A Scalable Non-Blocking Concurrent Hash Table Implementation with Incremental Rehashing

David R. Martin Richard C. Davis

December 15, 1997

Abstract

Highly concurrent systems often require high-throughput shared data structures to achieve high performance. Traditional lock-based implementations suffer from the problems of convoying, priority inversion, and deadlock. Furthermore, locks introduce overhead in the common case. Non-blocking concurrent data structures have been proposed to solve all of these problems, as they allow concurrent atomic updates without using locks.

There exist general methods to transform any lock-based data structure into a lock-free, non-blocking version. However, these methods invariably yield low performance implementations. Common practice is to design a custom lock-free version of the data structure. We present a fully non-blocking concurrent hash table implementation with incremental rehashing. Our fault model allows a thread to suspend or die at any point in our code without compromising the consistency of the table. Furthermore, our design is portable and is shown to achieve high throughput on workloads with high contention.

Two implementation notes are of interest. First, we use open addressing rather than the traditional bucket-and-chain data structure. This design choice was critical in enabling a relatively simple design and proving it correct. Second, our strong fault model requires garbage collection. Even a weaker fault model would benefit greatly from garbage collection, as it greatly simplifies analysis.

1 Introduction

Highly concurrent systems often require high throughput of certain data structures. For example, task queues, priority queues, and hash tables are ubiquitous data structures that are often the target of contention. Traditional locks can be used to enforce mutual exclusion and implement atomic operations on concurrent data structures. However, lock-based implementations have many problems.

First, the optimal granularity of locking depends on the dynamic level of concurrency. If the level of concurrency is very low, then a single root-level lock may be sufficient. If the level is very high, then fine-grain locking would be more appropriate. Fine-grained locking trades space for supporting more concurrency. Even if this trade-off is acceptable, the overhead of acquiring and releasing the lock is always present and could be significant.

Apart from overhead, there are at least three other significant problems associated with lock-based synchronization [8]:

1. **Convoying** is the queuing of threads behind a lock that is currently held by another thread. If convoys form, then they can persist after the lock has been passed. This creates hot-spots and reduces concurrency.

2. **Priority inversion** arises when a low-priority thread holds a lock that is required by a high-priority thread. The priority of the high-priority thread is effectively reduced to the low-priority thread's priority.

3. **Deadlock** arises when a thread is blocked on a lock that will never be released. This may occur because locks were not acquired in a partial order; it may occur because a thread that acquired a lock did not release it (i.e. a bug); or it may occur because a thread died while holding a lock, and the thread package was not able to release it.

Some of these problems can be solved by better thread and synchronization implementations. For example, priority inversion could be fixed with priority inheritance. Some deadlock cases could be eliminated by better thread cleanup or deadlock analysis tools. However, No amount of thread package re-engineering could fix convoying or all deadlock cases. *Lock-free* synchronization can solve all of these problems. Furthermore, it has the potential to be faster in the common case, as the overhead of acquiring and releasing locks is eliminated.

The benefits of lock-free synchronization are clear, but they do come at a cost. Lock-free concurrent programming is not as well understood as lock-based concurrent programming. Programming languages such as Modula-3 [5] and Java [2] have made lock-based programming quite painless as far as correctness is concerned. Consequently, the technique of lock-free synchronization is relevant only when complexity can be traded for performance. Highly-concurrent systems such as databases and web servers may we willing to make this trade-off.

There are two classes of lock-free data structures. A lock-free implementation can be either *non-blocking* or *wait-free*. According to [13],

- A data structure is *non-blocking* if some thread will always complete its operation in finite time, regardless of the actions of other threads.

- A data structure is *wait-free* if every non-faulty thread is guaranteed to complete its operation in finite time.

Clearly the wait-free property is more desirable, but it usually requires much overhead. Non-blocking implementations are often sufficient; if supplemented with exponential back-off, they can be made as robust as a wait-free implementation.

2 Background

There are well-known methods for transforming any lock-based data structure implementation into a wait-free implementation [7]. These methods invariably yield low performance implementations, since they are oblivious to *application-specific concurrency*. Without knowledge about how a data structure is used by an application, an automatic method must be conservative with respect to what sort of concurrency is possible by the application. Consequently, these methods result in much copying that may not be strictly necessary.

For example, there is a trivial way to modify any data structure to enable lock-free atomic updates: Add a root pointer to the data structure; an atomic operation is performed by first copying the entire data structure, making a modification, and swapping the root pointer. It is trivial to prove that this method is correct, but it suffers from two major problems. First, it is usually infeasible for every operation to copy the entire data structure. Second it does not allow concurrent operations.

Since memory allocation and copying are expensive operations on computers, any extraneous allocation or copying could cripple a data structure's performance. Consequently, common practice is to design lock-free concurrent data structures manually, so that application-specific concurrency can be leveraged to eliminate unnecessary copying.

Designing and coding such data structures is quite difficult. There is no programming language support for lock-free structures. Furthermore, highly concurrent systems are highly non-deterministic, and as a result, difficult to debug. These problems can be overcome by following some strict design rules and maintaining certain invariants. A design not proven correct has very little chance of being correct.

A building block for atomicity is required. Assuming an atomic memory primitive such as *compare-and-swap* (as provided by the x86 and Sparc v9 [14] architectures) or *load-linked/store-conditional* (as provided by the MIPS [10] and Alpha [12] architectures), one can build *restartable atomic sequences* [3]. For the purposes of building lock-free structures, any update must be structured as a restartable atomic sequence of code such that the update either completes atomically or can be safely aborted at any instruction in the sequence.

An additional constraint on the instruction set architecture is that in order to more safely provide atomicity, the atomic memory primitive must operate on a data type that is larger than the address type. This leaves room for a version, so that both a pointer and an associated version can be updated atomically. With a 32-bit pointer and a 32-bit version, this is a satisfactory solution to the *ABA problem* [13].

An important invariant to maintain is one of *linearizability* [6]. A concurrent object is linearizable if it appears to all threads as if the operations of all threads were executed in some linear sequence. This definition is nearly identical to the definition of *sequential consistency* in a multiprocessor system.

Memory allocation, and especially reclamation, is very difficult in a concurrent lock-free system. If a thread can suspend at any time for any amount of time with pointers stored in private variables, then objects referenced by those pointers may not be deleted elsewhere in the program. A weaker fault

model that prohibits threads from evaporating enables *reference counting* as a solution. Effectively, the data structure performs its own lock-free allocation and reference counting garbage collection. This solution is attempted in [13] in a lock-free linked-list algorithm. A stronger fault model that allows thread evaporation precludes reference counting as a viable solution, since reference counts could not be maintained consistently. It is a direct consequence of our fault model that a language with *garbage collection* is a requirement. The ideal system for lock-free design would include a lock-free concurrent allocator and garbage collection.

The literature contains several implementations of simple lock-free concurrent data structures such as stacks and queues [11]. More interesting data structures such as lists [13], binary trees [13], and priority queues [9] can be found. It is not clear, however, if the designs for these more complex data structures are feasible, as experimental results are either not emphasized or not present. This paper presents the design, implementation, and experimental analysis of a lock-free concurrent hash table.

3 Design

Our hash table borrows many of the general ideas mentioned in section 2, and takes advantage of some interesting properties of open addressed hash tables and garbage collection. This section details our approach to correctness, fault-tolerance, and scalability.

3.1 General Lock-Free Design

The restartable atomic sequence is an effective technique for performing non-blocking atomic operations on a shared object. It allows one to atomically move an object between consistent states, where the transition cannot be accomplished directly with a 64b atomic memory primitive. A level of indirection is introduced to enable arbitrarily large atomic updates. In this context, such an atomic update has this general structure:

1. Save a copy of the root pointer in a local variable $Root_{old}$.
2. Create an deep copy of all that $Root_{old}$ points to, and save the new root pointer in a local variable $Root_{new}$.
3. Modify the local replica ($Root_{new}$) as needed for the update.
4. Perform a compare-and-swap (CAS) on the real root, providing the old value as $Root_{old}$, and the new value as $Root_{new}$. This CAS will fail if the root pointer is no longer equal to $Root_{old}$ (i.e. if there has been any intervening update since this thread copied the root pointer into $Root_{old}$ in Step 1).

Because of the *ABA problem*, any pointer involved in such a CAS operation should be tagged with a version. The ABA problem arises when pointers are not tagged. Imagine that between steps 1

and 4 that the thread is suspended. While the thread is suspended, there are two atomic updates by other threads. The first such update is guaranteed to change the value of the root pointer (from A to B), since the new copy of the data structure is obtained from the memory allocator. However, the old copy that is swapped out by the CAS may be reclaimed by the allocator and re-allocated in the second intervening atomic update (back to A). When the original thread resumes, there have been two intervening atomic updates, but the root pointer has the same value as when the thread was suspended. The CAS should clearly fail in this case. The solution is to attach a version to each susceptible pointer, which is incremented by each atomic update. Though this does not provably remove the ABA problem, it clearly reduces the probability of an ABA occurrence a great deal.

This technique for atomic updates can be applied to the object as a whole, and though the implementation would be easily proved correct, it is often far too expensive to create a copy of the entire object for each atomic update. The solution is to apply the technique at a finer granularity, taking advantage of application-specific concurrency to reduce the amount of copying and allocation. It is only with intimate knowledge of the structure of concurrency in the object that such optimizations can be made, which is why automatic methods do not yield high performance implementations on large concurrent objects.

We extend the method of tagged pointers slightly by adding additional state to a pointer. Not all 64 bits swapped with 64b atomic memory primitives need be used for a pointer and a tag. Since 32b pointers are currently sufficient for many applications, and since 32b versions are far more than is needed, we can use some of the version bits for other purposes. Section 4 details these purposes.

3.2 Phases of Operation

The notion of phases is important in our design and our analysis of correctness. For example, we wish to perform concurrent, incremental rehashing of the table. This involves moving from a state where there is no rehashing, to a state where threads perform rehashing. The details are presented in Section 4.6. The general mechanism we employ is an extension of the tagged pointer technique, where we use one bit to make sub-structures immutable. This bit is initially zero, and *all* normal table modifications use CAS and provide an old value where this bit is set to zero. Thus, when the bit is set to one, any CAS attempts made by later table operations are guaranteed to fail. Since CAS is the only way to update that sub-structure, it has been made immutable. This allows us to freeze the state of pieces of the data structure so that we can reason about it's contents during concurrent table updates. Furthermore, there is never a need to copy an immutable structure. Without this guarantee of immutability, optimizations that remove copying are more difficult to analyze.

3.3 Synchronization and Portability

The *64-bit compare-and-swap* instruction provides the functionality needed to implement restartable atomic sequences with tagged pointers. We assume this operation is available. Since it can be synthesized from load-linked/store-conditional, it is trivial to implement CAS on all important architectures. If a lock-free data structure is to be portable, one must build upon commonplace

atomic memory primitives.

3.4 Fault Model & Garbage Collection

Because we are targeting this hash table at complex, highly concurrent applications with dynamic workloads, we assume that any thread operating on the table can suspend at any time for any amount of time, or even evaporate at any time. The use of restartable atomic sequences and the absence of locks ensures that no suspended thread can prevent an active thread from making progress. The one resource that a suspended thread does reserve, however, is memory.

Section 2 outlined a common solution to the problem of memory allocation in a lock-free object. If one can assume that threads do not evaporate, then reference counting can be used to determine when a piece of memory may be reclaimed. A lock-free object can thus perform its own memory allocation and reclamation using reference counts. If, however, a thread never returns once suspended (for example, because it is killed by an external agent after a time-out), then reference counts cannot be maintained consistently, and some memory will never be reclaimed. A direct consequence of this more powerful fault model is that garbage collection is required.

Once we made this observation, we were able to simplify our design dramatically, as memory allocation could be effectively ignored. Explicit memory allocation and reclamation is widely known to be a source of bugs. Furthermore, it is generally accepted that reliable, fault-tolerant systems should use garbage collection to guard against memory leaks. In these respects, our use of garbage collection is required by our assumptions, and is not simply a philosophical choice. Our implementation is in C++, and we use the Boehm Conservative Collector for C/C++.

3.5 Open Addressing & Linearizability

The most common data structure used to implement a hash table is the *bucket-and-chain* structure, where the table is an array of linked lists. If we used this approach, we would have to implement concurrent lock-free linked-lists, which are fairly complex. We searched for a simpler alternative.

In the less common technique of *open-addressing*, all bindings are stored directly in an array. When searching for (or inserting) a key, a *probe sequence* is generated for the key that is a permutation of all slots in the array. The slots are visited in this order until either they key is found or until an empty slot is found. As described in [4], open-addressed hash tables have several problems. First, the table has a finite capacity, unlike the bucket-and-chain approach. Second, a slot left by a deleted key cannot be entirely vacated, since subsequent searches must know to skip vacated slots. Consequently, the number of slots visited in the probe sequence increases and performance degrades with use.

In a lock-based open-addressed hash table, one could insert a new key into a vacated slot. However, in a concurrent lock-free table, this causes problems for linearizability, and makes it difficult, if not impossible, to maintain the invariant that a key is not replicated in the table. In our design, a key is inserted into either an unused slot or a slot previously occupied by the same key; also, a vacated

slot contains the key that was deleted. Since the probe sequence of all threads operating on the same key is the same, the key will always be found and will not be replicated in the table.

The open-addressed approach simplifies the analysis of correctness and linearizability. In fact, it also simplifies the process of rehashing in much the same manner. Rehashing is the main reason for leaving the keys in vacated slots. The details are discussed in sections 3.6 and 4.6. The open-addressing solution has these benefits, but has costs as well. The finite capacity of the table and the fact that deleted keys use table slots means that automatic rehashing is a requirement of our implementation. It is fortuitous that automatic rehashing and resizing was one of our design assumptions. It is workload-dependent how much additional rehashing is required of the open-addressed approach.

3.6 Rehashing

Dynamic rehashing of the table is the most complicated part of the design. The requirement that the table be scalable (to a large number of threads) precludes serializing the rehashing code. Since the number of rehashes is likely to increase with the number of client threads, if the rehashing is not parallelized across threads, Amdahl's Law [1] dictates that rehashing time would limit speedup quite severely. It follows that many threads must cooperatively rehash the table. Our implementation effectively parallelizes rehashing across all available threads, and even performs rehashing incrementally (i.e. concurrently) with table operations.

It is instructive (at least for the authors) to recount the story of how we arrived at our solution. After many failed ad-hoc designs of incremental and parallel rehashing, we (optimistically) suspected that there might have been a fundamental limitation of the compare-and-swap primitive that conflicted with our design goals, making our design impossible. This would indeed explain our failed designs! We set out to prove our case. Such a proof required us to approach the problem from first principles and to abstract the rehashing problem. We hoped that an abstract and formal representation of the problem would enable us to prove some inconsistency among our axioms. Alas, we could not prove that rehashing was impossible. In fact, in attempting to do so, we discovered in a surprisingly lucid manner a solution to the problem after all. This is a testament to the subtlety of the issues in designing complex lock-free objects. It is imperative to approach the problem in an abstract context so that inconsistencies are clear and correctness is provable.

Rehashing is difficult because each entry must move atomically from one location in memory to a new location (from the old table to the new table). In the abstract, rehashing a binding involves moving the table between two consistent states: (1) the binding is in the old table and not in the new; (2) the binding is in the new table and not in the old. Ideally, one would use an atomic memory primitive that can operate on two distinct memory locations. Since such an operation is not available, the table must pass through an intermediate state where the binding exists in both tables. This appears to be fundamental.

Further complications are that threads must agree when rehashing should start, when it is happening, and when it is finished. In order to simplify reasoning about our design, we introduce the notion of phases. As soon as some thread decides that rehashing should occur, it performs an oper-

ation that ensures that all threads will eventually enter the next phase of operation. Furthermore, no thread enters the new phase until it is guaranteed that any thread still in the previous phase cannot update the table in any harmful way. This is a sort of dynamic modularity that facilitates analysis considerably.

The last detail is that rehashing should complete before another rehash is needed. This guarantee is not fundamental to all implementations, but simplifies our implementation. Since rehashing is performed incrementally by client threads (not by an asynchronous daemon thread), rehashing and table updates are synchronized. The invariant that each client must rehash at least a constant number of keys for each table update it performs ensures that rehashing will complete before the table fills enough that rehashing is needed again.

3.7 Summary of Our Design

By approaching the problem from first principles and a set of basic assumptions, we have designed a portable lock-free concurrent hash table with scalable incremental rehashing. Restartable atomic sequences and tagged pointers are the general method employed for performing atomic operations. Garbage collection not only simplifies our design (as it would any system), but actually enables our strong fault model. Open addressing makes it easy to prove linearizability, and simplifies rehashing in some subtle ways. The notion of phases modularized our design so that we could design a more complex system, where rehashing is performed incrementally and concurrently with table operations. Finally, all aspects of the implementation are fully parallelized, so that it scales to a large number of concurrent threads.

4 Implementation

4.1 Interface

Figure 1 shows our implementation's C++ template class interface. The version numbers enable a client to perform atomic read-modify-write updates to the table through optimistic concurrency control.

4.2 Synchronization

Figure 2 shows the C prototypes for the atomic memory primitives that we assume. These functions were implemented directly with the SPARC compare-and-swap-extended (`casx`) and compare-and-swap (`cas`) instructions. In order to port our implementation to a new platform, it is only these functions that need to be re-implemented.

```
template <class Ckey, class Cval>
class Ctable
{
  // constructor provides a table size hint
  Ctable (uint ui_numBindings);

  // bind key to val; sets version on success
  // return false if the key is already bound
  bool insert (Ckey& key,              // in
               Cval& val,              // in
               u32& version);          // out

  // remove key from table if version matches
  // return false if the key is not bound or if version does not match
  bool remove (Ckey& key,              // in
               u32 version);           // in

  // sets val and version on success
  // return false if the key is not bound
  bool lookup (Ckey& key,              // in
               Cval& val,              // in
               u32& version);          // out

  // modifies existing binding if version matches
  // return false if the key is not bound or if version does not match
  bool modify (Ckey& key,              // in
               Cval& val,              // in
               u32& version);          // out
};
```

Figure 1: Hash Table Interface. This is the C++ template class that defines the hash table interface.

```
extern bool casx (u64 oldVal, u64 newVal, u64* p_mem);
extern bool cas (u32 oldVal, u32 newVal, u32* p_mem);
```

Figure 2: Atomic Memory Primitives. This is the C interface that we use to access atomic memory primitive operations. The 32b CAS is used to implement an atomic increments of 32b counters.

```
typedef union {
  struct {
    uint ptr       :  32;  // Cbinding*
    uint ver       :  28;  // uint
    uint state     :   2;  // eEntryState
    uint rehashing :   1;  // bool
    uint copied    :   1;  // bool
  } bits;
  u64 all;                 // for alignment and comparisons
} Tentry;
```

Figure 3: Tagged Pointer. This 64b-aligned, 64b value is what we use to store information about each binding in the table. The hash table consists primarily of an array of these entries.

4.3 Tagged Pointers

Figure 3 shows an example of a tagged pointer in the implementation. Since no more than 64b can be modified atomically by the memory system, all of the fields must fit into 64b. The `ptr` field is a pointer to the binding itself, which contains a copy of the key and value that were inserted into the table. The `ver` field is the pointer tag, which is incremented by each atomic update to the entry. The `state` field contains an enumerated type `eEntryState` that holds the state of the entry: `unused`, `vacated`, or `occupied`. The purpose of the one bit `rehashing` and `copied` fields are discussed in Section 4.6. In brief, the `rehashing` bit is set when rehashing has started, and has the effect of making the entry immutable (until it is garbage collected). The `copied` bit is conservatively set after the entry has been copied (i.e. rehashed) to the new table.

4.4 Open Addressing

Figure 4 shows the operation of an open-addressed hash table. An array is used to hold the bindings stored in the table. When a thread needs to operate on a key, it searches through the array in a permuted sequence called the *probe sequence*. The probe sequence is computed as a deterministic function of the key. Consequently, concurrent inserts result in the key being inserted only once into the table, since all threads examine the same sequence of array slots. Section 3.5 argues why this data structure and algorithm is linearizable.

Each slot in the table contains the structure shown in Figure 3. When traversing the probe sequence, a thread examines the state and contents of each slot. Figure 5 shows the actions taken when a thread encounters each type of slot for each of the four table operations. Note that "fail" here indicates that the operation fails, not that the CAS instruction fails: CAS failure results in a retry, not a failure. Also note that deleted bindings leave their keys in the table. The importance of this is revealed in Section 4.6.

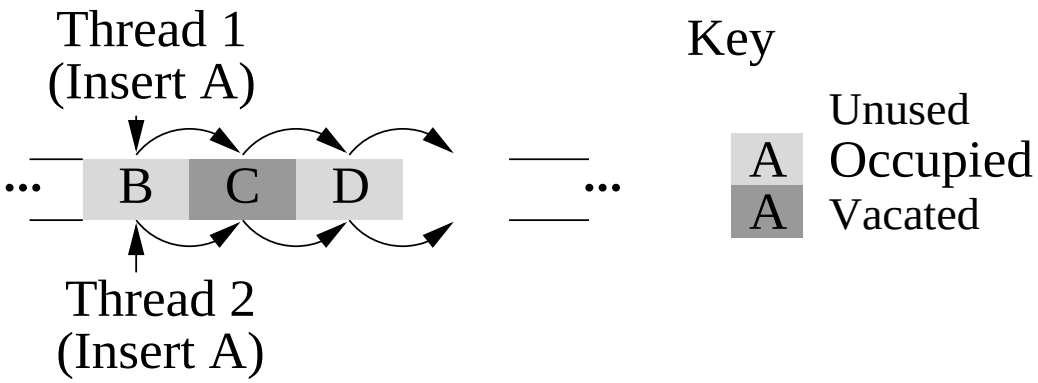


Figure 4: Open Addressing. Two threads concurrently insert a new key into the table. Note that their probe sequences are identical. Note also that deleted bindings leave their keys in the table in `vacated` slots.

Operation	**Entry States**				
	unused	vacated(k)	vacated(k')	occupied(k)	occupied(k')
insert(k)	insert here	insert here	skip	fail	skip
remove(k)	fail	fail	skip	change to vacated(k)	skip
modify(k)	fail	fail	skip	modify slot	skip
lookup(k)	fail	fail	skip	return contents	skip

Figure 5: Table operations and open addressing.

4.5 The Hash Table Data Structure

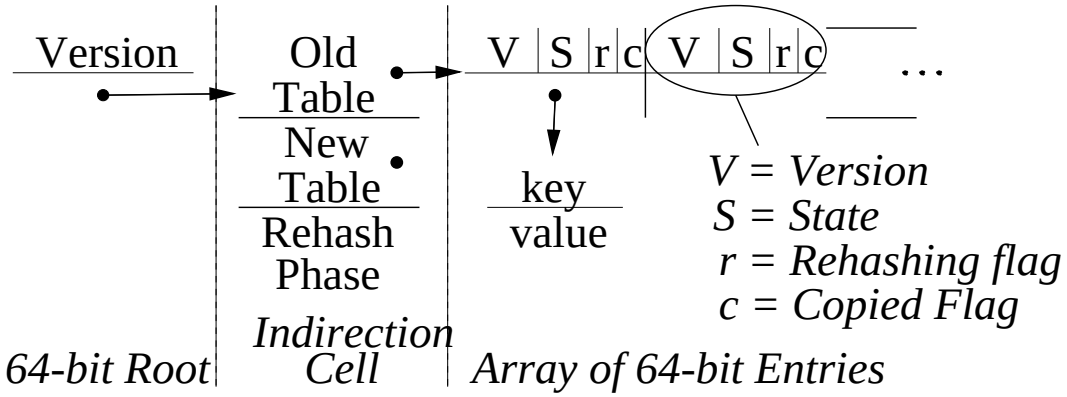


Figure 6: The Hash Table Data Structure. See Section 4.5 for an explanation.

Figure 6 shows the data structure we used in our implementation. The open-addressed array of Figure 4 is shown as the array of 64b entries. There is a 64b root which is a tagged pointer to the entire object. A level of indirection between the root and the array will become useful in moving between phases of rehashing (see Section 4.6). There is one structure not shown in Figure 6 that contains conservative counters for the number of slots in each state. In reality, the "Old Table" pointer in the figure would point to this object, which in turn contains a pointer to the array of entries. This additional object is conceptually a header of the array of entries, and was omitted from the figure for clarity.

It is important to note what portions of the data structure are 64b objects that may be modified with atomic memory primitives. Each entry in the array of entries is such an object, as is the root pointer.

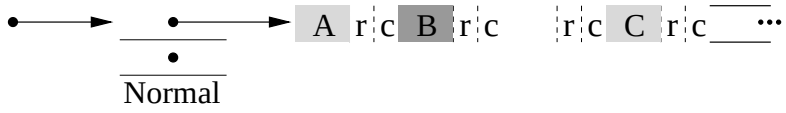
4.6 Parallel Incremental Rehashing

Figure 7 shows the phases of rehashing. The general technique of using phases was introduced in Section 3. This section describes the detailed operation of the table, and how atomic memory primitives are used to move threads through the phases.

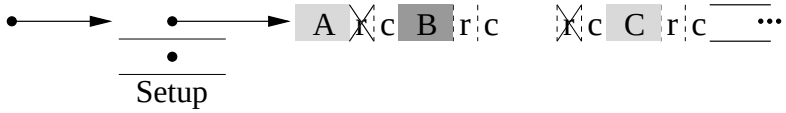
Phase I: Normal Operation

Concurrent updates are performed on the table. An operation must first gain access to the array of entries. The root pointer is copied into a local variable. The structure it points to is examined to ensure that (this copy of) the table is in the "normal" phase. If so, then slots are traversed as per the computed probe sequence. When a target slot is selected, the 64b slot value is copied into a local variable $Slot_{old}$. A replacement slot $Slot_{new}$ is constructed locally. A new entry is allocated into which the new key and value are copied for an `insert` or `modify`. This pointer is then stored

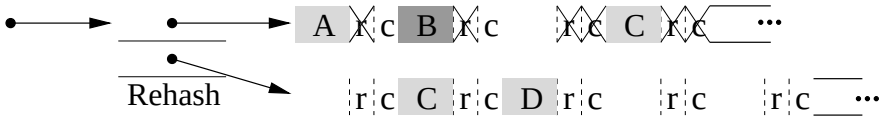
Phase I: Normal Operation



Phase II: Rehash Start



Phase III: Incremental Rehash



Phase IV: Normal Operation

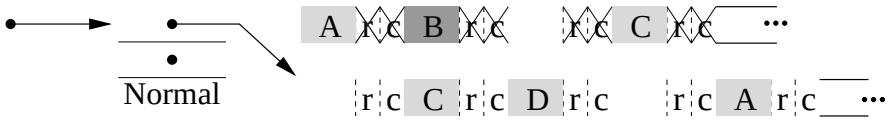


Figure 7: The Phases of Rehashing. Phases I and IV correspond to normal table operation, when there is no need for rehashing and no rehashing is occurring. Phase II is entered when the need to rehash is discovered; all threads cooperate in initializing data structures that will be needed in the next phase. Phase III is when rehashing is performed concurrently with table operations. Threads cooperatively copy bindings into the new table. When all bindings have been copied from the old table, rehashing is complete, and normal operation resumes.

into $Slot_{new}$. The version of $Slot_{new}$ is set to one larger than the version in $Slot_{old}$. The `rehashing` and `copied` bits of *both* $Slot_{old}$ and $Slot_{new}$ are set to zero. A compare-and-swap is then attempted to swap $Slot_{new}$ into the slot. This operation fails if the slot is different than $Slot_{new}$. If the CAS fails, then the operation begins completely anew at the root.

Phase II: Rehash Setup

As described in Section 4.8, conservative counters are maintained that record the number of slots in each of the three states {`unused`,`vacated`,`occupied`}. Before every table operation, a thread checks to see if rehashing should begin. If the number of unused slots is too low, or if the number of occupied slots is too high, then rehashing should occur. If rehashing needs to be done, then some thread will notice, and will set the number of unused slots to a large negative value. This effectively puts the table in the "Rehash Setup" phase. From this point onward, all threads entering a table operation will also decide that rehashing should occur.

There may be threads still operating on the table, however. Before the next phase can be entered, the table must be modified so that all such threads will fail their table updates. This modification is done during this phase by having all entering threads cooperatively set the `rehashing` bits in the table entries. During this phase, there may still be table updates to slots whose `rehashing` bits have not yet been set, but at the end of this phase, no updates to the old table will ever occur.

As soon as all `rehashing` bits have been set, some thread will notice. The process by which the bits are set and by which a thread knows they are all set is the topic of Section 4.7.1. The transition to Phase III is accomplished by some thread swapping in a new root that has the structure shown in Figure 7. The exact mechanism for this swap is detailed in Section 4.7.2.

When some thread notices that all of the `rehashing` bits are set, it proceeds to allocate a new table. A pointer to the new table is stored in a newly allocated indirection cell, along with a pointer to the old table. A new root is then swapped in that points to this new indirection cell. Note that since the old table is now immutable, there is no need to create a copy of it for the root swap.

Phase III: Incremental Rehashing

During this phase, rehashing occurs incrementally and concurrently with table updates. In order to maintain consistency and guarantee that rehashing completes before it is needed again, all operations on a key:

1. Guarantee that if the key is present in the table, it is copied to the new table before any operation is performed.

2. Guarantee that some amount of rehashing is performed.

It is easy to ensure that one rehash finishes before the next begins, since any table operation can consume at most one table slot.

The rehash of an entry involves inserting it into the new table. It is not removed from the old table, since the old table is immutable. After the binding has been copied, the `copied` bit is set in

the old table. It is not possible to perform these two updates atomically, so the ordering is crucial. Since the update of the `rehashing` flag occurs after the actual copy, it is a conservative measure as to whether or not the binding has been copied. Consequently, it is acceptable for the thread to evaporate before the bit is set so long as other threads can safely repeat the rehash. The safety of repeated rehashes is ensured by not deleting a key from the table when a binding is removed. Thus, a rehash-insert operation will fail if the key is in the new table, whether it is in an `occupied` *or* a `vacated` slot.

Phase IV: Back to Normal Operation

As soon as all bindings have been moved to the new table, Phase III can come to an end. At this point, no more rehash-insert operations can take place for the reasons detailed in the preceding paragraph. Some thread will notice that there is no more rehashing work to be done. It will allocate a new indirection object and set the fields as specified in Figure 7. This new view of the table is then swapped into the root, and normal table operation resumes until the next rehash.

4.7 Implementation Subtleties

In the description of rehashing in Section 4.6, some implementation details were omitted for clarity. Each is discussed in more detail in this section.

4.7.1 Rehash Setup

During Phase II of rehashing, all threads need to cooperate to set the rehash flags. Furthermore, some thread needs to know when all of the flags are set. A simple linear sweep of the flags will be functionally sufficient, but does not yield a scalable implementation. In order to have a scalable implementation, the work performed by each thread must scale as n/t, and must be performed in parallel (where n is the size of the table and t is the number of active threads).

Assuming that the number of threads $t < O(\sqrt{n})$, there is a simple solution. Divide the old table into $\sqrt{n}$ *chunks* of size $\sqrt{n}$. Have the threads examine the chunks in a random permuted order. At each chunk, the first flag is examined. If it is set, then this chunk can be skipped. If the flag is not set, then begin setting the flags in *reverse* order. With t actively participating threads, the total number of chunks examined is $O(\sqrt{n}/t)$, the total work per chunk is $O(\sqrt{n})$, and so the total amount of work performed per thread is O(n/t). Assuming that the random permuted order eliminates any chances of convoying, this method should scale perfectly with t. If $t > \sqrt{n}$, then a more sophisticated tree-based approach may be necessary.

4.7.2 Allocating the New Table

When moving from Phase II to Phase III, it is necessary to allocate and install a new table. It is harmless from a standpoint of correctness to allow more than one thread to attempt this allocation

and swap sequence, but it may be unacceptable from a memory allocation standpoint. Fortunately, a simple solution is available.

The thread that sets `rehashing` bit zero is "elected" to perform the allocation and swap. As soon as a thread succeeds in setting this bit, it raises its priority so that it is less likely to be descheduled. It then proceeds to allocate the new table and attempt the root swap. As soon as it succeeds the swap, it returns its priority to its previous value.

All other threads enter a spin-loop where they yield the processor if they find that the root has not yet changed. With this mechanism alone, if the elected thread died or was somehow descheduled, we would have deadlock. Consequently, the threads that spin must time-out, perform the allocation, and attempt the root swap.

In practice, we observe that this method performs in an exemplary manner. It has the desirable properties of being very fast in the common case and of being correct in all cases, even when threads die at inopportune moments.

4.7.3 Initializing the New Table

In Section 4.7.2, it is implied that the thread that allocates the new table also initializes it. This event is also implicitly depicted in Figure 7. For the same reason that a single thread cannot simply set all of the `rehashing` bits in Phase II, having a single thread initialize the new table does not yield a scalable implementation. This is easily mended by introducing a phase between Phases II and III where the new table is initialized. If the number of active threads is not too large, a similar $\sqrt{n}$ trick as that detailed in Section 4.7.1 may be used to initialize the table.

The modifications to the algorithm are as follows. The thread that swaps in the new table allocates and initializes a $\sqrt{n}$-length auxiliary array that is a record of what chunks have been initialized in the new array. The algorithm described in Section 4.7.1 is used to initialize the new table. When this is complete, Phase III is entered as before. Again, if the assumption that the number of active threads is $< \sqrt{n}$ is not valid, then a tree-based approach could be substituted.

4.8 Conservative Measures

There are situations in the implementation where we would like to atomically update two disjoint memory locations. Such problems can be solved by ordering the updates such that the state of the table is *conservative*. The two examples are:

1. During normal table operation, we would like to maintain counters that keep a record of how many table entries are in each state. These counters cannot possibly be maintained consistently with the table, but we make the observation that the counters can be wrong if they are wrong in the correct direction. For example, the counter that indicates how many occupied slots are in the table can safely be too high. The worst that can happen is that a rehash is triggered too soon. This will only occur in exceptional cases where threads repeatedly

evaporate at the most inopportune moments. This particular counter is incremented *before* a binding is actually added and *after* a binding is deleted.

2. During Phase III of rehashing, threads need to know if an entry has been rehashed. The `copied` bit is set after an entry has been rehashed. However, it is not possible to rehash an entry *and* set the `copied` bit atomically. So long as the bit is set *after* the rehash, the information is conservative. By ensuring that repeated rehash attempts are safe (see Section 4.6), consistency is maintained.

The notion of conservative sequences of atomic updates is important when implementing complex atomic updates that cannot be implemented directly with the available atomic memory primitives.

4.9 Correctness

The preceding sections have argued in a disjoint manner that certain parts of the implementation are correct. Figure 8 reiterates these points into a coherent argument of correctness by listing the hash table's phases of operation and the invariants maintained in each phase.

4.10 Implementation Extensions

There are a few areas in which we feel our implementation is somewhat deficient. This section identifies these problems and proposes solutions.

- Incremental rehashing proceeds by incrementally rehashing chunks of the old table. In our implementation, these chunks are of size $O(\sqrt{n})$. Consequently, though the throughput of table operations during rehashing is guaranteed to be non-zero (because it is incremental), it is extremely low. It would be better to allow the rehash chunks to be some small constant size and still retain scalable time bounds.

  The difficultly is maintaining $O(1)$ table operations during rehashing, since in order to check that rehashing is complete, one must know that all rehash flags have been set. We suspect that a tree-based approach could be employed to at least lower the cost of table operations during rehashing to $O(\log n)$. If we could assume that threads are long-lived with respect to table operations, then *thread contexts* may help in avoiding repeated work. If $O(1)$ operations during rehashing are important to clients, then an $O(1)$ solution is required.

- We did not implement automatic rehashing to smaller sizes, but this is a trivial addition.

- It may be nice to add to the interface methods that force a rehash so that rehashing occurs at opportune moments. A method that triggers an incremental rehash is trivial – it simply needs to set the counter for the number of unused slots to a large negative value (see Section 4.6). A sequential rehash method is trivial by nature, since it would not have to worry about atomicity. A version that concurrently rehashes the table and returns when rehashing has completed is simple to add. A production system would surely include some assortment of such methods.

Phase	**Some threads are doing this.**	**Old threads may try this, but will fail.**	**Why they will fail.**
Normal Operation	Any of the table operations.	Insert more than one entry with the same key (true in all phases).	After some thread succeeds in inserting a key into a slot, all other threads will pass that slot in their probe sequence.
		Perform non-atomic operations on a table entry (true in all phases).	All operations on entries use CAS.
Rehash Setup	Setting `rehashing` bits of some entries.	Modify entries with `rehashing` set (true in all phases).	All normal table operations attempt a CAS where the old value is provided with a `rehashing` bit set to zero.
Incremental Rehashing	Copying entries from old to new table; setting `copied` bits.	Copy an entry into the new table a second time.	Once a binding has been inserted, its key is never removed.
		Modify the old table.	The old table is immutable (all `rehashing` bits are set).
		Perform a table operation on an entry that is not yet in the new table.	Any entry is moved to the new table before it is operated upon.

Figure 8: Invariants. These demonstrate the correctness of the implementation.

5 Results

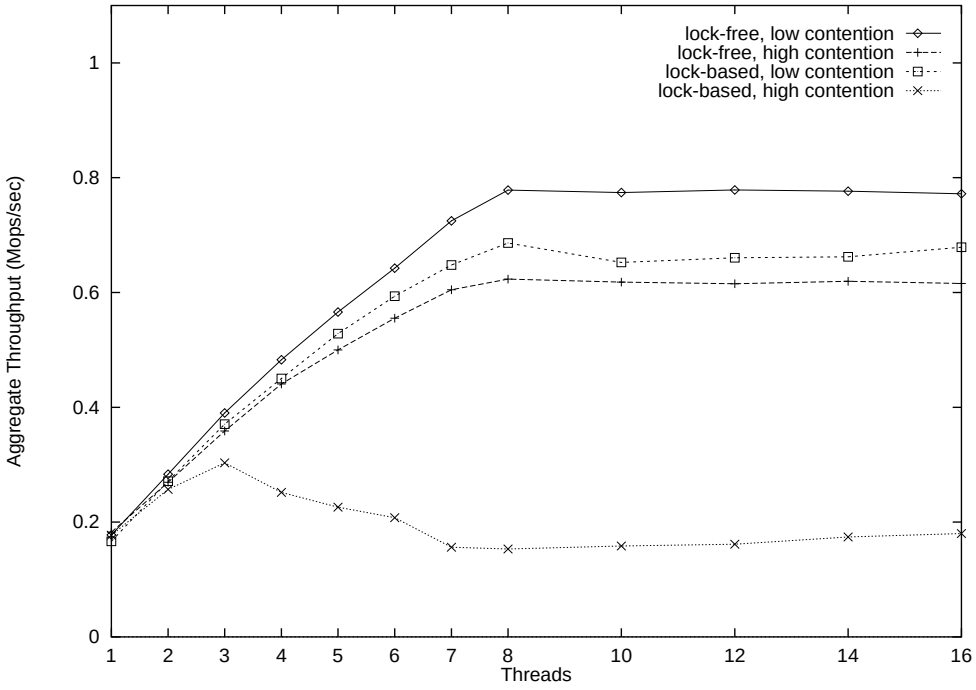


Figure 9: Aggregate Throughput of Lock-Free and Lock-Based Tables. Each thread performed 1M lookup/modify iterations on a table with 10K entries. For the low contention cases, threads randomly choose from all 10K possible keys at each iteration. For the high contention cases, threads randomly choose from only 10 of the 10K possible keys.

We tested and evaluated our implementation on a Sun Enterprise-5000 with 8 167 MHz UltraSPARC processors. In order to evaluate the lock-free table, we implemented a low-overhead lock-based table with per-bucket mutexes. We used the C++ language coupled with the Boehm conservative collector[1].

Figure 9 shows that the lock-free table consistently out-performs the lock-based table. When there is little contention in the table, the lock-based code is approximately 10% faster. In the presence of contention, the performance of the lock-based code degrades dramatically and irregularly, while the performance of the lock-free code degrades quite gracefully.

Figure 9 shows performance with an ideal memory allocator. Unfortunately, we did not have available and did not implement a lock-free concurrent memory allocator, so the end-to-end performance of the lock-free table is actually quite poor. Figure 10 shows how the performance of the lock-free

[1]The Hans-J. Boehm collector is a conservative garbage collector for C/C++. It can be found at Boehm's homepage: http://reality.sgi.com/employees/boehm_mti/.

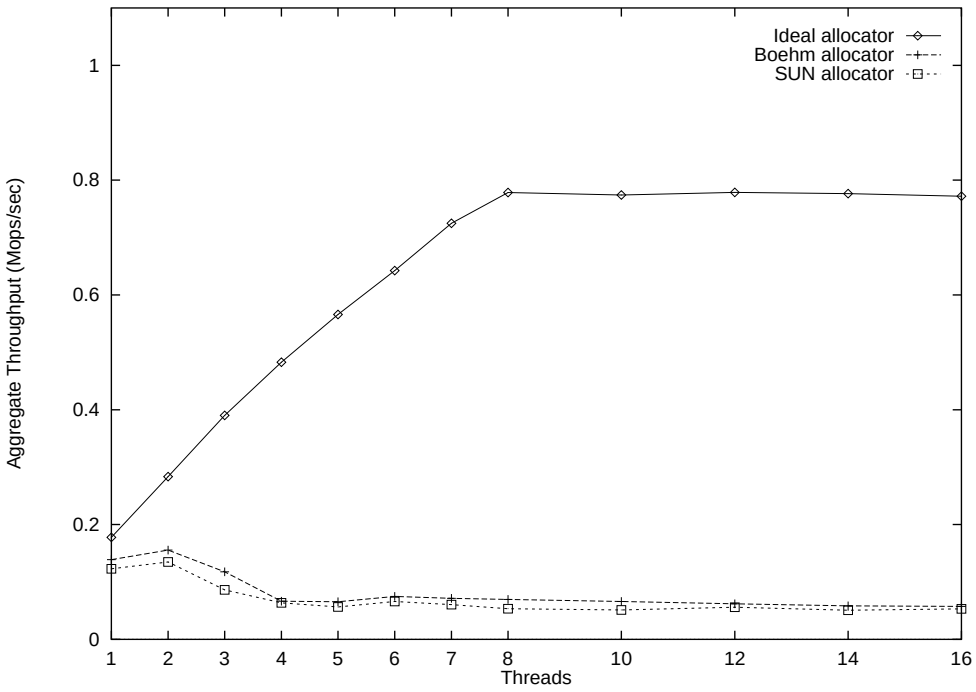


Figure 10: Effect of Allocation on Aggregate Throughput. In all three cases, each thread performs 1M low contention iterations as described in Figure 9.

table crumbles when it uses a lock-based allocator. Both the Boehm allocator and the standard SUN allocator use mutexes. It is important to note that lock-free is an end-to-end property. A lock-free memory allocator is a prerequisite to building complex lock-free data structures that need to perform dynamic allocation.

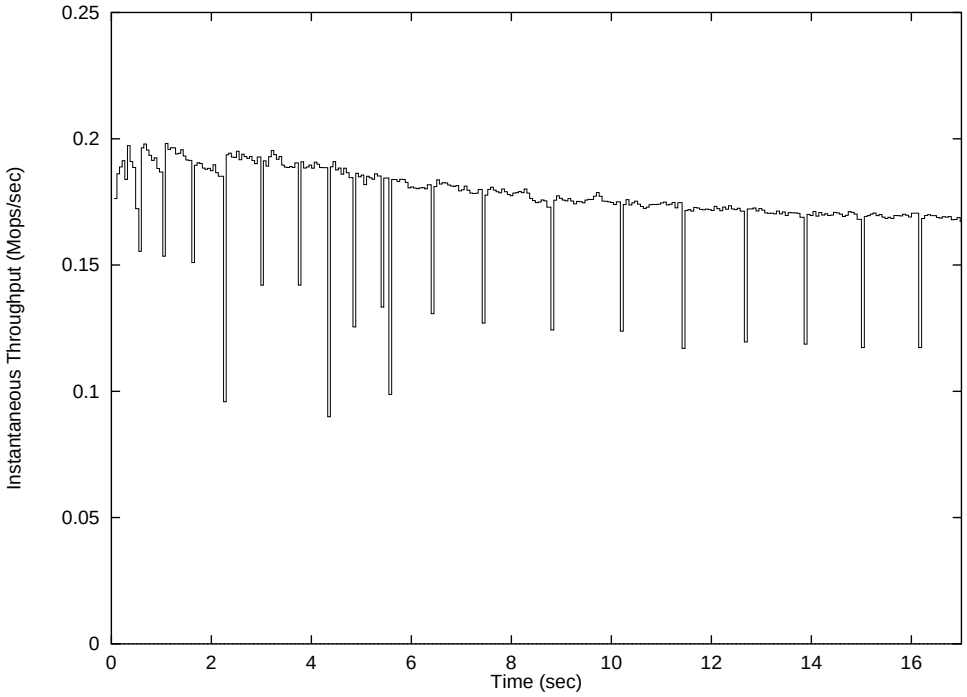


Figure 11: Effect of Rehashing and Garbage Collection on Instantaneous Throughput. This graph shows the instantaneous throughput as seen by a single thread operating on the lock-free table with no other active threads. Most of the spikes correspond to periodic garbage collections. The spikes at 1.0, 2.3, 4.3, and 5.55 seconds correspond to rehashing events. Note how performance degrades from 0.6 to 1.0 seconds at which point it increases after a rehash. It then decreases again until the next rehash at 2.3 seconds. After the rehash at 5.55 seconds, the table is big enough to absorb all new keys that are added, so no more rehashing occurs; all remaining spikes are due to the Boehm collector.

Figure 11 shows how garbage collection and rehashing affect instantaneous throughput. Even though this shows a synthetic benchmark that stresses the allocator, only a few percent of execution time can be attributed to garbage collection. Rehashing is even more infrequent, and consequently costs very little.

6 Conclusion

Lock-free data structures often have better performance than lock-based structures, and they also eliminate problems such as convoying, priority inversion, and deadlock. Thus, it is important that highly concurrent applications have access to such data structures. We have implemented a non-blocking hash table, which may be a useful building block for concurrent systems.

Though a concurrent hash table with dynamic rehashing is a complex data structure, the use of open addressing and careful application of well-formed design methods makes it relatively easy to reason about the correctness of our implementation. By providing incremental rehashing, our implementation transparently rehashes the table concurrently with table operations. Our implementation is shown to scale well with the size of the table and the number of concurrent threads, but since *non-blocking* is an end-to-end property, it relies on a non-blocking memory allocator. Because out fault model assumes that threads can evaporate, garbage collection is not only essential, but is simplifies the design significantly.

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485, April 1967.

[2] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[3] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. *Fifth Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1992.

[5] Samuel P. Harbison. *MODULA-3*. Prentice Hall, 1992.

[6] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, pages 463–492, 1990.

[7] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, pages 745–770, 1993.

[8] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *Proceedings of the 1993 International Symposium in Computer Architecture*, May 1993.

[9] Amos Israeli and Lihu Rappoport. Efficient wait-free implementation of a concurrent priority queue. In *Lecture Notes in Computer Science 725*, pages 1–17. Springer Verlag, September 1993.

[10] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1991.

[11] Maged M. Michael and Michael L. Scott. Simple, fast, and practial non-blocking and blocking concurrent queue algorithms. *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 267–276, May 1996.

[12] Richard L. Sites. *Alpha Architecture Reference Manual*. Digital Press, 1992.

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

[14] David L. Weaver and Tom Germond. *The SPARC Architecture Manual*. Prentice Hall, 1994.