Adaptive Lock-Free Maps: Purely-Functional to Scalable

Ryan R. Newton Peter P. Fogg Ali Varamesh
Indiana University, United States
{rrnewton,pfogg,alivara}@indiana.edu

Abstract

Purely functional data structures stored inside a mutable variable provide an excellent concurrent data structure—obviously correct, cheap to create, and supporting snapshots. They are not, however, scalable. We provide a way to retain the benefits of these pure-in-a-box data structures while dynamically converting to a more scalable lock-free data structure under contention. Our solution scales to any pair of pure and lock-free container types with key/value set semantics, while *retaining* lock-freedom. We demonstrate the principle in action on two very different platforms: first in the Glasgow Haskell Compiler and second in Java. To this end we extend GHC to support lock-free data structures and introduce a new approach for safe CAS in a lazy language.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications - Concurrent, Distributed, and Parallel Languages

General Terms Languages, Performance

Keywords Lock-free algorithms, Concurrent data structures

1. Introduction

Purely functional, or *persistent*, data structures have been wildly successful. High-quality libraries of immutable collection types are now broadly available in both functional and imperative languages, and these play an important role in the Scala, Clojure, OCaml, F#, and Haskell ecosystems, among others. One non-obvious benefit of these immutable datatypes is that they also provide the *easiest* way to build concurrency-safe, *mutable* datatypes. For example, it is difficult to engineer a threadsafe mutable set datatype, but it is easy to take an immutable set and place it in a mutable container—e.g., an `(IORef Set)` in Haskell.

This "pure data in a box" idiom is extremely common in systems software and servers written using Haskell. Because there is only a *single* mutable memory location, achieving atomic updates to the structure is straightforward. In Haskell this is done with the `atomicModifyIORef` primitive. The problem with this approach is that it *does not scale well under contention*, as threads must all modify the same memory location. Compounding this are additional problems with implicit locking in the runtime system related to lazy evaluation. In this paper, we show how to solve both the semantic problems (Section 3) and practical problems with laziness (Section 2), but this still leaves non-scalability as a fundamental problem with using *only* pure data in a box.

Therefore it remains important to engineer scalable concurrent data structures, even for functional languages. Fortunately, there is a wealth of literature to refer to in this area—on both fine-grained locking [8], as well as lock-free and wait-free structures [19]. Next, once we achieve a scalable mutable container, such as a set or map, we would ideally put it to work in *all* cases where we need a concurrent set or map. Unfortunately, there are drawbacks as well as advantages to these structures. They can be heavyweight. Papers which introduce new lock-free structures (e.g. [9, 18]), almost always evaluate them by subjecting a *single* instance of the data structure to stress testing. In the process, the data structure is sometimes evaluated under various levels of contention (the "*concurrency range*" [9]), but typically not in terms of:

- **memory overhead** relative to simple non-concurrent structures. This is especially relevant for many small collections.
- **allocation/initialization overhead**, which can increase due to additional state that needs to be allocated and initialized.

Our experience in Haskell is that overheads in both these categories are significant. Thus, rather than using scalable structures *everywhere*, the programmer makes a trade-off based on whether a *particular* collection is expected to have contended accesses. Should it be a lightweight but non-scalable pure-in-a-box data structure? Or a clunkier but scalable one? Often there is no statically knowable answer. Consider a mutable map where the values are themselves mutable sets:

```
table :: IORef (Map Key (IORef IntSet))
```

Some keys may be "hot", experiencing frequent modification, while others are cold. It might be clear that the outer `Map` should be replaced with a concurrent tree-based map or hashtable, but there is no right answer for what `IntSet` implementation to use. In this paper, we propose a simple solution: purely functional structures that transform into scalable ones under contention.

This transitioning allows the choice of a concrete implementation to be made at runtime, choosing a representation that performs well under the application's actual workload. A second goal is the "do no harm" principle—the hybrid data structure must remain as performant as a pure data structure in the uncontended case. Our aim is to eliminate the burden of choosing between pure and scalable structures, simplifying the programmer's job.

While the mechanism of swapping data structures at runtime is used in many contexts [2, 3, 23], it does not generally work well in concurrent settings. Nevertheless, our particular technique has an advantage: *using purely functional data as a starting point enables retaining lock-freedom—before, during and after representation swapping.* Conversely, if the starting structure were itself a scalable, mutable structure, then performing the transition would lose lock-freedom (due to an inability to snapshot the structure in an O(1) step). Indeed, perhaps the historically esoteric role of persistent

ICFP '2015, August 31-September 2, 2015, Vancouver, Canada.

ACM [to be supplied]. . . $15.00.
http://dx.doi.org/10.1145/TDB

data structures may be what led to this simple solution for lock-free, dynamic scalability being overlooked.

Our contributions in this paper are:

- We present an algorithm for swapping representations at runtime that leverages immutability in the starting state to achieve advantages over previous approaches to data structure adaptation [3, 23]. The algorithm assumes a map-like interface and semantics, but the synchronization strategy can also be adapted to bag-like interfaces (Section 4).
- We examine contention problems with pure-in-a-box datatypes in a mature functional compiler, GHC. Contention problems are compounded by mechanisms introduced for lazy evaluation, in particular the *black-hole* policy. We demonstrate how to avoid both explicit locking and black-holes—which defeat lock-free progress guarantees.
- We describe our *ticketed CAS* (compare and swap) approach to adding atomic operations to GHC, and demonstrate how it preserves the expected semantics of CAS in spite of being hosted in a non-strict language *without* meaningful pointer equality.
- We evaluate adaptive bags and maps compared to their non-adaptive counterparts, using both GHC Haskell and the Oracle Java VM as experimental platforms. We show that adaptive data outperforms either pure or scalable containers overall in the nested collection case with mixed hot and cold inner containers.

2. Prerequisite 1: Lock-Free Concurrent Haskell

As the central contribution of this paper involves lock-freedom for data structure operations, we must define lock-freedom in Haskell. Concurrent Haskell was defined in 1996 in two levels: a (deterministic) reduction of the purely functional language into an "infinite tree" of primitive `IO` actions [17], and a separate semantics for evaluating the action tree. These actions include forking threads and reading and writing dataflow variables, but the full Glasgow Haskell compiler also includes many other primitive IO actions such as system calls.

For a system of `IO` threads running in a Haskell runtime system, we can define lock-freedom as:

Definition 2.1. *For any execution schedule of `IO` threads, π, there exists a bound k, such that for all points in the execution, some thread makes progress within the k steps.*

We consider only mutator time, not garbage collection. Here, *progress* is defined as an `IO` thread completing any primitive, built-in IO action. The primitive IO actions include blocking reads and writes to *MVars*, which enable deadlock but not livelock. In general, most uses of MVars clearly violate lock-freedom (with some threads not making progress until other threads are scheduled). The more subtle problem comes from the interaction with lazy evaluation itself.

The problem with black holes Unlike later formulations of the Spineless Tagless G-Machine for Haskell (STG machine [13]), the original Concurrent Haskell semantics did not model thunk evaluation or sharing of thunks between threads. In fact, this is a critical issue, and a later paper in ICFP'09 [15] addressed thunk sharing and offered various policies for replacing a thunk at runtime with a *black hole*—a black hole locks a thunk on behalf of its owner and blocks other threads from attempting to compute that thunk.

Regardless of specific policy—eager black holing, or lazy black holing performed by the runtime between scheduling threads—black holes **directly defeat the lock-freedom property**. Just as with a regular lock, if the thread owning the black hole is not scheduled, then other threads may potentially not be able to make progress. Thus any Haskell program that makes a thunk accessible to other threads—e.g., by placing it in an `IORef`—violates lock-freedom (2.1). Indeed, we find that unexpected black-holing is a major problem when we tune parallel Haskell applications.

Unfortunately, the *only* way to perform an atomic operation on a mutable variable (`IORef`) in Haskell has been to use `atomicModifyIORef`, which is a black-hole risk. By its nature it must publish a thunk to other threads on *every* call, defeating lock-freedom. The reason for this can be seen in its type signature:

```
atomicModifyIORef
  :: IORef a → (a → (a, b)) → IO b
```

This provides *not just* a limited compare-and-swap (CAS) style operation, rather, it performs an arbitrary function *atomically*. Neither does it permit speculation: the function will only run once. Of course, there is no free lunch, and `atomicModifyIORef` actually only guarantees atomicity by placing a thunk in the IORef. (In fact, GHC's implementation contains a rather clever optimization where the retry loop upon failed CAS attempts does *not* allocate a new thunk, rather it allocates one suspended function application and then *mutates* it with new arguments until CAS succeeds.)

GHC also provides a strict variant, `atomicModifyIORef'`. But this variant simply forces the thunk *after* publishing it. Thus it cannot prevent threads touching the same thunk. To achieve lock-freedom, we need a new primitive—we need to expose CAS to Haskell code, directly and safely. That is our next, and final, prerequisite before describing our proposed algorithm in Section 4.

3. Prerequisite 2: Atomic Memory Operations in a Lazy Language

Lock-free algorithms and data structures have been slow to make their way to functional language implementations. In part, this may simply be because compilers—GHC, SML/NJ, OCaml, Racket, etc—need to support all the requisite atomic machine instructions. In Haskell's case there has been a deeper reason as well: unlike many of its strict counterparts, Haskell has no defined notion of *pointer equality*[1]. This is a boon to the optimizer, which is free to unbox and rebox a pure value, changing its physical identity but preserving its value—enabling, for example, the *worker/wrapper* optimization [7], which is often able to unbox values inside loops. But something is lost as well; after all, pointer equality is the only equality supported by the computer architecture's CAS instruction!

To see the problem, consider a direct attempt to implement a `(casIORef ref old new)` operation in Haskell, which returns a result indicating whether the operation succeeded:

```
casIORef :: IORef a → a → a → IO (Bool, a)
```

This, unfortunately, is the version of CAS that was exposed in GHC 7.2 and 7.4[2], though, as we will see, it is not safe to use. Consider this simple use of `casIORef`:

```
do old ← readIORef r
   let new = old + 3
   casIORef r old new
```

We expect the first and last uses of `old` to represent the same pointer. But, in fact, there is no way for the programmer to ensure this. For example, if `old` is of type `Int`, then GHC is free to unbox to the internal `Int#` type, and then rebox upon the call to `casIORef`.

[1] Hence the aptly named internal operation `reallyUnsafePtrEquality#`.

[2] Actually, the internal compiler primitive is called `casMutVar#` and has a lower level type operating on a `MutVar#` and passing `State#` tokens, but these are GHC-internal details.

Our approach to this problem is to introduce a technique we call *ticketed CAS*. In this approach, the `old` argument is replaced by an opaque value of type `Ticket a`. This ticket represents a record of a previous observation, and it is presented in future attempts to modify the value. A ticket is used in a manner akin to a logical *version number* for the mutable location—but tickets lack an ordering, and may repeat. There is no way to manufacture tickets from thin air; thus using the ticketed approach also requires a way to read them—not the value of a mutable variable, but a ticket representing its current state:

```
do t ← readForCAS r
   let new = peekTicket t + 3
   casIORef r t new
```

Here it is perfectly fine to *extract* from a `Ticket t` a value of type `t`, using `peekTicket`[3]. Peeking a ticket can happen outside of the IO monad and it does not violate referential transparency—while values do not have stable pointers, pointers have stable values.

Though not used above, the return value of `casIORef` returns a ticket as well, giving it the type:

```
casIORef :: IORef a → Ticket a → a
         → IO (Bool, Ticket a)
```

Atypically, this CAS returns a ticket corresponding to the *newest* value observed in the reference (counter to the usual convention of returning the old value). The `Bool` indicates success or failure, and in the case of failure, the ticket is needed for a retry, but in the case of success, the ticket may also be needed for the next operation on the IORef.

Implementing tickets In version GHC 7.6, we added additional atomic operations—for example exposing CAS for array elements as well as IORefs—and we switched to using a ticketed interface. Fortunately, the GHC compiler already had a notion of an `Any` type constructor, representing a value that the compiler knows is a pointer, but pointing to to a data constructor or function of unknown type. This serves to prevent any and all compiler optimizations that change the representation, while enabling the garbage collector to update the pointer during GC. Thus we use `Any` to represent tickets.

With ticketed CAS (and its friends such as fetch-and-add) we have the basic building blocks for implementing lock-free data structures and algorithms. In fact, even for operations as simple as updating a boxed `Int` counter, Haskell-level atomic operations offer a substantial benefit. For example, we can use CAS to implement a *speculative* alternative to `atomicModifyIORef`, which computes the updated result *before* modifying the reference. This avoids the problem of publishing a thunk to other threads (black holes), but in exchange it may waste work by computing a result that is not used.

In fact, because this speculative approach greatly improves the performance of atomically modifying pure-in-a-box data structures under high contention, it becomes our new baseline against which truly *scalable* structures must be measured, rather than the previous, `atomicModifyIORef'` baseline.

4. Adaptation Algorithm

In this section we present our algorithm for lifting a pair of data structures—pure and lock-free respectively—into a hybrid structure that retains lock-freedom. In Figures 2-3 we present the algorithm using the specific case of a *Map* interface. That is, we assume correct `put`, `get`, `remove`, and `putIfAbsent` both for the starting data structure (`S1`) and the target (`S2`). The difference between operations on `S1` and `S2` is that the former are *pure* and the latter are in the `IO` monad. You can see this difference on lines 43 and 44 of Figure 3. Moreover, we assume that `S2`'s operations are both *lock-free* and *linearizable*.

Note that while we have used Maps in our code samples here, the algorithm generalizes easily to:

- Sets – by simply omitting values in the put and get calls, or using `()` for the value.
- Bags – by also changing the semantics of the underlying `putS1`/`putS2` to allow duplicates, with the simplest case being a unit key type for a single bag.

[3] Incidentally, `peekTicket` must be marked NOINLINE, but this is a GHC-specific issue rather than something more fundamental. `peekTicket` becomes a type cast rather than a function call in the intermediate representation, and as such it is not sufficient to prevent the compiler of learning the representation of `t` enabling mischief. See `atomic-primops` package, issue #5.

```
1  -- A Map-like datatype indexed by key and value:
2  type Hybrid k v = Ref (HyState k v)
3  -- S1 is a pure data structure, S2 is mutable:
4  data HyState k v = A  (S1 k v)
5                   | AB (S1 k v) (S2 k v)
6                   | B  (S2 k v)
7
8  new :: IO (Hybrid k v)
9  new = newRef (A emptyS1)
```

Figure 1. Definition of the hybrid data structure. The structure is initialized in the `A` state, containing a pure value of type S1.

```
10 initiateTransition :: Hybrid k v → IO ()
11 initiateTransition r =
12   -- Must allocate before we try to modify:
13   do emptS2 ← newS2   -- Could be wasted!
14      case! atomicModify r (fn emptS2) of
15        Just s1 → fork (copyThread r s1 a
16                                    emptS2)
17        Nothing → return ()
18  where
19   fn emptS2 x =
20    case x of
21      A s1      → (AB s1 emptS2, Just s1)
22      -- Otherwise, someone beat us to it:
23      B s2      → (B s2,      Nothing)
24      AB s1 s2  → (AB s1 s2, Nothing)
25
26 -- The copy thread always uses putIfAbsentS2 so
27 -- as not to overwrite logically newer changes.
28 copyThread :: Hybrid k v → S1 k v
29            → S2 k v → IO ()
30 copyThread r s1 s2 =
31   do forM_ s1 (λ k v →
32                putIfAbsentS2 k v s2)
33      -- Finalize transition:
34      atomicModify r
35        (λx → case x of
36                A  _ → error "impossible"
37                AB _ s2 → (B s2, ())
38                B    s2 → (B s2, ()))
```

Figure 2. The algorithm for transitioning, including asynchronously copying from the `A` structure to the `B` structure. Note that copying can itself be parallelized, if desired. Also, while this implementation of `initiateTransition` does not return until success, the overall algorithm remains correct if transition gives up after an effort count.

The basic idea of the algorithm is to provide a wrapper data structure that is a sum type, `HyState`, combining the pure `S1`, the scalable `S2`, or both. A complete `Hybrid` is a mutable pointer to `HyState` that initially contains a value of type `S1`. In this state, it simply forwards all operations to the underlying `S1` implementation, applying the changes through an `atomicModify` at the top-level mutable reference (which uses the speculative approach described in the previous section to avoid thunks). Upon detecting contention on the mutable reference, a *transition* is initiated, creating a new structure of type `S2` and copying all data into it. Once the transition is complete, all operations are forwarded to the new, scalable implementation.

The definition of the `Hybrid` type is given in Figure 1. Its three states are named: `A`, representing the original pure structure; `B`, representing the post-transition scalable structure; or `AB`, indicating that a transition is in progress. We initialize the structure in the `A` state with an empty value of type `S1`.

The definition and use of the transition function (Figure 2) ensures two useful properties of *all* `Hybrid` objects: (1) that transitions through the `A`→`AB`→`B` lifecycle are monotonic, and (2) that at most *one* `S2` object in the heap is ever reachable from a given `Hybrid`'s reference cell. Thus any read of type `S2` is a valid read and will not go stale. These properties appear in the next section as Lemmas 5.1 and 5.2.

Building blocks Figures 3, 5, and 6 show the code for the core API operations on Hybrids. In this code we assume access to mutable references (`Ref`) which support reading and modification via compare-and-swap. Our code uses both `tryModify` and `atomicModify`. The former makes an attempt to modify the reference, but gives up and returns false if the CAS fails some finite number of times. Here is the definition for `tryModify`:

```
tryModify :: Ref a → (a → (a,b))
          → IO (Maybe b)
tryModify r fn = retryLoop numTries
  where retryLoop 0 = return Nothing
        retryLoop n = do
          t ← readForCAS r
          let old = peekTicket t
              (new, ret) = fn old
          (success, t') ← casRef r t new
          if success
            then return (Just ret)
            else retryLoop (n - 1)
```

The constant `numTries` is a tunable parameter of the algorithm. In contrast, `atomicModify` persists until success:

```
atomicModify :: Ref a → (a → (a,b)) → IO b
atomicModify r fn = do
  case! tryModify r fn of
    Nothing → atomicModify r fn
    Just x  → return x
```

To save space in our algorithm code, we above introduce syntactic sugar "`case! e...`", which desugars to "`do x ←e; case x...`".

The loop in `atomicModify` raises the question of termination, but we still have a lock-free guarantee—if a call to `tryModify` has failed, some other thread has successfully modified the location and thus made progress. It may be the case that some thread is continually failing its CAS operations (due to an unfair schedule), but the system as a whole is still making progress. (Further, in finite executions, eventually all contention dissipates and all `atomicModify` operations on all threads succeed.) Indeed, the CAS retry loop inside `tryModify` is the basic building block on which virtually all lock-free algorithms are based.

The copy thread During a transition, we fork off a copy thread (Figure 2) which is responsible for inserting old values from the `A` structure into the new, empty `B` structure. Since this copy occurs in a background thread it is *off the application's critical path*. As a result we can still provide low-latency operations on the data structure during its transitional phase. Unfortunately, we necessarily increase memory usage during the copy, since `A` cannot be freed until all of its information is inserted into `B`.

A side observation here is that the copy process can optionally be parallelized. In fact, this is a feasible option, as most immutable containers are balanced and well-suited to parallel divide-and-conquer traversal; and, of course, the target scalable `S2` structure is designed precisely for high-load concurrent insertions!

Logical time Transitioning asynchronously using the copy thread requires some finesse—in order to maintain correctness, the hybrid structure must simultaneously handle both an `S1` and `S2` while `S2` is concurrently modified, all the while preserving the semantics of a single, concurrent Map. In order to maintain this abstraction, the copy thread must not overwrite *newer* values in `S2` (or newer *removals*) with older values from `S1`. We thus need a notion of logical time, which relates values in flight to the point at which they were *first written*.

Each code path which calls one of `put`, `get`, `putIfAbsent`, or `remove` (specialized to `S1` or `S2`) has a *commit point* occuring at a specific line of code. The state transitions on lines 14 and 34, as well as the copies on line 32, are also atomic and can be considered to increment logical time. In fact, any asynchronous background events that increment logical time are perfectly fine, as they do not affect the *ordering* of two `put` events, which determines the "vintage" of the data. The important part is that the puts that occur via the copy thread on 32 are *not* associated with the current time, but with the moment that each item was *originally* added to `s1` through a `put` operation.

Further, there are two more properties on which the correctness of the algorithm depends:

- Non-blocking complete snapshots. This is the key ability of the pure structure `S1`—we can get an exact and exhaustive snapshot at a single point in logical time, using merely an $O(1)$ `readRef`.
- Causality: a `get` event may only reflect the latest `put` or `remove` event on the same key. Thus every `put` performed by `copyThread` must not change the state *if* newer modifications to the key have occurred.

The pure `S1` structure makes the first property a given. Then the way the algorithm ensures causality is twofold. First, the copy thread only uses `putIfAbsent` so that it can *never* interfere with any already written key in `S2`. Second, we use *tombstone* values to explicitly mark removals while in the transitioning phase[4].

Tombstone values and pseudocode A tombstone ☒ is a distinguished value which explicitly represents absence of a particular key. This scheme is necessary to ensure that the copy thread does not write logically older values which have since been deleted—if deletion were implemented merely using `S2`'s `delete` operation, the call to `putIfAbsent` at line 32 could insert values which were in fact removed at a later logical time.

Thus a removal during the transition phase is, in reality, an insertion (line 128). The tombstone can be easily implemented by wrapping all values in Haskell's `Maybe` type—which would change the type `S2 k v` into `S2 k (Maybe v)`—but we elide this detail for brevity and simply assume type `v` is lifted to include ☒. Likewise, we define helper functions such as $\texttt{getS2}_{\boxtimes}$ (Figure 3)

[4] We call these "tombstones" following the convention from the distributed systems literature [12].

```
39 get :: k → Hybrid k v → IO (Maybe v)
40 get key r =
41  case! readRef r of
42   -- The start/end cases are easy:
43    A s1 → return (getS1 key s1)
44    B s2 → getS2⊠ key s2
45    AB s1 s2 →
46      -- getS2 is the commit point for the
47      -- operation, vis a vis serializability:
48      case! getS2 key s2 of
49        -- It may still be in flight:
50        Nothing → return (getS1 key s1)
51        -- Logically more recent deletion trumps s1:
52        Just ⊠ →    return Nothing
53        -- Logically more recent s2 value trumps s1:
54        Just a  → return (Just a)
55
56 -- A helper to hide the use of tombstones (@⊠@):
57 getS2⊠ :: k → S2 k v → IO (Maybe v)
58 getS2⊠ k s2 = case! getS2 k s2 of
59                  Nothing → return Nothing
60                  Just ⊠ → return Nothing
61                  Just v → return (Just v)
```

Figure 3. The algorithm for reading an adaptive lock-free collection. Note that the `get` operation is indeed read-only; it cannot affect the state of the hybrid. Above, the ⊠ symbol is used as a "tombstone" to represent deletion.

```
62 -- Lifting functions from S1 to hybrids:
63 liftS1 :: (S1 k v → S1 k v) → HyState k v
64         → (HyState k v, Maybe (S2 k v))
65 liftS1 f h = case h of
66                A  s1   →
   (A (f s1), Nothing)
67                AB _ s2 → (h, Just s2)
68                B    s2 → (h, Just s2)
```

Figure 4. Lifting functions on `S1` to functions on the `Hybrid` type while in the `A` state. If the state transition beat us there, we "fail" by returning `s2`.

which "ignore" tombstone values, treating them as equivalent to absence of the key.

After transition to `B` state is complete, removes again become removes, and they release storage for both key and value. Optionally, we could launch a background thread to "clean up" any remaining tombstone values after the transition is complete (freeing the keys), but that improvement is beyond the scope of this paper.

5. Proof of Correctness

We follow the model of multiprocessor computation given by Herlihy and Wing [11]. For our proofs of correctness and linearizability, we use operational style arguments as is common in the concurrent data structure literature [9, 11, 18].

5.1 Global State Invariants

Here we return to the two properties mentioned early in Section 4.

Lemma 5.1. *Monotonicity 1. Every `Hybrid` transitions monotonically through the three states: $A \rightarrow AB \rightarrow B$.*

Proof. There are only two points where we case on the state of the data structure inside of an atomic operation and return a different state. On line 21, we modify the state from `A` to `AB`. On line 37, the

```
69 -- -- Overwriting put.
70 put :: k → v → Hybrid k v → IO ()
71 put key val r =
72  -- First peek to see if an atomic
73  -- instruction at the root is necessary:
74  case! readRef r of
75   A _ →
76    case! tryModify r
77            (liftS1 (putS1 key val))
78    of Nothing → do initiateTransition r
79                    put key val r
80       Just Nothing   → return ()
81       Just (Just s2) → putS2 key val s2
82   B    s2 → putS2 key val s2
83   AB _ s2 → putS2 key val s2
84
85 -- -- Gentle put.
86 putIfAbsent :: k → v → Hybrid k v
87             → IO ()
88 putIfAbsent key val r =
89  case! readRef r of
90   A _ →
91    case! tryModify r
92            (liftS1 (putIfAbsentS1 key val))
93    of Nothing → do initiateTransition r
94                    putIfAbsent key val r
95       Just Nothing   → return ()
96       Just (Just s2) →
97         putIfAbsentS2⊠ key val s2
98   B     s2 → putIfAbsentS2⊠ key val s2
99   AB s1 s2 →
100   case getS1 key s1 of
101      Nothing → putIfAbsentS2⊠ key val s2
102      -- Value may be in flight, write only if newer tombstone:
103      Just _  → casValS2 key ⊠ val s2
104
105 -- Only if entry is present, then compare-and-swap a new value:
106 casValS2 key old new s2 =
107   do mr ← getValRefS2 key s2
108      case mr of
109        Just r →
110          do t ← readForCAS r
111             if peekTicket t == old
112              then do _ ← casRef r t new
113                      return ()
114              else return ()
115        Nothing → return ()
```

Figure 5. The algorithm for inserting into an adaptive lock-free collection. Note that the value may be logically present in the map on line 103, even if it is physically absent from `S2`. Also, the call on line 101 assumes that `putIfAbsentS2⊠` is already structured to consider a ⊠ value "absent", and overwrite it in place (a non-structural operation on the container). Another non-structural operation on the container is enabled by `casValS2`, which reads the location of the reference containing the value, and then make a *single* attempt to swap it. Any failure on line 112 means that another operation (remove or put) has superseded the `putIfAbsent` call in question.

state moves from `AB` to `B`. These case statements are total and there are no other state transitions in any function. □

Lemma 5.2. *Constancy. If a `Hybrid` contains an object of type `S2`, then it only contains one such object over its lifetime (as determined by `Eq`, which reflects pointer equality for mutable values).*

Proof. The only call to `newS2` is at line 13. While this may be executed multiple times, the only code path along which the resulting

```
116  -- Remove, implemented similarly to put:
117  remove :: k → Hybrid k v → IO ()
118  remove key r =
119   case! readRef r of
120    A _ → case! tryModify r
121                  (liftS1 (removeS1 key))
122          of Nothing →
123               do initiateTransition r
124                  remove key r
125             Just Nothing  → return ()
126             Just (Just _) → remove key r
127    B    s2 → removeS2 key s2
128    AB _ s2 → putS2 key ☒ s2
```

Figure 6. The algorithm for removing from an adaptive lock-free collection. Note that memory cannot be physically freed until the transition to `B` state is completed. This can be seen in line 128, which represents a remove by an *insertion*, that actually increases physical memory use.

`emptS2` variable escapes is 21 and the corresponding 15. This event is the sole *introduction point* for a fresh `S2` heap value; in all other cases where we access a value of type `S2`, it is obtained via pattern matching on the `HyState`. □

5.2 Correct Set Semantics

Our aim is to show that the adaptive map implements correct abstract set semantics. We refer here to the "dynamic set with dictionary operations", which was defined by Cormen et al. [5]. We define the set of keys H as

$$H = \text{keys}(s1) \cup \text{keys}(s2) \setminus \text{tomb}(s2),$$

where $\text{keys}(s)$ refers to all keys included in the current state of s, and $\text{tomb}(s2)$ is the set of keys that are currently mapped to tombstones in $s2$. Tombstoned keys are logically absent from the abstract state H. Note that we assume that the keys function returns $\{\}$ when its argument is not *currently* present in the hybrid structure; that is, in the `A` state, $\text{keys}(s2) = \{\}$, and similarly for $s1$ in the `B` state.

We must now prove that each of the map operations correctly modify H, given some key k:

- If $k \in H$, `get` returns `Just` its value, and otherwise returns `Nothing`;
- after the `put` and `putIfAbsent` functions on k, k is in the set H;
- the `remove` function removes k from the set H.

As mentioned above, we assume that all operations on `S2` are linearizable; that is, all calls to `putS2`, `putIfAbsentS2`, and `removeS2` are valid linearization points. Furthermore, we assume that `S2` has correct set semantics.

Similarly, we assume that `S1` is correctly implemented. If we apply any of the above operations to a value of type `S1` having key set $keys(s1)$, we should receive a new pure data structure $s1'$ with a key set $keys(s1')$ which differs according to the semantics described above.

Indeed, these assumptions are crucial to the correctness of our algorithms—if the underlying implementations are incorrect, then we can provide no useful guarantees about the contents of the structure or linearizability of the operations upon them.

5.2.1 Linearization Points

Given a concurrent execution history, we define linearization points for the `get`, `put`, `putIfAbsent`, and `remove` operations, and thus map these events onto a sequential execution history listing all linearization points in order. The execution of these linearization points are the moments at which the corresponding changes to the abstract set H, listed above, occur. The linearization points are as follows:

- A `get` is linearized at line 41 if the `readRef` returns a value in the `A` state. In the `B` state, it is linearized at line 44, and in the `AB` state, at line 48.
- If the `readRef` on line 74 returns a value in the `A` state, `put` is linearized at:
  - Line 79 if `tryModify` returns `Nothing`,
  - line 76 if `tryModify` returns `Just Nothing`, and
  - line 81 if `tryModify` returns `Just (Just s2)`.

  In the `B` state, it is linearized at line 82, and at line 83 in the `AB` state.
- If the `readRef` on line 89 returns a value in the `A` state, `putIfAbsent` is linearized at:
  - Line 94 if `tryModify` returns `Nothing`,
  - line 92 if `tryModify` returns `Just Nothing`, and
  - line 97 if `tryModify` returns `Just (Just s2)`.

  In the `B` state, it is linearized at line 98. In the `AB` state, it is at line 101 when the key is not present in `s1`. If the key is present but mapped to ☒ in `s2`, it is linearized at line 112. Otherwise, `putIfAbsent` is a semantic no-op, and is linearized at line 100. Note that this is where we require the ability to take a snapshot of the original pure data structure. The call to `getS1` is a pure, non-blocking call which lets us know definitively whether or not the key is in the map.
- If the `readRef` on line 89 returns a value in the `A` state, `remove` is linearized at:
  - Line 124 if `tryModify` returns `Nothing`,
  - line 120 if `tryModify` returns `Just Nothing`, and
  - line 126 if `tryModify` returns `Just (Just s2)`.

  In the `B` state, it is linearized at line 127. In the `AB` state, it is at line 128.

Above we mark recursive calls as linearization points. As long as non-termination is not a problem (which we address below), these cases reduce to the base cases.

5.2.2 Correct Updates to H

We first prove that the above linearization points are correct for any code path that modifies S2.

Lemma 5.3. *For operations which modify `S2`, lines 44, 48, 82, 83, 98, 101, 112, 127, 128, 81, 97, and 126 are correct linearization points.*

Proof. First, we consider cases where the initial `readRef` observes a hybrid that is already in a `B` or `AB` state. From Lemma 5.2, these operations are all operating on exactly the same `S2` structure. We assume that `S2` operations are linearizable and have correct set semantics; therefore, these are correct linearization points and have the correct effect with respect to H.

Second, we consider cases where the initial `readRef` returns `A`, but the subsequent `tryModify` observes a transitioned state (lines 81, 97, 126). In these lines, there are no live variables that were bound based on the observation of the `A` state. Thus, the behaviour of a call such as `remove key r` on line 126 is identical to the behavior if the initial observation of the `A` state had not occurred.

In the case of line 126 (but not the others), this is a recursive call, which will necessarily recur to the `B` or `AB` case because `tryModify` *already* observed a non-`A` state, since it returned `Just (Just s2)`. □

Next, we prove correct linearization points for any modification of the `S1` structure (i.e. any *op*'s execution where both the initial `readRef` and the subsequent linearization point occurred in state `A`).

Lemma 5.4. *If a call to `tryModify` on lines 76, 92, or 120 returns `Just Nothing`, these lines are correct linearization points. Furthermore, line 41 is a correct linearization point.*

Proof. This case essentially relies on the correctness of a compare-and-swap operation on a single location. By the definition of `tryModify` and `liftS1`, the result of `tryModify` implies that the CAS has succeeded and the structure was in the `A` state both at the call to `readRef` and during the CAS. As noted in Section 4, a successful `tryModify` is an atomic operation, and therefore we treat the line containing the call to `tryModify` as a linearization point. As seen on line 66, the only modification of the state is to apply `f` to `s1`, after which the state of the reference points to a new pure data structure with the modification applied to it. Again, we assume our underlying data structures are correctly implemented, and so these operations modify H according to the set semantics.

For the `get` case (in `A` state), the initial `readRef` is the *only* effectful memory operation, and thus the linearization point. The call to `get1` on line 43 is a pure function, and simply returns the result of reading from `s1`, trivially obeying the correct set semantics by the definition of H.

□

It remains to show that all operations are linearizable when state transitions are triggered.

Lemma 5.5. *If a call to `tryModify` on lines 76, 92, or 120 returns `Nothing`, then linearization and correctness reduce to one of the aforementioned cases for the `AB` or `B` states.*

Proof. In this case we immediately call `initiateTransition`, and recurse upon its completion. This call does not return until the structure is in *at least* the `AB` state, since the `atomicModify` on line 14 does not return until one of lines 21, 23, or 24 has been executed. Since the transition life cycle is monotonic by Lemma 5.1, we reduce to Lemma 5.3.

□

Theorem 5.1. Linearizability - *the algorithms given in Figures 2-3 are linearizable, and have correct set semantics..*

Proof. Because these linearization points are on disjoint code paths, at most one may occur on any call to one of the operations, and so the proof is immediate from Lemmas 5.3, 5.4, and 5.5. □

Theorem 5.2. Lock Freedom - *the algorithm in Figures 2-3 is lock free as specified by definition 2.1.*

Proof. Consider a particular operation on the map, *op*. We show that at each return point of *op* **and** at the point of any recursive call, some operation has performed its linearization point and therefore the system as a whole makes progress. In cases all cases where *op* returns, it has passed its own linearization point. In all case where *op* recurses (except for line 126), it has failed a `tryModify`, which implies that some *other* operation has made progress. In the case of line 126, a state transition occurred during the execution of *op*, and therefore some other operation crossed its linearization point.

□

6. Implementations

Here we describe how we implement the algorithm presented in Section 4. We implement two concrete data structures using two compilers. By demonstrating our approach in Java as well as Haskell we are able to confirm that (1) the approach does not depend on idiosyncrasies of GHC Haskell, and (2) the hybrid approach has advantages *even* when compared with recognized, well-tuned scalable structures (`java.util.concurrent`).

6.1 Bags in Haskell

We begin with unordered lists, or bags, which are often used for work-queues and other producer/consumer communication scenarios where the order of the messages does not matter.

6.1.1 Pure-in-a-box Bag

Our pure bag implementation is extremely simple, requiring less than forty lines in total. The representation is a mutable container containing a pure list:

```
type PureBag a = IORef [a]
```

We support atomic addition and deletion via ticketed compare-and-swap operations on the reference.

6.1.2 Scalable Bag

The scalable bag implementation is necessarily more complex than `PureBag`. It uses thread-local storage to manage a mutable vector of sub-bags. Each thread is assigned an index into the vector, which has a length equal to the number of OS threads used by GHC.

```
type ScalableBag a = IOVector [a]
```

For a thread to insert into the bag, it must first look up its thread-local index modulo the length of the vector. It can then simply write to that index without fear of contention. Removal is somewhat more complicated; since absence of data at the thread's index does not imply absence of data in the whole bag, the removal operation must make one pass through the entire vector, searching for an index to remove from[5].

Using the ticketed interface described in Section 3, we provide per-element compare-and-swap operations on Haskell's mutable vector type. We also pad the vector to avoid false sharing. More complicated (and optimized) bag designs exist [21], but this design is simple and achieves much better scaling than pure-in-a-box.

6.1.3 Hybrid Bag

The hybrid bag is a combination of `PureBag` and `ScalableBag`:

```
type HybridBag a = IORef (HybridState a)
data HybridState a = A  [a]
                   | AB [a] (ScalableBag a)
                   | B      (ScalableBag a)
```

Since bags are a simpler data structure than maps, offering few strong guarantees, the management of the transitional `AB` state is made much easier than the presentation in Section 4. Rather than carefully coordinating the interaction between the copy thread and outside consumers, all reads and writes are simply forwarded to the `ScalableBag` (which does relax the definition of when transient empty can be observed). Beyond this small reduction in complexity,

[5] Empty tests must have the expected semantics in sequential executions. However, there's a standard problem with the any-empty test in a parallel region—with these bags a data structure can be observed as logically empty even if there was no physical moment in time when all slots in the bag were empty.

the code is straightforward and quite similar to that of that of Figures 2-3.

An unfortunate deficiency of this approach is the introduction of an extra level of indirection—-all accesses to the data structure must first dereference its `IORef`, then follow the pointer to the `HybridState`, and finally access the underlying structure. However, in practice the effect of this indirection is quite small, as can be seen in Section 7.

6.2 Maps in Java

In order to test adaptive data structures in multiple language ecosystems (and verify that our approach is not GHC-specific), we have also implemented an adaptive map type in Java. For the scalable component, we use the `ConcurrentSkipListMap` implementation from `java.util.concurrent` package which is based on work of Fraser and Harris in [6] and is known to scale well [10].

Likewise, we use an existing library for pure structures. The `PCollections`[6] library provides Okasaki-style persistent data structures akin to those present in Haskell, enabling a direct implementation of pure-in-a-box lock-free data. For our implementation, we use its `IntTreePMap` class, which gives us a persistent map from integer keys to non-null values. This implementation is, in fact, based on GHC's `Data.Map` [1, 16] .

We also require an equivalent to Haskell's `IORef` in Java. The standard `AtomicReference` class is used for this purpose, providing us with the ability to coordinate access to the object in a multi-threaded setting using the `compareAndSet` method.

Using these building blocks, the implementation of the hybrid data structure is straightforward. There are a few minor language issues which require some special treatment, however. Haskell's union types are replaced in the Java implementation by a product type `(state,s1,s2)`, where `s1` and `s2` are nullable references, and state is an enum with three values representing the `A`, `AB`, and `B` states. (Using inheritance to encode unions would also be a reasonable approach.)

This object, (analogous to `HyState`), is kept in an AtomicReference in order to manage concurrent attempts to transition the data structure and retain a tight correspondence to the algorithm of Section 4. As with the Haskell implementation, any updates to the state happen only via this atomic reference—the fields of the `(state,s1,s2)` record are not mutated. Thus creating a new hybrid object begins by creating a new record with an empty pure data structure, and a null pointer in the `ConcurrentSkipListMap` field. Method calls on the hybrid check the state, and dispatch to the appropriate implementation in the `s1` or `s2` field, again closely following the pseudocode in Section 4.

7. Evaluation

The standard approach for evaluating concurrent data structures is to stress-test them under contention and observe their maximum throughput (operations per second) under varying numbers of threads and mixes of operations. Indeed, we use some of the concurrent bag workloads from Sundell et al. [21]. However, what we are really interested in is not raw scalability. Rather, it is the ability of hybrid structures to replace pure-in-a-box ones where contention is unknown in advance. To this end we perform a parameter study to examine workloads on a *nested* data structure that has both *hot* and *cold* inner structures.

Evaluation platform We evaluate on a 2.6GHz, 16-core, dual socket Intel Xeon E5-2670 platform with 32GB of memory running Ubuntu 12.04. Because we perform a large parameter study, we use a 16-node cluster rather than a single machine to spread benchmark load, but each machine has an identical hardware and software configurations. Haskell code is compiled on GHC 7.8.3 and 7.10 (release candidate 2), with similar results. Due to space constraints we report the GHC 7.10 results only. These results are slightly better overall because of GHC 7.10's new support for generating *inline* code sequences for atomic operations (whereas previous versions made an out-of-line primop call).

Benchmarking methodology In Haskell, we use the *Criterion* statistical benchmarking package. Criterion estimates the time required for very short computations by varying the number of iterations performed and computing a linear regression between the number of iterations and time required. Thus Criterion estimates the expected marginal cost of adding *one more* operation when already performing many; i.e., cache warmed, etc. Each data point in each of our charts is computed through such a regression. Figure 7 illustrates the output of Criterion for one such regression.

One reason we discuss Criterion here is that there is a complication when using Criterion for running *parallel* benchmarks. In a typical sequential benchmark, the only constant overheads are the start/end timing code sequences themselves. But for a parallel benchmark, there is a *fork/join* step at the beginning and end of each run, respectively. Indeed, on the GHC 7.8.3 Haskell runtime it takes approximately 30,000 cycles between forking IO threads and the moment when all of them are actually running[7].

Normally, this fork/join overhead limits how small of a test can be measured. *However*, in cases where the data structure operations are $O(1)$, it is possible to use a different approach and allow criterion to choose the iteration sizes for a batch, but only fork one set of worker threads *for that whole batch*. This amounts to using Criterion's `Benchmarkable` constructor directly, which constructs a benchmark out of an `Int64 →IO ()` function:

```
Benchmarkable
    (λnum → forkJoin threads
              (doWork (num `quot` threads)))
```

This variant splits the total work across threads, e.g. timing the amount of real time required to perform N benchmark operations across P threads. We report this form of measurement where possible, estimating the overhead of a *single* data structure operation in a concurrent context.

Parameters We fork all worker threads on a designated OS thread (forkOn), and we use the `-qa` RTS option to allow the GHC runtime to pin OS threads to processor cores. To get good measurements of the scalability of the data structures themselves—without being overwhelmed by parallel garbage collection overheads in GHC—we change the GC parameters in ways we will specify in the individual benchmark discussions.

Also, our algorithm has a parameter as well—the hybrid data structure is parameterized by $casTries$—how many CAS failures in row are attempted before choosing to initiate transition to a scalable structure. This parameter is implicit in the behavior of `tryModify` from the algorithm pseudocode. It is good to keep this value low, and we study which setting to use in the evaluation. In each benchmark, we performed a parameter study, varying $threads \in [1, 16]$, $casTries \in \{1, 2, 5, 10\}$, and in the nested data benchmarks we additionally vary $hotRatio \in [0.0, 0.1, ..., 1, 0]$ (with three variants, this amounts to 2,112 runs per benchmark); in our figures, we pick a value of $casTries$ and show representative samples of $hotRatio$.

[6] http://pcollections.org

[7] Estimated by threadscope for the four core case. It can grow worse with more cores.

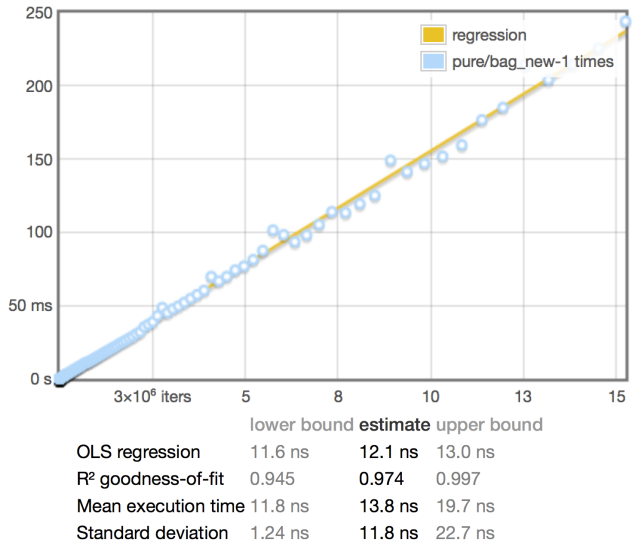

Figure 7. Example Criterion benchmark run measuring *batches* of varying numbers of iterations of the benchmark in question. This particular benchmark measures the time to create a new list inside an `IORef`, which serves as our pure-in-a-box bag container. The X axis is millions of iterations, whereas the Y axis measures time to perform that many iterations. Some variation is expected due to runtime system effects. Every point in the plots other than this one corresponds to the "OLS regression" figure computed by an experiment such as the above.

Java methodology Our Java benchmarks largely follow the same pattern as their Haskell counterparts. All reported runs are from the same evaluation cluster, using Java 1.8.

As we were not able to get a Criterion-like package working in Java, we use the simpler approach of running a constant number of iterations on each trial (100), and computing the mean time per iteration. We run our Java benchmarks using runtime options `-Xms16g -Xmx24g -d64`. Concurrency is achieved using the Java standard library's threading capabilities (with countdown latches for thread joins). The code that implements `tryModify` (CAS retry loop) is similarly parameterized by $casTries$.

7.1 Haskell: Pure to Scalable Bags

Here we evaluate the bag implementation described in Section 6.1. We refer to the "Pure", "Scalable", and "Hybrid" variants of the implementation in all results. First, we measure a *single* bag accessed by all threads to affirm that the operations have the expected relative advantages. Figure 8 shows the cost of the allocating a single new bag for each implementation variant. This cost is $O(1)$ for pure and hybrid, and does *not* respond to number of threads, which is as expected. The scalable bag, however, uses a vector of thread local storage, whose size must vary with the number of threads in the system. Therefore, not only does does it require over twice the time as Pure in a single-threaded setting (`+RTS -N1`), but the overhead of vector allocation grows as the number of threads increase. This penalty is worth paying when we use the scalable map in a multi-threaded program under contention, but it is unnecessary in uncontended settings.

Insert bench After transitioning to state B, the Hybrid bag still has an extra pointer indirection and extra branches relative to a straight scalable version. As a result, we expect it to have some additional overhead. This overhead can be seen in a simple insertion contention benchmark (Figure 9), measuring the cost of all

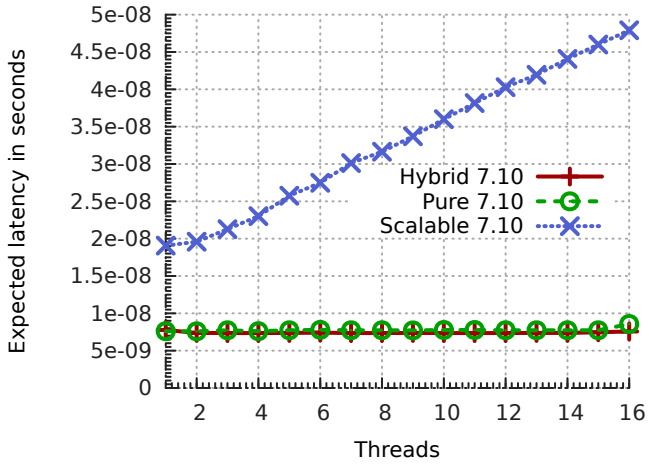

Figure 8. Cost of creating a new bag, in milliseconds. The cost of creating both pure-in-a-box and hybrid bags is low, while the scalable version pays an overhead relative to the number of threads in the system.

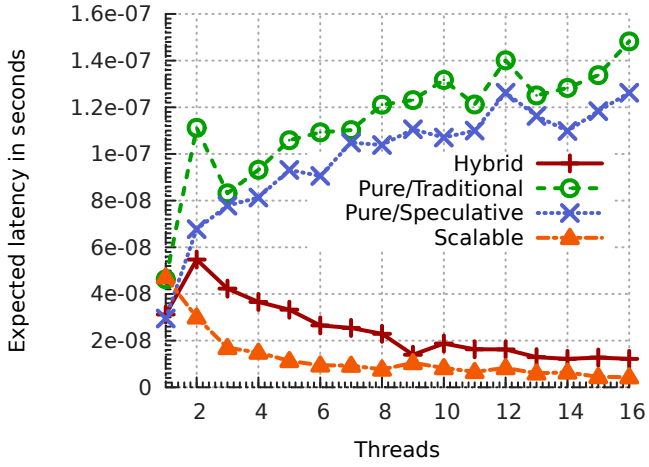

Figure 9. Cost of inserting into a single bag shared between threads, under high contention. The X axis tracks the number of threads simultaneously acting on the shared bag, and the Y axis measures average time *between operations committing* (latency). Lower is better.

threads inserting into the bag. We see that at one thread, the hybrid structure has the same performance as its pure counterpart, while the scalable version is actually 51% slower. Once more threads enter the picture, though, the scalable structure quickly overtakes the pure version, and the hybrid follows it—it has triggered a transition, and then gains the performance benefits of the more complex data structure.

Also, in Figure 9, we include one extra line, Pure/Traditional, which shows the `atomicModifyIORef'` approach, to contrast it with the speculative approach based on ticketed-CAS which we propose in this paper (Pure/Speculative, elsewhere just "Pure"). Here we confirm that not only do we need ticketed CAS to preserve lock-freedom in our algorithm, but it also typically offers a performance advantage for code that performs many small modifications to a shared structure under contention.

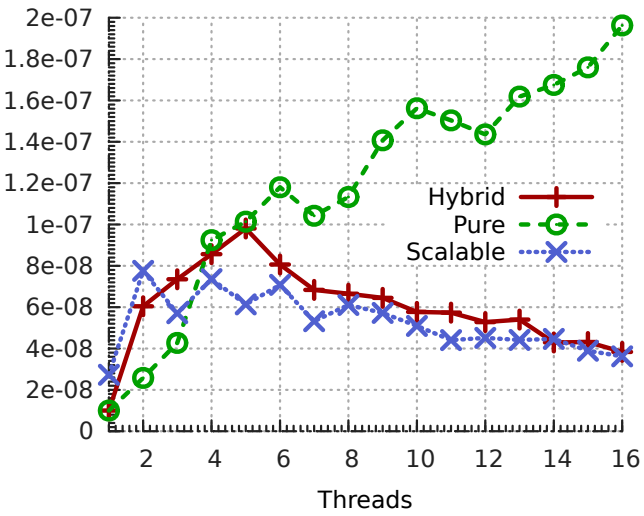

Figure 10. Similar to Figure 9, Y axis in seconds: the cost of operating on a single shared bag. This shows a random (50/50) mix of insert and remove operations, starting with an empty bag, as in Sundell et al. [21]. In this figure, the hybrid bag triggers a transition after a single failed CAS.

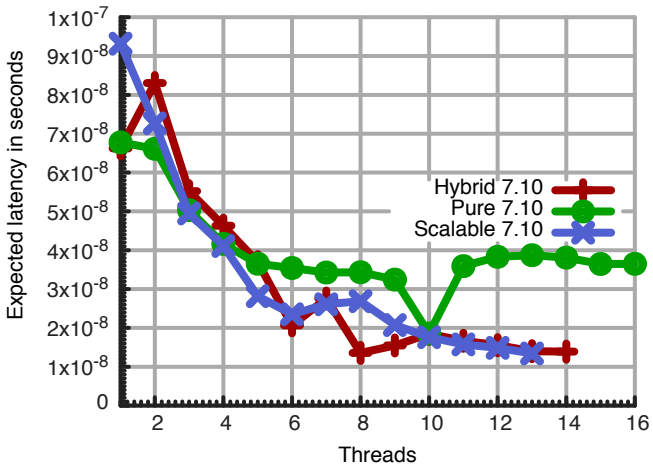

Figure 11. Skewed hot-key/cold-key benchmark, inserting into a nested array of bags. Here we show a balanced workload ($hotRatio = 0.5$). At the hot and cold ends of the spectrum, the benchmark reduces to stressing the individual operations as covered by Figures 8 and 9.

Insert/remove bench Figure 10 shows another contention benchmark, but with a different mix of operations. Here threads randomly insert or delete from a single, shared bag. A large vector of random bits is precomputed for these benchmarks before Criterion begins its measurements. In this figure, we see a similar pattern. Pure and Hybrid begin with the same performance—nearly six times as fast as scalable! But scalable overtakes Pure as threads increase. Further, as contention increases, a transition is triggered and soon the hybrid's performance approaches that of the scalable bag.

Setting casTries Both of these contention benchmarks can be affected by the setting of the $casTries$ parameter, which determines how quickly transition engages, and therefore how many elements must be handled by the copy thread when contention does occur. However, because the benchmarks in this section represent maximum contention, transition is very fast. We pick $casTries = 1$ as the setting for this section, but the other settings yield similar results.

The GC problem Unfortunately, Haskell does not yet have a scalable memory management system, in spite of substantial effort in this direction [14]. GHC's current garbage collector is parallel (load-balancing), but not concurrent, so it performs stop-the-world collection even on minor collections. This effect is especially pronounced on microbenchmarks of the kind described in this section. Furthermore, "scalable bag" triggers a performance weak point in current versions of GHC where the GC workload does not successfully load-balance consistently, yielding some runs where all GC work is serialized, and others with reasonable balance. To eliminate this source of nondeterminism, we have arranged to completely *avoid collection* during the benchmarks in question. We do this simply by setting the heap to one generation, with a size equal to 30GB (out of a 32GB RAM).

Nested data collections To test Hybrid's ability to withstand mixed contention within one benchmark, we perform a nested version of the insertion benchmark on an *array of bags* (Figure 11). Here, each thread randomly flips a coin (again with precomputed randomness) and either inserts into a single shared "hot" bag in a designated vector slot, or inserts into a randomly chosen slot containing a "cold" bag. The vector is sufficiently large that cold bags are unlikely to be contended (100 slots). Here we see again that pure bags suffer at more than one thread, while scalable bags suffer at one thread, but hybrids can function well in both categories. This demonstrates the hypothesized benefit of a dynamically adaptive data structure: the amount of contention cannot be known statically, but they still handle both contention regimes, without giving up lock-freedom.

Again, contention detection in all cases is based on the number of failed CASs per operation ($casTries$). This presents an important tuning parameter, representing the hybrid's level of sensitivity to contentions. Do we follow a hair-trigger policy of transitioning after a single failed CAS attempt, or should we be more tolerant in the hope that future modifications will be successful? In this benchmark we show results from $casTries = 1$.

7.2 Java: Pure to Scalable Maps

In this section we evaluate a nested combination of concurrent Map data structures, as described in Section 6.2. Note that this is an independent evaluation from the Haskell one—we do not *compare* Java and Haskell, because we implement different data structures: two instances of the hybrid approach[8]. Here we test a nested combination of Maps:

- A `ConcurrentSkipListMap`, forming the outer, definitely contended, structure
- An inner map, varying between the $\{pure, scalable, hybrid\}$ variants

Like in the Haskell nested-data benchmark, we perform a number of insert operations on $(keyOuter, keyInner, val)$, divided equally among the worker threads. Once again, we flip a coin on

[8] One reason for choosing two data structures rather than one is that Haskell currently lacks an efficient lock-free map implementation. We have implemented a lock-free concurrent skiplist similar to Java's as part of the LVish parallel-programming library, but in the course of this work we have found that it is not yet scalable enough to be worth switching to over pure-in-a-box *if* the speculative CAS approach is used for modifying the box (Section 3).

Table 1. Mean time in micro-seconds to allocate and initialize one new object of the specified type in Java. (Estimated by average time to create a batch of one million objects consecutively.)

Pure	Scalable	Hybrid
0.029	0.064	0.05

each operation, choosing either, with probability $hotRatio$, a designated hot outer-key or a uniform-random cold-key. Outer keys are chosen from $[1, 10^6]$, and inner keys are random 32-bit numbers. For a given $keyOuter$, if an entry does not already exist, it is created, stressing the constructor method for the inner map type. Table 1 shows the cost of these "new" operations. In line with our expectations, in the Java context creating an empty `ConcurrentSkipListMap` is almost two times as expensive as creating a pure one. Further, creating Hybrid objects is about 20% cheaper than creating `ConcurrentSkipListMaps`.

As in Section 7.1, we perform a parameter study, varying $threads$, $casTries$, and $hotRatio$. Figure 12 shows representative results from this study. In the "cold" extreme, with very little contention on inner maps, all variants perform remarkably similarly, with slightly better performance for Pure at low thread counts, and slightly better performance for scalable and hybrid at higher thread counts. In the hot extreme, with all inserts going to a single inner map, Scalable of course performs better, and Pure degrades, while Hybrid successfully tracks the scalable variant. Here, the benefit of Hybrid over Scalable is that it performs better in the one-thread uncontended case: heavy access on the hot key, but from only a single thread.[9] Finally, in the $hotRatio = 0.5$ case, we see a blend of these outcomes, and Hybrid has a small advantage over other variants, coping well with both one very hot key, and lots of keys that remain cold (and stay in their pure-in-a-box state).

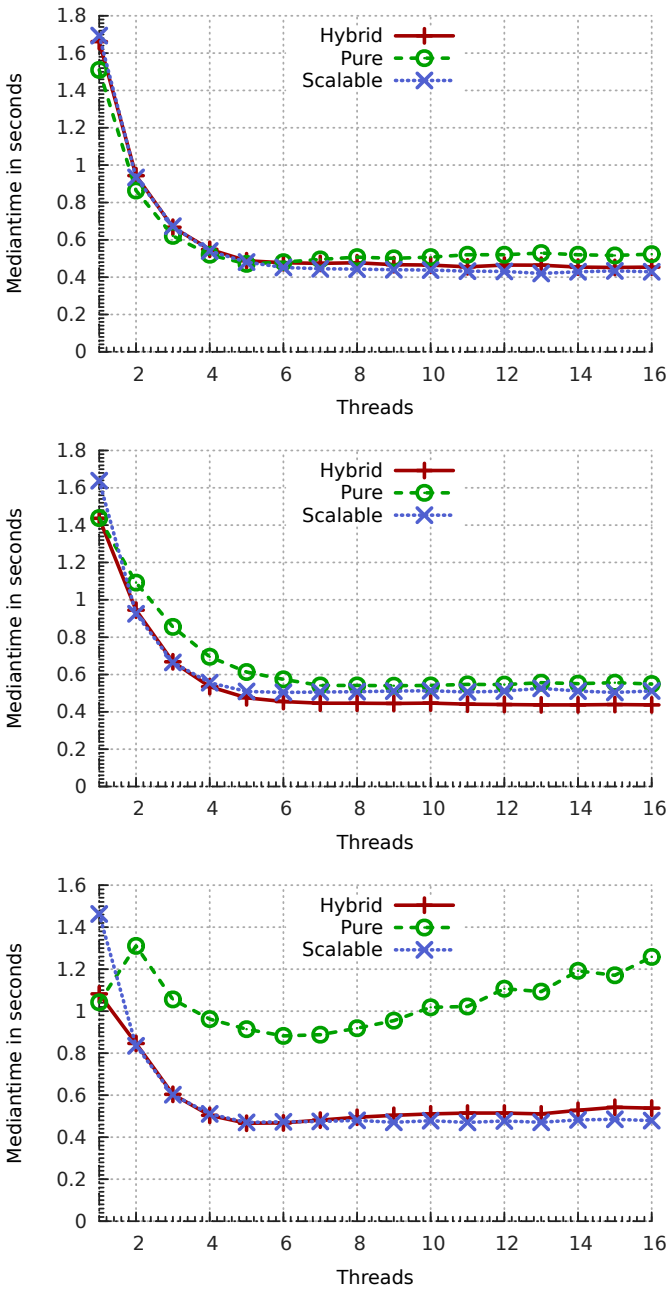

Figure 12. Nested map-of-maps insertion benchmarks in Java; time for 1M inserts. The top figure has $hotRatio = 0.0$, the middle $hotRatio = 0.5$, and the bottom $hotRatio = 1.0$. All other points interpolate smoothly between these three. Further, the benchmark is insensitive to $casTries$ until $casTries = 10$, at which point Hybrid mimics the poor 2-thread behavior of Pure, because it does not transition early enough.

8. Related Work

The idea of swapping out the representation of a data structure at runtime is an old one, and has appeared in a variety of contexts.

Data structure swapping Pypy is a Python virtual machine and tracing JIT which provides *storage strategies* [3]. Storage strategies enable homogeneously typed collections to specialize on their element type; an array of integers can thus unbox its contents, providing significant performance benefits. When a collection dehomogenizes—that is, when a value of a different type is inserted—the structure must dynamically switch to a generic, boxed representation.

The Coco system [23] provides application-level optimizations by dynamically switching between different implementations of standard Java container types. For example, a linked list can be swapped out for an array-backed implementation when many calls to `get` are made at a particular index, thus improving algorithmic complexity.

Dynamically responding to contention The idea of dynamically detecting and responding to contention is also well known in the literature surrounding multithreaded applications. It can be found in such areas as databases [4], low-level caching algorithms [24], and also previous work in concurrent data structures and algorithms [22]. Particularly related are *elimination trees* of Shavit and Touitou [20] which, like our adaptive data, spread out accesses to additional, dynamically created memory locations, preventing contention on a single reference.

Compared with these previous approaches, our work combines the two ideas—implementation swapping and action in response to contention—to a new end: pure-to-scalable transitions.

[9] We also see a surprising non-monotonicity in the behavior of Pure—results get *worse* at two threads *and then better again*. We would expect them to degrade monotonically with more threads. We have searched for the cause of this and not yet found it; the complexity and dynamic optimizations of the JVM make it difficult to track. This effect emerges gradually with increasing $hotRatio$, first appearing visible to the eye in $hotRatio = 0.7$, and then the two-thread time becoming worse than one thread at $hotRatio = 0.8$, and worse yet at 0.9 and 1.0.

9. Future Work and Conclusions

Choosing between a lightweight persistent data structure and a more complicated, but scalable, lock-free version shouldn't be a static decision. Indeed, in some scenarios, the optimal choice cannot be made statically. We show that it is possible to gain the benefits of both alternatives by transforming the data structure's internal representation in the event of contention. Furthermore, this strategy is applicable in such disparate languages as Haskell and Java.

In this work we only support monotonic changes from pure to scalable. In the future, we plan to explore reverse transitions when contenion abates—data structures which can return to a pure representation after "coming to rest"—which would restore cheap snapshots. Further, there are opportunities for applying elimination trees together with pure-in-a-box data structures. These have the potential to broadly improve concurrent programming in functional languages.

Acknowledgments

This work was supported by NSF grants CCF-1218375 and CCF-1453508.

References

[1] S. Adams. Functional pearls efficient setsa balancing act. *Journal of functional programming*, 3(04):553–561, 1993.

[2] S. Bauman, C. F. Bolz, R. Hirschfeld, V. Krilichev, T. Pape, J. Siek, and S. Tobin-Hochstadt. Pycket: A tracing jit for a functional language.

[3] C. F. Bolz, L. Diekmann, and L. Tratt. Storage strategies for collections in dynamically typed languages. *SIGPLAN Not.*, 48(10):167–182, Oct. 2013. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/2544173.2509531`.

[4] J. Cieslewicz, K. A. Ross, K. Satsumi, Y. Ye, and Q. Processing. Automatic contention detection and amelioration for data-intensive operations. In *In SIGMOD*, 2010.

[5] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN 0070131511.

[6] K. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, 2004.

[7] A. Gill and G. Hutton. The worker/wrapper transformation. *J. Funct. Program.*, 19(2):227–251, Mar. 2009. ISSN 0956-7968. . URL `http://dx.doi.org/10.1017/S0956796809007175`.

[8] J. N. Gray, R. A. Lorie, and G. R. Putzolu. Granularity of locks in a shared data base. In *Proceedings of the 1st International Conference on Very Large Data Bases*, VLDB '75, pages 428–451, New York, NY, USA, 1975. ACM. ISBN 978-1-4503-3920-9. . URL `http://doi.acm.org/10.1145/1282480.1282513`.

[9] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 206–215, New York, NY, USA, 2004. ACM. ISBN 1-58113-840-7. . URL `http://doi.acm.org/10.1145/1007912.1007944`.

[10] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems (OPODIS)*. Citeseer, 2006.

[11] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. ISSN 0164-0925. . URL `http://doi.acm.org/10.1145/78969.78972`.

[12] M. Letia, N. M. Preguiça, and M. Shapiro. Crdts: Consistency without concurrency control. *CoRR*, abs/0907.0929, 2009. URL `http://arxiv.org/abs/0907.0929`.

[13] S. Marlow and S. P. Jones. Making a fast curry: Push/enter vs. eval/apply for higher-order languages. *J. Funct. Program.*, 16(4-5):415–449, July 2006. ISSN 0956-7968. . URL `http://dx.doi.org/10.1017/S0956796806005995`.

[14] S. Marlow and S. Peyton Jones. Multicore garbage collection with local heaps. In *ACM SIGPLAN Notices*, volume 46, pages 21–32. ACM, 2011.

[15] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore haskell. *SIGPLAN Not.*, 44(9):65–78, Aug. 2009. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/1631687.1596563`.

[16] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM journal on Computing*, 2(1):33–43, 1973.

[17] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 295–308, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3. . URL `http://doi.acm.org/10.1145/237721.237794`.

[18] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, May 2006. ISSN 0004-5411. . URL `http://doi.acm.org/10.1145/1147954.1147958`.

[19] N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54:76–84, Mar. 2011. ISSN 0001-0782. . URL `http://doi.acm.org/10.1145/1897852.1897873`.

[20] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks: Preliminary version. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 54–63, New York, NY, USA, 1995. ACM. ISBN 0-89791-717-0. . URL `http://doi.acm.org/10.1145/215399.215419`.

[21] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. A lock-free algorithm for concurrent bags. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 335–344, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0743-7. . URL `http://doi.acm.org/10.1145/1989493.1989550`.

[22] G. Taubenfeld. Contention-sensitive data structures and algorithms. In *Proceedings of the 23rd International Conference on Distributed Computing*, DISC'09, pages 157–171, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 3-642-04354-2, 978-3-642-04354-3. URL `http://dl.acm.org/citation.cfm?id=1813164.1813186`.

[23] G. Xu. Coco: Sound and adaptive replacement of java collections. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 1–26, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-39037-1. . URL `http://dx.doi.org/10.1007/978-3-642-39038-8_1`.

[24] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 27–38, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0687-4. . URL `http://doi.acm.org/10.1145/1952682.1952688`.