FRC: A High-Performance Concurrent Parallel Deferred Reference Counter for C++

Charles Tripp
Terrain Data, Inc.
Palo Alto, CA, USA
charles@terraindata.com

David Hyde
Terrain Data, Inc.
Palo Alto, CA, USA
Stanford University
dabh@stanford.edu

Benjamin Grossman-Ponemon
Terrain Data, Inc.
Palo Alto, CA, USA
Stanford University
bponemon@stanford.edu

Abstract

We present FRC, a high-performance concurrent parallel reference counter for unmanaged languages. It is well known that high-performance garbage collectors help developers write memory-safe, highly concurrent systems and data structures. While C++, C, and other unmanaged languages are used in high-performance applications, adding concurrent memory management to these languages has proven to be difficult. Unmanaged languages like C++ use pointers instead of references, and have uncooperative mutators which do not pause easily at a safe point. Thus, scanning mutator stack root references is challenging.

FRC only defers decrements and does not require mutator threads to pause during collection. By deferring only decrements, FRC avoids much of the synchronization overhead of a fully-deferred implementation. Root references are scanned without interrupting the mutator by publishing these references to a thread-local array. FRC's performance can exceed that of the C++ standard library's shared pointer by orders of magnitude. FRC's thread-safety guarantees and low synchronization overhead enable significant throughput gains for concurrently-readable shared data structures.

We describe the components of FRC, including our static tree router data structure: a novel barrier which improves the scalability of parallel collection workers. FRC's performance is evaluated on several concurrent data structures. We release FRC and our tests as open-source code and expect FRC will be useful for many concurrent C++ software systems.

CCS Concepts • **Software and its engineering → Garbage collection**;

Keywords Reference counting, memory management, C++

ACM Reference Format:
Charles Tripp, David Hyde, and Benjamin Grossman-Ponemon. 2018. FRC: A High-Performance Concurrent Parallel Deferred Reference Counter for C++. In *Proceedings of 2018 ACM SIGPLAN International Symposium on Memory Management (ISMM'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3210563.3210569

ISMM'18, June 18, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5801-9/18/06…$15.00
https://doi.org/10.1145/3210563.3210569

1 Introduction

Memory management is central to the design of programming languages. Efficient allocation, manipulation, and reclamation of memory is essential for the performance of large-scale software systems. A variety of strategies for memory management are used in programming languages; Java, Python, JavaScript, PHP, C#, Ruby, Go and others all use automatic garbage collection. Garbage collection [15] relieves programmers of the burden of manually managing program memory, reducing memory errors and misuse [21]. This can be particularly important for applications where memory errors can lead to significant security issues. Garbage collection also allows highly concurrent data structures to be implemented without specific provisions for preventing use-after-free, leaks, or other memory errors. However, C, C++, and other performance-centric unmanaged languages have eschewed the potential overheads of automatic memory management in favor of placing the burden on the programmer.

We present FRC, a *fast reference counter* for C++ and other unmanaged languages. FRC brings efficient deferred reference counting to C++ with a drop-in replacement for the standard library's shared pointer implementation. We compare FRC to Boost and the standard library because they are the current standards for automatic memory management in C++. We emphasize that C++ is significantly different from languages such as Java, where reference counting has been carefully studied; this requires, for example, producing new benchmarks that are tailored specifically for C++ reference counting[1]. In contrast to naïve reference counting, FRC uses a variant of deferred reference counting which achieves performance competitive against typical C++ reference counting techniques. In particular, FRC can outpace the

[1]The authors did not find existing C++ reference-counted pointer benchmarks in the literature or on the web, and adapting other existing C++ codes would require substantial rewrites in order to use smart pointers.

C++ standard library and Boost reference-counted pointer implementations by orders of magnitude.

FRC builds upon the innovations in Bacon et al.'s two-phase concurrent deferred reference counter [2], and Deutsch and Bobrow's deferred counter [10]. However, we do not use a separate collector thread; instead, mutator threads participate in collection cooperatively, concurrently, and incrementally. Our implementation allows many threads to concurrently participate in collection. While we use count deferral [2, 10], we do not defer increments, avoid Deutsch and Bobrow's zero-count table, and do not coalesce counts. Further, FRC maintains a per-thread array of root references, allowing root references to be acquired and released without modifying counts or pausing the mutator thread. To the authors' knowledge, FRC is the first production-ready C++ implementation of such a reference counter in the literature.

This paper makes the following contributions: 1) we present a concurrent parallel partially-deferred reference counter for unmanaged languages, and its implementation in C++; 2) in support of our reference counter, we present a novel data structure, the static tree router; 3) we present and evaluate a set of microbenchmarks for memory management methods in C++ for concurrent data structures; and 4) throughout, we describe performance-enhancing optimizations that could be incorporated into existing memory management systems, such as the C++ standard library's smart pointers.

2 Background

There are several paradigms for managing memory in C++. Manual memory management can be performed using the `malloc()` and `free()` functions inherited from C, or the `new` and `delete` keywords in C++. Manual memory management can be highly efficient as the overhead of determining when and what memory to reclaim is placed on the programmer, rather than the program. However, for concurrent software systems, it is a well-known challenge to ensure correct memory behavior in all cases. A popular abstraction in C++ is the resource-acquisition-is-initialization (RAII) paradigm, where raw resources like pointers are wrapped in scoped variables which trigger automatic allocation and deletion of memory when entering and exiting scopes. As a convenience for C++ programmers, the RAII approach of binding resources to scope lifetimes helps to ensure memory correctness and avoid leaks. However, manual and RAII memory management become increasingly difficult as concurrency increases. Another approach for C++ memory management is to use an existing garbage collection library, such as the conservative tracing BDW collector [7] or the conservative reference counting scheme of Shahriyar et al. [19] that could be implemented in C++. Conservative collection schemes require less cooperation from the compiler and mutators; however, they still have to stop the mutators to examine stack slots and registers, which is avoided in FRC by publishing root references to thread-local arrays. Unlike managed languages where it is easy to filter out ambiguous references [19], for unmanaged languages supporting raw pointers, the inherent imprecision of conservative garbage collection may also be problematic for long-running programs [17]; the interested reader is referred to Boehm [5] and Shahriyar et al. [19].

The other major C++ memory management paradigm is reference counting. First proposed in 1960 [8], reference counting maintains a count of references to an object. When an object's reference count reaches zero, the object is collected. Naïve reference counting immediately processes each pointer change, updating that object's count and collecting it if necessary. There is significant cost to this naïve approach since any pointer mutation incurs overhead from the reference counter. As observed in Shahriyar et al. [18], even with optimizations like ignoring changes to stacks and registers [10], and many changes to heap references [14], a standard reference counter can be over 30% slower than a tracing garbage collector. Contention on object counts serializes accesses to objects, creating a scalability bottleneck. These performance characteristics have caused traditional reference counting to be less popular; however, reference counting has several advantages. Since garbage is immediately collected, there are no long pause times associated with tracing garbage collection. Reference counting systems can easily be extended to work with memory shared across threads by atomically incrementing and decrementing the reference count. Further, only the counts of accessed memory need to be scanned or modified. For these reasons, as well as the relative ease of implementation compared to tracing garbage collection [12, 18], some modern languages like Python and Rust support or explicitly use reference counting. This is also the case for C++, which has included reference counted "smart pointers" in the standard library since C++11. Also noteworthy is Objective-C's transition to Automatic Reference Counting (ARC), which is also used in Swift. On the other hand, reference counting is well-known to have issues such as long pause times when freeing recursive data structures (e.g. the root node of a tree) as well as requiring additional techniques like trial deletion in order to collect cyclic garbage [11].

We emphasize deferred reference counting. First proposed by Deutsch and Bobrow [10], deferred reference counting departs from naïve counting in that counting and collection operations are not immediately applied. Instead, increments and decrements are appended to a log and are applied at a later time. The collection phase of the algorithm usually involves collecting all objects in the zero count table (ZCT), a structure that contains objects with reference counts that have been decremented to zero. The ZCT is not a necessary data structure for deferred reference counting; for example, one may instead use stack maps and achieve performant implementations [2, 18, 20]. We note that in an unmanaged language like C++, stack maps may be less practical [12].

Another relevant development in memory management is hazard pointers [16], which are designed to protect against premature reclamation under optimistic concurrency scenarios (especially with concurrent data structures). A thread can acquire a hazard pointer to a member of a data structure before attempting to use that member. As long as the thread maintains ownership of the hazard pointer, no other thread will reclaim the underlying memory. In contrast to FRC and other automatic memory managers, hazard pointers require manual calls to a `free()` function which defers reclamation until no hazard pointers point to the freed memory.

In the recent memory management literature, we highlight Shahriyar et al. [18] and Shahriyar et al. [20], which together present a reference counting system that is comparable in performance to tracing garbage collection. They observe, for instance, that heap organization is critical to reference counting performance, and find that Immix's [4] heap layout enables superior performance over free list, regional, or contiguous organizations. Aside from heap layout, they demonstrate the benefit of allocating objects as dead in order to elide count operations for short-lived objects. They also find that ignoring count operations on an object beyond a specified maximum count saves significant work, and they show how to correct these "stuck" counts when tracing the heap during cycle collection. Moreover, both papers show evidence for comparable or improved performance over mark-and-sweep garbage collection, meriting additional research into modern, high-performance reference counting.

3 FRC: A Fast Reference Counter

In this section, we describe FRC's three types of pointers and FRC's design and implementation.

3.1 Shared, Atomic, and Private Pointers

FRC supports three different types of smart pointers: **shared**, **atomic**, and **private**.

Shared pointers may be used anywhere a reference to a concurrent object is desired; they are comparable to shared pointers in the C++ standard library. To this end, shared pointers are used to manage a shared object's lifetime by maintaining a reference count for each object. Shared pointers protect this count from data races so that multiple threads can create and destroy references to it concurrently. However, mutation of shared pointers in both FRC and the C++ standard library's implementation is not synchronized, so the programmer must ensure that only one thread mutates a shared pointer at a time.

The C++ standard library's implementation of the shared pointer type stores two data structures: the underlying raw pointer, and a control block that stores reference count information. Operations on the reference count are atomic, which provides a weak thread safety guarantee. However, this offers no thread safety on the underlying object, and additional protections are required to allow it to be mutated while being concurrently read.

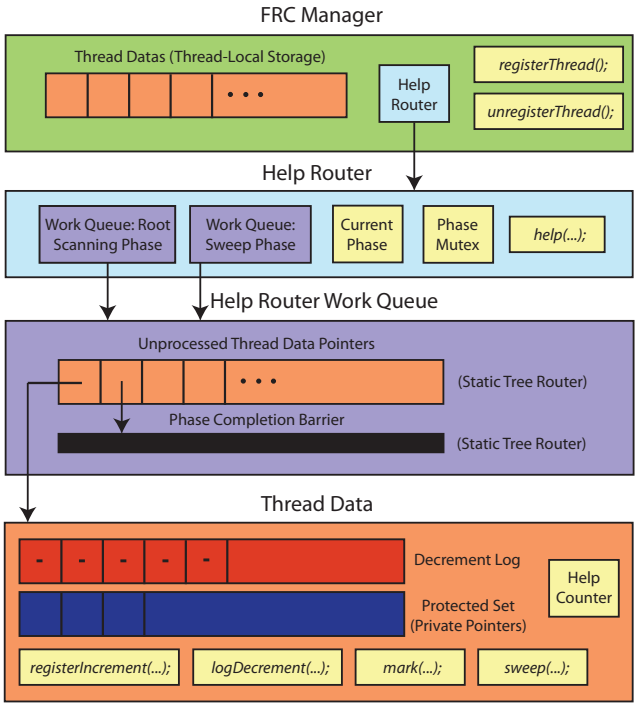


Figure 1. The hierarchy of FRC's internal data structures.

In contrast, FRC's shared pointer implementation allows concurrent readers to access a shared pointer while it is mutated. Although a small overhead is incurred to allow this, it also enables FRC's shared pointers to be used in concurrently-readable data structures without additional synchronization.

Additionally, FRC provides **atomic pointers**, which are completely thread-safe. Atomic pointers apply increments and defer decrements in the same way as shared pointers do. Atomic pointers are similar in function to Boost's atomic shared pointer, as well as explicit atomic accesses to the standard library's shared pointers in C++11.

The final type of pointer supplied by FRC is **private pointers**: thread-private pointers to shared objects which publish their referent's addresses to a thread-local root set. Private pointers incur very little synchronization overhead and rarely execute any atomic instructions. Thus, private pointers can be used to efficiently hold stack-allocated references while traversing concurrent data structures. Private pointers do not have a clear analogue in the standard library or Boost, but are similar to hazard pointers in design and goals. Further, this design allows mutator threads to read and traverse data structures without barriers or pauses. Because counts are not typically mutated by private pointers, threads concurrently accessing a shared object via private pointers see nearly zero contention on that object's count. All private pointers are scanned by the reference counting algorithm

before applying decrements or collecting objects with zero counts. Thus, private pointers prevent the collection of their target without modifying the target's count. In this respect, private pointers function similarly to hazard pointers, or root references in a tracing garbage collector.

Many accesses to concurrent data structures involve traversing a series of shared objects, such as nodes in a linked list or binary search tree. Traditional reference counting would cause contention on the counts of shared objects while traversing these data structures, see e.g. Johnson et al. [13]. In contrast, private pointers allow a wide variety of concurrent data structures to be traversed efficiently because they elide these mutations. This makes private pointers quite efficient, approaching the performance of raw pointers (see Section 4). Similar to a fixed-size stack, FRC has a fixed maximum number of private pointers per thread. Thus, the amount of work required to protect private pointers is proportional to the number of managed threads and the per-thread private pointer set size. In our tests, we used a set size of 128[2]. While all private pointers are scanned on each collection cycle, Figure 8b suggests that the set size could be increased without greatly impacting collection efficiency.

3.2 Main Algorithm

The core algorithm of FRC is divided into two global phases: **root scanning** and **sweeping**. As in Bacon et al. [2], we refer to the completion of both a root scan and subsequent sweep as a collection **epoch**. While threads perform each phase asynchronously, we require all threads to complete a phase before proceeding to the next one. The Appendix contains pseudocode for the procedures described here.

In the **root scanning** phase of the FRC algorithm, each thread's private pointers are scanned, and temporary increments are issued to their referents. Matching decrements are logged to later undo these increments during the next epoch. Delaying the application of these decrements for an epoch allows objects that remain in the protected root set (referents of private pointers) to be retained across epochs.

A synchronization method similar to a stack barrier is employed to ensure that the protected address is valid. As shown in Listing 1, when setting a private pointer from a non-private pointer, the mutator publishes a special busy signal to the private pointer slot, reads the source address, and then stores it in the root set. If a busy signal is encountered when scanning the root references, the scanning thread waits until the signal has been cleared[3]. The busy signal blocks the entry from being scanned while the reference is being read and the entry is being set, which prevents the source reference from being collected between when it is read and when the private reference is published. Once all private pointers have been scanned, the scanning phase ends and the sweep phase begins. This device appears to have minimal performance overhead compared to simply publishing the protected reference.

During the **sweep phase**, the decrements that were logged during the previous epoch are applied. Objects whose counts reach zero during this phase are collected. Because we apply increments immediately[4], and all private pointers' referents have been incremented during the root scanning phase, objects with counts that drop to zero can be safely freed. During the sweep phase, we also capture the set of decrements to be processed in the next sweep phase by inspecting each thread's decrement log and capturing all remaining entries. Decrements are pushed into a LIFO stack for processing. We found that a LIFO order avoids the breadth-first build up of decrements when freeing large tree structures we observed when using a FIFO queue. This challenge for deferral is discussed in detail in Boehm [6]. Once all the thread's decrements for this sweep have been processed, we pop off the top N entries ($1,024$ in our experiments) from each thread's decrement stack, and queue them for processing in the next sweep phase. When all of the epoch's decrements have been applied, the sweep phase ends and the next epoch begins. The ordering of the two phases and the selection of the set of decrements to apply is critical for correctness, discussed in the next section.

The simpler, partially-deferred structure of FRC's collection cycles stands in contrast to more complex designs used in other deferred reference counters, such as Deutsch and Bobrow's original deferred reference counter [10], Blackburn and McKinley's ulterior reference counting [3], Levanoni and Petrank's coalescing reference counter [14], or Shahriyar et al.'s high-performance JVM implementation [18, 20]. We immediately apply increments and eschew the zero-count table in favor of issuing temporary increments to private pointer referents. In doing so, we simplify our deferral logic and use only two phases per epoch.

3.3 Correctness

The FRC algorithm's correctness is similar to that of a standard deferred reference counter. For shared and atomic pointers, the only new concern is the immediate processing of increments. Any decrement to an object is necessarily generated after a corresponding increment is issued. Since increments are processed immediately, this means an object will only be freed when all pending increments and decrements are processed and the true reference count is zero.

Private pointers are handled specially to ensure correctness. Private pointer reference counts are incremented during the root scanning phase, and corresponding decrements

[2]The private pointer set can be dynamically sized, and we plan on doing so in a future version of FRC.

[3]Future work will replace this one-way spinlock with a more efficient method. We found that this signal was only rarely spun on.

[4]We found experimentally that using non-deferred increments and applying decrements atomically was considerably more efficient than deferring both increments and decrements. This was even true when increments and decrements were hashed into separate segments and applied non-atomically.

are deferred to the next epoch's sweep phase. References have total counts $c_t = h_t + |S|$ during the sweep phase, where c_t is the actual referent count, h_t is the recorded count, and S is the set of references scanned during the previous scanning phase. This means that $c_t = r_t + |D_t| + |S| \geq r_t$ during the sweep phase, where r_t is the actual count and D_t is the set of all pending decrements for the object at time t. So a scanned reference ($r_t > 0$) will have a positive c_t during the sweep phase, preventing collection of live objects.

Because scanning does not occur atomically, it is possible for an entry to be in the root set but to be missed by a scan. For this to happen, one of the following cases must occur. (1) The reference was removed before being scanned and no references were created from it. In this case the referent does not need to be protected. (2) The reference was added to the root set after its entry was scanned. The source reference protects the referent during this epoch. (3) The reference was removed before being scanned and after creating shared references. The shared references protect the referent. (4) The reference was removed before being scanned and after creating private references. The created reference might be missed by the scan, so we protect the referent by issuing increments and deferred decrements when copying private pointers as shown in Listing 1. Note that if the private references created in this case are references to descendant objects, then they will be protected because the referent object is live in this epoch, and if destroyed the decrements to its children will be deferred until the next epoch.

```
PrivatePointer& copyFrom(PrivatePointer const& that) {
  auto ptr = that.get();
  detail::registerIncrement(ptr);
  pin->store(ptr, memory_order_release);
  detail::registerDecrement(ptr);
  return *this;
}
// initialize from SharedPointer or AtomicPointer
auto& PrivatePointer::initialize(SharedPointer& that) {
  pin->store((void*) detail::FRCConstants::busySignal,
       memory_order_release);
  auto ptr = that.get(memory_order_acquire);
  pin->store(ptr, memory_order_release);
  return *this;
}
```

Listing 1. Private pointer protection, assignment, and initialization methods.

3.4 Data Structures and Organization

Several key data structures underlying the design of FRC are illustrated in Figure 1. At the heart of FRC is the `FRCManager`, a global object that coordinates the various collection phases across the execution threads. To use FRC, a thread first registers itself with the `FRCManager`. Registration associates a `ThreadData` object for the thread and enqueues it for processing in the next collection phase. When a thread is done interacting with FRC, it deregisters from the `FRCManager`, which detaches the `ThreadData` object from the thread.

Two data structures organize the work to be processed for threads and the state of reference counts for objects in threads: `ThreadData` and `HelpRouter`. `ThreadData` objects store a thread's log of deferred decrements, its set of private pointers, and a help counter. The help counter counts how many decrements have been logged by the thread since the thread participated in the collection process. When the counter reaches a given threshold, the thread calls the help method of the `HelpRouter`. The help method causes the mutator thread to cooperatively assist with the collection process. The exact number of decrements logged between calls to help is typically constant (32 in our experiments). However, the interval shrinks exponentially as the number of pending decrements for a thread grows. In our experiments, we used a threshold of 2^{13} decrements to begin decreasing the interval, and halved the threshold every 2^{11} entries beyond this threshold. This closed-loop control system prevents large quantities of unprocessed decrements from accumulating in the buffer faster than they are applied. If the buffer reaches a very large size, the thread makes a blocking call to help, waiting for the current collection phase to complete before returning. In our experiments, we set this threshold to 2^{17} entries and rarely saw such a call occur.

Collection work in FRC progresses at the rate at which threads issue decrements. Unlike in managed languages, there is no guarantee that a thread will "play ball" and issue decrements at a predictable rate. Instead, because the amount of collection work generated by a thread is proportional to the number of decrements it produces, the number of attempts it makes to help, and thus the amount of time it spends participating in collection will all be generally proportional. This ensures that collection progresses at a controlled rate without preempting mutator threads or stop-the-world pauses. This division of labor is similar to that found in non-deferred reference counters, with mutator threads taking on counting and destruction work proportional to the number of increments and decrements issued.

3.5 Memory Footprint

FRC's memory overhead includes uncollected garbage, unprocessed decrement log buffers, private pointer root sets, and object headers. The decrement log buffers occupy one 8-byte pointer per entry. In our tests, decrement log size typically varied between a few hundred entries to several thousands of entries per thread. The size of the decrement log can be traded off for collection performance by adjusting the help interval and the maximum decrement log size parameters. Smaller decrement log size limits yield more frequently processed epochs and hence less accumulated uncollected garbage, but as logs are processed more frequently, fewer entries will be processed per epoch, which decreases collection efficiency. The private pointer set has a fixed size

(128 entries in our tests), consuming one 8-byte pointer of memory per entry. Finally, each managed object has a header prepended to its memory. In our implementation, this header consumes 8 bytes: 4 bytes to store the object's count as an unsigned integer, and 4 bytes to store a type ID used to call the object's destructor. This header structure is similar to object headers and control blocks used in other reference counters such as `std::shared_ptr` and tracing collectors such as Java's G1 garbage collector [9].

3.6 Static Tree Router

The **static tree router** (STR) is a novel, efficient, tree data structure that serves two purposes in FRC: (1) allowing threads to find collection work to process, and (2) acting as a reference counting phase barrier.

3.6.1 Description of the Static Tree Router

An STR is a perfect binary tree, where each node contains two status bits which indicate whether its left and right children are acquired or released. False sharing is mitigated by ensuring that each node resides on a separate cache line. An STR consumes one cache line (64 bytes on our test system) for each of the $N-1$ internal nodes, where N is the number of STR inputs rounded up to the nearest power of two. In our experiments, we used STRs with 32 inputs (one per thread), for a total memory consumption of 1,984 bytes per STR.

An STR is managed by a router that acquires and releases the STR's inputs. When an input is acquired by the router, the corresponding status bit in the input's node is set. If the sibling's bit is also set, the parent node is recursively visited, propagating the information towards the root of the tree. Similarly, when an input is released, the corresponding status bit is cleared. If the sibling's bit is also clear, clearing recurses up the tree. If this clearing process reaches the root, and both root status bits are clear, then all inputs have been released. Since acquisition and release signals are propagated from leaf to root, cache line invalidation and contention are minimized. Synchronization of these `acquire` and `release` methods is achieved with simple spin locks on each node, locking from the leaf upwards. Calling the STR's lock-free `findAcquired` method randomly traverses the tree from the root to any acquired leaf node. If the traversal encounters a node which is no longer acquired, or no longer has any acquired children due to a concurrent release, it simply retries from the root.

3.6.2 Use of Static Tree Routers in FRC

When a thread calls the `HelpRouter`'s help method, it is assigned a `ThreadData` to help process. During the root scanning phase, this task consists of scanning and protecting a subset of the thread's private pointers. In the sweep phase, the helping thread applies a subset of the thread's deferred decrements. Upon completion of the work, the `HelpRouter` checks to see if all `ThreadData`s have been processed so that it can move to the next collection phase. In support of these tasks, FRC uses two STRs for each collection phase: one acts as a router to find remaining unclaimed tasks during calls to the help function, and the other acts as a barrier to detect when all tasks have been completed for a collection phase.

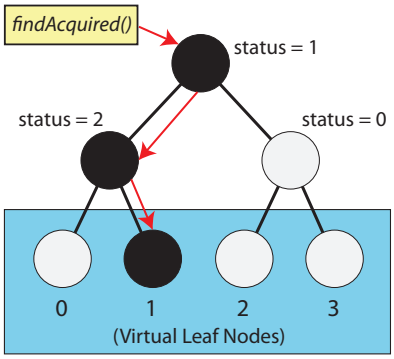


Figure 2. An STR with four inputs. Each interior node has a two-bit status flag indicating which of its children are acquired: neither (0), left (1), right (2), or both (3). Black nodes are acquired or have acquired children. Each node's status is stored in its parent node so that the statuses of both siblings reside in the same cache line. `findAcquired` finds an acquired leaf in $O(\log n)$ by traversing via the status flag.

In FRC, STR inputs are queues of work, with each queue containing a list of `ThreadData` pointers[5]. When a phase begins, all inputs are acquired in the STR corresponding to unprocessed work. When a `ThreadData` is processed successfully, its index is released from the first STR and is acquired in the phase barrier STR. Only when the phase barrier STR is fully acquired (identified by checking the root node's status flag) does FRC advance to the next collection phase.

In particular, helping threads can be routed to unclaimed tasks by traversing the static tree router from root-to-leaf. Without locking, the status bits of the root node are read. If neither child bit is set, we know that all inputs are released and no work remains unclaimed. Otherwise, we recurse randomly to one of the child nodes with a set status bit. If we reach a leaf node, we try to claim a remaining task from the corresponding queue. If this empties the queue, we release the corresponding input. Thus, the STR acts as a concurrent work queue when distributing collection tasks to helping threads. We believe that the STR could be applied to improve the scalability and efficiency of work queues in other applications, such as work-stealing and fork-join frameworks.

4 Results and Discussion

In this section, we illustrate the usage of FRC and detail our microbenchmark evaluations. We present high-level descriptions of the tests and algorithms we created to this end.

[5]Using queues rather than `ThreadData` pointers directly allows FRC to handle having more registered threads than the number of STR inputs.

4.1 Usability

Usability is key for any interface. `frc::SharedPointer` implements the same interface as `std::shared_ptr`, making it a drop-in replacement. The only requirement of FRC is that a mutator thread registers itself with FRC, via a one-line call. `frc::AtomicPointer` additionally allows the programmer to elide external synchronization code for concurrent mutations. For many data structures and algorithms, as shown in the following examples, `frc::PrivatePointer` is effective and easy to use. Intended users of FRC are C++ developers working with concurrent programs and data structures.

```
bool insert(Key key, Value value) {
  frc::PrivatePointer<Node> parent, curr, replacement;
  frc::AtomicPointer<Node>* currRef;
  replacement = nullptr;
  retry:
  parent = rootParent;
  currRef = &parent->left;
  for(;;) {
    curr = *currRef;
    if(curr == nullptr) {
      // Lock parent and insert node
      if(replacement == nullptr)
        replacement.make(key, value);
      auto guard = parent->lock.acquire();
      if(!parent->current) // Certify
        goto retry;
      if(currRef == nullptr) {
        swap(*currRef, replacement);
        return true; // Added node
      }
      curr = *currRef; // Resume
    }

    // Recursively find insertion site
    if(key == curr->key) {
      // Lock and overwrite existing
      auto guard = curr->lock.acquire();
      if(!curr->current) // Certify
        goto retry;
      curr->value = value;
      return false;
    } else if(key < curr->key)
      currRef = &curr->left; // Recurse left
    else if(key > curr->key)
      currRef = &curr->right; // Recurse right
    parent = curr;
  }
}
```

Listing 2. Insertion code for the concurrent BST implementation used in our experiments. As is standard practice, `currRef` requires no extra synchronization and so is a raw pointer.

To give a sense of the usability of FRC, we show in Listing 2 our implementation of an insertion function for a concurrent BST. The BST is searched recursively to find the insertion site, whereupon the function creates a new node or updates the value of an existing node. In contrast to the standard library or Boost, FRC allows the use of `frc::PrivatePointers`, which are used to safely traverse the tree without modifying object counts. Additionally, FRC's concurrently-readable pointers allow operations like `parent = curr` to be thread-safe without external synchronization. This allows readers to continue without interference from mutators. The simplicity, efficiency, and safety offered by FRC make it a compelling design for use in large-scale concurrent systems and data structures.

4.2 Experimental Design

All experiments were performed on HP Z620 workstations, equipped with two Intel Xeon E5-2660 CPUs at 2.20GHz. Each CPU has 8 cores, with 2 hyper-threads each, providing 32 hardware threads. The CPU scaling governor is set to performance mode in order to obtain the maximum static clock frequency for each core. Each workstation runs a standard desktop installation of Debian 9.1.

In all multithreaded tests, we used no more threads than the hardware concurrency of our test machines. Each thread's affinity is bound to a unique logical CPU. Unless otherwise stated, N refers to the number of threads used for a given test, and M refers the number of operations per thread to be performed. We compiled our binaries using g++ 6.3.0 with compiler flags `-O3 -mtune=sandybridge -std=c++14`. Timing was performed using `std::chrono::high_resolution_clock`[6]. We used Boost 1.65.0 Beta 1. In any plot, **error bars** indicate ±1 standard deviation from the mean observed value over all trials. All experiments with error bars were repeated 100 times, except for the two multi-operation tests, where trials were repeated only 5 times due to greatly increased runtimes of the tests.

4.3 General Performance Comparisons

In the following microbenchmarks, we show specific performance characteristics of FRC and compare these to other smart pointer implementations. Specifically, we investigated:

- *Contention on a Single Object:* In concurrent applications, it is often the case that multiple threads contend on a single object. For example, all threads must contend on the root node of a tree data structure when performing updates. Thus, we tested allocating a single pointer and then launching threads that created their own references to the pointer. We measured the combined execution time of each thread.
- *Pointer Setting:* We measured the time for threads to set pointers to existing objects. Each thread allocates three pointers, A, B, and C. Then, for M iterations, each thread sets their pointer $C = A$, then $C = B$. We recorded each thread's total

[6]While this is limited to a resolution of a few nanoseconds, we generally circumvent potential clock resolution issues by only recording the total time for M runs and computing an average after-the-fact.

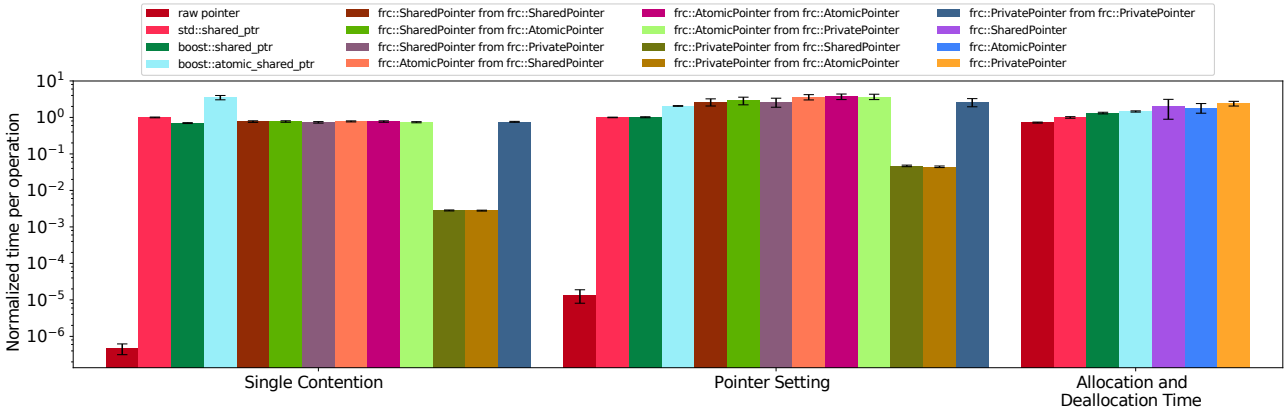


Figure 3. Normalized time per operation for each of the general performance comparisons. Each test used $N = 32$, $M = 100,000$. Timing includes both mutation and collection time in order to fully compare with manual management.

runtime, including mutation and collection time (increments and decrements occur when mutating the pointers).

- *Allocation and Deallocation Timing:* We measured the time to allocate and deallocate each pointer type. Each thread allocates and deallocates its own set of M pointers. We record the time for each thread to allocate and deallocate the pointers.

In Figure 3, we show the performance of these tests for a fixed $N = 32$ and $M = 100,000$. We normalized results by the time per operation when using `std::shared_ptr`. For contention on a single object, the worst-performing pointer type was `boost::atomic_shared_ptr`, due to the extra atomic operations during access of the object. `std::shared_ptr` and `boost::shared_ptr` both performed similarly, but neither offered any atomic protection when referencing a contended object. When contending on any of FRC's smart pointers, `frc::SharedPointer` and `frc::AtomicPointer` showed comparable performance to `std::shared_ptr`, while `frc::PrivatePointer` avoided referent count contention and thus showed performance several orders of magnitude closer to raw pointers, except in the case of creating a reference of one `frc::PrivatePointer` to another, which incurs an increment and decrement as discussed in Section 3.3.

For setting pointers, `boost::atomic_shared_ptr` was over 2x slower than `std::shared_ptr` and `boost::shared_ptr` due to atomic load and store operations. Performance further deteriorated with C as a `frc::SharedPointer` or `frc::AtomicPointer`. Logging and later processing decrements is the primary cost of deferred reference counting. However, when creating C as a `frc::PrivatePointer` from another pointer type, we saw excellent performance.

Allocation and deallocation performance was comparable for all pointer types. `std::shared_ptr` and `boost::shared_ptr` were nearly as fast as raw pointers, as there was no contention on the reference counts. The extra atomic operations of `boost::atomic_shared_ptr` incurred a small cost. `frc::SharedPointer` and `frc::AtomicPointer` performed similarly to `boost::atomic_shared_ptr` on allocation, but were slower when we measured deallocation time. This difference is due to the overhead of deferring decrements. `frc::PrivatePointer` had the slowest performance overall. This is because when constructing a `frc::PrivatePointer`, we allocate an object with a count of 1, then log a decrement (so that the object has net-zero count). `frc::PrivatePointers` are designed for fast access to objects, not fast allocation of memory; we recommend using `frc::SharedPointers` or `frc::AtomicPointers` when allocating new memory.

4.4 Concurrent Data Structures

FRC is well-suited for use in concurrent data structures. To demonstrate this, we implemented several such structures: a binary search tree (BST), a B-Tree, and a hash map. All concurrent data structures were templatized on two pointer types: `SharedPtr`, a generic shared pointer type, and `PrivatePtr`, a generic private pointer type. Using these templates, we instantiated data structures using each pointer type. There is no private pointer analogue in the standard library or Boost, so we used the shared pointer types for both template parameters when using `std::shared_ptr` and `boost::atomic_shared_ptr`. `std::shared_ptr` has no thread safety when accessing its data, so we wrapped access to these pointers with atomic loads and stores.

Each data structure is highly concurrent; all reads are wait-free, while writes use minimal two-phase locking accompanied by a certification step which verifies that the locked object is the current version before mutating or replacing it. For each data structure, we performed the following tests:

- *Read:* We randomly constructed a data structure with the keys 0 to 100,000. We then launched N threads, each performing M concurrent reads.

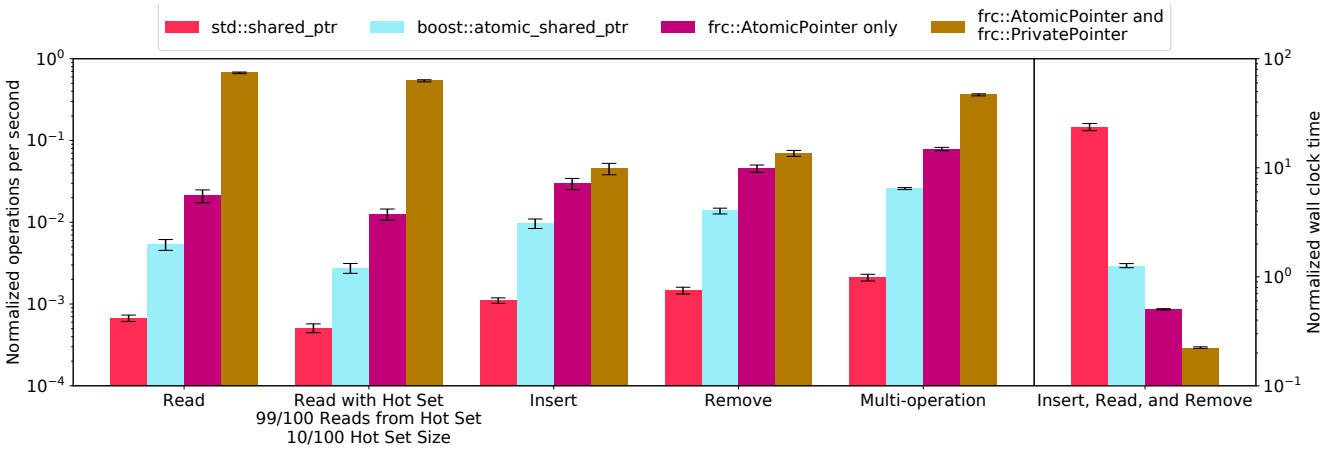


Figure 4. Normalized throughput (left) and wall clock time (right) on a concurrent BST. Each test used $N = 32, M = 100,000$.

• *Read with Hot Set:* As a more realistic version of the read test, we let the first 10% of our data set denote a "hot set." In the test, each thread generated a sequence of M reads from the tree, where 99% of reads came from the hot set [7].
• *Insert:* We randomized the keys 0 to NM and assigned M of these random keys to each thread. Each thread then concurrently inserted its keys.
• *Remove:* We randomly constructed a data structure with keys 0 to $10,000N$. Each thread removed M unique keys.

For each of the above tests, we recorded the wall-clock time taken to perform the operations, and inverted to get throughput. We normalized by the timing for a *non-concurrent* implementation with manually-managed memory and raw pointers[8]. The tests were run in two cases. First, we ran each test at hardware concurrency with $M = 100,000$ operations per thread. Second, we fixed the workload at $M = 10,000$ operations per thread while we varied the number of threads.

We also performed these multi-operation tests:

• *Insert, Read, and Remove:* As a comprehensive test, we combined the above operations. Using real-world string datasets obtained from Askitis [1], we built a data structure containing the 28772169 strings from the `distinct_1` set. We then queried it with the 177999203 strings from `skew1_1` set, and then destroyed the structure. Here, $N = 32$. We recorded the wall-clock time required to complete all three operations.
• *Mixed Operations:* Using the data from Askitis [1], we built a data structure with the strings from `distinct_1`. Then we evenly divided queries from `skew1_1` among $N = 32$ threads. Each thread then performed a random sequence of operations with 95% reads, 4% inserts, and 1% removes. We recorded the wall-clock time per operation.

In the following sections, we show the results of tests for each data structure. For each data structure, we also briefly discuss our concurrent design and implementation. For full details, the reader is referred to the source code: https://github.com/terraindata/frc.

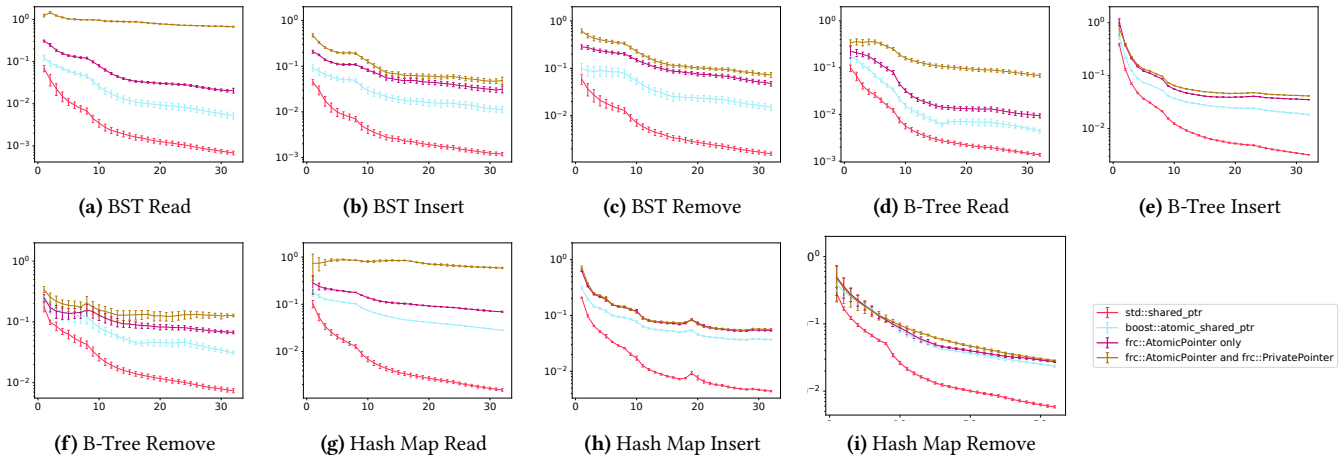
[7] Similar results were obtained for 50% and 90% (not presented here).
[8] Thus, our results are quite conservative, since a concurrent raw pointer implementation requires locking etc., incurring contention and other overhead.

4.4.1 Binary Search Tree

We implemented a concurrently-readable BST. Our BST allows for non-blocking reads and only locks leaf nodes for insert operations. For remove operations on internal nodes, the tree locks downward from the pivot to the minimal descendant, building a new path as it descends. We atomically insert the new path into the tree by swapping the pointer to the pivot node. This design allows for non-blocking reads to proceed during removals without returning false negatives.

Figure 4 shows the performance of each pointer type on the BST for each test. In each experiment, `std::shared_ptr`, `boost::atomic_shared_ptr`, and `frc::AtomicPointer` all contend on node counts while traversing the tree. Since each query begins at the root node, its count experiences particularly heavy contention. `std::shared_ptr` was hardest hit by this contention, while `boost::atomic_shared_ptr` performed slightly better, and `frc::AtomicPointer` better still. `frc::AtomicPointer` may experience slightly lower contention due to the temporal offset of decrement deferral.

Using `frc::PrivatePointer` to elide count mutations resulted in the highest performance in all cases. For both read benchmarks, we recorded throughput approximately two orders of magnitude higher than `std::shared_ptr`, while insert and remove benchmarks recorded smaller 4x to 25x increases in throughput. The benefit of `frc::PrivatePointer` is reduced for mutating operations because some contention occurs on locks, and shared pointer counts must be mutated for the affected nodes. However, because all these operations traverse the BST, they all benefit materially from using `frc::PrivatePointer`. This benefit greatly outweighed the additional cost of deferring reference counts,



(a) BST Read **(b)** BST Insert **(c)** BST Remove **(d)** B-Tree Read **(e)** B-Tree Insert

(f) B-Tree Remove **(g)** Hash Map Read **(h)** Hash Map Insert **(i)** Hash Map Remove

Figure 5. Performance results for the read, insert, and remove tests on each of our concurrent data structures, varying the number of threads. Each plot shows the normalized operations per second versus the number of threads, which was varied from $N = 1$ to 32. Each thread performed $M = 10,000$ operations, so the total work NM increased with N; growing contention evidently prevents linear scaling (a flat line).

even for remove operations which can hold many locks simultaneously. In fact, `frc::PrivatePointer` was able to come within a factor of 2 of the performance of a raw pointer.

We show in Figure 5 the performance of the concurrent BST on the read, insert, and remove tests as we vary the number of threads. As expected, we saw poor scaling from most pointer types while recording nearly linear scaling of reads when using `frc::PrivatePointer` (along with improved scaling for insert and remove operations). This performance is due to reduced serialization and contention on object counts when using thread private pointers.

4.4.2 B-Tree

We tested FRC with a concurrently-readable B-Tree, using two-phase locking and certification to achieve write synchronization and consistency. The B-Tree's index blocks had 512 children and leaf blocks had 32 key-value pairs. To insert or remove keys from the tree, we traverse down the tree, find the desired leaf block, lock it, and certify it is valid (the current version of that block). Next, we construct a replacement block or blocks (when splitting). We mark the previous block to be invalid and then proceed up the tree, locking and certifying index blocks, and performing the necessary replacements and splits. If we encounter an invalid block, we record our location and re-traverse down the tree to our last position to find the current version of the block. Reads are wait-free, with no locking or certification steps.

In Figure 6, we show the performance of several smart pointer implementations on each of our concurrent data structure tests. As with the BST, the combination of `frc::AtomicPointer` and `frc::PrivatePointer` performed best across all tests. However, we observed a tightening of performance on insertion and removal operations compared to those on a BST. This is likely because B-Trees are flatter data structures, containing many more children than those in a BST. Thus, operations on the tree are more likely to conflict, causing higher lock and count contention.

In Figure 5, we show the performance of the concurrent B-Tree as we varied the number of threads on the read, insert, and remove tests. Once again, we saw `frc::PrivatePointer` scale nearly linearly for read operations, but achieved little benefit for insert and remove operations.

4.4.3 Hash Map

In addition to tree structures, we performed benchmarks on a chained partitioned hash map. The hash map is partitioned into segments, each of which is a concurrently-readable chained hash map that serializes mutations with a lock.

In Figure 7, we show the test results for the hash map. As with the B-Tree, we saw comparable performance on insertions and removals when using FRC. This is because the hash map experiences high locking contention on each segment, overshadowing the benefits of reduced count contention. However, read throughput was again more than an order of magnitude higher when using `frc::PrivatePointer`.

In Figure 5, we see that insert and remove scaling was comparable between all pointer types, likely due to contention on segment locks. However, reads once again scaled very well when using `frc::PrivatePointer`.

4.5 FRC Timing Breakdown

We investigated the time spent in the FRC algorithm for the Insert, Read, and Remove benchmark on a BST. We logged the

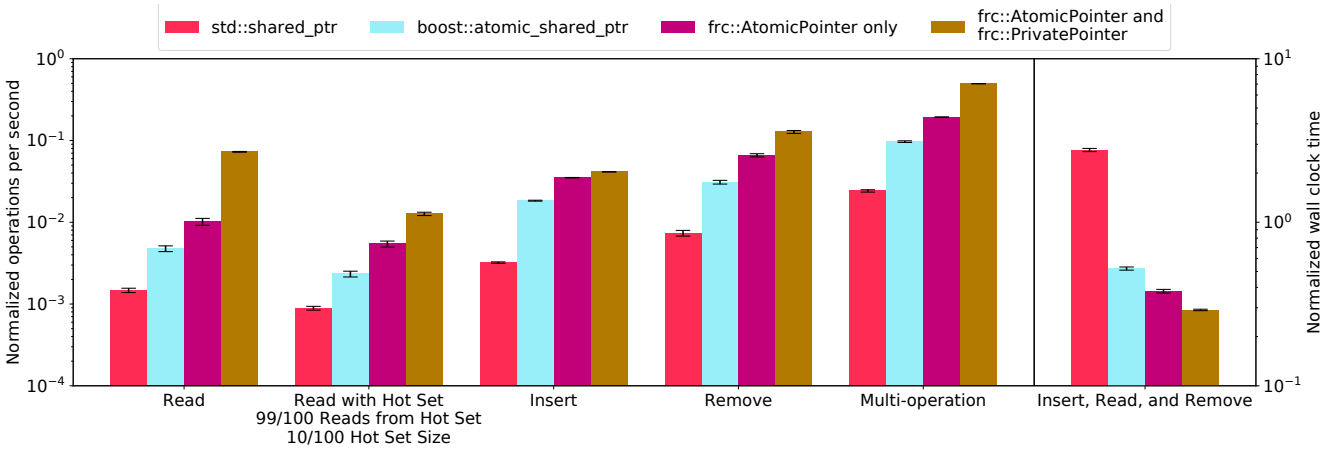


Figure 6. Normalized throughput (left) and wall clock time (right) on a concurrent B-Tree. Each test used $N = 32, M = 100,000$.

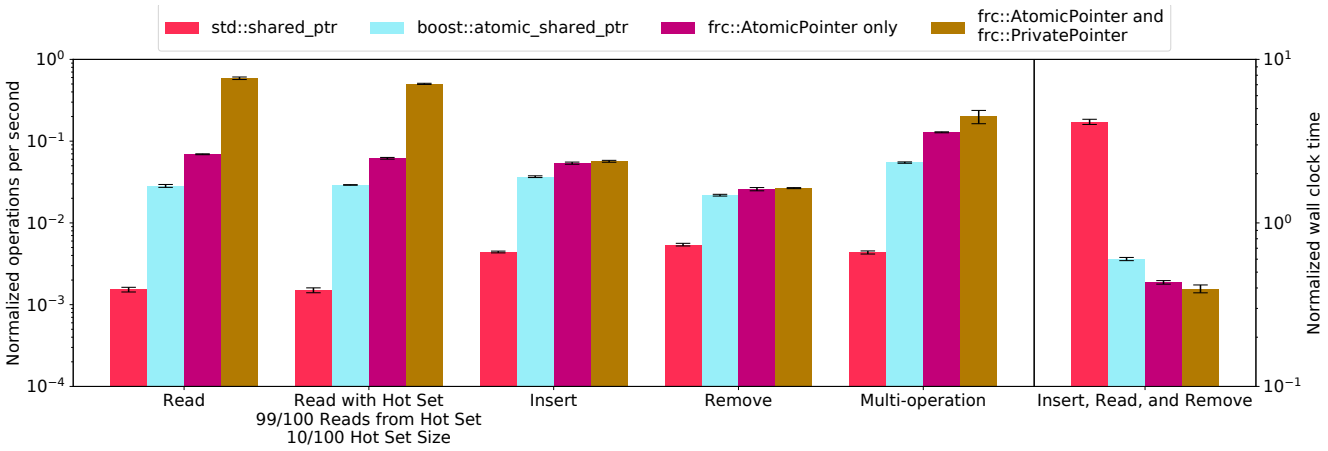


Figure 7. Normalized throughput (left) and wall clock time (right) on a concurrent hash map. $N = 32, M = 100,000$.

individual "pause times" for the test, as well as the time spent applying increments to reference counts, logging decrements into buffers, and performing each collection phase.

In Figure 8a, we show a breakdown of the total time spent in FRC for the Insert, Read, and Remove benchmark on a BST. These "pause times" include both the fast and slow paths of FRC, including for example the time to log a decrement without performing any collection work. These pause times were typically brief (especially in the fast path), with longer times corresponding to the destruction of complex object chains, including when the BST's root node was destructed. We reiterate that FRC is fully concurrent; these are thread-local, not stop-the-world pauses.

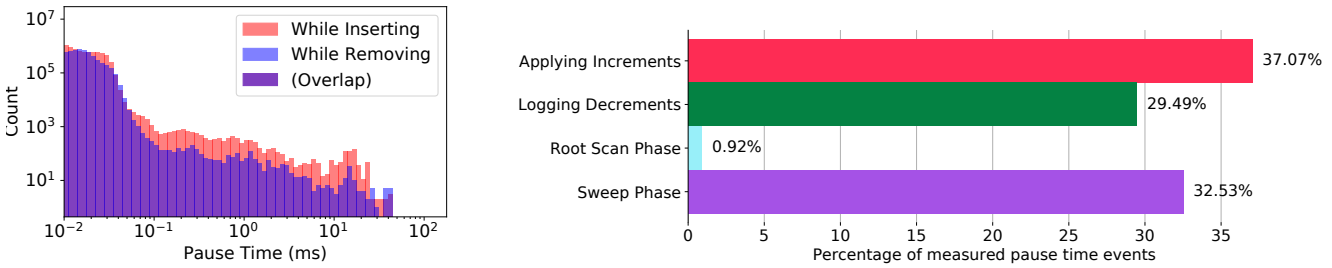
We show the percentage breakdown of overheads in Figure 8b. Most time was spent registering increments, which are atomic instructions executed e.g. upon any pointer allocation. Unsurprisingly, FRC's sweep phase dominated the root scanning phase. This example gives an indication of the division between mutator and collection times for FRC.

4.6 Slop Size Estimation

The primary purpose of a garbage collector is to prevent the accumulation of dead objects in memory during runtime. We investigated FRC's handling of this task for two tests. In Figure 9, we show the number of objects destroyed in each sweep phase during the operation of FRC on these two tasks.

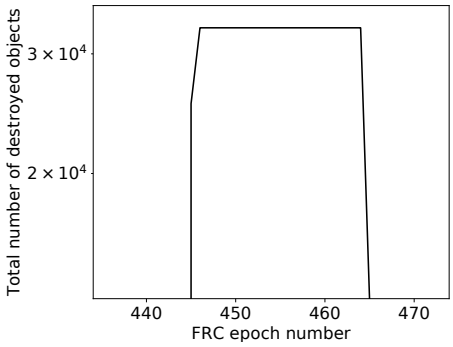
First, we considered an extreme case where each of 32 threads repeatedly allocated and deallocated an `frc::AtomicPointer`. As the number of deletes per thread per epoch is capped at $1,024$ in our implementation, we observed a plateau of $32,768 = 32 * 1024$ while slop is generated. The help interval deterministically decreases to 1 as the decrement buffer fills, leading it to trigger on every decrement if the buffer is nearly full.

As a more realistic example, we similarly evaluated the BST remove test. FRC collected the slop from the remove operations within 100 epochs. The large plateau on the right (peaking at $1,942$ objects) corresponds to the destruction of the BST. When the last thread released its reference to

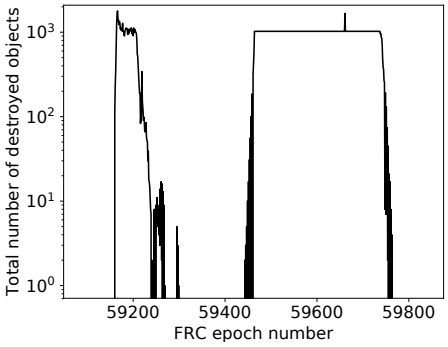


(a) (b)

Figure 8. Pause times for the Insert, Read, and Remove test on a concurrent BST. We show a histogram of the pause times during the test and a breakdown of these times by function. (Left) Pauses while constructing and destructing the BST follow a similar distribution; we elide pause times less than 1μs, which make up 99.5% of the pauses. The longest pause time is 44ms.



(a) Results from a simple for loop test.



(b) Results from a BST removal test.

Figure 9. Evaluation of our proxy metric for slop size in two extreme cases. *Top*: a loop generates one dead object per iteration—a constant, small stream of slop. *Bottom*: 32,000 nodes are removed from a 320,000 node BST, before the BST is freed entirely—a sudden, large burst of slop.

the root node, the master thread began cleanup of the tree. FRC's collection mechanism is consistently triggered until the entire tree is freed.

5 Conclusion and Future Work

As core counts and concurrency increase, scalable concurrent memory management is increasingly important for high-performance programming. While C++ is commonly used for high-performance systems programming, it lacks an easy and efficient method for automatically managing the memory for these data structures. FRC delivers exactly this capability to C++ with a drop-in replacement for `std::shared_ptr`. As a fully concurrent parallel reference counting system, FRC offers attractive performance and a convenient interface to the end user. Processing increments immediately yields a partially deferred scheme that appears well-suited for a language like C++. By deferring decrements, FRC is able to safely use thread-private pointers, which in turn elide most mutations of object counts. This elision significantly reduces contention on counts, allowing for highly scalable traversal of concurrent data structures. While a price is paid to defer decrements, the performance costs are small and typically greatly outweighed by the benefits gained from the use of thread-private `PrivatePtr`'s. In addition, the static tree router serves as an efficient barrier and router for collection work, contributing to FRC's performance.

Our future work on FRC will expand on improving the efficiency of the scan and sweep phases, increasing the performance and balance of work distribution, experimenting with coalescing the temporary increments which protect private pointers, testing alternate methods of protecting pointers, and deeper and more flexible integrations with custom allocators. Additionally, cycles may be collected by combining FRC with trial-deletion or other cycle-detection methods.

Acknowledgments

The authors thank Xi Yang and Abhishek Kulkarni for helpful and insightful ideas, comments, discussions, and suggestions.

A Appendix

In this appendix we include pseudocode for several key functions in FRC. In Algorithm 1, we show the logic for how work is identified and processed when a thread pauses to help with collection. The static tree router for the current phase is queried to find any remaining work; upon success, the identified subqueue of remaining work is processed by the thread, calling scan or sweep as appropriate on an identified `ThreadData`. Care is taken to ensure proper synchronization. In Algorithm 2, we show the logic for the scan and sweep phases of FRC, for a given `ThreadData` that has work to process. In the scan phase, temporary increments are issued to the referents of the `ThreadData`'s private pointers in order to protect from subsequent deletion in the sweep phase. In the sweep phase, decrements logged during the previous epoch are applied, and objects whose counts reach zero are collected. Care is taken to ensure proper synchronization. Selected implementation details are elided here for clarity of presentation. For full details, the reader is referred to the main portion of the paper and online to the published source code: https://github.com/terraindata/frc.

Algorithm 1 FRC Help Routing Logic

```
1:  procedure HELP
2:      phase ← currentPhase
3:      subqueueIndex ← queues[phase].router.findAcquired()
4:      if subqueueIndex == notFound then
5:          return false
6:      return tryHelpSubqueue(phase, subqueueIndex)
7:  procedure TRYHELPSUBQUEUE(phase, subqueueIndex)
8:      subqueue ← queues[phase].subqueues[subqueueIndex]
9:      subqueue.lock()
10:     if phase ≠ currentPhase || subqueue.empty() then
11:         return false // phase changed while claiming a task, or queue empty
12:     targetThreadData ← subqueue.back()
13:     threadDataFinished ← false
14:     if phase == scan then
15:         threadDataFinished ← scan(subqueueIndex, targetThreadData)
16:     else
17:         threadDataFinished ← sweep(subqueueIndex, targetThreadData)
18:     if !targetThreadData.isReadyToDestruct then
19:         enqueueThreadData(phase ∧ 1, targetThreadData) // enqueue in next phase work queue
20:     if  threadDataFinished && subqueue.numPendingTasks.fetch_sub(1) ≤ 1 // acquire release semantics
21:        && queues[phase].release(subqueueIndex) // subqueue completed: release barrier input  then
22:         phaseLock.lock()
23:         if phase == currentPhase then
24:             phase ← phase ∧ 1
25:         phaseLock.unlock()
26:         phaseConditionVariable.notify_all()
27:     if targetThreadData.isReadyToDestruct then
28:         delete targetThreadData
```

Algorithm 2 FRC Scan and Sweep Phase Logic

```
1:  procedure SCAN(subqueueIndex, targetThreadData)
2:      begin ← targetThreadData.lastScanIndex
3:      targetThreadData.lastScanIndex ← targetThreadData.lastScanIndex + protectedBlockSize
4:      lastTaskClaimed ← (targetThreadData.lastScanIndex == pinSetSize)
5:      unlockThreadData(subqueueIndex, targetThreadData, lastTaskClaimed)
6:      for i ← 0; i < protectedBlockSize; i ← i + 1 do
7:          pointer ← null
8:          do
9:              readFence()
10:             pointer ← protectedPointers[begin + i]
11:         while pointer ≠ busySignal
12:         if pointer ≠ null then
13:             getHeader(pointer).increment()
14:     // returns true iff there are no more pending scan tasks
15:     return targetThreadData.pendingScanTasks.fetch_sub(1) ≤ 1
16: procedure SWEEP(subqueueIndex, targetThreadData)
17:     begin ← targetThreadData.consumerIndex
18:     end ← begin + min(logBlockSize, targetThreadData.captureIndex − begin)
19:     targetThreadData.consumerIndex ← end
20:     lastTaskClaimed ← (end == targetThreadData.captureIndex)
21:     unlockThreadData(subqueueIndex, targetThreadData, lastTaskClaimed)
22:     for i ← begin; i < end; i ← i + 1 do
23:         header ← targetThreadData.decrementBuffer[i%bufferCapacity]
24:         header.decrement() // destructs and deletes the object if count reaches zero
25:     if threadData.pendingSweepTasks.fetch_sub(1) > 1) then // acquire release semantics
26:         return false // there are pending sweep tasks
27:     // initialize targetThreadData for next epoch
28:     targetThreadData.lastScanIndex ← 0
29:     targetThreadData.numRemainingScanBlocks ← numScanBlocks
30:     targetThreadData.captureIndex ← targetThreadData.lastHelpIndex
31:     targetThreadData.numSweepBlocks ←
            ceil((targetThreadData.captureIndex − targetThreadData.consumerIndex)/logBlockSize)
32:     return true // this was the last pending sweep task
33: procedure UNLOCKTHREADDATA(subqueueIndex, targetThreadData, lastTaskClaimed)
34:     subqueue ← queues[phase].subqueues[subqueueIndex]
35:     if lastTaskClaimed then
36:         subqueue.dequeue()
37:         if subqueue.empty() then // if subqueue is empty, release its route
38:             queues[phase].release(subqueueIndex)
39:     subqueue.unlock()
```

References

[1] Nikolas Askitis. 2012. Cache-conscious String Data Structures, Sorting and Algorithms: theory and practice. http://web.archive.org/web/20120206015921/http://www.naskitis.com/

[2] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. 2001. Java Without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM, New York, NY, USA, 92–103. https://doi.org/10.1145/378795.378819

[3] Stephen M. Blackburn and Kathryn S. McKinley. 2003. Ulterior Reference Counting: Fast Garbage Collection Without a Long Wait. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA '03)*. ACM, New York, NY, USA, 344–358. https://doi.org/10.1145/949305.949336

[4] Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *ACM Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, 22–32. https://doi.org/10.1145/1379022.1375586

[5] Hans-J. Boehm. 2002. Bounding Space Usage of Conservative Garbage Collectors. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM, New York, NY, USA, 93–100. https://doi.org/10.1145/503272.503282

[6] Hans-Juergen Boehm. 2004. The Space Cost of Lazy Reference Counting. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. ACM, New York, NY, USA, 210–219. https://doi.org/10.1145/964001.964019

[7] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Softw. Pract. Exper.* 18, 9 (Sept. 1988), 807–820. https://doi.org/10.1002/spe.4380180902

[8] George E. Collins. 1960. A Method for Overlapping and Erasure of Lists. *Commun. ACM* 3, 12 (Dec. 1960), 655–657. https://doi.org/10.1145/367487.367501

[9] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1029873.1029879

[10] L. Peter Deutsch and Daniel G. Bobrow. 1976. An Efficient, Incremental, Automatic Garbage Collector. *Commun. ACM* 19, 9 (Sept. 1976), 522–526. https://doi.org/10.1145/360336.360345

[11] Daniel Frampton. 2010. *Garbage Collection and the Case for High-level Low-level Programming*. Ph.D. Dissertation. Australian National University.

[12] Ivan Jibaja, Stephen M. Blackburn, Mohammad R. Haghighat, and Kathryn S. McKinley. 2011. Deferred Gratification: Engineering for High Performance Garbage Collection from the Get Go. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC '11)*. ACM, New York, NY, USA, 58–65. https://doi.org/10.1145/1988915.1988930

[13] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '09)*. ACM, New York, NY, USA, 24–35. https://doi.org/10.1145/1516360.1516365

[14] Yossi Levanoni and Erez Petrank. 2001. An On-the-fly Reference Counting Garbage Collector for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. ACM, New York, NY, USA, 367–380. https://doi.org/10.1145/504282.504309

[15] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (April 1960), 184–195. https://doi.org/10.1145/367177.367199

[16] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (June 2004), 491–504. https://doi.org/10.1109/TPDS.2004.8

[17] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. 2009. Precise Garbage Collection for C. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM '09)*. ACM, New York, NY, USA, 39–48. https://doi.org/10.1145/1542431.1542438

[18] Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. 2012. Down for the Count? Getting Reference Counting Back in the Ring. In *Proceedings of the 2012 International Symposium on Memory Management (ISMM '12)*. ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/2258996.2259008

[19] Rifat Shahriyar, Stephen M. Blackburn, and Kathryn S. McKinley. 2014. Fast Conservative Garbage Collection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 121–139. https://doi.org/10.1145/2660193.2660198

[20] Rifat Shahriyar, Stephen Michael Blackburn, Xi Yang, and Kathryn S. McKinley. 2013. Taking off the Gloves with Reference Counting Immix. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 93–110. https://doi.org/10.1145/2509136.2509527

[21] Paul R. Wilson. 1992. Uniprocessor Garbage Collection Techniques. In *Proceedings of the International Workshop on Memory Management (IWMM '92)*. Springer-Verlag, London, UK, UK, 1–42. http://dl.acm.org/citation.cfm?id=645648.664824