Fast Non-intrusive Memory Reclamation for Highly-Concurrent Data Structures

Dave Dice
Oracle Labs, USA
dave.dice@oracle.com

Maurice Herlihy
Brown University and Oracle Labs, USA
maurice.herlihy@oracle.com

Alex Kogan
Oracle Labs, USA
alex.kogan@oracle.com

Abstract

Current memory reclamation mechanisms for highly-concurrent data structures present an awkward trade-off. Techniques such as epoch-based reclamation perform well when all threads are running on dedicated processors, but the delay or failure of a single thread will prevent any other thread from reclaiming memory. Alternatives such as hazard pointers are highly robust, but they are expensive because they require a large number of memory barriers.

This paper proposes three novel ways to alleviate the costs of the memory barriers associated with hazard pointers and related techniques. These new proposals are backward-compatible with existing code that uses hazard pointers. They move the cost of memory management from the principal code path to the infrequent memory reclamation procedure, significantly reducing or eliminating memory barriers executed on the principal code path.

These proposals include (1) exploiting the operating system's memory protection ability, (2) exploiting certain x86 hardware features to trigger memory barriers only when needed, and (3) a novel hardware-assisted mechanism, called a *hazard look-aside buffer* (HLB) that allows a reclaiming thread to query whether there are hazardous pointers that need to be flushed to memory. We evaluate our proposals using a few fundamental data structures (linked lists and skiplists) and libcuckoo, a recent high-throughput hash-table library, and show significant improvements over the hazard pointer technique.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques — concurrent programming

General Terms Algorithms, Performance

Keywords memory reclamation, concurrent data structures, hazard pointers, memory barriers

1. Introduction

The widespread use of multicore platforms has produced a growing interest in the design and implementation of concurrent data structures that minimize the use of locks. These data structures typically consist of a collection of nodes linked by pointers. Threads navigate through these links, adding nodes to or removing nodes from the structure. These data structures present the problem of *memory management*: when a node is unlinked from the structure, the memory it occupies must eventually be reclaimed. In managed languages such as Java or Go, unused memory is reclaimed automatically by a garbage collector. For languages like C and C++, however, memory management is the explicit responsibility of the programmer.

Current memory reclamation mechanisms present an awkward trade-off. Techniques such as *epoch-based reclamation* [15] perform well when all threads are running on dedicated processors, but the delay or failure of a single thread will prevent any other thread from reclaiming memory. Alternatives such as *hazard pointers* [29] are highly robust, but they are expensive because they require additional memory barriers (also known as memory fences) [18]. In particular, on mainstream multicore architectures with a total-store ordering (TSO) memory model, such as SPARC and x86, the hazard pointer technique requires a store-load memory barrier after every store of a hazard pointer.

The principal shortcoming of hazard pointers is that the common case is expensive: every thread that traverses a data structure must incur the cost of a memory barrier at each node it encounters, no matter how infrequently memory reclamation actually occurs or even if it does not occur at all (e.g., in read-only workloads). It would be preferable to displace such costs to the reclamation process itself, moving them out of the principal code paths.

In this paper, we propose three novel ways to alleviate the costs of the memory barriers associated with hazard pointers and related techniques. These proposals are backward-compatible with existing code that uses hazard pointers, in the sense that they require only minor changes to the module that manages hazard pointers, and no other changes to the application itself. Apart from preserving the simplicity of hazard pointers, the common ground for all these proposals is an attempt to move the cost of memory reclamation from the principal code path to the infrequent memory reclamation procedure. In particular, all these ideas significantly reduce or eliminate memory barriers executed on the principal code path.

The first idea uses the operating system's *memory protection* ability, available in most modern operating systems, to replace (frequent) memory barriers during traversals with (infrequent) global memory barriers executed by all threads during memory reclamation. The second idea does not require any operating system support, but it is x86-specific, and relies on a less known feature of this architecture that allows to force a global barrier. While this idea is not portable to other architectures, it allows us to get a better sense of the inherent costs of global memory barriers. Finally, instead of using indirect techniques to trick the hardware into making hazard pointers visible at the right time, we propose a simple hardware-assisted mechanism, called a *hazard look-aside buffer* (HLB), that allows a reclaiming thread to query whether there are hazardous pointers that need to be flushed to memory. As we discuss later in the paper, the functionality of HLB can be implemented as a part of store buffers employed by modern architectures, or as a separate unit operating in parallel with the store buffer. Given that most data

```
1  // shared table of hazard pointers
2  Node** hazardTable;
3
4  Node* hazardRead(Node** object, int index) {
5    while (true) {
6      Node* read = *object;
7      hazardTable[index] = read;
8      membar();
9      Node* reread = *object;
10     if (read == reread) {
11       return read;
12     }
13   }
14 }
```

Figure 1: Creating a hazard pointer

```
1  // thread-local list of retired nodes
2  __thread Node *retired;
3
4  void reclaim() {
5    Node* prev = retired;
6    Node* curr = retired->next;
7    while (curr != NULL) {
8      if (contains(hazardTable, curr)) {
9        curr = curr->next;          // hazardous!
10     } else {
11       prev->next = curr->next; // safe
12       free(curr);
13       curr = prev->next;
14     }
15   }
16 }
```

Figure 2: Reclaiming a node using hazard pointers

structures that use hazard pointers store them in a circular buffer, using HLB of a size as large as the circular buffer will eliminate all hazard-induced barriers, including the ones that might be required by the reclamation procedure.

We evaluate the first two of our ideas in the context of a few fundamental data structures (linked lists and skiplists) and libcuckoo [25], a recent high-throughput hash-table library, and show significant improvements over the hazard pointer technique. In addition, we use the Pin tool [3] to evaluate the potential benefits of HLB. We find based on the Pin traces that in practice, depending on the size of the store buffer, even a very small HLB of a few entries can eliminate all hazard-induced memory barriers.

2. Background

Many data structures, such as linked lists [17], skiplists [14, 20], B-trees [4], queues [28], heaps [23], and hash maps [32], consist of a collection of nodes linked by pointers. Often, threads navigate through these nodes without acquiring locks. For example, using the *lazy list* algorithm [17, 19], threads traverse the data structure speculatively, without acquiring locks, then validate the target nodes (using locks or atomic operations) before making changes.

While lock-free navigation is typically more efficient than lock-based navigation, it requires more complex memory management, because nodes unlinked from the data structure cannot be reclaimed right away. The problem is that a thread might still have a reference to that node in a local variable at the time the node is unlinked. If the node's memory is immediately reclaimed, then that thread may dereference that address and find that node in an unexpected state. Instead, when a node is unlinked from the data structure, it is said to be *retired*, and a *grace period* must elapse before its memory can be safely reclaimed.

Using *hazard pointers* [29] (and related techniques [21]), a thread about to dereference an address publishes that address to warn other threads not to reclaim that memory. The procedure for reading a memory location using a hazard pointer protection is shown in Figure 1 (we omit C-style *volatile* keywords for clarity). In this example, the hazard table is implemented as a shared array, in which each thread is assigned an equal number of entries. The thread repeatedly reads the given pointer (Line 6), stores it in the shared hazard table (Line 7), performs a memory barrier (Line 8), and rereads the pointer (Line 9). If the pointer in memory is unchanged, that pointer is returned, and otherwise the loop resumes. As noted, this frequently-invoked memory barrier is the expensive part of this method.

A retired node is added to a thread-local list of retired nodes. As shown in Figure 2, to reclaim memory, the thread iterates through its retired list, testing whether that pointer is present in the shared hazard table (Line 8). The auxiliary *contains()* procedure simply scans the hazard table and checks whether any of its entries contains the given pointer. If not, it is safe to reclaim the node.

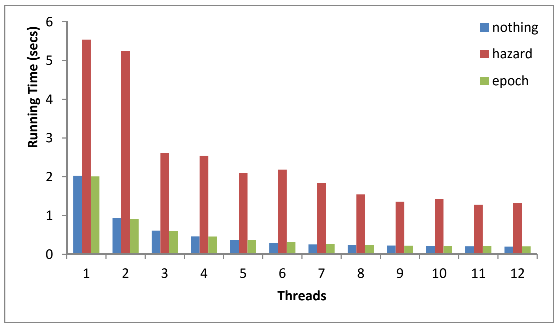
Hazard pointers are expensive because memory barriers are expensive, and a thread traversing a structure must execute a memory barrier each time a new node is traversed, making common operations expensive. Hazard pointers, however, are robust: a failed or delayed thread can prevent certain nodes from being recycled, but will not prevent other threads from allocating, retiring, and recycling memory.

By contrast, using *epoch-based* reclamation [15], threads execute in a succession of stages called *epochs*. Nodes retired during one epoch can be recycled as soon as all active threads have reached a sufficiently later epoch. The Read-Copy-Update (RCU) synchronization technique [27] is closely related to the epoch-based reclamation. To allow concurrency between multiple readers and an updater, RCU maintains multiple copies of shared objects (composing a shared data structure) and ensures that old copies are not freed up until all readers potentially accessing these copies complete their critical sections.

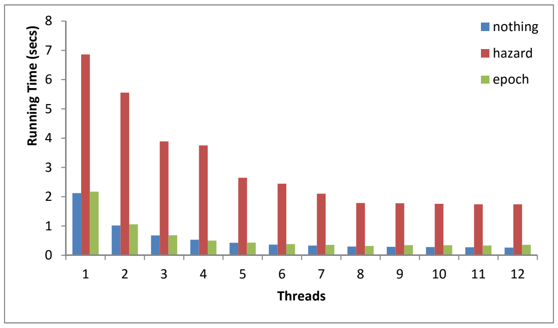
Under normal circumstances, epoch-based reclamation is typically faster than hazard pointers [18] because it requires fewer memory barriers, but it is not robust: if even a single thread is delayed, epoch-based reclamation prevents *every* thread from recycling retired nodes. Furthermore, while hazard pointers allow prompt reclamation of deleted objects, epoch-based approaches can allow quite a bit of “garbage" to accumulate between epochs, impacting memory footprint and cache residency. This is because these approaches have to wait for *all* threads to move into a later epoch, which may take a while if even one thread runs a long operation (e.g., scans a long linked list).

To illustrate these effects, we compared how these memory management algorithms perform on two simple highly-concurrent list implementations. As mentioned above, in the *lazy list* [19] threads search through the list without acquiring locks. An operation that modifies the list then locks the affected nodes, validates that they are correct, and then perform the modifications. By contrast, the *lock-free list* [17] replaces the lazy list's lock acquisitions with atomic compare-and-swap operations.

We ran a simple synthetic benchmark comparing the lazy and lock-free list implementations. We used an Intel Haswell processor (Core i7-4770), running at 3.40 GHz. The machine has a total of 8GB



(a) Lazy linked list



(b) Lock-free linked list

Figure 3: Linked lists with varying memory management algorithms (lower is better)

of RAM shared across four cores, each having a 32 KB L1 cache. Hyper-threading was enabled, yielding a total of eight hardware threads. Threads were not pinned to cores. The list implementations were compared using the following settings. List values range from zero to 10,000, and the list is initialized to hold approximately half of those values. The number of threads varies from 1 to 12, and collectively they call 100,000 operations, divided equally among the threads. Each time a thread calls an operation, it searches the list with 80% probability, and otherwise adds or removes list elements with equal probability.

Figure 3 shows the results. The left-hand bar shows the benchmark running time with no memory reclamation, the middle bar with hazard pointers, and the right-hand bar with epoch-based memory management. We are left with an unattractive choice: the hazard pointer mechanism is slower but tolerates thread delays and failures, while the epoch-based mechanism is normally faster, but easily disrupted by delays such as cache misses or by thread failures.

3. Related Work

A thread is said to be *quiescent* if it holds no references to any shared nodes in local variables. It is safe to reclaim a retired node's memory after every thread that accesses that shared data structure has passed through one or more quiescent states. Proposals such as *quiescent state-based reclamation* (QSBR) [18] and *epoch-based reclamation* [15] require each thread to report to the memory management library each time its application reaches a quiescent state. A retired node can be reclaimed after every thread has made such a report.

Although quiescence-based techniques have been observed to perform well in general [18], they are not *robust*: the delay (or failure) of a single thread will delay (or completely prevent) any thread from reclaiming any retired memory, arguably undermining the robustness benefits of lock-free traversals in the first place. By contrast, hazard pointers are more robust: the delay (or failure) of a single thread will inhibit reclaiming that thread's retired memory, but it will not inhibit reclamation of any other memory for any other thread.

Braginsky et al. [5] attempts to reduce the costs of hazard pointer memory barriers by updating hazard pointers (and triggering the associated memory fences) less frequently than proposed by the original technique [29]. To ensure robustness, a costly recovery procedure is invoked if a thread is suspected to fail. This recovery, which is expected to be executed very infrequently, requires copying the part of the data structure that the suspected thread may still be referencing. Here, memory fence overhead is shifted off the common path onto the more complicated failure recovery path. While Braginsky et al. [5] showed that this trade-off substantially improves performance of linked lists, applying their ideas to other data structures is challenging.

Morrison and Afek [31] observe that fence-free hazard pointers are possible in a so-called *temporally bounded total store ordering* (TBTSO) memory model, where the time it takes a store to drain from the store buffer to memory is bounded. They discuss hardware extensions required to provide such a bound in systems implementing total-store order. Furthermore, they show how to adapt TBTSO to existing x86 systems by generating periodic timer interrupts on every core [31]. In practice, however, these interrupts impose a constant overhead on the system even when none of the threads accesses a shared data structure and/or reclaims memory.

In a different context of work-stealing algorithms, Gidron et al. [16], inspired by ideas from Dice et al. [10], describe a way to eliminate fences by binding the slow thread (the stealer in their context) directly to the core on which the fast thread (the consumer) is currently running. In the setting of memory reclamation, the slow thread would be the one reclaiming memory and the fast thread would be one of the threads accessing the data structure. As in the instance of TBTSO emulation, however, this approach is quite intrusive, and requires paying the cost of multiple thread migrations even when none of the threads accesses the data structure during memory reclamation.

Several recent papers describe alternative approaches to memory reclamation. Dragojevic et al. [13] and Alistarh et al. [1] show how to exploit hardware transactional memory and its strong atomicity feature to achieve effective memory reclamation. Multiple papers explore the use of inter-process signaling for memory reclamation. For instance, Brown describes an efficient extension to an epoch-based reclamation to support fault-tolerance [6]. When a thread reclaiming memory suspects that another thread T has failed, it signals T. In the signal handler, T jumps to specially-designed recovery code that restarts the current operation. Devising recovery code requires non-trivial reasoning about the correctness of the memory reclamation scheme. Furthermore, to avoid starving threads with long-running operations (such as traversals of long lists), one has to carefully tune the signaling timeouts.

Alistarh et al. [2] also utilize signaling to automatically detect which memory locations are accessed by concurrent threads. More specifically, threads keep retired nodes in a shared buffer; when this buffer overflows, the last thread reclaiming a node initiates a reclamation procedure, which includes sending signals to all other threads, requesting them to scan their stack and registers to identify any references to the retired nodes. Without maintaining a set of active threads, i.e., threads currently applying operations on the data structure, however, this approach might introduce unnecessary overhead to threads not accessing the data structure. Besides, this technique assumes that threads may have references to shared nodes only in their registers or on their stacks. This assumption does not hold for some data structure implementations, e.g., when heap-allocated pointers are used for traversals.

Finally, Cohen and Petrank [7] present a hybrid technique, where hazard pointers (and accompanying fences) are used for writes only. For reads, they employ an optimistic approach, where the validity of a read instruction is established *after* the read is performed. Specifically, each thread is equipped with a so-called warning bit, which is turned on by a reclaiming thread to notify threads that there is a danger of stale read access; a thread that performs a read and finds its warning bit set restarts its operations. This techniques aims to provide fast read operations. However, it requires that reads from stale memory location never trap. (A trap might happen if, for instance, the deallocated memory is unmapped by the operating system). Furthermore, this technique can be applied only to data structures expressed in a special *normalized form* [7].

In summary, the techniques introduced in this paper take advantage of the relative simplicity and robustness of the hazard pointer technique, which does not require refactoring the original data structure implementation. Moreover, our techniques are substantially less intrusive than those proposed by the prior work cited above, imposing virtually no overhead on threads that do not access shared data structures. This lack of overhead is achieved without introducing additional restrictions on the underlying system or data structure implementations: for example, there is no need to track the active set of threads or to use non-trapping loads. Furthermore, our techniques significantly reduce, and in some cases completely eliminate, the number of required memory fences by moving them to the infrequently executed reclamation procedure, away from the principal code path[1].

4. Exploiting Memory Protection

Modern operating systems provide the ability to *write-protect* memory pages. If a thread write-protects a page, then any thread that attempts to write to that page takes an interrupt, passing control to that thread's signal handler. When the signal handler returns, the interrupted write instruction is retried. The memory protection call forces a global memory barrier, flushing any pending writes to that page from store buffers to memory before that page can be protected.

When a thread traverses a data structure, it stores the address in the hazard table as in Figure 1, except it omits the memory barrier at Line 8.

For ease of presentation, we assume that the hazard table holding hazard pointers of all threads resides in a single memory page; the scheme works the same when the hazard table spans multiple pages. To test whether a retired node can be reclaimed, a thread scans the hazard table as shown in Figure 2, except that before it scans the table it write-protects the page where it resides and immediately removes write protection. Protecting the page forces to memory any pending writes to the table, so the execution looks as if traversing threads had been doing memory barriers all along.

If a thread tries to write a hazard pointer to a write-protected table, it takes an interrupt and jumps to a signal handler, which simply blocks until the page becomes writable again. When the page is writable, the thread returns from the signal handler and the write is retried. Note that the read retry loop of Figure 1 is still needed because the block could have been reclaimed between when its address was read and when its address was written to the hazard table.

This technique relies on the fact that write-protect operations are synchronous across all processors. That is, the write-protect operation involves a Translation Lookaside Buffer (TLB) coherence protocol, and does not return until all its effects have been made visible to all other processors and all pending stores to the write-protected page have been made visible as well. (For more details, see [11]]). The technique is safe on x86 and SPARC TSO. However, proving the safety of this approach on architectures with weaker memory models or alternative TLB coherence mechanisms is left for future work[2].

This scheme blocks threads accessing the hazard table as long as the page remains write-protected. Although this typically happens for a very brief moment, some delays in removing write protection are possible, e.g., if the thread performing memory reclamation is swapped out. To alleviate possible performance implications, one may extend this scheme by allocating hazard pointers to each thread on a separate page. During memory reclamation, a thread will write-protect the pages holding hazard pointers of other threads one by one, thus potentially blocking only one other thread at a time. As long as the memory reclamation is invoked infrequently, the increase in its cost due to using an API for protecting and unprotecting pages multiple times would be negligible.

We note that some Linux kernels provide a new system call, *sys_membarrier()*, which implements a system-wide memory barrier on all threads of the current process [8]. Conceptually, this system call has a similar functionality to the memory-protection technique described above, but without involving a memory management unit. It was introduced primarily in order to enhance an RCU implementation, and in particular, eliminate memory fences from reader code paths in *liburcu*, a library providing user-space RCU implementation (available at `http://liburcu.org`). While the Linux kernels of our systems do not support this system call, our future work includes evaluating it once it becomes more widely available.

5. Exploiting Hardware Features

Here is another way to force an infrequent global memory barrier (when memory is to be reclaimed), instead of frequent memory barriers (when data structures are traversed).

It turns out that on certain x86 architectures, an attempt to execute a split compare-and-swap instruction (i.e., a compare-and-swap instruction that spans two cache lines) will also force a global barrier. This feature exists for backward compatibility with legacy code developed for older processors. These processors implemented atomic instructions (e.g., compare-and-swap) by locking the memory bus ([26] pp.78-79. For additional details, a reader is referred to [9]). This was a conceptually simple implementation, but it impaired overall performance as unrelated atomic operations could not be executed concurrently (and in fact, any operation that required a memory bus could not execute concurrently with an atomic oper-

[1] We note that "asymmetric” techniques, i.e., techniques that shift work from a fast path to a slow path, exist in other contexts as well. One such example is a *biased locking* approach [12, 24, 31], in which a lock is reserved for a thread that acquires it frequently; the reservation allows subsequent acquisitions to take place without atomic operations.

[2] We believe that putting more code between the calls to write-protect the page and remove this protection (e.g., scanning the hazard pointer table before removing write protection from its page) might suffice.

ation). Modern x86 architectures switched to use cache-locking, where atomic instructions are implemented directly in the local cache of each processor by locking a single cache line holding the target memory location of the atomic operation. (This design is used by other architectures as well, e.g., SPARC [9]). Along with that, the bus locking was preserved to handle the case of atomics that span two cache lines. Clearly, it may not hold for all future x86 implementations, yet we investigate it here because it gives insight into how future hardware might be adapted to provide better support for this kind of memory management. We want to emphasize that this technique should not be considered a general-purpose approach nor a portable solution to the problem of enhancing hazard pointer performance.

As before, a thread traverses the data structure without executing memory barriers. When a thread seeks to reclaim memory, it performs a split compare-and-swap instruction immediately before reading the hazard table.

6. Architectural Extensions

In this section, we propose a simple hardware-assisted mechanism that combines the robustness of hazard pointers with the performance of epoch-based reclamation. It requires the following architectural changes:

- Two new instruction codes are needed, a special test operation, and a special store operation.
- One new hardware unit, the *hazard look-aside buffer* (HLB), snoops on the cache coherence protocol and interacts with the store buffer.
- There are no changes to the native cache coherence protocol.

This proposal makes minimal changes to the memory hierarchy. It replaces the frequent memory barriers required by hazard pointer reclamation with an infrequent additional cache-coherence transaction, while providing the same level of robustness as the hazard pointer scheme. In more detail, each core has a *hazard look-aside buffer* (HLB), a device, similar to a store buffer, that snoops on coherence traffic, and keeps track of hazardous pointers that may be in the store buffer. Before a thread reclaims a potentially hazardous memory block, it issues a cache-coherence transaction for that pointer, which queries the HLBs, who respond if that pointer is being written to memory. If no HLB responds, then as in the usual hazard pointer algorithm [29], the querying thread must check the hazard table residing in memory. However, recycling memory is done infrequently, out of the critical path.

Comparing this approach to the mechanisms discussed in Sections 4 and 5, we note one important difference. Just like virtually any mechanism forcing a global barrier on the system, the two mechanisms discussed in Section 4 and 5 may present an opportunity for malicious code to harm system performance (e.g., by repetitive invocation of write-protection operations on a shared page). The HLB-based mechanism, however, does not force a global barrier and thus is less susceptible to malicious side effects. It relies on cache-coherence queries, while the cost of such an HLB query is comparable to that of a load instruction.

6.1 Hazard Lookaside Buffers

We introduce two new instructions. The first is `hstore(ptr,tab)`, which replaces the store and memory barrier calls at Lines 7–8 in Figure 1. The second is `htest(ptr)`, which returns a Boolean (or sets a flag). If it returns **true**, then `ptr` is hazardous. Otherwise, the caller must check the hazard table to finish determining whether that pointer is hazardous.

There are now two steps to deciding whether it is safe to reclaim a node. First, if `htest(ptr)` returns **true**, then the pointer is a potential hazard. Otherwise, if that pointer is present in the hazard table, then it is a potential hazard. As explained below, every hazardous pointer will fail at least one of these two tests, so there is no need for a memory barrier after every store to a hazard pointer.

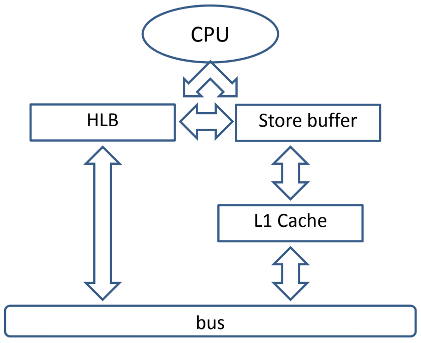


Figure 4: Architecture, including HLB

For ease of exposition, we assume a conventional bus-based architecture consisting of multiple CPUs, each with a store buffer and an L1 cache, where the caches run a MESI protocol over a shared bus (Figure 4). We assume that all addresses on the bus are cache-aligned (multiples of the cache line size). If `a` is an address, let `line(a)` denote the address of the first word in a cache line containing `a`.

There is one additional unit: each CPU has an HLB in parallel with the store buffer. The HLB also snoops on the bus[3].

The HLB is a set of *entries*, where each entry is a pair consisting of a *hazard pointer* and a *table address*. The HLB can be smaller than or the same size as the store buffer. Each HLB entry keeps track of a hazard pointer that has not yet been written to memory and the address to which it will be written. The HLB is searchable by both pointer and table fields.

Normally, when the CPU stores value `v` at address `a`, it creates an entry `(v,a)` in the store buffer, and the L1 issues a read-to-own bus transaction for address `line(a)`. (If the store buffer is full, the store stalls while the contents of the store buffer are moved to the L1 cache). If the CPU issues two stores to address `a`, and the second store takes place while the first is still in the store buffer, the second store's entry overwrites the first store's entry in the store buffer, a process called *write absorption*. A CPU can issue a *memory fence* that forces all entries in the store buffer into the L1 cache.

When the CPU issues a `hstore(ptr, tab)` instruction, it adds a `(ptr,tab)` entry to the HLB, and a `(ptr,tab)` entry to the store buffer. The HLB, like the store buffer, performs write absorption. If the HLB is full, it signals a memory fence to the store buffer, emptying both buffers. Conversely, if the store buffer is flushed by a fence, the HLB is also emptied.

When a CPU issues an `htest(ptr)` call, it broadcasts a bus transaction. In some architectures, this transaction would be similar to the one broadcasted for a load instruction. All HLBs snoop the bus, and if any of them has an entry `(ptr,∗)`, it replies.

[3] The HLB must offer latencies of the store buffer while having the ability to snoop the memory bus. We believe this is possible based on the fact that modern architectures feature L1 caches that are globally coherent, but still provide low latencies. Furthermore, HLB is expected to be very small. As we discuss in Section 7, HLB with just a few entries is expected to eliminate most hazard-induced memory barriers.

If a thread is swapped out, a memory fence will force stores in the store buffer to be written to the L1 (and dirty cache lines will be written back to memory), causing the HLB to be emptied as well.

Most data structures that use hazard pointers store them in a circular buffer, overwriting older pointers with new ones. For instance, a linked list implementation requires just two hazard pointers per thread, regardless of the size of the list [29]. If the HLB is at least as large as the circular buffer, then a thread can traverse a data structure without a single hazard-induced memory barrier. Otherwise, if the HLB is smaller and has a size of k entries, then the number of memory barriers is reduced by a factor of k.

6.2 Variations

If the HLB and store buffer have the same size, then they could be merged into a single unit combining the functionality of both. This architecture eliminates the need for a specialized `hstore` instruction, replacing it with a regular `store`.

The HLB could be designed to snoop on traffic from the store buffer to the L1 cache, and could discard an entry whose address is observed to be removed from the store buffer. This design reduces the likelihood of HLB overflow.

An alternative design combines reading the pointer from memory and placing that pointer in the HLB in a single `hload(ptrLoc,tab)` instruction. This design eliminates the loop in Figure 1. It requires more complicated logic because one must ensure that loading the pointer and writing it to the HLB happen atomically.

7. Experimental Evaluation

7.1 Data structure microbenchmarks

To evaluate and compare our proposals, we ran a series of simple benchmarks comparing the various memory management schemes. We used an Intel Haswell processor (Core i7-4770) as described in Section 2. Threads were not pinned to cores.

We focused on two linked data structures whose operations require traversals: lists and skiplists. We started with the *lazy* list and skiplist implementations [22], since they are known to be efficient. A lazy implementation deletes a node from the data structure in two stages: first, the node is marked as deleted (*logically deleted*), and second it is unlinked from the data structure (*physically deleted*). To add or delete a node, a thread traverses the data structure, without acquiring locks, until it finds its target node. The thread then locks both the target node and its predecessor (or predecessors, in case of skiplists), and validates that the locked node is the correct one. If so, the thread proceeds to modify the data structure, and if not, the thread unlocks the nodes and restarts the operation.

Because the original lazy list and skiplist implementations rely on garbage collection for storage management, they support *wait-free* membership queries: a querying thread traverses the data structure without acquiring locks. That thread might encounter logically or physically deleted nodes, but the garbage collector ensures that any such traversal is safe: no deleted node will be reclaimed while a traversal is in progress.

Unfortunately, we do not know any way for the lazy list or skiplist implementations to support wait-free queries when memory is managed using hazard pointers. The problem is that a physically deleted node may still be reachable from earlier physically deleted nodes, so it is not safe to reclaim that node even if it does not appear in the hazard table. To fix this problem, we adopt the approach used in Michael's original hazard pointer paper [29]: when a query traversal encounters a node marked as deleted, it simply restarts the query.

The list implementations were compared using a simple synthetic benchmark based on the following parameters. List values range from zero to 1000, and the list is initialized to hold approximately

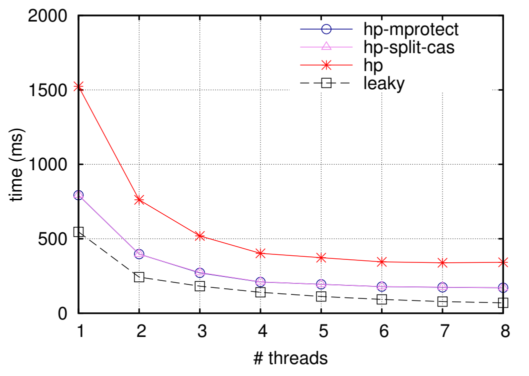


Figure 5: List Benchmark: 0% modifications

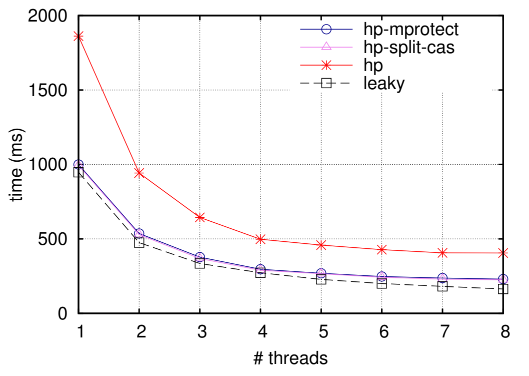


Figure 6: List Benchmark: 25% modifications

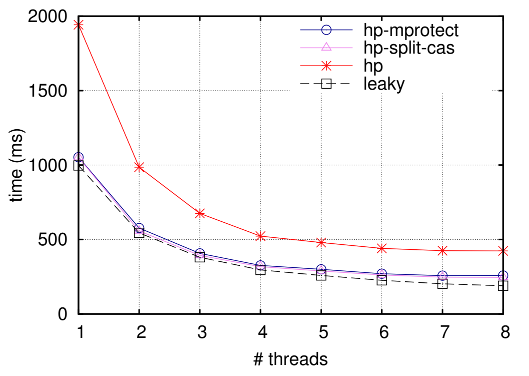


Figure 7: List Benchmark: 50% modifications

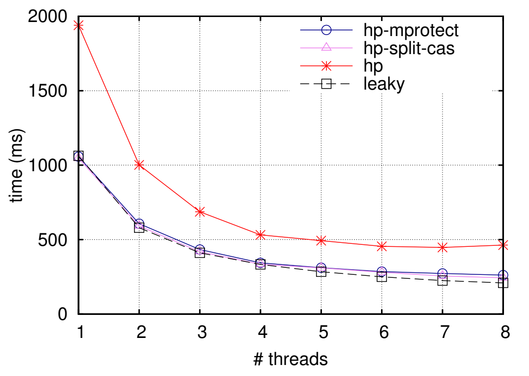


Figure 8: List Benchmark: 100% modifications

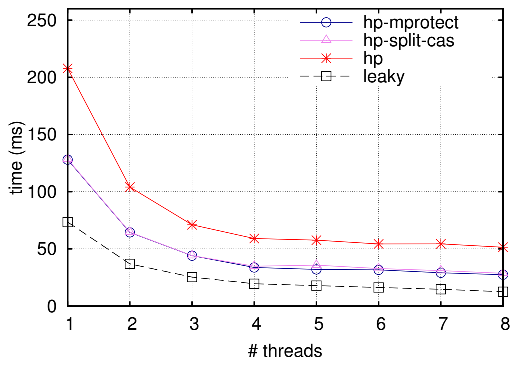


Figure 9: Skiplist Benchmark: 0% modifications

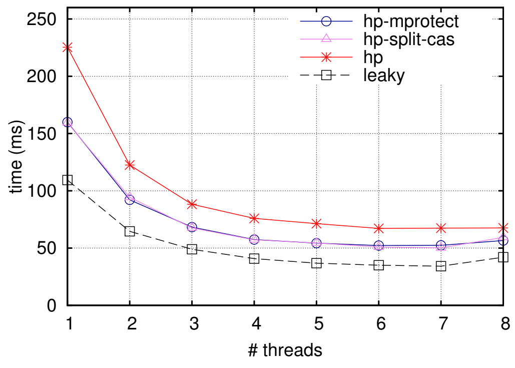


Figure 10: Skiplist Benchmark: 25% modifications

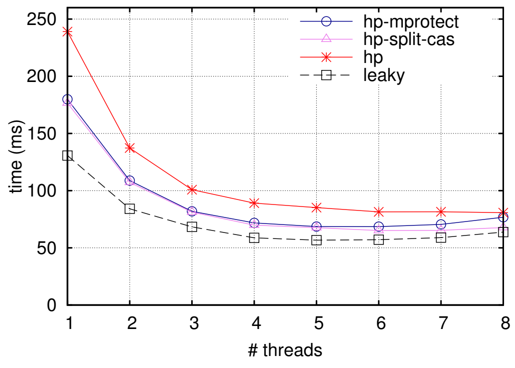


Figure 11: Skiplist Benchmark: 50% modifications

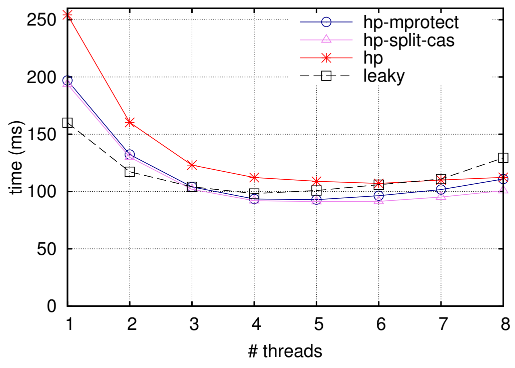


Figure 12: Skiplist Benchmark: 100% modifications

half of those values. The number of threads varies from 1 to 8, and collectively they call 1M operations. Each time a thread calls an operation, it chooses `contains()` with a probability that varies from 0 to 1, and otherwise calls one of the mutators (`add()` or `remove()`) with equal probability. We ran experiments with 0%, 25%, 50%, and 100% mutator operations. For each workload configuration, we performed five runs and report the median result. We note that the standard deviation of the reported results is negligible, in the order of a few percents or less from the mean for most of the results.

We tested the following memory management schemes.

- The *leaky* scheme does not use hazard pointers. It allocates but never recycles memory. This scheme provides an idealized baseline for the others.
- The *hp* scheme uses hazard pointers as described above, applying a memory barrier each time a node is traversed.
- The *hp-mprotect* scheme uses hazard pointers with memory protection replacing barriers.
- The *hp-split-cas* scheme uses hazard pointers with split-CAS replacing barriers.

The latter three schemes that use hazard pointers were configured to trigger the reclamation procedure when a new node needs to be allocated and the local list of retired nodes has reached the size of 64. This is done to amortize the cost of hazard table scans [29].

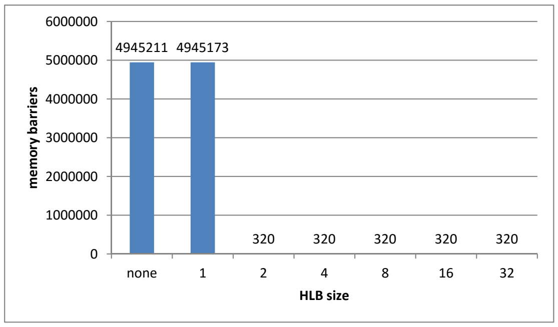
Figures 5–8 show the list benchmark running times for the various memory management schemes, numbers of threads, and mutator percentages. The memory management schemes fall into three rough categories: the *hp* scheme yields the worst performance by far, while (the unrealistic) *leaky* scheme yields the best. The *hp-mprotect* and *hp-split-cas* perform noticeably better than *hp* and close to (but slightly worse than) the *leaky* version (that does not use hazard pointers at all), especially for workloads that include mutation operations.

We note that all variants (except for *leaky*) have similar spatial performance in terms of the amount of retired nodes that could be deallocated, but that remain unreleased. The hazard pointer technique defers releasing only memory blocks that are still referenced by hazard pointers, and even then only for brief periods of time. Given that our proposals rely on hazard pointers, the spatial performance of all compared techniques (except for *leaky*) is the same for lists, as well as all other benchmarks. This is supported by the fact that the number of allocated and recycled nodes that we have measured, as well as the number of hazard table scans was similar for *hp*, *hp-mprotect* and *hp-split-cas* variants.

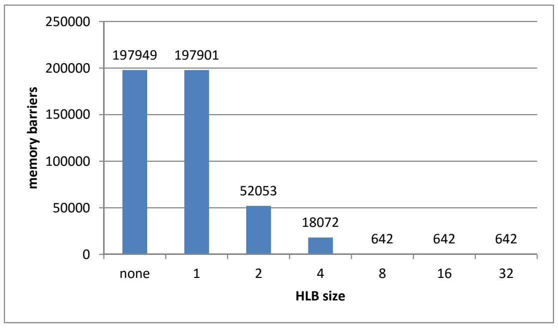
Figures 9–12 show the results of the skiplist benchmarks. Here, too, the memory management schemes fall roughly into the same three categories. Along with that, note that in Figure 12, both *hp-split-cas* and *hp-mprotect* outperform the *leaky* scheme starting at 4 threads. We believe this is because in this benchmark with a relatively high number of allocations and deallocations (due to the high rate of mutations), the small overhead of reclamation in *hp-split-cas* and *hp-mprotect* is easily compensated by reduced memory footprint of the skiplist, which results in improved memory locality. The observation that efficient memory management schemes can outperform leaky schemes was also reported in [6].

We used the PIN tool [3] to instrument the memory references produced by the list and skiplist benchmarks using the standard hazard pointer memory management scheme. The tool produced traces of memory reads and writes, distinguishing regular memory writes from writes to the hazard pointer table.

We wrote a simple simulator in Python to replay the traces and to simulate the effects of different-sized HLBs. For simplicity of the analysis, we assume each thread has its own core. Each core has a 32-line store buffer, where each cache line holds 64 bytes. Each core



(a) List



(b) Skiplist

Figure 13: Number of memory barriers with different HLB sizes

also has a hazard lookaside buffer of sizes 0 (no HLB), and each power of 2 from 1 to 32. Each address is divided by 64 to detect when addresses map to the same cache line. The simulator begins tracing each thread's memory references only after its first hazard pointer reference. Each regular write is placed in the store buffer, and each hazard table write is placed in both the store buffer and the HLB. Whenever either buffer reaches capacity, a memory fence is triggered, and both buffers are emptied.

This simulation is, of course, crude, but it provides a "back of the envelope" estimate of the kinds of gains that might be provided by various-sized HLBs. It does not take into account memory barriers forced by atomic operations, interrupts, and so on. We emphasize that in practice, we do not require that the store buffer actually waits until it is full before draining its contents. Instead, the interval between two successive memory barriers in the model represents a period of time during which every write that had entered the store buffer at the start must have left it by the end, perhaps one at a time. Moreover, because every write in the HLB is also in the store buffer, every write that had entered the HLB at the start of the period must also have left it by the end.

For a baseline, we count the number of barriers that occur when no HLB is in use: every store to the hazard table provokes a fence. We count the number of barriers that occur for each HLB size, and take the ratio of the two quantities. We also measure *gaps*, defined to be a sequence of successive regular memory stores to different cache lines that occur between successive stores to the hazard pointer table. If the typical gap size multiplied by the HLB size is larger than the store buffer size, then most barriers will occur when the store buffer reaches capacity, so the cost associated with hazard pointer management is minimal. If, however, the typical gap size multiplied by the HLB size is less than the store buffer size, then most barriers will occur when the HLB reaches capacity, and the cost associated with hazard pointer management will dominate.

We analyzed traces from the list and skiplist benchmarks. Since these benchmarks spend all their time traversing their data structures, they both make intensive use of hazard pointers. The benchmarks were compiled with the `-O3` option to encourage the compiler to minimize the number of memory references. For each benchmark, the number of threads ranged from 1 to 8, and the percentage of mutator operations from 0 to 50. These parameters had little effect on memory behavior.

The results appear in Figure 13. For the list benchmark, an HLB of size 1 provides no benefit, while an HLB of size 2 provides dramatic reduction in the number of barriers. The explanation is simple: Over 99% of the gaps were observed to be much smaller than the store buffer size, so an HLB of size 1 forces a fence at almost every write to the hazard table. The list implementation requires two hazard pointers per thread, since it keeps track of the current and previous nodes. Once the HLB can hold both of those entries, it suppresses any further hazard-related memory barriers.

For the skiplist benchmark, which tracks a variable number of hazard pointers, the improvement is more gradual, but equally dramatic. As with the list benchmark, an HLB size of 1 provides no benefit. As the HLB size increases to 2 and 4, the number of memory barriers shrinks by an order of magnitude, and when it reaches 8, it shrinks by another order of magnitude.

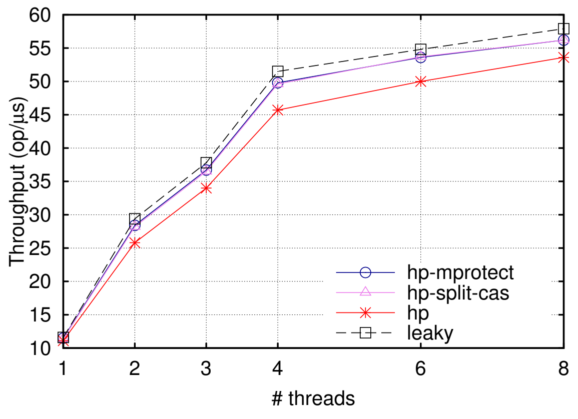
These benchmarks are synchronization-intensive, so one might expect more realistic benchmarks to have longer minimum gaps. In such a case, even an HLB of size 1 can significantly reduce the number of hazard-related memory barriers.

7.2 Experimenting with libcuckoo

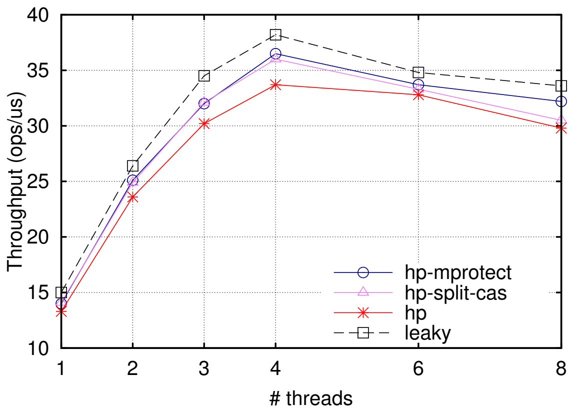
The libcuckoo library provides an implementation of a high-throughput, memory-efficient concurrent hash-table [25]. It uses fine-grained locks to support concurrent access of multiple readers and writers. As a result, it does not require hazard pointer to manage data stored in hash-tables. However, it uses hazard pointers to manage the hash-table itself, or more precisely, different instances of the table that may be created due to table expansion[4]. Thus, before every operation on the table, a thread places a hazard pointer on the current instance of the table. This ensures that this instance would not be deallocated if another concurrently running thread decides to expand the table and create another table instance.

The original implementation lacked a memory fence after setting a hazard pointer. We fixed this issue, and also implemented *hp-split-cas* and *hp-mprotect* variants. The latter was implemented by simply overloading the standard memory allocator for the std::list class used by the original application to implement the hazard table. In

[4] Recently, the authors of libcuckoo redesigned their implementation and removed the use of hazard pointers. We are using a version before the redesign, which was used in [25]. We note, however, that the git commit comment on the redesign change indicates that hazard pointers removal was done as a simplification step, not because of performance.



(a) Read-only workload



(b) 25% updates workload

Figure 14: Experimental results with libcuckoo.

addition, both variants required one line of code being inserted in the function traversing hazard table: calling a function invoking a split CAS for *hp-split-cas*, and causing the allocator to write-protect and unprotect the page it uses for allocation (i.e., the page holding all hazard pointers) for *hp-mprotect*. Hence, this benchmark exemplifies the compatibility of our ideas with the existing code that uses hazard pointers. In particular, no code changes outside of the module managing hazard pointers are required.

The results of the libcuckoo benchmark are shown in Figure 14. The read-only workload is shown in Figure 14(a), while the workload in which 25% operations are updates (that insert or remove a key) is shown in Figure 14(b). We used the default key range (32M) and the initial load of the table of 50%. Like with the microbenchmarks discussed in Section 7.1, the memory management schemes fall into the three categories. Although the differences in performance between all the four schemes are relatively modest, recall that in this benchmark a hazard pointer is updated once per operation rather than per traversed node. Thus, having the *leaky* variant outperform *hp* by up to 14% is remarkable. The *hp-split-cas* and *hp-mprotect* variants are able to uncover most of the performance degradation caused by the use of hazard pointers in read-only workload and roughly half of it in the workload that includes updates.

8. Conclusions

This paper observes that one of the performance shortcomings of the popular hazard pointer technique for memory management of concurrent data structures is the need for memory fences on the principal execution path that traverses nodes of the given data structure. We present three ideas for reducing this overhead by displacing the cost of memory fences to the much more infrequently executed reclamation phase. The first idea requires operating system support, available in most modern operating systems. The second idea does not require any support from an operating system, but is x86 specific. Lastly, the third idea proposes a simple hardware-assisted mechanism that allows a reclaiming thread to query whether there are hazardous pointers that have not been written to memory yet. We evaluate these ideas directly and by using the PIN tool [3] on a number of benchmarks, and find that they always improve over the hazard pointer technique, in some cases by very large margins. Critically, all the ideas discussed in this paper are backward-compatible with existing code that uses hazard pointers. As we demonstrate with the libcuckoo benchmark [25], they require only minor changes to the code managing hazard pointers and do not require any changes to the application itself.

We believe the ideas discussed in this paper are interesting and applicable in a more general context of memory barrier avoidance. Beyond memory management, this context includes the problems of efficient work stealing [30] and biased locking [12, 31]. Exploring the generalization of our ideas and their application to these and other problems is a part of our future work.

Acknowledgments

We would like thank our anonymous reviewers and our shepherd Christoph Kirsch for valuable suggestions to improve presentation.

References

[1] 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)*, 2014.

[2] 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)*, pages 123–132, 2015.

[3] M. M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal. Analyzing parallel programs with pin. *Computer*, 43(3):34–41, Mar. 2010.

[4] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9:1–21, 1977.

[5] A. Braginsky, A. Kogan, and E. Petrank. Drop the anchor: Lightweight memory management for non-blocking data structures. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 33–42, 2013.

[6] 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)*, pages 261–270, 2015.

[7] N. Cohen and E. Petrank. Efficient memory management for lock-free data structures with optimistic access. In *Proceedings*

of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA), pages 254–263, 2015.

[8] J. Corbet. sys_membarrier(). `https://lwn.net/Articles/369567`, Jan. 2010. Date Accessed: February 10, 2016.

[9] D. Dice. Qpi quiescence. `http://blogs.oracle.com/dave/entry/qpi_quiescence`, Feb. 2010. Date Accessed: November 11, 2015.

[10] D. Dice, H. Huang, and M. Yang. Asymmetric Dekker synchronization. Technical report, Sun Microsystems, 2001.

[11] D. Dice, H. Huang, and M. Yang. Techniques for accessing a shared resource using an improved synchronization mechanism, 2004. US Patent 7644409 B2.

[12] D. Dice, M. S. Moir, and W. N. Scherer, III. Quickly reacquirable locks, 2002. US Patent 7814488 B1.

[13] A. Dragojevic, M. Herlihy, Y. Lev, and M. 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.

[14] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 50–59, New York, NY, USA, 2004. ACM.

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

[16] E. Gidron, I. Keidar, D. Perelman, and Y. Perez. SALSA: Scalable and Low Synchronization NUMA-aware Algorithm for Producer-consumer Pools. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 151–160, 2012.

[17] T. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of 15th International Symposium on Distributed Computing (DISC 2001), Lisbon, Portugal*, volume 2180 of *Lecture Notes in Computer Science*, pages 300—314. Springer Verlag, Oct. 2001.

[18] 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, Dec. 2007.

[19] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. S. III, and N. Shavit. A lazy concurrent list-based set algorithm. In J. H. Anderson, G. Prencipe, and R. 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.

[20] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. In *SIROCCO*, pages 124–138, 2007.

[21] 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.

[22] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Mar. 2008.

[23] G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott. An efficient algorithm for concurrent priority queue heaps. *Inf. Process. Lett.*, 60(3):151–157, 1996.

[24] K. Kawachiya, A. Koseki, and T. Onodera. Lock reservation: Java locks can mostly do without atomic operations. *SIGPLAN Not.*, 37(11):130–141, 2002.

[25] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 1–14, 2014.

[26] R. Maddox, G. Singh, and R. Safranek. *Weaving High Performance Multiprocessor Fabric*. Intel Press, 2009.

[27] P. McKenney and J. Slingwine. Read-Copy Update: Using execution history to solve concurrency problems. *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[28] 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 Press, 2002.

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

[30] A. Morrison and Y. Afek. Fence-free work stealing on bounded tso processors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 413–426, 2014.

[31] A. Morrison and Y. Afek. Temporally bounding TSO for fence-free asymmetric synchronization. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–58, 2015.

[32] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. In *The 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 102–111. ACM Press, 2003.