ThreadScan: Automatic and Scalable Memory Reclamation

Dan Alistarh
Microsoft Research
dan.alistarh@microsoft.com

William M. Leiserson
MIT
willtor@mit.edu

Alexander Matveev
MIT
amatveev@csail.mit.edu

Nir Shavit
MIT and TAU
shanir@mit.edu

ABSTRACT

The concurrent memory reclamation problem is that of devising a way for a deallocating thread to verify that no other concurrent threads hold references to a memory block being deallocated. To date, in the absence of automatic garbage collection, there is no satisfactory solution to this problem; existing tracking methods like hazard pointers, reference counters, or epoch-based techniques like RCU, are either prohibitively expensive or require significant programming expertise, to the extent that implementing them efficiently can be worthy of a publication. None of the existing techniques are automatic or even semi-automated.

In this paper, we take a new approach to concurrent memory reclamation: instead of manually tracking access to memory locations as done in techniques like hazard pointers, or restricting shared accesses to specific epoch boundaries as in RCU, our algorithm, called ThreadScan, leverages operating system signaling to automatically detect which memory locations are being accessed by concurrent threads.

Initial empirical evidence shows that ThreadScan scales surprisingly well and requires negligible programming effort beyond the standard use of Malloc and Free.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Data Structures, Design, Performance

Keywords

Synchronization, Memory Management

1. INTRODUCTION

An important principle for data structure scalability is having *traversals that execute without any synchronization*: sequences of reads with no memory updates (hence no memory fences, contention or cache pollution [25]) that work correctly by utilizing the semantics of the given data structure. The gain from such unsynchronized traversals is significant because traversals account for a large fraction of data structure operations, whether these search for a given item, or lead to updates, insertions or deletions of a small set of elements.

As a simple example, consider the lazy-list algorithm [22]. This is a concurrent linked list algorithm in which modifications to the list are done by acquiring fine-grained locks on the two nodes adjacent to where an insert or remove of a node is to take place. Because chances are typically low that there will be concurrent modifications to adjacent nodes, acquiring locks for these modifications introduces virtually no overhead. However, the frequent search operations traversing the list to reach the insertion or deletion point or to check if it contains a given item, are executed by reading along the sequence of pointers from the list head, ignoring the locks, and thus incurring no synchronization overhead. These unsynchronized traversals are key to the scalability of the data structure.

Not surprisingly, the unsynchronized traversal approach is increasing in popularity: it is at the base of the widely used read-copy-update (RCU) framework for designing concurrent data structures [36], as well as high performance structures such as hash-tables [26, 30, 42], search trees [1, 4, 13, 19, 23, 31], and priority queues [3, 43]. The unsynchronized traversals are in many cases *wait-free*, that is, they complete in a finite number of operations independently of ongoing data structure modifications. They can be used both for data structures that use locks for modifications and ones that do not (see for example [23, 31]), and deliver improved performance in both cases. A detailed survey of such structures can be found in [25, 38].

These high performance, unsynchronized traversals, although a boon to languages like Java (in the form of Java's Concurrency Package [29]), are more difficult to use in languages like C and C++ which have no garbage collection. Because they use no locks, memory fences, or shared memory writes, traversing threads leave no indication for other threads to detect what they are reading at any given time. To C and C++, the unsynchronized traversals are thus *invisible*. This invisibility makes memory reclamation a nightmare, because a thread wishing to reclaim a memory block has no way of knowing how many threads are concurrently traversing that block.

SPAA'15, June 13–15, 2015, Portland, OR, USA.

http://dx.doi.org/10.1145/2755573.2755600.

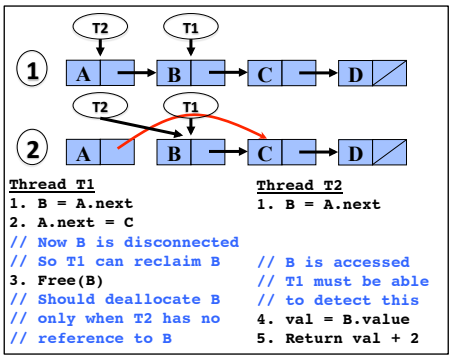

Figure 1: Concurrent memory reclamation for a linked-list: thread T1 deletes node B while thread T2 concurrently accesses the same node. Before calling free(B), thread T1 first "disconnects" B, by removing any heap (or shared) pointers that may lead to B, and only then it calls free(B) that is responsible to detect if some other thread has a reference to B. In memory reclamation, the programmer is responsible to disconnect the node, and the memory reclamation protocol is responsible to implement the free() function.

1.1 Concurrent Memory Reclamation

The *concurrent memory reclamation problem* is thus to devise techniques that will allow a deallocating thread to verify that no other thread has a reference to that block. More precisely, a programmer first makes this block "unreachable," by removing any shared reference that may lead to this block, and then uses the free(..) method of the memory reclamation scheme to safely deallocate this block. Any concurrent data-structure to be used in practice, in particular high performance ones that have invisible traversals, must include a solution to this problem. Otherwise, after a shared object gets deallocated, threads that still have a private reference (or a *dangling pointer*) to this object may access "garbage" memory and execute on an inconsistent memory state. As an example, consider the execution shown in Figure 1 on a concurrent linked-list: thread T1 deletes node B while thread T2 concurrently accesses the same node. Notice that, before calling free(B), thread T1 first "disconnects" B, by removing the shared reference from A to B (that may lead to B), and only then it calls free(B) that initiates the memory reclamation protocol.

This makes concurrent memory reclamation a simpler problem than garbage collection [5, 34, 8, 28], as we are not interested in shared references that are used to construct a data-structure on the global heap, or any shared references that are used to transfer or communicate nodes between threads. It is the responsibility of the programmer to eliminate those shared references before actually calling free(..). Providing a scalable and easy-to-use solution to the memory reclamation problem is important because historically, many high performance applications are written in unmanaged languages like C and C++ that have no built-in garbage collector support, and because of the large existing code base, these languages are likely to remain in high demand for decades to come.

As it turns out, despite being simpler than garbage collection, to date, there is no satisfactory solution to the problem: existing tracking methods are either prohibitively expensive or require significant programming expertise. Known techniques roughly fall into three categories:

Reference counting. These techniques originated in the realm of garbage collection. Algorithms such as [11, 17, 44], smart pointers [39, 41] and more recent improvements using hardware transactions [12], assign shared counters to objects, and use them to count the number of references to an object at any given time. Reference counting schemes are automatic: counters and counter modifications can be added to code at compile time. However, in all these schemes, the counting process is expensive, requiring a shared memory read-modify-write operation for each shared memory read, making them prohibitively expensive and eliminating the scalability advantages of using invisible read-based traversals.

Pointer-based. Hazard pointers [37] and *pass-the-buck* [24], or the more recent Drop-the-Anchor technique [7] require the programmer to explicitly declare the currently accessed locations, and then track them dynamically. This is a complex task even for simple data structures such as linked lists. In practice, using hazard pointers also requires modifications to the data structures to make them "hazard pointer friendly." The tracking process itself introduces a significant performance overhead, as all threads must synchronize with the reclaiming thread by executing a memory fence for each new hazard pointer. Thus, the performance advantages of having invisible readers are partly lost.

Epoch-based. Quiescence-based techniques [20, 14, 21] such as RCU [36] have threads accessing the data structure register the start and end of methods, and have the reclaiming thread wait a sufficiently long period of time until it can be ensured that no registered thread holds a reference to a deleted object. These techniques are efficient since here is no per-read tracking, making them compatible with invisible traversals. However, a delayed thread may prevent the reclamation process. Also, there are still significant programming complexities: for example, in the RCU technique, references that move outside methods must be tracked manually in ways similar to hazard pointers. This makes the design of complex data structures using such techniques a matter for concurrent programming experts [9] and does not allow for automation.

Recent work [12, 2] has explored the potential of Hardware Transactional Memory (HTM) to simplify memory reclamation. Dragojevic et al. [12] showed that HTM can be used to speed up and significantly simplify prior reclamation schemes, while Alistarh et al. [2] devised a technique which streamlines the tracking of node references, by ensuring that each data structure operation appears as having executed inside a single hardware transaction. While these techniques are promising, their performance fundamentally relies on HTM, whose availability is currently limited.

Summing up, there is currently no concurrent memory reclamation scheme that is widely-applicable, has low overhead, and works without careful programmer involvement.

1.2 Our Contribution

In this work, we take a new approach to concurrent memory reclamation: instead of manually tracking accesses to memory locations as done in techniques like hazard pointers,

or limiting reclamation to specific code boundaries as done in RCU, we use the operating system signaling and thread control to automatically detect which memory locations are being accessed.

Our protocol, called ThreadScan, is designed as a memory reclamation library: the programmer provides a data structure implementation with correct *free* calls, and ThreadScan will implement it automatically ensuring efficient memory reclamation. Our main technical contribution is the protocol for tracking memory references both automatically and efficiently.

At a high level, ThreadScan works as follows: when a thread deletes a node, it adds it to a shared delete buffer. When the buffer becomes full, the thread inserting the last node initiates a Collect procedure, which examines memory for references to nodes in the delete buffer, and marks nodes which still have outstanding references. The reclaiming thread frees nodes which are no longer referenced.

The key challenge in ThreadScan is to provide an automatic and efficient implementation of Collect.

Figure 2 illustrates the key idea of ThreadScan: when initiating a Collect, the reclaiming thread sends signals to all threads accessing the data structure, asking them to scan their own stacks and registers for references to nodes in the delete buffer, and to mark nodes in the delete buffer which might still be referenced. Threads execute this procedure as part of their signal handlers. At the end of this process, each thread replies with an acknowledgment, and resumes its execution. Once all acknowledgments have been received, the thread reclaims all unmarked nodes and returns.

There are two main advantages to this design. First, ThreadScan is shielded from errors in data structure code, such as infinite loops: these will not prevent the protocol from progressing, since the operating system signal handler code always has precedence over the application [27].

Second, ThreadScan offers strong progress guarantees as long as the operating system does not starve threads. In particular, notice that since the reclaiming thread waits for acknowledgments, the reclamation mechanism could in theory be *blocking*. This is not an issue in a standard programming environment, since each participant must finish executing ThreadScan in the signal handler before returning to its code. Therefore, the only way a thread may become unresponsive is if it is starved for steps *by the operating system*. This phenomenon is highly unlikely in modern operating systems, which schedule threads fairly: for instance, the Linux kernel avoids thread starvation by using dynamic priorities [35, 40]. At the same time, we emphasize that all other data structure operations preserve their progress properties, as ThreadScan adds a bounded number of steps to their execution.

The cost of the memory scan is amortized among threads by having each thread scan its own stack and registers, marking referenced nodes in the delete buffer. The scan is performed word-by-word, checking each chunk against pointers in the delete buffer. ThreadScan does assume that the programmer will not actively "hide" pointers to live nodes.[1]

We implemented ThreadScan in C to provide memory reclamation for a set of classic data structures: Harris' lock-free linked list [20], a lock-based skip-list, and a concurrent hash-table algorithm [25]. We ran tests on an Intel Xeon chip with 40 cores, each multiplexing two hardware threads. We compared ThreadScan against Hazard Pointers [37], epoch-based reclamation [20], and StackTrack [2], as well as an un-instrumented implementation which leaked memory. ThreadScan matches or outperforms all prior techniques, improving performance over Hazard Pointers by 2x on average, and provides similar performance to the leaky implementation. These findings hold even when the system is oversubscribed. Stack scans are the main source of overhead for ThreadScan, although the empirical results show that the overhead is well amortized across threads and against reclaimed nodes.

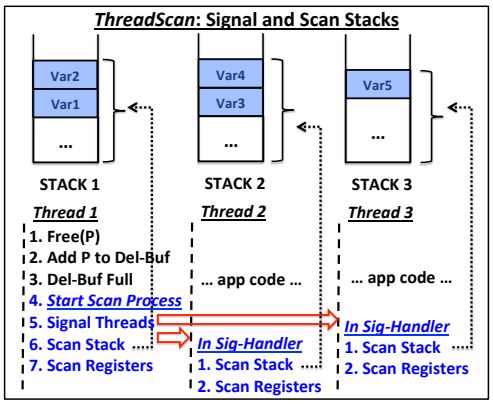

Figure 2: ThreadScan protocol illustration. Thread 1 calls Free(P) and this makes the delete buffer full. As a result, Thread 1 initiates a reclamation process that sends a signal to other threads and makes each thread to scan its own stack and registers. After all threads are done, Thread 1 traverses the delete buffer and deallocates nodes that have no outstanding reference to them.

Besides its low footprint, the key advantage of ThreadScan is that it is automatic: the programmer just needs to pass nodes to its interface, which handles reclamation. We believe this ease-of-use can make ThreadScan a useful tool for designing scalable data structures.

ThreadScan works under the following assumptions on the application code. First, we assume that nodes in the delete buffer have already been correctly unlinked from the data structure, as is standard for memory reclamation [24]. Second, pointers to live objects should be visible to the scan, which precludes the use of pointer masking techniques. Finally, we assume a bound on the number of reclamation events that may occur during the execution of a method call, which is enforced by batching deletes.

The ThreadScan library "hooks" into the signaling mechanism of the operating system, so it is independent of bugs or infinite loops that may occur inside the application. As a result, the ThreadScan library guarantees that each operation completes within bounded time as long as all threads continue to be scheduled by the operating system. In particular, it is sufficient that signal handler code is scheduled fairly by the operating system.

The source code is available from Github [33]. It is released to the public under the MIT license, which permits copying, modifying, and redistribution with attribution.

Roadmap. We give an overview of related techniques in Section 2, and give a formal definition of the problem in Sec-

[1]This assumption is similar to assumptions made in conservative garbage collectors [6], and is necessary for *automatic* reclamation.

tion 3. We describe ThreadScan in Section 4, where we also provide an overview of implementation details. We provide a proof of correctness in Section 5. Section 6 describes the experimental setup and the empirical results. We discuss the results in Section 7.

2. COMPARISON WITH PREVIOUS WORK

The concurrent memory reclamation problem was first formally described in [24]. Subsequently, a considerable amount of research went into devising practical solutions for this problem. As described previously, known approaches can be split across four different categories: reference counting [10, 17], epoch-based [20, 15], pointer-based [37, 24, 7], and HTM-based [12, 2].

Our scheme could be seen as a generalization of epoch-based reclamation, where we *enforce* quiescent states through signaling: the reclaiming thread signals all other threads, which scan their memory, isolating the subset of nodes which can be safely freed. The signal-based implementation avoids some of the main shortcomings of quiescence: references that cannot be used outside epoch boundaries, and thread delays inside application code that may delay the reclamation indefinitely. ThreadScan can be seen as enforcing quiescent periods via the signaling mechanism. The idea of producing a consistent view of memory for the reclaiming thread and scanning this view for references was also used in [2]. The key difference is that their technique guarantees view atomicity by wrapping all data structure operations inside hardware transactions; here, we avoid HTM by using an inter-thread signaling protocol.

The key departure from pointer-based techniques, such as hazard pointers, is the fact that we do not ask the programmer to explicitly mark live references through hazards, guards, or anchors. References are checked by the protocol, and therefore the interaction with the application code is limited to the signal handlers. However, our technique assumes that node references are not obfuscated by the programmer.

ThreadScan has a roughly similar structure to on-the-fly garbage collectors with deferred reference counting, e.g., [5, 34]. Such garbage collectors are typically structured in phases, separated by handshakes. Regular threads are usually aware of the collector state, as they may execute barrier operations, and co-operate with the collector in scanning their stacks. In particular, some collectors require threads to provide them with a view that is equivalent to an atomic snapshot of the stack [8, 28]. We exploit a weaker property in ThreadScan, by performing a non-atomic scan of the threads' memory, which we use to identify references to reclaimed nodes. Memory reclamation is fundamentally simpler than garbage collection, and as such we focus on minimizing the performance overhead of reclamation for the original data structure operations, while working within the confines of an un-managed programming language.

3. MEMORY RECLAMATION

3.1 Problem Definition

In the following, we consider a set of n threads, communicating through shared memory via primitive memory access operations. These operations are applied to *shared objects*, where each object occupies a set of (shared) memory locations. A *node* is a set of memory locations that can be viewed as a single logical entity by a thread. A node can be in one of several states [37]:

1. *Allocated.* The node has been allocated, but not yet inserted in the object.
2. *Reachable.* The node is reachable by following valid pointers from shared objects.
3. *Removed.* The node is no longer *reachable*, but may still be accessed by some thread.
4. *Retired.* The node is removed, and cannot be accessed by any thread, but it has not yet been freed.
5. *Free.* The node's memory is available for allocation.

Memory Reclamation. The *concurrent memory reclamation* problem is defined as follows [24, 37]. We are given a node (or a set of nodes) which are *removed*, and we are asked to move them first to state *retired*, then to state *free*. Notice that, in state *retired*, nodes are no longer accessible by any thread besides the reclaiming thread, and therefore cannot lead to access violations. The key step in the problem is deciding when a node can be *retired*, i.e., when it is no longer accessible by concurrent threads. Since we wish to perform memory reclamation automatically, it is important to describe which code patterns are disallowed by the above problem definition.

3.2 Mapping to Code

Reference Types. We split node references into two types, depending on the possible thread access patterns.

1. *Shared References:* A reference that can be accessed by more than one thread.
2. *Private References:* A reference that is local to a thread, and can be used only by a particular thread.

Data structure pointers are a standard example of *shared* references. Such references are usually located on the heap, and may be accessed concurrently by multiple threads. Local pointers, such as the ones a thread uses to traverse a list, are an example of *private* reference. Such references are usually stored in the threads' stack or registers.

Code Assumptions. In developing ThreadScan, we make the following assumptions, discussed below.

ASSUMPTION 1. *We assume the following hold.*

1. Shared References. *Nodes in the delete buffer have already been* removed*, and cannot be accessed through shared references, or through references present on the heap.*
2. Reclamation Rate. *There exists a fixed finite bound on the number of distinct reclamation events that may occur during any method's execution.*
3. Matching References. *Node references are word-aligned, and they can be matched to node pointers via comparison. Arbitrary memory words do not match to node addresses.*

Discussion. Assumption 1.1 follows from the definition of concurrent memory reclamation [24, 37], and in fact is one of the main distinctions from concurrent garbage collection. Assumption 1.2 is justified by the fact that reclamation is usually batched, freeing several nodes at once. Assumption 1.3 prevents the programmer from hiding references through arithmetic operations such as XOR. It is standard for similar tasks such as conservative garbage collection, and is intuitively necessary to allow automatic reclamation.

4. THREADSCAN

Generic Structure. Our algorithm is based on the following blueprint: once a thread wishes to reclaim a node, it adds a pointer to this node to a delete buffer, whose size is fixed by the application. Whenever the buffer is full, the thread which inserted the last node into the buffer becomes the *reclaimer*. For simplicity, we assume that there can only be a single reclaimer at a given point in time. (In practice, this is enforced by a lock and choosing a delete buffer of appropriate size.)

The chosen thread starts a ThreadScan Collect operation by signaling all other threads to help with examining references to nodes in the buffer. Thus, the ThreadScan algorithm consists of the implementation of the collect procedure on the reclaimer side, and of the implementation of the scan signal handler for all other threads accessing the data structure.

In the following, we describe the implementation of Collect which works under Assumption 1. The implementation ensures that, at the end of the Collect, each node in the buffer is either *marked* or *unmarked*. Marked nodes may still have outstanding references, and cannot yet be reclaimed. Unmarked nodes are safe for reclamation, and are freed by the thread as soon as the collect procedure completes.

4.1 ThreadScan Collect

Reclamation is organized in a series of *reclamation phases*, illustrated in Figure 2. In each phase, we are given a set of *unreachable* data structure nodes, and must reclaim a subset of them ensuring that this process does not cause any access violations.

The TS-Collect procedure, whose pseudocode is given in Figure 1, works as follows. The reclaiming thread first sorts the delete buffer, to speed up the scan process. Next, the reclaimer signals all other participating threads to start a TS-Scan procedure, and executes this procedure itself. This procedure will mark all nodes with outstanding references. The reclaimer then waits for an acknowledgment from all other threads. Once it receives all thread acknowledgments, it scans the delete buffer for unmarked nodes to free.

The TS-Scan procedure is called by the signal handler for all participating threads. Each thread scans its stack and registers word-by-word,[2] and looks for each in the delete buffer. If a possible reference is found, the node is marked in the delete buffer, which prevents it from being deleted in this reclamation phase. At the end of the scan, the thread sends an acknowledgment to the reclaimer, and returns to the execution of its application code.

4.2 Implementation Details and Limitations

The previous section provides a detailed overview of our technique, but omits several important implementation details, for simplicity of presentation. We provide these details below.

[2]The details of this procedure are given in Section 4.2.

Algorithm 1 ThreadScan Pseudocode.

```
 1: function TS-Collect( delete_buffer )
 2:     sort(delete_buffer)
 3:     for each thread t do
 4:         signal(t, scan)
 5:     end for
 6:
 7:     TS-Scan(delete_buffer)
 8:
 9:     wait for ACK from all other threads
10:
11:     for each pointer p in delete_buffer do
12:         if delete_buffer[p].marked == false then
13:             free(p)
14:         end if
15:     end for
16: end function
17:
18: function TS-Scan( delete_buffer )
19:     for each word chunk in thread's stack and registers do
20:         index = binary-search(delete_buffer, chunk)
21:         if index ≠ -1 then
22:             delete_buffer[index].marked = true
23:         end if
24:     end for
25:     signal(reclaimer, ACK)
26: end function
```

Signaling. We use POSIX Signals [45] for inter-thread communication. A thread that receives a signal is interrupted by the OS and begins running the signal handler immediately [27]. If another signal arrives during the execution of ThreadScan's signal handler, then a second handler will be pushed onto the thread's stack.

If the thread is blocked in a system call, the signal interrupts the system call and executes the signal handler. In this case, a system call implicitly restarts or returns the *EINTR* error code to the caller, that passes the restart responsibility to the programmer [3].

In general, if a thread is stalled due to any reason (for example, a context-switch), the OS resumes the thread immediately, and invokes the signal handler on the resume. Notice that this behavior is a standard OS feature that is used to kill or terminate threads that are stalled or stuck, and the idea of ThreadScan is to use this feature to ensure a prompt response from all of the participating threads.

Progress. The ThreadScan library uses the OS signaling system to scan the stacks. As a result, the only way to introduce delays into ThreadScan is by delaying the OS signals. This means that the progress guarantees of ThreadScan depend on the specific progress guarantees of the OS implementation. In modern systems, such as Linux, the system scheduler has a fair execution policy that is guaranteed not to starve threads, and therefore, in such systems the ThreadScan has a non-blocking progress guarantee.

Stack Boundaries. Our current implementation hooks the pthread's library *pthread_create* function call to detect the boundaries of thread stacks. In most practical cases, using this method is enough to identify the stacks, however, sometimes the stacks may have a more complex structure,

[3]A system call that returns EINTR documents this behavior and requires the programmer to handle these case

like the "Cactus" stacks in the Cilk programming language runtime [32]. Providing support for more complex stack structures is an interesting topic for future work.

Reclamation. For simplicity, our presentation assumed a single shared buffer to which pointers to reclaimed nodes are added. We implement a more complex, distributed version of this buffer to avoid false sharing on the buffer and contention on the index. Specifically, each thread has its own local buffer to which it adds pointers. When an individual buffer becomes full, that thread becomes the reclaimer and aggregates the pointers from all of the threads' buffers into a master buffer that is used for scanning. Individual buffers are circular arrays that are guaranteed to be single-reader, single-writer, so concurrent accesses are simple and inexpensive.

Further, we ensure that there is always at most a single active reclaimer in the system via a lock. In general, large delete buffer sizes ensure that this lock is not contended. Also, the above buffer construction has the consequence that a thread waiting to become a reclaimer will probably discover that its buffer has been drained into the master buffer, and that it can go back to work.

Pointer Operations. We assume that the underlying code does not employ pointer obfuscation. The scanning process masks off the low-order bits of memory it reads on a stack chunk, but if the pointer is obfuscated in some other way, for instance by XOR-ing, or if it is not word-aligned, the reference will not be detected by TS-Scan. We also assume that arbitrary memory chunks can not be interpreted as addresses to existing nodes in the delete buffer. While breaking this assumption does not affect the correctness of our protocol, it could prevent the reclamation of the target nodes. We consider this phenomenon unlikely.

4.3 ThreadScan Extension

Our ThreadScan algorithm, as described in Section 4, is able to detect private references that reside inside the stacks and registers of threads. However, a programmer may decide to pre-allocate a heap block and use this block to store private references on the heap (which violates code Assumption 1). For such specific scenarios, we extend the API of ThreadScan in a way that allows it to detect private references that reside on the heap.

Our extension is based on a simple observation: a heap block that holds private references is private to each thread. As a result, our extension introduces two additional methods:

1. $TS\_add\_heap\_block(start\_addr, len)$
2. $TS\_remove\_heap\_block(start\_addr, len)$

The add method allows the programmer to declare a preallocated heap block that the thread uses to hold private references, so that the ThreadScan signal handler can add those heap blocks to the scan process of this thread (in addition to the scan of the stack and registers). The remove method unregisters the region.

This extension makes the ThreadScan semi-automatic: the programmer must declare the per-thread heap blocks that may hold private references. However, ThreadScan still handles the scanning and reclamation process automatically.

5. CORRECTNESS PROPERTIES

5.1 Shared Memory Model

We assume a group of n threads which cooperate to implement a general abstraction O, providing a set of methods M. The implementation of each method $m \in M$ consists of a sequence of *steps*.

The interleaving of the threads' steps is decided by an abstraction called the *scheduler*. A scheduler is *fair* if it guarantees that each thread is scheduled to perform an infinite number of shared-memory steps. This implies a finite bound on the number of steps between the invocation of an operation by a thread and its application to shared memory.

5.2 Safety Properties

In the following, we prove correctness for the basic version of ThreadScan. These properties can be easily generalized for the extended variant presented in Section 4.3.

We first prove that, if all nodes in the delete buffer are in *removed* state, then ThreadScan will not reclaim any node that is not *retired*. Thus, reclamation cannot lead to any memory access violation.

LEMMA 1. *Under Assumption 1, any node* reclaimed *by* ThreadScan *has already been* retired.

PROOF. Assume for the sake of contradiction that there exists a memory node d reclaimed by the algorithm at time τ, but is not retired at this time. In particular, the node can be reached by a thread as part of the execution after τ.

We now take cases on the location of references to node d at time τ. Since the node was reclaimed, it must have been part of the delete buffer, and therefore it must have been removed (unreachable) after the beginning of the current reclamation phase, which we denote by t_0. Therefore, by Assumption 1.1, we have that 1) no such reference could have been present in the heap after time t_0 and that 2) no such reference can be accessed by more than one thread after time t_0.

It therefore follows that the reference must have been present in the stack or registers of some thread after t_0. Further, the reference is not *shared*, therefore it cannot travel between thread stacks. Hence, in order to be able to access the node after time τ, the thread must possess a reference to this node in its stack or registers throughout the time interval $[t_0, \tau]$. However, the algorithm ensures that the thread will scan its stack and registers during this time interval, and we are guaranteed to identify the reference by Assumption 1.3. This contradicts the assumption that the node d is reclaimed by the algorithm, completing the proof. □

5.3 Liveness Properties

Termination. We focus on the termination properties of operations under ThreadScan. First notice that, under Assumption 1.2, ThreadScan does not influence the termination of method calls that do not invoke free. This follows since ThreadScan may only add a bounded number of steps to any method invocation that does not invoke free.

LEMMA 2. *Under Assumption 1.2, any method implementation m that does not call* free *preserves its progress properties under* ThreadScan.

PROOF. Consider a method m consisting of a finite number of shared-memory steps, executing in the context of

ThreadScan. Since m does not call free, its corresponding thread may not initiate a reclamation phase. However, each time it receives a reclamation request, the thread must perform a bounded number of additional steps, scanning its stack and registers. Let B be a bound on this number of extra steps. Moreover, by Assumption 1.2 there exists a finite bound k on the number of times that the method invocation may receive scan requests. Therefore, ThreadScan adds a total of at most kB steps to any method invocation. Further, by the structure of the algorithm, if m does not call free, no such step is a busy-wait. This implies that the method preserves its progress properties, as claimed. □

We then prove termination for the free implementation. We assume a *fair* scheduler, and show that TS-Collect completes within a finite number of steps. We note that this property holds irrespective of the structure (lock-free, lock-based) of the original data structure.

LEMMA 3. *Under Assumption 1.2 and any fair scheduler, the* TS-Collect *call completes within a finite number of steps, irrespective of the progress conditions of the original implementation.*

PROOF. Recall that we assume that a single reclaimer exists at any one time. Assume such a reclaimer p, calling TS-Collect. Thread p has a finite number of steps to perform before busy-waiting for the replies to its reclamation signal. Since the scheduler is fair, these steps will be completed within finite time. Further, under Assumption 1.2, each non-reclaiming thread involved in the invocation of TS-Collect has a bounded number of steps B to perform before sending the acknowledgment to the reclaimer. Since the scheduler is fair, all threads will complete these steps within a finite number of steps, and send back an acknowledgment. Once these signals are received by the reclaimer, it returns within a finite additional number of steps. This implies that the reclaimer returns within a finite number of scheduled steps. □

Eventual Reclamation. Finally, we prove the following claim about the set of nodes freed in a reclamation phase. The proof follows from the fact that stack scans do not generate false positives. Hence, any node in the delete buffer which is not referenced by any thread will not match the result of a scan, and will hence be reclaimed.

LEMMA 4. *Under Assumption 1.3, for any reclamation phase, all nodes which can not be accessed through references in stacks or registers at the beginning of the phase will be retired by* ThreadScan.

6. EXPERIMENTAL RESULTS

Experimental Setup. Tests were performed on an 80-way Intel Xeon 2.4 GHz processor with 40-cores, where each core can multiplex 2 hardware threads. The ThreadScan library was configured to store up to 1024 pointers per thread. Threads tended to have relatively full buffers, so the total number of pointers any reclaimer worked with was roughly 1,000 times the number of threads in the process. For all tests, we used the highly scalalable TCMalloc [16] allocator.

Data Structures. Tests were run on three data structures:

1. **Lock-free Linked List:** Code was adapted for C from the Java provided in [25]. Each node was padded to 172 bytes to avoid false sharing.
2. **Lock-free Hash Table:** The Synchrobench suite [18] provided a hash table that used its own lock-free linked list for its buckets. This implementation was replaced with the [25] list.
3. **Lock-based Skip List:** StackTrack [2] provided an implementation with 104 byte nodes (representing the maximum size due to height). No padding was added to these nodes.

Techniques. We tested the data structures using the following reclamation techniques.

1. **Leaky:** The original memory leaking data-structure implementation without any memory reclamation.
2. **Hazard Pointers:** As introduced by Michael et al. [37]. The programmer manually declares and constantly updates the hazard pointer tracking information for shared memory accesses, and the reclaiming thread scans this information to determine nodes that can be deallocated. This was simulated in the linked list and hash table by introducing barriers after each read while advancing along the list. Actual hazard pointers were already provided in the skip list implementation [2].
3. **Epoch:** As introduced by Harris et al. [20] and McKenney et al. [36]. The programmer delimits the epoch-start and epoch-end points in the code, and the reclaimer waits for the epoch to pass, at which point it is safe to deallocate nodes. This was simulated in all three data structures by adding thread-specific counters to be updated before and after each operation. A thread that had removed 1024 nodes would read all epoch counters before continuing.
4. **Slow Epoch:** Represents the sensitivity of Epoch to application code that has thread delays: simulated by a 40ms busy-wait by the affected thread during its cleanup phase.
5. **ThreadScan:** Our new fully automatic technique as described in Section 4.

Methodology. Each data point in the graphs represents the average number of operations over five executions of 10 seconds. The update ratio was set at 20%, so about 10% of all operations were node removals.

Linked lists were 1024 nodes long, and the range of values was 2048. Hash tables contained 131,072 nodes with a range of 262,144. The expected bucket size was 32 nodes. Skip lists contained 128,000 nodes with a range of values of 256,000.

Results. Figure 3 shows the results for the three data structures under the various memory reclamation schemes. Up to the full 80 hardware threads, ThreadScan and Epoch scale along with the Leaky implementation. ThreadScan amortizes the cost of its reclamation phase over the operations being performed by the user application. Even with 10% removals, the cost of signaling and reclaiming nodes is distributed over the cheap operations performed on the hash table. Epoch likewise scales because the burden it imposes

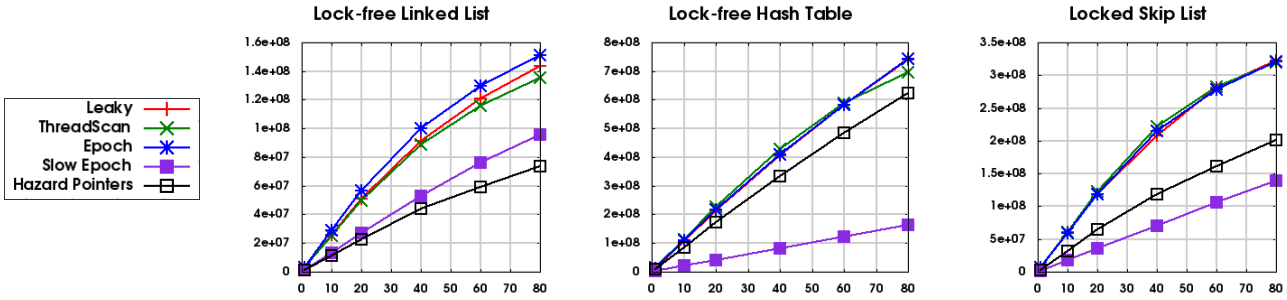

Figure 3: Throughput results for the lock-free linked list, lock-free hash table, and locked skip list: X-axis shows the number of threads, and Y-axis the total number of completed operations.

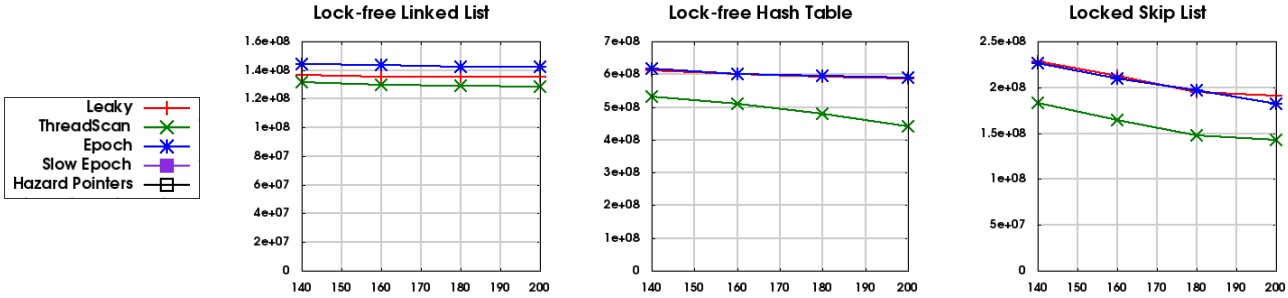

Figure 4: Throughput results for the oversubscribed system.

on the operations is low: two writes per method, except during reclamation where it must read every thread's epoch counter. But when all threads are completing their operations in a timely way, this overhead is low.

Slow Epoch, with an errant thread, shows significant burden because a thread that wants to free its pointers cannot do so until the errant thread updates its epoch counter. The reclaiming thread must wait until it has seen a change in the epoch counter of every thread that was in the midst of an operation. In this case, the thread that does not complete its operation for whatever reason holds up the reclaiming thread.

Hazard pointers scale well in the lock-free hash table because bucket traversals are short and so there are few memory barriers per operation. More overhead is visible in the other two data structures, however, as the number of hazard pointer updates increases. In the list and skip list data structures, the expected number of steps in a traversal is $O(n)$ and $O(\log(n))$, respectively. Since each step requires a barrier, even in a non-mutating operation, the overhead becomes significant.

The results of oversubscription tests, run on the same machine, are presented in Figure 4. Slow Epoch and Hazard Pointers were not included in the oversubscription experiment since they were shown not to scale well in normal circumstances. Oversubscription does not help their performance.

ThreadScan begins to show overhead versus the leaky implementation because not all threads can run simultaneously and the reclaimer must wait for all of them to complete. Additionally, overheads are higher because more signals are sent and the list of pointers to collect is larger, leading to more cache misses. Increasing the size of the delete buffer, and thereby reducing the frequency of reclamation iterations, is a useful way of amortizing the cost of signals and of waiting. However, it also increases the size of the list of pointers. The limitations of amortizing reclamation are clearly correlated with the costs of operations on the various data structures: The linked list overhead is negligible, the skip list is about 25% at 200 threads.

ThreadScan was tuned for the hash table to improve performance. The ThreadScan line presented in the hash table graph shows the results of increasing the length of the per-thread delete buffer length to 4096. Although this led to an improvement in performance, 25% overhead at 200 threads, comparable to the skip list, the overhead is still significant. Solving the oversubscription problem in a general way is an important avenue of future work.

7. CONCLUSION AND FUTURE WORK

Discussion. The memory reclamation problem has limited the adoption of high-performance non-blocking data structures in non-garbage-collected languages like C and C++. Since a thread may access a node without notifying other threads, a thread that wants to free the node must be defensive when reclaiming memory. Existing techniques for detection typically complicate the algorithm and/or render it inefficient.

In this paper, we have presented ThreadScan, an efficient method for detecting otherwise invisible reads. The user simply hands nodes to ThreadScan, which buffers them until there are enough to start a reclamation phase. It then leverages OS signals to force all threads to make their invisible reads visible. Because a thread need only make its reads visible during a scan, that cost can be amortized over the cost of freeing the pointers being tracked.

Our empirical results show that ThreadScan matches or outperforms the performance of previous memory reclamation techniques, taking advantage of the fact that its code runs as part of signal handlers, and is thus isolated from application code. Besides good performance, ThreadScan has the advantage of being completely automatic, as the programmer simply needs to link its data structure to use the TC-Collect calls implemented by ThreadScan.

We believe ThreadScan can be a useful general tool for the design and prototyping of non-blocking concurrent data structures. It makes implementation of these structures practical in C and C++ because of the low overhead and because it encapsulates all of the complexity of tracking down references.

Available at: `https://github.com/Willtor/ThreadScan`

Future Work. The main usability limitation of ThreadScan is in responsiveness of the reclaimer. The reclaiming thread must wait on the other threads and perform all the free calls, itself. The number of these calls is expected to scale linearly with the number of threads on the system, and therefore the reclaimer may become unresponsive at large thread counts.

In future work, we plan to investigate whether the latter problem may be solved by sharing the reclamation overhead, requiring scanning threads to call free for some subset of retired nodes in the subsequent TS-Scan call. TS-Scan would then check to see whether there are any pending nodes to free (from a previous iteration) after it has scanned its stack for the new set of nodes. This creates a trade-off between the latency improvement for the reclaiming thread versus the added overhead for the other operations. Another direction of future work is to apply ThreadScan to large legacy systems, such as concurrent databases or the kernel reference counted data-structures (for example, the VMA), to test both its interface and its potential to improve performance in a complex practical system.

8. ACKNOWLEDGEMENTS

Support is gratefully acknowledged from the National Science Foundation under grants CCF-1217921, CCF-1301926, and IIS-1447786, the Department of Energy under grant ER26116/DE-SC0008923, and the Oracle corporation. In particular, we would like to thank Dave Dice, Alex Kogan, and Mark Moir from the Oracle Scalable Synchronization Research Group for very useful feedback on earlier drafts of this paper.

9. REFERENCES

[1] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert 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.

[2] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir 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.

[3] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir 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] Hillel Avni, Nir Shavit, and Adi 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.

[5] Stephen M. Blackburn and Kathryn S. McKinley. 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, pages 344–358, New York, NY, USA, 2003. ACM.

[6] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 197–206, New York, NY, USA, 1993. ACM.

[7] Anastasia Braginsky, Alex Kogan, and Erez 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.

[8] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 125–136, New York, NY, USA, 2001. ACM.

[9] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using rcu balanced trees. *SIGPLAN Not.*, 47(4):199–210, March 2012.

[10] David Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

[11] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele, Jr. Lock-free reference counting. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, PODC '01, pages 190–199, New York, NY, USA, 2001. ACM. `http://doi.acm.org/10.1145/383962.384016`.

[12] Aleksandar Dragojevic, Maurice Herlihy, Yossi Lev, and Mark Moir. On the power of hardware transactional memory to simplify memory management. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 99–108, 2011.

[13] Mikhail Fomitchev and Eric 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. `http://doi.acm.org/10.1145/1011767.1011776`.

[14] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004.

[15] Keir Fraser and Timothy L. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007.

[16] Sanjay Ghemawat and Paul Menage. Tcmalloc, Retrieved 2015. Available at http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[17] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Trans. Parallel Distrib. Syst.*, 20(8):1173–1187, 2009.

[18] 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.

[19] Sabine Hanke. The performance of concurrent red-black tree algorithms. In Jeffrey Vitter and Christos Zaroliagis, editors, *Algorithm Engineering*, volume 1668 of *Lecture Notes in Computer Science*, pages 286–300. Springer Berlin / Heidelberg, 1999. `http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.6504`.

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

[21] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, 2007.

[22] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS 2005), Revised Selected Papers*, volume 3974 of *Lecture Notes in Computer Science*, pages 3–16. Springer, 2006. `http://dx.doi.org/10.1007/11795490_3`.

[23] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir 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. `http://dl.acm.org/citation.cfm?id=1760631.1760646`.

[24] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, May 2005.

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

[26] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Proceedings of the 22nd international symposium on Distributed Computing*, DISC '08, pages 350–364, Berlin, Heidelberg, 2008. Springer-Verlag. `http://dl.acm.org/citation.cfm?id=1432316`.

[27] Michael Kerrisk. *The Linux Programming Interface*. No Starch Press, Inc., San Francisco, CA 94103, 2010.

[28] Gabriel Kliot, Erez Petrank, and Bjarne Steensgaard. A lock-free, concurrent, and incremental stack scanning mechanism for garbage collectors. *SIGOPS Oper. Syst. Rev.*, 43(3):3–13, July 2009.

[29] Doug Lea. Java concurrency package, 2005. Available at http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/.

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

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

[32] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In Valentina Salapura, Michael Gschwind, and Jens Knoop, editors, *19th International Conference on Parallel Architecture and Compilation Techniques (PACT 2010), Vienna, Austria, September 11-15, 2010*, pages 411–420. ACM, 2010.

[33] William M. Leiserson. Threadscan git repository, 2015. Available at https://github.com/Willtor/ThreadScan.

[34] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for java. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, January 2006.

[35] Robert Love. *Linux System Programming, 2nd Edition*. O'Reilly Media, Sebastopol, CA 95472, 2013.

[36] 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.

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

[38] Mark Moir and Nir Shavit. Concurrent data structures. *Handbook of Data Structures and Applications*, pages 47–14, 2007.

[39] Objective-C, 2014. `http://en.wikipedia.org/wiki/Automatic_Reference_Counting`.

[40] Mark Russinovich and David A. Solomon. *Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition*. Microsoft Press, 5th edition, 2009.

[41] Anthony Savidis. The implementation of generic smart pointers for advanced defensive programming. *Softw., Pract. Exper.*, 34(10):977–1009, 2004.

[42] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53:379–405, May 2006. `http://doi.acm.org/10.1145/1147954.1147958`.

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

[44] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 214–222, 1995.

[45] WIKI. `http://en.wikipedia.org/wiki/Unix_signal`.