	Strassen matrix multiply	4096×4096
fft	Fast Fourier transform	2^{26}

Table 1: CilkPlus benchmark applications.

Results Figure 10 shows the results using the maximum level of parallelism on each platform without hyperthreading (i.e., 10 threads on the Westmere-EX, one assigned to each core, and 4 threads on the Haswell). Due to space constraints we omit the figures depicting results with hyperthreading enabled, but we summarize the findings below.

On the Westmere-EX (Figure 10a) the THEP algorithm improves the run time on 8 of the benchmarks by up to 23% and by 11% on (geometric) average, and degrades the remaining 3 programs by 3%. The average improvement across the entire suite is by 7%. On the Haswell (Figure 10b), the run time of 9 programs improves by up to 23% (13% on average) and is not affected on the rest. The average improvement over the entire suite is by 11%.

Varying δ does not significantly impact the performance of the THEP variants on most programs, as stealing is infrequent enough that waiting for the worker does not make much of a difference. However, for `Heat` on the Westmere-EX introducing more stealing opportunities resolves the run time degradation caused by THEP.

In contrast, FF-THE is very sensitive to δ. On 6 programs the default δ prevents FF-THE from stealing altogether and makes the programs run at single threaded speed. Decreasing δ to a more precise value resolves the problem in all programs but `LUD`, showing that the ability to resolve a thief's uncertainty can be important.

When using hyperthreading the processor can schedule one hyperthread when its sibling stalls due to a memory fence. Consequently, the improvements from avoiding fences are reduced. On the Westmere-EX, the average improvement of programs that improve drops to 4% and is at most 12%, while the degradation increases to 5%, yielding an average improvement of 3.6% across the whole suite. The

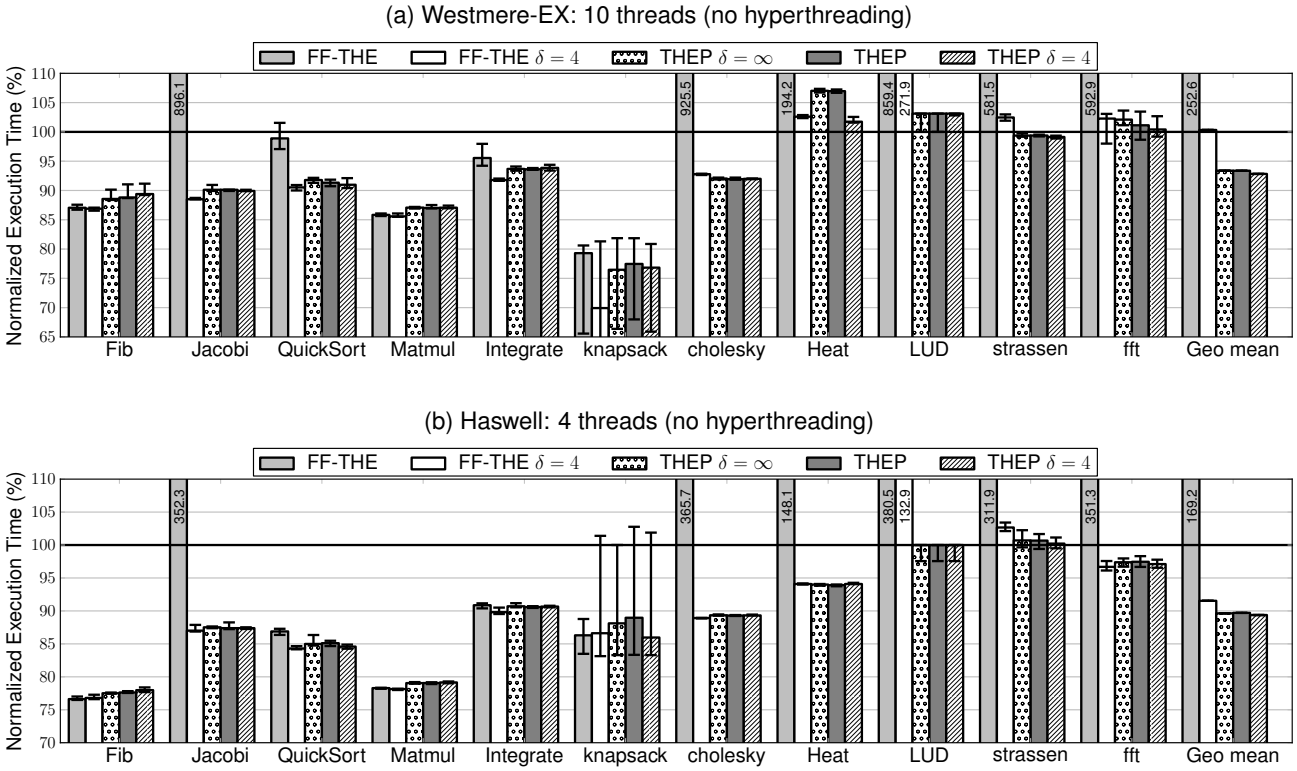


Figure 10: CilkPlus programs run time, normalized to the default CilkPlus runtime.

effect on the Haswell is similar, with overall improvement dropping to 7% (and at most 12%).

8.2 FF-CL vs. idempotent work stealing queues

Here we compare FF-CL to Michael et al.'s idempotent work stealing queues [31]. (Since our goal is not to evaluate how well Chase-Lev does compared to THE, we do not test the THE variants in these experiments.) We test the LIFO and double-ended FIFO idempotent queues. The LIFO queue is a stack in which both worker and thieves remove (possibly the same) tasks from the top of the stack. In the double-ended FIFO queue, the last task can be removed concurrently by both the worker and a thief.

We use Michael et al.'s benchmark programs and inputs, but we implement the idempotent task queues ourselves as their code is not publicly available. (However, our performance results match those of Michael et al. [31].) There are two benchmarks, computing the transitive closure and the spanning tree of a graph. The parallel algorithms used [15] manage synchronization internally, because the same task (e.g., "visit node u") can inherently be *repeated* by different threads (e.g., who are working on different neighbors of u). We report results only for the transitive closure; spanning tree results are similar.

The input graphs consists of: (1) a K-graph, which is a K-regular graph in which each node is connected to K nodes, (2) a random graph of n nodes and m edges, and (3) a two-dimensional torus (grid).

We run the transitive closure program 10 times on each input, using the maximum level of parallelism usable by the workload, both with and without hyperthreading. For the K-graph and random graph, this is the maximum parallelism in the machine, but on the torus graph the programs do not scale past 2 threads, so we report results from 2 threads.

Figure 11a depicts the Haswell results for the transitive closure application with large inputs without hyperthreading. (Due to space constraints, we omit hyperthreading and Westmere-EX results, which are similar.) We show median, 10-th and 90-th percentile run times, normalized to the standard Chase-Lev algorithm.

All the fence-free work stealing queues obtain comparable performance. The torus input enjoys the greatest improvement in running time, $\approx 33\%$ for both our FF-CL and the LIFO idempotent queue.

In contrast to the CilkPlus experiments (Figure 10), here the default δ value does not prevent a thread from stealing. This can be seen by observing the non-zero percent of stolen work for our variant in Figure 11b.

Also apparent in Figure 11b is that the vast majority of work is performed by the worker and not by thieves, thus emphasizing the importance of removing overhead – e.g., the memory fence – from the worker's code path.

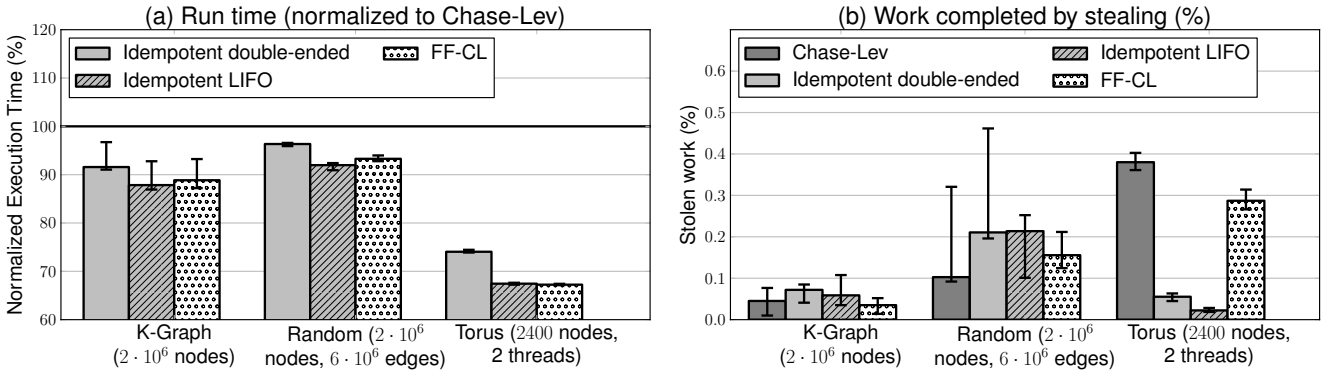


Figure 11: Transitive closure at maximum parallelism on Haswell (no hyperthreading).

9. Related work

Fence-free work stealing task queues Michael et al.'s idempotent work stealing algorithms [31] avoid worker fences, but are only applicable to applications that can tolerate a task being executed twice, whereas our techniques are relevant to any application on a bounded TSO processor. Kumar et al. [26] use yieldpoint mechanisms to stop the worker at a known-safe location before stealing. Such mechanisms are not available in unmanaged environments such as C/C++, whereas our technique applies there. Dice et al.'s asymmetric synchronization [16] can be used to eliminate memory fences in work stealing. However, this requires heavyweight actions by the thief (e.g., suspending the worker thread) whereas our approach is lightweight.

Eliminating fence penalty in hardware Below we describe microarchitectural designs that eliminate the penalty of fences, thereby obviating the need for our fence-free techniques. In contrast to all these proposals, our algorithms offer an *immediately usable software-only solution* for mainstream multicore architectures available today.

Speculative memory fences Store-wait-free processing [36] and Invisifence [12] use speculation to eliminate the penalty of memory fences. Instead of a fence stalling the processor until all prior stores are written to memory, these designs initiate transactional memory [22] style speculative execution which commits when all prior stores have been drained to memory. However, this speculation may interact badly with work stealing, as each time a thief reads the queue's tail it might abort the worker's speculative execution, which can contain several `take()`s.

Stalling fences only when needed In WeeFence [17] and address-aware fences [29] a fence stalls the processor only if a post-fence access is about to violate the memory model. Thus, a steal attempt can stall the worker, whereas with our techniques – which appear to be applicable to these designs when fences are not used – steals do not affect the worker.

Multiple store buffers Singh et al. [35] propose to use different store buffers for shared and private memory locations. Fences then only need to drain the shared-location store buffer. However, the processor needs to distinguish between private and shared accesses, which requires compiler and instruction set changes, or extending the hardware memory management unit and page table structures [35].

10. Conclusion and future work

This paper shows that mainstream TSO processors only allow *bounded* store/load reordering and that this can be exploited to derive fence-free work stealing algorithms. The idea is that we can compensate for reading stale values from memory by reasoning about the number of stores that are hidden in the store buffer.

More generally, because our approach enables circumventing the recent "laws of order" impossibility result [10] by violating its *tightness* assumption, we hope it opens the door for removing memory fences in other concurrent algorithms.

The notion of a memory model with bounded reordering raises several questions for future research. How does bounded reordering extend to weaker memory models which admit other forms of reordering beyond store/load? Do mainstream implementations of weak memory models such as PowerPC and ARM also exhibit forms of bounded reordering? Finally, it will be interesting to explore microarchitectures that *explicitly* provide bounds on reordering.

Acknowledgments

We thank our shepherd, Martin Vechev, and the ASPLOS reviewers for their insightful comments which helped to considerably improve the presentation of this paper.

This work was supported by the Israel Science Foundation (grants 1386/11 and 1227/10) 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] UltraSPARC T1 Supplement to the UltraSPARC Architecture 2005. `http://www.oracle.com/technetwork/systems/opensparc/t1-08-ust1-uasuppl-draft-p-ext-1537736.html`, March 2006.

[3] Intel CilkPlus Language Specification. Technical report, Intel Corporation, 2011.

[4] Intel Threading Building Blocks. `http://threadingbuildingblocks.org/`, June 2012.

[5] 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.

[6] 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.

[7] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 219–228, New York, NY, USA, 2013. ACM.

[8] Samy Al Bahra. Nonblocking algorithms and scalable multicore programming. *Communications of the ACM*, 56(7):50–61, July 2013.

[9] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34:115–144, 2001.

[10] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 487–498, New York, NY, USA, 2011. ACM.

[11] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, March 2009.

[12] Colin Blundell, Milo M.K. Martin, and Thomas 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.

[13] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[14] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM.

[15] Guojing Cong David A. Bader. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 65(9):994–1006, 2005.

[16] Dave Dice, Hui Huang, and Mingyao Yang. Asymmetric Dekker Synchronization. `http://home.comcast.net/~pjbishop/Dave/Asymmetric-Dekker-Synchronization.txt`, 2001.

[17] Yuelu Duan, Abdullah Muzahid, and Josep 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.

[18] Jason Evans. Scalable memory allocation using jemalloc. `http://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919`, 2011.

[19] Karl-Filip Faxen. Efficient work stealing for fine grained parallelism. In *Proceedings of the 2010 39th International Conference on Parallel Processing*, ICPP '10, pages 313–322, Washington, DC, USA, 2010. IEEE Computer Society.

[20] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the 19th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

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

[22] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

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

[24] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12:463–492, July 1990.

[25] David Kanter. Haswell Transactional Memory Alternatives. `http://www.realworldtech.com/haswell-tm-alt/`, August 2012.

[26] Vivek Kumar, Daniel Frampton, Stephen M. Blackburn, David Grove, and Olivier Tardieu. Work-stealing without the baggage. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 297–314, New York, NY, USA, 2012. ACM.

[27] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

[28] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 411–420, New York, NY, USA, 2010. ACM.

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

[30] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 429–440, New York, NY, USA, 2012. ACM.

[31] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 45–54, New York, NY, USA, 2009. ACM.

[32] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[33] James Reinders. *Intel Threading Building Blocks*. O'Reilly Media, July 2007.

[34] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus 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] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal 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] Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas 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.