Correct and Efficient Work-Stealing for Weak Memory Models

Nhat Minh Lê, Antoniu Pop, Albert Cohen, Francesco Zappa Nardelli

To cite this version:

HAL Id: hal-00802885
https://inria.hal.science/hal-00802885
Submitted on 20 Mar 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Correct and Efficient Work-Stealing for Weak Memory Models

Nhat Minh Lê Antoniu Pop Albert Cohen Francesco Zappa Nardelli
INRIA and ENS Paris

Abstract
Chase and Lev’s concurrent deque is a key data structure in shared-memory parallel programming and plays an essential role in work-stealing schedulers. We provide the first correctness proof of an optimized implementation of Chase and Lev’s deque on top of the POWER and ARM architectures: these provide very relaxed memory models, which we exploit to improve performance but considerably complicate the reasoning. We also study an optimized x86 and a portable C11 implementation, conducting systematic experiments to evaluate the impact of memory barrier optimizations. Our results demonstrate the benefits of hand tuning the deque code when running on top of relaxed memory models.

Categories and Subject Descriptors D.1.3 [Programming Techniques]: Concurrent Programming; E.1 [Data Structures]: Lists, stacks, and queues

Keywords: lock-free algorithm, work-stealing, relaxed memory model, proof

1. Introduction
Multicore POWER and ARM architectures are standard targets for server, consumer electronics, and embedded control applications. The difficulties of parallel programming are exacerbated by the relaxed memory models implemented by these architectures, which allow the processors to perform a wide range of optimizations, including thread-local reordering and non-atomic store propagation.

The safety-critical nature of many embedded applications call for solid foundations for parallel programming. This paper shows that a high degree of confidence can be achieved for highly optimized, real-world, concurrent algorithms, running on top of weak memory models. A good test-case is provided by the runtime scheduler of a task library. We thus focus on the Chase and Lev’s concurrent doubly-ended queue (deque) [3], the cornerstone of most work-stealing schedulers. Until now, no rigorous correctness proof has been been provided for implementations of this algorithm running on top of a relaxed memory model. Furthermore, while work-stealing is widely used on the x86 architecture (an evaluation under a restrictive hypothesis of idempotence of the workload can be found in [10]), few experiments target weaker memory models.

Our first contribution is a correctness proof of this fundamental concurrent data structure running on top of a relaxed memory model. We provide a hand-tuned implementation of the Chase and Lev’s deque for the ARM architectures, and prove its correctness against the memory semantics defined in [12] and [7]. Our second contribution is a systematic study of the performance of several implementations of Chase–Lev on relaxed hardware. In detail, we compare our optimized ARM implementation against a standard implementation for the x86 architecture and two portable variants expressed in C11: a reference sequentially consistent translation of the algorithm, and an aggressively optimized version making full use of the release–acquire and relaxed semantics offered by C11 low-level atomic. These implementations of the Chase–Lev deque are evaluated in the context of a work-stealing scheduler. We consider diverse worker/thief configurations, including a synthetic benchmark with two different workloads and standard task-parallel kernels. Our experiments demonstrate the impact of the memory barrier optimization on the throughput of our work-stealing runtime. We also comment on how the ARM correctness proof can be tailored to these alternative implementations. As a side effect, we highlight that our optimized ARM implementation cannot be expressed using C11 low-level atomic, which invariably end up inserting one redundant synchronization instruction.

2. Chase–Lev deque
User-space runtime schedulers offer an excellent playground for studying low-level high-performance code. We focus on randomized work-stealing: it was originally designed as the scheduler of the Cilk language for shared-memory multiprocessors [4], but thanks to its merits [2] it has been adopted in a number of parallel libraries and parallel programming environments, including the Intel TBB and compiler suite. Work-stealing variants have also been proposed for distributed clusters [5] and heterogeneous platforms [1]. The scheduling strategy is intuitive:
• Each processor uses a dynamic array as a deque holding tasks ready to be scheduled.
• Each processor manages its own deque as a stack. It may only push and pop tasks from the bottom of its own deque.
• Other processors may not push or pop from that deque; instead, they steal tasks from the top when their own deque is empty. In most implementations, the stolen deque is selected at random.
• Initially, one processor starts with the “root” task of the parallel program in its deque, and all other deques are empty.

The state-of-the-art algorithm for the work-stealing deque is Chase and Lev’s lock-free deque [3]. It uses an array with automatic, asynchronous growth. Assuming sequentially consistent memory, it involves only one atomic compare-and-swap (CAS) per steal, no CAS on push, and no CAS on take except when the deque has exactly only one element left.

We implemented and tested four versions of the concurrent deque algorithm, with different barrier configurations: (1) a sequentially consistent version, written with C11 seq. cst atomic, following the original description in [3]; (2) an optimized version, which takes full advantage of the C11 relaxed memory model, reported in Figure 1; (3) a native version for ARMv7, reported in Figure 2,
and (4) a native version for x86. These native versions rely on compiler intrinsics and inline assembly to leverage architecture-specific assumptions and thus reduce the number of barriers required.

In our implementations of Figure 1 and Figure 2, we assume that the Deque type is declared as:

```c
typedef struct {
    atomic_size_t size;
    atomic_size_t top, bottom;
    atomic_int buffer[];
} Array;
typedef struct {
    Atomic<Array *> array;
} Deque;
```

In the code of Figure 1 the atomic_ and memory_order_ prefixes have been elided for clarity. The ARMv7 pseudo-code of Figure 2 uses the keywords R and W to denote reads and writes to shared variables, and atomic indicates a block that will be executed atomically, implemented via LL/SC instructions. The x86 version is based on prior work [10] and only requires a single mfence memory barrier in take, in place of the call to thread_fence in the C11 code.

### 2.1 Notions of correctness

The expected behavior of the work-stealing deque is intuitive: tasks pushed into the deque are then either taken in reverse order by the same thread, or stolen by another thread. We say that an implementation is correct if it satisfies four criteria, formalized and proven correct for our ARMv7 optimized code in Section 4:

1. tasks are taken in reverse order;
2. only tasks pushed are taken or stolen (well-defined reads);
3. a task pushed into a deque cannot be taken or stolen more than once (uniqueness);
4. given a finite number of push operations, all pushed values will eventually be either taken or stolen exactly once, if enough take and steal operations are attempted (existence).

These criteria hold because of the following assumptions and properties of the Chase–Lev algorithm:

- For any given deque, push and pop operations execute on a single thread. Concurrency can only occur between one execution of push or take in the owner thread, and one or more executions of steal in different threads.
- Newly pushed tasks are made visible to take and steal by the increment to bottom in push. As we shall see in Section 4, our ARMv7 implementation enforces this by placing a sync barrier before the update of bottom, guaranteeing that the pushed element can not be stolen before bottom is updated.
- Taken tasks are reserved first by updating bottom; again, in our ARMv7 code, the sync barrier placed after the update to bottom will ensure that it will not be concurrently stolen.
- Stolen tasks are reserved by updating top. The only situation where steal and take contend for the same task is when the deque has a single element left; this particular conflict is resolved through the CAS instructions in both take and steal. This scenario allowed Chase and Lev to make the CAS in take conditional upon the size of the deque being 1. The correctness of this optimization on a relaxed memory model depends on the presence of the two full barriers in take and steal, to ensure that at least one of the participants will have a consistent view of the size of the deque. Having just one take or steal seeing a consistent view of the size of the deque is enough: if it is take, that will force a CAS to be performed; if it is steal, the index reservation will ensure an empty return value.
- Finally, stolen tasks are protected from being concurrently stolen multiple times by the monotonic CAS update to top in steal. This CAS orders steal operations and makes them mutually exclusive. At the same time, steal operations that abort due to a failed CAS do not change the state of the deque.

### 2.2 Comparison of the C11 and ARM Implementations

Our C11 implementation in Figure 1 is optimal in the sense that no C11 synchronization can be removed without breaking the algorithm. However, if low-level atomics are compiled using the mapping of McKenney and Silvera [9] on ARMv7/POWER or the mapping of Tehrekov [14] on x86, the generated code contains more barriers than the hand-optimized native versions on both x86 and ARMv7. We show in Section 5 that this happens because of the need for seq.cst atomics to simulate ARMv7/POWER cumulative semantics. Concretely, on ARMv7, an extra dmb instruction is inserted before each CAS operation [11], compared to the native version where a relaxed CAS—coherent and atomic only—is sufficient. On x86, an mfence instruction is added between the two reads in steal. The fully sequentially consistent C11 implementation inserts many more redundant barriers [11].

### 3. The memory model of ARMv7

The memory model of the ARMv7 architecture follows closely that of the POWER architecture, allowing a wide range of relaxed behaviors to be observable to the programmer.

1. The hardware threads can each perform reads and writes out-of-order, or even speculatively. Basically any local reordering is allowed unless there is a data/control dependence or synchronization instruction preventing it.
2. The memory system does not guarantee that a write becomes visible to all other hardware threads at the same time point. Writes performed by one thread are propagated to (and become visible from) any other thread in an arbitrary order, unless synchronization instructions are used.
3. A dmb barrier instruction guarantees that all the writes which have been observed by the thread issuing the barrier instruction are propagated to all the other threads before the thread can continue. Observed writes include all writes previously issued by the thread itself, as well as any write propagated to it from another thread prior to the barrier. This semantics of barrier instructions is referred to as cumulative.

We build on the axiomatic formalization of POWER and ARMv7 memory model by Mador-Haim et al. [7], which has been proved equivalent to the operational semantics of Sarkar et al. [12]. A gentle introduction can be found in [8].

Axiomatic execution witnesses capture abstract memory events associated with memory-related instructions and internal transitions of the model. Unlike in stronger models such as x86, each memory access is represented at run-time by two distinct events: an issuing event—called set for reads and init for writes—eventually followed by a commit event when the speculative state of the instruction is resolved. Once a write instruction is committed, events that propagate it to other threads can be observed—propagation to thread A is denoted ppA. All the relations part of an execution witness are listed in Table 1.

The core of the axiomatic model builds on the evord relation, modeling the happens-before order between events. This satisfies the fundamental property:

```
word \(\rightarrow\) after \(\cup\) before \(\cup\) comm \(\cup\) mem \(\cup\) local
```

and must be acyclic for an execution to be consistent.

We assume that the atomic sections, used to represent CAS-like behaviors, are executed atomically and obey a total order. We model them either as a single instance of a read instruction (failed CAS) or an atomic read–write pair of instruction instances (successful CAS). The atomicity of these accesses is captured by the pown relation. We do not assume any other property on these atomic sections (e.g., cumulativity). In practice, atomic sections can be implemented with LL/SC instructions.

We use several notation shortcuts. We refer to the deque global variables top, bottom, and array as t, b, and a. Elements of the buffer are written \(x_i\), where \(i\) is the virtual index in natural numbers.
before any wrap-around is applied. Barrier instructions are omitted for brevity when implied by the presence of a \_or\_ctrl-isync relation. Irrelevant values in reads and writes are replaced with the placeholder \_ (e.g., \_Rx\_). We do not label instruction instances individually, but decorate them with a disambiguating execution prefix, identified by a dot. These prefixes do not only distinguish between instruction instances, but also group related instruction instances within a same execution unit (usually an invocation of one of \push, \take or \steal). For this, when no prefix is specified, the last prefix in left-to-right order is assumed.

4. Proof of correctness of the ARMv7 code

The proof is divided into five parts; it validates the criteria 2 to 4 enumerated in Section 2.1. Since \push and \take never execute concurrently and \b is only ever modified in one of these functions, the proof of Criterion 1 does not involve reasoning about concurrency and we omit it here.

The proof builds on a precise analysis of all the possible execution witnesses of arbitrary invocations of the algorithm. We recall that an execution witness, as defined by the ARMv7 axiomatic model, is a graph capturing all memory events occurring during an execution (vertices), as well as the relations that link them (edges). Individual lemmas strive to narrow down the set of possible execution witnesses, based on properties of the algorithm and the architecture. To that end, we pinpoint specific subgraphs of an execution witness (hereafter, execution graphs) that cannot occur together in the same consistent execution witness. We then show that all incorrect executions, such as those containing two instances of \steal reading the same value added by a single instance of \push, cannot have consistent execution witnesses and, as such, cannot happen.

The proof is structured as follows. In 4.1 we provide basic technical definitions and properties of the memory model, which are used throughout the proof. In 4.2 we describe all the possible execution graphs for each of the three operations (\push, \take and \steal), following the control flow of the ARMv7 code in Figure 2. In 4.3 we show how the succession of dynamic arrays built by resizing can be abstracted as a single sequence of unique abstract values independent of resize operations, with strong coherence and consistency properties. Corollary 2 establishes Criterion 2 (well-defined reads). In 4.4 we build on the previous abstraction to prove Theorem 1, pertaining to the uniqueness of elements taken and stolen, which corresponds to Criterion 3 (uniqueness). Finally, in 4.5, we rely on all previous results to prove Theorem 2 establishing Criterion 4 (existence): the existence of matching \take or \steal operations for every pushed element, under the appropriate hypotheses.

4.1 Preliminary properties

Before delving into the details of the proof itself, we introduce some support definitions and related properties.
Table 1. Summary of relations used in the ARMv7 axiomatic model

<table>
<thead>
<tr>
<th>Relation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sync}(X)$</td>
<td>Memory barrier (usually implied by $\text{ctl}(X)$)</td>
</tr>
<tr>
<td>$\text{pp}(X)$</td>
<td>Propagates to thread of $X$</td>
</tr>
<tr>
<td>$\text{po}(X)$</td>
<td>Program order</td>
</tr>
<tr>
<td>$\text{po-stm}(X)$</td>
<td>Atomic operation in program order (for CAS; see below)</td>
</tr>
<tr>
<td>$\text{po-loc}$</td>
<td>Non-cumulative local ordering barrier (see below)</td>
</tr>
<tr>
<td>$\text{co}(X)$</td>
<td>Communication edge</td>
</tr>
<tr>
<td>$\text{before}(X,Y)$</td>
<td>Before barrier edge</td>
</tr>
<tr>
<td>$\text{intra}(X,Y)$</td>
<td>Intra-instruction order edge</td>
</tr>
<tr>
<td>$\text{local}(X)$</td>
<td>Local order edge</td>
</tr>
<tr>
<td>$\text{read}(X,Y)$</td>
<td>Event happens-before order (usually typeset as $\rightarrow$)</td>
</tr>
</tbody>
</table>

On ARMv7, $\text{sync}(X)$ corresponds to a dmb instruction while $\text{ctl}(X)$ corresponds to a conditional branch followed by an ish instruction.

Lemma 1. The following properties involving $\text{pp}$ apply:

1. If $X_1 \text{pp} X_2$ and $X_2 \text{pp} X_3$, then $X_1 \text{pp} X_3$.
2. If $X_1 \text{pp} X_2$ and $X_2 \text{pp} X_3$, then $X_1 \text{pp} X_3$.

Proof. The case of an address or control dependence, the result is an immediate consequence of the definition of intra-instruction and local orders. It remains to be shown that the result holds for $\text{ctl}(X)$: a dependent conditional branch instruction, $\text{ctl}(X)$, followed by an $\text{sync}(X)$ barrier. Suppose $X_1 \text{pp} X_2$. Then we have:

$$\text{pp}(X_1, X_2) \Rightarrow \text{pp}(X_2, X_3)$$

We define the relation $\text{pp-stm}(X,Y)$ between instruction instances, $A.W_x \text{pp-stm} B.Y_z$, as follows:

$$A.W_x \text{pp-stm} B.Y_z \iff A \text{pp-stm} B$$

Intuitively, $\text{pp-stm}(X,Y)$ represents a "known-to" relation in the following sense: $A.W_x \text{pp-stm} B.Y_z$ means that, at the time of reading $y$, that specific write to $x$ (as well as any write that is coherence-before it) is known to the thread executing $B$. It is clear that $\text{pp-stm}$ does not imply $\text{pp}$, by definition of communication edges (if threads are different) or uniprocessor constraints (if same thread).

Lemma 2. The following properties involving $\text{pp}$ hold:

1. $A.W_x \text{pp} B.Y_z \Rightarrow A.W_x \text{pp} B.Y_z$, $B.Y_z \text{pp} B.Y_z$. $A.W_x \text{pp} B.Y_z \Rightarrow B.Y_z \text{pp} B.Y_z$.
2. $A.W_x \text{pp} B.Y_z \Rightarrow A.W_x \text{pp} B.Y_z$.
3. $A.W_x \text{pp} B.Y_z \Rightarrow B.Y_z \text{pp} B.Y_z$.
4. $A.W_x \text{pp} B.Y_z \Rightarrow A.W_x \text{pp} B.Y_z$.

Proof. We prove each point separately:

(i) If $A.W_x \text{pp} B.Y_z$ happens in the same thread, then all instruction instances belong to that thread, and program order prevails. Otherwise, either $A.W_x \text{pp} B.Y_z$ and the result is immediate, or $A.W_x \text{pp} B.Y_z$. $B.Y_z \text{pp} B.Y_z$, which implies the following:

$$\text{pp}(A.W_x, B.Y_z) \Rightarrow \text{pp}(B.Y_z, B.Y_z)$$

(ii) Suppose $A.W_x \text{pp} B.Y_z$. Then $B.Y_z \text{pp} B.W_y$, and we have the following cycle in the event happens-before order:

$$\text{pp}(B.Y_z, B.W_y) \Rightarrow \text{pp}(B.W_y, B.W_y)$$

Lemma 3. The following properties involving $\text{pp}$ hold:

1. $A.W_x \text{pp} B.Y_z \Rightarrow A.W_x \text{pp} B.Y_z$.
2. $A.W_x \text{pp} B.Y_z \Rightarrow A.W_x \text{pp} B.Y_z$.
3. $A.W_x \text{pp} B.Y_z \Rightarrow A.W_x \text{pp} B.Y_z$.

Proof. We prove each point separately:

(i) If $A.W_x$ and $B.Y_z$ occur in the same thread, then all instruction instances belong to that thread and program order prevails. Otherwise, suppose $A.W_x$ executes in $A$; we have two cases:

$$\text{pp}(A.W_x, \text{sync}(A)) \Rightarrow \text{pp}(A.W_x, \text{sync}(A))$$

(ii) Suppose $A \sim B$. Then $A.W_x \text{pp} B.W_y$, and we have the following:

$$\text{pp}(A.W_x, B.W_y) \Rightarrow \text{pp}(B.W_y, B.W_y)$$

In both cases, $\text{pp}(B.W_y, B.W_y) \Rightarrow \text{pp}(B.W_y, B.W_y)$

(iii) Suppose $A \sim B$. Then $A.W_x \text{pp} B.W_y$, and we have the following:

$$\text{pp}(A.W_x, B.W_y) \Rightarrow \text{pp}(B.W_y, B.W_y)$$

Otherwise, suppose $A \not\sim B$. If $A \sim B$, then $A.W_x \text{pp} B.W_y$, and we have the result from (i). If not, we have:

$$\text{pp}(A.W_x, B.W_y) \Rightarrow \text{pp}(B.W_y, B.W_y)$$

Thus, we have $\text{pp}(A.W_x, B.W_y) \Rightarrow \text{pp}(B.W_y, B.W_y)$

(iii) Suppose the contrary. If $B \sim D$, then $\Rightarrow$ and $\Rightarrow$ form a path that goes against $\text{pp}$, the graph is invalid according to uniprocessor constraints. Otherwise, $B \not\sim D$ and the following holds (omitting intermediate steps in elaborating $\Rightarrow$ for conciseness):

$$\text{com}(B.W_y, D.W_y) \Rightarrow \text{com}(D.W_z, \text{sync}(B.W_y), \text{sync}(D.W_z), \text{sync}(B.W_y), \text{sync}(D.W_z))$$

if $B \not\sim C$. 

Note that $\text{pp-stm}(X,Y)$ does not imply $\text{pp}$, by definition of communication edges (if threads are different) or uniprocessor constraints (if same thread).
We consider the three operations of the work-stealing algorithm: take, push and steal. Each of them exhibits different execution paths depending on control flow. Data and address dependences are implicit in the notations and are omitted for brevity. Control dependences are implied by the guard conditions in each case and are also omitted, but we explicitly the constraints on the \(b\) and \(t\) variables carrying the control dependence. Greek letters \(\beta, \tau, \xi\) denote the memory values of \(b, t\), and some \(x_i\), respectively. Reads and writes are annotated with the corresponding line from Figure 2.

For **take** and **steal**, we say that an instance of the operation is successful if it returns one element; otherwise (including if it returns empty) it is considered failed.

### 4.2 Execution paths

We consider the three operations of the work-stealing algorithm: take, push and steal. Each of them exhibits different execution paths depending on control flow. Data and address dependences are implicit in the notations and are omitted for brevity. Control dependences are implied by the guard conditions in each case and are also omitted, but we explicitly the constraints on the \(b\) and \(t\) variables carrying the control dependence. Greek letters \(\beta, \tau, \xi\) denote the memory values of \(b, t\), and some \(x_i\), respectively. Reads and writes are annotated with the corresponding line from Figure 2.

For **take** and **steal**, we say that an instance of the operation is successful if it returns one element; otherwise (including if it returns empty) it is considered failed.

#### 4.2.1 Take

Two failure cases return no element (empty), and two success cases return one element from the deque. All four paths start with:

\[
(a) Rb, \beta \rightarrow (b) Ra, \& x \rightarrow (c) Wb, \beta - 1 \rightarrow (d) Rt, \tau
\]

Specific continuations for each path are listed below.

**Return empty without CAS**, \(\beta - \tau \leq 0\):

\[
\vdots \rightarrow \psi (i) Wb, \beta
\]

**Return empty with (failed) CAS**, \(\beta - \tau = 1, \tau \neq \tau'\):

\[
\vdots \rightarrow \psi (e) Rx_{\beta - 1}, \xi \rightarrow (f) Rt, \tau \rightarrow (h) Wb, \tau + 1
\]

**Return one without CAS**, \(\beta - \tau > 1\):

\[
\vdots \rightarrow \psi (e) Rx_{\beta - 1}, \xi \rightarrow (f) Rt, \tau \rightarrow (g) Wt, \tau + 1 \rightarrow (h) Wb, \beta
\]

#### 4.2.2 Push

There are two paths: a straight case, and a resizing case which grows the underlying circular buffer.

**Straight**, \(\beta - \tau < \text{size}(x) - 1\):

\[
(a) Rb, \beta \rightarrow (b) Rt, \tau \rightarrow (e) Ra, \& x \rightarrow (f) Wb, \xi \rightarrow (f) Wb, \beta + 1
\]

**Resizing**, \(\beta - \tau \geq \text{size}(x) - 1\): where \(z'\) refers to the new array

\[
(a) Rb, \beta \rightarrow (b) Rl, \tau \rightarrow (e) Ra, \& x \rightarrow \text{resize}
\]

\[
\rightarrow (d) Ra, \& x' \rightarrow (e) Wb', \xi \rightarrow (f) Wb, \beta + 1
\]

where \(\text{resize} = Rx', \xi_p \rightarrow Wx', \xi_p \rightarrow Rz_{\xi_p} \rightarrow Rz_{\xi_p} \rightarrow Rz_{\xi_p}\)

#### 4.2.3 Steal

There are three paths: two failure cases and one success case. Failure returns no element and success returns a stolen element.

**Return empty without CAS**, \(\beta - \tau \leq 0\):

\[
(a) Rl, \tau \rightarrow (b) Rb, \beta
\]

**Return empty with (failed) CAS**, \(\beta - \tau > 0, \tau \neq \tau'\):

\[
(a) Rl, \tau \rightarrow (b) Rl, \beta \rightarrow \text{climbdown}(c) Ra, \& x \rightarrow (d) Rx_{\tau}, \xi \rightarrow (e) Rl, \tau'
\]

**Return one with (successful) CAS**, \(\beta - \tau > 0\):

\[
(a) Rl, \tau \rightarrow (b) Rl, \beta \rightarrow \text{climbdown}(c) Ra, \& x \rightarrow (d) Rx_{\tau}, \xi \rightarrow (e) Rl, \tau \rightarrow (f) Wt, \tau + 1
\]

### 4.3 Significant reads and writes

We define the sequence \(\beta_i\) of values taken by the variable \(b\) over the course of the program, according to the write coherence relation. Initially \(\beta_0 = 0\). Since all push and take operations occur in a single thread, and steal operations never alter the value of \(b\), the elements of \(\beta_i\) correspond to writes to \(b\) in program order within the push and take operations. Similarly, we define the sequence \(\tau_i\) of values taken by the variable \(t\). We assume \(\tau_0 = 0\). Furthermore, since all writes to \(t\) are from CAS instructions, which are sequentially ordered, and all such CAS instructions increment \(t\) by one, \(\tau_i\) is monotonically increasing, and \(\tau_i = m\).

For each index \(i\), we define the sequence \(\xi_i\) of success values given to the element at index \(i\) in the deque by the last write \(W_{x_i}\) of a push operation, regardless of the address \& \(x\) of the underlying array. Only the last such write is called significant as it induces a new value in \(\xi_i\) sequence, while writes due to resizing do not. For all \(i, \xi_i = 0\), the value before the first significant write to \(x_i\) location, is undefined: \(\xi_i = \bot\). Similarly, a read is significant if it occurs in a successful instance of take or steal.

**Lemma 4.** For all \(i, \xi_i\) is globally coherent.

**Proof.** Given two significant writes \(W_{x_i}\) and \(W_{x_j}\), at index \(i\) (regardless of the address of the underlying array). If \(W_{x_i}\) and \(W_{x_j}\) both write to the same memory location, then they are ordered by write coherence. If they do not, then there must be a resize operation after the first write and before the second (all writes happen in the same thread). Because of the cumulative barrier after a resize operation, threads that see the second value must have seen the first beforehand. Hence, there is a global coherence order on the writes, which corresponds to the order of push operations.

We define the relation \(\text{read from} \) as follows: for some memory locations \(m_0, \ldots, m_n\) and some value \(v\), \(W_{m_0} \rightarrow v \rightarrow \cdots \rightarrow R_{m_n} \rightarrow v\) if \(W_{m_0} \rightarrow v \rightarrow \cdots \rightarrow R_{m_n} \rightarrow v\) or there exists a sequence of copies carrying the value of the write to the read:

\(W_{m_0} \rightarrow v_1 \rightarrow \cdots \rightarrow R_{m_n} \rightarrow v_2 \rightarrow \cdots \rightarrow \cdots \rightarrow R_{m_n} \rightarrow v\).

For conciseness, we hereafter omit the variable name from reads and writes whenever the variable can be inferred from the value: e.g., \(W_{\beta_i}\) stands for \(W_{xb}, \beta_i\). Let \(\xi_{i_0}\) denote the \(\xi_{i_0}\) significant write at index \(i\), and \(R_{i_0}\) a significant read s.t. \(W_{i_0} \rightarrow R_{i_0}\).

**Lemma 5.** Given a write \(W_{x_i}\) and a read \(R_{x_j}\),

\[
i \neq j \implies W_{x_i} \rightarrow R_{x_j}
\]

**Proof.** If the addresses of the underlying arrays differ, then the memory locations read and written are distinct and there can be no read from relation.

Otherwise, since old arrays are never reused, the addresses are the same and \(i = j\) mod \(\text{size}(x)\) \(R_{x_j}\) belongs to a successful instance of take, push (with resizing), or steal. Let \(X\) be that instance.

Let \(P\) be the instance of push to which \(W_{x_i} \rightarrow R_{x_j}\) belongs. In \(P\), we have the following execution graph:

\(P; \tau P \rightarrow \text{addr} W_{x_i} \rightarrow Wb, \beta + 1\)

where \(\tau P \leq i \leq \beta P\) and \(\beta P - \tau P < \text{size}(x) - 1\)

Let us assume \(i \neq j \neq W_{x_i} \rightarrow R_{x_j}\), and show it is indeed impossible.

Assume \(X\) is a successful instance of take or push. Since \(X\) and \(P\) belong to the same thread, \(P\) must occur before \(X\) in program order (the order of loads and stores to the same location is preserved:

\(W_{x_i} \rightarrow \text{read} X.R_{x_j}\)).

If \(j < i\), then \(j < \text{size}(x)\). However, the following must hold in \(P\):

\(\tau P \leq i \leq \beta P \land \beta P - \tau P < \text{size}(x) - 1\)

hence \(j < \text{size}(x) + 1 < \beta P - \text{size}(x) + 1 < \tau P\)

Furthermore, if \(X\) is a take operation, \(R_{x_j}\) reads the last element of the deque, and \(j = X - 1 \geq \tau X\); if \(X\) is a push operation, \(R_{x_j}\) results from a copy operation of the resizing code, hence \(j \geq \tau X\). Since \(X\) occurs after \(P\) in program order and \(\tau\) is monotonically increasing, \(P; \tau P \rightarrow \text{addr} X.R_{x_j} \land j \leq \tau P \land \tau P \leq \tau X\).

If \(j < i\), then, since \(j \geq X - 1 \geq \tau X\) between the write in \(P\) and the read in \(X\), hence, there must be an instance \(P'\) of push between \(P\) and \(X\) (in program order) that increments \(m\) to \(j + 1\).

Indeed, the only writes that increase the value of \(b\) in push and take; and the effect of take as a whole never increases the value of \(b\) since it first
Lemma 6. Given a significant write \( W_1 \) and a significant read \( R_1 \):

\[ (i) \quad W_1^{*} p_{\text{post}} R_1, \& x \quad \implies \quad u \leq v \]

\[ (ii) \quad 0 \leq u \iff W_1^{*} p_{\text{post}} R_1, \& x \]

Proof. We prove each point separately:

(i) Suppose \( v < u \). We define \( W'' \) such that:

\[ W'' = W_1 \& x \]

(ii) If \( v = 0 \), \( c_{1i} \) is an undefined value; let \( W'' \) be the initialization of \( x \). Then \( W'' \) comes before \( W_1^{*} p_{\text{post}} R_1, \& x \) in program order.

Otherwise, \( 0 < v < u \). Let \( W'' \) be the write that is the last write of the set \( \{ W_1, \ldots, W_n \} \) with value \( v \). In other words, there exists a sequence of writes carrying the value of \( c_{1i} \) to \( R_1, \& x \). That sequence ends with a write \( W'' \) that increases in size, so the inequality still holds if the sizes of \( W_1, \ldots, W_n \) are incremented.

Thus, from Lemma 2 (ii), \( W'' \) is a significant read.

Now, assume \( X \) is a successful instance of steal. We have the following execution graph for \( X \):

\[ X \quad \text{exec} \quad X. \quad \text{steal} \quad X. \quad \text{read} \quad X. \quad \text{write} \quad X. \quad \text{close} \quad X. \]

Hence \( p_{\text{post}} R_1, \& x \).

If \( j < i \), then \( j \leq i - \text{size}(x) \). However, the following must hold in \( P \):

\[ j < i - \text{size}(x) + 1 \leq \beta x - \text{size}(x) + 1 < \tau \]

Hence \( \tau x \neq j < \tau \). Since \( \tau x \) increases monotonically, it must be that:

\[ X \quad \text{exec} \quad X. \quad \text{steal} \quad X. \quad \text{read} \quad X. \quad \text{write} \quad X. \quad \text{close} \quad X. \]

Hence \( X. \quad \text{exec} \quad X. \quad \text{steal} \quad X. \quad \text{read} \quad X. \quad \text{write} \quad X. \quad \text{close} \quad X. \)

Lemma 2 (iv) tells us that it is impossible. Thus \( W_1^{*} p_{\text{post}} R_1, \& x \).

Suppose \( \xi \neq \xi' \), then \( \xi = \xi \) can only be an undefined value from the uninitialized array, prior to copying. Indeed, if \( x \) is not affected by copying, then it must be one of the new slots allocated by the resizing, hence its initial value is \( \xi ' \). Let \( R \) be the push operation that allocates the array. Then there exists a \( c_{1i} \) such that:

\[ W_1^{*} p_{\text{post}} R_1, \& x \]

Corollary 1. Given a significant write \( W_1 \) and a significant read \( R_1 \):

\[ i \neq j \implies W_1^{*} p_{\text{post}} R_1, \& x \]

Proof. If \( i \neq j \), we know that \( W_1^{*} p_{\text{post}} R_1, \& x \). Furthermore, all copies, which happen during a resize operation, copy from and to the same index. Since there are less copies than the size of the expanded array, there can be no two copies writing to the same memory location in the new array. Hence, there can be no sequence of copies from \( W_1^{*} R_1, \& x \).

Lemma 6. Given a significant write \( W_1 \) and a significant read \( R_1 \):

\[ (i) \quad W_1^{*} p_{\text{post}} R_1, \& x \quad \implies \quad u \leq v \]

\[ (ii) \quad 0 \leq u \iff W_1^{*} p_{\text{post}} R_1, \& x \]

Proof. We prove each point separately:

(i) Suppose \( v < u \). We define \( W'' \) such that:

\[ W'' = W_1 \& x \]

(ii) If \( v = 0 \), \( c_{1i} \) is an undefined value; let \( W'' \) be the initialization of \( x \). Then \( W'' \) comes before \( W_1^{*} p_{\text{post}} R_1, \& x \) in program order.

Otherwise, \( 0 < v < u \). Let \( W'' \) be the write that is the last write of the set \( \{ W_1, \ldots, W_n \} \) with value \( v \). In other words, there exists a sequence of writes carrying the value of \( c_{1i} \) to \( R_1, \& x \). That sequence ends with a write \( W'' \) that increases in size, so the inequality still holds if the sizes of \( W_1, \ldots, W_n \) are incremented. Moreover, according to the definition of \( (\xi') \) and the semantics of resizing, \( W_1^{*} p_{\text{post}} R_1, \& x \) must come before \( W_1^{*} p_{\text{post}} R_1, \& x \) in program order.

We have two cases: either \( W_1^{*} R_1, \& x \) and \( R_1, \& x \) refer to the same memory location or they do not.

Assume that they refer to the same memory location \( x \). Then it must be that \( W'' \) comes before \( W_1^{*} p_{\text{post}} R_1, \& x \) and we have:

\[ W'' \quad \text{write} \quad x \quad W_1^{*} R_1, \& x \quad \implies \quad W'' \quad \text{write} \quad x \quad W_1^{*} R_1, \& x \]

Hence, from Lemma 2 (ii), \( W'' \) is a significant read.

Conversely, assume that they do not refer to the same memory location. Then there must be a resize operation between \( W'' \) and \( W_1^{*} R_1, \& x \).

Hence, from Lemma 3 (ii), \( W'' \) is a significant read. And from Lemma 2 (ii), \( W'' \) is a significant read. Since there is only one write \( W_1 \) that gives the value \( k \) to \( a \), we have a contradiction.

Corollary 2. Given a significant read \( R_1, \& x \):

\[ \forall \xi \in \mathbb{R}^{+} \quad \sum v \quad W_1, \& x \quad \implies \quad W_1^{*} p_{\text{post}} R_1, \& x \]

It follows from Lemma 2 (iii) (i) and (ii) that \( W_1, \& x \).

4.4 Unique non-significant reads

The results from the previous section establish that two significant reads at different indexes cannot retrieve the same element \( \xi' \). The only possible cause of duplicate significant reads is thus reduced to the case where the reads access the same index.

Theorem 1. (Work-stealing: uniqueness of significant reads). Given a worker thread executing a sequence of push and take operations, and finite number multiplier of threads each executing steal operations, all against a same dequeue. If \( X \) and \( Y \) are two distinct successful instances of steal or take, then:

\[ \forall \xi \quad X, \& R_1, \& x \quad Y, \& i \neq i' \quad v \neq v' \]

Proof. All writes to \( t \) atomically increment it (by atomically CAS). Hence two successful steal operations cannot write (thus read) the same value of \( t \). Reads from \( x \) in a steal operation access the index given by the value of the \( t \) variable. Hence \( R_t, \& x \).

Lemma 7. Given \( S_1 \) and \( S_2 \) distinct significant instances of steal, then:

\[ \forall \xi \quad S_1, \& R_1, \& x \quad S_2, \& i \neq i' \quad v \neq v' \]

Proof. All writes to \( t \) atomically increment it (by atomically CAS). Hence two successful steal operations cannot write (thus read) the same value of \( t \). Reads from \( x \) in a steal operation access the index given by the value of the \( t \) variable. Hence \( R_t, \& x \).


Lemma 8. Given $T$ a successful instance of take and $P$ an instance of push. If $P$ comes after $T$ in program order, then:

$$\forall \xi^R_T \in T, \forall \xi^P_P \in P, i \neq j \lor v \neq u$$

Proof. Assume $i = j \land v = u$. We have $\xi^R_T \rightarrow P \rightarrow \xi^P_P$, therefore $\xi^R_T \rightarrow P \rightarrow \xi^P_P$. From Lemma 6 (ii), it follows that $u > v$. We have a contradiction.

Lemma 9. Given $T_1$ and $T_2$ distinct successful instances of take, $R \xi^R_{T_1} \rightarrow T_1, \forall \xi^R_{T_2} \rightarrow T_2, i \neq j \lor v \neq u'$

Proof. We have the following execution graphs:

$$T_1 \cdot R_{\beta_1} \rightarrow \tau_{\beta_1} \rightarrow T_2 \cdot R_{\beta_2} \rightarrow \tau_{\beta_2} \rightarrow T_3 \cdot R_{\beta_3}$$

And $\beta_1 = 1$ and $\beta_2 = 1$. Since all instances of take occur in the worker thread, we have either:

$$T_1 \cdot W_{b, \beta_1} \rightarrow \tau_{\beta_1} \rightarrow T_2 \cdot R_{\beta_2} \lor T_2 \cdot W_{b, \beta_2} \rightarrow \tau_{\beta_2} \rightarrow T_1 \cdot R_{\beta_1}$$

Let us assume the first case as well as $i = i' \land v = v'$ and show it is impossible, the other case being symmetrical. We have $\beta_1 - 1 = i = i' = \beta_2$, and $T_1 \cdot W_{b, \beta_1} \rightarrow \tau_{\beta_1} \rightarrow T_2 \cdot R_{\beta_2} \rightarrow T_1 \cdot R_{\beta_1}$.

Hence ($\beta_1$) must increase from $i$ to $i + 1$ between $n$ and $n'$; there exists an instance $P$ of push that writes $W_{b, \beta_1}$.

We get the following graph:

$$T \cdot R_{\beta_1} \rightarrow \tau_{\beta_1} \rightarrow T \cdot W_{b, \beta_1} \rightarrow \tau_{\beta_1} \rightarrow T \cdot R_{\beta_1} \rightarrow \tau_{\beta_1} \rightarrow T \cdot W_{b, \beta_1}$$

Lemma 3 (ii) yields $P \cdot \xi^P_P \rightarrow P \cdot \xi^P_P \rightarrow R_{\beta_1} \rightarrow \xi^R_{T_1}$. It then follows from Lemma 6 (i) that $u \leq v'$ and from Lemma 8 that $v < u$. Impossible.

Lemma 10. Given $T$ a successful instance of take and $S$ a successful instance of steal,

$$\forall \xi^R_{T} \in T, \forall \xi^S_{S} \in S, i \neq j \lor v \neq u$$

Proof. We have the following execution graphs:

$$T \cdot R_{\beta_1} \rightarrow \tau_{\beta_1} \rightarrow T \cdot W_{b, \beta_1} \rightarrow \tau_{\beta_1} \rightarrow T \cdot R_{\beta_1} \rightarrow \tau_{\beta_1} \rightarrow T \cdot W_{b, \beta_1}$$

Also, for $T$ to succeed, we must have $\beta_m < \beta_f$. Two cases:

- If $\beta_1 = \beta_m + 1$, then a successful CAS occurs in $T$. Moreover, $\beta_1 = \beta_m + 1$ implies $\beta_1 = \beta_f = \beta_2$, and $\beta_f = \beta_m + 1$. Impossible, since $t$ is monotonically increasing and $S$ also contains a successful CAS with the same value of $t$. $\beta_f = \beta_m + 1$.

- If $\beta_1 > \beta_m + 1$, then no CAS occurs in $T$ and $t' > n$. Since $t$ monotonically increases, there must be a write $A \cdot \xi^P_A \rightarrow B \cdot \xi^R_B$ s.t.

$$A \cdot \tau_{\beta_1} \rightarrow B \cdot \tau_{\beta_1} \rightarrow B \cdot \tau_{\beta_1}$$

Impossible according to Lemma 3 (ii). Therefore $\beta_{\max}$, the source of $S \cdot \beta_f$, must come before $W_{b, \beta_{f+1}}$ (in coherency order, hence in program order as both occur in the same thread). Consequently, ($\beta_n$) must increase from $\beta_n - 1 = i$ to $\beta_n'$ between $n + 1$ and $n'$. $\beta_n < n'$. $\beta_n = \beta_{\max}$.

We get the following execution graph:

$$P \cdot \xi^P_P \rightarrow P \cdot \xi^P_P \rightarrow R_{\beta_1} \rightarrow \xi^R_{T_1}$$

Hence we have $\xi^P_P \rightarrow \xi^P_P \rightarrow \xi^R_{T_1}$ from Lemma 3 (ii) and Lemma 2 (ii). Finally, it follows from Lemma 6 that $u \leq v'$, and from Lemma 8 that $v < u'$. We have a contradiction.

Theorem 1 follows directly from Lemmas 9, 10 and 7.

4.5 Existence of significant reads

Theorem 2 (Work-stealing: existence of significant reads). Consider a worker thread executing a sequence of push and take operations, and a finite number of thief threads each executing steal operations, all against a same deque. If the number of push is finite, then all threads reach a stationary state where $b = t$ in a finite number of transitions, and the following holds globally:

$$\forall \xi^R_T, v > 0 \implies \exists \xi^R_T \text{ in some thread before the stationary point}$$

Let $P_F$ be the last instance of push in the worker thread, in program order. Let $W_{b_{\max}} \in P_F$ and $R_{\tau_{\max}} \in P_F$. We say that an instance $X$ of take or steal is trailing if $R_{b_{\max}} \in X$.

Lemma 11. Given $X$ a successful trailing instance of take or steal:

$$\forall \xi^R_T \in X \implies m > m_F$$

Proof. We have two cases:

- Assume $X$ is an instance of take. $X$ follows $P_F$ in program order:

$$P_F \cdot R_{\tau_{\max}} \rightarrow X \cdot \xi^R_T$$

Then $m \geq m_F$ by uniprocessor constraints.

- Assume $X$ is an instance of steal. Since $X$ is successful, $X$ contains a successful instance of a CAS instruction, hence the two reads from $t$ must yield the same value. Due to the barrier between $X \cdot \tau_{\max}$ and the following read $X \cdot \tau_{\max}$ and the barrier before $P_F \cdot W_{b_{\max}}$, we have:

$$W_{b_{\max}} \rightarrow P_F \cdot R_{\tau_{\max}} \rightarrow \xi^P_P \rightarrow W_{b_{\max}} \rightarrow R_{b_{\max}} \rightarrow X \cdot R_{\tau_{\max}} \rightarrow R_{\tau_{\max}}$$

From Lemma 3 (ii), we have $W_{b_{\max}} \rightarrow \xi^P_P \rightarrow X \cdot R_{\tau_{\max}}$. It then follows from Lemma 3 (i) that $W_{b_{\max}} \rightarrow \xi^P_P \rightarrow R_{\tau_{\max}}$. Total order on CAS instructions and Lemma 2 (ii) guarantee that $\forall k < m_F, \forall \xi^P_P \rightarrow W_{b_{\max}} \rightarrow \xi^P_P \rightarrow R_{\tau_{\max}}$. Therefore, $m > m_F$.

Lemma 12. Given $X$ and $Y$ distinct successful trailing instances of take or steal, then:

$$\forall \xi^R_T \in X, \forall \xi^R_T \in Y, i \neq i'$$

Proof. Assume $i = i'$. According to Theorem 1, $v \neq u'$, hence there exist two distinct significant writes $\xi^R_T$ and $\xi^R_T$. Without loss of generality, let us assume $v < u'$; $P_F \cdot W_{b_{\max}}$ occurs after both writes, in program order. Furthermore, there is a cumulative barrier in push after each significant write, and before $P_F \cdot W_{b_{\max}}$, since $X$ reads from $P_F \cdot W_{b_{\max}}$, we have:

$$\forall \xi^R_T \in X, \forall \xi^R_T \in Y \rightarrow R_{\tau_{\max}}$$

Hence we have $\xi^P_P \rightarrow \xi^P_P \rightarrow Y \cdot \xi^R_T$ from Lemma 3 (i) and Lemma 2 (ii). It then follows from Lemma 6 that $v' \leq v$; thus, $v < u' \leq v$. Impossible.

Corollary 3. The combined number of successful trailing instances of take and steal is less than or equal to $b_{\max} - t_{\max}$.

Proof. Let $X$ be a successful trailing instance of take or steal, and $R_{\beta_n} \in X$ and $\forall \xi^R_T \in X$. We know that $n \geq n_F$ (by definition) and $m \geq m_F$ (from Lemma 11). Hence $\tau_{\max} \geq \tau_{\max}$.

Furthermore, a take operation always contains one decrementing write to $b$ (by one), which may be followed by one incrementing write to $b$ (by one). Hence $n \geq n_F$ implies $\beta_n \leq \beta_{\max}$.

Therefore, $X$ can only read at an index $i$, s.t. $\tau_{\max} \leq i \leq \beta_{\max}$. Lemma 12 tells there can be no more than $\beta_{\max} - \tau_{\max}$ such $X$.

Lemma 13. There is a finite number of successful (trailing or non-trailing) instances of take or steal.

Proof. It follows from Corollary 3 that there is a finite number of successful trailing instances of take or steal.

Furthermore, there must be a finite number of non-trailing take operations, which come before $P_F$ in program order.

Lastly, there is a finite number of push operations, thus $b_{\max}$ has a maximum, $\beta_{\max}$. Since two successful steal operations must read different values of $t$ less than some value of $b$, there can be no more than $\beta_{\max}$ successful instances of steal.

Hence the finite number of successful instances of take or steal.

Lemma 14. In each thread, there exists $X$ a failed instance of take or steal s.t. $\forall R_{\beta_n} \in X, \forall \xi^R_T \in X, \beta_n \leq \tau_{\max}$. Furthermore, each
thread makes no more than \(1 + m_F + \beta_{n_F} - \tau_{m_F}\) attempts at take or steal that result in a failed CAS instruction.\(^2\)

Proof. It follows from Lemma 13 that there is a finite number of successful instances, hence a finite number per thread. Thus, there must exist a failed instance of take or steal. A failure can occur either because the fence is empty (\(\beta_n \leq \tau_n\)) or because of a failed CAS instruction. Suppose there is no \(X\) where \(\beta_n \leq \tau_n\); then all failures must be due to a failed CAS instruction. A failed CAS occurs if the two values of \(t\) read during the instance \(X\) differ. Let \(Y_1\) and \(Y_2\) be two such failed instances executing in a same thread; let us assume that \(Y_2\) follows \(Y_1\) in program order, \(n_1 \neq n'_1\) and \(n_2 \neq n'_2\):

\[
Y_1, R_{n_1} \xrightarrow{\text{po-lo c}} R_{n'_1} \xrightarrow{\text{po-lo c}} Y_2, R_{n_2} \xrightarrow{\text{po-lo c}} R_{n'_2}
\]

There exists a write \(W_{n'_1} \xrightarrow{\text{po-lo c}} Y_2, R_{n_2}\). Due to Lemma 2 (i), we have \(W_{n'_1} \xrightarrow{\text{po-lo c}} Y_2, R_{n_2}\), and, as in the proof of Lemma 11, we deduce that \(n'_1 \leq n_2\). Since \(n_1 \neq n'_1\) \(\land\) \(n_2 \neq n'_2\) and \(t\) is monotonically increasing, it must be that \(n_1 \leq n'_1 \leq n_2 < n'_2\). Hence successive CAS-failing instances in a same thread read increasing values of \(t\). It follows from Corollary 3 that \(t\) takes no more than \(1 + m_F + \beta_{n_F} - \tau_{m_F}\) different values.

Therefore, there can be no more than \(1 + m_F + \beta_{n_F} - \tau_{m_F}\) CAS-failing instances of take or steal per thread. Since there is also a finite number of successful such instances, any further take or steal operations must return empty, and the thread reaches its stationary point. \(\square\)

**Corollary 4.** The combined number of successful (trailing or not) instances of take and steal is equal to the number of push.

Proof. A successful instance of take either decreases the value of \(b\) by one or increases the value of \(t\) by one; a successful instance of steal increases the value of \(t\) by one. An instance of push increases the value of \(b\) by one.

It follows from the previous lemma that the worker thread reaches a stationary point where \(b = t\). Clearly, this cannot occur before all push operations and all successful instances of take have occurred.

Since \(b = t\) at the stationary point and all increases to \(b\) precede the sum of increases to \(t\) and decreases to \(b\) (the combined number of successful instances of take and steal) must be at least equal to the number of increases to \(b\) (the number of push operations). It is exactly equal, as otherwise there would be more significant reads than significant writes, which is impossible according to Theorem 1. \(\square\)

One may finally prove Theorem 2. On the one hand, Corollary 4 tells that the number of significant reads (from a successful instance of take or steal) is equal to the number of significant writes (from an instance of push). On the other hand, Theorem 1 states that significant reads uniquely map to significant writes. By injectivity, there exists a unique significant read for each significant write.

5. On the C11 implementation

The sequentially consistent implementation is a direct translation of the original algorithm using C11 seq_cst atomic variables for all shared accesses. It is obtained from the code in Figure 1 by replacing all memory order constants with seq_cst; doing so makes fences unnecessary, hence they should be removed.

The optimized C11 implementation improves upon the previous version by replacing sequential consistency with release-acquire pairs where appropriate. It establishes happens-before relations between reads and writes, as required by the proof. Unfortunately, without relying on seq_cst, using only release, acquire and consume operations, we were unable to reproduce the required memory ordering constraints needed on the POWER and ARMv7 architectures while adhering to C11 semantics.

Although designed for ARMv7/POWER, most of the arguments developed in the proof informally translate to the rules of C11 in a straightforward fashion. In all cases that do not involve cumulativity, the happens-before relation (defined in 4.1) combined with dependencies, which form the core of the ARMv7/POWER proof, may be replaced with analogous properties pertaining to the C11 happens-before relation combined with release–acquire semantics. The one notable difference between the two models lies in the absence of cumulativity in the design of the C11 abstract machine: neither C11 fences nor C11 atomic accesses guarantee cumulativity. A similar effect can be achieved by chaining alternating release–acquire writes and reads, which form a happens-before path. But this device does not work in situations where propagation needs to be asserted between two reads, rather than a read followed by a write.\(^3\) This situation occurs in the steal operation. Informally (see Lemma 10 for the formal description), it must be that two concurrent steal and take do not read “old” values of both bottom and top, where “old” could be defined as “older than the value known to the other party in coherence order”. The presence of the two cumulative barriers in steal and take on ARMv7 guarantee such a condition:

- if the take barrier is ordered before the barrier in steal, then the program-order previous write to bottom will be propagated to the instance of steal;
- conversely, if the steal barrier is ordered before the barrier in take, then value read by the program-order previous read from top will be propagated to the instance of take.

In the second case, it is important to remark that the write that produced the value read in steal might belong to another thread, and thus not be sequenced before the barrier. In the absence of cumulativity, it need not be propagated to the instance of take.

To enforce this particular case of cumulativity in C11, we rely on the properties of sequential consistency. By making all writes (actually, CAS operations) to top sequentially consistent, we ensure that there is a total ordering between the two fences (in take and steal) and the write that produced the value of top read in the instance of steal. Furthermore, if that read uses acquire semantics, then there is a happens-before relation between it and the steal barrier. Hence, the write must come before said barrier in sequential consistency total order. Then, either the barrier in steal is ordered before the barrier in take, or the other way around:

- if the steal barrier is ordered before the barrier in take, then it follows from seq_cst barrier semantics that the value of top read by take cannot be older than the one read in steal;\(^4\)
- conversely, if the take barrier is ordered before the one in steal, then the value of bottom read by steal cannot be older than the one written in take.\(^5\)

6. Experimental results

We present experimental results on three current and widely used architectures: (1) a Tegra 3 ARMv7 processor rev 9 (v7l) with 4 cores at 1.3GHz and 1GB of RAM; (2) an Intel Core i7-2720QM machine with 4 cores (hyper-threading disabled) at 2.2GHz and 4GB of RAM; and (3) a dual-socket AMD Opteron Magny-Cours 6164HE machine with 2 × 12 cores at 1.7GHz and 16GB of RAM.

All tests were compiled with GCC 4.7.0, the first release of GCC to introduce built-in support for C11 atomics.

\(^2\) Hence a thread eventually reaches a stationary state where \(b = t\); it should be noted that the model does not guarantee progress; it is legal for a thread to end up looping on a non-final state where \(b = t\) but \(b \neq \beta_{n_F}\).

\(^3\) C11 defines a happens-before relation, which does not fully encapsulate the notion of cumulativity. The only inter-thread edges in happens-before come from write–read pairs with release–acquire semantics (see [6] 5.1.2.4p11 and p16). In the absence of a write instruction, no fence or other operation can propagate accumulated information to another thread—in other words, it is not possible to establish a happens-before path between two reads in different threads without an intervening write. Hence the reliance on seq_cst primitives, enforcing a sequentially consistent total order.


We designed a synthetic benchmark to simulate the depth-first traversal of a balanced tree—with breadth \( b \) and depth \( d \)—of empty tasks by a main worker thread, reproducing the prototypical execution of a Cilk program. One or more thieves attempt to steal these tasks. For robustness and predictability, the worker always creates and pushes the same number of tasks in the deque, following the depth-first pattern, regardless of whether a specific continuation has been stolen by another thread (it is simply recorded as stolen, but subsequent tasks spawn normally and locally). The thieves perform `steal` actions at a configurable rate, and discard stolen tasks.

We have experimented with two different methods of `steal` distribution, the goal being to uniformly spread the contention over the entire life of the worker thread. The first method is based on the CPU clock of the core dedicated to each thief; with this technique, the clock is regularly sampled and the appropriate number of `steal` operations is performed accordingly. The second method relies on a random number generator, called in a busy loop, which allows `steal` operations with a set probability. While the clock-based approach produces more reliable results, it can only be used if a low-overhead CPU clock is available from user space, which is unfortunately not the case on our ARM-based system.\(^6\) Conversely, the second technique suffers from imprecision when targeting smaller ranges of frequencies, which is necessary on faster processors or when the number of cores increases.\(^7\) Hence, the former is used on x86 and the latter on ARMv7, with appropriate empirical tuning to gather results over a common representative range of steal throughput.

We selected two workloads: a reasonably broad tree (\( b = 3; d = 15 \)) and, as a special case, a degenerate comb-shaped tree (\( b = 1; d = 10^7 \)). The former is meant to reproduce normal contention with `steal` operations alongside both `push` and `take`, while the latter illustrates a case of contention between `take` and `steal` only.

We measure the time taken by the worker thread to complete the specified number of task creations and consumptions. This in turn serves to compute the push/take throughput—the combined number of `push` and `take` operations completed per unit of time, as well as the effective `steal` throughput, defined as follows: the test protocol strives to perform a number of steals over time, at a fixed, nominal steal throughput; the effective throughput is the real throughput as could be observed after the experiment, i.e., how many steals were actually performed during the lifetime of the worker thread. These metrics provide a measure of the efficiency of the algorithm on its critical path at various levels of contention.

In order to assess the impact of the added barriers on the different architectures, raw throughput values have been normalized by the near-ideal throughput on the same workload (see Table 2), obtained on a single thread with no contention and no synchronization: memory barriers are replaced with simple compiler fences, and CAS operations with a simple branch and conditional assignment. These numbers provide a good approximation of the upper bound on the achievable throughput on each machine, though other minor factors can contribute to higher observable values. In particular, it should be noted that counting the throughput in number of operations per second is, by design, a generalization: the execution time of each operation depends on its nature and the exact control path taken; for example, an invocation of `take` returning empty will be faster than one returning a task.

---

### 6.1 Synthetic benchmarks

In all diagrams, we have included a set of points labeled `nofences`, for comparison purposes. These correspond to the least common denominator among all the tested barrier placement strategies: only relaxed CAS operations are included, with otherwise no memory barriers. The `nofences` version violates the semantics of the work-stealing dequeue. Each of our proposed implementations of the algorithm can be seen as adding a different set of barriers to `nofences`, making it correct. Hence, results obtained with `nofences` should be taken as no more than a general baseline, as the complete lack of fences can lead to unexpected behavior. For instance, Figures 3 and 4 show greater throughput values at high contention for the comb-shaped workload on ARM. Those are the results of a long tail of fast empty `take` operations, an artifact due to the nature of the comb-shaped test and the absence of synchronization between empty `take` and `steal` (enabled by the lack of barriers in `take`).\(^8\)

<table>
<thead>
<tr>
<th></th>
<th>Core 17 (2 threads)</th>
<th>Opteron (2 threads)</th>
<th>Tegra 3 (2 threads)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b = 1; d = 10^7</code></td>
<td>4.87862 × 10^8</td>
<td>2.55142 × 10^8</td>
<td>5.47223 × 10^7</td>
</tr>
<tr>
<td><code>b = 3; d = 15</code></td>
<td>3.60838 × 10^8</td>
<td>2.04978 × 10^8</td>
<td>4.12112 × 10^7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Core 17 (4 threads)</th>
<th>Opteron (24 threads)</th>
<th>Tegra 3 (4 threads)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b = 1; d = 10^7</code></td>
<td>4.88018 × 10^8</td>
<td>2.56214 × 10^8</td>
<td>5.48473 × 10^7</td>
</tr>
<tr>
<td><code>b = 3; d = 15</code></td>
<td>3.66404 × 10^8</td>
<td>2.03235 × 10^8</td>
<td>4.11242 × 10^7</td>
</tr>
</tbody>
</table>

**Table 2.** Near-ideal throughput (s\(^{-1}\))

All the results based on the mixed `push` and `take` “tree” workload show a marked improvement of the hand-written native and `c11` versions over the naïve sequentially consistent translation of the original Chase–Lev algorithm, `seqcst`. While the relative gain remains stable at all levels of contention on the Core i7 and Tegra 3, it drops sharply on the Opteron, presumably because of the higher number of cores. Nevertheless, for low values, which more closely model realistic scenarios, the optimized implementations perform at least 1.5 times better than `seqcst` on both x86 and ARM.

Comparing x86 and ARM, we note that a higher relative throughput is achieved on ARM (peak at above 85%) than on x86 (peak at above 50%), indicating that the first serializing instruction introduced in the code is very costly on x86, especially if it is added to the critical path (as is the case in native, `c11` and `seqcst`, but not in `nofences`). This could suggest either the stronger guarantees of the x86 memory model—a full memory fence is required to linearize history in order to maintain total store order [13]—or aggressive local optimizations for single-thread execution without communication.

From these observations, we can postulate that advanced ARM architectures such as the Tegra 3 benefit the most from a well-written concurrent program that takes full advantage of the flexibility allowed by their memory model, and conversely struggle relatively more with literal interpretations of algorithms designed with stricter, simpler hypotheses in mind.

### 6.2 Task-parallel benchmarks

We further experiment on common task-parallel benchmarks, mostly extracted from the Cilk benchmark suite,\(^9\) to evaluate the impact of the memory barrier optimization on realistic workloads and load-balancing scenarios.

**Fibonacci** is the tree-recursive computation of the 35th Fibonacci number; it illustrates the raw cost of the scheduling algorithm as each task only performs a single addition.

---

\(^6\) The ARMv7 C15 cycle counter register can only be queried if first enabled from kernel mode, and is delegated to a monitoring co-processor, with unclear consequences for the bus, caches, and memory model as a whole.

\(^7\) On higher end processors with multiple cores acting as thieves, higher steal probabilities can yield many times more steal attempts than there are tasks created over a set period.

\(^8\) In the case where the deque is empty, neither `take` nor `steal` needs to execute a CAS instruction; furthermore, in the absence of barriers, the ARMv7 memory model does not require successive decrements and increments of `bottom` in `take` to propagate to the thieves.

\(^9\) http://supertech.csail.mit.edu/cilk
**Figure 3. Synthetic single-thief benchmarks**

**Figure 4. Synthetic multi-thief benchmarks**

**Figure 5. Task-parallel benchmark speedups against the C11 sequentially consistent baseline**

**FFT-1D** computes the Cooley-Tukey fast Fourier transform over a vector of $2^{20}$ elements.

**Matmul** is the blocked matrix multiplication, of size $256 \times 256$ on the Tegra 3 and Core i7 platforms, and of size $384 \times 384$ on Opteron to ensure a sufficient computation time.

**Strassen** is an optimized matrix multiplication algorithm, running on matrices of size $512 \times 512$ on the Tegra 3 and Core i7 platforms, and of size $2048 \times 2048$ on Opteron.

**Knapack** is the usual resource allocation problem. A set of objects, each with a given weight and value, must be picked from
a pool to fit a total weight constraint while maximizing value. We use 33 objects.

Seidel simulates heat transfer using the Gauss-Seidel method which iterates a 5-point stencil over an two-dimensional array. We used a resolution of 1024×1024 points with 20 iterations.

We compare the four implementations of the Chase–Lev deque presented in Section 2. The sequentially consistent, direct translation to C11 serves as a baseline to measure the speedups obtained with the three other implementations. We observe that the nofences version is inherently incorrect and generally results in erroneous executions. This version is only presented as a rough upper-bound on the performance gains of memory barrier optimization.

Figure 5 shows similar trends to the balanced tree synthetic kernel, with a clear advantage to the two optimized implementations (native and c11) over the seqcst baseline. Fibonacci, FFT-1D, Matmul and Strassen use a recursive divide-and-conquer pattern, leading to balanced binary trees. They appear in increasing order of granularity, ranging from a single addition to the multiplication of matrix blocks of size 16×16. On Fibonacci, the lowest granularity kernel, the results are very similar to the throughput achieved by the synthetic benchmark: the optimized versions show up to 1.19× speedup on the Tegra 3 platform against the seqcst version, 1.3× on Core i7 and up to 1.13× on Opteron. These speedups gradually decrease as granularity increases, hiding the cost of the scheduling deque. Yet we still observe significant speedups on the Matmul kernel, with a granularity of 64 (vector) multiply-add operations per task: we obtain up to 1.03× speedup on Tegra 3, 1.1× on Core i7 and 1.04× on Opteron. On Strassen, the largest granularity kernel, we no longer observe any significant improvement: the deque operations are entirely hidden by the work performed in each task.

The Knapsack kernel is also based on a divide-and-conquer pattern, yet it does not result in a balanced tree because of the non-deterministic, dynamic pruning of sub-optimal branches. Unsynchronized communications are used to share the best total value reached on any branch; this value is used to stop exploring branches known to represent sub-optimal prefixes. The nofences version shows lower performance, on Tegra 3, than our optimized c11 and native versions because of longer delays until the best value is propagated to all cores, resulting in less pruning and more work. The performance improvement is reduced on this benchmark because additional barriers improve the accuracy on the current best value. Finally, the Seidel kernel iterates on skewed wave-fronts of data-parallel tasks. A single main thread is responsible for spawning every task in a wave-front. It enqueues all tasks on its own work-stealing deque until it reaches a synchronization barrier. This scheme relies on stealing exclusively for the distribution of work among cores. This behavior puts a lot of strain on a particular deque, and induces a high level of contention. This explains the high performance gains of our optimized implementations on the 24-core Opteron platform: up to 1.3× speedup against the seqcst baseline, even higher than the speedups observed at a lower granularity on Fibonacci. However, despite its somewhat low granularity, corresponding to 16 additions and 4 multiplications of double precision floating point values per task, this benchmark only shows up to 1.05× improvement from our optimized versions on the Tegra 3 platform and up to 1.2× speedup on the Core i7. This is in line with the equivalent speedups observed for the similar granularity on FFT-1D, as the low number of cores on these platforms induce much less contention compared to the Opteron configuration.

Interestingly, our experiments show that the performance of our optimized versions is very close to the nofences version on the Tegra 3 platform. This validates our hypothesis about the benefits of an implementation that takes full advantage of the ARM relaxed memory model, rather than translating the classical sequentially consistent algorithm. Furthermore, the large performance gains on the two x86 platforms show that even in the case of stricter memory models such as total store order [13], relying on sequentially consistent algorithms represents a significant source of overhead.

7. Conclusion
We provided optimized implementations of Chase and Lev’s concurrent deque, targeting the weak memory models of the POWER and ARM processors, as well as the C11 standard. Based on recent progress in the formalization of memory consistency, we established the first proof of the Chase–Lev deque for the weak memory model of a real-world processor. This paves the way for robust parallel library and programming language implementations based on a work-stealing scheduler.

Comparing our optimized implementation with portable C11 versions, we observed unrecoverable overheads in the interaction between atomic operations and the non-cumulativity of memory barriers in C11, a slight mismatch with the POWER and ARM memory models. We obtained strong performance gains on ARM and x86, and performance levels comparable to an (incorrect) fence-free version. This indicates that a high-throughput scheduler can be implemented efficiently on a weak memory model such as a multicore ARM, benefiting from its scalability and energy savings.

Acknowledgments This work was partly supported by the European FP7 projects PHARAON id. 288307 and TERAFLUX id. 249013, and by ANR grant ANR-11-JS02-011.

References