MIT Open Access Articles

Forkscan

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation	Alistarh, Dan et al. "Forkscan: Conservative Memory Reclamation for Modern Operating Systems." EuroSys '17: Proceedings of the Twelfth European Conference on Computer Systems, April 2017, Belgrade, Serbia, Association for Computing Machinery, April 2017 © 2017 The Authors
As Published	http://dx.doi.org/10.1145/3064176.3064214
Publisher	Association for Computing Machinery
Version	Author's final manuscript
Citable link	https://hdl.handle.net/1721.1/123336
Terms of Use	Creative Commons Attribution-Noncommercial-Share Alike
Detailed Terms	http://creativecommons.org/licenses/by-nc-sa/4.0/

Massachusetts Institute of Technology

DSpace@MIT

Forkscan: Conservative Memory Reclamation for Modern Operating Systems

Dan Alistarh
ETH Zurich
dan.alistarh@inf.ethz.ch

William Leiserson
MIT
willtor@mit.edu

Alexander Matveev
MIT
amatveev@mit.ed

Nir Shavit
MIT and Tel-Aviv University
shanir@csail.mit.edu

Abstract

The problem of efficient concurrent memory reclamation in unmanaged languages such as C or C++ is one of the major challenges facing the parallelization of billions of lines of legacy code. Garbage collectors for C/C++ can be inefficient; thus, programmers are often forced to use finely-crafted concurrent memory reclamation techniques. These techniques can provide good performance, but require considerable programming effort to deploy, and have strict requirements, allowing the programmer very little room for error.

In this work, we present Forkscan, a new conservative concurrent memory reclamation scheme which is fully automatic and surprisingly scalable. Forkscan's semantics place it between automatic garbage collectors (it requires the programmer to explicitly retire nodes before they can be reclaimed), and concurrent memory reclamation techniques (as it does not assume that nodes are completely unlinked from the data structure for correctness). Forkscan's implementation exploits these new semantics for efficiency: we leverage parallelism and optimized implementations of signaling and copy-on-write in modern operating systems to efficiently obtain and process consistent snapshots of memory that can be scanned concurrently with the normal program operation.

Empirical evaluation on a range of classical concurrent data structure microbenchmarks shows that Forkscan can preserve the scalability of the original code, while maintaining an order of magnitude lower latency than automatic garbage collection, and demonstrating competitive performance with finely crafted memory reclamation techniques.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroSys '17, April 23 - 26, 2017, Belgrade, Serbia

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4938-3/17/04... 15.00

DOI: http://dx.doi.org/10.1145/3064176.3064214

CCS Concepts • **Software and its engineering** → **General programming languages**; • **Theory of computation** → *Program analysis*

Keywords Synchronization, Memory Management

1. Introduction

A key requirement for modern multicore software is *scalability*: the capacity to perform better under higher thread counts. The last decade has seen a tremendous amount of progress in this area, such that, nowadays, scalable concurrent versions are known for many classic data structures, such as linked-lists [5, 24, 31, 35, 38], hash-tables [36, 39, 52], search trees [1, 30], and priority queues [3, 53].

While parallelizing data structures shows great promise for improving system performance, it also poses new challenges to existing programming methods and tools. We focus on one of the major obstacles to scalability: *concurrent memory reclamation*. This is the problem of deallocating memory while ensuring that no concurrent thread holds references to target memory blocks. This is especially pertinent since many high-performance concurrent data structures are implemented in unmanaged languages, such as C/C++, where developers are especially concerned with performance. To illustrate this problem, let us consider Harris and Michael's popular non-blocking linked list algorithm [31, 43].

The design is based on two key observations. The first is that, for scalability, list traversals should avoid using expensive synchronization, in the form of locks or compare-and-swap (CAS) operations. This *unsynchronized traversals* principle is common to many scalable data structure designs as listed above, and to parallelization techniques such as read-copy-update [42] and read-log-update [41]. The sec-

ond observation is that naively manipulating node pointers only via CAS operations is unsafe due to race conditions. Instead, the algorithm should "mark" nodes for deletion *before* they are actually made physically unreachable within the data structure. This is done by setting a mark bit in the *next* pointer of the node: an attempt to update this pointer after it is marked will fail. Figure 1 depicts how node deletion works in the Harris-Michael list.

From the perspective of reclaiming memory, this type of data structure implementation presents two key challenges. First, since logical and physical node deletion are decoupled, logically deleted nodes may still be reachable from other nodes in the data structure. Second, and more importantly, even unreachable nodes *may still be accessed* by concurrent threads holding references to such nodes. In the Harris-Michael list a stalled thread may still hold references to unreachable nodes as part of its traversal. Because traversals involve no synchronization, it is impossible to tell from the structure's state if a traversing thread holds a reference to a given node.

This is the problem of *invisible readers*: threads that traverse the data structure and read from nodes without alerting other threads to their presence may read old or junk data when another thread performs an update that frees a node. This can lead to wrong outputs or even segmentation faults. Guaranteeing that no thread is looking at a removed node, allowing it to be freed and reallocated, requires *invisible readers*, threads traversing the data structure without writing to it, to make their activities known in some way.

A programmer wishing to reclaim memory for an implementation of the concurrent list data structure in C/C++ has, broadly, two sets of options. The first, general one, is to employ a thread-safe automatic garbage collector, such as the popular implementation by Boehm et al. (BDW) [17]. Automatic memory reclamation tends to be very easy to apply, and takes the burden of discovering "who knows what and when" away from the data structure developer, shielding the end-programmer from this complexity. Garbage collectors usually have an `automalloc()` interface that is a drop-in replacement for a traditional allocator's `malloc()`. `automalloc()` returns memory that is tracked by the garbage collector, which will search for external references and reuse the memory if no such references can be found. However, such solutions can have a high cost in terms of raw performance and unpredictable high latency imposed on user threads, especially when the implementation is multi-threaded. (Please see Section 4 for experimental evidence.)

A second, fine-grained approach is to employ one of many existing *concurrent memory reclamation* schemes, e.g. [2, 4, 26, 31, 32, 34, 44], which we discuss in detail in the next section. These are methods specifically designed to solve the problem of concurrent access to reclamation candidates, and can reduce the performance overheads of garbage collection. The price for performance is the fact that most of these methods are *not automatic*: the programmer must manually apply them to the data structure implementation.

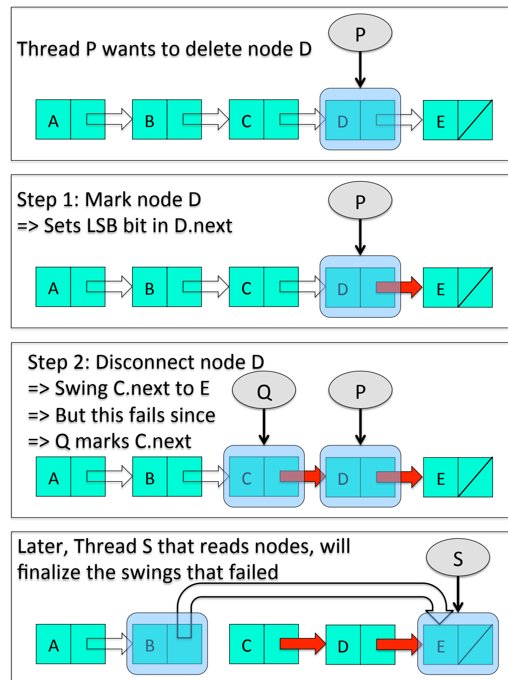

Figure 1. Node deletion in the Harris-Michael list: the thread (P) first performs a traversal; if the node (D) is found, the operation *marks the node* as described above, thus "logically removing" it. Then, the thread (P) performs a single attempt to physically remove the node from the data structure by swinging its incoming pointer (C.next to E). Importantly, this attempt may fail if the previous list node has changed since the traversal (update by thread Q in this case). However, the thread (P) will not try again: future traversals (thread S) encountering logically deleted nodes physically remove them from the list and reclaim their memory (swing B.next to E). Notice that there is no easy way to identify when a node is fully disconnected and can be reclaimed.

Frustratingly, concurrent memory reclamation schemes leave the programmer little room for error: these schemes critically rely on the assumption that nodes which are removal candidates *are unreachable in the data structure*. This requirement significantly complicates otherwise simple operations: for instance, the removal of multiple nodes from a singly-linked list must first disconnect each node in turn. Moreover, the resulting code base is fragile and great care must be taken when modifying it. To illustrate, consider the two-phase delete of Harris' list (Figure 1). Ensuring that the deleted node is fully unreachable after the delete completes requires handling of failures in the second phase that swings the "previous" node to the next node. However, this requires finding the proper "previous" node that needs to be modified, which may require re-traversal of the whole list. Ignoring the cost in performance, the cost in code complexity means that

such systems are full of edge cases, difficult to implement correctly, and difficult to understand.

We note that similar issues have been addressed in the context of *managed* programming languages such as Java or C#. In particular, fast, near-real-time concurrent garbage collection is known to be possible in such languages, e.g. [46], and efficient techniques have been proposed to speed up collection for concurrent data structures written in Java [14].

However, the programmer's current choice when writing C/C++ code is between solutions that are generic, but possibly inefficient or unpredictable in terms of latency, and finely-crafted reclamation techniques which can be efficient, but require a significant amount of programming effort, leaving no room for error.

In this paper, we show that this choice is not inherent. We present Forkscan, a new conservative reclamation scheme tailored for concurrent data structures, which is both *automatic* and *scalable*. Forkscan's semantics are simple: it implements a *retire* procedure, which takes a pointer to a memory node that the programmer wants to reclaim. Whereas garbage collectors treat `automalloc()` as a drop-in replacement for `malloc()`, and eliminate the `free()` call, Forkscan uses `forkscan_malloc()` and `forkscan_retire()` as drop-in replacements for `malloc()` and `free()`, respectively. The `forkscan_retire()` behaves, from the programmer's perspective, just as a `free()` call would. It is necessary to call it exactly once on a node in order to reclaim (and subsequently reuse) the memory. However, the call does not immediately reclaim the memory, but acts as a "hint" to Forkscan that the node is thought to be free of external references, and reuses it only once no such references exist. In contrast to previous schemes, Forkscan does *not* automatically assume that the node has been made unreachable in the data structure, and will preserve program correctness if this assumption does not hold, by postponing reclamation of the node. These semantics are useful since 1) they prevent catastrophic failure in case of a programming error and 2) they simplify programming in the case of compound node operations, such as removing a range of nodes from a linked list. At the same time, empirical results show that our implementation of Forkscan can preserve the performance of the original application to a large extent, both in terms of performance and latency.

Forkscan's semantics place it between automatic garbage collectors (since it requires the programmer to explicitly retire nodes), and concurrent memory reclamation techniques (since it does not require unlinking for correctness). Forkscan only performs reclamation when the number of outstanding retired nodes reaches a certain threshold. Additionally, since C/C++ programmers often tailor their memory allocators to their applications, Forkscan does not implement malloc itself but can be configured to use the preferred malloc implementation under the hood.

Our implementation is based on a simple principle: we aim to push the expensive components of reclamation onto the operating system, taking advantage of kernel optimizations. In particular, we use modern operating system support for efficient signaling and copy-on-write mechanisms to obtain an inexpensive coherent snapshot of memory for the collector, and design the memory scan to take advantage of prefetcher optimizations. Forkscan is both *concurrent*, in the sense that the application can run concurrently with the scanning process (except for a brief "freeze" period), and *parallel*, in that it leverages thread parallelism to speed up the scanning and the reclamation.

Some of the techniques behind Forkscan—such as exploiting copy-on-write to obtain inexpensive snapshots of memory—have been proposed previously for garbage collection [10, 48], but exhibited poor performance. We revisit these ideas in the context of modern operating systems, which provide efficient support for these constructs. To our knowledge, Forkscan is the first fully automatic memory reclamation technique for C/C++ with high scalability on many cores.

1.1 Algorithm Overview

At a high level, Forkscan has a simple division of labor. We structure the execution around *collection operations*. Each such operation consists of two phases: *freeze-and-fork* and *scan-and-mark*.

The freeze-and-fork phase works as follows. We reserve a reclaiming thread (the "reclaimer"), whose purpose is to wait for and receive a list of pointers to memory blocks that are candidates for deletion. Upon receiving such a list from a user thread, the reclaimer starts a reclamation operation by sending a signal to all other threads. Upon receipt of this signal, each thread writes out its current stack boundaries and register contents, replies with an acknowledgment, and waits. When the reclaimer has received acknowledgments from all threads, the memory is "frozen" of thread activity. At this time, the reclaimer thread forks off another process (the "scanner"), and then immediately signals all threads to resume their regular execution.

Next, the child process performs a parallel scan of memory. Note that the child inherits a proper memory snapshot at the "freezing" point, whose consistency is maintained by the system through the copy-on-write mechanism. The scanner partitions memory, and spawns siblings which iterate sequentially through each partition, attempting to interpret words as pointers to nodes in a list of reclamation candidates.[1] If a presumed pointer to a node is found, it is marked and recursively searched. This technique allows Forkscan to detect any cycles between retired nodes. At the end of the scan, the last forked child notifies its parent and terminates.

[1] Since it always pessimistically assumes that matched pointers are node references, Forkscan is *conservative*.

Our Forkscan implementation involves a few non-trivial observations and techniques:

First, it may seem that having a *retire* call detracts from the simplicity of the programming model and is not really necessary. However, our experience shows that the actual complexity in concurrent programming is not to identify what objects one wants to free (on what to call *retire*), but how to identify that there are no references to this object. This is because concurrent data-structures may remove references asynchronously or lazily, in order to improve performance or ensure progress guarantees, and identifying when all of the asynchronous updates render the node unreachable is not a trivial task for programmers. Forkscan automates this task.

Second, it is important to note that having a *retired node list* can make collection significantly more efficient. ForkScan's implementation exploits this fact to perform a *linear scan* of memory (as opposed to tracing) comparing each memory location against the list via a carefully optimized binary search procedure. This is a critical performance optimization, since most of the memory is "outside" the retired node list, and this memory is scanned linearly by the first phase, which is friendly for the CPU pre-fetching, caching, and page-fault mechanisms. As a result, the second phase, which performs an expensive recursive search and mark and is responsible for detecting cycles (in GC style), needs to scan much less memory – only memory that is part of the retired node list.

Third, node de-allocation is carefully piggybacked on top of allocation calls, to avoid the overheads of bulk deletes while bounding memory usage. A thread which wants to perform an allocation must first see if nodes are available to be freed. This allows the user threads to perform cleanup without introducing unpredictable wait times.

Finally, Forkscan induces blocking thread behaviour in theory; however, the handshake mechanism is implemented through signaling, and each thread must complete handler code before returning to user-level code. Thus, a thread is unresponsive only if starved for steps by the operating system, which only occurs in extreme conditions. Hence, we argue that in practice, Forkscan preserves the non-blocking nature of data structures.

Forkscan is allocator-agnostic. Whereas garbage collectors tend to own their allocators, Forkscan will sit on top of another allocator, such as [22, 27], and `malloc()` and `free()` from it. This provides some flexibility to C and C++ programmers who often choose allocators tailored to their specific workloads.

1.2 Evaluation Overview

We implemented Forkscan in C on Linux, and tested it on a machine with 80 hardware threads, to provide memory reclamation for a set of classic concurrent data structures: Harris and Michael's non-blocking linked list [31, 43], Fraser's skip list [25], and a concurrent hash table.

Empirical results on these microbenchmarks show that Forkscan scales well with the original data structure, with overhead typically falling somewhere between manual memory reclamation techniques and conventional garbage collectors. In trials, it is able to handle allocation frequencies of 18.8 million nodes/second, corresponding roughly to 4.4 GB of allocated memory per second. Further, Forkscan's overhead can be divided between snapshot latency and throttling, where throttling can be ameliorated by increasing memory allowances. Snapshot latency in Forkscan is low, even for large applications. At less than 60ms for programs upwards of 5GB running on 80 hardware threads, this is more than an order of magnitude lower than with automatic garbage collection. This feature addresses a core concern of C/C++ programmers regarding automated memory reclamation.

We also tested the limits of our approach, and identified the "breaking point" allocation frequencies after which Forkscan fails to keep up with the heap size.

Additionally, memcached [23] was modified, and its built-in reference counting mechanism was replaced with Forkscan, demonstrating its real-world applicability. Trials demonstrated no visible overhead versus the original code.

In sum, we show that it is possible to provide (to the extent testable on our current 80-way machine) fully scalable conservative memory reclamation for C/C++, by exploiting parallelism and tailoring it to take advantage of mechanisms that are highly optimized in modern operating systems. Moreover, memcached demonstrated that, without sacrificing performance, users need not deal with the complexities of reference counting or the like. Our implementation is in Linux [15], but we believe the ideas behind it can be applied to other state-of-the-art operating systems as well.

It is important to note that Forkscan has the following limitations. First, it is *conservative*, in that "false positive" reference detection may prevent it from reclaiming memory. This appears inherent due to the pointer semantics of C; moreover, a study by Boehm [7] showed that this space overhead is limited to an additive constant in most scenarios. Second, Forkscan is tailored towards modern concurrent data structure designs: it takes advantage of concurrency, and of the fact that in standard workloads, the majority of operations do not mutate the data structure. Different access patterns, for example, ones that are mutation-heavy, would not necessarily allow the same level of scalability. This is made evident in Section 4.

2. Related Work

The literature on memory reclamation is extremely vast, and a complete survey is beyond the scope of this paper. Instead, we focus on two topics: 1) data-structure aware garbage collection for managed languages, and 2) memory reclamation and conservative garbage collection for C/C++. (While interesting recent techniques [16, 47] are able to provide *precise* collection for C, they are single-threaded.)

Managed Languages. Recent work by Cohen and Petrank proposed a *data-structure aware* (DSA) garbage collector for Java [14], which takes advantage of structural information to improve both collection time (by up to 75%) and overall running time (by up to 30%) for real applications. More precisely, DSA requires that 1) Data structure nodes are identified via class annotations; 2) Remove is called explicitly on such nodes; 3) The memory allocator is customized to allow for optimized placement of such nodes, and 4) A full garbage collector is available for completeness.

By comparison, Forkscan also assumes that nodes be explicitly retired, but does not require the other three assumptions. In particular, we do not annotate data structure nodes, and Forkscan is allocator-agnostic. As Forkscan is developed for an unmanaged language, its technical details are fundamentally different from DSA. For instance, DSA uses node co-location to improve tracing, and locality. Since Forkscan does not control allocation, we developed other techniques such as the freeze-and-fork mechanism, linear scans, and piggy-backed allocation to gain efficiency.

C/C++. Arguably, the most well-known line of work on parallel conservative collection for C is the popular collector of Boehm, Demers, and Weiser [17]. Developed and improved over the space of two decades, e.g., [7, 8, 10], it has proved popular among practitioners.

The BDW collector is based on a classic mark-and-sweep pattern. In each collection phase, the collector thread signals all other participating threads to write out their register contents. It then marks all objects referenced directly by pointer variables (*roots*) and then iterates, each time marking newly reachable objects. Finally, it performs a *sweep*, identifying unmarked (unreachable) objects, adding them to free lists for use in satisfing allocation requests. Note that objects cannot be moved, due to the pointer semantics of C/C++.

In a seminal paper [10], Boehm, Demers, and Schenker gave a way to run the collector thread “mostly concurrently” with application threads: the collector first stops all threads while it protects memory pages for writing; then threads may resume. To maintain consistent marking, threads which modify memory during collection incur a page write protection fault, and will re-trace the nodes whose pointers they modify. They also show that the sweep phase is parallelizable; later work [21] showed that the mark phase is also parallelizable. Subsequently, Kermany and Petrank [37], and Barabash, Ossia, and Petrank [37] proposed several optimizations to reduce pause times and memory fragmentation. In general, this approach can suffer from high overheads due to the page protection mechanism [20]. The garbage collection work probably closest to ours is that of Shahriyar et al. [51], which performs conservative reference counting and cycle detection, with write barriers on pointer mutations.

Forkscan, though automated, does not do garbage collection, and differs from the above approaches in significant ways. Instead of tracing objects, we choose to perform a full, sequential sweep of the heap; thus, it will likely scan more memory, but will gain much better locality during the scan. Instead of explicitly enforcing consistent marking via page protections, it takes advantage of the semantics of fork to ensure that the scanner sees a consistent snapshot of memory. Finally, Forkscan takes hints from the concurrent data structure developer about when a node is likely to be available for freeing (when it is removed).

The idea of using the fork semantics to obtain a snapshot of memory for the scanner is not new. It is briefly mentioned by Boehm, Demers, and Shenker [10], and implemented by Rodriguez-Rivera and Russo [48], where it shows roughly the same performance as the BDW collector on a four-processor SPARC machine. We revisit this idea here, and show that, if used carefully, it can lead to significant performance improvements on modern operating systems.

A second line of research is *concurrent memory reclamation* methods [34]. These techniques fundamentally rely on the assumption that nodes which are candidates for reclamation are already unreachable in the data structure. Based on their characteristics, they can be roughly split into four categories.

The first is that of *reference-counting* schemes [18, 28], which associate a reference count with each node. Once this counter is cleared, the node is safe for reclamation, as no new references to it may be created. These methods can in theory be automated, and are the closest to classic garbage collection. Unfortunately, they suffer from considerable performance degradation, as they induce expensive synchronization on every new node access. Where multiple threads can read the same data cheaply, reference counting adds writes that may contend. Put another way, reference counting renders invisible readers visible by adding a potentially-contending write to every read.

Second, *quiescence-based* techniques [26, 31, 32] delay reclamation until *quiescent periods* when threads pass through states where no shared references are held, such as when data structure operations return. These schemes are lightweight, but rely on per-thread progress, as thread stalls or crashes can significantly delay reclamation. Recent work [6, 11, 12] showed that such schemes can recover from thread crashes, in specific instances. Third, pointer-based schemes [34, 44] require the programmer to explicitly mark live nodes (which may be accessed) to prevent de-allocation. Such schemes are data-structure specific, and add an expensive *validation* step, which exposes marked nodes to other threads by means of a memory fence.

Additionally, recent work by Cohen and Petrank [13] presented an automatic reclamation scheme for lock-free data structures, inspired by mark-and-sweep garbage collection, which assumes that data structures are written in a specific normalized form. While many data structures can be rewritten according to this pattern, it is not known whether this process can be made automatic.

Other recent solutions rely on either hardware or operating system support. In particular, Dragojevic et al. [19] and Alistarh et al. [2] proposed schemes based on hardware transactional memory. (The latter scheme is also semi-automatic, in that it only requires the programmer to specify a fallback path.)

In the recently proposed ThreadScan mechanism [4], the reclaiming thread signals all other threads to scan their own stacks and registers for references to nodes in a delete buffer, as part of the signal handler. Candidate nodes without such references are clear for removal. This mechanism is similar to our first snapshot stage, and to previous garbage collector mechanisms, e.g. [48]. Because ThreadScan scans only stacks and an optional range of memory, it restricts where pointers can be stored. Moreover, its scanning phase must be completed before any thread returns to work.

3. The Forkscan Algorithm

3.1 Overview

Forkscan uses the following pattern: Each thread maintains a pool of nodes to be reclaimed. The pool is populated by concurrent data structures, which call `forkscan_retire()` on nodes as they are removed. In the pattern of popular concurrent data structures, these nodes are unlikely to be seen by other threads. When a thread's pool becomes full, it consolidates all thread pools and hands it to a specialized *reclaimer* thread from the Forkscan runtime. We call the consolidated set of pools the *delete buffer*.

The reclaimer then starts a *reclamation phase*, attempting to purge the delete buffer. A reclamation phase consists of several steps. We describe these steps in detail below, and present pseudo-code in Algorithm 1.

Step 1: Freeze-and-fork. The first step aims to obtain a coherent snapshot of the application's memory. For this, the reclaimer first broadcasts a signal to all other threads, which handle the signal by writing out their current stack boundaries and register contents. Subsequently, each thread sends an acknowledgment, and waits for confirmation. Once the reclaimer has collected acknowledgments from all application threads, it *forks* a new process, whose task will be to scan memory. As soon as the fork returns, the reclaimer releases all other threads to return to their regular execution, and waits for the child to complete the scan.

Two observations are important at this point. The first is that, at the time T when the reclaimer calls `fork()`, all threads have written their current stack boundaries and register contents to memory, without executing further. Second, by the semantics of `fork()`, the child process will observe a *consistent snapshot* of the program's memory at time T, including heap, stack, and register contents. Therefore, to determine whether outstanding references to delete candidates still exist, it is sufficient for this child process to simply scan the heap, stack, and register contents as it observes them.

Step 2: Scanning. The child process begins by identifying the memory ranges which need to be scanned, and partitions them into M disjoint ranges, where $M \geq 1$ is a parameter. It then forks $M - 1$ sibling processes, such that each has a subrange that can be scanned in parallel. Scanning is broken into two parts: 1. *find roots* and 2. *recursive mark*. Finding roots means searching through memory for references outside of the delete buffer to nodes inside of it. The processes scan memory, avoiding the nodes in the buffer, for references, and marking the pointers in the buffer when they are found. The delete buffer is in shared memory between the sibling processes, so writes are visible to all siblings.

After the roots have been found, the delete buffer is broken up into chunks for the sibling processes, and marked references are recursively searched for further references. At the end of this phase, all nodes that are visible from the user program have been discovered and marked. Any unmarked nodes are no longer known to the user, and are available to be freed. This marking technique allows cycles to be discovered, so that nodes that point to one another, but are not pointed to from outside can be freed.

When the scan is complete, the last of the children notifies the reclaimer thread in the parent (via a pipe) and winds down. The reclaimer thread runs down the buffer looking for reachable nodes and preserves them for the next round of reclamation.

Step 3: Deletion. The previous step identified a set of nodes which can be safely deleted. It is tempting to free all these blocks at this time. However, in practice this leads to poor performance. If freeing is done by the reclaimer, reclamation iterations are delayed. If it is done by other internal threads, those threads compete for time and memory resources with user threads. And if user threads are signaled again to perform the task, Forkscan introduces high latency (which is what C/C++ programmers want to avoid).

We therefore delay the free calls by piggybacking them on future malloc calls, allowing the user threads to perform the deletion phase without introducing high latency. This amortizes the free calls via malloc calls, while at the same time roughly matching the frequency of allocation with that of de-allocation.

The delete buffer is preserved, and a user thread that wants to allocate memory will first free some of the nodes in it. Each thread, when it has no nodes to free, will reserve a portion of the buffer that it is responsible for freeing. With each allocation, the thread will traverse part of its range, identify a small number of unmarked references, and free them. The delete buffer is reference counted, so when it has no more ranges to reserve, and when its reference count hits zero, it can be reused.

3.2 Implementation Details

Allocation and Retirement. `forkscan_malloc()` is provided as a wrapper for `malloc()` and has the same profile. If

Algorithm 1 ForkGC Pseudocode.

```
 1: function CONSOLIDATE_PTRS
        ▷ Called when a user thread pool is full.
        ▷ Aggregate pointers from all threads
 2:     delete-buffer ← ∅
 3:     for th ∈ threads do
 4:         delete-buffer ∪ = GET_PTRS_POOL(th)
 5:     SORT(delete-buffer)
 6:     SIGNAL-CONDITION-VAR(reclaimer-conditional)

 7: function RECLAIMER_THREAD(ctx)
 8:     while 1 do
 9:         WAIT-ON-CONDITION-VAR(reclaimer-conditional)
        ▷ Signal all threads
10:         for th ∈ threads do
11:             SIGNAL(th, snapshot)
12:         wait for ACK from all threads
        ▷ At this point, the system is "frozen," so we fork
13:         pid ← FORK()
14:         if pid = 0 then
        ▷ The child scans the memory snapshot
15:             SCAN(ctx)
16:             EXIT()
        ▷ This is the parent
17:         resume all threads via signal          ▷ Child got snapshot
18:         wait for child to finish
19:         PUSH-BACK(delete-buffer)        ▷ Free memory

20: function SCAN(ctx)
21:     memory-ranges ← GET_MAPPED_RANGES()
22:     Split memory-ranges into M partitions
23:     for id ∈ [1..M - 1] do
24:         pid ← FORK()
25:         if pid = 0 then
26:             SCAN_FOR_REFS(memory-ranges[id])
27:             EXIT()
28:     SCAN_FOR_REFS(memory-ranges[0])
29:     Wait for children to finish

30: function SCAN_FOR_REFS(memory-ranges)
31:     for each word ∈ memory-ranges do
        ▷ Check if word is a reference to some object
32:         i ← BINARY-SEARCH(word, delete-buffer)
33:         if i ≠ 0 then
        ▷ Found a reference → record it
34:             SET-LOW-BIT(delete-buffer[i])
35:             SCAN_FOR_REFS(delete-buffer[i])

36: function SNAPSHOT_SIGNAL_HANDLER(ctx)
        ▷ Executed by thread on snapshot signal
37:     Spill registers and stack boundaries to stack
38:     Send ACK to reclaimer-thread
39:     Wait for resume signal   ▷ Freeze point for snapshot
```

nodes are available to be freed, it will do so before returning new memory to the user. The main interface to Forkscan is `forkscan_retire()`, which behaves like `free()` from the user's perspective except that instead of immediately freeing the node, it adds it to the thread's pool, checks to see if the pool is full, and possibly becomes the consolidator. It should be noted that `forkscan_retire()` is a proper-replacement for `free()` in that retiring the same node multiple times or from multiple threads could have the same unpredictable effects as a double-free. Likewise, both retiring and freeing a node will lead to unpredictable behavior.

Forkscan is allocator-agnostic and treats the underlying library as a black box. The je_malloc allocator [22] was selected for experimentation because it had the best performance in trials on the microbenchmarks. However, Forkscan only requires the names of `malloc()`, `free()`, and `malloc_usable_size()` (the last is a function that, given a pointer, returns the number of bytes available in that block). It can be configured at run-time to use any allocator the application developer prefers, as long as it implements those three functions.

Thread List Consolidation. The per-thread lists of addresses are implemented as deques: values are pushed on one end by the owner as allocations occur, and are popped from the other by the consolidating thread. The consolidating thread grabs a global lock before it begins, so deques are only popped by one thread at any time. This makes them single-reader, single-writer data structures, even though the popping thread may be different each time consolidation happens. The implementation is taken from ThreadScan [4].

Once the thread buffers have been drained by the consolidator, the thread pushes the consolidated buffer onto a list of waiting buffers for the reclamation thread to find. The final delete buffer is created by the reclamation thread as an aggregate of waiting consolidation buffers and leftover addresses from previous reclamation iterations.

This architecture is highly concurrent, in spite of the global lock, when thread buffers are big enough to make consolidation rare. Therefore, contention is low. The thread buffers have a configurable size, but they default to 65,536 (64K) addresses, a number that was picked based on hand-tuning.

Capping Memory. Without an unreferenced memory limit, a process could grow unbounded for data structures with frequent writes if write operations, that might cause fast memory turnover, are exceedingly fast. Forkscan caps the amount of unreferenced memory by limiting the number of unreferenced pointers. Thus, the cap is proportional to the average size of allocations. The limit of unreferenced pointers is enforced by the consolidation system: When a consolidated buffer is pushed onto the waiting list, the consolidator increments a counter. If the counter exceeds the waiting limit, the consolidating thread stalls until the counter is reset by the collection thread when it starts a new iteration. Memory is bounded because no other thread can become the consolidator until the current one relinquishes its role, thereby throttling user threads when memory threatens to grow beyond the (configurable) predetermined limits.

Naturally, a lack of memory can lead to massive throttling, thereby introducing latency. But the user can configure the size of the thread buffers and maximum number of waiting consolidation buffers, trading off memory for latency. This is tested in Section 4.

Thread Handling. In order for Forkscan to function, it must know about all threads that may access the data structures that use `forkscan_retire()`. To obtain this information, Forkscan wraps `main()` and as well as all user calls to `pthread_create()`, storing metadata about the thread ID and stack bounds. At present, it does not allow threads to opt out. This behavior is similar to that of the classic BDW collector [17]. The Forkscan algorithm does not inherently disallow threads from opting out. But such threads would have to be restricted from operating on concurrent data structures or moving pointers to concurrent nodes around in memory, thereby "hiding" them.

Signaling. Signaling is based on `pthread_kill()`, an API originally intended for killing threads, which targets a thread by ID. When the process starts, it registers a signal handler that catches the signal on the targeted thread. A thread that receives a signal will be interrupted unless it is in the midst of a system call, in which case it responds before it returns to user code. With the ID of all of the threads in the process, the collector is able to iterate through the list and force every thread to pause its execution and respond through the signal handler.

The signaling mechanism additionally forces the thread to dump its registers to the stack for the purpose of saving the context. The operating system will use this context to resume the thread after the signal has been handled. However, having preserved its register contents, the forked process can see the register contents each thread had at the time it paused.

Forking. When the collector thread forks, the children need to communicate a potentially large amount of data with the parent about the reference counts they calculate. To make this communication efficient, the delete buffer is allocated on shared pages. Changes to the reference counts in the delete buffer made by child processes are visible to the parent without any explicit communication. Since the reclamation thread in the parent has no work to do, it waits on a pipe for notification that the scan has completed. The children can exit as they finish scanning their regions of memory, atomically incrementing a shared `scanner_completed` counter as they leave. The last child sends a message to the parent through the pipe, waking it up and allowing it to proceed.

Finding Memory to Scan. The first `fork()` generates a scan child that calculates how much memory needs to be scanned by reading from the `/proc/self/maps` file. It keeps track of memory ranges that might contain references to concurrent nodes, excluding regions allocated by Forkscan, itself. The latter exclusions are easy to detect because Forkscan does all of its own internal memory management.

Expedited Scanning. Comparing a range of memory addresses, m, to an arbitrary list of delete buffer addresses, d, is a $O(m \times d)$ problem. Forkscan sorts its delete buffer to make the scan a $O(m \times \log d)$ problem. However, accessing the delete buffer is still slow because it can potentially fill thousands of pages, leading to frequent cache misses.

Performance is improved by creating a minimap of addresses: a subset of addresses from the bigger pool. The minimap is created by striding across the overall delete buffer, a page at a time, and collecting the first address stored on each page. Therefore, each entry in the minimap corresponds to the first entry of each page in the delete buffer. When searching for an address, p, the minimap can be queried for the closest address without going over, q. Since the delete buffer is sorted, the location of q in the minimap identifies the exact page on which p exists, if it is present in the delete buffer.

For 4096-byte pages and 8-byte addresses, this means the minimap represents a 512-fold reduction in space, or the equivalent of 9 steps in a binary search. In general, it also means that the whole minimap fits in cache. The consequence is that a search for a particular address typically misses the L3 cache exactly once. In practice, this optimization reduced Forkscan's overhead to negligible levels in many tests.

Cache-friendly Scanning. In the root finding phase of the scan, potential references are collected and not searched in the delete buffer until a threshold has been reached. Once enough potential references are found, they are sorted, and searched sequentially. A pointer to the last searched location in the delete buffer is retained since the next potential reference is likely to be very close. The next reference can be checked against the rest of that cache line in the delete buffer. This makes binary searches rare during root finding, and keeps accesses mostly sequential.

Scan Parallelization. The amount of memory to be scanned determines the number of siblings the first child will `fork()` to help. In practice, every extra 128 MB warrants another scanner, up to a system maximum of 16. This number was selected based on trials run on three different machines (with very different architectures) that all gave best performance at this number. The subsequent `fork()` calls are cheap, and the processes are lightweight, because they modify almost no memory except for what is in the shared buffer. The memory they scan is treated as read-only, so copy-on-write is never invoked.

Scan children communicate with one another about what addresses they have seen using the shared delete buffer. All manipulation of the reference counts are done with an atomic increment, which is a Read-Modify-Write (RMW) operation. The struct at the head of the delete buffer is shared, itself, and contains the `scanner_completed` counter.

De-allocation. Our experiments show that deallocating memory via a long sequence of `free()` calls is expensive due to the system calls to `madvise()` (controls page release/purge to the Linux OS). Therefore, to avoid contention and latency, nodes which are marked for deletion are not freed immediately. Instead, the protocol pushes the delete buffer onto the back of a list of delete buffers from previous iterations. Threads that want to allocate memory and don't have anything to free query this list and reserve a subrange from the front delete buffer.

Potentially, a thread that only calls `forkscan_malloc()` once could retain a reference to a delete buffer and keep it from being reused, but no thread could monopolize more than one delete buffer. And practically, a thread that mutates a concurrent object once is likely to do so again.

False positives. Nodes from previous iterations that have no outstanding references, but are still waiting to be freed, may contain references to nodes in the current reclamation iteration. In practice, this is a significant source of false positives that leads to loss of scalability. The reclaimer, therefore, creates a list of *dead nodes* from the previous delete buffer to give to the forked children for reference. A child, while it is scanning memory, uses this list and skips scanning anything from a dead node. List creation is performed after all threads have acknowledged the signal, but before the `fork()`, making the protocol slightly more costly.

3.3 Correctness Properties

Forkscan makes the following set of assumptions.

1. (*No False Negatives*) References to memory blocks in the scanned space are word-aligned, and can be matched to the interior of allocated blocks by comparison. Additionally, Forkscan masks off the low 3 bits of any word it reads when scanning. This covers the common form of "pointer-hiding" used by many data structures, and it means those that overload those bits are discovered.
2. (*No Thread Crashes*) Threads do not crash.
3. (*Bounded Allocation Rate*) There exists a finite bound on the allocation rate of the application.
4. (*No False Positives*) Arbitrary memory words do not match block addresses.

Under these assumptions, Forkscan provides the following guarantee:

THEOREM 1. *Forkscan ensures the following.*

1. *Reachable memory blocks cannot be de-allocated.*
2. *Every unreachable memory block is eventually de-allocated.*
3. *There exists a finite bound on the amount of memory employed by the application at any point in time.*

Proof (Sketch). For the proof of the first two statements, consider an arbitrary collection phase, and let T be the time when the `fork()` call completes. A key invariant is that allocated memory nodes which are not referenced (either in thread stacks and registers or in the heap) at time T cannot be referenced at later times in the execution (unless first recycled), as they are currently *unreachable*. Further, we rely on the fact that the child thread resulting from the `fork()` operation receives a consistent *snapshot* of the entire parent process memory at time T. By assumption (1), every reachable node will have a non-zero reference count at the end of the scan. By assumption (2), no unreachable node can have non-zero reference count at the end of the scan. These two properties will imply that no reachable memory blocks can be allocated. Since Forkscan checks for cycles, every unreachable block is eventually de-allocated. The third property follows since we assume no false positives, and that there exists an upper bound on the allocation frequency.

4. Experimental Results

4.1 Microbenchmarks

Setup. Forkscan was tested on an 40-core (4 sockets, 80 threads) Intel Xeon computer at 2.4 GHz with 1TB of RAM running Ubuntu 15.04 with the 3.13.0-57 kernel. Software threads were scheduled by the operating system, though the Linux kernel tended not to migrate threads very often, but instead scheduled threads on the same cores, generally. The data structures that were tested were a lock-free linked list [31, 43], a lock-free hash table from Synchrobench [29], and a lock-based skip list from StackTrack [2].

For comparison, we used versions of these structures that leak memory ("Leaky"), the latest Boehm-Demers-Weiser Garbage Collector 7.4.2 ("BDW-GC") [9], and simulated a Hazard Pointers [44] implementation by burdening reads during list traversals. StackTrack had an actual Hazard Pointer implementation. We compiled BDW-GC with parallel mark and thread local optimizations. The Leaky, Forkscan, and Hazard Pointer tests ran with the JEMalloc 3.6 [22] allocator. BDW-GC uses its own internal allocator.

The linked list was initialized with 1024 nodes and executed with 2048 possible values. Nodes were padded to 176 bytes in order to avoid false sharing and prefetching. This was beneficial for all systems. The skip list was given 12,800,000 nodes and 25,600,000 possible values. Unlike nodes in serial skip lists, which are only as large as they need to be, ours were made 256 bytes with a maximum height of 20. Again, eliminating short nodes was necessary to eliminate false sharing. It is worth noting that at that height, guaranteed $O(lgn)$ access complexity only allows for 1,048,576 nodes, so accesses were slightly more expensive. The total size of the skip list was about 3.1GB. Last, to test a high performance structure, the hash table was given 32,000,000 initial nodes with a range of 64,000,000 possible values. Buckets were implemented using the lock-free linked list, with 32 average list length. The hash table's size was about 5.4GB.

Forkscan was configured for conservative memory usage: each thread had a pool capacity of 16K nodes, and no more than 4 aggregate lists could queue up before Forkscan began throttling threads trying to allocate memory. No tests were run with larger per-thread pools because performance was good with the smaller ones.

Figure 2 shows benchmark results on the data structures. Results are averaged over 3 executions of 4 minutes each. The benchmarks were set to perform 20% modify operations (reads and writes), a very heavy workload, to show performance under pressure.

In the linked list case, BDW-GC performed along a similar curve to "Leaky" with visible overheads, and the Hazard Pointer simulation showed the cost of adding writes to every read. However, this structure has slow enough operations that Forkscan's performance overheads were almost unnoticeable. This is also visible in the memory usage graph, which has no elbow in the curve. Memory usage is expected to grow linearly with the number of threads, since each thread has its own pool, and a consolidation buffer size is proportional to the pool size and the number of threads. A linear growth shows that Forkscan did not need to allocate extra consolidation buffers or delete buffers. BDW-GC, on the other hand, has a roughly fixed overhead and is able to track all of its pointers because it owns its allocator. Snapshot latency was especially low for Forkscan, topping out at 12ms on 80 threads, because the whole application used very little memory. BDW-GC's latency was roughly 242x above Forkscan because collection happens inline with the running benchmark. The extremely high latency, in this case, was likely due to the length of the chain.

The skiplist has cheaper operations overall, even though it is over-filled. Additionally, although StackTrack's skiplist takes out locks during add and remove operations, there is not very much contention and the skiplist beats the linked list based on its access time. In this case, Hazard Pointers and Forkscan both outperformed the Leaky implementation. Leaky performed poorly because, on the 40 and 80 core executions, the large size caused it to have poor cache performance. BDW-GC took a hit in about the same place but in this case, it was probably due to excessive scan times on the large data structure. Even though Forkscan performed better than Leaky, it was burdened due to throttling. As we demonstrate below, the throttling latency can be overcome.

Memory usage is again higher for Forkscan than BDW-GC, since Forkscan uses extra memory proportional to the thread count. The elbow in memory usage happens early, at 10 threads, as the cheaper operations caused Forkscan to queue consolidation buffers. It never quite found equilibrium before the queue filled, and throttled user threads. At 80 threads, it reached 60ms, as high as any of Forkscan's trials. However, when compared against the average 6.3 second scan time of BDW-GC, Forkscan's latency was still very low.

Hash tables have cheaper operations than skiplists. In general, accesses are expected to be constant-time operations, so the Hazard Pointer simulation performs very well. From the outset, Forkscan has a difficult time keeping up with it, but continues to scale linearly (albeit, linearly with a small constant multiplier). Again, the overhead can be attributed entirely to throttling of user threads as they perform allocations. The time it takes to perform a snapshot, even on this large data set, however, remains low: reaching a maximum of 54ms on 80 threads. The BDW-GC collector was unstable on this workload, even when provided with lots of extra memory, so it could not be tested.

An additional stress test was run on the hash table to demonstrate the breaking point of Forkscan. Instead of 20% updates, a high value for real world applications, 40% updates was specified. The results are shown in fig. 3. As above, read and write operations are all about the same to Hazard Pointers. However, Forkscan does not gain appreciably from doubling the number of threads from 40 to 80. At this point, throttling is significant and almost all of the overhead is attributable to that, as the snapshot time has not increased appreciably over the 20% update trials.

High latency due to throttling may not be any more palatable to C/C++ users than if the memory reclamation system simply stopped the world and did all of its work using the user threads. However, further tests demonstrate that, unlike stopping the world and recruiting the user's threads for memory scanning, latency due to throttling can be reduced where additional memory is available. In the cases above, most overhead was throttling and very little was due to stopping the world to take a snapshot.

Figure 4 shows how memory is used by Forkscan and how it impacts performance. The first graph shows the total memory footprint of Forkscan run on the hashtable with 20% updates over a 7 minute execution. Forkscan can queue up to 4 consolidation buffers (configurable) at a time before throttling, and delete buffers are only reused after all freeable nodes are actually freed. These buffers are proportional to the number of threads and individual thread pool size, and

Lock-free Linked List Performance

Operations

Threads

Leaky
Hazard
BDW-GC
Forkscan

Lock-free Linked List Memory Usage

Memory (4KB pages)

Threads

BDW-GC
Forkscan

Lock-free Linked List Latency

Time (ms)

Threads

BDW-GC
Forkscan snapshot

Skip List Performance

Operations

Threads

Leaky
Hazard
BDW-GC
Forkscan

Skip List Memory Usage

Memory (4KB pages)

Threads

BDW-GC
Forkscan

Skip List Latency

Time (ms)

Threads

BDW-GC
Forkscan snapshot

Hash Table Performance

Operations

Threads

Leaky
Hazard
Forkscan

Hash Table Memory Usage

Memory (4KB pages)

Threads

Forkscan

Hash Table Latency

Time (ms)

Threads

Forkscan snapshot

Figure 2. Performance results on the linked list, skiplist, and hash table data structures. For each: total operations, memory usage of the application, and average latency per reclamation iteration (logscale for the first two structures, to compare Forkscan with BDW-GC).

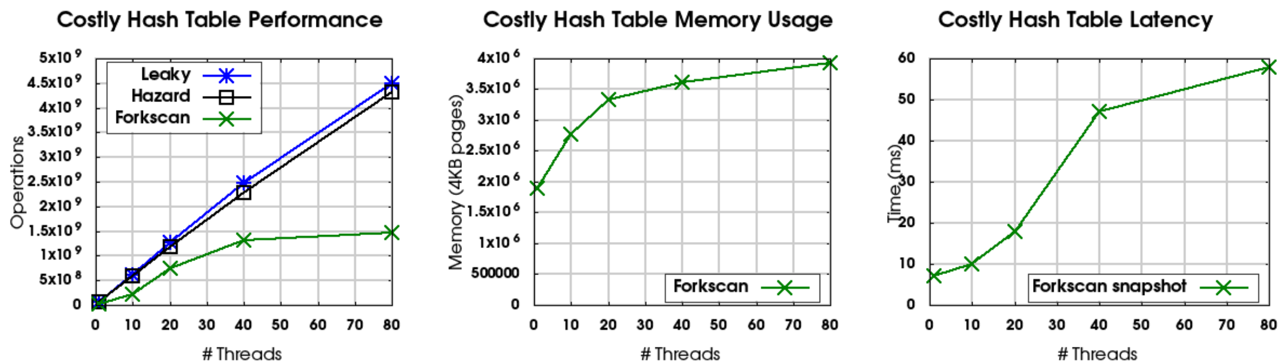

Figure 3. Performance results for a hash table with 40% update operations.

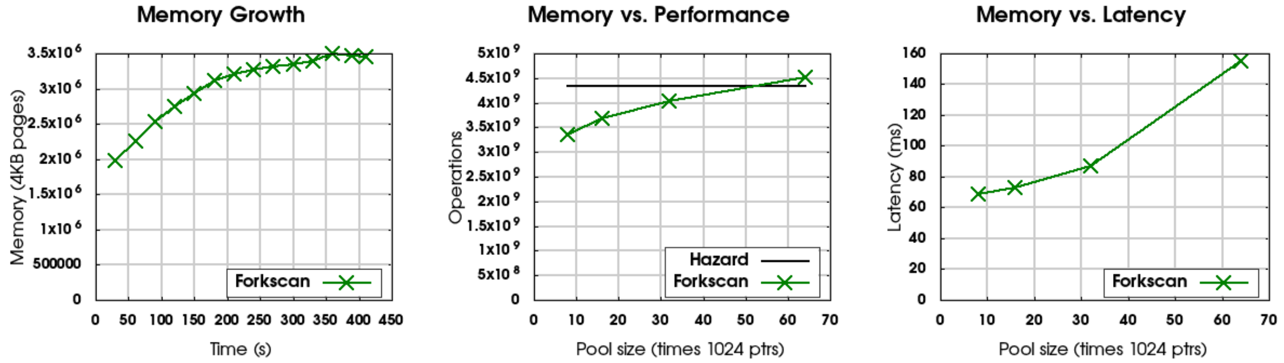

Figure 4. Forkscan memory usage and memory/latency vs. performance tradeoffs.

the total overhead corresponds to the total number of pointers times the average size of nodes.

The second graph is based on the skiplist run with 80 threads and shows that performance can be bought with more memory, as is typical with other automated reclamation systems. The Hazard Pointers result on 80 threads from fig. 2 is shown for comparison. This tradeoff is the mechanism by which the user amortizes the cost of reclamation over the normal cost of performing operations in the application. The third graph, however, shows the snapshot latency over those same executions. Whereas, in garbage collectors, increased memory to improve performance increases individual thread latency, Forkscan imposes no significant increase in this metric.

This demonstrates that Forkscan is able to provide comparable performance to manual memory reclamation schemes at a fraction of the latency of traditional automatic reclamation systems. A larger process takes longer to fork than a small one, but the vast bulk of the cost of doing memory reclamation is invisible to user threads, even when Forkscan is configured to use large amounts of memory. The maximum 155ms (at 64K pointers per thread pool) is human noticeable, but it is a short duration compared with conventional automatic systems.

The last point regarding latency of concern to C and C++ programmers is the overhead on the burdened allocation. Since `forkscan_malloc()` attempts to free nodes from previous iterations, the actual allocation is more costly than in a serial execution. The overall amount of work is no more than in a serial application since one `free()` corresponds to one `malloc()` in the underlying allocator in both cases. But a call to `forkscan_malloc()` attempts to free multiple nodes per allocation in order to keep memory low.

The Forkscan library was instrumented to capture the amount of time spent freeing nodes, and the original hash table trial was rerun with 80 threads. Figure 5 shows a histogram of the amount of time (in tens of nanoseconds) spent on each allocation. The vast majority of allocations were burdened by no more than 100ns, though there was an extended tail due to differences in operating system scheduling. Calls with overhead of more than half a microsecond were all collected into the last bucket, causing the apparent bump.

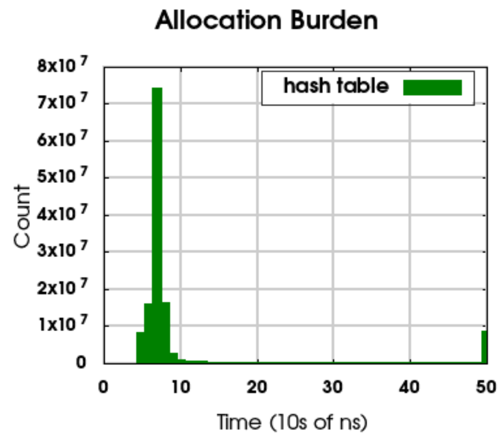

Figure 5. Histogram of overhead per allocation.

The lack of variance during freeing is expected since the nodes have been sorted, and for adjacent pointers in the list, those pointers are likely to have good spacial locality. Therefore, many calls to `free()` should not be much more costly than a single one. Since this experiment is dependent on the implementation of the underlying allocator (JEMalloc, in this case), the shape of the graph is more interesting than the specific numbers.

4.2 Real-world application

Finally, to demonstrate Forkscan's effectiveness in a real-world application, memcached [23], was modified to create Leaky and Forkscan versions, replacing its default reference-counting. In the altered versions, the builtin slab allocator was removed and replaced with je_malloc for simplicity.

In the Forkscan version, all accesses to individual item reference counters were eliminated, and when an item was unlinked from the structure, the thread that succeeded in un-

Threads	Default	Leaky	Forkscan
1	156532	160715	173726
2	233993	276594	227535
4	306870	280548	314001
10	523586	534209	510895
20	245087	277168	259428
40	193803	198706	200100

Table 1. memcached performance in operations/second.

linking it then retired it. The Leaky version differed only in that the retire call was commented out. Since the memcached implementation was not changed, apart from how memory was managed, Leaky was intended to act as an upper-bound for performance.

To test performance, it was necessary to create a large enough database that many connections would be supported without making contention on individual items a bottleneck. Such a bottleneck would have masked the best-case (for memcached) scenario limiting factor. However, this had to be balanced against the ability to fill the database quickly and force replacements to happen frequently. Therefore, memcached servers were created with 1GB of memory, storing items of 1024 bytes, allowing roughly 1 million individual items.

The memcached servers were configured to run locally, avoiding network overhead and latency. Trials were run using memtier_benchmark [40] with 16 threads for 12 seconds with a set/get ratio of 1:4, and using 40 multi-key gets to inflate the number of requests through a limited number of connections. Each trial, for each version, was run 3 times and the average number of operations/second was computed. Trials above 40 threads were not run because of performance degradation for all versions.

Table 1 shows the results. Performance was comparable in all cases. The high variance in execution times between trials indicates that other factors are more important to performance than memory reclamation (or lack thereof), as sometimes Leaky was outperformed by Forkscan or the Default reference-counted implementations. A drop-off in operations per second occurred after 10 threads making locks a likely culprit. The amount of time it took to freeze and fork was typically around 5ms or less, and never exceeded 9ms.

These results indicate that Forkscan works in a practical setting, making the code simpler without impeding performance. Moreover, the individual data structure benchmarks, especially the hash table, test Forkscan far more strenuously than memcached. The simplified application code, which no longer needs to count references nor carefully needs to verify correctness, makes Forkscan a valuable alternative for reclaiming memory from concurrent data structures.

5. Discussion

We have presented Forkscan, a new memory reclamation system which shows that it is possible to provide fully scalable conservative memory reclamation for C/C++ by exploiting parallelism and tailoring it to take advantage of mechanisms that are highly optimized in modern operating systems. Our implementation focuses on the Linux operating system [15], but we believe the ideas behind it can be applied to other state-of-the-art operating systems as well.

Performance of Forkscan is competitive with other automatic reclamation systems such as the popular BDW garbage collector. On the other hand, although manual application of certain memory reclamation systems can eliminate unpredictable delays, they are often difficult to apply correctly and impose themselves on the end-programmers who use the data structure. Forkscan takes a meaningful step in the direction of reduced latency imposed on user threads, while maintaining an automated interface. Notably, even in applications with large data sets Forkscan's snapshot causes only brief interruptions.

Our experimental setup was developed for Linux, which offers an efficient copy-on-write mechanism through fork. In theory, a similar mechanism can be implemented in Windows via Virtual Memory Functions [45]. An implementation such a copy-on-write mechanism is described and benchmarked in [49], to provide concurrent garbage collection for the D programming language. Although in theory this implies that Forkscan could work on Windows, its implementation would probably be quite complex.

Forkscan's interface, requiring a *retire* call, is a feature designed to improve performance (both in time and memory) and give the programmer control over how memory is handled. Memory that is known to be visible to a single thread can be free'd directly. That memory need never be tracked by Forkscan, saving time and resources. A programmer can simulate a GC-style interface by retiring a pointer as soon as it's allocated – Forkscan even provides an `automalloc()` function as an alternate interface – but we expect that C/C++ programmers prefer the control of the default interface.

5.1 Limitations

Conservative Reclamation. Forkscan is conservative, in that memory words which could be pointers to a memory block are automatically treated as references. We share this limitation with other automatic reclamation systems for C/C++ [4, 17, 50]. In theory, this assumption could prevent memory from being de-allocated, e.g. in the case of a list whose head node has a false reference. A study by Boehm [7] considers this issue in detail, and concludes that, in most practical scenarios, the space overhead of conservatism is bounded by an additive constant. This analysis generalizes to Forkscan.

We also assume that references are word-aligned, and that the programmer does not obfuscate references to memory

blocks. This assumption is also standard for conservative reclamation. Forkscan mitigates this, slightly, by masking the low 3 bits when it reads a word, since some data structures overload those bits to save space. There are, of course, many ways to hide pointers that are not detectable in a general sense, but this is the most common and is easy to catch.

Concurrent Data Structures. Forkscan is tailored towards concurrent data structures. Specifically, it leverages thread parallelism for performance, and takes advantage of the fact that most concurrent operations *do not mutate* the data structure. We implicitly exploit this natural workload skew to avoid high pressure on the copy-on-write mechanism used during the scan phase. This is by design, and Forkscan should not be compared to general garbage collectors for other applications that don't fit this profile.

Multiple Threads Retiring a Single Node. At present, retiring a node multiple times from different threads might cause the same node to be tracked (and not found) in two subsequent iterations, causing a double-free. For this reason, adapting Forkscan to support this usage model would require fundamental design changes. However, in each of the microbenchmarks, as well as in memcached, finding the right place in the code to call `forkscan_retire()` was obvious: the thread that successfully marked the node removed was responsible for retiring it. A quick look at a variety of concurrent data structure designs in Herlihy and Shavit [33] shows that this is a common pattern. Therefore, we think that finding this place is generally easy, so there is no need to support multiple threads retiring the same node. That said, this is a possible avenue of future work if many concurrent data structures require or are simplified by that interface.

5.2 Future Work

In terms of extensions, we note that our design can be improved to provide several additional features.

Space Usage. Currently, Forkscan introduces a constant multiplicative overhead by storing a pointer to each retired object. This can be eliminated by directly utilizing the node information stored in the allocator, which subsumes these lists, by merging Forkscan with the allocator, itself. Many C/C++ programmers choose an allocator that suits their specific performance needs, however, and marrying Forkscan to its allocator makes it less general purpose. For portability, Forkscan is designed to work with any allocator.

Destructors. To work with classes in C++, the destructor must be called on objects before they are freed. The common concurrent data structures tested didn't need destructors or cleanup function calls, but user-built structures might. In particular, pointers to non-inlined objects could recursively be retired. This is more complicated than it appears because the destructor could be called when a thread's pool is nearly full, and if the destructor adds multiple objects, it could fill the pool. This would preclude using spare cycles to free objects while waiting for the aggregator to run, which user threads do in Forkscan.

Further Reduced Latency. After threads are signalled, they busy-wait until the fork is complete and they are allowed to continue. There is a cost associated with responding to the OS signal, as well as time spent waiting. Altogether, this does not impose significant latency, but the latency increases with thread count. On future architectures, it may begin to impose an unacceptable burden. The latency could be reduced if the `fork()` procedure, itself, had knowledge of the Forkscan algorithm. Instead of signalling the threads, Forkscan could simply fork the process, allowing each thread to continue until it hit a copy-on-write page, at which point the OS would know to stall it until the fork is complete. The OS also has access to the contents of the registers, allowing it to spill them into a special location known to Forkscan. Any thread that has not performed a write when the fork is complete can be signalled by the OS, and its register state recovered. This optimization would allow Forkscan to avoid signalling, and threads could run until the last possible moment.

Acknowledgments

We are grateful to our reviewers for their insightful comments and critiques. The paper is stronger for their input. William Leiserson, Alexander Matveev, and Nir Shavit were supported by the NSF under grants IIS-1447786 and CCF-1563880, and Dan Alistarh was supported by a Swiss National Fund Ambizione Fellowship.

References

[1] Y. Afek, H. Kaplan, B. Korenfeld, A. Morrison, and R. E. Tarjan. Cbtree: A practical concurrent self-adjusting search tree. In *Proceedings of the 26th International Conference on Distributed Computing*, DISC'12, pages 1–15, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-33650-8. . URL `http://dx.doi.org/10.1007/978-3-642-33651-5_1`.

[2] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 25:1–25:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. . URL `http://doi.acm.org/10.1145/2592798.2592808`.

[3] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit. The spraylist: A scalable relaxed priority queue. In *20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, San Francisco, CA, USA, 2015. ACM.

[4] D. Alistarh, W. M. Leiserson, A. Matveev, and N. Shavit. Threadscan: Automatic and scalable memory reclamation. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 123–132, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-

3588-1. . URL `http://doi.acm.org/10.1145/2755573.2755600`.

[5] H. Avni, N. Shavit, and A. Suissa. Leaplist: Lessons learned in designing tm-supported range queries. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 299–308, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2065-8. . URL `http://doi.acm.org/10.1145/2484239.2484254`.

[6] O. Balmau, R. Guerraoui, M. Herlihy, and I. Zablotchi. Fast and robust memory reclamation for concurrent data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 349–359, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4210-0. . URL `http://doi.acm.org/10.1145/2935764.2935790`.

[7] H.-J. Boehm. Bounding space usage of conservative garbage collectors. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 93–100, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9. .

[8] H. J. Boehm. Space efficient conservative garbage collection. *ACM SIGPLAN Notices*, 39(4):490–501, 2004.

[9] H.-J. Boehm. Boehmgc, 2015. Available at http://www.hboehm.info/gc/.

[10] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 157–164, New York, NY, USA, 1991. ACM. ISBN 0-89791-428-7. . URL `http://doi.acm.org/10.1145/113445.113459`.

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

[12] T. A. Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 261–270, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3617-8. . URL `http://doi.acm.org/10.1145/2767386.2767436`.

[13] N. Cohen and E. Petrank. Automatic memory reclamation for lock-free data structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 260–279, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5. . URL `http://doi.acm.org/10.1145/2814270.2814298`.

[14] N. Cohen and E. Petrank. Data structure aware garbage collector. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 28–40, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3589-8. . URL `http://doi.acm.org/10.1145/2754169.2754176`.

[15] L. Community. Linux 3.13, 2014. Available at http://kernelnewbies.org/Linux_3.13.

[16] C. Cutler. *Reducing Pause Times With Clustered Collection*. PhD thesis, Massachusetts Institute of Technology, 2014.

[17] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 261–269, New York, NY, USA, 1990. ACM. ISBN 0-89791-343-4. .

[18] D. Detlefs, P. A. Martin, M. Moir, and G. L. S. Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

[19] A. Dragojevic, M. Herlihy, Y. Lev, and M. Moir. On the power of hardware transactional memory to simplify memory management. In C. Gavoille and P. Fraigniaud, editors, *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 99–108. ACM, 2011. ISBN 978-1-4503-0719-2. . URL `http://doi.acm.org/10.1145/1993806.1993821`.

[20] T. Endo and K. Taura. Reducing pause time of conservative collectors. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, pages 119–131, New York, NY, USA, 2002. ACM. ISBN 1-58113-539-4. . URL `http://doi.acm.org/10.1145/512429.512432`.

[21] T. Endo, K. Taura, and A. Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Supercomputing, ACM/IEEE 1997 Conference*, pages 48–48. IEEE, 1997.

[22] J. Evans. Jemalloc, 2015. Available at http://www.canonware.com/jemalloc/.

[23] Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 124:5, Aug. 2004. URL `http://dl.acm.org/citation.cfm?id=1012894`.

[24] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the 23rd annual ACM symposium on Principles of Distributed Computing (PODC' 04)*, pages 50–59, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-802-4.

[25] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

[26] K. Fraser and T. L. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007.

[27] S. Ghemawat and P. Menage. Tcmalloc, Retrieved 2015. Available at http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[28] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Trans. Parallel Distrib. Syst.*, 20(8): 1173–1187, 2009.

[29] V. Gramoli. More than you ever wanted to know about synchronization: Synchrobench. In *Proceedings of the 20th Annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015.

[30] S. Hanke. The performance of concurrent red-black tree algorithms. In J. Vitter and C. Zaroliagis, editors, *Algo-*

rithm Engineering, volume 1668 of *Lecture Notes in Computer Science*, pages 286–300. Springer Berlin / Heidelberg, 1999. ISBN 978-3-540-66427-7. http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.6504.

[31] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the International Conference on Distributed Computing (DISC)*, pages 300–314, 2001.

[32] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, 2007.

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

[34] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of the 16th International Conference on Distributed Computing (DISC)*, pages 339–353, 2002.

[35] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. In *Proceedings of the 14th international conference on Structural information and communication complexity*, SIROCCO'07, pages 124–138, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-72918-1. URL http://dl.acm.org/citation.cfm?id=1760631.1760646. http://dl.acm.org/citation.cfm?id=1760631.1760646.

[36] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *Proceedings of the 22nd international symposium on Distributed Computing*, DISC '08, pages 350–364, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87778-3. URL http://dl.acm.org/citation.cfm?id=1432316. http://dl.acm.org/citation.cfm?id=1432316.

[37] H. Kermany and E. Petrank. The compressor: concurrent, incremental, and parallel compaction. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 354–363, 2006. . URL http://doi.acm.org/10.1145/1133981.1134023.

[38] D. Lea, 2007. http://java.sun.com/javase/6/docs/api/java/util/concurrent/ConcurrentSkipListMap.html.

[39] D. Lea, 2007. http://g.oswego.edu/dl/jsr166/dist/docs/java/util/concurrent/ConcurrentHashMap.html.

[40] R. L. Ltd. Memtier benchmark, Retrieved 2016. Available at https://github.com/RedisLabs/memtier_benchmark.

[41] A. Matveev, N. Shavit, P. Felber, and P. Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *SOSP*, 2015.

[42] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, , and M. Soni. Read-copy update. In *In Proc. of the Ottawa Linux Symposium*, page 338?367, 2001.

[43] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002.

[44] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15 (6):491–504, 2004.

[45] Microsoft. Windows virtual memory functions. https://msdn.microsoft.com/en-us/library/windows/desktop/aa366781(v=vs.85).aspx#virtual_memory_functions, Accessed: 2017-02-28.

[46] F. Pizlo, E. Petrank, and B. Steensgaard. A study of concurrent real-time garbage collectors. *SIGPLAN Not.*, 43(6):33–44, June 2008. ISSN 0362-1340. . URL http://doi.acm.org/10.1145/1379022.1375587.

[47] J. Rafkind, A. Wick, J. Regehr, and M. Flatt. Precise garbage collection for c. In *Proceedings of the 2009 international symposium on Memory management*, pages 39–48. ACM, 2009.

[48] G. Rodriguez-Rivera and V. F. Russo. Nonintrusive cloning garbage collection with stock operating system support. *Softw., Pract. Exper.*, 27(8):885–904, 1997.

[49] R. Schuetze. Concurrent garbage collection in D. http://rainers.github.io/visuald/druntime/concurrentgc.html, Accessed: 2017-02-28.

[50] R. Shahriyar, S. M. Blackburn, and K. S. McKinley. Fast conservative garbage collection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 121–139. ACM, 2014.

[51] R. Shahriyar, S. M. Blackburn, and K. S. McKinley. Fast conservative garbage collection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 121–139, 2014. . URL http://doi.acm.org/10.1145/2660193.2660198.

[52] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53:379–405, May 2006. ISSN 0004-5411. . URL http://doi.acm.org/10.1145/1147954.1147958. http://doi.acm.org/10.1145/1147954.1147958.

[53] N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 263–268. IEEE, 2000.