Fast, Dynamically-Sized Concurrent Hash Table*

J. Barnat, P. Ročkai, V. Štill, and J. Weiser

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{barnat,xrockai,xstill,xweiser1}@fi.muni.cz

Abstract. We present a new design and a C++ implementation of a high-performance, cache-efficient hash table suitable for use in implementation of parallel programs in shared memory. Among the main design criteria were the ability to efficiently use variable-length keys, dynamic table resizing to accommodate data sets of unpredictable size and fully concurrent read-write access.
We show that the design is correct with respect to data races, both through a high-level argument, as well as by using a model checker to prove crucial safety properties of the actual implementation. Finally, we provide a number of benchmarks showing the performance characteristics of the C++ implementation, in comparison with both sequential-access and concurrent-access designs.

1 Introduction

Many practical algorithms make use of hash tables as a fast, compact data structure with expected $\mathcal{O}(1)$ lookup and insertion. Moreover, in many applications, it is desirable that multiple threads can access the data structure at once, ideally without causing execution delays due to synchronisation or locking. One such application of hash tables is parallel model checking, where the hash table is a central structure, and its performance is crucial for a successful, scalable implementation of the model checking algorithm. Moreover, in this context, it is also imperative that the hash table is compact (has low memory overhead), because the model checker is often primarily constrained by available memory: therefore, a more compact hash table can directly translate into the ability to model-check larger problem instances. Another desirable property is an ability to dynamically resize (grow) the hash table, in accordance with changing needs of the model checking algorithm as it explores the state space. Finally, it is often the case that the items (state vectors) stored in the hash table by the model checker have a dynamic size, for which it is difficult to predict an upper bound. Hence, we need to be able to efficiently store variable-length keys in the hash table.

While the outlined use-case from parallel model checking was our original motivation, a data structure with the same or similar properties is useful in many other applications.

* This work has been partially supported by the Czech Science Foundation grant No. 15-08772S.

1.1 Related Work

As we demonstrate in Section 4, our design is highly competitive, improving on the state of the art in parallel data structures, as represented by the venerable Intel Threading Building Blocks library [1]. The design presented in this paper offers faster sequential access, better multi-threaded scalability and reduced memory overhead. Most of these attributes can be derived from the fact that our design is based on an open hashing scheme, in contrast to almost all existing concurrent hash tables. Often, concurrent hash table designs take advantage of the simplicity of concurrent access to linked lists (eg. [2], but also the designs in Intel TBB [1]), leading to a closed hashing scheme. Alternatively, a concurrent, open-hashing table based on our earlier (sequential) design has been described in [3], but while providing very good performance and scalability, it was limited to statically pre-allocated hash tables (i.e. with a fixed number of slots). Our design, however, does not explicitly deal with key removal: a standard 'tombstone' approach can be used, although it may also be possible to leverage the scheme proposed in [4], where authors focus on deletion in a concurrent (but fixed size) hash table with open addressing.

A more closely related design (without an implementation, however) was presented in [5]. In this paper, the authors present a concurrent hash table based on open hashing and arrive to solution that are in many cases similar to ours. Especially the approach to ensuring that resize operations do not interfere with running inserts is very similar – in this particular case, we believe that the extensive and detailed correctness proofs done in [5] would transfer to our design with only minor adjustments. Our present paper, however, places more emphasis on the implementation and its practical consequences. By comparing notes with existing work on the subject, we can conclude that the design approach is sound in principle; while we did basic correctness analysis on the design, our main concern was correctness of the implementation. Unlike existing work, we make use of software model checking to ascertain that the implementation (and by extension, the design) is indeed correct.

2 Design

There are many considerations that influence the design of a data structure. Our first priorities were performance and scalability of concurrent access; in both cases, it is important to consider the hardware which will execute the code.

First, we need to realize that modern multi-core and SMP systems exhibit a deep memory hierarchy, with many levels of cache. Some of this cache is shared by multiple cores, some is private to a particular core. This translates into a complex memory layout. To further complicate matters, multi-CPU computers nowadays often use non-uniform access architecture even for the main memory: different parts of RAM have different latency towards different cores. Most of this complexity is implicitly hidden by the architecture, but performance-wise, this abstraction is necessarily leaky.

Moreover, the gap between the first and the last rungs of the hierarchy is huge: this means that compact data structures often vastly outperform asymptotically equivalent, but sparse structures. Due to cache organisation constraints, memory cells that live close to each other are usually fetched and flushed together, as part of a single "cache line". They are also synchronised together between core-private caches. A modern data structure should therefore strive to reduce to an absolute minimum the number of cache lines it needs to access in order to perform a particular operation. On the other hand, when concurrency is involved, there is a strong preference to have threads use non-overlapping sets of cache-line-sized chunks of memory, especially in hot code paths.

2.1 Hash Functions

A hash table is represented as a vector of values in memory, associated with a function that maps *keys* to indices within this vector. The function is known as a *hash function* and should posses a number of specific properties: the distribution of key images should be uniform across the entire length of the vector, a small change in the key should produce a large change in the value, the function should be fast to compute and such a function should be available for arbitrary index range.

In practice, to implement the last criterion, hash functions for hash tables are usually implemented over the range of all 32 (64, 128 bit) integers in such a way that the remainder of division by an arbitrary integer n (or at least a power of two) will yield a uniform distribution in $\{1, ..., n\}$. The current practice is to use a purpose-built lookup function, either providing 64 (`lookup3` [6] is a good candidate) or even 128 bits of output (the currently best available are spooky hash [7] and the city hash [8]).

2.2 Open vs Closed Hashing

Even with the best lookup function, hash collisions, and more importantly, index collisions will happen in a dynamic hash table. Hence, an important part of the hash table design is dealing with such collisions, and there are two main options: open and closed hashing (also known as open and closed *addressing*). With a closed hashing scheme, each position in the hash table is a "bucket" – capable of holding multiple values at the same time. This is implemented using an auxiliary data structure, usually a linked list. While closed hashing is easier to implement and to predict, it usually gives poor performance. An alternative is to make each position in the table only hold at most one value at a time, using alternate positions for items that cause a collision. Instead of using a single fixed position for each value, the hash table has a list of candidate indices. The most common such series are $h + ai + b$ where i is the sequence number of the index, h is the index assigned by a lookup function and a, b are arbitrary constants (a linear probing scheme). Another common choice is $h + ai^2 + bi + c$, obviously known as quadratic probing. An important property of a probing scheme is that it does not (significantly) disrupt the uniform distribution of values across indices. In

case of a quadratic function and a hash table with a size that is a power of 2, a simple set of constraints can be shown to give a good distribution [9].

2.3 Cache Performance

There are many considerations when choosing a good hash table implementation for a particular application. In model checking, as well as many other use cases, the hash table becomes very big, and as such, it cannot fit in the CPU cache entirely. For that reason, it is very important that all hash table operations have as much spatial and temporal locality as possible, to make best possible use of the CPU cache. The very nature of a hash table means that insert or lookup operations on different keys will access entirely different memory regions: this is unavoidable. However, with a naive implementation, even a single lookup or insert can cause many cache misses: a closed-hashing scheme, for example, will need to traverse a linked list during collision resolution, which is a notoriously cache-inefficient operation. Even if we would use a different auxiliary data structure, we would still face at least one level of indirection, causing an extra cache miss. With open hashing and a linear probing function, we can expect a high degree of spatial locality in the collision resolution process: all candidate positions can be fetched in a burst read from a continuous block of memory. In fact, this is a cache-optimal solution, as it only incurs the one unavoidable initial cache miss per lookup.

However, linear probing has other problems: the property that makes it cache efficient also means that it has a strong tendency to create uneven key distribution across the hash table. The clumping of values makes the collision chains long, and even though it is cache-efficient, the linear complexity of walking the chain will dominate after reaching a certain chain length. In contrast, a quadratic scheme will scatter the collision chain across the table. Hence, as a compromise, a hybrid probing function can be used: a quadratic function with a linear tail after each "jump": $h + q(\lfloor i \,/\, b \rfloor) + i \bmod b$ where q is a quadratic function and b is a small multiple of cache line size. This has the advantage of scattering keys across the table, but in small clumps that load together into cache, without seriously compromising uniformity.

2.4 Variable-Length Keys

If there is substantial variation in key size, it is inefficient to store the entire key inline in the hash table, and impossible if no upper bound on key size is known. This means that we need to store pointers in the table and the key data becomes out-of-line. Unfortunately, this has disastrous effects on cache performance: each key comparison now requires an extra memory fetch: in order to find a key in the table, we need to compare it to each element in the collision chain.

To negate this effect, we can store the actual hash value of each key inline in the table: this way, we can first compare the hash values, without incurring a memory fetch. In vast majority of cases, a 64-bit hash will only test as equal if the actual keys are equal – we will only pay the price of an extra memory fetch

in the cases where the keys are actually equal, which is at most once per lookup, and in only a tiny fraction of cases where the keys are distinct.

Even though efficient, this approach doubles the memory overhead of the hash table, storing a pointer and an equal-sized hash value for each key. This is especially problematic on 64-bit machines, making the overhead 16 bytes per slot when using a 64-bit hash value. Moreover, a 64-bit hash value is needlessly big, a much smaller, 32 or even 16 bit value would provide nearly the same value in terms of avoided cache misses , as long as the part of the hash saved in the cell is distinct from the part used for computation of a cell index. On most platforms, though, this will require arranging the hash table in terms of cache lines, as 96 or 80 bit slots will cause serious mis-alignment issues. With the knowledge of a cache-line size, we can organise the hash table into "super-slots" where each super-slot fits in a cache line, and packs the pointers first and the corresponding hash values next, in the tail.

On 64-bit machines, though, there is another option, which avoids most of the layout complexity at the table level. Contemporary CPUs only actually use 48 bits out of the 64 bit pointer for addressing, the rest is unused. While it is strongly discouraged to use these 16 extra bits for storing data (and CPU vendors implement schemes to make it hard), this discouragement is more relevant at the OS level. At the expense of forward portability of the hash table implementation, we could use these 16 bits to store the hash value, reconstructing the original pointer before dereferencing it. Finally, it is also possible to use an efficient pointer indirection scheme, which explicitly uses 48-bit addressing in a portable, forward-compatible fashion [10].

2.5 Capacity & Rehashing

As we have already said, a hash table is normally implemented as a vector, whether it contains single-value slots or multi-value buckets. As such, this vector has a certain size, and as keys are added into the table, it becomes increasingly full. The ratio of slots taken to slots available is known as a load factor, and most hash table implementations perform reasonably well until load of approximately 0.75 is reached (although factors as high as 0.9 can be efficient [11]). At certain point, though, each hash table will suffer from overlong collision chains. This problem is more pronounced with open hashing schemes: in the extreme, if there is only one free slot left, an open hashing scheme may need to iterate through the entire vector before finding it. There are two options on how to avoid this problem: the more efficient is to approximately know the number of keys that we'll store beforehand. However, this is often impossible, and in those cases, we need to be able to resize the table. This is usually done in the manner of a traditional dynamic array, only the values are not copied but rehashed into the newly allocated vector, which is usually twice the size of the current one.

Rehashing the entire table is at best a linear operation, but amortises over insertions down to a constant per insert. In real-time applications, gradual rehashing schemes are used to avoid the latency of full rehashing. However, in most application, latency is of no concern and monolithic rehashing is in fact more

efficient. As a small bonus, rehashing the table will break up existing collision chains and give the table an optimal uniform layout.

2.6 Concurrent Access

As we have discussed, open hashing is more cache efficient, and compared to a simple closed hashing scheme is also more space efficient. However, closed hashing has an important advantage: linked lists are a data structure easily adapted for lock-free concurrent access. Hence, most concurrent hash table implementations are based on closed hashing. The situation with open hashing is considerably more complex. It is relatively straightforward to implement a fixed-size hash table (i.e. for the scenario where we know the size of the working set in advance). Since this is not the case in DIVINE [12], we have implemented a (nearly) lock-free, resizable open-hashed table, to retain the advantages of open hashing, while at the same time gaining the ability to share the closed set of the graph exploration algorithm among multiple threads.

Let us first discuss how a fixed-size open-hashed table can accommodate concurrent access. The primary data race in a non-concurrent table is between multiple inserts: it could happen that two insert operations pick the same free slot to use, and both could write their key into that slot – this way, the insert that wrote later went OK; however, the first insert apparently succeeds but the key is actually lost. To prevent this, write operations on each slot need to be serialised. The simple way to achieve this is with a lock: a spinlock over a single bit is simple and efficient on modern hardware, and since each hash table slot has its own lock, contention will be minimal. Using a lock is necessary in cases where the key cannot be written atomically, i.e. it is too long. If the key fits within a single atomic machine word, a locking bit is not required, and an atomic compare-and-swap can be used to implement writing a slot. When a lock is used, the lock is acquired first, then the value to insert and the locked slot are compared and possibly written. When using a compare-and-swap, in case it fails, we need to compare the keys – concurrent inserts of the same key could have occurred, and the same key must not be inserted at two different indices.

Concurrent lookups are by definition safe, however we need to investigate lookups concurrent with an insert: it is permissible that a lookup of an item that is being inserted at the same time fails, since there is no happens-before relationship between the two (this is in fact the definition of concurrency). It can be easily seen that an insert of a different key cannot disrupt a lookup of a key that is already present: all inserts happen at the end of a collision chain, never in the middle where they could affect a concurrent lookup.

In cases where variable-length keys are used based on the scheme suggested in section 2.4, lock-free access is only possible for variants where the pointer and the hash (if present) are located next to each other in memory, i.e. a hash-free (pointers only) table, or the 64 bit + 64 bit variant (only on machines with atomic 128-bit compare-and-swap), or the variant with the pointer and the hash combined into a single 64 bit value.

2.7 Concurrency vs Resizing

The scheme outlined in last section does not take the need for resizing and subsequent rehashing into account. The first problem of a concurrent resize operation is that we cannot suspend running inserts, as this would require a global lock. However, insert as a whole is not, and cannot be made, an atomic operation: only the individual probes are atomic. As a consequence, if we were to re-allocate the table at a different address and de-allocate the existing one, a concurrent insert could be still using the already freed memory. Since we cannot interrupt or cancel an insert running in a different thread, nor can we predict when will it finish, the best course of action is to defer the de-allocation. Unfortunately, even if we avoid writing into invalid memory, the same set of circumstances can cause an insert to be lost, since at the point it is written, the copying (rehashing) of the table might have progressed beyond its slot (and since the probing order is not, and cannot be made, monotonic, this cannot be prevented).

In order to clean up unused memory as soon as possible, and to solve the "lost insert" problem, we can, after each insert, verify that the currently active table is the same as the table that was active when the insert started. When they are the same, no extra work needs to be done, and the insert is successful: this case is the same as with a fixed-size table. If, however, the active table has changed, the insert has to be restarted with the new table. Additionally, we can use the opportunity to also clean up the old table if it is no longer used – if there are no further threads using the table. To reliably detect this condition, we need to associate an atomic reference counter with each table generation pointer.

Finally, if an insert has been restarted and succeeds, but the reference count on the old table pointer is not yet zero, the thread doing the insert can optionally help rehashing the table. This way, the resize operation can be executed safely in parallel, greatly reducing the time required: since an individual insert is already thread-safe, it is sufficient to slice the old table into sections and let each thread rehash keys from a non-overlapping subset of slices. The assignment of slices to threads can be implemented using a standard concurrent work queue.

3 Implementation

We have implemented the design laid out in the previous section[1], in order to evaluate and verify it, and also for use in the DIVINE model checker. We provide pseudocode for the most important parts of the implementation (see Algorithm 1), but for full details we refer the reader to the C++ implementation, which is unfortunately too extensive to be included here. The basic design of a sequential open-hashing hash table is very straightforward, including rehashing: the table is entirely stored in a sequential area of memory, consisting of fixed-size cells. For long or variable-length keys, the cells contain a pointer to the data itself; small fixed-size keys can be stored directly. Rehashing is realised by

[1] The C++ source code for the hash table implementation can be found online: `https://divine.fi.muni.cz/trac/browser/bricks/brick-hashset.h#L481`

allocating a new, empty hash table of a larger size (usually a small multiple of the current size) and invoking the 'insert' procedure for each element present in the current table. When all elements have been rehashed this way, the old table can be de-allocated.

Our implementation follows the same scheme, but with a few provisions to deal with data races arising in concurrent use. These have been outlined in Sections 2.6 and 2.7 – the implementation follows the design closely. To better illustrate the principles behind those provisions, we provide a schematic of the table layout in memory (Figure 1), as well as an example course of a concurrent *insert* operation in Figures 2 and 3 and a scheme of the concurrent resize algorithm in Figures 4 through 6.

3.1 Verification

In order to ensure that the hash table works as expected, we have used DIVINE to check some of its basic properties[2]. The properties are expressed as small C++ programs – basically what a programmer would normally call a unit test. They are usually parametric, with the parameters governing the size and parameters of the data structure as well as the way it is used.

Clearly, the parameter space for various properties is infinite, and admittedly, even for fairly small values the verification problem becomes very large. Nevertheless, most bugs happen in boundary conditions, and these are identical for all parameter instantiations upwards of some structure-specific minimum.

The second limitation is that we can only currently verify the code under the assumption of sequential consistency. At first sight, this may seem like a severe limitation – on a closer look, though, it turns out that vast majority of relevant memory access is already tagged as sequentially consistent using appropriate `std::atomic` interfaces (this translates to appropriate architecture-specific memory access instructions that guarantee sequential consistency on the value itself, as well as working as a memory fence for other nearby memory accesses). In this light, the limitation is not quite fatal, although of course it would be preferable to obtain verification results under a relaxed memory model.

For verification of the concurrent hashset implementation, we have opted for a property parametrised with two numbers, T – the number of threads accessing the shared data structure, and N – the number of items each of those threads inserts into the data structure. The item sets inserted by each thread are disjoint (but a similar test was done with overlapping item sets).

First, we define a few types, namely the `hasher` which defines how to compute hashes from items and a few other item properties, like which value of the item

[2] Doubts could arise when using a model checker which uses the hash table to be verified internally. An analysis of failure modes of the hash table along with the properties of the model checking algorithm indicate that this could not cause the model checker to miss a valid counterexample. Nonetheless, the entire issue is easy to side-step by using a much simpler sequential hash table and just waiting longer for the result.

```
1  Function ReleaseMemory(index) is
2      if refCount[ index ] - 1 = 0 then
3          deallocate row at index;
4  Function Rehash() is
5      while segment is available do
6          for cell ∈ segment do
7              lock the cell;
8              if cell is not empty then
9                  mark cell as invalid;
10                 insert cell to the new row;
11         if was it the last segment then
12             ReleaseMemory( currentRow - 1);
13 Function Grow(newIndex) is
14     lock the growing lock;
15     if current row has changed then
16         unlock;
17         return false;
18     row[ newIndex ] ← array[ NextSize( oldSize) ];
19     refCount[ newIndex ] ← 1;
20     allow rehashing;
21     Rehash();
22     return true;
23 Function InsertCell(value, hash, index) is
24     for attempt ← 0 . . . maxAttempts do
25         cell ← row[ index ][ Index( hash, attempt) ];
26         if cell is empty then
27             if store value and hash into cell then
28                 return (Success, cell);
29             if index ≠ currentRow then
30                 return (Growing)
31         if cell is (value, hash) then
32             return (Found, cell);
33         if index ≠ currentRow then
34             return (Growing)
35     return (NoSpace)
36 Function Insert(value, hash, index) is
37     while true do
38         res ← InsertCell(value, hash, index);
39         switch res.first do
40             case Success
41                 return (res.second, true);
42             case Found
43                 return (res.second, false);
44             case NoSpace
45                 if Grow(index + 1) then
46                     index ← index + 1;
47                     break;
48             case Growing
49                 Rehash();
50                 UpdateIndex();
```

Algorithm 1: Pseudocode for key procedures.

type (`int` in this case) represents non-existence. The `HS` type represents the test hash table itself, and `t` is a pointer to the instance of the hash table we will be using for expressing the property.

```
using hasher = brick_test::hashset::test_hasher< int >;
using HS = brick::hashset::Concurrent< int, hasher >;

HS *t = nullptr;
```

With those definitions out of the way, we define what our model thread is like: namely, it takes two parameters – from and to, and inserts those items into the hash table. When all the items have been inserted, it verifies that the items it has inserted are indeed present.

```
struct Insert : brick::shmem::Thread {
  int from, to;
  HS::ThreadData td;

  void main() {
    try {
      int i = from;
      for ( int i = from; i < to; ++i )
        t->withTD(td).insert(i);
      for ( int i = from; i < to; ++i )
        assert(t->withTD(td).count(i));
    } catch (...) {}
  }

  Insert() : from( 0 ), to( 0 ) {}
};
```

Finally, we need to set up the worker threads and the hash table. We use a table which is limited to 4 size increases (the hash table normally uses a factor 16 resize until it reaches the size of at least 512k cells, but this fast initial growth is suppressed for verification runs, using the normal 2-fold increase in each growth step). The initial size of the hash table is set to 2 cells, so that in most cases at least 1 resize is required during the test.

```
int main() {
  try {
    Insert th[T];
    t = new HS(hasher(), 4);
    t->setSize(2);
```

We then proceed to set up parameters of the individual threads, setting their `from`/`to` parameters to non-overlapping, consecutive ranges.

```
    for ( int i = 0; i < T; ++i ) {
      th[i].from = 1 + N * i;
      th[i].to = 1 + N * (i + 1);
    }
```

The threads are then started, with the main thread taking a role of a worker thread as well, in order to reduce state space size by not spawning an extra thread when it is not needed. Since memory allocation can happen during thread creation, we need to guard the loop for exceptions and clean up in case of an allocation failure.

```
    int i;

    try {
      for ( i = 0; i < T - 1; ++i )
        th[i].start();
    } catch(...) {
      for ( int j = 0; j < i; ++ j )
        th[i].join();
    }

    try {
      th[T - 1].main();
    } catch (...) {}
```

When the main thread finishes its portion of the work, it waits for the other threads and cleans up.

```
    for ( int i = 0; i < T - 1; ++i )
      th[i].join();
  } catch (...) {}

  delete t;
  return 0;
}
```

In this scenario, we can observe the huge impact of the exponential state space increase. For $T = 3, N = 1$, verification of the above test-case took multiple days using 32 cores, generated over 716 million states and used about 80GiB of RAM. On the other hand, verification for $T = 2, N = 1$ finishes in less than 3 minutes, uses 1.4GiB of RAM and generates fewer than 100 000 states.

This means that out of the desirable properties, we were able to verify that a cascade of two growths (possibly interleaved) is well-behaved when two threads access the table – using $T = 2, N = 4$ – in this scenario, a single thread can trigger a cascade of 2 growths, while other threads are inserting items. We were also able to verify that a single growth is correct (it does not lose items) in presence of 3 threads. A scenario with cascaded growths and 3 threads, however, seems to be out of our reach at this time. Nevertheless, the verification effort has given us precious insight on the behaviour of our concurrent hash table implementation.

While the hash table described in this paper was in a design and prototyping phase, we have encountered a race condition in the (prototype) implementation. The fact that there is a race condition was discovered via testing, since it happened relatively often. The problem was finding the root cause, since the

observable effect of the race condition happened later, and traditional debugging tools do not offer adequate tools to re-trace the execution back in time.[3] In the end, we used DIVINE to obtain a counterexample trace, in which we were able to identify the erroneous code.

4 Benchmarks

Earlier, we have laid out the guiding principles in implementing scalable data structures for concurrent use. However, such considerations alone cannot guarantee good performance, or scalability. We need to be able to compare design variants, as well as implementation trade-offs and their impact on performance. To this end, we need a reliable way to measure performance.

The main problem with computer benchmarks is *noise*: while modern CPUs posses high-precision timers which have no impact on runtime, modern operating systems are, without exceptions, multitasking. This multitasking is a major source of measurement error. While in theory, it would be possible to create an environment with negligible noise – either by constructing a special-purpose operating system, or substantially constraining the running environment, this would be a huge investment. Moreover, we can, at best, hope to reduce the errors in our measurement, but we can hardly eliminate them entirely.

One way to counteract these problems is to choose a robust estimator, such as median, instead of the more common mean. However, since we only posses finite resources, we can only obtain limited samples – and even a robust estimator is bound to fluctuate unless the sample is very large. Ideally, we would be able to understand how good our estimate is. If our data was normally distributed (which we know is, sadly, not the case) we could simply compute the standard deviation and base a confidence interval for our estimator on that. However, since we need a computer for running the benchmarks anyway, we can turn to bootstrapping: a distribution-independent, albeit numerically intensive method for computing confidence intervals.

While bootstrapping gives us a good method to compute reliable confidence intervals on population estimators, it does not help to make those confidence intervals tighter. Given a sample with high variance, there are basically two ways to obtain a tighter confidence interval: measure more data points, or eliminate obvious outliers. While a bigger sample is always better, we are constrained by resources: each data point comes at a cost. As such, we need to strike a balance. In the measurements for this paper, we have removed outliers that fell more than 3 times the interquartile range (the distance from the 25th to the 75th percentile) of the sample from the mean, but only if the sample size was at least 50 measurements, and only if the confidence interval was otherwise more than 5% of the mean.

[3] An extension to `gdb` to record execution exists, but we were unable to use it successfully. Either the window in which time reversal was possible was too narrow, or the memory and time requirements too high.

To assess performance of the final design with concurrent resizing, we have created a number of synthetic benchmarks. As the baseline for benchmarking, we used implementation of `std::unordered_set` provided by `libc++` (labelled "std" in results). Additionally, we have implemented a sequential open-hashed table based on the same principles as the final design, but with no concurrency provisions (tables "scs" and "sfs") – this allowed us to measure the sequential overhead of safeguarding concurrent access.

Since `std::unordered_set` is only suitable for sequential access, as a baseline for measuring scalability, we have used a standard closed-hashing table (labelled as "cus", from `concurrent_unsorted_set`) and a similar design primarily intended for storing key-value pairs, `concurrent_hash_map` (labelled "chm"), both implementations provided in Intel Threading Building Blocks [1]. The final designs presented here are labelled "ccs" and "cfs". The middle letter indicates the size of the hash table cell c for "compact" and f for "fast": the "fast" variant uses a hash cell twice as wide as a pointer, storing a full-sized (64b) hash inside the cell. The "compact" variant uses a truncated hash that fits in the spare bits inside a 64-bit pointer. (The hash inside cells is only useful in hash tables with out-of-line keys; for integer-keyed tables, they are simply overhead).

As the common performance measure, we have chosen average time for a single operation (an insert or a lookup). For benchmarking lookup at any given load factor, we have used a constant table with no intervening inserts. Four types of lookup benchmarks were done: miss (the key was never present in the table), hit (the key was always present) and a mixture of both ($\frac{1}{2}$ hit chance, and $\frac{1}{4}$ hit chance). For insertions, we have varied the amount of duplicate keys: none, 25 %, 50 % and 75 %.

All of the insertion benchmarks have been done in a variant with a pre-sized table and with a small initial table that grew automatically as needed. Finally, all of the benchmarks outlined so far have been repeated with multiple threads performing the benchmark using a single shared table, splitting workload equivalent to the sequential benchmarks, distributed uniformly across all threads. All the benchmarks have been done on multiple different computers, with different number of CPU cores and different CPU models, although we only report results from a single computer – a 12-core (2 sockets with 6 cores each) Intel Xeon machine.[4] We have chosen 4 plots to include in this paper; they can be seen in Figure 7, along with descriptions.

5 Conclusions

We have described, implemented and verified a hash table suitable for both small and large data sets, with fully concurrent lookup and insertion and with dynamic, concurrent resizing. The benchmarks we have done show that both the design and the implementation are highly competitive, and our experience with using the hash table as presented here in the implementation of a parallel explicit-state

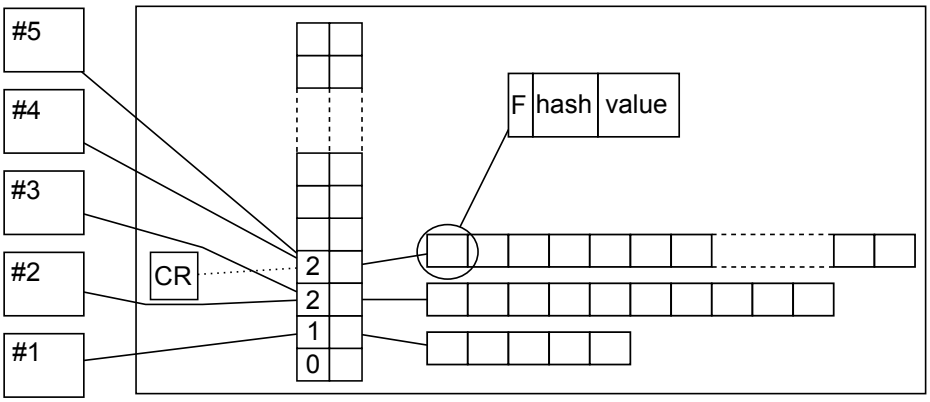
[4] The full data set will be eventually published online, but is too extensive to fit in a paper. Please check `http://divine.fi.muni.cz/benchmarks`.

model checker confirms that it is well-suited for demanding applications. The C++ source code of the implementation is available online[5] under a permissive BSD-style licence. The provided code is production-ready, although for use-cases where item removal is required, it would need to be adapted using one of the approaches described in existing literature.

References

1. Intel Corporation, "Threading Building Blocks (2014-06-01)," 2014. [Online]. Available: http://www.threadingbuildingblocks.org
2. O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," *J. ACM*, vol. 53, no. 3, pp. 379–405, May 2006. [Online]. Available: http://doi.acm.org/10.1145/1147954.1147958
3. A. W. Laarman, J. C. van de Pol, and M. Weber, "Boosting Multi-Core Reachability Performance with Shared Hash Tables," in *FMCAD 2010*, N. Sharygina and R. Bloem, Eds. IEEE Computer Society, 2010.
4. C. Purcell and T. Harris, "Non-blocking hashtables with open addressing," in *DISC 2005: Proceedings of the 19th International Symposium on Distributed Computing*, September 2005, a longer version appears as technical report UCAM-CL-TR-639. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=67422
5. H. Gao, J. F. Groote, and W. H. Hesselink, "Lock-free dynamic hash tables with open addressing," *Distributed Computing*, vol. 18, no. 1, pp. 21–42, 2005.
6. R. J. Jenkins, "A Hash Function for Hash Table Lookup," 2006. [Online]. Available: http://burtleburtle.net/bob/hash/doobs.html
7. ——, "SpookyHash: a 128-bit Noncryptographic Hash," 2012. [Online]. Available: http://burtleburtle.net/bob/hash/spooky.html
8. G. Pike and J. Alakuijala, "Introducing CityHash," 2011.
9. V. Batagelj, "The Quadratic Hash Method When the Table Size Is Not a Prime Number," *Communications of ACM*, vol. 18, no. 4, pp. 216–217, 1975. [Online]. Available: http://doi.acm.org/10.1145/360715.360737
10. P. Ročkai, V. Štill, and J. Barnat, "Techniques for Memory-Efficient Model Checking of C and C++ Code," in *Software Engineering and Formal Methods*, 2015, submitted.
11. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible Hashing -— A Fast Access Method for Dynamic Files," *ACM Transactions on Database Systems*, vol. 4, no. 3, pp. 315–344, 1979.
12. J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser, "DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs," in *Computer Aided Verification (CAV 2013)*, ser. LNCS, vol. 8044. Springer, 2013, pp. 863–868.

[5] `https://divine.fi.muni.cz/trac/browser/bricks/brick-hashset.h#L481`

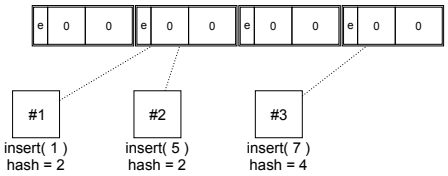


The hash table consists of two parts – global (shared) data and thread-local data. The global data are shown inside the big rectangle. All rows of the table are accessible through a row index, which is shown in the left part of the global data rectangle. There are two columns for each row – one holds a reference counter while the other stores a pointer to the row itself. The pointer CR (current row) points to a globally valid row of the hash table (this reference is not included in the reference count). Every row consists of cells, where every cell has three fields: flag, hash, and value. The flag may have four possible values: empty (e), writing (w), valid (v), and invalid (i).
The thread-local data are represented by small rectangles labeled #1 – #5, each belonging to one thread. Every thread needs to remember which row it is currently using.

Fig. 1. Overall layout of the hash table.

Let there be three threads – #1, #2, and #3 – inserting values 2, 6, and 8 respectively; insertion is happening in a row of four cell, which is initially empty. A solid connector means that a thread is manipulating a cell (i.e. the cell is locked by that thread); a dotted connector represents a thread reading a cell. For this example, we use following formula to calculate a hash of value: $(value \bmod row.size) + 1$.

1. Both threads #1 and #2 are accessing the second cell; thread #3 is accessing fourth cell:



2. Thread #1 has atomically modified the flag of the cell from 'empty' to 'writing' and stored a hash of the value so that thread #2 cannot modify the content of the cell and is forced to wait until the pending writing operation finishes:

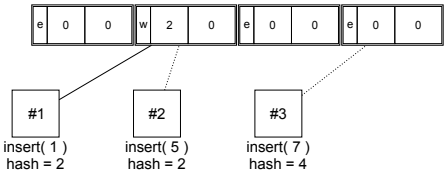
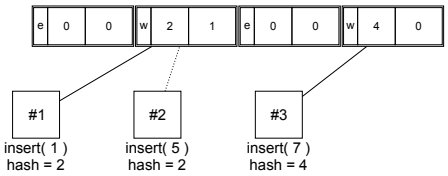
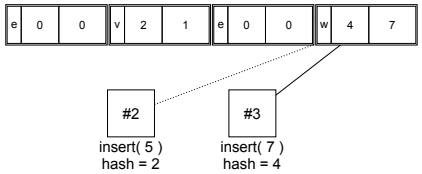


Fig. 2. Insertion algorithm, part 1.

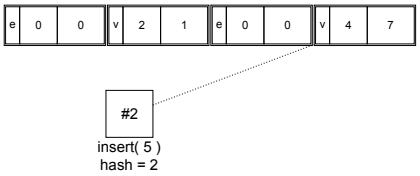
3. Thread #1 stored the value, thread #2 is still waiting. Thread #3 has atomically modified the flag of the cell to 'writing':



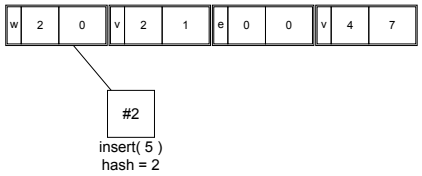
4. Thread #1 has changed the flag of the cell to 'valid' and finished the insert operation. Thread #2 found that the values are different and by using quadratic lookup, it turned to the fourth cell. Meanwhile, thread #3 has stored the value:



5. Thread #3 has finished the insert operation. Thread #2 is comparing hashes:



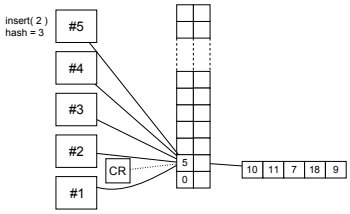
Thread #2 found an empty cell, changed the flag and stored the hash:



Thread #2 has finished the insert operation:

Fig. 3. Insertion algorithm, part 2.

1. The initial situation: all five threads are pointing to the second row which is also the current row. Thread #5 starts an insert operation:



2. As the current row is full, thread #5 signalized that the table needs to be grown, allocated a new row, and changed the value of CR to this new row:

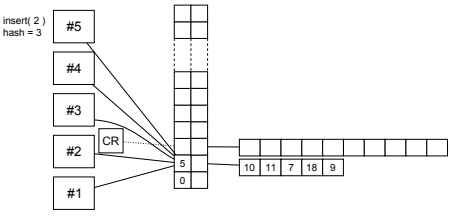
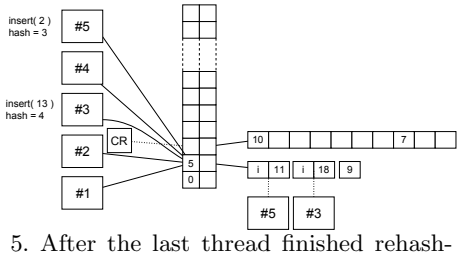
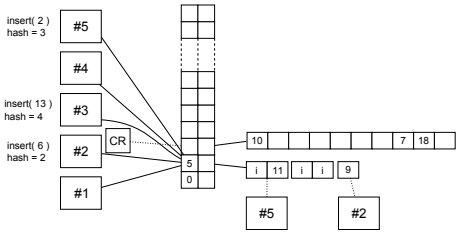


Fig. 4. Resizing the hash table, part 1.

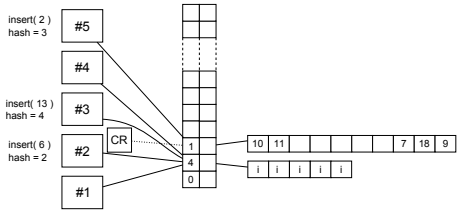
3. Thread #5 has split the original row into segments and started rehashing the first available segment. Thread #3 was about to start an insert operation, but as the table is growing, it is impossible to insert new items; thread #3 hence started rehashing the next available segment:



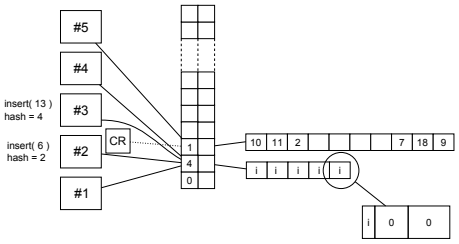
4. Thread #2 was about to start an insert operation, but it also started rehashing the next (and last, in this case) available segment. Meanwhile, thread #3 has finished rehashing and is waiting for the table to be unlocked:



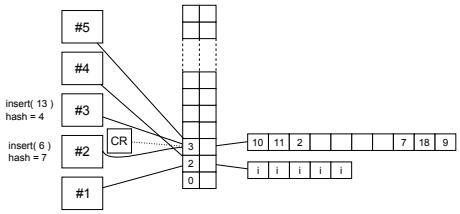
5. After the last thread finished rehashing, thread #5 unlocked the table and updated its current row index. From this moment on, the table is ready for insert and find operations (please note the reference counts for table rows: only one thread is now using the current row, so the previous row cannot be deallocated yet):



6. Thread #5 has finished its insert operation. The detail of the cell shows how an invalid state is encoded:



7. Both thread #2 and #3 have updated their current row indices, as well as the reference counters of the corresponding rows:



8. Both thread #2 and #3 have finished their insert operations. Threads #1 and #4 are about to perform a find operation, while their thread-local row pointer is still pointing at the old row:

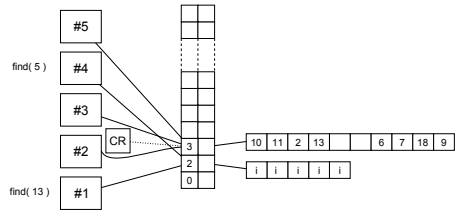
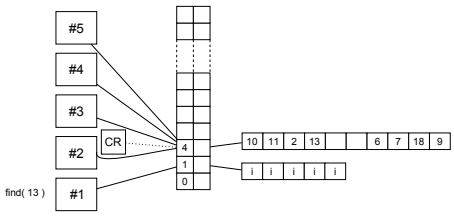


Fig. 5. Resizing the hash table, part 2.

9. Thread #4 has updated its current row index and decided that the value 5 is not present in the table:



10. Finally, thread #1 has updated its current row index and deallocated the old row. It also found the value 13 present in the table:

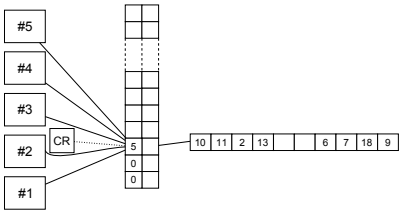


Fig. 6. Resizing the hash table, part 3.

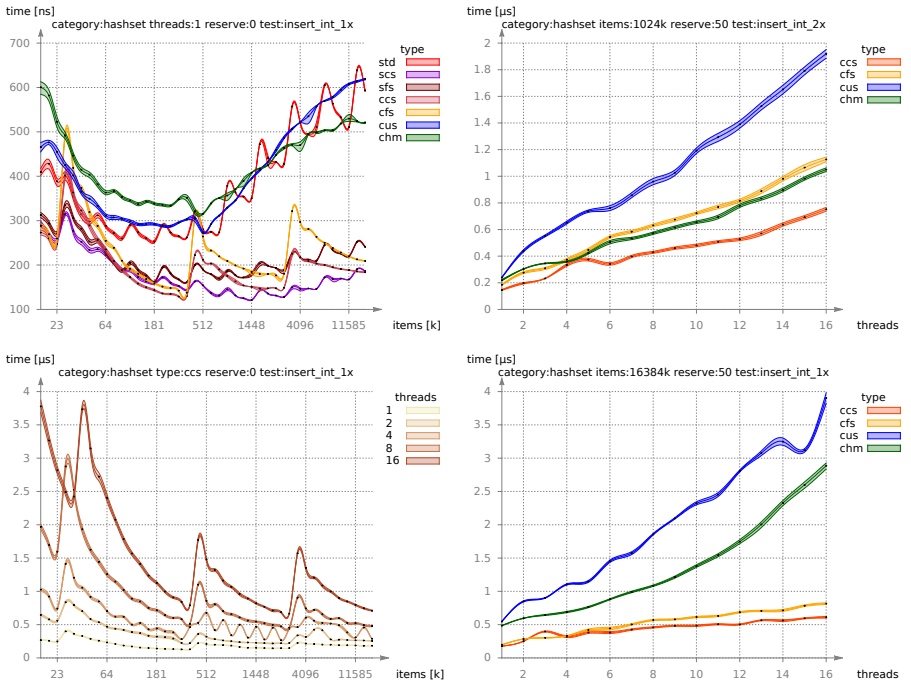


Fig. 7. Performance measurements with integer-sized keys. From top left, clockwise: 1) comparison of sequential performance of various hashtable designs 2) insertion with 50 % key duplicity rate, with 1 million items and a pre-sized hashtable with half a million cells 3) insertion with no key repeats, 16M items 4) behaviour of the final design (ccs) as a function of hash table size and a number of threads (no reserve, the hashtable is resized dynamically). The implementations are labelled as follows: std = `std::unordered_set`, scs = sequential compact set, sfs = sequential fast set, ccs = concurrent compact set, cfs = concurrent fast set, cus = `tbb::concurrent_unordered_set` and chm = `tbb::concurrent_hash_map`.