2017 26th International Conference on Parallel Architectures and Compilation Techniques

Redesigning Go's Built-In Map to Support Concurrent Operations

Louis Jenkins
Bloomsburg University
lpj11535@huskies.bloomu.edu

Tingzhe Zhou
Lehigh University
tiz214@lehigh.edu

Michael Spear
Lehigh University
spear@lehigh.edu

***Abstract*—The Go language lacks built-in data structures that allow fine-grained concurrent access. In particular, its `map` data type, one of only two generic collections in Go, limits concurrency to the case where all operations are read-only; any mutation (insert, update, or remove) requires exclusive access to the entire map. The tight integration of this `map` into the Go language and runtime precludes its replacement with known scalable map implementations.**

This paper introduces the Interlocked Hash Table (IHT). The IHT is the result of language-driven data structure design: it requires minimal changes to the Go `map` API, supports the full range of operations available on the sequential Go `map`, and provides a path for the language to evolve to become more amenable to scalable computation over shared data structures. The IHT employs a novel optimistic locking protocol to avoid the risk of deadlock, and allows large critical sections that access a single IHT element, and can easily support multi-key atomic operations. These features come at the cost of relaxed, though still straightforward, iteration semantics. In experimentation in both Java and Go, the IHT performs well, reaching up to 7× the performance of the state of the art in Go at 24 threads. In Java, the IHT performs on par with the best Java maps in the research literature, while providing iteration and other features absent from other maps.

I. Introduction

Safe, scalable, general-purpose concurrency support is an essential feature of modern programming languages [1]. This support typically includes a precisely-defined language-level memory model [2], [3], first-class support for threads and locks [4], and a standard library of highly concurrent data structures that facilitate the coordination of threads via shared memory [5], [6]. It may also include support for specific patterns, such as Communicating Sequential Processes (CSP) [7], Actors [8], and Transactional Memory [9].

As a modern programming language, Go offers many features that support concurrent programming. Its thread abstraction is a multi-CPU variant of Capriccio [10] that can multiplex hundreds of thousands of lightweight *goroutines* onto all the physical cores of a machine. Its *channel* abstraction provides first-class support for CSP, and scales to thousands of threads. Go also allows shared-memory synchronization, through the `sync` package's `Mutex`, `RWMutex`, and atomic versions of primitive data types.

Unfortunately, Go does not provide scalable concurrent data structures. In many languages, such a deficiency could be remedied through updates to a library, (for example, the `java.util.concurrent` package is continually evolving [5]). However, Go does not allow user-defined generic collections. Go provides two built-in, generic, sequential collections, the `map` and `slice` (dynamic array). Both are tightly integrated into the Go compiler and runtime. To create a scalable collection *in a library*, a programmer would need to use Go's opaque `interface{}` type, which incurs an extra level of indirection, and would not be able to employ features that are exclusively available to Go's sequential `map` and `slice`. 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 the key), but this mechanism is not available to library code.

Nonetheless, Go is an appealing language for concurrent data structure design. It is garbage collected, which makes it easier for concurrent data structures to employ speculation: the memory accessed by a thread executing a doomed but incomplete speculation cannot be recycled while any thread retains a reference to it. Go is type-safe, and provides reflection and auto-boxing. Like C++, Go allows fine-grained control of the placement of hardware memory fences, and can avoid indirection by storing data, rather than references, in its built-in collections. Go also allows direct pointer access, through its `unsafe` package.

Because, Go's `map` and `slice` are tightly coupled with the compiler and run-time system, it is not straightforward to use existing concurrent data structures efficiently. In the case of the `map`, the compiler selects a data layout based on the size of the key and value types, and the run-time interface to the `map` returns a pointer to a map element, rather than performing an insert or lookup directly. Go's `range` keyword, used for iteration, must produce a randomized starting position. These properties prevent the use of state-of-the-art map implementations (e.g., those from JSR-166): nonblocking Java maps cannot support pointer-based access to map elements, and even the blocking Java maps do not allow iteration to begin from a random starting point.

Our solution is to create a new concurrent map data structure specifically for Go. Our Interlocked Hash Table (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

978-1-5090-6764-0/17 $31.00 © 2017 IEEE
DOI 10.1109/PACT.2017.45

through which programmers can write large critical sections over a single map element, or even multiple elements. In return for these features, the IHT offers weaker iteration guarantees than a sequential map: iteration will never return an item twice, or miss an item that was present for the duration of iteration, but it is not linearizable [11].

II. Background and Related Work

We briefly discuss the features that most significantly affect high-performance concurrent data structure design, and also discuss the implementation of Go's sequential map.

A. Concurrent Data Structure Design

In order to scale, concurrent data structures must avoid interaction among threads. For lock-based algorithms, using fine-grained locks prevents 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 [12], 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 [13].

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 multi-word atomic operations (e.g., LLX/SCX [14] or multi-word CAS [15], [16]). 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.

Techniques like optimistic synchronization are often more important than nonblocking guarantees. Consider the lazy list [17]: it provides nonblocking lookup operations, but uses locks when inserting and removing elements. It avoids lock acquisitions during traversal; validates 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 leverages 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 [18] 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 [20] lazily rehashes elements upon overflow of a bucket.

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

Concurrent data structures are usually designed to ensure linearizability [11]. 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 [21].

Linearizable iteration is particularly challenging. The most straightforward approach, atomic snapshots, are complex and may not scale [22]. Worse, programmers wishing to perform modifications during iteration are poorly served by snapshots, which can return a stale copy of a large data structure. Many concurrent data structures relax their iterator semantics. In JSR-166, iteration over a priority queue may not return elements in priority order, and iteration over a queue may "miss" elements added to 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 [23], [13], [21]. Others allow multi-location atomic operations, but only when the locations are known in advance [24].

B. The Go Map: Implementation and Interface

A simplified description of the interface to the Go map appears in Table I. 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 common in nonblocking data structures. It allows keys with sizes greater than a machine word, and hence techniques that rely on atomic reads of keys, such as the lazy list's wait-free

Listing 1: IHT types. The `ParentStruct` type encapsulates information about the bucket within the parent `PointerList` that references any `BaseObject`.

```
Fields of BaseObject:
  l          : CMLock           // spinlock + type identifier
  parent     : ParentStruct     // parent bucket
Fields of PointerList (extends BaseObject):
  size       : Integer          // size of buckets array
  hashkey    : Integer          // seed for hash function
  buckets    : BaseObject[]     // ElementLists or PointerLists
Fields of ElementList (extends BaseObject):
  count      : Integer          // number of active elements
  keys       : KeyType[]        // the keys stored in this ElementList
  values     : ValType[]        // the values stored in this ElementList
```

`contains` operation, are not available. 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 nonblocking techniques, because the linearization point of the read occurs after the response of the lookup function. In a naïve concurrent implementation of this interface, the internal pointer returned by `mapaccess` could be concurrently mutated, causing a race.

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.

III. Design of a Concurrent, Lock-based Map

We now present pseudocode and describe the behavior of the IHT. Listing 1 introduces the two main data types used by the IHT: `ElementLists` are used to hold key/value pairs, and `PointerLists` form a wide tree by holding references to either `ElementLists` or other `PointerLists` in their `buckets` arrays.

The IHT is defined by its root `PointerList` and three constants: $DEPTH$, the maximum depth of the tree; $EMAX$, the maximum number of elements in any `ElementList`; and $PINIT$, the size of the `buckets` array of the root `PointerList`. Child `PointerLists`' `buckets` arrays hold twice as many elements as their parents' `buckets` arrays, except at depth $DEPTH$, where the `PointerList` can be doubled as many times as needed. Each `PointerList`'s hash function uses a different seed, to avoid collisions when elements are rehashed.

The IHT can grow, but not shrink. Thus once a bucket references a `PointerList`, it will never again be **nil** or reference an `ElementList`. In addition, excluding the deepest level of the tree, once a bucket references a `PointerList`, it is immutable. At the deepest level, a bucket may reference a larger `PointerList` in the future.

In Java, the `CMLock` is simply a spinlock, and runtime type information is used to distinguish between `ElementList`, `PointerList`, and depth-$DEPTH$ `PointerList` references. In Go, `ElementLists` and `PointerLists` are organized so that their first word is a `CMLock` field. 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 spin lock, 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.

This enables us to use opaque types in a `PointerList` in Go: the lock state suffices to indicate the object type.

A. IHT Behavior

Figure 1 shows seven concurrent operations, which illustrate the key behaviors of the IHT. 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 `ElementList`. 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 its key hashes to a **nil** `ElementList`. If it is an insert, it constructs a new `ElementList`, inserts its element into the list (as a striped block), and then inserts its `ElementList` via a `CAS`.

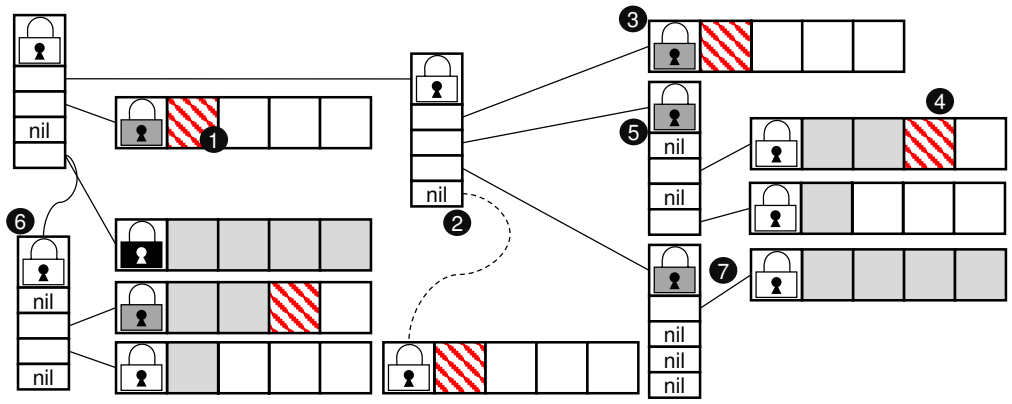


Figure 1: Concurrent hash shape and behavior. PINIT and EMAX are set to 4. DEPTH is set to 2. To avoid clutter in the figure, PointerLists at levels 1 and 2 are shown with size $PINIT$. In Section V, these PointerLists will have a size of $2^n \times$PINIT

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. Note that it does not also lock an ElementList: ElementLists at $DEPTH + 1$ are never locked.

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. Next, the thread would create a new locked PointerList with greater capacity, rehash all elements from all child ElementLists of the old PointerList into the new PointerList, perform its operation on the new PointerList, and then install the new PointerList, using an atomic store.

B. 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 for a Java-like API appear in Algorithm 2.

Algorithm 2 has the following 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.

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 in order to read that bucket's value. This sort of inductive, speculative object access is inspired by RCU [26], sequence locks [27], and Software Transactional Memory [28].

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. 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.

Algorithm 1: Concurrent map expanding traversal

```
// Given a map and key, this function returns a tuple consisting of a reference
// to a lock and a reference to an ElementList. The ElementList represents
// the sole place in the map where that key might exist.
function GetEList (map, key)
1   curr ← map // Start at the root PointerList
2   (found, l) ← (nil, nil) // The found ElementList and its lock
3   loop
4       idx ← curr.hash(key)%curr.size
5       next ← curr.buckets[idx]
        // on nil bucket, insert new ElementList; ensure one lock is held
6       if next = nil then
7           found ← new ElementList(0, e_avail)
8           if l = nil then
9               found.lock ← e_lock
10              l ← found
11          if cas(&curr.buckets[idx], nil, found) then
12              return ⟨found, l⟩
        // if bucket is a terminal PointerList, lock it and traverse
13      else if next.lock = p_term then
14          if cas(&next.lock, p_term, p_lock) then
15              l ← curr ← next
        // on inner PointerList, traverse
16      else if next.lock = p_inner then
17          curr ← next
        // if ElementList, we may need to resize
18      else if next.lock = e_avail then
            // Ensure one lock held and stored in l
19          if l ≠ nil ∨ cas(&next.lock, e_avail, e_lock) then
20              if l = nil then l ← next
                // if bucket not full, return it
21              if next.count < EMAX then
22                  return ⟨next, l⟩
                // if key in bucket, return it
23              if next.bucket.contains(key) then
24                  return ⟨next, l⟩
                // Need to resize. Invalidate the ElementList
25              next.lock ← GARBAGE
                // Simple case: no locked PointerList
26              if curr.lock = p_inner then
                    // Create PointerList from ElementList
27                  p ← EToP(next, p_inner)
28                  if backPath(p, map) = DEPTH then
29                      p.lock ← p_lock
30                      l ← p
                    // atomically install new PointerList
31                  atomic curr[idx] ← p
                // Tricky case: need to resize locked PointerList
32              else
                    // Create larger PointerList from old PointerList
33                  p ← PToBiggerP(curr)
34                  p.lock ← p_lock
35                  curr.lock ← GARBAGE
36                  l ← p
                    // replace curr with p in curr's parent
37                  atomic curr.parent.setTo(p)
                    // prepare for next iteration, with p replacing curr
38                  curr ← p
```

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`.

Algorithm 2: Insert, lookup, and removal operations

```
function Lookup (map, key)
1   res ← NOTFOUND
2   (elist, lock) ← GetEList(map, key)
3   for i ∈ 0 . . . elist.count − 1 do
4       if elist.keys[i] = key then
5           res ← ⟨elist.keys[i], elist.values[i]⟩
6           break
7   lock.release
8   return res

function Insert (map, key, value)
1   (elist, lock) ← GetEList(map, key)
2   for i ∈ 0 . . . elist.count − 1 do
3       if elist.keys[i] = key then
4           elist.values[i] ← value
5           lock.release
6           return
7   elist.keys[count] ← key
8   elist.values[count] ← value
9   elist.count ← elist.count + 1
10  lock.release

function Remove (map, key)
1   (elist, lock) ← GetEList(map, key)
2   for i ∈ 0 . . . elist.count − 1 do
3       if elist.keys[i] = key then
4           elist.keys[i] ← elist.keys[elist.count − 1]
5           elist.values[i] ← elist.values[elist.count − 1]
6           elist.count ← elist.count − 1
7           break
8   lock.release
```

C. Iteration

Go's map supports iteration, but the iteration order is not guaranteed to be the same, even if the map is unchanged since 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 starting bucket at random, and repeat the process. During 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 participate in deadlocks. Second, Go's requirement of an unpredictable iteration order is enhanced: we randomize at the level of each `PointerList`. Third, we leverage randomized iteration

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

```
// Perform a function (λ) on every element of the map
function StartIteration (map, λ)
    // Keep track of passed-over buckets
1   deferred ← new set⟨PointerList, Integer⟩()
2   EnterPList (map, λ, deferred) // Recall: the map is a PointerList
3   HandleDeferred (λ, deferred) // Visit passed-over buckets

// Visit each bucket of a PointerList, starting at a random position
function EnterPList (plist, λ, deferred)
1   start ← random(plist.size)
2   for i ∈ 1 ... plist.size do
3       ProcessPList
          (plist, (start + idx)%plist.size, λ, deferred)

// Within a bucket, decide whether to recurse or process an ElementList
function ProcessPList (plist, i_p, λ, deferred)
1   if plist[i_p] = nil then
2       return // no data to pass to λ from this bucket
3   else if plist[i_p].lock = p_inner then
        // Recurse into child PointerList
4       EnterPList (plist[i_p], λ, deferred)
5   else if plist[i_p].lock = e_avail ∧ cas(&plist[i_p], e_avail, e_lock)
      then
        // Iterate over entries in locked ElementList
6       for i ∈ 1 ... plist[i_p].count do
7           λ(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 ← deferred ∪ ⟨plist, i_p⟩

// Handle PointerList elements that were deferred
function HandleDeferred (λ, deferred)
1   for ⟨plist, i_p⟩ ∈ deferred do
2       if plist[i_p].lock = p_inner then
            // Bucket was rehashed, so recurse into it
3           deferred ← deferred − ⟨plist, i_p⟩
4           EnterPList (plist[i_p], λ, deferred)
5       else if
          plist[i_p].lock = e_avail ∧ cas(&plist[i_p], e_avail, e_lock)
          then
            // Iterate over entries in locked ElementList
6           deferred ← deferred − ⟨plist, i_p⟩
7           for i ∈ 1 ... plist[i_p].count do
8               λ(plist[i_p].keys[i], plist[i_p].values[i])
9           plist[i_p].lock = e_avail
10  if deferred ≠ {} then
11      optionalBackoff()
12      goto 1
```

order to prevent 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, though the guarantees are weaker than Go's sequential map.

IV. Implementation

The IHT provides the same API as the sequential map. As we shall see in Section V, 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.

A. Compiler Integration and Transformations

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` or `mapassign` 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 one of these calls returns, it is unreasonable to delegate lock release to the programmer.

When the compiler generates a `mapaccess` or `mapassign` 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. Subsequently, the pointer is not live, so the lock can be released. In our lock implementation, the same function can release the lock, regardless of whether it protects an `ElementList` or a maximum-depth `PointerList`. To exploit this property, we extend the API in Listing 1 so that functions retrurn references to both the lock and the value.

An additional complication is that multiple calls to `mapaccess` or `mapassign` 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`, as the seed for each `PointerList`'s hash function is randomly generated at run-time. 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 spin locks reentrant.

B. 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, supports the Go 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's runtime detects racy map accesses. Similarly, we track calls to `sync.Interlocked` 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.) We leverage goroutine-local storage to limit overhead for these dynamic checks to detect and prevent acquisition of multiple locks during interlocked execution. We do not forbid overlapping interlocked accesses to *different* maps.locking order.

When keys are known statically, it is trivial to support multi-key operations. Deadlock is not possible because the locks can be hashed in advance, and then acquired in an order that is equivalent to in-order traversal of the IHT. To handle keys that map to the same `ElementList`, and to continue to support race detection, requires overhead linear in the number of keys.

C. Iteration

Existing approaches to iteration in concurrent collections 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 linearizable: 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 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.

V. Performance Evaluation

In this section, we explore two performance criteria. First, we evaluate the performance of the IHT algorithm against state-of-the-art. We perform this evaluation in Java, so that all algorithms can be on the same footing. The experiments focus on the common set of operations available to the IHT and its competitors: insertion, removal, and lookups of a single element. Second, we look at the behavior of the IHT when implemented in Go, and compare against the best concurrent map implementations for Go. The experiments consider both elemental operations and iteration.

Both sets of experiments were conducted 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 used the 64-bit Server JVM version 1.8.0_11-b12. We conducted additional experiments (not presented in this manuscript) on a single-chip Core i7-4770, and observed the same performance trends.

A. Raw IHT Performance Evaluation in Java

The two most scalable known hash table implementations [13] are the Split Ordered List (SOList) [18] and the re-

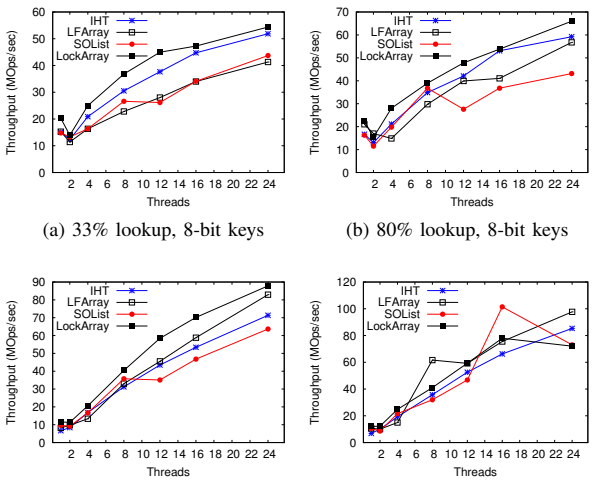
sizable nonblocking hash set (LFArray) [20]. Both are lock-free data structures with highly optimized implementations in Java. Based on these data structures, we consider three comparison points:

- The SOList uses a sorted lock-free list [29], [19] to store key/value pairs, and employs an auxiliary fixed-depth "directory" of hash values to rapidly jump to the position within the list where an insert, remove, or lookup should take place. To iterate over the SOList, a thread accesses the underlying list directly. As a nonblocking data structure, operations are achieved by using a CAS to insert or remove a list node. Update operations are not supported. The SOList can expand while preserving $O(1)$ overhead, but like the IHT, can not shrink in response to a significant decrease in the number of items it holds.
- The LFArray uses one level of indirection to reach an array of pointers to "freezable sets", or "FSets". An FSet resembles our `ElementList`, in that it includes an array of keys. Like the SOList, a lookup does not require any CAS, and reaches the appropriate FSet in $O(1)$ time. However, to insert or remove an element, the LFArray uses copy-on-write of an FSet, and then uses a CAS to install the copy. The LFArray resizes (both expanding and shrinking) by creating a new array, and then lazily moving elements into it.
- LockArray is a locking version of LFArray. The use of locks within FSets avoids copying. However, lookups must lock an FSet before performing a lookup, which can create more overhead in read-dominated workloads.

In all three cases, the data structure constrains the hash function, in order to ensure the correctness of resizing. For consistency, we use the same hash function for the SOList, LFArray, LockArray, and our Java IHT.

Strictly speaking, the available Java implementations of LFArray. LockArray, and SOList are not maps, but rather hash sets. We emulate a set in our Java IHT implementation by making the value equal the key. Figure 2 compares the four algorithms with two different key ranges (8-bit and 16-bit), and two different operation mixes: On the left, inserts, lookups, and removes are selected with equal probability. On the right, 80% of operations are lookups, with the remainder split between inserts and removes. Note that a slowdown at 2 threads is expected, due to cross-chip communication.

When the key range is 8-bits, `LockArray` has the best performance, and IHT performs second best. In these cases, SOList suffers from the overhead of its directory, and LFArray suffers from wasted work: if two operations are simultaneously copying the same bucket, one will eventually fail and retry its operation. In such cases, throughput is better when one of those threads waits on the other, via a spinlock, and then modifies an object without performing a copy. In the 8-bit experiment, the shape of the LockArray and IHT are virtually the same, with the main difference



(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: Microbenchmark performance in Java

being the cost of lock acquisition: in LockArray there are simple spinlocks, whereas IHT uses `CMLocks`.

With 16-bit keys, the IHT has additional levels of indirection versus the LFArray and LockArray. Thus while all algorithms scale roughly equivalently, the IHT pays a penalty. In addition, we see that at 80% lookup, the lock-free algorithms are more likely to outperform the lock-based ones: when conflicts are rare, the lock-free techniques avoid any CAS overheads on 80% of their operations. This more than compensates for the cost of copying in the LFArray, and for the cost of pointer chasing in the SOList. However, the IHT provides stable, scalable performance that remains competitive. In particular, the high rate of memory allocation in the SOList leads to unstable performance.

These experiments favor prior work, since the IHT uses a key and value, while the other data structures only manage keys. Nonetheless, the IHT always shows good scalability. Furthermore, the IHT supports features that are not available in those data structures, such as iteration (not present in the LFArray and LockArray) and a map interface (not present in the SOList and LFArray). We now turn our attention to the Go language, where we can assess the importance of our tight integration into the Go `map` API.

B. The Benefit of Integrating Into Go

We now evaluate IHT performance in Go. Integration of the IHT into Go was not trivial, requiring several thousands of lines of changes to the Go compiler and run-time libraries. However, doing so made it possible to leverage the same features as are available to the sequential Go `map`.

The LFArray and LockArray algorithms do not support iteration: if a resize occurs, it is possible for an item to be visited twice, or missed entirely. Thus we only carry forward

the SOList and IHT evaluation from the prior section. In this case, we use the open-source SOList implementation, available in the `gotomic` [30] package. Note that it supports a map interface, instead of the set interface of the Java SOList implementation. We add three more comparison points:

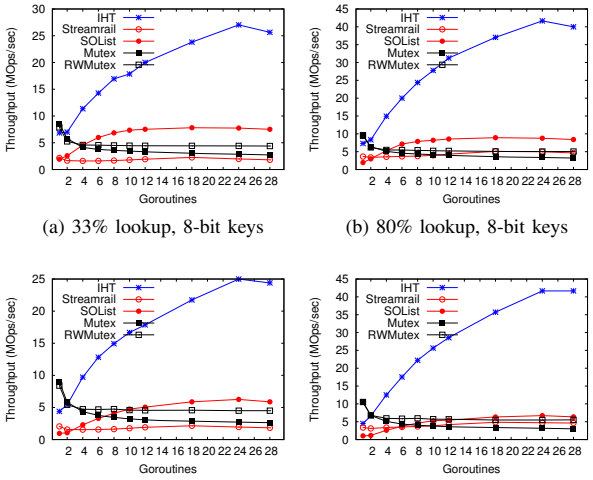
- Streamrail – A lock-based map, implemented as a fixed-size array of RWMutex-protected Go maps [31].
- RWMutex – A RWMutex-protected default Go map.
- Mutex – A Mutex-protected default Go map.

Since SOList and Streamrail are library-based, 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{}`, 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 lock-free list.

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.

Microbenchmark Performance: Figure 3 repeats the experiments from Section V-A. However, now integer keys are 64 bits, and each implementation stores a byte-sized value at each key. As before, we consider 8-bit and 16-bit keys, and 33% and 80% lookup ratios. Again, each data point is the average of 10 trials.

Surprisingly, the IHT's raw performance is worse in Go than in Java. Some of this is attributable to a different testing harness. The remainder is due to differences between the Go compiler's static inlining policy, and the ability of the JVM to inline at run time, based on a program profile. We suspect



(a) 33% lookup, 8-bit keys (b) 80% lookup, 8-bit keys

(c) 33% lookup, 16-bit keys (d) 80% lookup, 16-bit keys

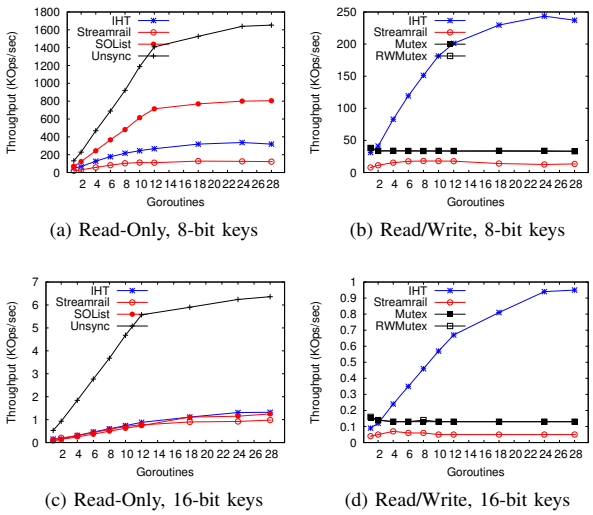
Figure 3: Microbenchmark performance in Go

that differences in garbage collection algorithms could also explain part of this gap. In all cases, these are overheads that are common to all of the Go map implementations we considered, and do not affect relative performance.

As expected, the IHT has more latency than a lock-protected default map. At one thread, Go's `mutex` implementation is very 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 since we use the built-in Go hash function, collisions cause some 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.

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) [32] begins). While the RWMutex provides better



(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 4: Iteration microbenchmark

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.

C. Iteration Performance

One of the essential features of the IHT is its support for iteration. We created a microbenchmark with a 100% mix of iterations by each goroutine. 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 4, 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.

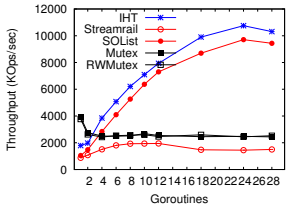
For small maps, the SOList also outperforms the IHT under read-only iteration. The SOList's nonblocking implementation can avoid any `CAS` instructions during an iteration, and the lack of insert and remove operations avoids the need for any accesses to the shared counters used by the SOList to manage directory height. Thus the SOList enjoys disjoint-access parallelism [33]. 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 API call). Thus each iteration acquires many locks. 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.

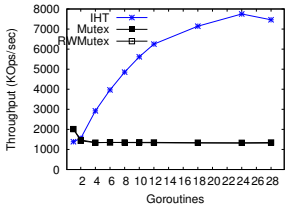
Unfortunately, SOList does not support mutating iteration, because the nonblocking implementation cannot guarantee that, upon returning a key/value pair, that pair remains in the map. Streamrail provides mutating iteration, so long as there are not concurrent insert/remove/lookup operations. As with 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 on our 24-thread machine.

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

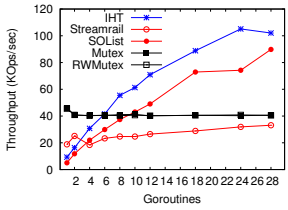
We observe that Streamrail, SOList, and IHT all provide weak iteration guarantees. In Streamrail, one sub-map is locked at a time, and thus each element returned by the iterator is present in the map, but the set of returned elements is not an atomic snapshot. In SOList, one element is returned at a time, without preventing concurrent mutations throughout the remainder of the underlying list. SOList cannot handle mutation during iteration. IHT's behavior is like Streamrail:



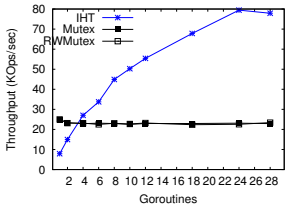
(a) Read-Only iteration, 8-bit keys



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



(c) Read-Only iteration, 16-bit keys



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

Figure 5: Mixed workload with 2.5% iteration, and remaining operations split evenly.

every element that is present in the map for the duration of an iteration is returned by the iterator, but the entire set of returned items is not an atomic snapshot.

D. 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 sub-map changes after an iterator creates the channels used by its spawned goroutines.

Figure 5 shows 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. For the first time, SOList outperforms 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 perform equivalently, the higher performance of IHT's elemental accesses gives it a slight edge. As in Figure 4, 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 3.

VI. 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× the peak achieved speedup by the default (typically at one thread). Furthermore, even in Java, where the IHT cannot benefit from tight integration with the run-time libraries, the IHT is competitive with the best known map implementations.

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, which can easily be extended 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 operations over a set of map locations known at the beginning of the critical section, we believe it will be possible to guarantee atomicity and deadlock-freedom.

Acknowledgments

This material is based upon work supported in part by the National Science Foundation under Grant CAREER-1253362. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

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

[2] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing C++ Concurrency," in *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, Austin, TX, Jan. 2011.

[3] J. Manson, W. Pugh, and S. Adve, "The Java Memory Model," in *Proceedings of the 35th ACM Symposium on Principles of Programming Languages*, Long Beach, CA, Jan. 2005.

[4] Standard C++ Foundation, "C++11 Overview," 2017, https://isocpp.org/wiki/faq/cpp11.

[5] D. Lea, "JSR-166 Interest Site," 2017, http://gee.cs.oswego.edu/dl/concurrency-interest/.

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

[7] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[8] G. Agha and C. Hewitt, "Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming," in *Research Directions in Object-Oriented Programming*. MIT Press, 1987, pp. 49–74.

[9] ISO/IEC JTC 1/SC 22/WG 21, "Technical Specification for C++ Extensions for Transactional Memory," May 2015. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4514.pdf

[10] 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.

[11] M. P. Herlihy and J. M. Wing, "Linearizability: a Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[12] 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.

[13] 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.

[14] 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, Jul. 2013.

[15] 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, Jun. 2003.

[16] 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.

[17] 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.

[18] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," *Journal of the ACM*, vol. 53, no. 3, pp. 379–405, 2006.

[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] 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, Jul. 2014.

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

[22] 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, Jun. 2012.

[23] 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.

[24] A. Spiegelman, G. Golan-Gueta, and I. Keidar, "Transactional Data Structure Libraries," in *Proceedings of the 37th ACM Conference on Programming Language Design and Implementation*, Santa Barbara, CA, Jun. 2016.

[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, Jun. 2012.

[26] 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*, vol. 23, no. 2, pp. 375–382, 2012.

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

[28] 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.

[29] T. Harris, "A Pragmatic Implementation of Non-Blocking Linked Lists," in *Proceedings of the 15th International Symposium on Distributed Computing*, Lisbon, Portugal, Oct. 2001.

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

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

[32] 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, Jun. 1995.

[33] 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.