# A Concurrent Map for Go

## Abstract

The Go programming language offers many features that simplify the creation of highly concurrent systems. However, it lacks built-in data structures that allow fine-grained concurrent access. In particular, its `map` data type, one of only two generic collection types in the language, limits concurrency to the case where all operations are read-only; any mutation (inserts, updates, and removals) requires exclusive access to the entire map.

In this paper, we extend the Go compiler and run-time to support a new concurrent map data type, the Interlocked Hash Table (IHT). By leveraging a novel optimistic synchronization protocol, the IHT permits concurrency among reads, writes, and iteration operations, without risking deadlock. We also allow large critical sections that access a single IHT element. The net effect is a highly scalable, high-performance data structure that offers straightforward and useful semantics, and outperforms all known concurrent maps for Go, achieving performance up to 7× better than the state of the art at 24 threads.

## 1. Introduction

Scalable concurrent data structures are notoriously difficult to craft. They must offer low latency at one thread, and high scalability as the number of threads increases. Additionally, programmers expect these data structures to have a wide interface, so that they can be used in a large number of situations. Concurrent data structures must offer a reasonable correctness criteria, so that programmers know what to expect when their are concurrent accesses, and they should be resilient to performance pathologies such as deadlock, starvation, and lock convoying.

To meet these requirements, language designers increasingly choose to provide a set of high-performance concurrent data structures as part of the language. For example, Java, through JSR 166, has provided general-purpose lock-based and nonblocking concurrent data structures for over a decade [20]. More recently, C++ added parallel data structures to its standard libraries [12], and is considering adding transactional constructs for concurrency [11]. In both languages, the data structures are being designed and implemented by experts who possess deep knowledge of concurrent algorithms, language-specific run-time overheads, and performance engineering [15].

Language features and concurrent data structure design go hand-in-hand. For example, garbage-collected languages (e.g., Java) have an easier time leveraging speculation, because the memory accessed by a thread executing a doomed but incomplete speculation cannot be recycled while any thread retains a reference to it. C++ allows fine-grained control of the placement of hardware memory fences, and template instantiation can avoid indirection by storing data, rather than references, in concurrent collections. Algorithms designed for one language are not easily ported to another language, especially if the goal is use in production systems.

In this paper, we focus on a concurrent `map` data structure for the Go language. Go's features are simultaneously high- and low-level, and it occupies a unique space in terms of concurrency. Like Java, Go is type-safe, garbage collected, and provides reflection and auto-boxing. Like C++, Go allows direct pointer access, through its `unsafe` package. Go's only support for generic programming is its opaque `interface{}` type. Its thread abstraction, the *goroutine*, is best thought of as a multi-CPU extension to Capriccio [24]: Go can multiplex hundreds of thousands of lightweight "goroutines" onto all the physical cores of a machine.

From a concurrent data structure perspective, goroutines primarily synchronize and communicate via message-passing (through `channels`). Shared-memory synchronization is possible through the `sync` package's `Mutex` and `RWMutex`; `sync` also provides atomic versions of primitive data types. While there are no built-in concurrent data structures, there are two built-in, generic, sequential collections, the `map` and `slice` (dynamic array). Both are highly optimized, and tightly integrated into the Go compiler and runtime. For example, the Go compiler generates a suitable hash function for a map, based on its key type (e.g., for multi-field struct keys, it will hash each field of that key), but it does so using mechanisms only available to the runtime.

In this paper, we introduce a new lock-based concurrent map for Go, which we call the Interlocked Hash Table (IHT). The IHT leverages Go's garbage collection, `unsafe` pointer access, and unconventional iterator semantics, to deliver low latency and high scalability. The IHT is implemented in the Go compiler and runtime, supports concurrent insert, lookup, remove, update, and iteration, and also provides a facility through which programmers can write large critical sections over a single map element.

The remainder of this paper is organized as follows. In Section 2, we review background material in concurrent data structure design, and then discuss properties of Go that impact the design of the IHT. Section 3 introduces the IHT, and Section 4 discusses IHT implementation. In Section 5, we present performance evaluation, using stress-test microbenchmarks. Lastly, Section 6 summarizes our findings and suggests future work.

## 2. Background and Related Work

In this section, we briefly discuss the factors that have the most influence on high-performance concurrent data structure design, and also discuss the implementation of the default (sequential) Go map.

### 2.1 Concurrent Data Structure Design

The first obligation of concurrent data structures is to avoid unnecessary interaction among threads. For lock-based algorithms, using a plurality of fine-grained locks can prevent threads from attaining mutual exclusion over too large of a region of memory, but introduce extra latency for each lock acquire/release. For nonblocking algorithms [7], the state of any thread cannot impede the forward progress of other threads. Particularly appealing are lock-free data structures, which do not allow deadlock or livelock but may admit starvation under pathological interleavings. These are often the most scalable and performant concurrent data structures [4].

Not all data structures can be made lock-free and fast. When an operation must atomically modify multiple locations to achieve its desired change to a data structure, the use of a single atomic hardware instruction, such as compare-and-swap (CAS), may not be sufficient, necessitating the use of a software simulation of multiword atomic operations (e.g., LLX/SCX [2] or multiword CAS [5, 18]). The latency of multiple CAS instructions within these simulations, and the complex helping protocols needed to ensure forward progress, can reduce throughput and increase latency. Even when only one CAS instruction per operation is required, many lock-free data structures require atomic copying. For example, updating an element in a nonblocking set, typically entails copying the element out of the set, modifying the copy, and then writing the new version back into the data structure. When the collection stores types larger than the machine word size, atomic copying becomes expensive.

Low-level techniques such as optimistic synchronization and laziness are often more important than nonblocking guarantees. A prime example is the lazy list [6]: it provides nonblocking list lookup operations, but uses locks when inserting and removing elements. The key techniques include avoiding lock acquisitions during traversal, validating the presence of a node in the list *after* locking it, leaving marked-but-invalid entries in the list for other threads to clean up at a later time, and leveraging garbage collection to ensure that data being accessed by concurrent "doomed" speculations is not reclaimed and re-allocated until after those speculations restart. The three most popular nonblocking maps also employ some of these techniques: the Split-Ordered List [21] and fixed-size nonblocking hashtable [19] use a nonblocking precursor to the lazy list as their fundamental data structure, and the lock-free resizable hashtable [16] lazily rehashes elements upon overflow of a bucket.

Concurrent data structures typically achieve a strong correctness guarantee, known as linearizability [9]. Linearizability guarantees that every operation appears to happen at a single instant in time, somewhere between when the operation was invoked, and when it provided a response to its caller. In nonblocking data structures, the point at which an operation linearizes is usually some CAS operation it issues. In lock-based data structures, the linearization point is usually some instruction within a lock-based critical section [8].

Linearizabile iteration is particularly hard to achieve. The most straightforward approach, atomic snapshots, are complex and may not scale [1]. Worse, programmers wishing to perform modifications during iteration are poorly served by snapshots, which can return a stale copy of a large data structure. As a result, many concurrent data structures have relaxed iterator semantics. In JSR166, iteration over a priority queue may not return elements in priority order, and iteration over a queue may "miss" elements in the queue. Still, these data structures guarantee that every element returned by the iterator was present in the data structure at the time when the iterator returned it. Several lock-based concurrent skiplists offer non-linearizable read-only iteration [4, 8, 17].

### 2.2 The Go Map: Implementation and Interface

A simplified description of the interface to the Go map appears in Table 1.[1] The compiler translates map accesses into calls to five core functions. When a `map` is indexed as an `rvalue` (e.g.,`value := map[key]`), a call to `mapaccess` is generated. When a `map` is indexed as an `lvalue` (e.g., `map[key] := value`), a call to `mapassign` is generated. Calls to `delete` an element in the `map` (e.g., `delete(map, key)`) are replaced with a call to `mapdelete`. Finally, both `mapiterinit` and `mapiternext` are generated during a `for...range` iteration over a `map`.

The Go map API fundamentally differs from the interfaces that are common in nonblocking data structure research. The map allows keys of sizes greater than a machine word, and hence techniques that rely on atomic reads of keys, such as lazy list's wait-free contains operation, be-

[1] The Go runtime includes several versions of each of these functions, based on the size and type of the key.

| | |
|---|---|
| `mapaccess(k)` | Returns an internal pointer to the `v` corresponding to `k`, if found. |
| `mapassign(k, v)` | Inserts `k` and `v` into the map, or updates them if they are already present. |
| `mapdelete(k)` | Removes `k` and its `v` from the map. |
| `mapiterinit(map, it)` | Initializes an iterator that iterates over `k`/`v` pairs in a randomized order. |
| `mapiternext(it)` | Yields a `k`/`v` pair from the map. |

Table 1: Compiler API for map accesses. `k` and `v` refer to a key and its value, respectively.

come significantly more complex. For lookup operations, `mapaccess` explicitly returns an internal pointer to a value inside of the map. This behavior, which resembles barriers in garbage collectors [25], is not compatible with known non-blocking techniques, because the linearization point [9] of the read occurs after the response of the lookup function. In a naive concurrent implementation of this interface, the internal pointer returned by `mapaccess` could be mutated while it is being accessed by a concurrent operation, yielding undefined behavior.

At the same time, the map interface and specification provide unique opportunities. Iteration is a fundamental feature of Go maps, and is required in order for certain runtime operations to interact with the map; this restricts the use of research concurrent data structures that do not provide iteration. However, the Go map is specified such that programmers cannot expect a map iteration to produce values in any particular order. For our purposes, this enables the use of randomization to prevent convoying when multiple threads iterate simultaneously. Additionally, the Go map is resizable (the implementation will grow, but never shrink, a map), and this is achieved via indirection. Thus concurrent maps need not incur overhead simply for introducing indirection: it is already inherent in the baseline.

## 3. Design of a Concurrent, Lock-based Map

We now present pseudocode and describe the behavior of the Interlocked Hash Table (IHT). While the presentation is not Go-specific, we assume certain Go features are present, such as garbage collection and simple atomic primitives (i.e., compare and swap (CAS), atomic loads and stores).

Whereas the Go sequential map is a flat array, which is resized by rehashing all elements in the array, our concurrent map is a fixed-depth tree. Figure 1 depicts the shape of the tree, and illustrates seven concurrent operations in-flight. Listing 1 introduces the two main data types used in the map construction: the `PointerList` and the `ElementList`.

The IHT consists of a `PointerList` and three constants: `DEPTH`, the maximum depth of the tree; `EMAX`, the maximum number of elements in an `ElementList`, and `PINIT`, the capacity of the root `PointerList`. `PointerList` buckets can be **nil**, reference `PointerLists`, or reference `ElementLists`. Once a bucket references a `PointerList`, it will never again be **nil** or reference an `ElementList`. Excluding the deepest level of the tree, once a bucket references a `PointerList`, it is immutable. At the last level, a bucket may reference a larger `PointerList` in the future.

**Listing 1:** IHT types. Array sizes are known at construction time, so that arrays can be inlined into `PointerLists`. The `ParentStruct` type encapsulates information about the bucket within the parent `PointerList` that references an object.

| | | |
|---|---|---|
| **Fields of PointerList Object:** | | |
| $l$ | : `CMLock` | // spinlock + type identifier |
| $parent$ | : `ParentStruct` | // parent bucket |
| $size$ | : `Integer` | // size of buckets array |
| $hashkey$ | : `Integer` | // seed for hash function |
| $buckets$ | : `ElementList[]` | // pointers to ElementLists |
| **Fields of ElementList Object:** | | |
| $l$ | : `CMLock` | // spinlock + type identifier |
| $parent$ | : `ParentStruct` | // parent bucket |
| $count$ | : `Integer` | // number of active elements |
| $keys$ | : `KeyType[]` | // the keys stored in this ElementList |
| $values$ | : `ValType[]` | // the values stored in this ElementList |

The novelty of our algorithm draws, in part, from the lock type embedded in both list types. A `CMLock` couples mutual exclusion information with knowledge about the type of the object in which the lock is embedded. The `CMLock` is used as a spinlock, and releasing the lock can always be achieved by subtracting 1. The possible states appear below:

- $e_{avail}$ – An unlocked `ElementList`.
- $e_{lock}$ – A locked `ElementList`.
- $p_{inner}$ – A `PointerList` at depth $< DEPTH$. Such `PointerLists` are always unlocked.
- $p_{term}$ – An unlocked `PointerList` at $DEPTH$.
- $p_{lock}$ – A locked `PointerList` at $DEPTH$.
- $GARBAGE$ – A locked list undergoing rehashing.

Both `ElementLists` and `PointerLists` begin with a `CMLock` field, enabling us to reference either type from a PointerList: the lock state suffices to indicate the object type.

### 3.1 IHT Behavior

Figure 1 shows seven concurrent operations, which illustrate the key behaviors of the map. Each of these operations is represented by a number in a black circle, and is described below. In the figure, vertical stacks of rectangles indicate `PointerLists`, and horizontal stacks indicate `ElementLists`. White locks are unheld, gray locks are held, and black locks are $GARBAGE$. Gray boxes represent occupied positions in an `ElementList`, and striped boxes indicate the location where an action (lookup, insert, remove) takes place. Curved lines represent atomic stores; dashed lines represent `CAS` operations.

Operation 1 could be an insert, lookup, or remove. It hashes its key, using the hash function of the root `PointerList`, and finds an ElementList. If it can lock that `ElementList`, it can search through the list and decide whether the first element matches the provided key. If it matches, a lookup will return the value, whereas an insert will update it. A remove will remove the key/value pair from the

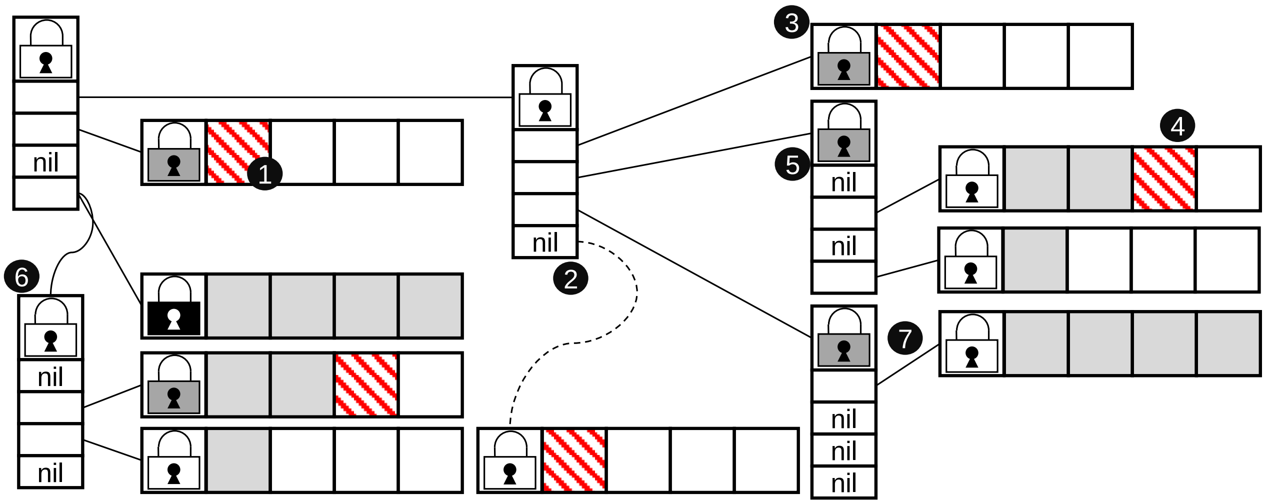


Figure 1: Concurrent hash shape and behavior. INITIAL_BUCKETS and RESIZE_THRESH are set to 4. DEPTH is set to 2.

ElementList. In this case, since the ElementList becomes empty, it will be marked as garbage, and its parent's pointer will be set to **nil**. If the key does not match, a lookup or remove will return an appropriate failure result, but an insert will add a new key/value pair to the ElementList.

Operation 2 encounters an inner PointerList, but when it hashes its key at that list, it finds a **nil** bucket. If it is an insert, it constructs a new ElementList, inserts its element into the list (as a striped block), and then replaces the **nil** bucket with a reference to the new ElementList, via a CAS.

Operation 3 is identical to Operation 1, except the operation is on an ElementList at the maximum depth; It locks this ElementList. In contrast, when Operation 4 reaches a PointerList at the same level, it acquires the PointerList's lock, traverses once more, locates its desired ElementList, but does not acquire another lock, since it already acquired the lock of its parent.

Operation 5 is blocked attempting to acquire the lock on the PointerList owned by Operation 4. This is a necessary consequence of the fixed depth of the map: even if the operation would hash to a different bucket than Operation 4, it cannot run concurrently with Operation 4.

Operation 6 is a common-case resize: an insertion encounters a full ElementList that is not at $DEPTH$. It marks the ElementList as $GARBAGE$, which does not release the lock, creates a new PointerList, and rehashes both the ElementList's elements, and the key/value pair it is adding, into it. Finally, it installs the PointerList by atomically overwriting the reference to the now-defunct ElementList. Note that if Operation 3 encountered a full ElementList, it would operate in the same manner.

Operation 7 is a resize at maximum depth. The figure does not show the completed state, only the point where the ElementList is found to be full. From there, the thread would create a new locked PointerList with more capacity than the old locked PointerList, copy all elements from all child ElementLists of the old PointerList into children of the new PointerList, and then perform its operation on a child of the new list. Finally, it would install the new PointerList, using an atomic store.

### 3.2 Algorithm Correctness and Key Properties

Having sketched the key behaviors of the concurrent map, we now present pseudocode and discuss the key invariants to ensure the correctness of the synchronization mechanisms. To simplify the discussion, we encapsulate the traversal and expansion behaviors of the map in a single function, GetEList (Algorithm 1). Specific insert, lookup, and remove operations are then presented in Algorithm 2

The algorithm for finding an ElementList carries a few simplifications: it does not backoff when encountering a held lock, and sometimes inserts an ElementList or rehashes into a new PointerList during a lookup or remove for a key not present in the map. While these inefficiencies are not present in our implementation, they simplify the discussion below. We also use the shorthand of EToP and PToBiggerP to represent the sequential operations of hashing an ElementList's elements into a new PointerList, and rehashing a PointerList's elements into a larger PointerList, respectively.

As a lock-based algorithm, the correctness arguments are substantially simpler than in the case of non-blocking data structures. In particular, note that an operation never requires more than one lock, and hence deadlock is not possible. The ability to avoid multiple lock acquisitions is a direct consequence of the state transition mentioned above: if a bucket points to an inner PointerList, then the bucket is immutable, and the enclosing object need not be locked

**Algorithm 1:** Concurrent map expanding traversal

```
// Given a map and key, this function returns a locked ElementList within that
// map, representing the sole place where that key may exist
function GetEList ($map, key$)
    $curr \leftarrow map$ // The search path and its top
    $(found, l) \leftarrow (nil, nil)$ // The ElementList and its lock
    loop
        $idx \leftarrow curr.hash(key)\%curr.size$
        $next \leftarrow curr.buckets[idx]$
        // on nil bucket, insert new ElementList; ensure one lock is held
        if $next = nil$ then
            $found \leftarrow$ new ElementList$(0, e_{avail})$
            if $l = nil$ then
                $found.lock \leftarrow e_{lock}$
                $l \leftarrow found$
            if $cas(\&curr.buckets[idx], nil, found)$ then
                return $\langle found, l\rangle$
        // if bucket is a terminal PointerList, lock it and traverse
        else if $next.lock = p_{term}$ then
            if $cas(\&next.lock, p_{term}, p_{lock})$ then
                $l \leftarrow curr \leftarrow next$
        // on inner PointerList, traverse
        else if $next.lock = p_{inner}$ then
            $curr \leftarrow next$
        // if ElementList, we may need to resize
        else if $next.lock = e_{avail}$ then
            // Ensure one lock held and stored in l
            if $l \neq nil \vee cas(\&next.lock, e_{avail}, e_{lock})$ then
                if $l = nil$ then $l \leftarrow next$
                // if bucket not full, return it
                if $next.count < EMAX$ then
                    return $\langle next, l\rangle$
                // if key in bucket, return it
                if $next.bucket.contains(key)$ then
                    return $\langle next, l\rangle$
                // Need to resize. Invalidate the ElementList
                $next.lock \leftarrow GARBAGE$
                // Simple case: no locked PointerList
                if $curr.lock = p_{inner}$ then
                    // Create PointerList from ElementList
                    $p \leftarrow EToP(next, p_{inner})$
                    if $backPath(p, map) = DEPTH$ then
                        $p.lock \leftarrow p_{lock}$
                        $l \leftarrow p$
                    // atomically install new PointerList
                    atomic $curr[idx] \leftarrow p$
                // Tricky case: need to resize locked PointerList
                else
                    // Create larger PointerList from old PointerList
                    $p \leftarrow PToBiggerP(curr)$
                    $p.lock \leftarrow p_{lock}$
                    $curr.lock \leftarrow GARBAGE$
                    $l \leftarrow p$
                    // replace curr with p in curr's parent
                    atomic $curr.parent.setTo(p)$
                    // prepare for next iteration, with p replacing curr
                    $curr \leftarrow p$
```

in order to read that bucket's value. This sort of inductive, speculative object access is inspired by RCU [3], sequence locks [14], and Software Transactional Memory [22].

Another key feature of the algorithm is that (excluding max-depth PointerLists), the lock protecting an ElementList is embedded in the list itself, not in its parent.

**Algorithm 2:** Insert, lookup, and removal operations

```
function Lookup ($map, key$)
    $res \leftarrow NOTFOUND$
    $(elist, lock) \leftarrow$ GetEList$(map, key)$
    for $i \in 0 \ldots elist.count - 1$ do
        if $elist.keys[i] = key$ then
            $res \leftarrow \langle elist.keys[i], elist.values[i]\rangle$
            break
    $lock.release$
    return $res$
function Insert ($map, key, value$)
    $(elist, lock) \leftarrow$ GetEList$(map, key)$
    for $i \in 0 \ldots elist.count - 1$ do
        if $elist.keys[i] = key$ then
            $elist.values[i] \leftarrow value$
            $lock.release$
            return
    $elist.keys[count] \leftarrow key$
    $elist.values[count] \leftarrow value$
    $elist.count \leftarrow elist.count + 1$
    $lock.release$
function Remove ($map, key$)
    $(elist, lock) \leftarrow$ GetEList$(map, key)$
    for $i \in 0 \ldots elist.count - 1$ do
        if $elist.keys[i] = key$ then
            $elist.keys[i] \leftarrow elist.keys[elist.count - 1]$
            $elist.values[i] \leftarrow elist.values[elist.count - 1]$
            $elist.count \leftarrow elist.count - 1$
            break
    $lock.release$
```

This improves locality, since common-case insert and remove operations only perform writes to a single object. Furthermore, since locks protecting references are in the payload ElementLists, instead of the parent PointerLists, we do not require padding of the pointers in the PointerList: they are read-shared in the cache.

Armed with the GetEList function, Algorithm 2 presents lookup, insert, and remove operations. They employ the same pattern: they use GetEList to get a locked ElementList in which their operations can occur. They perform their operation, and then unlock the ElementList.

## 3.3 Iteration

Go requires its map to support iteration, with a caveat: the iteration order is not guaranteed to be the same, even if the map is unchanged from the previous iteration. While this feature was not designed with concurrency in mind, it is an essential enabler for our iteration algorithm.

A sketch of the iteration algorithm appears Algorithm 3. To iterate through the map, we begin by selecting a random bucket in the root PointerList. From that point, we iterate over the entire set of buckets in the root, via a linear traversal. For each bucket, we follow roughly the behavior of GetEList: If the bucket is nil, it is skipped. If it is an ElementList, we lock it and then iterate over its elements. If it is an inner PointerList, we recurse into it, select a random starting point, and repeat the process. Dur-

**Algorithm 3:** Simplified pseudocode for iteration. For clarity of presentation, we do not limit $DEPTH$.

```
// Perform a function ($\lambda$) on every element of the map
function StartIteration ($map, \lambda$)
    // Keep track of passed-over buckets
1   $deferred \leftarrow$ new $set\langle PointerList, Integer\rangle()$
2   EnterPList ($map, \lambda, deferred$) // Recall: the map is a PointerList
3   HandleDeferred ($\lambda, deferred$) // Visit passed-over buckets

// Visit each bucket of a PointerList, starting at a random position
function EnterPList ($plist, \lambda, deferred$)
1   $start \leftarrow random(plist.size)$
2   for $i \in 1 \ldots plist.size$ do
3       ProcessPList
          ($plist, (start + idx)\%plist.size, \lambda, deferred$)

// Within a bucket, decide whether to recurse or process an ElementList
function ProcessPList ($plist, i_p, \lambda, deferred$)
1   if $plist[i_p] =$ nil then
2       return // no data to pass to $\lambda$ from this bucket
3   else if $plist[i_p].lock = p_{inner}$ then
        // Recurse into child PointerList
4       EnterPList ($plist[i_p], \lambda, deferred$)
5   else if $plist[i_p].lock = e_{avail} \wedge cas(\&plist[i_p], e_{avail}, e_{lock})$
      then
        // Iterate over entries in locked ElementList
6       for $i \in 1 \ldots plist[i_p].count$ do
7           $\lambda(plist[i_p].keys[i], plist[i_p].values[i])$
8       $plist[i_p].lock = e_{avail}$
9   else
        // Bucket is garbage, locked, or being resized... defer processing
10      $deferred \leftarrow deferred \cup \langle plist, i_p\rangle$

// Handle PointerList elements that were deferred
function HandleDeferred ($\lambda, deferred$)
1   for $\langle plist, i_p\rangle \in deferred$ do
2       if $plist[i_p].lock = p_{inner}$ then
            // Bucket was rehashed, so recurse into it
3           $deferred \leftarrow deferred - \langle plist, i_p\rangle$
4           EnterPList ($plist[i_p], \lambda, deferred$)
5       else if
          $plist[i_p].lock = e_{avail} \wedge cas(\&plist[i_p], e_{avail}, e_{lock})$
          then
            // Iterate over entries in locked ElementList
6           $deferred \leftarrow deferred - \langle plist, i_p\rangle$
7           for $i \in 1 \ldots plist[i_p].count$ do
8               $\lambda(plist[i_p].keys[i], plist[i_p].values[i])$
9           $plist[i_p].lock = e_{avail}$
10  if $deferred \neq \{\}$ then
11      $optionalBackoff()$
12      goto 1
```

ing the recursion, if we encounter a terminal PointerList, we lock it, and then recurse into it, taking care not to lock its ElementLists. To reduce convoying, we maintain per-iteration lists of "busy" objects. Whenever an iteration encounters a locked ElementList or PointerList, we save the address of its parent's reference to it, and defer visiting it until later in the execution.

There are several benefits to this algorithm. Only one lock is held at a time, and hence iteration cannot deadlock with other iterations, or with concurrent lookup/insert/remove operations. Second, Go's requirement of an unpredictable iteration order is enhanced: rather than pick a random starting point in a single flat array, we randomize at the level of each PointerList. Third, the fact that randomization is built into the language provides a guard against convoy effects: iterators do not start at the same point, and hence are unlikely to visit ElementLists in the same order. The guaranteed variation in order also allows us to maintain the busy object list, without presenting unexpected behavior to the programmer. In essence, Go's desire to prevent programmers from relying on implementation artifacts transforms into a language-level semantics that enables concurrent iteration with minimal waiting.

# 4. Implementation

The IHT provides the same API as the sequential map. As we shall see in Section 5, this does not hold for library-based concurrent maps for Go. In this section, we describe the IHT implementation, and discuss the guarantees it provides.

## 4.1 Compiler Integration and Transformations

As discussed in Section 2.2, the default Go map implementation is tightly coupled with the compiler and runtime. To provide the same syntax for the IHT, it must be implemented by the compiler as well. However, the existing compiler infrastructure is insufficient: a mapaccess does not return a value, but instead returns a live, internal pointer into the map. While we can acquire the lock protecting the referenced data before mapaccess returns, it is unreasonable to delegate lock release to the programmer.

When the compiler generates a mapaccess call, the returned pointer is live for a short duration. The next instruction dereferences the pointer, either to memcpy the (large) value to memory, or to copy the (machine word-sized or smaller) value to a register. Immediately thereafter, the pointer is not live; consequently, the lock can be released. In our lock implementation, the same function releases the lock, regardless of whether it protects an ElementList or a PointerList. However, the lock may be hard to locate, if the returned value is in an ElementList reached from a maximum-depth PointerList.

Our solution is for mapaccess to return a reference to the lock, as well as a reference to the value, This increases the coupling between the IHT and the compiler, but saves overhead, as the tree need not be re-traversed to find the lock. An additional complication is that multiple calls to mapaccess could occur in a single statement (e.g., a = m[b] + m[c]). The current Go implementation performs the accesses sequentially, and thus we only hold one lock at a time. In the interests of remaining future-proof, we observe that the keys could hash to the same ElementList. If the Go compiler were to allow both pointers to be live simultaneously, in addition to needing deadlock avoidance we would need to make our spinlocks reentrant.

## 4.2 The sync.Interlocked Interface

The above mechanism provides atomicity for individual map accesses, but not atomicity for multiple statements accessing the same map element. For complex individual statements, we could automatically defer all lock releases until the end of the statement, but doing so would introduce the possibility of deadlock when multiple map accesses, with different keys, are performed in a single statement.

Instead, we provide a means for exposing the map's locks to the programmer. The `sync.Interlocked(map, key)` library function acquires the lock associated with a particular key in a particular map, and `sync.Release(map)` releases that key's lock. Between the calls, a thread can make multiple accesses to a map element, without intermediate results being visible to other goroutines. Exposing these operations as functions, instead of as a keyword and lexical scope, enables the Go's idiom in which the `defer` keyword can be used to ensure that locks are released upon error.

When `sync.Interlocked` is passed a key not present in the map, room for the key/value pair is created in the map. We extended the runtime to track uses of the pair; if an automatically-created pair is never assigned, it is deleted during `sync.Release`. Similarly, if a key is deleted from the map during `sync.Interlocked` execution, the space is not reclaimed until `sync.Release`.

Each goroutine has a private context, which is visible only to the runtime. This context can be used in scenarios where runtime features require thread-local storage. In our implementation, interlocked access exploits this space to optimize map accesses: in the IHT's `mapaccess`, `mapassign`, and `mapdelete` functions, as well as the lock release functions we insert during compilation, we check if an interlocked operation over the map/key combination is active. If so, all traversal required to locate the key/value pair can be elided, as can any locking/unlocking.

Go includes run-time facilities for detecting races and dangerous behaviors. In the case of `sync.Interlocked`, we track its use during execution, and ensure that multiple keys from the same map are never simultaneously interlocked by one goroutine. Since the mapping of keys to `ElementLists` is invisible to the programmer, this ensures that deadlocks will not occur when the run-time choice of hash function leads to two goroutines issuing conflicting interlocking accesses while holding locks. (Note that when atomicity can be ensured through other means, the programmer can use a new goroutine to concurrently access other keys in the map. If the goroutine conflicts with the parent, it will block until the parent's interlocked execution completes.) Leveraging goroutine-local storage, ensures there is little overhead for dynamic checks to detect and prevent acquisition of multiple locks during interlocked execution. We do, however, allow overlapping interlocked accesses to *different* maps, since it ought to be possible for programmers desiring this ability to design a safe locking order.

We leave as future work deadlock-free multi-key interlocked execution. Transactional Memory [26] is one approach, though subject to hardware capacity constraints and limits on the behavior of code during interlocked execution. Another option is to provide a multi-key version of `sync.Interlocked`, and allow the runtime to infer a safe locking order after it hashes the keys it is passed. While promising, this would still leave open the question of how to support critical sections for which the set of keys to access cannot be determined before the critical section begins.

### 4.3 Iteration

Existing approaches to iteration in a concurrent collection take one of two approaches. On the one hand, an atomic snapshot provides a copy of the collection, such that there existed a point in time when the contents of the collection were identical to those presented in the snapshot. On the other hand, non-atomic iteration provides weaker guarantees, but is typically less costly. For example, in Java, an iteration through a concurrent collection is not serializable: it can "miss" items that were concurrently added by other threads.

Despite the appeal of atomic snapshots, we deemed them impractical for the IHT. If we were to provide snapshots without copying, then an iteration would continually grow its lock set, until it held locks over the entire map. Such a technique would strangle concurrency, and forbid concurrent iteration. Indeed, since Go specifies iteration returns keys in a random order, we would need to eagerly serialize all iterations, since concurrent iterations would otherwise start at different parts of the map and then deadlock. If, instead, we created a snapshot by copying all map contents to a temporary location, we would incur space overhead proportional to the number of concurrent iterations. This could cause out-of-memory errors for large maps. Furthermore, a copy-based atomic snapshot offers weak guarantees to programmers: a key in the snapshot may no longer be present in the map, necessitating additional error handling.

Our approach is more like that in Java. Our iterator holds one lock at a time, and generates values from one locked subtree at a time. Since resizing is localized to a subtree of a PointerList, we can safely release one lock before acquiring the next: once an element is visited during iteration, it cannot be moved such that the iterator encounters it again. This simplifies reasoning about correctness: since only one lock is held by an iterator at any time, two iterations cannot deadlock. At the same time, while some key/value pairs can be missed, every pair returned by the iterator is guaranteed to be present in the map at the time it is returned. Furthermore, the pair is present in a locked subtree, owned by the thread performing the iteration. Consequently, the iterating thread can safely modify or remove the pair without racing with concurrent map operations that attempt to use the same key.

## 5. Performance Evaluation

We evaluate the IHT through a series of microbenchmarks, designed to stress the behavior of the varying aspects of the map. In this section, we compare five different map implementations:

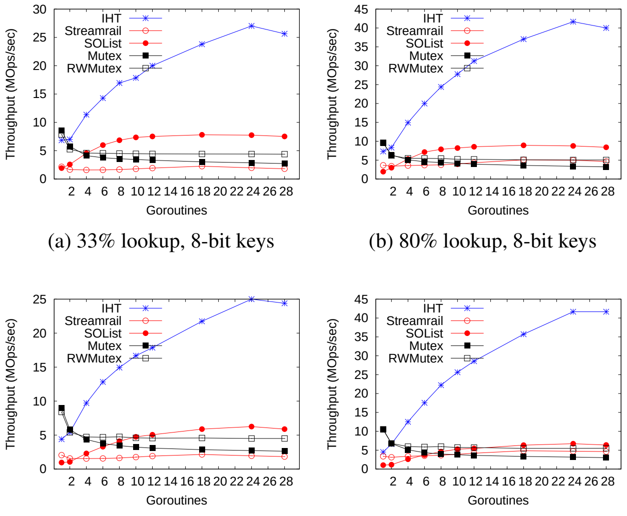
- IHT – Our Interlocked Hash Table
- SOList – The nonblocking split-ordered list [21].
- Streamrail – A lock-based map, implemented as a fixed-size array of RWMutex-protected Go maps.
- RWMutex – A RWMutex-protected default Go map.
- Mutex – A Mutex-protected default Go map.

The SOList and Streamrail source codes are publicly available through the go package manager, under the names `gotomic` [27] and `streamrail` [10].

Of these implementations, SOList and Streamrail are library-based. Since neither is integrated into the Go runtime, each must provide its own hashing strategy. SOList allows arbitrary key types, but the programmer must provide an appropriate hash function. Streamrail requires keys to be strings, and values to be `interface{}` (Go's opaque type), and then uses its own hash function. We used the default configuration for each map: in Streamrail, there are 32 buckets in the top-level map. In SOList, there is no bound on the maximum depth of the directory tree that indexes into the underlying lock-free list.

We present results on a machine with two Xeon X5650 CPUs (6 cores/12 threads per CPU), 12 GB of RAM, Ubuntu Linux 16.04.1 (kernel version 4.4.0), and the Go 1.6 compiler. We also ran these experiments on a single-chip Core i7-4770, and observed the same relationship between the performance of the different systems.

We configured the IHT with 8-entry `ElementLists` and variable `PointerList` sizes: the root `PointerList` was 32 elements, with a doubling of `PointerList` capacity at each subsequent level. With this configuration, and a default $DEPTH$ of 4, the IHT can grow to hold up to 53M elements without resizing a last-level `PointerList`. When configured to store 64-bit integer key and 64-bit integer value pairs, the IHT grew to require roughly 2GB of RAM when 10M random elements were inserted, whereas the default Go map consumed 600MB to hold the same data. We ran the same test with Streamrail's Concurrent Map, which is backed by 32 Go maps. Each of Streamrail's maps holds fewer elements, but must be resized independently, resulting in 1.1GB of memory consumption. The SOList, which uses a linked list as its underlying data structure, requires many small allocations. Even though it is free of internal fragmentation, the cost of individual list nodes results in a total space overhead of 2.6GB. When elements are removed from the SOList, marker nodes in the list, and all nodes of the directory, must remain. However, nodes holding data can be reclaimed. Similarly, in the IHT, removals result in `ElementLists` being reclaimed, but not `PointerLists`. The IHT shrinks to about 500MB when filled with 10M elements and then emptied. The Go map, and the set of maps in Streamrail, never shrink.



(a) 33% lookup, 8-bit keys

(b) 80% lookup, 8-bit keys

(c) 33% lookup, 16-bit keys

(d) 80% lookup, 16-bit keys

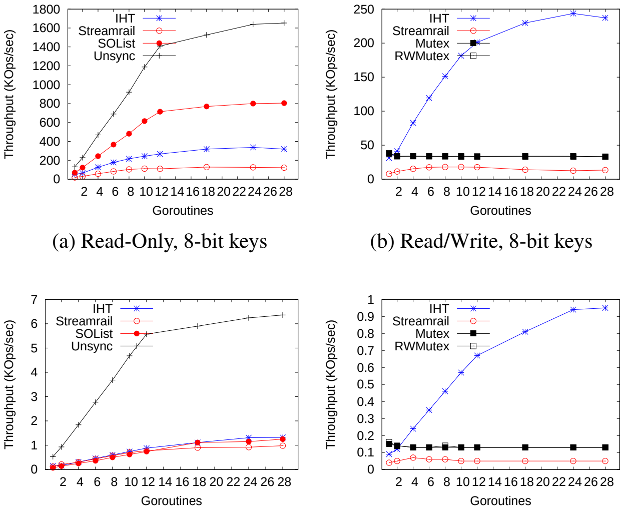
Figure 2: Integer set microbenchmark

### 5.1 Elemental Access Performance

Our first set of experiments consider IHT performance on traditional microbenchmarks. In these tests, we configure each map to store 64-bit integer keys. The values stored in the maps are not important, so we use values of type `byte`. In the first set of tests, the key range is limited to $0 \ldots 255$; In the second, the range increases to $0 \ldots 64K$. We consider two mixes of operations: 80/10/10, in which 80% of operations are lookups and the remainder split evenly between insertions and removals, and 33/33/33, in which an equal number of lookup, insert, and remove operations are attempted. Each data point is the average of 10 trials. Results appear in Figure 2.

First, we observe that IHT has more latency than the lock-protected default maps. This result is expected. Go's Mutex implementation is efficient, and its map is highly optimized for sequential code. There is only one level of indirection in the common case, all hashing is performed in the runtime, and the use of a single flat array to store all data results in good locality. While the IHT also has good locality and an efficient lock implementation, it has more indirection: in the 8-bit case, some ElementLists are reached directly from the root PointerList, but hash collisions cause other elements to have two PointerLists before the ElementList is reached. For the 16-bit case, the cost goes up to three PointerLists for some keys, or four levels of indirection.

The IHT outperforms the SOList and Streamrail at one goroutine. In the case of SOList, the implementation keeps the depth of the directory low, but each key/value pair is in its own list node, leading to little locality. In Streamrail, the overhead of string types for the keys, and one additional level of indirection for the sub-maps, create less latency than SOList, but more than IHT.



(a) Read-Only, 8-bit keys

(b) Read/Write, 8-bit keys

(c) Read-Only, 16-bit keys

(d) Read/Write, 16-bit keys

Figure 3: Iteration microbenchmark

The IHT quickly scales past the default Go maps. At 2 goroutines, the IHT matches the 2-goroutine performance of the Go map, and at 4 goroutines, the IHT outperforms the Go map's peak performance. It then scales up to the full size of the machine (24 hardware threads), with a slight bend at 12 goroutines (where simultaneous multithreading (SMT) [23] begins). While the RWMutex provides better performance than a simple Mutex, cache contention for the lock is a significant impediment to scalability even when 80% of operations are read-only.

Neither the SOList nor Streamrail scales as well as the IHT. This is even true in SOList with 80% lookups, where SOList lookup operations do not use any CAS instructions. We identified three main causes for the superior scaling of IHT. First, the Go SOList implementation relies on shared counters to manage the maximum depth of its directory, and these counters can become a bottleneck, especially on multi-chip machines. Second, Streamrail's use of a lock table, instead of locks embedded with data, means that concurrent lock acquisitions can cause cache invalidations in concurrent hardware threads. Lastly, the SOList provides less locality than IHT, since each key/value pair is its own list element. With 16-bit keys, the cost is especially great, since the key range causes an increase in the depth of the SOList directory, and directory nodes also have little locality.

Additional experiment revealed that the scalability trends in Figure 2 amplify as the key range increases. We also confirmed that the decreased performance after 24 goroutines is due to interactions between our spinlocks and the Go scheduler. Goroutines are not preemptive in the same way as OS kernel threads, and yield statements must be employed when spin waiting. With yielding, performance remains steady under preemption.

## 5.2 Iteration Performance

We next consider workloads with 100% iteration. The map is pre-filled with either 256 or 64K elements, but now we consider two iteration approaches: read-only, in which no operation changes a value or inserts/removes elements, and read-write, in which all operations change the value of each key they encounter. We count a complete iteration through the data structure as a single operation.

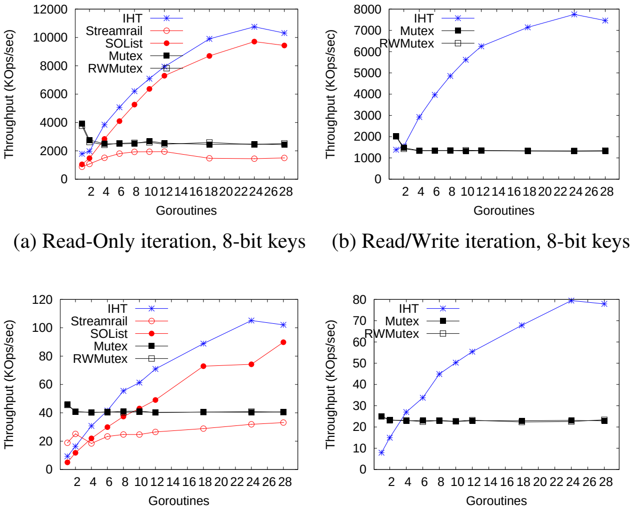
For read-only iteration, no concurrency control is required. In this case, Go allows concurrent access to the default map. Thus in Figure 3, we compare IHT, Streamrail, SOList, and an unsynchronized map. When there is no Mutex to acquire, the unsynchronized map scales perfectly up to 12 goroutines, and then continues to scale, at a slower rate, as SMT results in hardware threads sharing cores.

For small maps, the SOList also outperforms the IHT under read-only iteration. The SOList's nonblocking implementation can avoid any CAS instructions during an iteration, and the lack of insert and remove operations avoids the need for any accesses to the shared counters used by the SOList to manage directory height. Thus the SOList enjoys disjoint-access parallelism [13]. In contrast, the IHT is unaware of the read-only nature of the workload, because it uses the Go map interface (in which reads and updates use the same code). Thus it acquires locks over subtrees of elements. With 16-bit keys, however, the cost of locking in IHT is roughly equal to the indirection overheads and lack of locality in SOList, and the two maps perform equivalently.

Both SOList and IHT outperform Streamrail. For a single iteration operation, Streamrail creates one goroutine per bucket, and then each of the 32 goroutines executes in parallel. While the locks protecting the buckets are acquired for reading, and hence goroutines can make progress, each tick along the X axis corresponds to an additional 32 goroutines launching and coordinating with their parent, each time the parent performs an iteration. These goroutines communicate with the parent goroutine via channels, and the aggregate overhead is greater than the gain in concurrency.

Unfortunately, SOList does not support mutating iteration, because the nonblocking implementation cannot guarantee that, upon returning a key/value pair, that pair remains in the map. Streamrail provides mutating iteration, so long as there are not concurrent insert/remove/lookup operations. As with read-only iteration, the heavy use of goroutines creates high latency. In addition, the per-bucket locks must now be acquired exclusively, rather than in read mode. With only 32 buckets, and 32 goroutines per thread, most goroutines are blocked at any time during the 24-thread workload.

Comparison of IHT to mutating iteration in the default Go map shows equivalent performance at 2 threads, with IHT outperforming the peak Go map performance at 4 threads and above. For the workload with 8-bit keys, our convoy avoidance plays an essential role. As concurrency increases, goroutines scatter through the IHT as they choose random



(a) Read-Only iteration, 8-bit keys (b) Read/Write iteration, 8-bit keys

(c) Read-Only iteration, 16-bit keys (d) Read/Write iteration, 16-bit keys

Figure 4: Mixed workload with 2.5% iteration, and remaining operations split between lookups, inserts, and removes.

starting points. However, with such a shallow tree, goroutines quickly collide as concurrency passes 8 goroutines. By deferring processing of locked ElementLists, and revisiting them later, the average iteration avoids spin-waiting entirely.

We conclude our discussion of iteration by observing that Streamrail, SOList, and IHT all provide weak iteration guarantees. In SOList, one sub-map is locked at a time, and thus each element returned by the iterator is present in the map, but the set of returned elements is not an atomic snapshot. In SOList, one element is returned at a time, without preventing concurrent mutations throughout the remainder of the underlying list. SOList also does not randomize the order in which pairs are returned from the iterator. IHT's behavior is like Streamrail: every element returned by the iterator is present in the map for the duration of that iteration, but the entire set of returned items is not an atomic snapshot.

### 5.3 Combined Performance

Lastly, we consider workloads in which iteration is concurrent with inserts, lookups, and removals of key/value pairs. As above, we count each iteration across the map as a single operation. However, threads now have a 2.5% chance of performing an iteration, with the remaining operations split evenly among inserts, lookups, and removes. We consider read-only iteration and read-write iteration. In the case of read-only iteration, the possibility of concurrent accesses necessitates that the default Go map be protected by a Mutex. Again, SOList does not allow read/write iteration, and is not presented. Additionally, Streamrail deadlocks for the read/write workload, because of a race when the number of elements in a submap changes after an iterator creates the channels used by its spawned goroutines.

The results in Figure 4 represent a composition of the prior two sets of experiments. In both SOList and IHT, neither iteration nor elemental operations impedes the expected performance of the other, and both scale well. These experiments also represent the first time that SOList's peak performance is greater than the default Go map's peak. In the 8-bit test with read-only iteration, where SOList's iteration greatly outperformed IHT before, we now see equivalent performance, tipping slightly in favor of IHT. In the 16-bit case, where SOList and IHT had equivalent iteration performance, the higher performance of IHT's elemental accesses gives it a slight edge. As in Figure 3, adding goroutines does not lead to contention in the IHT, despite long-running iteration. The invariant that operations only hold one lock at a time, coupled with randomized start points for iteration, result in steady scaling. In contrast, read-only iterations, which reduce the rate at which shared counters are modified, reduce the significance of a bottleneck in SOList, and help it to recover performance relative to its scaling in Figure 2.

## 6. Conclusions and Future Work

In this paper, we introduced the Interlocked Hash Table (IHT), a highly concurrent lock-based map designed specifically for the Go programming language. The IHT employs a speculative traversal of a fixed-max-depth tree of intermediate nodes, which enables it to acquire exactly one lock per insert/remove/update/lookup operation. By co-locating locks with data, the net result is negligible contention even when 24 hardware threads are performing simultaneous random accesses to a small map. The scalability of the IHT, and its optimized implementation inside of the Go compiler and runtime, enable it to outperform all known alternatives in Go, to include lock-free and lock-based open-source maps. In microbenchmarks, we observed performance up to 7× the performance of the default map at high thread counts, and a peak throughput more than $4 times$ the peak achieved speedup by the default (typically at one thread).

The IHT exploits Go's randomized iteration requirement. This allows concurrent IHT iterators to begin at random locations within the data structure, and to delay processing of any locked regions they encounter during iteration. Our experiments show an absence of convoying effects, which enables both read-only and read/write iteration to scale to the full size of the machine.

Through a minor addition to the `sync` package, we provide support for large critical sections over a single map element. As future work, we plan to extend this capability to multi-element critical sections. Because the hash functions within the IHT vary from one execution to the next, the programmer cannot infer a safe locking order to prevent deadlock cycles. However, the runtime can determine this information, and for critical sections over a set of map locations known at the beginning of the critical section, we believe it possible to guarantee atomicity and deadlock-freedom.

## References

[1] A. Braginsky and E. Petrank. A Lock-Free B+tree. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, Pittsburgh, PA, June 2012.

[2] T. Brown, F. Ellen, and E. Ruppert. Pragmatic Primitives for Non-blocking Data Structures. In *Proceedings of the 32nd ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, July 2013.

[3] M. Desnoyers, P. McKenney, A. Stern, M. Dagenais, and J. Walpole. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, 2012.

[4] V. Gramoli. More Than You Ever Wanted to Know about Synchronization. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming*, San Francisco, CA, Feb. 2015.

[5] T. Harris, K. Fraser, and I. Pratt. A Practical Multi-word Compare-and-Swap Operation. In *Proceedings of the 16th International Conference on Distributed Computing*, Toulouse, France, Oct. 2002.

[6] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit. A Lazy Concurrent List-Based Set Algorithm. In *Proceedings of the 9th international conference on Principles of Distributed Systems*, Pisa, Italy, Dec. 2006.

[7] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Structures. In *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, Mar. 1990.

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

[9] M. P. Herlihy and J. M. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[10] IronSource Neon. A thread-safe concurrent map for go, 2016. https://github.com/streamrail/concurrent-map/.

[11] ISO/IEC JTC 1/SC 22/WG 21. Technical Specification for C++ Extensions for Transactional Memory, May 2015.

[12] ISO/IEC TS 19570:2015. Technical Specification for C++ Extensions for Parallelism, 2015.

[13] A. Israeli and L. Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, 1994.

[14] C. Lameter. Effective Synchronization on Linux/NUMA Systems. In *Proceedings of the May 2005 Gelato Federation Meeting*, San Jose, CA, May 2005.

[15] D. Lea. Abstraction Failures in Concurrent Programming (Keynote Address). In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, Pittsburgh, PA, June 2012.

[16] Y. Liu, K. Zhang, and M. Spear. Dynamic-Sized Nonblocking Hash Tables. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing*, Paris, France, July 2014.

[17] I. Lotan and N. Shavit. Skiplist-Based Concurrent Priority Queues. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, Cancun, Mexico, May 2000.

[18] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the 15th ACM Symposium on Parallel Algorithms and Architectures*, San Diego, CA, June 2003.

[19] M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, Winnipeg, Manitoba, Canada, Aug. 2002.

[20] M. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 2009.

[21] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM*, 53(3):379–405, 2006.

[22] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, Ottawa, ON, Canada, Aug. 1995.

[23] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[24] R. von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct. 2003.

[25] X. Yang, D. Frampton, S. Blackburn, and A. Hosking. Barriers Reconsidered, Friendlier Still! In *Proceedings of the International Symposium on Memory Management*, Beijing, China, June 2012.

[26] R. Yoo, C. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High Performance Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, CO, Nov. 2013.

[27] Zond. Non blocking data structures for Go, 2016. https://github.com/zond/gotomic/.