Providing an Efficient Lockless Hash Table for Multi-Core Reachability in Java

Bas van den Brink
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
b.vandenbrink-1@student.utwente.nl

ABSTRACT

In order to gain substantial performance when using state space explorators in model checkers by using multiple cores, the data structure that is being used has to be designed in such a way that it can be used concurrently. Currently, an efficient lockless hash table for state space explorators in a model checker has been implemented in C. This article provides a Java variant of this hash table that is scalable in the number of cores by limiting cache updates and refraining from using locks. Benchmarks are designed and executed to show this scalability.

Keywords

hash table, multi-core data structures, lockless algorithms, parallel programming, Java

1. INTRODUCTION

This article describes a Java variant of an implementation of an efficient lockless hash table, used for state space explorators in model checkers. According to Laarman et al., *"[a state space explorator] searches through all the states of the program under verification to find errors or deadlocks"* [5]. In order to memorise states that have already been visited, every visited state is stored as a state vector in a data structure, which generally is a hash table. Boosting the speed of reachability algorithms can be accomplished by making use of multiple processor cores. However, this requires a data structure that provides thread safe access and is efficient when using multiple cores. According to Oracle, *"a procedure is thread safe when the procedure is logically correct when executed simultaneously by several threads"* [6]. This implies that no data races may occur when multiple threads access the data structure.

Concurrent data structures use synchronisation mechanisms in order to prevent data races. When coarse-grained synchronisation is used, large parts of the data structure are locked when a thread accesses the data structure. In the context of reachability problems, this significantly decreases performance. This performance decrease is even greater when large state vectors are stored in the hash table. This is the result of the whole hash table being locked for a significant amount of time when large state vectors are written to the hash table, compared to cases where hash tables are used to store relatively small data.

Therefore, fine-grained synchronisation, where only small parts are locked at a time, is more suitable for this purpose. However, Laarman et al. stated that this would still be infeasible since *"the lock itself introduces another synchronization point; and synchronization between processor cores takes time"*. Fine-grained synchronisation still requires that processor caches are updated with every access of the concurrent data structure. This is needed, since every processor core has its own cache. Therefore, changes in the state of a lock would otherwise not be observed by other cores and thus data races may occur.

Hence, an efficient hash table has been proposed in by Laarman et al. that is lockless [4]. The implementation has been accomplished in C. While an interface could be provided to use this implementation in Java, this solution does not benefits from a number of advantages of the Java language, such as cross-platform portability.

Therefore, a Java implementation of this hash table is a better solution to use this hash table in model checkers that are written in Java. However, since Java is a cross-platform language, it uses a higher abstraction level of the system it is running on. Hence, low-level APIs to control cache updates, which are used in the C variant, are not available in Java.

This paper therefore provides a Java variant of Laarman's hash table (algorithm 3 and 4). This hash table could be used by any model checker that is written in Java and scales well over the number of cores, because of its locklessness and the limited number of cache updates. Furthermore, this paper provides benchmarks of this hash table to show its scalability (section 4.2).

2. BACKGROUND

Atomic operations can be used to implement concurrent algorithms without using expensive locks. While this increases performance, it is more difficult to implement a correct algorithm using atomic operations than implementing an algorithm that uses locks. Laarman's hash table uses such atomic operations. Section 2.1 describes a method, which is used by this hash table, to use atomics in C. Java does provide different methods to implement atomic operations. These methods are describes in section 2.2.

Laarman's hash table itself is described in section 2.3. Furthermore, the basis of our hash table, Click's hash table, is described in section 2.4.

2.1 Atomics in C

A common atomic operation that is used by Laarman's hash table is the `compareAndSwap` (`CAS`) operation. Many

21st Twente Student Conference on IT June 23, 2014, Enschede, The Netherlands.

modern CPUs implement this operation as a processor instruction, which ensures atomic memory modification and preserves data consistency if used in a correct manner. This operation has three parameters: `int* reg`, `int oldval` and `int newval`. `reg` is the location to be modified, `oldval` the expected old value and `newval` the value that is to be set at `reg`. Only one thread can execute a `CAS` operation at a time. `CAS` checks whether `*reg` is equal to `oldval` and sets `newval` when this results true. `CAS` returns whether the operation has succeeded.

2.2 Atomics in Java

Since Java is a high-level language, it uses a higher abstraction level of atomic operations. These operations are bundled in the toolkit `java.util.concurrent.atomic`. According to Oracle, this is *"[a] small toolkit of classes that support lock-free thread-safe programming on single variables. In essence, the classes in this package extend the notion of `volatile` values, fields, and array elements to those that also provide an atomic conditional update operation of the form: boolean `compareAndSet(expectedValue, updateValue)`"* [7]. The `compareAndSet` method has the same semantics as the `CAS` operation implemented in C 2.1, but has stronger reordering constraints.

According to the Java Language specification [2], the following is guaranteed when applying `volatile` to a field: *"A write to a `volatile` variable `v` synchronizes-with all subsequent reads of `v` by any thread (where "subsequent" is defined according to the synchronization order)."* In short, when thread `A` write a `volatile` variable `v`, thread `B` sees this write when it is reading this variable after the write. This is accomplished by enforcing a memory fence. A memory fence causes the compiler and CPU to enforce an ordering constraint on memory operations issued before and after the fence instruction. Therefore, the processor caches are updated. Furthermore, the writes and reads of `volatile` variables are atomic. The `atomic` toolkit uses `volatile` variables underneath.

Furthermore, `atomic` classes use the `Unsafe` class for its implementation. According to docjar.com, which contains documentation for the OpenJDK API, `sun.misc.Unsafe` is *"A collection of methods for performing low-level, unsafe operations. Although the class and all methods are public, use of this class is limited because only trusted code can obtain instances of it."* [3]. It is however possible to use instances for our hash table. `Unsafe` provides methods as `compareAndSwapObject` and `compareAndSwapInt`, which are natively implemented by the Java Virtual Machine (JVM). The JVMs from Oracle and OpenJDK for x86 processors use the `compareAndSwap` processor instruction to execute these methods.

`Unsafe` is used by the `AtomicInteger` class, which is a class part of the `atomic` toolkit, that supports lock-free thread-safe programming with integers, to implement the `compareAndSet` operation. This is accomplished by using `compareAndSwapInt`.

2.3 C variant hash table

Laarman et al. proposed the C variant of the lockless hash table [4]. This hash table only contains one operation, the `find-or-put` operation. This operation searches for a value. If it succeeds, `true` will be returned. Otherwise, the value will be put into the hash table and `false` will be returned. The pseudo code is shown in algorithm 1. Instead of utilising locks for providing thread safety, every key-value pair contains a state, which can be *Empty*, *Write* or *Done*. These states are encoded in the memoized hashes.

```
Data: size, Bucket[size], Data[size]
Input: vector
Output: seen
 1  num ← 1;
 2  hash_memo ← hash_num(vector);
 3  index ← hash mod size;
 4  while true do
 5  |  foreach i in walkTheLineForm(index) do
 6  |  |  if empty = Bucket[i] then
 7  |  |  |  if
    |  |  |  CAS(Bucket[i], empty, ⟨hash_memo, write⟩)
    |  |  |  then
 8  |  |  |  |  Data[i] ← vector;
 9  |  |  |  |  Bucket[i] ← ⟨hash, done⟩;
10  |  |  |  |  return false;
11  |  |  |  end
12  |  |  end
13  |  |  if hash_memo = Bucket[i] then
14  |  |  |  while ⟨−, write⟩ = Bucket[i] do
    |  |  |  ...wait...;
15  |  |  |  if ⟨−, write⟩ = Bucket[i] ∧ Data[i] =
    |  |  |  vector then
16  |  |  |  |  return true;
17  |  |  |  end
18  |  |  end
19  |  end
20  |  num ← num + 1;
21  |  index ← hash_num(vector) mod size;
22  end
```

Algorithm 1: The find-or-put algorithm [4]

The `find-or-put` operation first hashes the value, which is a state vector, and then checks the corresponding state. Since the hashes are memoized, hashes are only calculated once per `find-or-put` operation. If the state equals *Empty*, `find-or-put` uses `CAS`, explained in section 2.1, to set the state from *Empty* to *Write* (line 9). If it succeeds, `find-or-put` puts the value in the hash table and sets the state to *Done*. If it fails, apparently another thread has set the state to *Write*. Then, it waits until the state has been set to *Done* (line 14) and checks whether the corresponding state vector equals the one that has to be put. If this is `true`, `find-or-put` has found the value and returns `true`. Otherwise, linear probing will be used to find or put the value.

```
Data: cache_line_size, Walk[cache_line_size]
Input: index
Output: Walk[]
1  start ←
   ⌊index/cache_line_size⌋ × cache_line_size;
2  for i ← 0 to cache_line_size − 1 do
3  |  Walk[i] ← (start + index + i)
   |  mod cache_line_size;
4  end
```

Algorithm 2: Walking the (cache) line [4]

In order to achieve high performance, the number of cache line updates are hereby limited. Algorithm 2 shows how *walking the cache line* is accomplished. Instead of linear probing across the border of cache lines, first the whole cache line will be searched. If the value or an empty bucket is not found within the cache line, the algorithm will probe within a new part of the cache. Therefore, the cache has only to be updated when an empty bucket or the value is

not found within a cache line.

Table 1 gives an example of the hash table. When using linear probing, `find-or-put` first walks the cache line before comparing hashes stored at another cache line. This is important, because it minimises the number of expensive cache updates. In this example, using `find-or-put (v5)` results in starting search at index 3. Index 2 will be searched before jumping to cache line 1. The probe order will therefore be $3 \rightarrow 2 \rightarrow 4$.

Table 1. Example of a hash table

Cache line	Index	Key	Value	State
0	0	k1	v1	Done
	1	k2	v2	Done
	2	k3	?	Write
	3	k4	v4	Done
1	4	k5	v5	Done
	5			Empty
	6	k6	v6	Done
	7	k6	v7	Done

2.4 Non-blocking hash table in Java

A non-blocking hash table and hash map has already been implemented in Java by Click [1]. This hash map has the same operations and can therefore be used as an alternative for `java.util.ConcurrentHashMap`. However, Click's hash map performs better than `ConcurrentHashMap` when using many threads and performs as least as good when using few threads. Click's hash table is a variant of `ConcurrentHashMap` which has the same behaviour as `java.util.Hashtable`. The main difference is that Click's hash table does not use locks at all, while `ConcurrentHashMap` uses locks when writing entries.

The non-blocking hash table of Click uses one array of `Objects` for storing hashes, keys and values. Hereby, the first entry contains a `class` named `CHM` for storing information about the size of the hash table and defining methods to efficiently resize the hash table when it is full. The second `Object` contains memoized hashes of the keys. All other even entries contain keys and all odd entries contain the corresponding values. Table 2 shows the structure of this `Object` array.

Table 2. Object array Click's hash table

index	content
0	`CHM class`
1	[hash 0, hash 1, ... , hash n]
2	key 0
3	value 0
4	key 1
5	value 1
...	...
2n + 1	key n
2n + 2	value n

Since an array is an object in Java, Click's hash table only stores references to vectors, which are integer arrays, in the hash table. Therefore, storing a value in the hash table is done atomically with a `CAS` operation. This would not be possible in Laarman's hash table, since his `find-or-put` operation copy the whole vector. Doing this atomically would lock the whole hash table for a significant amount of time.

In order to ensure correctness, key-value pairs cannot be removed from the hash table. Instead of removing this key-value pair, the value will be replaced by a tombstone, a specific `Object` for marking deleted values. Figure 1 shows the states a key-value can have. At first, both the key and value are set to `null`. Thereafter, `Unsafe` is used to do a `CAS` operation to set the key. If it fails, another thread inserted a key just before this thread. This thread then has to check whether the key and value that are put by that thread are the same as we wanted to insert. If it succeeds, this thread knows that this key is never going to be removed. Then, this thread can set the corresponding value by using `CAS`. If this succeeds, the key-value pair has been put. Otherwise, another thread has inserted the same key with another value, which will be overwritten.

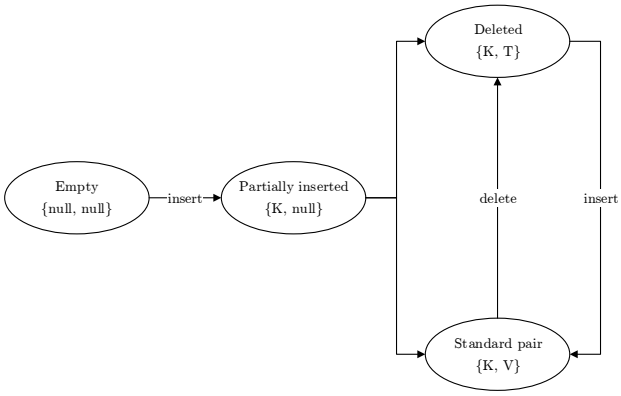

Figure 1. States hashtable Cliff Click

3. PROBLEM STATEMENT

Our research goal is to provide an efficient lockless hash table in Java for state space explorators. We want this hash table to be scalable in the number of cores. Therefore, the number of cache updates has to be limited. Instead of the C operation `compareAndSwap`, Java variants of this operation can be used can be used, which are explained in section 2.2. It is however important to use alternatives that limit the number of cache updates.

Therefore, using `AtomicInteger`, explained in section 2.2, would not be a good alternative. Spinlocking by executing the `get()` method when another thread has won the competition when setting the state to *Write*, is rather inefficient. This is the result of `get()` updating the cache with every call, because `AtomicInteger` uses a `volatile` value underneath. Furthermore, Java does not provide the same functionality as C for walking cache lines as used in algorithm 2. This is the result of Java being a high level object-oriented language. Hence, low-level memory access is not provided. This results in the following question: *how can the number of cache updates be reduced when looking up values in our hash table in Java?*

In the hash table implemented in C, cache updates are explicitly defined. One situation to update the cache is when some thread has changed the state of a certain row in the hash table. Therefore, the spinlock used for waiting until another thread completes writing a value does not have to update the cache by every call of the `get()` method. `get()` however does result in a memory fence and therefore updates the cache. This has a negative impact at the scalability in the number of cores. The second question that is answered by this paper is therefore: *in Java, how can thread-safety of the hash table be achieved while accomplishing close to linear scalability in the number of cores?*

Furthermore, in order to be able to verify whether our research goal is achieved, benchmarks have to be executed. Since only the hash tables will be benchmarked and not the state space explorators that use them, the state space explorators have to be simulated in order to benchmark the hash tables in a manner that reflects the situation it is going to be used in. Therefore, the third research question is: *how can our hash table be benchmarked with respect to its intended purpose?*

The last research question continues on the result of the third research question. The fourth research question is: *using the techniques that are the result of the third research question, how does the Java variant of the hash table perform in terms of time and scalability in the number of cores?*

4. METHODOLOGY

The research consists of two parts: implementing an efficient lockless hash table and measuring its performance in time and scalability. Therefore, this section is divided in two parts: section 4.1, hash table and section 4.2, benchmark.

4.1 Hash table design

As explained in section 7 and in background section 2.4, Click implemented an efficient non-blocking hash table for general use in Java. This hash table is a concurrent version of `java.util.Hashtable` and therefore offers much functionality we do not need. However, it is significantly faster and scales much better than `java.util.Concurrent HashMap`, the concurrent version of `java.util.HashMap`, since is does not use locks. Instead, it uses the `CAS` operation 2.1 to update its values. Therefore, this hash table forms the basis of our hash table. There are however a couple of differences between our hash table and Click's hash table:

- Every feature of `java.util.Hashtable` is implemented by Click's hash table. Therefore, the hash table also supports operation to remove values, resize the hash table and more which we do not need. When these operations are not supported, the hash table can be more efficient.
- A method that implements `find-or-put`, explained in section 2.3, is needed for our hash table. This method is not implemented by Click's hash table.
- Since our hash table only contains vectors, no key-value pairs are inserted, but only values. Performance gain can be accomplished by only storing values and less memory is needed by the hash table.

There are two variants of the hash table. We will focus at their variant of the `find-op-put` operation. Both variants use an array of integer arrays to store the vectors. However, one variant, algorithm 3, uses an `AtomicIntegerArray` to store the hashes, which stores integers as `volatile`s.

The other variant, algorithm 4, uses a regular integer array for storing the hashes. The latter however uses `Unsafe`'s `putIntVolatile()` when putting an hash value in the array to enforce a memory fence. Section 6 explains the differences in more detail.

Both variants of the `find-or-put` operation have the same structure. Since classes of the `Atomic` toolkit use `volatile` variables underneath, every read results in a memory fence.

Data: $size, Table[size][], Hashes$: $AtomicIntegerArray(size)$
Input: *vector*
Output: *seen*

```
1  hash_memo ← hash(vector);
2  index ← hash_memo & (size − 1);
3  while true do
4  |  if null = Table[index] then
5  |  |  if CAS(Tablet[index], null, vector) then
6  |  |  |  Hashes.set(index, hash_memo);
7  |  |  |  return false;
8  |  |  end
9  |  end
10 |  while Hashes.get(index) = 0 do ...wait...;
11 |  if hash_memo = Hashes.get(index) &
   |  vector = Table[i] then
12 |  |  return true;
13 |  end
14 |  index ← (index + 1) & (size − 1);
15 end
```

Algorithm 3: The java variant of the find-or-put algorithm using `AtomicIntegerArray`

Data: $size, Table[size][], Hashes[size]$
Input: *vector*
Output: *seen*

```
1  hash_memo ← hash(vector);
2  index ← hash_memo & (size − 1);
3  while true do
4  |  if null = Table[index] then
5  |  |  if CAS(Tablet[index], null, vector) then
6  |  |  |  putIntVolatile(index, hash_memo);
7  |  |  |  return false;
8  |  |  end
9  |  end
10 |  while Hashes[index] = 0 do ...wait...;
11 |  if hash_memo = Hashes[index] & vector =
   |  Table[i] then
12 |  |  return true;
13 |  end
14 |  index ← (index + 1) & (size − 1);
15 end
```

Algorithm 4: The java variant of the find-or-put algorithm using `putIntVolatile`

Therefore, this toolkit has not been used for storing state vectors. Instead, `Unsafe`, explained in section 2.2, has been used.

At first, the hash value of the vector is calculated. This value is used to calculate the index of the array. Then, a `CAS` operation will be executed with expected value `null` and `vector` as the new vector. The `CAS` operation is needed to ensure correctness. If this succeeds, this entry of the array was empty and the new value has been put. Otherwise, there already exists a vector on that entry. This vector may have been put just before we wanted to put a vector. Therefore, it is important that the `CAS` operation is used, which enforces a memory fence.

After the negative result of the `CAS` operation, the hash of the vector will first be compared with the hash of our vector. This can be done fast, since an hash as an integer. If they do not equal, the vectors cannot be equal either. Otherwise, an expensive comparison between the vectors will be done. If they are the same, the `vector` has been found and `true` will be returned. Otherwise, linear

probing will be used to try other entries of the array.

The main difference with the hash table of Laarman et al. is that only reference to vectors are stored by our hash table, instead of the whole vector. This is caused by the fact that an array is an object in Java. Therefore, putting the vector in the hash table can be done in a single `CAS` operation. We therefore do not need to implement the *Empty*, *Wait* and *Done* states.

4.2 Benchmark design

The original hash table is designed to be used by LTSmin, a model checker. However, LTSmin is implemented in C and can therefore not use our hash table. Therefore, LTSmin is edited in order to output state vectors that will be stored in the hash table to a file. This file then contains every state vector from states that the model checker has visited and may include duplicated. Since reading files is slow, this file will be preloaded as an array of integer arrays prior to the actual benchmark.

OpenJDKs JHM API is used for the actual benchmarks. According to the website, *"JMH is a Java harness for building, running, and analysing nano/micro/milli/macro benchmarks written in Java and other languages targetting the JVM."* [8]. JHM allows us to only implement the benchmark itself and use JHM as the framework which executes the benchmark. Furthermore, JHM warms up the just in time (JIT) compiler of the JVM. Warming up the JIT compiler ensures that the timings correctly reflects the execution timings when used in a production environment. After a few iterations, code runs faster since it has been compiled by the JIT compiler.

From the BEEM database, three examples are randomly chosen. These examples are then explored by using the modified LTSmin. Every state vector that is encountered is then written to a file. These files form the input for the three benchmarks. Since the hash table lookups are redirected to the files, these files contain duplicate vectors, in the same order they are put. This ensures that the benchmarks reflects the original state space explorator.

For each benchmark, the corresponding vector file is divided in eight chunks. Each thread gets one chunk to lookup in the hash table. This means that every thread lookups the same amount of vectors. Therefore, if threads are not given an equal amount of processor time by the thread scheduler, one thread is done earlier than another thread. In a real situation, a load balancer ensures that every thread has to do some task. Therefore, a simple solution is implemented to simulate the benchmark by looking up random vectors from the vector file after a thread finished its work. However, after measuring how many random vectors a thread lookups, it can be concluded that this is only a very small percentage. Therefore, there is no need to simulate the load balancer. Current benchmarks do not simulate the load balancer.

Table 3 shows some details about the benchmarks.

Table 3. Details per benchmark

BEEM file	Length	Vectors	Uniques
firewire_tree.2.dve	83	5693	2441
elevator2.2.dve	16	1036801	179200
hanoi.2.dve	53	1594321	531441

The setup of the computer used for the benchmarks is as follows:

CPU Intel Core i7-2630QM @ 2.00 GHz, with 4 cores and 8 threads using hyperthreading.

RAM 4 GB

OS Windows 8.1 x64

JVM Oracle JDK 7 Update 60 (HotSpot)

5. RESULTS

Both variants of the hash table are benchmarked.

The results of the two sets of benchmarks are represented in figure 2 and figure 3. Every benchmark is executing five times, each time using a different amount of threads. Each benchmark is executing using one, two, four, six and eight threads. At the y-axis, the average time per operation in $\mu s/operation$ is shown. This indicates the speed of the hash table.

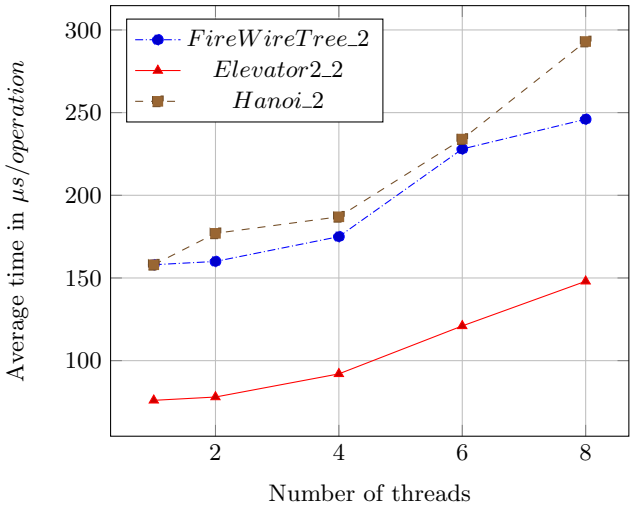

Figure 2. Benchmark results putVolatileInt variant

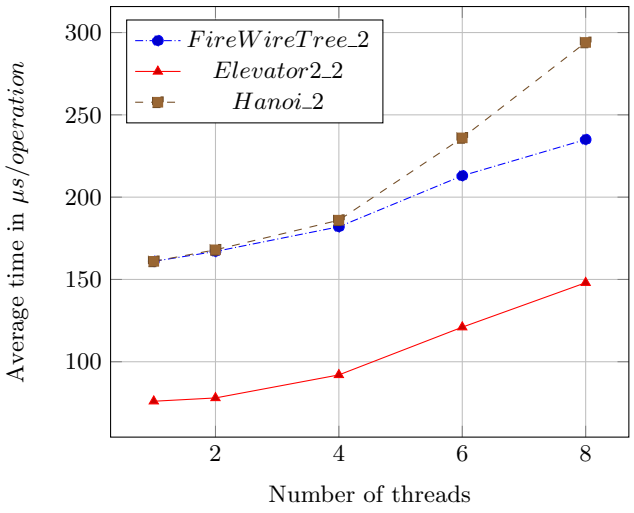

Figure 3. Benchmark results AtomicIntegerArray variant

The curve of the three benchmarks do have almost the same shape. Increasing the number of threads from one to two results in a minor increase of time per operation. Increasing the number of threads from two to four results in a more significant increase of time per operation. It has to be mentioned that the overall amount of work doubles

when the number of threads doubles, since every thread always gets the same amount of vectors allocated, even when the number of threads increase. The curves become steeper when the number of threads is greater than 4.

6. DISCUSSION

This section is aimed at discussing the correctness of our hash table. Furthermore, the results of the benchmarks are interpreted in section 6.2. Moreover, some test cases are designed. These are discussion in section 6.3.

6.1 Correctness

While lockless data structures, if well designed, can perform significant better that data structures that do use locks, it is less clear whether they are correct or not. A lock can ensure that only on thread can access (parts of) the hash table. By locking the correct components of the hash table, correctness can be show straightforwardly. Since multiple threads can access a lockless hash table simultaneously, correctness is more difficult to prove. To ensure correctness, this has to be mathematical proven for every state the hash table can occur in. This is however outside the scope of this research.

Therefore, this section describes a critical situation and shows that our hash table is correct in this situation. It will therefore be plausible, not proven, that our hash table is correct. Only the correctness of algorithm 4 will be shown. It should follow that algorithm 3 is correct too, since there are minor differences in semantics. The greatest difference is that algorithm 3 has an extra memory fence at line 10. This section will show that this memory fence is redundant.

A critical situation is shown in figure 4. Here, thread `t1` uses the `CAS` operation to put a value in the hash table. Then, it updates the corresponding hash value. However, before the update, thread `t2` reads this hash value. The hash value is used for a first check to determine whether two vectors are equal. When the hashes are not equal, the vectors cannot be equal too. Therefore, thread `t2` will wait by spinlocking until the hash value is updated too (line 10). It is therefore important that updating the hash value enforces a memory fence. Reading a hash value does not need to enforce a memory fence, since every change already enforces a memory fence. Therefore, a read always sees the most recent value.

After the hash value is updated, thread `t2` will not spinlock any more, since this value does not equal zero. Since thread `t2` wants to `find-or-put` a different vector than `t1`, the hashes do not equal either. Therefore, the index, which is based on the hash value, will be updated. Then, `CAS` is used for updating the vector. No other thread is putting a vector in the same bucket, thus the `CAS` operation returns `true`. As the final step, `t2` updates the hash value.

We have now shown, but not proven, that our hash table is correct in this situation with two threads. However, a situation where a thread reads the hash table just before another thread has updated it, is very rare when using a consumer computer. Therefore, there is a possibility that other situations arise at different hardware that are not foreseen. Excluding those erroneous situations requires mathematical validation.

6.2 Benchmarks

Section 5 shows the results of the benchmarks. There is not much difference between the benchmarks of the `AtomicIntegerArray` and `putIntVolatile` variant of the hash table. Therefore, using `AtomicIntegerArray` results in almost the same performance as using `putIntVolatile` for storing the hashes, in our case.

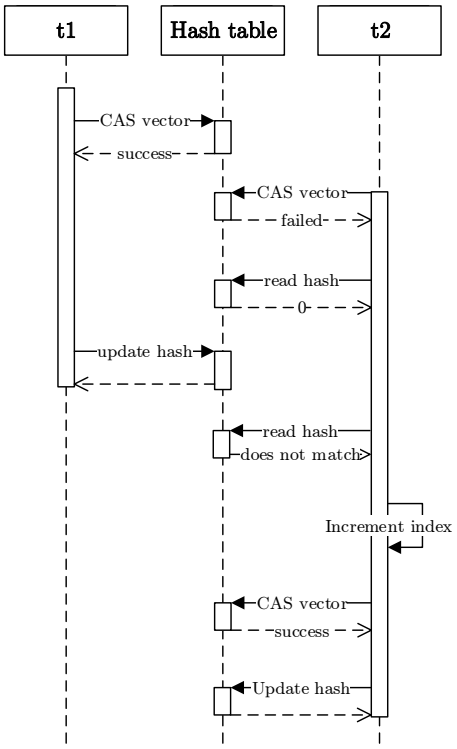

Figure 4. A critical situation

There could however exist cases in which the `AtomicIntegerArray` has low performance. Since `AtomicIntegerArray` uses a `volatile` variable for each element, reading a value from the array enforces a memory fence. Therefore, the cache of every processor core will be updated. At line 10, every `get()` thus enforces a memory fence. When this variant is used on a machine with many processor cores, a situation as the one described by figure 4 is more likely to occur. Then, one of the threads will spinlock at the `get()` operation. Since this operation enforces a memory fence, all the caches of every core will be updated many times. Therefore, the variant that uses the `putIntVolatile` operation is more efficient in this case.

The lines in both graphs in section 5 become steeper when the number of cores exceeds four. This could be the result of the fact that the benchmarks are run at a machine which only has four cores with hyperthreading. Hyperthreading enables the processor to behave as having more cores, in our case eight. Accordingly, each thread has less processor cycles per second compared to the case where at most four cores are active, while the amount of work per thread remains equal. Therefore, it cannot be concluded that the scalability decreases when enabling more than four cores.

6.3 Test cases

In order to test whether the hash table behaves correctly, two test cases are set up. The first test case uses a file with 4880 state vectors. Half of the state vectors are duplicate. Eight threads are then started with 1220 vectors assigned to each of the thread. Each thread then concurrently stores its chunk of vectors one at a time. The output shows that half of the vectors are already found in the hash table. Therefore the hash table has succeeded this test.

The second test case is intended to detect possible deadlocks. This test uses a vector file with 1594321 state vectors. The method for collecting these vectors is described in section 4.2. The BEEM example that has been used is hanoi.2. Each threads gets one hundredth of the vector file assigned. Then, these threads are started. The result of this test was that the test terminated correctly and no exceptions were thrown. While this does not show that no deadlock could occur, it is an indication that our hash table works well on a machine with multiple cores.

7. RELATED WORK

An efficient lock-free hash table implemented in Java already exists [1]. However, this is a multi-purpose hash table. Therefore, this hash table has much overhead considering the purpose of our hash table, since it provides operations that we do not need. An example of an operation we do not need is the remove operation, which removes a value from the hash table. Nevertheless, this hash table forms the basis of the hash table described by this paper and is therefore explained in more detail in section 2.4. Another extensible lockless hash table has been proposed by Shalev et al. [9]. However, this hash table is implemented in C++ and is therefore less suitable for using as basis for our hash table.

8. CONCLUSIONS

We designed a Java variant of the lockless hash table proposed by Laarman et al. [4]. This hash table is aimed for concurrent state space explorators for model checkers. Furthermore, a state space explorator is simulated in order to benchmark the hash table in way respecting its intended purpose. The remaining part of this section is aimed at answering the research questions.

1. How can the number of cache updates be reduced when looking up values in our hash table in Java?
Our hash table is designed in a way that only updates of vectors (the values) and hash values result in memory fences. A memory fence causes a central processing unit or compiler to enforce an ordering constraint on memory operations issued before and after the fence instruction and therefore forces the caches to update. Hence caches are not forced to update when reading values.

2. In Java, how can thread-safety of the hash table be achieved while accomplishing close to linear scalability in the number of cores?
Thread-safety is achieved by using atomic operations that enforces a memory fence when updating values and hashes. This enforces the synchronised-with relationship. Hence, these updates are visible for other threads. Scalability that is close to linear is accomplished by refraining from using locks and limiting the number of cache updates. Benchmarks show that our hash table is scalable when using a limited number of cores.

3. How can our hash table be benchmarked with respect to its intended purpose?
The hash table is intended for state space explorators. Since the state space explorator that uses the hash table of Laarman et al. [4] is written in C, and our hash table is intended to be a Java variant of that hash table, we simulated this state space explorator. Files with state vectors are generated from BEEM examples, as explained in section 4.2. These are used as input for the benchmark to put in the hash table.

4. Using the techniques that are the result of the third research question, how does the Java variant of the hash table performs in terms of time and scalability in the number of cores?
Benchmark results show that the hash table scales linear when using a limited number of cores. We did not benchmark the hash table with a machine that has more than four CPU cores. We expect however that the hash table will scale linear too when used on a machine that has more CPU cores, up to a certain limit.

8.1 Future work

As mentioned in section 6.1, the correctness of our has table is not mathematical verified. Therefore, we cannot be sure that our hash table performs well in every situation. While this hash table is tested, as explained in section 6.3, these test cases are executed on a machine with a limited number of cores. While not expected, usage of this hash table in environment with many cores could lead in unexpected results. Therefore, future work can be accomplished in the verification of this hash table.

While we expect that the scalability in our hash table holds when using it on a many-core machine, the hash table is only benchmarked on a machine with four cores. It is therefore recommended to benchmark this hash table on a machine with many cores. Furthermore, our hash table has not been compared with Laarman's hash table. This can be accomplished by porting the benchmark to the C language and executing both versions of the benchmark while using the same vector files. The results can be used to compare the two variants.

9. REFERENCES

[1] C. Click. A lock-free hash table. In *JavaOne Conference*, 2007.

[2] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. The java language specification - java se 7 edition, 2013. [Online; accessed 9-June-2014].

[3] Jax Systems LLC. sun.misc.unsafe, 2011. [Online; accessed 29-May-2014].

[4] A. Laarman, J. van de Pol, and M. Weber. Boosting multi-core reachability performance with shared hash tables. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 247–256, Austin, TX, 2010. FMCAD Inc.

[5] A. Laarman, J. van de Pol, and M. Weber. Parallel recursive state compression for free. In A. Groce and M. Musuvathi, editors, *Model Checking Software*, volume 6823 of *Lecture Notes in Computer Science*, pages 38–56. Springer Berlin Heidelberg, 2011.

[6] Oracle. In *Multithreaded Programming Guide*, page 205. Oracle, October 2012.

[7] Oracle. Documentation of java.util.concurrent.atomic, 2013. [Online; accessed 23-March-2014].

[8] Oracle. Openjdk: Jhm, 2014. [Online; accessed 30-May-2014].

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