Correct and Efficient Bounded FIFO Queues

Nhat Minh Lê   Adrien Guatto   Albert Cohen   Antoniu Pop

INRIA and ÉNS, Paris, France

first.last@inria.fr

Abstract—Bounded single-producer single-consumer FIFO queues are one of the simplest concurrent data-structure, and they do not require more than sequential consistency for correct operation. Still, sequential consistency is an unrealistic hypothesis on shared-memory multiprocessors, and enforcing it through memory barriers induces significant performance and energy overhead. This paper revisits the optimization and correctness proof of bounded FIFO queues in the context of weak memory consistency, building upon the recent axiomatic formalization of the C11 memory model. We validate the portability and performance of our proven implementation over 3 processor architectures with diverse hardware memory models, including ARM and PowerPC. Comparison with state-of-the-art implementations demonstrate consistent improvements for a wide range of buffer and batch sizes.

Keywords-FIFO queue, lock-free algorithm, weak memory model, data-flow programming, Kahn process network

I. MAKING LOCK-FREE ALGORITHMS EFFICIENT

Single-producer, single-consumer (SPSC) FIFO queues are ubiquitous in embedded software. They arise from a variety of parallel design patterns and from the distribution of Kahn process networks over multiprocessor architectures. Formal reasoning about SPSC bounded queues dates back to the seminal work of Lamport, illustrating proof techniques for concurrent programs [10]. The algorithm is shown in Figure 1. It is essentially identical to the code for a sequential ring buffer, mutual exclusion to a given portion of the underlying array being enforced through index comparisons. Lamport proved that this algorithm does not need any additional synchronization primitives such as locks to work properly. As in most theoretical works on concurrent data-structures, Lamport assumed what he later termed *sequential consistency* [11]: in broad strokes, the concurrent execution of a set of sequential programs is said to be sequentially consistent if it is an interleaving of the executions of the programs. In the case of an imperative program, this implies that its memory actions (loads and stores) are totally ordered and that there is only one global view of memory at a given point in time.

Computer architects consider sequential consistency to be too expensive to implement in practice. This cost is particularly acute on embedded systems under tight silicon area and power budgets. As a result, multi-processors do not offer a sequentially consistent view of memory, but so-called *weak* (or *relaxed*) memory models. The observable memory orderings of various processor architectures has been clarified and formalized in recent publications [14], [16].

A low-level programming language such as C strives to offer both portability and performance to the programmer.

For concurrent programs, this means that its semantics w.r.t. memory cannot be stronger than the processor architectures it may be compiled to. It must also allow usual compiler optimizations, and offer reasonable forward compatibility with future computer architectures. In the case of the C language, these design choices led to the compromise included in the current standard [8]. The general idea is that expert programmers can write portable high-performance lock-free code, enforcing the precise memory ordering required by the task at hand. Non-specialists need not care and can assume sequential consistency. Race-free programs may only exhibit sequentially consistent behavior on their shared variables.[1] The semantics of programs with data races is undefined. Expert programmers are offered a set of primitive data types, called *low-level atomics*, on which concurrent accesses are allowed. Through an associated set of builtin operations, they expose a full spectrum of memory access behavior, from sequential consistency to *relaxed* accesses where basically everything can happen.

Bounded SPSC FIFO queues naturally arise from a number of important parallel programming patterns, starting with streaming languages [7]. They implement flow control and capturing the deterministic semantics of Kahn networks over MPSoC architectures [9]. Their optimization is so important that dedicated hardware implementations have flourished and are still the subject of active research [3]. Beyond the obvious throughput benefits, a fine-tuned FIFO implementation translates into lower communication latency, facilitating the satisfaction of real-time constraints and reducing the memory footprint of in-flight computations, a critical asset for memory-starved embedded processors and many-core architectures. *This motivates the search for the FIFO queue with the highest throughput for a given buffer and batch size.*

This paper introduces WeakRB, an improved SPSC FIFO queue with a portable C11 implementation. WeakRB is proven correct using an axiomatic formalization of the C11 memory model. Its portability and performance is validated over 3 architectures with diverse hardware memory models, including 2 embedded platforms. Our experiments demonstrate consistent improvements over state-of-the-art algorithms for a wide range of buffer and batch sizes. The 2 embedded platforms demonstrate the highest reduction in the minimum buffer and batch sizes sustaining close-to peak throughput across processor cores. Section II introduces our improved FIFO queue. Section III recalls the fundamental concepts and axioms of

---

[1]Two memory accesses form a data race if they concurrently access the same location and at least one of them is a write.

```
                                bool push(T elem) {                    bool pop(T *elem) {
                                  if ((back + 1) % SIZE == front)        if ( front == back)
size_t front = 0;                   return false ;                        return false ;
size_t back = 0;                  data[back] = elem;                   *elem = buffer[front];
T data[SIZE];                     back = (back + 1) % SIZE;            front = (front + 1) % SIZE;
                                  return true ;                         return true ;
                                }                                     }
```

Figure 1.   Lamport's lock-free FIFO queue algorithm

```
                                bool push(T elem) {                    bool pop(T *elem) {
atomic_size_t front;              size_t b, f;                          size_t b, f;
atomic_size_t back;               b = atomic_load(&back, seq_cst);      b = atomic_load(&back, seq_cst);
T data[SIZE];                     f = atomic_load(&front, seq_cst);     f = atomic_load(&front, seq_cst);
                                  if ((b + 1) % SIZE == f)              if (b == f)
void init (void) {                  return false ;                       return false ;
  atomic_init(front, 0);          data[b] = elem;                      *elem = data[b];
  atomic_init(back, 0);           atomic_store(&back, (b+1)%SIZE, seq_cst);   atomic_store(&front, (f+1)%SIZE, seq_cst);
}                                 return true ;                         return true ;
                                }                                     }
```

Figure 2.   Direct C11 translation of the Lamport queue

the C11 memory model, then goes on with the complete, commented proof. Section IV discusses experimental results, before we conclude in Section V.

II. AN EFFICIENT, PORTABLE, CONCURRENT FIFO QUEUE

Translating the algorithm from Figure 1 into C11 is simple. The front and back index variables are accessed concurrently and should thus be of atomic type. Elements of the internal array are never accessed concurrently, hence do not need an atomic type. The next step is to translate loads and stores to index variables, specifying their memory ordering constraints. Since Lamport proved his algorithm in a sequentially consistent setting, we may (for now) rely on memory_order_seq_cst in the explicit stores and loads. Figure 2 shows the resulting code.[2]

To measure the performance of this direct translation, we use a microbenchmark with the simplest usage pattern: two threads and one queue, the first thread writes data to the queue while the second one reads from it. This is not a realistic benchmark, but it heavily stresses the queue's performance. On an embedded ARM processor, our experiments show that this code only exploits 1.25% of the best observed throughput. Situation on a high-end desktop processor is even worse at 0.6%. Can we do better?

A. From Lamport's queue to WeakRB

Applying a profiler (or common sense) to the code in Figure 2 reveals numerous performance anomalies. First, the systematic use of sequentially consistent memory operations causes the compiler to emit numerous memory barriers. Furthermore, concurrent access to index variables incurs heavy coherence traffic. More subtly, we can also argue that the code does not offer any native facility for producing or consuming bulk data, which may be important for applications with coarser-grained communication. Such applications will end up looping around push or pop, increasing contention even for sporadic but large data access.

[2]Omitting memory_order_ prefix from memory order arguments and _explicit suffix from atomic_load and atomic_store, for conciseness.

In the rest of the paper, we suppose that the "client program behaves": no code outside of the push and pop functions may access the front, back and data variables. Moreover, and since we only study SPSC queues, only one thread may call push (resp. pop). We say that back and front are respectively the *owned* and *foreign* indexes for the producer, and symmetrically for the consumer.

a) Relaxed memory ordering: Improving the code begins with a simple but essential remark: a thread at one end of the queue only ever writes to its owned index. In other words, the producer never writes to front, nor does the consumer write to back. Thus, following the C11 memory model, the loads to back and front do not require restrictive memory orderings, since here no relaxed behavior can be observed. We change memory_order_seq_cst to memory_order_relaxed accordingly.

We can then focus on the interaction between each queue's end and its foreign index. What could possibly go wrong if we replaced the remaining sequentially consistent accesses with relaxed ones? Intuitively, the problem is that we are no longer guaranteed that each thread sees a consistent view of memory. E.g., the consumer may read an undefined value from the array: even if push first writes to back and only then to data, with relaxed atomic accesses there is no guarantee that, in pop, reading the new value of back and thus potentially passing the emptiness check implies reading the updated value of data[back].

What we need is a way to enforce exactly this constraint: one should not observe values of indexes inconsistent with the contents of the array. Informally, writing to data[back] in push should be performed "before" incrementing back, in the sense that reading back at the other end of the queue implies that the write to data will effectively be seen. Obviously, the same constraint should apply symmetrically to the read in pop and the corresponding update of front, otherwise push may (metaphorically) overwrite data that has not yet been consumed.

The C11 standard provides memory orderings specifically tailored to this kind of situation: *release* and *acquire*. If an

atomic read tagged with *release* reads the value from an atomic write tagged with *acquire*, all the writes performed **before** the release will be seen by the reads **after** the acquire[3]. We thus replace memory_order_seq_cst in the write to (resp. read of) the owned (resp. foreign) index with memory_order_release (resp. memory_order_acquire).

b) Software caching: At this point, no obvious improvement can be performed on the memory orders for atomic accesses, but the effects of contention on indexes remain. Moreover, one can remark that most of these accesses are likely to be useless. Indeed, imagine the following scenario: the producer thread pushes two items to the back of the queue. Assume the value r of front read in the first call to push is such that front $+ 2 \not\equiv r \pmod{\texttt{SIZE}}$. Then, reading front again in the second call is useless: the consumer could not have advanced past the producer, we are sure that the free space ahead has not disappeared. This insight leads to the following change: we introduce a thread-private variable at each end of the queue, to hold the last value read from its foreign index. Then, instead of reading the foreign index in each call, we use the cached value. If it passes the capacity test, we proceed as planned. If not, we update our cached index with the usual atomic read from the foreign index, and recheck against the new value. This is a classic idea in concurrent programming, and its use in concurrent ring buffers was pioneered by Lee et al. in [13].

c) Batching: To wrap-up our improvements, we extend the programming interface to enable bulk transfers, calling them "batches". Instead of producing and consuming elements one by one, push and pop now handle arrays, and these do not have to be of fixed size. Empty and full checks must be updated to handle multiple elements. The benefits are twofold. First, we amortize the cost of synchronization over multiple elements. Second, with a slight adaptation and since the content of our array is not atomic, it enables the use of a platform-optimized memcpy function. Experiments in Section IV show that this is a clear win.

Figure 3 shows the final code for the queue, incorporating all the optimizations mentioned above. As we gradually improved the performance and portability of Lamport's queue on modern machines and programming languages, we stepped away from the familiar setting of sequential consistency: his proof no longer applies. The correctness of our implementation relies on a formal version of the C11 memory model, which we introduce now.

B. The C11 *memory model*

The memory model described in the standard is axiomatic in nature: it defines constraints on what may constitute the proper memory behavior of a given C program. More precisely, it defines what we call a (candidate) *execution trace*: a set of memory accesses together with relations between them. These relations obey a number of consistency conditions imposed

[3]This is a simplification and that the exact semantics is much more complex, as formalized by Batty et al. in [2].

by the model that we detail below; in turn, they enable the definition of data races, sufficient to rule out nonsensical programs. The elementary building blocks are the *memory locations* and *actions*. Memory locations, denoted by x in this paper, partition memory into *atomic* or *non-atomic* species. We use the following syntax for memory actions, where $\hat{x}$ denotes a value (more on this unusual notation later):

$$
\begin{array}{rcll}
O & ::= & \mathsf{NA} \mid \mathsf{RLX} \mid \mathsf{REL} \mid \mathsf{ACQ} & \text{memory order} \\
X & ::= & P \mid C & \text{thread identifier} \\
a, b, \ldots & ::= & X.\mathsf{R}_O\, x = \hat{x} \mid X.\mathsf{W}_O\, x = \hat{x} & \text{memory action}
\end{array}
$$

$\mathsf{R}_O\, x = \hat{x}$ stands for a load of variable x reading value $\hat{x}$ with memory order O. $\mathsf{W}_O\, x = \hat{x}$ stands for a store writing value $\hat{x}$ to variable x with memory order O. _ is a shorthand for any value. The thread identifier X is prefixed to actions when disambiguation is required. The values of O stand for: non-atomic (NA), relaxed (RLX), acquire (ACQ; only applies to loads), release (REL; only applies to stores). The model ensures that non-atomic accesses are only performed at non-atomic locations, and that release, acquire and relaxed accesses only affect atomic ones.

The observable memory actions are constrained by various relations, modeling interactions, synchronization and interference inside and between threads. In our context, the only relevant relations are:

sequenced-before

> or $\xrightarrow{\text{sb}}$, models control flow, in a given thread. One action a is sequenced before b if it comes before in (sequential) program order.

reads-from

> or $\xrightarrow{\text{rf}}$, relates writes to reads at the same location. It models the fact that a given write produced the value consumed by a given read. Several reads may read from the same write.

happens-before

> or $\xrightarrow{\text{hb}}$, is a strict partial order describing synchronization. In this work, $\xrightarrow{\text{hb}}$ is built from the two previous relations: $a \xrightarrow{\text{hb}} b$ if $a \xrightarrow{\text{sb}} b$, or a is a release-write, b an acquire-read and $a \xrightarrow{\text{rf}} b$.

Two axioms rule out nonsensical behavior in these relations:

AXIOM 1. Happens-before is acyclic.

AXIOM 2. For any load c, and stores a and b, at the same location, $a \xrightarrow{\text{hb}} b \xrightarrow{\text{rf}} c$ implies that $a \not\xrightarrow{\text{rf}} c$. In other words, a load can only read from the latest store that happens before it, at the same location.

We can now precisely define data races. A data race is a pair of actions (a, b) over the same *non-atomic* location, at least one of them being a write, such that neither $a \xrightarrow{\text{hb}} b$ nor $b \xrightarrow{\text{hb}} a$. Let us remark that this implies, together with the coherency axiom, that in race-free programs non-atomic reads always read from the unique last write w.r.t. $\xrightarrow{\text{hb}}$. Programs with at least one racy (candidate) execution trace have undefined behavior and thus no proper execution traces to speak of.

These are the only axioms needed in our proof. The full memory model and its formalization proposed by Batty et al.

```
atomic_size_t front;              bool push(const T *elems, size_t n) {        bool pop(T *elems, size_t n) {
size_t pfront;                      size_t b, f;                                  size_t b, f;
atomic_size_t back;                 b = atomic_load(&back, relaxed);              f = atomic_load(&front, relaxed);
size_t cback;                       if ( pfront + SIZE - b < n) {                 if ( cback - f < n) {
                                      pfront = atomic_load(&front, acquire);        cback = atomic_load(&back, acquire);
_Static_assert(SIZE_MAX % SIZE == 0,  if ( pfront + SIZE - b < n)                   if ( cback - f < n)
  "SIZE div SIZE_MAX");                   return false ;                              return false ;
T data[SIZE];                       }                                             }
                                    for (size_t i = 0; i < n; i++)                for (size_t i = 0; i < n; i++)
void init (void) {                    data[(b+i) % SIZE] = elems[i];                elems[i] = data[(f+i) % SIZE];
  atomic_init(front, 0);            atomic_store(&back, b + n, release);          atomic_store(&front, f + n, release);
  atomic_init(back, 0);             return true ;                                 return true ;
}                                   }                                             }
```

Figure 3.   C11 code for the WeakRB bounded FIFO queue

in [2] and [1] is much more complex. This discrepancy comes from several facts: we only use relaxed and release/acquire semantics for atomics; our code never exhibits release sequences of length greater than one; and, since a same location is always written to in the same thread, modification-order is included in happens-before (through sequenced-before).

III. CORRECTNESS PROOF

Let us first complement the above definitions.

A. Definitions and hypotheses

d) Actions and values: For convenience, we represent initializations as pseudo-stores $W_{INI} x = \hat{x}$, and $W_{INI/REL} x = \hat{x}$ stands for either a release-store or the pseudo-store writing the initial value. For any variable x, $(\hat{x}(v))_{v \in \mathbf{N}}$ is the v-indexed sequence of values of x in modification order. $\hat{x}(0)$ is the initial value of x, such that $W_{INI} x = \hat{x}(0)$. Hereafter, we assume shared variables are initialized to zero; hence $\hat{x}(0) = 0$.

e) Threads and sequences of operations: For a given single-producer single-consumer queue, we consider two threads: the producer P and the consumer C. In the producer (resp. consumer) thread, push (resp. pop) operations are sequentially ordered. We note P_k the push of rank k and C_k the pop of rank k. Note that only the producer thread stores new values in the shared variable b (back in the code); symmetrically only the consumer writes to the shared variable f (front in the code). The sequence of push (resp. pop) operations alternate between cached, successful uncached and failed push (resp. pop) instances.

- A *cached* push (resp. pop) determines locally that it has enough space ahead in the buffer, and does not reload the shared variable f (resp. b).
- A successful *uncached* push (resp. pop) observes that it does not have sufficient space left over from its previous operation to complete its current request, and reloads the shared variable f (resp. b). It then ascertains that sufficient space is available and proceeds successfully.
- A *failed* push (resp. pop) is an uncached push (resp. pop) that observes an insufficient amount of space available after reloading f (resp. b).

We define $\hat{x}^X(k)$ as the value of x as could be observed at the beginning of the operation of rank k in thread X.

Execution traces studied in the proof do not explicit thread-private variables that are used to cache copies of variables f

and b (see pfront and cback in Figure 3). Instead, $\hat{x}^X(k)$ is the value read by the load immediately preceding the operation of rank k, assumed to be kept locally by the thread from its previous operation by use of a thread-private variable.

Thread-private variable are initialized to zero. Hence the first push operation in the producer thread will be uncached, and the first pop operation in the consumer thread will be uncached or failed. We subsequently define, for a thread X, the functions L_X and E_X on ranks as follows:
$$L_X(k) = \max\{i \le k \mid X_i \text{ is uncached}\}$$
$$E_X(k) = \max\{i \mid L_X(i) \le L_X(k)\}$$
Intuitively, $L_X(k)$ is the index of the nearest preceding uncached instance; $E_X(k)$ is the highest rank in the sequence of cached instances to which X_k belongs.

f) Wrap-around and modulo arithmetic: Modulo-arithmetic on machine integers is natively supported. The shared variables f and b are implemented as $\log_2(M)$-bit unsigned integers, where M is SIZE_MAX+1 in the C11 code. However, the value sequences $\hat{f}(v)$ and $\hat{b}(v)$ are *not* wrapped; instead, the bit width constraint is reflected in the proof through the use of modulo-M arithmetic on the variables. This adjustment makes for easy distinction between equal wrapped values obtained from successive increments of f and b. The $x \bmod y$ operation denotes the integer remainder of x divided by y. The $Q[i]$ pseudo-variable denotes the memory location with index $i \bmod m$ in the underlying array backing the queue, where m is the size of the array, i.e., SIZE in the code. For the definition of $Q[i]$ to match the C11 code given in Figure 3, m must divide M, so that $\forall i \in \mathbf{N}, (i \bmod M) \bmod m = i \bmod m$. Additionally, if M is chosen different from SIZE_MAX+1, the remainder operations need to be made explicit.

B. Execution paths

We now formally define the three kinds of push and pop instances (cached, uncached and failed), following the execution paths through the control flow graph of the corresponding function. Figure 4 shows all three execution paths of a push or pop operation. Paths are split into their constituent shared memory accesses, both atomic and non-atomic. For accesses to the data buffer, which depend on the batch size argument n, only the first and last are represented. When multiple outgoing edges are possible, each one is annotated with the
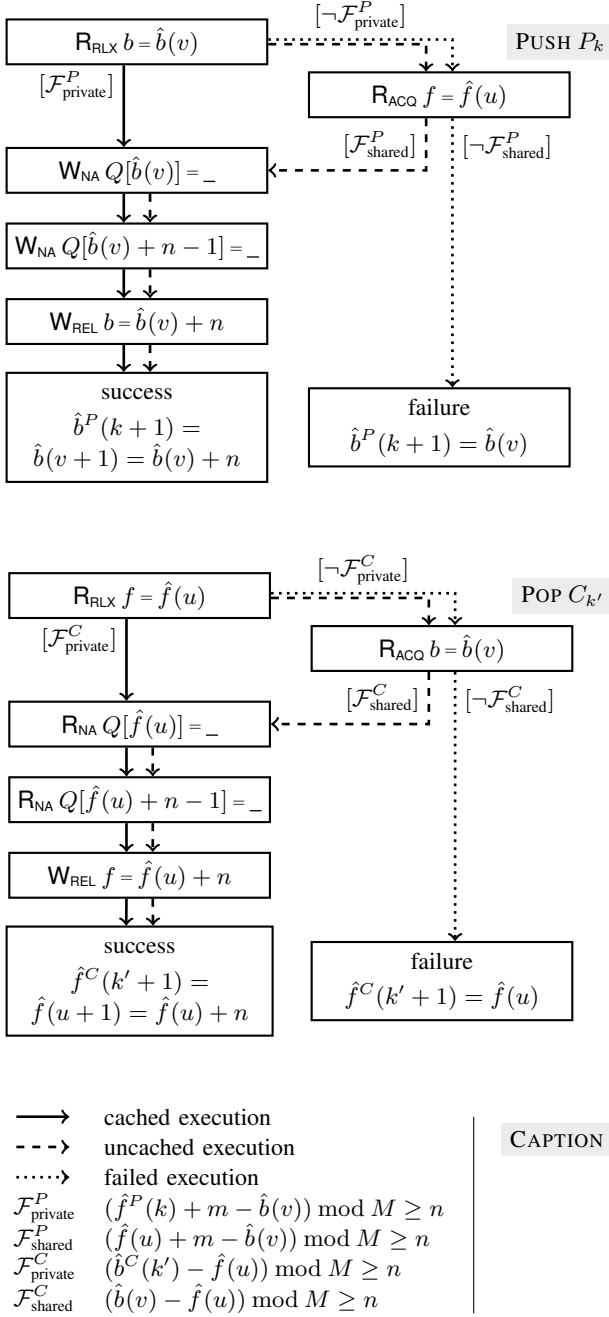
PUSH P_k

$\mathsf{R_{RLX}}\ b = \hat{b}(v)$

$[\mathcal{F}^P_{\text{private}}]$ $[\neg\mathcal{F}^P_{\text{private}}]$

$\mathsf{R_{ACQ}}\ f = \hat{f}(u)$

$[\mathcal{F}^P_{\text{shared}}]$ $[\neg\mathcal{F}^P_{\text{shared}}]$

$\mathsf{W_{NA}}\ Q[\hat{b}(v)] = \_$

$\mathsf{W_{NA}}\ Q[\hat{b}(v) + n - 1] = \_$

$\mathsf{W_{REL}}\ b = \hat{b}(v) + n$

success
$\hat{b}^P(k + 1) =$
$\hat{b}(v + 1) = \hat{b}(v) + n$

failure
$\hat{b}^P(k + 1) = \hat{b}(v)$

POP $C_{k'}$

$\mathsf{R_{RLX}}\ f = \hat{f}(u)$

$[\mathcal{F}^C_{\text{private}}]$ $[\neg\mathcal{F}^C_{\text{private}}]$

$\mathsf{R_{ACQ}}\ b = \hat{b}(v)$

$[\mathcal{F}^C_{\text{shared}}]$ $[\neg\mathcal{F}^C_{\text{shared}}]$

$\mathsf{R_{NA}}\ Q[\hat{f}(u)] = \_$

$\mathsf{R_{NA}}\ Q[\hat{f}(u) + n - 1] = \_$

$\mathsf{W_{REL}}\ f = \hat{f}(u) + n$

success
$\hat{f}^C(k' + 1) =$
$\hat{f}(u + 1) = \hat{f}(u) + n$

failure
$\hat{f}^C(k' + 1) = \hat{f}(u)$

⟶	cached execution
⇢	uncached execution
⋯⋯⋗	failed execution
$\mathcal{F}^P_{\text{private}}$	$(\hat{f}^P(k) + m - \hat{b}(v)) \bmod M \geq n$
$\mathcal{F}^P_{\text{shared}}$	$(\hat{f}(u) + m - \hat{b}(v)) \bmod M \geq n$
$\mathcal{F}^C_{\text{private}}$	$(\hat{b}^C(k') - \hat{f}(u)) \bmod M \geq n$
$\mathcal{F}^C_{\text{shared}}$	$(\hat{b}(v) - \hat{f}(u)) \bmod M \geq n$

CAPTION

Figure 4. Execution paths of push P_k and pop $C_{k'}$

corresponding predicate condition, indicated between brackets, under which it is taken.

C. Proof sketch

The proof is split in two. In the first half, up to Corollary 2, we prove useful invariants on the index variables f and b, using coherency and release–acquire semantics. These first results focus on establishing bounds on the locally observable values of the indexes (e.g., $\hat{f}^P(k) \leq \hat{b}^P(k)$ between the locally observable values of f and b in P_k). The latter half, from

Lemma 6 onwards, exploits these invariants to prove our three main theorems.

- Theorem 1 establishes that calls to pop either fail or return a different element each time.
- Theorem 2 establishes that successful calls to pop read all elements pushed in the queue, without skipping.
- Finally, Theorem 3 asserts that the algorithm does not contain any data race, despite accessing its data buffer non-atomically.

The flow of this proof can be understood as building systems of inequations that substantiate the goals. We use local hypotheses (i.e., predicates listed in Section III-B) as elementary inequalities, further refined through constraints derived from the partial orders that comprise the memory model.

The following are some examples of how the memory model may affect established relations:

- A read-from edge ($\xrightarrow{\text{rf}}$) identifies two observable values of the same variable as being equal. The values may be observed from the same or different threads.
- A happens-before edge between two stores hides the older value from loads happening after the newer store. This limits the set of observable values for a variable in a given context; under hypotheses of monotony, this can be written as an inequality.
- Conversely, a happens-before edge ($\xrightarrow{\text{hb}}$) from a load to a store may induce an opposite inequality, under the same hypotheses.

Additionally, where needed, more complex constraints that expand significantly—and perhaps unintuitively—upon the premises of the memory model, are elaborated inductively. The inequations themselves consist of straightforward modular arithmetic. We included the calculations for completeness, but they should not distract from the construction of the formulas as the main contribution of this proof.

D. Proof

In this extended abstract, we are able to provide the intermediate lemmas leading to the three main theorems of the proof, but the proofs for these intermediate lemmas and for the theorems themselves can be found in the associated technical report [12].

Lemma 1: Reading a foreign index value $\hat{y}$ prevents any later acquire-load in the same thread from obtaining a value older than $\hat{y}$, and any earlier acquire-load from obtaining a newer value (relative to the modification order of y).

Conversely, storing an owned index value $\hat{x}$ that is read by the foreign thread as $\mathsf{R_{ACQ}}\ x = \hat{x}$ prevents any acquire-load of the foreign index sequenced before the store from obtaining a value newer than that at the point of $\mathsf{R_{ACQ}}\ x = \hat{x}$.

Formally, for all $k \geq 0$ and $k' \geq 0$.
If $\mathsf{W_{INI/REL}}\ b = \hat{b}^P(k) \xrightarrow{\text{rf}} C_{k'}.\mathsf{R_{ACQ}}\ b = \_$, then:
$\forall l < k,\ P_l$ is not cached
$\implies \exists l' \leq k',\ \mathsf{W_{INI/REL}}\ f = \hat{f}^C(l') \xrightarrow{\text{rf}} P_l.\mathsf{R_{ACQ}}\ f = \_$
$\forall l' \leq k',\ C_{l'}$ is not cached
$\implies \exists l \leq k,\ \mathsf{W_{INI/REL}}\ b = \hat{b}^P(l) \xrightarrow{\text{rf}} C_{l'}.\mathsf{R_{ACQ}}\ b = \_$

If $\mathsf{W}_{\text{INI/REL}}\, f = \hat{f}^C(k') \xrightarrow{\text{rf}} P_k.\,\mathsf{R}_{\text{ACQ}}\, f = \_$, then:

$\forall l \leq k,\ P_l$ is not cached
$$\implies \exists l' \leq k',\ \mathsf{W}_{\text{INI/REL}}\, f = \hat{f}^C(l') \xrightarrow{\text{rf}} P_l.\,\mathsf{R}_{\text{ACQ}}\, f = \_$$
$\forall l' < k',\ C_{l'}$ is not cached
$$\implies \exists l \leq k,\ \mathsf{W}_{\text{INI/REL}}\, b = \hat{b}^P(l) \xrightarrow{\text{rf}} C_{l'}.\,\mathsf{R}_{\text{ACQ}}\, b = \_$$

From Lemma 2 to Corollary 1, we prove the following local bounds on the index values, under various hypotheses:

$$0 \leq \hat{b}^X(k) - \hat{f}^X(k) \leq m$$

where X is either the producer P or the consumer C, and X_k designates a specific instance of push or pop. We say that an instance is *bounded* if it satisfies the above predicate.

Lemma 2: If a cached instance of push or pop is bounded, then all following operations up to and including the next non-cached instance are also bounded.

Formally, let X be the producer P or the consumer C. For all k such that X_k is cached and $0 \leq \hat{b}^X(k) - \hat{f}^X(k) \leq m$:
$$\forall l \in \{k, \ldots, E_X(k) + 1\},\ 0 \leq \hat{b}^X(l) - \hat{f}^X(l) \leq m$$

Lemma 3: If an instance of push or pop reads the initial value of its foreign index, then every operation up to and including the next uncached instance is bounded.

Formally, given push P_k and pop $C_{k'}$,

(i) if $\mathsf{W}_{\text{INI}}\, b = 0 \xrightarrow{\text{rf}} C_{k'}.\,\mathsf{R}_{\text{ACQ}}\, b = 0$, then the following holds:
$$\forall l' \leq E_C(k') + 1,\ 0 \leq \hat{b}^C(l') - \hat{f}^C(l') \leq m,$$

(ii) if $\mathsf{W}_{\text{INI}}\, f = 0 \xrightarrow{\text{rf}} P_k.\,\mathsf{R}_{\text{ACQ}}\, f = 0$, then the following holds:
$$\forall l \leq E_P(k) + 1,\ 0 \leq \hat{b}^P(l) - \hat{f}^P(l) \leq m$$

Lemma 4: If an instance X_k of push or pop reads a foreign index value written by a foreign bounded operation, then the next operation X_{k+1} in the same thread as X_k is also bounded.

Formally, given push P_k and pop $C_{k'}$, such that $0 \leq \hat{b}^C(k') - \hat{f}^C(k') \leq m$ and $0 \leq \hat{b}^P(k) - \hat{f}^P(k) \leq m$.
If $P_{k-1}.\,\mathsf{W}_{\text{REL}}\, b = \hat{b}^P(k) \xrightarrow{\text{rf}} C_{k'}.\,\mathsf{R}_{\text{ACQ}}\, b = \hat{b}^P(k)$, then:
$$0 \leq \hat{b}^C(k' + 1) - \hat{f}^C(k' + 1) \leq m$$
If $C_{k'-1}.\,\mathsf{W}_{\text{REL}}\, f = \hat{f}^C(k') \xrightarrow{\text{rf}} P_k.\,\mathsf{R}_{\text{ACQ}}\, f = \hat{f}^C(k')$, then:
$$0 \leq \hat{b}^P(k + 1) - \hat{f}^P(k + 1) \leq m$$

Lemma 5: If an instance of push or pop reads a foreign index value from a release-store, then every operation up to and including the next uncached instance is bounded.

Formally, given push P_k and pop $C_{k'}$.
If $C_{k'}$ is not cached, then the following holds:
$$\forall l' \in \{k' + 1, \ldots, E_C(k') + 1\},\ 0 \leq \hat{b}^C(l') - \hat{f}^C(l') \leq m$$
If P_k is not cached, then the following holds:
$$\forall l \in \{k + 1, \ldots, E_P(k) + 1\},\ 0 \leq \hat{b}^P(l) - \hat{f}^P(l) \leq m$$

Corollary 1: All instances of push or pop are bounded. In other words, for X either the producer P or the consumer C, and all $k \geq 0$, we have: $0 \leq \hat{b}^X(k) - \hat{f}^X(k) \leq m$.

Corollary 2: All accesses—both loads or stores—to the data buffer Q take place at an index within the local bounds previously established.

Formally, given a push P_k and a store $\mathsf{W}_{\text{NA}}\, Q[i] = \_$ in P_k:
$$0 \leq \hat{b}^P(k) \leq i < \hat{f}^P(k) + m$$
And given a pop $C_{k'}$ and a load $\mathsf{R}_{\text{NA}}\, Q[j] = \_$ in $C_{k'}$:
$$0 \leq \hat{f}^C(k') \leq j < \hat{b}^C(k')$$

The remaining lemmas and theorems pertain to the data transferred through the single-producer single-consumer queue. We recall that all accesses to the data buffer are made by the FIFO code alone. Consequently, any load (resp. store) from the data buffer is implicitly assumed to take place during a pop (resp. push).

Lemma 6: Reading from the data buffer yields a well-defined value, written by a corresponding store.

In other words, given a load $\mathsf{R}_{\text{NA}}\, Q[j] = \_$ from an instance of pop: $\exists\, \mathsf{W}_{\text{NA}}\, Q[i] = \_,\, \mathsf{W}_{\text{NA}}\, Q[i] = \_ \xrightarrow{\text{rf}} \mathsf{R}_{\text{NA}}\, Q[j] = \_$.

Lemma 7: A load from the data buffer reads exactly the value written by a store at the same extended index (in $\mathbf{N}$). In other words, if $\mathsf{W}_{\text{NA}}\, Q[i] = \_ \xrightarrow{\text{rf}} \mathsf{R}_{\text{NA}}\, Q[j] = \_$, then $i = j$.

Lemma 8: All stores to the data buffer at some index i happen before any load at an index $j > i$.

Formally, given a store $\mathsf{W}_{\text{NA}}\, Q[i] = \_$ and a load $\mathsf{R}_{\text{NA}}\, Q[j] = \_$, we have the following implication:
$$i \leq j \implies \mathsf{W}_{\text{NA}}\, Q[i] = \_ \xrightarrow{\text{hb}} \mathsf{R}_{\text{NA}}\, Q[j] = \_$$

Theorem 1: [Unicity] A value stored in the data buffer is never read more than once.

Formally, given a store $\mathsf{W}_{\text{NA}}\, Q[i] = \_$ in an instance of push, there exists at most one load $\mathsf{R}_{\text{NA}}\, Q[j] = \_$ from an instance of pop, such that $\mathsf{W}_{\text{NA}}\, Q[i] = \_ \xrightarrow{\text{rf}} \mathsf{R}_{\text{NA}}\, Q[j] = \_$.

Theorem 2: [Existence] For any store to the data buffer, there is a matching load that reads its value, provided enough data is requested by the consumer.

Formally, given a store $\mathsf{W}_{\text{NA}}\, Q[i] = \_$, there is at least one load $\mathsf{R}_{\text{NA}}\, Q[j] = \_$ such that $\mathsf{W}_{\text{NA}}\, Q[i] = \_ \xrightarrow{\text{rf}} \mathsf{R}_{\text{NA}}\, Q[j] = \_$, provided the consumer thread contains enough non-failed non-empty pop operations: $\sum_{k' \in \text{NFC}} n_{k'} \geq i$, where $\text{NFC} = \{l' \mid C_{l'} \text{ is non-failed}\}$, and $n_{l'}$ denotes the batch size argument passed to $C_{l'}$.

Theorem 3: [Data-race freedom] All (non-atomic) accesses to the data buffer are data-race-free. That is, given a store $\mathsf{W}_{\text{NA}}\, Q[i] = \_$:
$$\forall\, \mathsf{W}_{\text{NA}}\, Q[j] = \_,\, i \equiv j \pmod m,$$
$$\mathsf{W}_{\text{NA}}\, Q[i] = \_ \xrightarrow{\text{hb}} \mathsf{W}_{\text{NA}}\, Q[j] = \_ \vee \mathsf{W}_{\text{NA}}\, Q[j] = \_ \xrightarrow{\text{hb}} \mathsf{W}_{\text{NA}}\, Q[i] = \_$$
$$\forall\, \mathsf{R}_{\text{NA}}\, Q[j] = \_,\, i \equiv j \pmod m,$$
$$\mathsf{W}_{\text{NA}}\, Q[i] = \_ \xrightarrow{\text{hb}} \mathsf{R}_{\text{NA}}\, Q[j] = \_ \vee \mathsf{R}_{\text{NA}}\, Q[j] = \_ \xrightarrow{\text{hb}} \mathsf{W}_{\text{NA}}\, Q[i] = \_$$

IV. EXPERIMENTS

Let us now evaluate our queue on a variety of multicore processors. Our reference platforms are mid-grade ARM and POWER processors, and a recent x86_64 processor from Intel serving as high-performance baseline. Table I details their characteristics. In particular, the last row displays the best throughput that we managed to achieve between two threads, measuring the raw speed of core-to-core coherence.

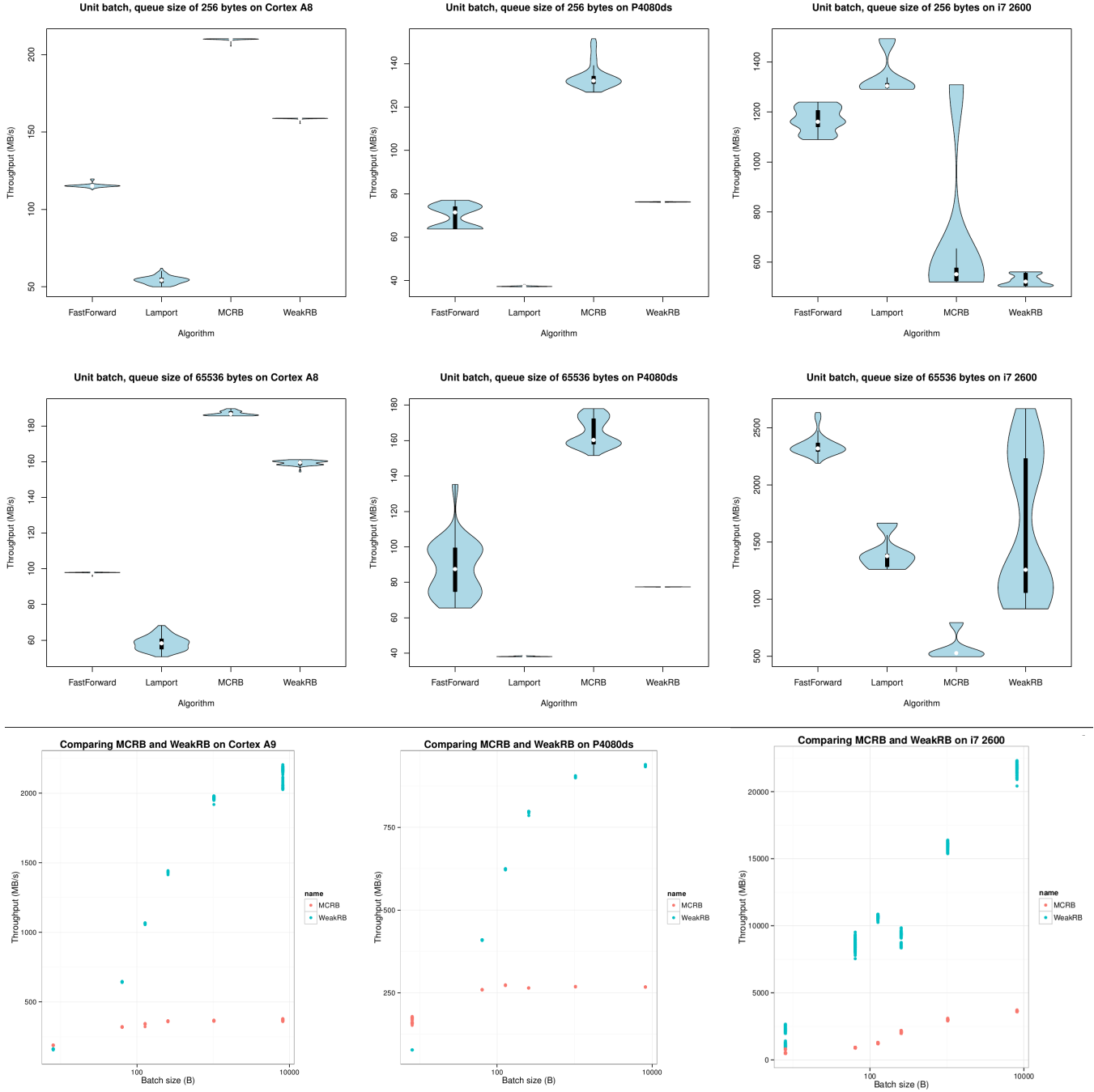| Machine	Cortex A9	P4080	Core i7
Manufacturer	Samsung	Freescale	Intel
ISA	ARMv7	POWER	x86_64
Number of cores	4	8	4 (8 logical)
Clock frequency	1.3 GHz	1.5 Ghz	3.4 GHz
Best throughput	2.2 GB/s	1 GB/s	22 GB/s

Table I
PLATFORM CHARACTERISTICS

Figure 5.   Performance results

We compare the performance in terms of throughput of our queue against the most popular algorithms from the literature, for which an optmized C11 implementation has been derived.

- Lamport's algorithm ported to C11.
- The FastForward queue from Giacomoni et al. [6], using private indexes, and synchronizing through a special value denoting empty elements.
- The MCRB queue designed by Lee et al. [13], extending Lamport's queue with software caching of foreign indexes, and batching updates to the owned indexes.

The benchmark is a synthetic producer-consumer loop with no other computations in between FIFO operations; it is parameterized in buffer and batch size. We measure the throughput with increasing batch sizes, comparing WeakRB with MCRB, the state-of-the-art queue featuring batching. Figure 5 presents our results using the R *vioplot* library. These *violin plots* combine box plots and kernel density estimators. The width represents the density of data points for a given value, with a logarithmic bias. Solid black bars represent the 95% confidence intervals for the median. The first two rows of

graphs show how the queues perform on the communication of single machine words. The first row shows the performance with a small queue size (256 B), the second with a larger one (64 kB).

One may first remark that on the architectures with weaker memory consistency, ARM and POWER, reducing the number of release/acquire ordering constraints is very important. Indeed, the direct translation of Lamport's algorithm performs very poorly on such architectures. Even more convincing is the fact that MCRB, which uses software caches for the foreign indexes, dominates FastForward for all queue sizes on ARM and POWER, but that the situation is reversed on the stronger memory model of the x86_64 machine.

The results also show that WeakRB is at a performance disadvantage for the smallest batch sizes ($\leq$ 16 B). This is due to the use of a generic (yet optimized) memcpy function supporting variable batch sizes. To alleviate this penalty, our implementation could be specialized for tiny, constant size batches. We choose not to do it to avoid polluting the results with unrelated memcpy optimizations, and because batches smaller than a cache line are of low practical interest due to false sharing. This is actually reflected in the highly unstable performance of WeakRB on unit batches, where the consumer starves on data, reading incomplete cache lines.

The last row of Figure 5 features the results for WeakRB and MCRB for increasing batch sizes and with a queue size of 64 kB. It shows the tremendous performance benefits of batched transfers for WeakRB across all architectures. WeakRB exceeds MCRB's throughput as soon as the batch is greater than a cache line, and continues to increase regularly until reaching the maximal throughput once synchronization costs are completely amortized. Interestingly, performance grows faster on the 2 embedded architectures: over 75% of the best observed throughput is obtained with a batch size of 1024 B on ARM and only 256 B on POWER.

Note that the original FastForward paper described a temporal slipping optimization, where the consumer slows down temporarily if it the producer is deemed to be too close. However, the technique used to measure the distance between producer and consumer is not explained in the paper and seems to boil down to two alternatives: either accessing the foreign index or walking the internal array looking for nearby non-empty elements. The former would defeat the algorithm's purpose, and in a relaxed setting the latter is likely to either be incorrect or very slow. Furthermore, it is not available in the public version of FastForward.

V. CONCLUSION AND PERSPECTIVES

WeakRB is the first SPSC FIFO queue with a formally proven, portable implementation for a weak memory model. Our algorithm and implementation reach the peak observable throughput on 3 hardware platforms, and do so with small batch sizes. We plan to use WeakRB to optimize a streaming data-flow language [15], and for the correct-by-construction distribution of high-performance control systems [5], [4].

REFERENCES

[1] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *POPL*, pages 509–520, New York, NY, 2012.

[2] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66, New York, NY, 2011.

[3] G. Bloom, G. Parmer, B. Narahari, and R. Simha. Shared hardware data structures for hard real-time systems. In *EMSOFT*, pages 133–142, 2012.

[4] A. Cohen, L. Gérard, and M. Pouzet. Programming parallelism with futures in Lustre. In *EMSOFT*, pages 197–206, Tampere, Finland, 2012.

[5] G. Delaval, A. Girault, and M. Pouzet. A type system for the automatic distribution of higher-order synchronous dataflow programs. In *LCTES*, 2008.

[6] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *PPOPP*, pages 43–52, New York, NY, 2008.

[7] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS*, San Jose, CA, 2006.

[8] *Programming Language C*. Number ISO 9899:2011. ISO, Geneva, Switzerland, 2011.

[9] R. L. Jeronimo Castrillon and G. Ascheid. Maps: Mapping concurrent dataflow applications to heterogeneous mpsocs. *IEEE Trans. on Industrial Informatics*, page 19, 2011.

[10] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering*, SE-3(2):125–143, 1977.

[11] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.

[12] N. M. Lê, A. Guatto, A. Cohen, and A. Pop. Correct and Efficient Bounded FIFO Queues. Research Report RR-8365, INRIA, Sept. 2013.

[13] P. P. C. Lee, T. Bu, and G. Chandranmenon. A lock-free, cache-efficient shared ring buffer for multi-core architectures. In *Proc. of the 5th Symp. on Architectures for Networking and Communications Systems*, pages 78–79, New York, NY, 2009.

[14] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOL*, LNCS, pages 391–407, Berlin, Heidelberg, 2009. Springer.

[15] A. Pop and A. Cohen. OpenStream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Trans. on Architecture and Code Optimization*, 9(4), 2013. HiPEAC presentation.

[16] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *PLDI*, pages 175–186, New York, NY, 2011.