Scalable Concurrent Hash Table in Java

Willem Siers
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
w.h.siers@student.utwente.nl

ABSTRACT

Sequential performance of CPU's is reaching its limits, which makes concurrent computing a necessity for improving software performance. This requires not only concurrent, but also scalable data structures. Concurrent hash table implementations exist in Java, however these suffer from poor scalability. In this research we developed a new hash table with improved scalability over existing implementations and high sequential performance. The goal was to implement a hash table design by Laarman et al. in Java in order to provide developers with new means for creating scalable concurrent software, while also offering the benefits Java provides such as portability and maintainability. Throughput and speedup of the new hash table was compared with three existing implementations. With a maximum speedup 14.3 on 32 cores, the new implementation greatly improved upon existing implementations. Results from Laarman's paper regarding a C implementation of the hash table design in LTSmin were also compared. This literature-based comparison found that their benchmark achieved larger speedup than we obtained with our implementation. Even though Java used to be stigmatized as a slow language, we found that it provides sufficient means for creating scalable algorithms, and usage should be considered over lower-level languages, as it allows for great scalability without sacrificing portability and maintainability.

Keywords

Concurrency, Scalability, Hash table, Cache optimization, Java

1. INTRODUCTION

This research describes a new concurrent hash table implementation for Java, that is designed to achieve high performance scaling on up to 64 CPU cores. The aim is to improve upon the available thread-safe Java hash tables: the two implementations of the Java Base Library's lock-based *java.util.Hashtable* [16] and the more recent lockless *java.util.concurrent.ConcurrentHashMap* [15], as well as a non-standard library concurrent hash table developed by Dr. Click [4, 5].

25th Twente Student Conference on IT July 1st, 2016, Enschede, The Netherlands.

Aside from providing Java developers with a highly-scalable hash table as a tool to create scalable software, low runtimes in our benchmarks provides a good argument in favour of using Java as a sensible choice when designing software that requires high performance. If high speedup is achieved when running the algorithm on multiple cores, our research provides evidence for Java as a suitable language for developing distributed and multi- and many-core software.

The design of the Java hash table based on the design in the paper of Laarman et al. in which a lockless hash table algorithm is described, developed for the LTSmin model checker [12]. In their paper it is shown that their C implementation outperforms two state-of-the-art multi-core model checkers SPINS [8] and DiVinE [1]. Low-level optimizations are used in the implementation, most notably cache line optimizations, and the current research depends on the extent to which these low-level optimizations are achievable in Java and if the same effect on performance can be observed, also possibly in the absence of some low-level features. In the past Java has been stigmatized for being a slow language, however through developments on the Java Virtual Machine and its Just-in-time compiler it is now considered as having a comparable or equal performance to lower-level languages like C [21].

For many programmers Java is a more attractive language to work with than C, among other because of its portability, maintainability and its widespread popularity. Java is also widely taught in higher education institutes during introductory programming courses [7]. Attracting more students to work on LTSmin may be desirable and help the development of LTSmin. Additionally, other software that requires a high-performance hash table, that is, or will be, written in Java, can benefit from the resulting hash table implementation.

Benchmarks are performed in order to compare the new hash table with existing Java implementations, and also a theoretical comparison is done by comparing our results to those found by Laarman [12]. The Java implementation, described fully in section 4.1, uses Java's *Unsafe* API for its compare-and-swap and memory allocation methods, and supports generic value types. The main operation the hash table provides is the *find-or-put* method, and the performed benchmarks resemble how a model checker such as LTSmin would generally interact with it, that is by performing many find-or-put operations.

1.1 Research questions

The current research aims to answer the following research question:

- How does the performance of a Java implementation of Laarman's hash table design [12] scale on multi-

core systems, compared to existing Java hash tables?

With scalable performance we mean the decrease of computational time when the number of CPU cores increases. The following hash tables are considered:

1. nl.utwente.hashtable.FastSet (The new hash table)
2. java.util.concurrent.ConcurrentHashMap (Lockless)
3. java.util.Hashtable (Lock-based)
4. org.cliffc.high_scale_lib.NonBlockingHashMap

Additionally, the following sub-questions are answered:

1. Which of the aforementioned hash tables has the shortest sequential runtime?
2. Which means for implementing a scalable concurrent hash table does Java provide?
3. How does the achieved speedup compare to the speedup of a C implementation of the hash table design, as reported in Laarman's paper[12]?

2. RELATED WORK

The paper by Laarman et. al. describes how a scalable concurrent hash table for use in the LTSmin model checker was designed[12]. They show that it outperforms two state of the art SPIN [8] and DiVinE [1]. The paper describes design decisions, but its results are based on a comparison of model checkers (written in C) and are results on scalability limited to 16 cores. Van den Brink developed a Java implementation based on Laarman's work[25], showing scalability on 4 cores. Comparisons were not done and due to its single-purpose design their work was not used in our research. We use alternative techniques and aim for multi-purpose usage through Java generics.

3. BACKGROUND

3.1 CPU Caches

Modern x86 CPU architectures have multiple levels of memory caches to feed the CPU core with data from the main memory faster, which can be utilized to speedup algorithms that perform many memory accesses, such as a hash table [12]. Some of these caches are shared among cores (L2) and some are local to each core (L1), which causes a problem called *cache line sharing*. With *true sharing misses* the cache data is invalidated because another core has written to data that was needed by another core, while *false sharing misses* invalidates the cache data when a write occurs to another part of the cache, which is not directly needed by the other core [18]. These cache misses are costly and degrade performance of an algorithm, and should therefore be minimized through cache-aware programming. Some techniques that minimize the number of cache misses are keeping the memory working set small, and controlling where in the memory the data is stored by aligning the data on the cache lines. Explicit control over the memory access patterns help with analyzing how effectively caches are utilized in a program.

3.2 Hash table

A hash table is a data structure that provides fast operations for storing data and looking up data. These operations are performed with a time-complexity of $O(1)$ [19]. It associates a key with a value, both of which are provided as inputs to an insertion operation. The key can be used to retrieve its associated value from the hash table. The load-factor of a hash table is defined as the number of values it holds divided by its capacity. The low time-complexity of a hash table is made possible by storing the *hash* of the *value*, which can be generated, compared and mapped to a memory location in linear and constant time. In absence of a perfect hash function, with which every value is mapped to a unique hash, collisions will occur and need to be resolved. In order to resolve collisions various strategies exist, commonly used are external probing, double hashing, linear- and quadratic probing.

3.2.1 External probing

External probing resolves hash collisions by inserting lists of values into the slot where two or more collided hashes map to, for example a linked list. This has the advantage being able to implement a hash table with a constant size backing array while supporting an unlimited (although dynamic memory bound) number of elements. It supports load-factors > 1, although this case can result in a time-complexity of $O(n)$, because the hash table starts to behave like a list. It requires dynamic allocation of elements in the list and has the problem of unpredictable memory access patterns, as pointers of the linked list could point to any memory location, which causes many cache misses, thus degrading performance.

3.2.2 Double hashing

Double hashing resolves collisions by generating an additional hash value h_n for the n-th collision, which is added to the original hash value to find a new index into the table. This method provides better distribution of keys than linear (and quadratic) probing.

3.2.3 linear probing

Linear probing resolves hash collisions by using linear search through the memory that succeeds the memory address obtained from the hash, and insert the value in the first available location. *"Walk-in-the-line"* is a form of linear probing, where the probing is performed on a single cache line in order to minimize cache misses. In isolation, linear probing causes clustering because values where hash collisions occurred are inserted in a narrow memory address range. This effect is amplified when a bad hash function is used. The aforementioned hash collision resolving strategies can sometimes be combined in order to profit from the advantages of multiple strategies.

3.3 Concurrency and Speedup

Graph traversal on large graphs can require a large amount of time to complete. In the past component count on integrated circuits was expected to double every two years (also known as Moore's law [14]), implying that the serial performance of CPU's would keep rising. Recent research shows that this trend must eventually flatten out because of fundamental limits of physics [10]. Also Intel announced that they currently expect a two and a half year cycle to match Moore's law [3], indicating a slowed down increase of serial CPU performance. An widely used alternative to increasing serial performance is to use concurrency, for example by utilizing multi-core CPU's. While it is harder to program software to run correctly in a concurrent way, it can drastically decrease the required time to run the program. *Speedup* represents the performance increase when the program is run on multiple cores (parallel) compared to a single core (sequential): $Speedup = t_{sequential}/t_{parallel}$. Ideally the value of the speedup equals the amount of cores the program runs on (Linear speedup), although reaching such a value is not re-

alistic due to parts of the program that require sequential execution.

3.4 C and Java

The C programming language is regarded as a low-level programming language compared to Java. In C the programmer is tasked with, among others, memory management and writing system dependent code, while Java manages memory for you, through garbage collection for heap allocations, and Java provides an abstraction layer between the programmer and the hardware. These features of Java enables writing more maintainable code with great portability. The abstraction layer for Java is formed by its typical target platform, the Java Virtual Machine (JVM). The JVM does not execute native instructions, but interprets Java bytecode, which enables the possibility to perform platform specific optimizations at runtime through the Just-in-time compiler (JIT). In the past, Java was stigmatized as a slow language, however through development in the Java ecosystem (e.g. better JVM, JIT) it presently achieves performance comparable or equal to programs written in C [21]. Still, the hardware abstractions of the higher-level language can make it either difficult or impossible to implement some architecture specific optimizations or memory management schemes a high-performance program requires.

3.5 Compare-and-swap

Compare-and-swap is an operation, supported by modern CPU's, which makes it possible to perform an atomic comparison of a value in memory (A) with another value (B) combined with writing a new value (C) to the memory location of A. This write only occurs if value A equals value B. Algorithm 1 describes the functionality of the operation, but it must be stressed that its usefulness comes from its atomicity.

```
inputs: address A, value B, value C
if (value at A) = B then
    (value at A) ← C
    return true
else
    return false
end
```

Algorithm 1: compare-and-swap operation

Compare-and-swap enables the creation of any wait-free data object [6]. Similar operations are available through Java classes such as AtomicReference, which encapsulate calls to native CAS instructions, but also Java's Unsafe class provides methods such as compareAndSwapLong.

3.6 Unsafe

Java has an API for internal use, that provides low-level functionality that is not available through "regular" methods, defined in *sun.misc.Unsafe*. It is used by many projects that require more control over the platform it is running on, including Grails and Apache Spark. For our purposes, it provides methods for direct memory allocation and management, as well as the *compareAndSwapLong* method. It is proposed to adapt parts of the *Unsafe* API and make it public in the upcoming JDK 9 release [20].

4. METHOD

4.1 Hash table implementation

4.1.1 Overview

The hash table storage is divided over two locations, the *buckets buffer* and a *data array*. The *buckets buffer* tracks information about the data array's contents. Querying and maintaining this buffer facilitates fast data lookups and insertions, by providing information on the contents and state of actual data without requiring expensive lookups and comparisons on the data array's elements. The *find-or-put* operation is the main operation of our hash table. It looks up a given value and returns false the value was already found in the hash table, and returns true if the value was not found. In the latter case, the value is put into the hash table. The find-or-put operation is described in Algorithm 2.

Figure 1. Bitlayout of a bucket

4.1.2 Buckets

A *bucket* is an element in the *buckets buffer*. It is a 64-bits long integer comprised of the bit-layout in figure 1.

Occupied flag is 1 if the *bucket* has been written to, 0 if otherwise. *Writing flag* is 1 if the *bucket* is taken but the value is not written to yet, 0 if otherwise. *Reserved* bits: for future use or longer hash prefixes. *Hash prefix*: the 32 highest bits of the hash value.

The *buckets buffer* is allocated through the allocateMemory(int bytes) method of the Unsafe's API, with size being a power of two. Subsequently *buckets* must be accessed with the base memory location and an offset into the buffer, for example with Unsafe.getLong() and Unsafe.compareAndSwapLong(). Using compareAndSwapLong() an attempt is done to atomically set the occupied flag, writing flag and hash prefix, while expecting an empty value. Before calling compareAndSwapLong() for a *bucket*, first the contents of the memory are checked for writing and occupied flags, to prevent wasting cycles in the expensive atomic operation. As long as the writing flag is set, other threads that attempt to use this *bucket* will wait for the write operation to complete, unless the hash prefixes don't match. When a *bucket* is obtained, the value is written to the *data array* and the writing flag is removed from the *bucket*.

4.1.3 Hash function

Initially the *original hash* of a to be inserted *value* is calculated using Java's Object.hashCode(), which returns a 32-bit integer hash value, dependent on *value*'s type's implementation of this method. This *original hash* value is inserted in the hash field of a bucket. While probing for an available *bucket*, the hash may have to be recalculated for better distribution. Specifically this occurs after the algorithm reached the end of the cache line, after which rehash(originalHash, n) is called, where n is the number of times the value has already been rehashed, yielding a new hash that is used to determine from where to continue linear probing. Because a hash value can exceed the size of the *buckets buffer*, the index into the buffer is determined by calculating $index = h\%|buffer|$. In Java, the % operator is the *remainder* operator, where the result always has the sign of the dividend (possibly negative). Although the absolute value can be used, using the % operator already has a large performance hit compared to a bitwise operation. Therefore an alternative $index = h\&(|buffer| - 1)$, where $|buffer|$ is a power of two, is used for calculating an index from a hash value.

```
inputs: Vector v, Vector[] values,
long[] buckets, int size
h_0 ← v.hashCode()
a ← 1
while true do
    start ← rehash(h_0, a)&(size − 1)
    for i ← start to start+8 do
        b ← i&(size − 1)
        v ← buckets[b]
        new ←<1, 1, h_0 & MASK_HASH>
        if v not occupied and CAS(b, EMPTY, new)
         then
            values[b] ← v
            remove writing flag from buckets[b]
            return false
        else
            if hash prefixes of h_0 and v match then
                Wait for write to complete...
                If values[b] = v return true
            end
        end
    end
    a ← a + 1
end
```

Algorithm 2: find-or-put algorithm

4.2 ByteBuffer and Atomic classes

The *ByteBuffer* and *DirectByteBuffer* classes were developed for high throughput of mostly file based data. While they provides a nice interface to retrieve and store the data, tests found it to have lower throughput than off-heap memory managed with Unsafe, similar to arrays of primitives[22, 11, 23]. More importantly, it lacks features needed for our purposes: the ability to store more than 2GB of data due to it using 32-bit signed indices, and it does not allow atomic operations, such as compare-and-swap. AtomicLongArray does support these two features, however from the source[17] can be seen that it uses a regular heap allocated long[] internally, thus will likely perform similarly to regular primitive arrays in Java[22].

4.3 Benchmarking

4.3.1 Metrics

To analyze the speedup that can be reached by using different hash table implementations, the throughput of the hash table will be measured. The throughput is defined here as the number of vectors inserted into the hash table, divided by the time required to insert these vectors: $Throughput = \frac{N}{t}[\frac{V}{s}]$. It was chosen to measure the throughput instead of applying the hash table in a graph traversal algorithm, such as breadth-first search, in order to keep the benchmark simple and concrete, and so that it is not influenced by the design difficulties that come with a concurrent graph traversal algorithm, such as load-balancing.

We define a run as an execution step of the benchmark for a given set of parameters: statespace size (also denoted as $|statespace|$), load-factor (lf) and thread-count. The speedup of a given run is defined as the inverse of the time required to complete this run divided by the time required to complete an identical, but sequential run (i.e. thread-count= 0): $Speedup_{run} = 1/\frac{t_{run}}{t_{sequential}} = \frac{t_{sequential}}{t_{run}}$, where *run* is any parallel run. Speedup is a good measurement of the scalability of an algorithm, which is why this is regarded as the most important metric for comparing the concurrent hash tables.

4.3.2 Benchmark design

Because the metric to be measured is the throughput, the benchmark consists of a large number of *find-or-put* operations (which is often the main interaction between a graph search algorithm and a hash table). The vectors that are inserted are generated on the heap, before the threads start executing. The amount of generated vectors is referred to as statespace size ($|statespace|$). The generated vectors are unique Java Strings consisting of 2 to 9 alphanumeric characters. All vectors are distributed over n worker-threads, giving each worker-thread $\frac{|statespace|}{n}$ unique vectors to insert. The amount of times the find-or-put operation is applied to each unique vector can be tweaked in order to change the ratio of insertions to reads (i.e. the ratio of find-or-put applications that return true respectively false). In our benchmarks this ratio is set to 1, so that for $|statespace| = 2^{25}$ the number of performed operations is 2^{26}. Using Barriers, the worker-threads and the benchmark timer are started simultaneously.

The following hash tables are compared using the same benchmark:

1. nl.utwente.hashtable.FastSet
2. java.util.Hashtable (Lock-based)
3. java.util.concurrent.ConcurrentHashMap (Lockless)
4. org.cliffc.high_scale_lib.NonBlockingHashMap (by Dr. Cliff Click)

The hash tables are constructed with a given load-factor in mind, by passing an initial capacity of $\frac{1}{load-factor} * |statespace|$.

4.3.3 Warming up the JIT compiler

In order to minimize the effects the Just-In-Time (JIT) compiler of the JVM has on the execution times of the benchmarks, every benchmark will be preceded by "warm-up" runs. These runs consist of the same code that will be executed during the real benchmark, and allows for the JIT compiler to analyze and optimize parts of this code without affecting the execution time of the "real" runs. The results of the warm-up runs are disregarded for determining the speedup of the hash tables, however they are critically analyzed to determine how many warm-up runs are required for the execution time of a benchmark to stabilize, within a small margin (determined by manual inspection). The final results were obtained by repeating every run 10 times.

4.3.4 System

The benchmarks have been performed on a Dell R815 with 4 Opteron 6376 CPU's, which provides a total of 64 available cores, with 512GB of RAM, running Ubuntu 14.04.4 LTS and Java HotSpot™ 64-Bit Server VM (build 24.80-b11, mixed mode). Means for gaining exclusive access to the resources were not enabled on this server (e.g. a job-scheduler), however through process logs and manual inspection it was ensured that no other tasks were executed throughout the benchmarks. Multiple benchmarks are performed using subsequently 1, 2, 4, 8, 16, 32 and 64 threads, mapping each thread to a physical core.

5. RESULTS

Table 1 and figure 2 show the wall-clock time (runtime) to perform the benchmarks for multiple thread counts, averaged over load-factors (lf) of $\frac{1}{2}$ and $\frac{1}{4}$ using

nl.utwente.hashtable.FastSet (FastSet), java.util.concurrent.ConcurrentHashMap (CHM), java.util.Hashtable (Hashtable) and org.cliffc.high_scale_lib.NonBlockingHashMap (NBHM). Table 2 and figure 3 show the calculated speedup of the benchmarks. The appendices include the results on speedup and runtimes for different load-factors in tables 3, 4, 5 and 6.

Table 1. Runtime, avg. over $lf = \{\frac{1}{2}, \frac{1}{4}\}$

Threads	FastSet	CHM	Hashtable	NBHM
1	13.614	21.029	4.974	41.794
2	8.972	21.518	18.681	45.945
4	4.780	13.062	30.393	33.285
8	2.542	8.146	28.268	26.050
16	1.467	8.120	28.842	21.941
32	1.078	8.599	24.544	21.226
64	1.108	9.025	23.265	43.481

Table 2. Speedup, avg. over $lf = \{\frac{1}{2}, \frac{1}{4}\}$

Threads	FastSet	CHM	Hashtable	NBHM
1	0.925	0.930	0.842	0.962
2	1.407	0.918	0.219	0.875
4	2.633	1.498	0.133	1.208
8	4.939	2.390	0.143	1.541
16	8.574	2.395	0.139	1.830
32	11.859	2.261	0.164	1.922
64	11.668	2.160	0.174	1.221

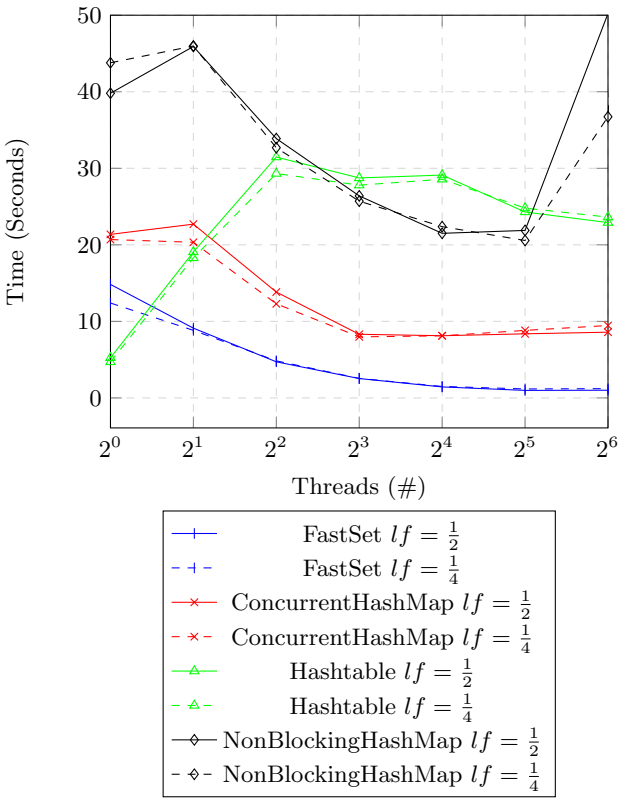

Figure 2. Runtimes for |statespace|=2^{25}, avg. over $lf = \{\frac{1}{2}, \frac{1}{4}\}$

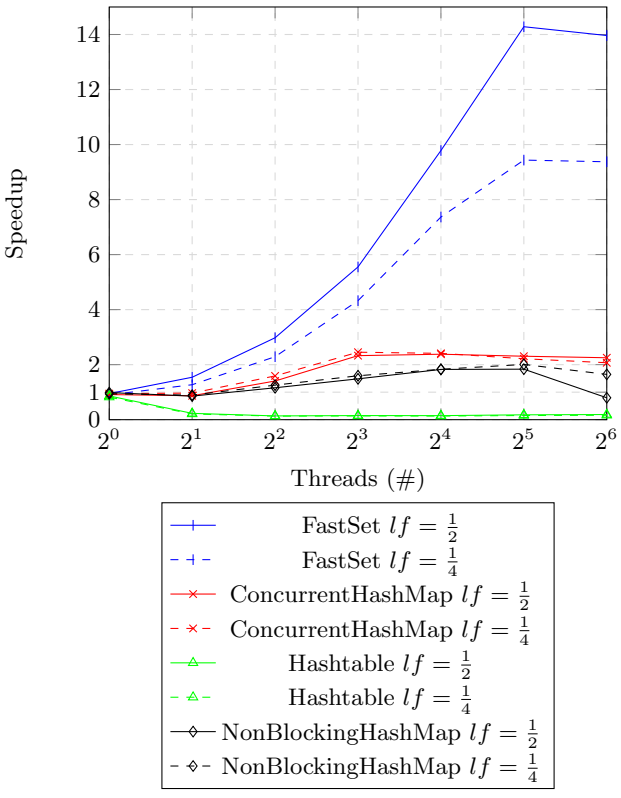

Figure 3. Speedup for |statespace|=2^{25}, avg. over $lf = \{\frac{1}{2}, \frac{1}{4}\}$

6. DISCUSSION

6.1 Java.util.Hashtable

The results in figure 2 match the expectations regarding Java's *Hashtable*. It shows that *Hashtable* has a significantly shorter runtime than any other hash table implementation in the sequential case, but slows down to about a tenth of the throughput as soon as concurrency is introduced. This serves as an argument against lock-based algorithms when scaling is required. When designing for performance, *Hashtable* can be considered when thread-safety is a requirement and the larger part of the interaction with the hash table is expected to be single-threaded.

6.2 ConcurrentHashMap

With ConcurrentHashmap a maximal speedup of 2.750 is achieved (table 2), although the sequential runtime is slower than *Hashtable*. It doesn't suffer from multithreading like Hashtable does, therefore ConcurrentHashmap should be considered when running software that requires mostly parallel access to the hash table. Figure 2 shows an increased runtime on 2 threads compared to 1 thread, which is likely caused by a large runtime deviation in one of the runs with 2 threads. This is apparent in figure 2 but not in figure 2 due to the fact that in the latter figure outliers were filtered out.

6.3 FastSet

FastSet scales better than the other tested implementations, and also has a good sequential runtime of 13 seconds. With a load-factor of $\frac{1}{2}$ it achieves a speedup of 9.435, and with a load-factor of $\frac{1}{4}$ it achieves a speedup of 14.283, both peaking at 32 cores. Running on 64 cores did not yield additional speedup.

6.3.1 Comparison to Laarman's version

For reference, in the paper of Laarman et al., for which benchmarks with up to 16 cores were performed, speedup values between 12 and 14 were found[12, figure 2 and 3], while for our implementation speedup between 7.366 and 9.783 was found on 16 cores. Their benchmarks run LTSmin, while our benchmark tests the throughput of the hash table directly, which may make our benchmarks more memory limited. Therefore this comparison is only meant as a vague reference, and comparing runtimes does not provide any meaningful information.

6.4 NonBlockingHashMap

NonBlockingHashMap scales well up to 16 cores, but when using it on more than 32 cores the runtime saw a large increase and several more serious problems arose, as further discussed in section 6.5.1.

6.5 Limitations

6.5.1 NonBlockingHashMap

In our tests *NonBlockingHashMap* did not only achieve not the advertised linear scaling up to 768 CPUs[5] but it also failed to prove its correctness. In our benchmarks some seemingly arbitrary pattern of runs did not pass the verification of checking that the number of vectors in the hash table equals the size of the statespace. Also, when using a statespace larger than 2^{25} (meaning a hash table capacity of 2^{26} and 2^{27} elements to account for load factors of $\frac{1}{2}$ and $\frac{1}{4}$) the benchmark was terminated after an hour, for it had not been able to perform a single run. The hash table also claims (too) much memory in certain cases, and throws an OutOfMemoryError. This was also reported to the author by others, however these issues were not resolved [24, 9]. In order to incorporate *NonBlockingHashMap* into the research, the results of a statespace with a size of 2^{25} were compared as opposed to larger statespaces.

6.5.2 Limited operations

A reason that may provide large perceived benefit for FastSet is the fact that it does not support all operations a typical hash table provides, such as deletions or dynamic resizing. It can be argued that its interface is more like a set instead of a hash table. Noted about Set implementations in Java must be that they are backed by a Map implementation, where the key is the to be inserted value, and the value is a static, reused object.[13].

7. CONCLUSION

Our research found that the new hash table achieves better performance scaling than existing Java hash table implementations. It scales up to 32 cores, reaching speedup values between 9.4 and 14.3, depending on the load-factor. *ConcurrentHashMap* scaled up to 8 cores with a maximal speedup of 2.4, *NonBlockingHashMap* up to 32 cores with a speedup of 1.9 and Hashtable suffers from slowdown when used concurrently. Issues regarding NonBlockingHashMap's results were discussed in section 6.5.1.

Sequential runtime of the new hash table is shorter the other scalable hash tables, but longer than the non-scalable *Hashtable*.

Results in the paper by Laarman et al. show higher speedup of between 12 to 14 on 16 cores, although comparability of the results is debatable due to our benchmark possibly being more memory bound than their LTSmin based benchmark.

We found that Java provides sufficient means for implementing high-performance concurrent algorithms. Multiple possibilities were explored in this research, and for achieving control over cache-alignment and compare-and-swap operations the methods in the *Unsafe* API were a necessity, and an essential alternative to classes that encapsulated these concepts, such as *ByteBuffer* and *AtomicReferenceArray*.

This research provides an argument in favour of the possibility to create scalable algorithms in Java. It is not rare for a programming language related argument to spark explosive debates, so to conclude on an *Unsafe* note: future software projects should consider Java over lower-level languages, as it allows for great scalability without sacrificing portability and maintainability.

8. FUTURE WORK

8.1 Benchmarking bias

The benchmark and FastSet were both developed during the same research, focussing mostly on FastSet. Though serious thought was put into developing benchmarks that are fair for all tested hash tables, it is possible for the benchmark to unintendedly have become biased towards certain implementations. Also, the performance of FastSet has not been determined for all possible combinations of parameters such as load-factor, cache-line size or element type. Future work could incorporate the hash table in other projects to see how well it performs in different contexts.

8.2 Verification

FastSet has not been mathematically or programmatically verified. Before the hash table can be used in any safety-critical application, like for example a model checker, first its correct functioning needs to be verified. A possibility for programmatic verification that became available by writing the hash table in Java is to apply the existing tool VerCors, a tool for verification of concurrent data structures [2]. Future work can be done with this tool, or by devising a mathematical proof, in order to either proof or disproof correctness.

9. REFERENCES

[1] J. Barnat, L. Brim, and P. Rockai. Scalable Multi-core LTL Model-checking. In *SPIN*, volume 7, pages 187–203. Springer, 2007.

[2] S. Blom and M. Huisman. The VerCors Tool for verification of concurrent programs. In *International Symposium on Formal Methods*, pages 127–131. Springer, 2014.

[3] D. Clark. Intel Rechisels the Tablet on Moore's Law. http://blogs.wsj.com/digits/2015/07/16/intel-rechisels-the-tablet-on-moores-law, 2015. Accessed: 2016-05-11.

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

[5] C. Click. NonBlockingHashMap GitHub page. https://github.com/boundary/high-scale-lib/tree/3654434eda00b68d37d22dcd70e4f65db9432d06, 2016. Accessed: 2016-06-10.

[6] D. Dice, D. Hendler, and I. Mirsky. Software-based contention management for efficient compare-and-swap operations. *Concurrency and Computation: Practice and Experience*, 26(14):2386–2404, 2014.

[7] P. Guo. Python is Now the Most Popular Introductory Teaching Language at Top U.S.

Universities. http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext, 2014. Accessed: 2016-06-15.

[8] G. J. Holzmann and D. Bosnacki. The Design of a Multicore Extension of the SPIN Model Checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, Oct 2007.

[9] C. interest mailing list. Unexpected memory usage in NonBlockingHashHashMap. http://jsr166-concurrency.10961.n7.nabble.com/Unexpected-memory-usage-in-NonBlockingHashHashMap-td9913.html, 2013. Accessed: 2016-06-14.

[10] S. Kumar. Fundamental Limits to Moore's Law. *Fundamental Limits to Moore's Law. Stanford University*, 9, 2012.

[11] A. Kutuzov. Unsafe. http://senior-java-developer.com/unsafe/, 2014. Compares Byte Buffer, mmap and Unsafe, Accessed: 2016-06-17.

[12] 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*, pages 247–256. FMCAD Inc, 2010.

[13] D. Lea. How HashSet Internally Works in Java. https://java67.blogspot.com/2014/01/how-hashset-is-implemented-or-works-internally-java.html, 2014. Accessed: 2016-06-10.

[14] G. Moore. Cramming More Components Onto Integrated Circuits. *Electronics volume 38 no. 8*, 1965.

[15] Oracle. java.util.concurrent.ConcurrentHashMap documentation. https://github.com/boundary/high-scale-lib/issues/9, 2016. Accessed: 2016-06-15.

[16] Oracle. java.util.HashTable documentation. https://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html, 2016. Accessed: 2016-06-15.

[17] J. Paul. java.util.concurrent.atomic.AtomicLongArray implementation. http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/atomic/AtomicLongArray.java. Accessed: 2016-06-19.

[18] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems*, pages 335–348. ACM, 2010.

[19] A. Prakash and A. S. Gautam. Fast and Scalable IP Address Lookup with Time Complexity of LogmLogm (n). *Journal of Advances in Information Technology*, 5(2):58–64, 2014.

[20] M. Reinhold. JEP 260: Encapsulate Most Internal APIs. http://openjdk.java.net/jeps/260, 2016. Accessed: 2016-06-15.

[21] A. Shafi, B. Carpenter, M. Baker, and A. Hussain. A comparative study of Java and C performance in two large-scale parallel applications. *Concurrency and Computation: Practice and Experience*, 21(15):1882–1906, 2009.

[22] A. Sharma. Which memory is faster Heap or ByteBuffer or Direct ? https://www.javacodegeeks.com/2013/08/which-memory-is-faster-heap-or-bytebuffer-or-direct.html, 2013. Accessed: 2016-05-16.

[23] M. Thompson. Native C/C++ Like Performance For Java Object Serialisation. https://mechanical-sympathy.blogspot.nl/2012/07/native-cc-like-performance-for-java.html, 2012. Accessed: 2016-06-19.

[24] G. user: xfeep. Issue: Always leads to OutOfMemoryError when we use large scalar random keys. https://github.com/boundary/high-scale-lib/issues/9, 2015. Accessed: 2016-06-14.

[25] B. van den Brink. Providing an efficient lockless hash table for multi-core reachability in java. *21st Twente Student Conference on IT*, 21(June 23), 2014.

APPENDIX

Table 3. Runtime in seconds, $load - factor = \frac{1}{2}$

Threads	FastSet	CHM	Hashtable	NBHM
1	14.829	21.369	5.242	39.805
2	9.116	22.703	19.068	45.911
4	4.719	13.830	31.467	33.883
8	2.530	8.315	28.739	26.396
16	1.436	8.126	29.115	21.505
32	0.984	8.375	24.294	21.884
64	1.005	8.589	22.906	50.232

Table 4. Runtime in seconds, $load - factor = \frac{1}{4}$

Threads	FastSet	CHM	Hashtable	NBHM
1	12.399	20.688	4.706	43.784
2	8.829	20.333	18.295	45.980
4	4.841	12.293	29.318	32.687
8	2.554	7.977	27.797	25.705
16	1.499	8.114	28.569	22.377
32	1.172	8.823	24.794	20.569
64	1.211	9.461	23.624	36.730

Table 5. Speedup values, $load - factor = \frac{1}{2}$

Threads	FastSet	CHM	Hashtable	NBHM
1	0.902	0.947	0.821	0.939
2	1.271	0.966	0.208	0.894
4	2.289	1.591	0.129	1.256
8	4.326	2.453	0.136	1.596
16	7.366	2.411	0.132	1.833
32	9.435	2.216	0.152	2.008
64	9.374	2.070	0.161	1.646

Table 6. Speedup values, $load - factor = \frac{1}{4}$

Threads	FastSet	CHM	Hashtable	NBHM
1	0.948	0.913	0.864	0.985
2	1.542	0.870	0.230	0.857
4	2.978	1.406	0.137	1.160
8	5.552	2.327	0.150	1.485
16	9.783	2.379	0.147	1.827
32	14.283	2.307	0.177	1.836
64	13.963	2.251	0.188	0.796