Concurrent High-performance Persistent Hash Table In Java

by

Xianda Sun

A thesis

presented to the University of Waterloo

in fulfillment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Current trading systems must handle both high volumes of trading and large amounts of trading data. One crucial module in high-performance trading is fast storage and retrieval of large volumes of data simultaneously accessed by multiple computer traders. To speed up access, a high-performance in-memory software-cache stores the dynamic working-set of trades during a trading day. To utilize memory effeciently, it is beneficial to provide a single shared cache for multiple trading applications. Much of the cache access is read-only, as information is gathered before a transaction to determine its value. Hence, extremely fast lookup is essential to support quick information gathering for assessment. This thesis presents a software-cache, called MapHash, that is a high-performance hash-table for use in Java.

Acknowledgements

I would like to express deepest gratitude to my advisors Dr. Martin Karsten and Dr. Peter Buhr for their full support, expertise guidance and understanding throughout my research.

I would also like to thank Mr. Thomas Lo and Mr. Michael Salvagna from FINANCIALOGIX Group for their advice and collaboration on this research thesis.

I would also like to thank my fellow graduate students from the David R. Cheriton School of Computer Science for the help and inspirations.

Finally, I would like to thank my parents for their unconditional support during the process of completing the degree.

Dedication

This is dedicated to my parents.

Table of Contents

List of Tables **ix**

List of Figures **x**

1 Introduction **1**

1.1 Problem Statement 3

2 Mutual Exclusion **6**

2.1 Concurrent Queue 8

2.2 Queue Experiment 10

3 Design of MapHash **14**

3.1 Hash Table 15

3.2 Collision Resolution 16

3.3 Number of Hash Buckets 17

3.4 Concurrency 23
3.5 Persistence 24
3.6 MappedFile Storage Layout 28
3.7 MapHash Locks 30
3.8 MapHash Get Operation 31
3.9 MapHash Put Operation 32
3.10 MapHash Remove Operation 34
3.11 Allocation 35
3.12 Eviction 35
3.13 Large Dataset Support 36
3.13.1 Multiple-File Buffer Proxy 36
3.13.2 Multiple-File Backing Storage 37

4 Performance of MapHash **39**

4.1 Chronicle Map and Tuning 40
4.2 Java ConcurrentHashMap and Tuning 41
4.3 Experiment 43
4.3.1 Design 43
4.3.2 Setup 44

4.4 Experiment Setup 45
4.5 Different Experiments 47
4.6 Removal: JavaHashMap Experiment 47
4.7 Memory Placement 48
4.8 MapHash Eviction Experiment 52
4.9 MapHash Multiple JVM 54
4.10 Scaling Experiment 56

5 Conclusion **59**

5.1 Future Work 60

References **62**

List of Tables

3.1 Bucket Chain Length Distribution 19

3.2 Bucket Chain Length Distribution With 1 million Buckets 20

List of Figures

2.1 Queue Throughput, AMD 13

2.2 Queue Throughput, Intel 13

3.1 MapHash Performance With Different Bucket Numbers, AMD 22

3.2 MapHash Performance With Different Bucket Numbers, Intel 22

3.3 Design of MapHash Storage 27

4.1 Throughputs of All Maps, AMD 49

4.2 Throughputs of All Maps, Intel 49

4.3 Placement Performance, default policy, AMD 50

4.4 Placement Performance, default policy, Intel 50

4.5 Placement Performance, interleaving, AMD 51

4.6 Placement Performance, interleaving, Intel 51

4.7 Eviction Performance With Different Trace Size, AMD 53

4.8 Eviction Performance With Different Trace Size, Intel 53

4.9 Multiple JVM Performance, AMD 55

4.10 Multiple JVM Performance, Intel 55

4.11 Scaling Performance, AMD 58

4.12 Scaling Performance, Intel 58

Chapter 1

Introduction

An automated trading system is software that creates orders and automatically submits the orders to market centres or exchanges. In today's financial sector, most of the traditional floor-based trading has been replaced by automated trading-systems. Reported by Greenwich Associates, in 2014, electronic trading accounts for 79% of the stock trading volume in the United States [3]. To be competitive in this domain, financial institutions need to develop and use both complex and fast software systems comprised of many inter-operating modules.

Current trading systems must handle both high volumes of trading and large amounts of trading data. One crucial module in high-performance trading is fast storage and retrieval of large volumes of data simultaneously accessed by multiple computer traders, controlled by both human and algorithmic trading. Traditional databases are used to store and retrieve large amounts of data. However, databases are often too slow for many high-

performance trading-systems. To speed up access, the database is often front-ended or replaced with a high-performance in-memory software-cache, which stores the dynamic working-set of trades during a trading day.

Much of the cache access is read-only, as information is gathered before a transaction to determine its value. Hence, extremely fast lookup is essential to support quick information gathering for assessment, which can lead to aborting the transaction, or making a purchase, or transferring a financial vehicle. The main problem in looking up data in any corpus of information is searching. Given a looked-up key, it must be compared to corresponding keys in the corpus to find a match. Once there is a match, associated data can be extracted (or updated). The search speed for a large corpus of information can dominate the time to process a trading transaction.

One of the best in-memory data structures for fast lookup is the *hash table*, which has expected average access time of O(1). Most software and hardware caches are based on some form of hash table. For trading applications, the hash-table size is large: from 1 to 512 gigabytes, containing from 1 to 1,000 million records. Additionally, high concurrent access for both read and write operations is required with only a weak notion of consistency. That is, a read or write operation is atomic, but a read copies a record from the hash table so it may immediately change in the hash table by a write. Hence, there is no notion of multiple-step transactions that preserve consistency across a number of read/write operations, because the entire trading system is beyond the control of any individual trader. As well, the hash table must be *persistent*, meaning it can be stored on disk and reused across trading days, making it available during or after each trading period for analysis. Finally, many financial institutions use the Java programming language so the hash table

must perform well in the Java runtime environment and support simultaneous access by multiple Java Virtual Machines (JVM).

This thesis presents a software-cache, called MapHash, that is a high-performance hash-table for use in Java. MapHash is used as one of the key modules in an automated trading-system called F1, a trading platform developed by FINANCIALOGIX Trading Inc. [6].

1.1 Problem Statement

A traditional trading-platform stores information like user-account data and equity prices in a persistent database, front-ended by a data cache to improve performance. User programs running in different JVMs (different operating-system processes) request data from the data cache. All the data is stored in the form of key-value pairs. For efficiency, it is common for a data cache to only handle fixed-size records (containing key and values), where the record size is the maximum among the different records. If a cache miss occurs, the application asynchronously sends a request to the database to read the missing data. The cache must meet the following requirements:

- performance: minimize the average latency of accessing records and maximize the throughput.
- concurrency: support multiple simultaneous access within a JVM by multiple threads and across multiple JVMs. Read and write operations must be atomic. In the data cache, a read operation copies all fields of a record. It is acceptable if a reader reads

stale data, but the data must be internally consistent, i.e., all the fields read are from a single write.

- workload: the read to write ratio can vary significantly among threads, but in most cases, the number of reads is much greater than writes.

- lookup: searching is based on a primary key, e.g., a 64 or 128-bit random key, like a Universally Unique IDentifier (UUID).

- replacement and expansion: writes with existing keys update values in the cache; writes with new keys increase the size of the cache up to a maximum, after which writes with new keys replace (evict) existing key-value pairs.

There are some key-value store systems which can be used to build a data cache. Redis [18] is an open source, in-memory data structure store, used as database, cache and message broker. It supports a wide variety of data structures. Redis achieves persistence by having the key field reside in memory and the value field swap between memory and disk. However, Redis is a single-thread server [17], it allows concurrent access by the Inter-Process Communications (IPC). This approach introduces a significant overhead compared to the shared-memory approach. Moreover, as a single-thread server, Redis cannot benefit from multiple CPU cores.

MICA [12] is a scalable in-memory key-value store that handles 65.6 to 76.9 million key-value operations per second using a single general-purpose multi-core system. MICA uses a log-structure memory-allocator, and employs a garbage collector to recycle removed records. The data structure MICA uses for indexing is a lossy hash-table, in this hash

table, new records remove old records instead of resolving collisions. Although MICA also implements an internal hash table, it is designed to run in a network environment. This fact means that MICA has to solve the network stack overhead bottleneck, which is not applicable to this data cache scenario. MICA does not support persistency, so it cannot be used to build the cache.

In sum, the specific requirements and relaxations make it worthwhile to design a hash table dedicated to work in this case.

In addition to the design of the hash table, a testing program (harness) has been developed to evaluate different performance metrics of the data structure. The harness runs benchmarks and measures the throughput of the system. It also gathers statistics about the hash table that are beneficial to the design and parameter tuning.

The thesis is organized as follows. Chapter 2 compares two approaches to multiple thread coordination. Chapter 3 describes the design of the data cache, MapHash. Chapter 4 displays the experimental setup and the results. Chapter 5 introduces related works. Chapter 6 is the conclusion of the thesis.

Chapter 2

Mutual Exclusion

Data-cache operations (reads, writes, removes) manipulate complex internal data structures. For example, in a hash table, collisions occur when multiple keys hash to the same hash bucket. There are multiple techniques to deal with collisions, such has chaining duplicate key mappings or searching for an empty hash bucket. When a data structure is accessed concurrently, these complex operations are at risk of corruption because the operations are not atomic, e.g., two threads attempt to simultaneously add a new record to the same hash chain, which corrupts the link fields. Hence, mutual exclusion is required to provide atomicity of operations during simultaneous read/write and write/write operations. The data-structure operations require some form of low-level locking, and there are multiple techniques to achieve mutual exclusion : software, hardware, or a combination called lock-free.

Fine-grained lock-based data-structures for concurrent access have been examined for 50

years and are well investigated. These data structures use classical mutex locks, involving a duration of lock ownership during which no other operations can occur on the data structure. During the past decade, new research has developed a class of data structures that use a combination of atomic instructions and software to achieve mutual exclusion of operations without an ownership (blocking) phase, called *lock-free data-structures*. Lock-free approaches leverage the most complex atomic instructions to perform each data-structure operation as a single atomic action. Because the atomic instruction is weaker than the data-structure operation, additional software is required to complete each operation, often adding significant complexity to the implementation. As for lock-based mutual exclusion, lock-free has two forms: not providing or providing a guarantee of eventual progress, where the former has potential starvation. The terms "lock-free" and "wait-free" denote the two forms. While most lock-free data-structures are relatively simple, such as a stack, list, or queue, newer literature and open-source software covers more complex data structures, such as hash tables or other forms of dictionaries.

Because of the high-performance nature of this project, it is crucial to have the best possible concurrency. Hence, it is necessary to evaluate different mutual exclusion techniques for protecting data-structure operations to find one with the highest performance. As well, there is controversy about the performance claims of lock-free approaches. To discover whether lock-free data structures have an advantage over lock-based ones, a simple experiment is performed using a concurrent queue, where the mutual exclusion for queue operations is implemented with a number of different locking techniques. A queue is used because the operations are simpler and constant versus a hash table, and hence, the data-structure implementation does not dominate the operation, so differences in locking

techniques should stand out more.

Two classes of locks are examined: on and off heap. An on-heap lock is allocated from the Java runtime dynamic-allocation area (heap) and can be garbage collected. An off-heap lock is allocated in another programming language's runtime (often C or C++); its storage is outside the Java runtime and hence is not garbage collected. Furthermore, all inter-language operations must go through an interface to bi-directionally convert data into the appropriate form for each programming language. This project requires off-heap locks because of the persistent requirement and access by multiple JVMs. Nevertheless, testing is performed using both kinds of locks to determine if there is any real difference between on and off heap locking, including the cost of inter-language operations.

2.1 Concurrent Queue

Building a concurrent queue requires both the queue data-structure and appropriate locking. The following locks are tested:

- Java Reentrant Lock: is an on-heap blocking mutual-exclusion lock with the same basic behaviour and semantics as the implicit monitor lock accessed using synchronized methods and statements. A reentrant lock has additional state to remember the lock owner, and the lock owner may reacquire the lock multiple times, with a corresponding number of releases.

- Java Spin Lock: is an on-heap spinning mutual-exclusion lock built from Java atomic-integer using the test-and-set instruction.

Listing 2.1: JNI Spinlock with Exponential Back-off

```
JNIEXPORT void JNICALL Java_LockAccess_lock( JNIEnv *jenv, jclass obj ) {
    enum { SPIN_START = 4, SPIN_END = 64 * 1024, };
    unsigned int spin = SPIN_START;
    for ( unsigned int i = 1;; i += 1 ) {
      if ( *lock == 0 && __atomic_test_and_set( lock, __ATOMIC_SEQ_CST ) == 0 ) break;
        for ( unsigned int s = 0; s < spin; s += 1 ) Pause(); // exponential spin
        spin += spin;                                        // boost spin
        if ( spin > SPIN_END ) spin = SPIN_START;            // prevent overflow
    } // for

} // Java_LockAccess_lock

JNIEXPORT void JNICALL Java_LockAccess_unlockOne( JNIEnv *jenv, jclass obj ) {
    __atomic_clear( &lock1, __ATOMIC_RELEASE );
} // Java_LockAccess_unlock
```

- Java Concurrent Queue: is an on-heap lock-free queue using a wait-free algorithm based on [15].

- JNI Spin Lock: is an off-heap spinning mutual-exclusion lock built from atomic instructions using the g++ atomic intrinsics. The spin lock uses exponential backoff to handle contention [2] (see Listing 2.1). The off-heap lock is linked into the Java and accessed through the Java Native Interface (JNI).

2.2 Queue Experiment

The queue experiment is a maximal contention experiment, where threads perform repeated queue operations with very little code between operations. Hence, locking behaviour dominates the experiment.

A queue experiment has the Java main program create the shared queue and insert N nodes (1-32) initialized to Integer.MAX_VALUE. The main program then creates N worker threads, each assigned to a different core (core affinity), where each worker accesses the shared queue and the queue has different locking. The four locking mechanisms are described in Section 2.1. The main program then sets a shared start variable and sleeps for T seconds. After the sleep, the main program sets a shared stop variable and joins with the threads.

Listing 2.2 shows the run member of a worker thread. A worker spins on the shared variable start until all threads are created. Once started, a worker spins removing and adding nodes from the shared queue. Because each thread only accesses one node from the queue at a time and there is one node per thread, all remove operations are successful (no delay). The data value in a queue node is changed after removal and changed again on add. A worker checks during the changes to detect if the queue mutual exclusion is violated by another worker. Each thread counts the number of remove/add operations until the shared variable stop is set by the main program, at which point the thread ends. The main program accumulates the operation counts in each worker thread via a getter member in the worker thread.

Listing 2.2: Java Worker Thread Accessing Shared Queue

```
public void run() {
    Data data = new Data();

    while ( ! start  );                          // wait until harness starts experiment

    while ( ! stop ) {
        data = queue.pop();
        if ( data.i != Integer.MAX_VALUE ) { System.out.println( "interference0" );}
        data.i = tid;                            // set check
        if ( data.i != tid ) { System.out.println( "interference1" );}
        data.i = Integer.MAX_VALUE; // reset check
        queue.push( data );
        operations += 1;
    } // while
} // run
```

Figure 2.1 and Figure 2.2 shows the results of the queue experiments for the four locking approaches on two different machine architectures, Intel and AMD. The experiment is run for 60 seconds. Higher values in the graph are better indicating the locking approached allowed more remove/add operations during the timed experiment.

The AMD graph shows the JNI off-heap spin lock performs well up to 8 cores, after which only the Java reentrant lock is better. The Java on-heap spinlock is next in performance and the slowest performance is by the Java concurrent lock-free. The performance of the Java blocking lock after 8 cores is unusual because it is linear as more contention is added via additional threads. This strange behaviour occurs because the Java blocking lock is unfair, i.e., an arriving thread can steal the lock from threads that are waiting for an arbitrary amount of time. This behaviour eliminates significant contention and results in running one thread sequentially for long periods. (Note, large scale unfairness of this

kind is unacceptable for financial trading.) The blocking lock has a fair setting, and when toggled, the curve for the blocking lock behaves like the other curves, and lower than all the other curves (not shown). The poor performance of the lock-free queue may result from the maximal contention experiment, where the lock-free queue does not perform well.

The Intel graph again shows the JNI off-heap spinlock performs well across all cores. The Java reentrant lock performance is still unusual showing linear performance. The Java spinlock shows good performance on chip (8 cores) and then drops to the level of the Java concurrent lock-free queue.

Except for the anomaly of the Java reentrant lock, the JNI off-heap spinlock gives the best performance for maximal contention. All the locks give similar performance for minimal (zero) contention (see one thread). These result show that off-heap locking is not an impediment for high-performance and that attempting to implement lock-free data structures off-heap is unlikely to provide any advantages especially given the additional complexity of managing storage, requiring epoch-based reclamation [7], read-copy-update (RCU) [13] or hazard pointers [14].

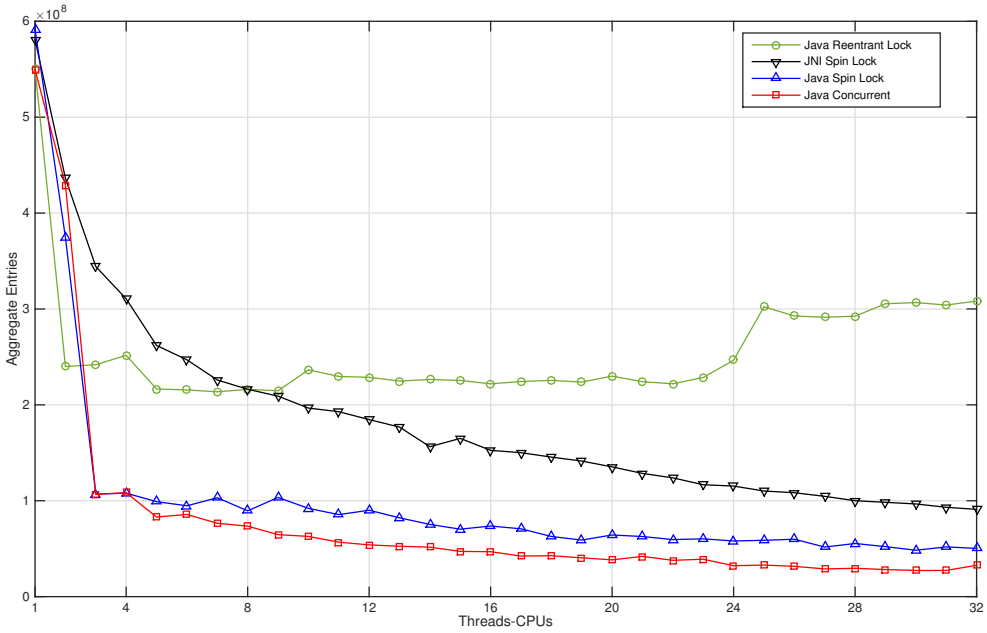

Figure 2.1: Queue Throughput, AMD

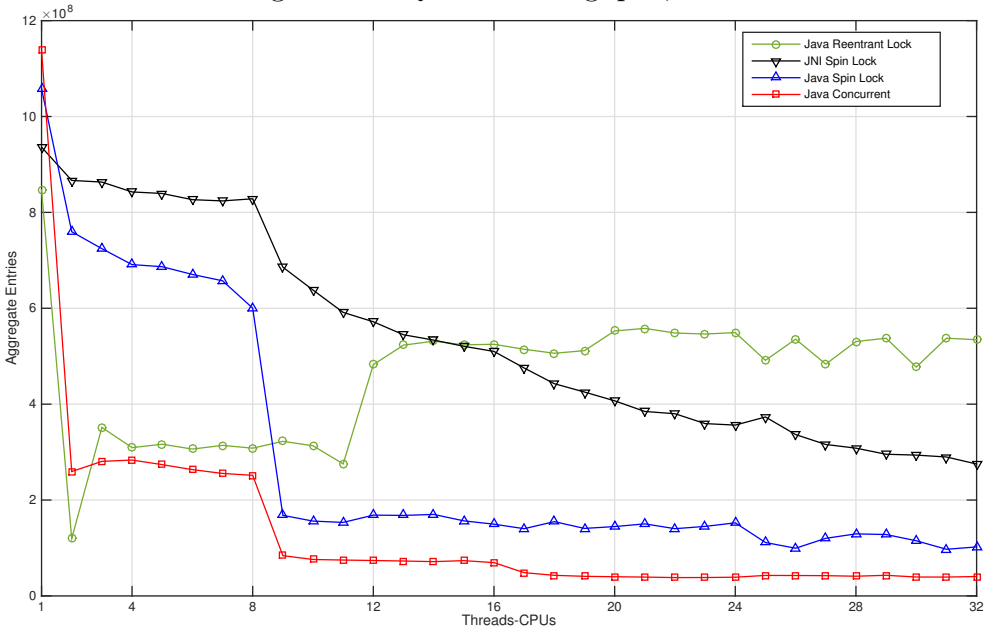

Figure 2.2: Queue Throughput, Intel

Chapter 3

Design of MapHash

The data-cache system provided by FINANCIALOGIX is a shared cache, i.e., the system stores one copy of data instead of multiple copies and handles access requests from multiple JVMs. As a result, the data-cache needs to be a persistent concurrent shared-memory dictionary. The resulting new data-cache, MapHash, is a high-performance off-heap concurrent hash table implemented in Java to facilitate the F1 automated trading-system for commercial business trading.

The requirements for the hash table had the following relaxations:

i) There is only one writer inserting or removing stored data;

ii) The hash table only has to deal with fixed-length records;

iii) The hash table must be able to grow dynamically with respect to records but the number of hash buckets may be fixed at creation.

iv) There are more read operations than write operations.

The requirements for the hash table are:

i) The data cache must be off-heap, because multiple JVMs need to share the same data in different memory spaces and it must be persistent, i.e., have direct backing store in the file system;

ii) Access to the off-heap storage in Java is only provided by Java's ByteBuffer, which has a maximum buffer size of 2GB, but the data cache must handle sizes exceeding 400GB;

iii) While the data cache can grow in the number of records, there is a maximum size (sum of hash buckets and records), at which point, adds of new key-value pairs must evict existing pairs;

iv) While a single writer may simplify concurrency, other concurrency issues are still important, e.g., data must be internally consistent so a read is from a single write.

v) High-performance is defined as low latency and high throughput.

One of the best data structures to implement a dictionary data-structure is a hash table. The MapHash hash-table is an array of buckets with a chain of key-value pairs to handle key collisions. When looking up a key, MapHash first computes a bucket from the key value, and then searches the linked-list chain for an exact match.

3.1 Hash Table

A hash table has three components: hash function, number of hash buckets, and collision resolution. The quality of each component dictates the performance of key lookup and removal, and possibly the complexity of dynamically increasing or decreasing the number

of hash buckets. Given a key range R, a set of keys K, and a set of hash buckets B, it is often the case that $R \gg K > B$.

The quality of the hash function is related to its ability to compress the large key range R down to a smaller range B with few duplicate mappings of different key values. In general, the hash function is key specific and cannot be generalized. Hence, a specialized hash function is often created for each specific application for best results. In this work, the key range R is a 64-bit value, and a simple modulo hash-function is used, where the modulus is the number of hash buckets.

The number of collisions generated by the hash function is related to the actual key values and the number of hash buckets. By spreading out the keys in the key range and increasing the number of hash buckets, a hash function can generate fewer collisions. Often, only the number of hash buckets is available for adjustment as the keys are preassigned.

3.2 Collision Resolution

There are few perfect hash functions [4], so dealing with collisions is an important aspect of a hash table. There are two types of collision resolution: open addressing and chaining.

Open addressing approaches include linear probing [5], HopScotch [8] and Cuckoo [16]. These approaches resolve collisions by putting the colliding key into an existing empty hash bucket rather than use more storage elsewhere. There is a cost to find the unused storage, and filling a hash bucket with a key that does not hash to that bucket can result in more collisions when a legitimate hash value hashes to the bucket. Performance of open

addressing largely depends on the insertion pattern of the hash keys, and as the hash table fills, there is a cascade of collisions as buckets fill with collision keys. When storage is scarce, open addressing is often the best option.

Chaining uses additional storage, often in the records stored in the hash table, to link the records from a hash bucket for the keys that collide. The extra storage often reduces search time and manipulation of entries in the hash buckets, while ensuring hash buckets are only used by legitimate keys, eliminating collision cascading. When storage is plentiful, chaining is often the best option.

Since MapHash is required to perform well on large hash tables with plentiful storage, the hash table is optimized for speed, and hence, the approach of bucket chaining is used for collision resolution.

3.3 Number of Hash Buckets

Most hash tables have initialization parameters estimating the number of unique keys to be inserted, from which the hash table estimates the number of hash buckets to help reduce the number of collision. This parameter can have a significant influence on performance. When memory is optimized, the number of hash buckets is significantly less than the estimated number of keys. When speed is optimized, the number of hash buckets is similar to the number of keys in an attempt to reduce collisions. MapHash seeks a balance between performance and memory usage.

Focusing on a single hash-bucket chain, the longer a bucket chain, the slower it takes to

access items on it on average. Assume accessing a record from a byte buffer costs t time. The average time of accessing n items on the same chain is as following:

$$
\begin{aligned}
t(n) &= \sum_{i=1}^{n} it \\
&= \frac{tn^2(n+1)}{2} \\
t^{'}(n) &= \frac{t}{2}(3n^2 + 2n)
\end{aligned}
$$

The derivative of t(n) is a quadratic function, in other words, once the number of items on a chain is greater than 1, increasing the chain size by one would result in more than 2.5t growth of average accessing time. Since only the number of hash buckets can be specified, what is the relationship between bucket number and chain length?

To discover the connections between bucket chain length and bucket/record number, an experiment with various bucket numbers is done. In the experiment, the record number is equal to the bucket number, MapHash is populated with random keys, and the bucket-chain length distribution is counted. Table 3.1 shows that the distribution is independent with the record size, given that the bucket-to-record ratio is fixed. The data also shows that the usage of hash buckets is very uneven, it is notable that in the last column of the table the frequency of number of hash chain elements tops at 7, and there is only 0.63% hash buckets with 1 item, which is a feature of a normal distribution. A hypothesis is that the majority of hash buckets have a hash chain of n/b items, where n is the total number of records and b is the number of buckets. The frequency chain-length distribution is a

Table 3.1: Bucket Chain Length Distribution

bucket	1k	1M	17M	17M
records	1k	1M	17M	120M
chain length	#buckets			
1	35.08%	59.94%	61.99%	0.63%
2	30.66%	28.47%	27.69%	2.25%
3	18.23%	8.99%	8.16%	5.36%
4	9.41%	2.13%	1.79%	9.49%
5	4.88%	0.40%	0.31%	13.33%
6	0.93%	0.06%	0.04%	15.51%
7	0.70%	0.01%	0.01%	15.34%
8	0.12%			13.13%
9				9.93%
10				6.69%
11				4.06%
12				2.25%
≥ 13				2.03%

normal distribution.

In order to validate the hypothesis, a similar experiment is done. The only difference is the bucket-to-record ratio. The experiment counts the hash-chain length with 1 million buckets and 1, 2, 3, 4 and 10 million records. The result is shown in Table 3.2. The hypothesis about the majority is very close to the experiment. In this experiment, the most common chain length is n/b - 1. Theoretically, the average chain length is n/b , but in the experiment, a portion (5%) of the records are removed, so the total existing records are less than the parameter value, therefore, the most common chain length is not n/b.

The 68-95-99.7 empirical rule [19], also known as a special case of the 3-sigma rule indicates that about 68% of values drawn from a normal distribution are within one standard deviation σ away from the mean. If the frequency chain-length distribution is a normal dis-

Table 3.2: Bucket Chain Length Distribution With 1 million Buckets

records	1M	2M	3M	4M	10M
chain length	#buckets				
1	**36.80 %**	**28.47 %**	16.49 %	8.45 %	0.07 %
2	17.48 %	26.97 %	**23.50 %**	16.17 %	0.32 %
3	5.52 %	17.19 %	22.38 %	**20.50 %**	1.06 %
4	1.31 %	8.09 %	15.89 %	19.45 %	2.54 %
5	0.25 %	3.09 %	9.09 %	14.79 %	4.78 %
6	0.04 %	0.97 %	4.27 %	9.37 %	7.72 %
7	0.01 %	0.26 %	1.75 %	5.02 %	10.42 %
8		0.06 %	0.62 %	2.42 %	12.32 %
9		0.01 %	0.20 %	1.02 %	**12.99 %**
10			0.06 %	0.39 %	12.42 %
11			0.01 %	0.13 %	10.67 %
12				0.04 %	8.41 %
13				0.01 %	6.15 %
14					4.17 %
15					2.62 %
≥16					3.33 %

tribution, it should have 68% of buckets in the $[9-\sigma, 9+\sigma]$ range. The standard deviation is approximately:

$$\sqrt{\sum_{i=1}^{16}(i-9)^2} = 3.02 \tag{3.1}$$

With a Normal probability plot [1] test , it is possible to verify this distribution is normal. Let X be the length of chain, the estimated distribution is:

$$X \sim N(\frac{n}{b}, \sqrt{\frac{n}{b}}) \tag{3.2}$$

An average chain size of 1 means that n = b and approximately 50% of the hash buckets are empty, 68% / 2 = 34% of hash buckets have 1 item, (95% - 68%) / 2 = 14% of hash bucket chains have 1 + sqrt(1) = 2 items, which covers 98% of the buckets of MapHash. The expected time cost of accessing a record is 34%t+14%t+14%x2t / 34%+14%+14% = 1.23t. The best expected time cost MapHash can achieve by increasing the bucket number is t. It is safe to conclude that n = b is a satisfying parameter.

In conclusion, a good balance of performance to storage, is for the hash buckets to be equal to the number of records, which does not take much extra space when records takes hundreds of bytes of storage and a hash bucket only takes 16 bytes. For 100 million 256-byte record, having 100 million hash buckets only uses extra space that is 100x16 / 100x256 = 6.25% of the actual data.

To validate the derivation, a throughput experiment is performed (see Section 4.4 for experiment details). The result is displayed in Fig 3.1 and Fig 3.2.

Performance with hash buckets equal to the number of records produces virtually equivalent performance to hash buckets equal to 1.5 times of the number of records while hash buckets equal to 0.65 times the number of records shows only a minimal slow down. Because MapHash is designed to work in a data cache, it is certain that the hash table knows the maximum number of records before running. Therefore, the number of hash buckets is set to the number of maximum input records on all throughput experiments.

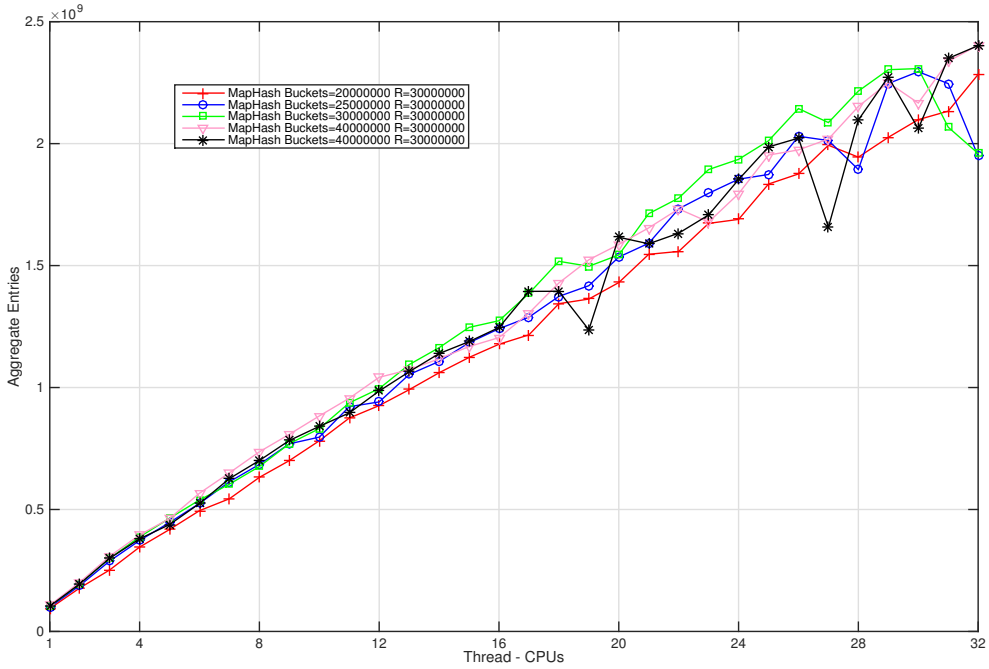

Figure 3.1: MapHash Performance With Different Bucket Numbers, AMD

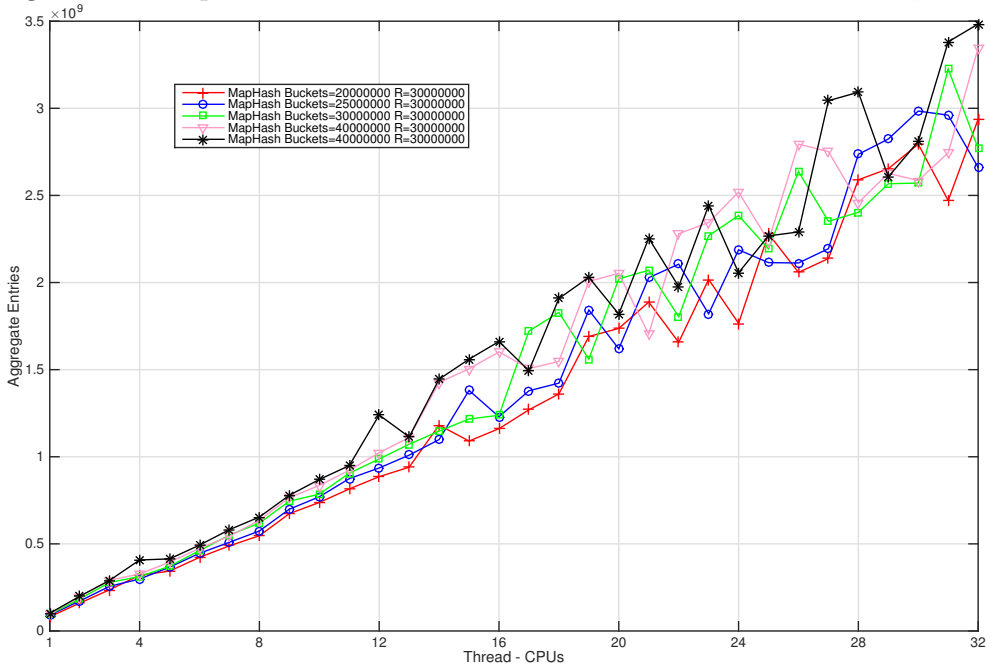

Figure 3.2: MapHash Performance With Different Bucket Numbers, Intel

3.4 Concurrency

There are two main approaches to handle concurrency:

- Atomically read/write a value using copy out/in from the hash table. For a read, the copy acts as the consistent value, but the value in the hash table may immediately change, implying the reader has stale data. For a write, the data in the hash table is always consistently updated with a new complete value. (medium-grain locking)
- Acquire a pointer to a key-value pair and lock the value or fields of the value. For a read or write, no operation can proceed on that value/field until the lock is released, so the data is always fresh (most up-to-date) until the lock is released. (fine-grain locking)

In the copy approach, a user does not have to acquire and release locks because the copying performs all locking implicitly. For the in-situ approach, a user has to explicitly manage locks and duration. Clearly, the copy approach moves more data if only a few fields are accessed. In the end, the copy was selected because of its ease of use, users are likely to access many fields in a record, and reading stale data is virtually impossible to prevent so the copy approach is an easier form of consistency. Hence, the medium-grain locking is used for MapHash, and it is implemented by each hash bucket having a lock to guard operations on the associated hash chain.

Alternative approaches for protecting each hash chain is to use a lock-free queue within the off-heap data-cache. However, the experiments in Chapter 2 reveal a simple spinlock

has better performance compared with a lock-free queue. Another approach is to protect the hash chains with a reader/writer lock to allow multiple readers to simultaneously search the chains, given there are more readers than writers. However, the read operations are fast because the hash chains are usually very short, there is only a 1/#bucket chance two readers hash to the same bucket causing contention, and the number of hash buckets can be very large in MapHash, hence using sophisticated locking per hash chain provides little performance benefit but substantially increases complexity.

3.5 Persistence

A memory-mapped file is used as the backing storage to achieve the persistent requirement. On UNIX, memory mapping is done by the mmap system-call. The mmap call creates a new mapping in the address space of the calling process. The mapping lazily accesses the disk when memory is read/written, and memory is evicted by the virtual-memory system. Java provides a class called MappedByteBuffer to map disk storage through a buffer interface into off-heap memory, meaning the off-heap data is not garbage collected. MappedByteBuffer provides interfaces to get and put a primitive types at a given position in the buffer. The most common primitive type is "long" because it transfers the most bytes (8) in a single operation. MapHash provides an interface similar to a Java hash-table for transferring data to/from the mapped buffer.

While the MapHash data-structure is built in a contiguous storage-area that is byte addressable, it is impossible to use pointers because the contiguous area may be mapped at different addresses each time the backing storage is loaded or by different JVMs. Hence, all

reference within the storage area are accessed using offsets (indexes/subscripts) from the starting address of the mapped storage. Furthermore, Java objects cannot be used in the storage area without serializing the object because of memory layout issues; hence there is a field-wise copy of an object in/out of the mapped storage. Fig 3.3 shows the storage area layout. The MapHash storage has three components: control information, hash buckets, and records. The control information includes the following variables:

- Lock: global lock used for memory allocation and deallocation.
- Sbrk: the index of the last unassigned record before the start of the unused contiguous space of the mapped file. Sbrk is short for segment break.
- End: the index of the first unassigned record at the end of the unused contiguous space of the mapped file.
- Free: the head of a linked list of free records located within the used space of the mapped file. Free space occurs by remove operations.
- RecSize: fixed size of record.
- Files: variable number of N-byte files that compose the hash table, which grows to the maximum size.
- Buckets: fixed number of hash buckets in the hash table.
- MaxSize: fixed maximum size of the hash table in gigabytes, including all internal data.

The hash buckets are an array of tuples:

- Top: head of a linked list connecting records that collide on this bucket.
- Lock: to provide mutual exclusion of the hash chain. All hash-table operation must acquire this lock before accessing a hash chain.

The records are an array of tuples, where each record must:

- start with a key and next pointer field.

The current version of MapHash assumes a record key is a long integer (64-bits). (FINANCIALOGIX also requires a version with double-long integers (128-bits).) The next pointer field is a long integer pointing to the index of the next record in the same hash chain or -1 (null: null is set to be -1 in MapHash because 0 is a valid index). The rest of the record fields store user data. Users need to create marshal/de-marshal operations to store complex user-defined data structures to/from a record. The Rec class used by MapHash to store user data in the dictionary is not required to be a subclass of BytesMarshallable if a user's copy-in and copy-out methods have already provided the effect of serialization.

Java does not provide an off-heap lock service. Therefore, off-heap locking must be provided by a Java Unsafe class or C++ code to do lock and unlock operations on a mapped file. Since support for Java Unsafe may be deprecated, MapHash uses C code via JNI. The Java LockAccess class provides a link to the C++ implementation routines of the exponential back-off spinlock.

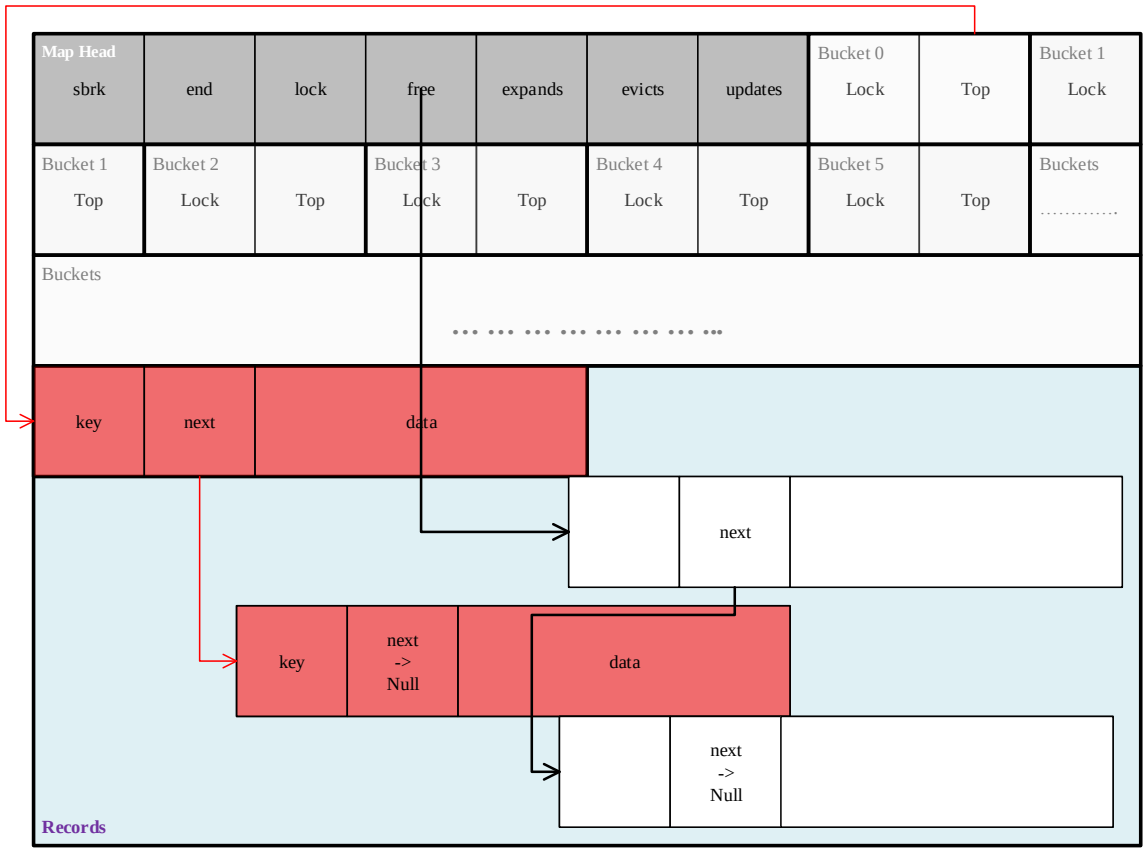

Figure 3.3: Design of MapHash Storage

3.6 MappedFile Storage Layout

The class MappedFile is the main Java class of MapHash, and is the on-heap proxy for the off-heap mapped storage-buffer. While the structure of MappedFile mimics the structure of the off-heap storage, there are substantial differences. MappedFile consists of five parts.

The first part is a set of control variables. The most important variable is an array of byte buffers allowing the mapped file to exceed the Java limit of 2GBs for any individual byte buffer. A record address is computed as a buffer index and offset within the buffer. Hence, a record reference/dereference is treated as a two-level virtual-memory access, where pages are at most 2GB.

The second part is an encapsulation of the byte-buffer interface, where MapHash wraps the getLong and putLong methods with a record interface to translate a record address into the logical byte-buffer into a physical byte-buffer subscript and offset within the specific buffer.

The third part is some basic components for building logical data structures in the mapped file, since Java does not support off-heap objects. The three off-heap data structures are Record, Array and ArrayOfRecords. These three patterns are very expressive, allowing construction of the hash table for MapHash.

- Record pattern supporting field reference/dereference, i.e., $r.f$ or accessing a field of an object at a given address. For simplicity, all fields are assumed to be the same size b bytes. Hence, to access $r.f_i$, the address is computed by $r_{addr} + b \times (i - 1)$;
- Array pattern supporting subscripting of an element, i.e., $a[i]$ or accessing the ith

entry of an array starting at addr. For simplicity, all elements are assumed to be the same size b bytes. Hence, to access $a[i]$, the address is computed by $a_{addr} + b \times i$;

- Array of record pattern supporting subscripting and field reference/dereference of an array element, i.e., $a[i].f$ or accessing a field of an element in an array. For simplicity, all fields are assumed to be the same size b bytes, making all array elements the same size $b \times number\_of\_fields$. Hence, to access $a[i].f_j$, the address is computed by $a_{addr} + b \times number\_of\_fields \times i + b \times (j - 1)$.

These logical data-structures simplify writing code for the actual data structures by providing simple syntax to access off-heap data similar to accessing on-heap data, and hence, decreasing coding time and reducing errors.

The fourth part is the actual data structures components of MapHash, the HashTable and the Records classes. HashTable is a subclass of ArrayOfRecord and is used to store the array of bucket records. The maximum number of buckets is specified at creation and is fixed thereafter. The Records class, also a subclass of ArrayOfRecord, is used to access fields of a record. The hash table only accesses the key and link to perform operations, and data fields to copy values into or out of a record. As stated, copying is used for concurrency reasons, so an internal Record is copied to/from its corresponding external Rec class. Two members copyin and copyout transfer the data. These two routines are restricted to be used internally only, by the Access class. By default, the two routines work at the granularity of long words, but a user can replace these routines for more complex data structures. The reason for these two interfaces is explained in the Get operation section.

The last part is the main body, the Access class. The Access class exposes the hash-table

get/put/remove interface. Access also has its own control variables, instances of HashTable and Records. This organization gives the Access class the ability to manage and access the underlying buffers as if it is a hash table. The rest of this section explains how the internal lock, get, put and remove operations work.

3.7 MapHash Locks

In MapHash, locks are fundamental and support all the concurrent operations. To allow off-heap and multiple JVM access, lock variables must exist in the mapped file. To manipulate the off-heap locks, C++ routines are used and Java JNI is the standard programming interface for inter-language calls. Using JNI, MapHash is able to manipulate the Java byte-buffer from C++ code. The experiments in Section 2.1 indicate that using JNI off-heap locks is not an impediment for high-performance.

Before using the MapHash object, the address of each byte buffer containing the hash table is passed to the C++ lock handler by JNI. When MapHash needs to acquire/release a lock, the logical address of the lock is passed to the JNI locking routine, which then decomposes the address into a physical buffer subscript and offset. The purpose of storing all mapped byte buffers in the C++ scope is to save passing the buffer and computing the buffer's address on each locking call, because computing a buffer's address is costly. Since the hash-bucket array does not grow in size and all locks are in the hash buckets, the number of byte buffers passed to the C++ scope is small, and never needs updating even as the hash grows and adds byte buffers for records.

LockAccess provides two implementations of spinlock: basic and ticket. The basic spinlock does not guarantee eventual progress. The ticket spinlock provides FIFO service ensuring eventual progress. Both the basic and ticket spinlock provide the option of exponential backoff. Testing (not shown) indicated the basic exponential back-off locks is the fastest lock, so it is used for all locking in the MapHash.

3.8 MapHash Get Operation

When MapHash performs a get(key) operation, it first computes the corresponding hash bucket with the hash function. Then it attempts to acquire the bucket lock. After successfully acquiring the lock, it searches through the hash chain for an exact key match and copies that value out if the key exists; otherwise it returns a null pointer.

MapHash passes a record back to the caller by copy instead of by reference. If the get routine returns a record by reference, there is no trivial method to detect when the reader finishes reading. In Java's on-heap data structures, readers get the record by reference, and further writes to the record are completed by read-copy-update, where the isolated record held by a reader is later recycled by garbage collector. However, in MapHash, garabage collector cannot be used. To keep track of the references to a record, MapHash would have to employ a counter for each record, which is deemed too complex. It is shown in Chapter 4 that copying in not a performance bottleneck.

The get operation returns the record by copying out the values into an argument object. It also returns the index of the found record. MapHash returns results through the reference

parameter to eliminate creating a local object to return the data. This approach allows the caller to reuse an object at the call site to hold the record data, and hence, reduce garbage collection over creating a new object to return for each call.

3.9 MapHash Put Operation

When MapHash receives a put(key,rec) operation, it first computes the corresponding hash bucket with the hash function. Then it attempts to acquire the bucket lock. After successfully acquiring the lock, it search through the hash chain for an exact key match and copies the values into the record; otherwise it gets a new empty record by calling the allocation function, and puts the new record at the head of the hash chain because the record may be accessed again quickly.

There is a complex concurrency interaction between the put operation and eviction caused when the hash table is full and cannot expand.

- During a put operation, a thread may need to acquire two locks: the bucket lock and the global allocation lock if a new record must be allocated. Holding two locks is always dangerous as it is easy to create a deadlock.

- During the allocation of a new record from the put operation, an eviction may need to steal a record to make space for the new record. The thread performing the put operation first steals a record from its hash chain, if one is available; otherwise, it must search other hash chains to find a record and steal its storage. The searching thread has acquired the global lock as part of allocation protection and now attempts

to acquire the next hash-bucket lock to find a record. However, another thread may already have acquired this hash-bucket lock and is now attempting to acquire the global lock to allocate a new record. This scenario is a classical deadlock as both threads are holding a lock and waiting for the other thread's lock to be freed. Either the searching thread must drop the global lock or the other thread must drop its hash-bucket lock or both locks must be dropped. In general, dropping a lock means prior work must be redone as data can change the moment the lock is dropped, and starvation can occur because the thread dropping the lock never makes progress. All three dropping scenarios have problems and increase the complexity of the implementation.

It was decided to drop the hash-bucket lock before attempting to acquire the global lock for an allocation. Hence, a thread holding the global lock and performing an eviction search knows any thread holding a hash-bucket lock will release that lock before it tries to perform an allocation and progress can occur.

The lock dropping is achieved as follows. The put routine first acquires the hash-bucket lock, and if the key is found, copies in the data, releases the hash-bucket lock, and returns. If the key is not found, a new record is needed, and the hash-bucket lock is released before performing a speculatively allocation for a new record. The allocation is speculative because the moment the hash-bucket lock is dropped another thread can add the same record so the new storage may be determined as unnecessary after the hash-bucket lock is reacquired, and hence, the speculative record must be freed. This situation should be extremely rare. The only extra cost is the double search for the key: first to

determine if the key is not there, and then to redetermine this fact after obtaining a new record and reacquiring the hash-bucket lock. Since hash chains are short and writes of new keys are rare, this solution shows no negative performance effects and is straightforward to implement. Finally, if the speculative record must be freed, the hash-bucket lock is released first before freeing the record because free acquires the global allocation lock; hence two locks are never held simultaneously.

A disadvantage of this approach is that multiple evicts cannot occur simultaneously, because the global lock is held for the duration of the search for a victim record to steal.

However, evictions should be rare, and holding the global lock ensures two or more threads cannot cycle endlessly attempting to locate a victim record.

3.10 MapHash Remove Operation

When MapHash receives a remove(key) operation, it first computes the corresponding hash bucket with the hash function. Then it attempts to acquire the bucket lock. After successfully acquiring the lock, it searches through the hash chain for an exact key match and unlinks the record from the hash chain and pushes it onto the free-list stack and returns true; otherwise it returns false. The remove routine releases its hash-bucket lock before freeing the record because free acquires the global allocation lock; hence two locks are never held simultaneously.

3.11 Allocation

Allocation is only required by the put operation. Every allocation call is guarded by a global lock, in other words, there can be at most one thread performing memory operations at the same time. Allocation is expected to happen often when constructing the cache and when the records exceeds the storage limit of MapHash, so it is a rare event during the working period of the data cache. MapHash first tries to obtain free space from a free list of removed records. If the free list is empty, the second option is to extend the file size, which involves logically resizing the mapped file and re-mapping the byte buffer to the file. If the hash table cannot be extended because it has reached its maximum size, eviction of a record occurs to steal its storage for the new record.

3.12 Eviction

If the allocator runs out of space, the eviction routine is called, which tries to remove the last record of the current putter-thread's bucket chain (which is locked). If the chain is empty, the putter-thread walks cyclicly around the hash-bucket array looking for a non-empty hash chain. The search is guaranteed to terminate because the hash-table is full so there must be a non-empty hash-chain and no storage can be reallocated as the searching thread holds the global allocation lock. Whenever a hash-chain is chosen, the last record (oldest inserted record) is evicted.

3.13 Large Dataset Support

The project requirements are to handle hash tables to 400GB and beyond. However, the maximum index of Java's MappedByteBuffer is 2GB, so multiple byte buffers are needed for hash tables larger than 2GBs. The rest of this subsection focuses on the design of the large dataset support.

3.13.1 Multiple-File Buffer Proxy

MapHash creates multiple buffers and a byte-buffer proxy implementing the putLong and getLong methods that exposes itself as a normal byte buffer. The hash table treats the proxy layer as an infinitely large buffer.

To compute a physical address from a virtual address in the proxy requires dividing the virtual address by the buffer size to get the quotient and remainder, which are the subscript of the buffer and offset within the buffer, respectively. Since putLong and getLong are performed multiple times for every get/put/remove operation, and divide/modulus are expensive hardware operations (80 to 95 CPU cycles [9]), these routines are a performance bottleneck. To reduce cost, the buffer size is set to a power of 2 so low-cost shifts (1 CPU cycle) are used, as shown in Listing 3.1.

Listing 3.1: Bit Shift of Long Index

```
private long getLong( long word ) {
   return buffers[(int(word>>>MaxFileBits)].getLong((int)(word&(1<<MaxFileBits)−1));
}
private void putLong( long word, long value ) {
```

```
    buffers[(int)(word>>>MaxFileBits)].putLong((int)(word&(1<<MaxFileBits)−1),value);
}
```

Unfortunately, the Java restriction on the byte-buffer size is not 2GB but 2GB-1, so 1GB buffers are used.

3.13.2 Multiple-File Backing Storage

The file backing the hash table can be any size. Furthermore, it is possible to map multiple byte buffers to an arbitrary starting location in the large file. Finally, it is possible to expand and contract the single file.

However, instead of using a single file for backing storage, individual files are created for each byte buffer, numbered 0 to $N-1$. The reason is that multiple files may be able to take better advantage when placed on multiple disks or allow better implicit stripping across multiple disks. Each file is 1GB to match with size of the byte buffer. Correspondingly, hash-table storage is extended in units of 1GB chunks, adding an extra file.

Finally, when a MapHash hash-table is access via a file name, such as "myhashtable", a check is made for file "myhashtable.0". If "myhashtable.0" does not exist, the necessary byte buffers and files are created to contain the components of the hash table as specified by other start up arguments, such as initial number of hash buckets and records. Otherwise, file "myhashtable.0" is mapped, and the control information at the start of the hash table indicates how many other files must exist for this hash table, and the appropriate number of byte buffers are created and mapped to the associated file. Through this latter mechanism, the hash table is persistent and can be reused at different times, or different JVMs can

connect to the hash table at any time.

Chapter 4

Performance of MapHash

In this section, the performance of MapHash is compared to ConcurrentHashMap [11], ChronicleMap [10] and Java-HashMap based on throughput. The other hash-tables are chosen for the following reasons.

- MapHash, ConcurrentHashMap and ChronicleMap are designed for a concurrent reader/writer threads. JavaHashMap is sequential, but if the members are wrapped with a reentrant lock, it can support multiple threads.
- ConcurrentHashMap only uses on-heap memory; hence, it cannot persisted on disk as a data cache. But with faster on-heap memory accesses and non-blocking features backed by a garbage collector, ConcurrentHashMap can provide an optimal upper bound for MapHash. JavaHashMap only uses on-heap memory; hence, it cannot persisted on disk as a data cache.

- ChronicleMap is from the net.openhft package, where hft means High-frequency Trading. ChronicleMap is designed to provide an efficient persisted storage to traders. This goal is identical to MapHash's goal, so ChronicleMap provides a peer performance comparison;

Because of MapHash's simple design, it requires much fewer settings to achieve good performance, and it is simpler to use.

Although latency is also important, there is no convenient way to measure it, moreover, measuring the latency can undermine the performance. However, since MapHash uses spin locks and worker threads are never blocked, a high throughput implies a low average latency.

4.1 Chronicle Map and Tuning

ChronicleMap is a low-latency replicated-key value-store that can be used across the network. It has the features of eventual consistency, persistence, and good performance, as claimed by the developers. Hence, it is a candidate for the data cache. The most important feature that makes ChronicleMap different from the Java native dictionaries is persistence, allowing ChronicleMap to be stored in a file and shared across JVMs. ChronicleMap is recommended in the following situations:

- a large number of small records (64-256 bytes);
- to minimize garbage produced, and medium lived objects;

- when persistence data is needed;
- when data has to be shared between JVMs.

ChronicleMap achieves off-heap access by serializing objects to bytes on the disk. Therefore it requires a Serializable object, or implementing a set of serialization routines. Using this general serialization approach that can result in poor performance. Thus, in order to get higher performance on ChronicleMap, the second approach is often applied, implementing an abstract class called BytesMarshallable. The fastest serialization routine is to write each field to the byte buffer.

In order for ChronicleMap to work with a large number of records, several parameters must be specified for the map builder, such as the number of segments and the number of records; otherwise, the map can run out of the automatic allocated memory. (Segments are subdivisions of the hash table, e.g., divide N hash buckets into M segments of size N/M, resulting in two-level hashing. This partitioning is done to reduce the load on each segment by multiple threads, as thread operations on each segment are independent.)

4.2 Java ConcurrentHashMap and Tuning

Java's ConcurrentHashMap uses an implementation where the hash table stores records in bucket lists. However, there are three differences from MapHash:

- ConcurrentHashMap is in on-heap memory, and thus operations on the hash table affect the garbage collection;

- ConcurrentHashMap uses a two layer index. When retrieving records, it first locates the segment then the bucket;

- ConcurrentHashMap only locks on writes, and uses a write lock per segment to guarantee mutual exclusion among write operations to the same segment. A write creates a new record, and leaves prior versions of the record for access by outstanding readers with references to these records. Hence, a reader always see a consistent record from a single write. Prior records are eventually garbage collected when readers no longer reference them.

For workloads composed of mostly reads, ConcurrentHashMap achieves very good performance because there is no locking. The advantage of on-heap memory and no reader lock makes ConcurrentHashMap a practical upper bound for comparison with other hash tables when there is sufficient memory.

However, by default, Java sets a memory limit for a virtual machine. When the size of the hash table grows to this limit, the garbage collector interrupts the entire program (all threads) to recycle memory. But the collection fails because it needs memory to collect and there is none. Hence, the program starts an endless cycle of garbage collection. This phenomenon was observed during experiments. To mitigate this issue, the maximum available memory is specified for experiments using the `-Xmx` option. In general, ConcurrentHashMap needed 2-4 times more memory to achieve high performance than the off-heap hash-tables, which only use storage for administration, hash buckets and records. Furthermore, the off-heap hash-tables do not contribute to garbage collection. Hence, ConcurrentHashMap has a large memory footprint, which affects the maximum hash-table size

and other applications on a machine.

4.3 Experiment

The following experiment is a micro-benchmark used to compare the three hash tables. While all the hash tables are not directly comparable given the different project requirements, the comparisons are still interesting.

4.3.1 Design

As stated, MapHash is required to have low latency and high throughput, provide access by multiple JVMs and persistence, with a workload of more readers than writers. (The relaxation of a single writer is dropped because all the tested hash tables can handle multiple writers.) The micro-benchmark is designed to measure a likely usage scenario for the FINANCIALOGIX-F1 trading system.

The workload is a trace of largely unique keys. For each key, a random choice is made for an operation, where the operation distribution is 80% reads, 15% writes and 5% removes. The same trace is run by separate threads starting at different offsets, and each thread is run on a separate core to ensure maximum parallel access of the hash table. The number of threads is varied from 1 to 32, and the hash table is populated with 256-byte records, which includes an 8-byte key and link field (leaving 30 $\times$ 8-byte data fields). The number of hash buckets and records are two additional variables. In the target system, the server has more cores (64-256) with each core running an automated trading-system.

Listing 4.1: General Map Interface

```
interface MyHashMap<R> {
   // read record matching key, return record reference or null
   abstract public R get( long key );
   // read record matching key into rec, return record reference or null if no key
   abstract public R get( long key, R rec );
   // write key/value matching key, return previous value for update, or null for new key
   abstract public R put( long key, R rec );
   // remove key/value matching key, or null if no key
   abstract public R remove( long key );
}
```

To compare the performance of MapHash, Java ConcurrentHashMap and ChronicleMap, a universal interface is created to mask the differences among the hash-table interfaces (see Listing 4.1).

4.3.2 Setup

An experiment consists of three phases: initialization, running and statistics phase. During initialization, the hash map is constructed and populated with N randomly generated key-value pairs. The random-number generator produces a sequence of almost distinct keys for large values of N (10M to 100M). (The keys have less than 0.1% duplicates.) All keys are inserted before the experiment to mimic a fully running system, as the start up phase is normally very short. In some runs, only a subset of the keys is loaded to fill the hash table to its maxiumum size; the remaining keys cause evictions on puts because the hash table is full.

During running, W worker threads are created to run workload traces on the hash table. As shown in Listing 4.2, a worker loops randomly performing a get, put, or remove operation based on the 80%, 15%, 5% pattern. Each worker has a set of counters accumulating performance information. The main program sets a shared start variable and sleeps for T seconds. After the sleep, the main program sets a shared stop variable and joins with the threads. The worker delays until the start signal and terminates on the stop signal. For each inserted record, the key is written into the record and into the last field. For each fetched record, the key used to get the record must be the same at the key in the record and in the last field. If this test fails, it implies the hash table is corrupted, most likely by a concurrency failure. This validation check also helps prevent the JIT from identifying some or all of the worker code as dead code, and changing the intent of the experiment.

During statistics gathering, the main program collects data from workers and calculates the throughput, averages, and standard deviation.

4.4 Experiment Setup

The performance experiments are performed on two different multicore systems to determine if there is consistency across platforms:

- Supermicro AS-1042G-TF with four sockets, each containing an AMD Abu Dhabi 6380 16 core 2.5 GHz, equals 64 cores, running Linux v3.13.0-49, compiling with javac 1.8.0_60; and

Listing 4.2: Worker's routine

```
public void run() {
    Rec rec = new Rec( 0 );                    // if possible reuse object

    while ( ! running  );                      // wait until harness starts experiment

    for ( int i = Id * Trace / Threads; running; i = (i + 1) % Trace ) {
        long key = test.keys[(int)i];          // get key from trace
        int op = test.operations[i % test.Operations]; // get operation for key
        switch ( op ) {                        // process operation
          case 0: // get
            Rec ret = dictionary.get( key, rec );  // return may be null so do not overwrite rec
            gets += 1;
            if ( ret == null ) fails += 1;     // get non-existent key ?
            // check for corrupted record
            else if ( ret.key != key || ret.f[29] != key ) { // access start/end of record
                System.err.println( "bad record: key " );
                System.exit( 1 );
            } // if
            break;
          case 1: // put
            if ( Reuse ) {                     // if possible reuse object
                rec.key = key;
            } else {
                rec = new Rec( key );
            } // if
            rec.f[29] = key;
            dictionary.put( key, rec );
            puts += 1;
            break;
          case 2: // remove
            dictionary.remove( key );
            removes += 1;
            break;
        } // switch
    } // for
} // run
```

- Supermicro SYS-8017R-TF+ with four sockets, each containing an Intel Xeon 8 core 2.6 GHz E5-4620V2, equals 32 cores, turbo-boost on, running Linux v3.13.0-49, compiling with javac 1.8.0_60.

Only two sockets are used on the AMD, and all four on the Intel for the 32 core experiments; hence, NUMA effects with respect to accessing shared data occur.

4.5 Different Experiments

The following experiments are performed on as many of the hash tables as appropriate:

- MapHash, ConcurrentHashMap and Chronicle non-uniform memory placement (NUMA) performance test.
- MapHash eviction performance test.
- MapHash multiple JVM performance test.
- MapHash, JavaHashMap, ConcurrentHashMap and ChronicleMap throughput performance test.

4.6 Removal: JavaHashMap Experiment

JavaHashMap is a sequential hash map implemented by adding a Java blocking lock (see Section 2.2) to Java's native hash map. This level of locking is coarse-grained, while

the other hash tables all use medium fine-gain locking to support concurrent hash-table operations. Nevertheless, the JavaHashMap provides a performance lower bound for a concurrent dictionary. Figures 4.1 and 4.2 show that JavaHashMap performs poorly in comparison to the other hash tables. Even though the Java blocking lock has unfair lock acquisition, the coarse-grain locking dominates the performance, as hash table operations are serialized. Hence, JavaHashMap is removed from the testing suite because it slowed down obtaining experimental results.

4.7 Memory Placement

The hardware on multi-socket computers associates different banks of memory with each socket. There is a higher cost for accessing the memory bank of another socket, resulting in NUMA memory costs for the cores on that socket. During early experiments, significant jitter was noted in the results, as seen in Figures 4.3 and 4.4, on both the AMD and Intel computers. (The exact meaning of the experiment is immaterial for this discussion.) It was determined that the jitter results from the way the operating system assigns memory banks. The default memory policy is to favour one node, which becomes a contention bottleneck when there are large data accesses. By changing the memory policy to interleaving, data is distributed to the memory banks in a round-robin manner. As a result, the jitter disappeared, as seen in Figures 4.5 and 4.6, even though more NUMA accesses occur, because the contention (random delays) on each memory bank is significantly less. Due to these advantages, all other experiments are done with the interleaving policy.

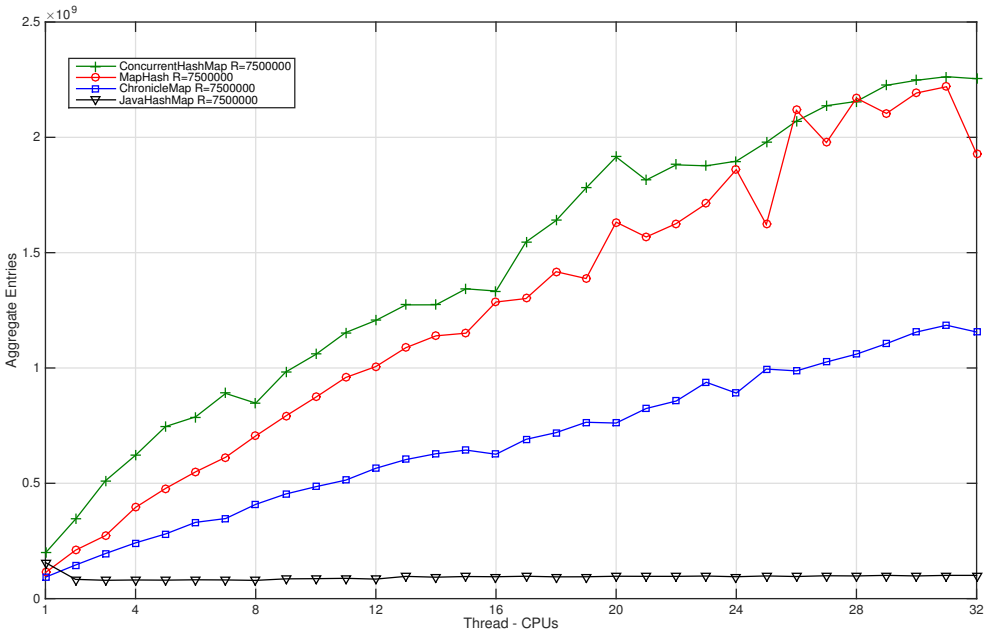

Figure 4.1: Throughputs of All Maps, AMD

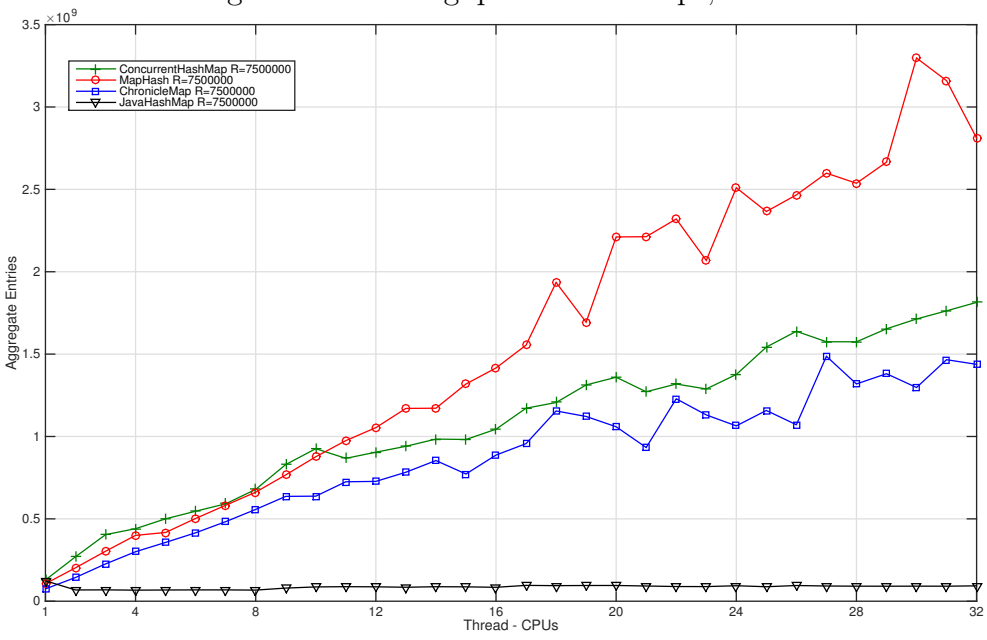

Figure 4.2: Throughputs of All Maps, Intel

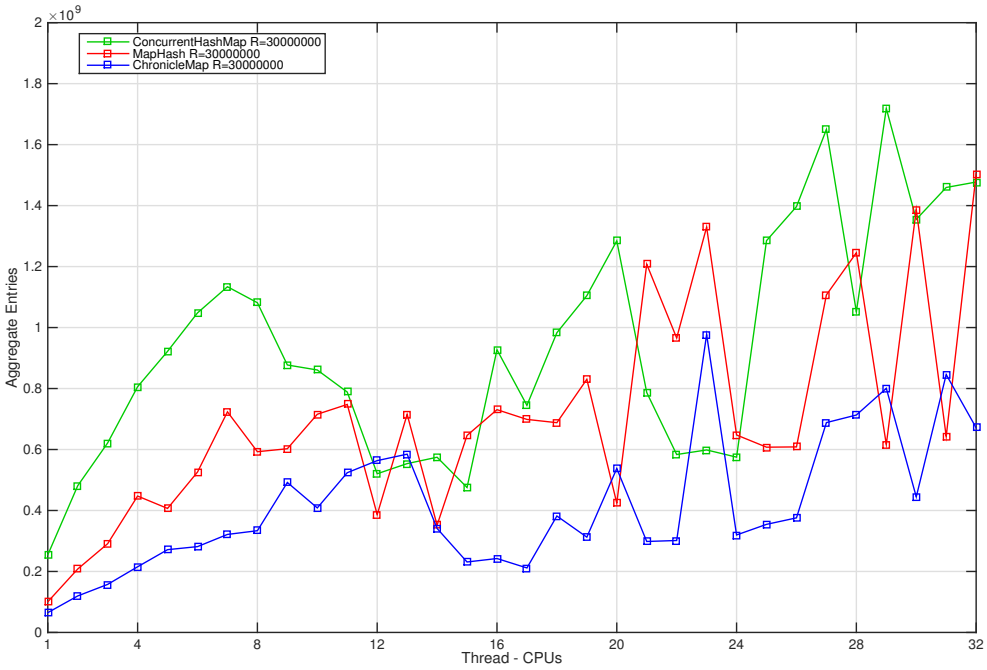

Figure 4.3: Placement Performance, default policy, AMD

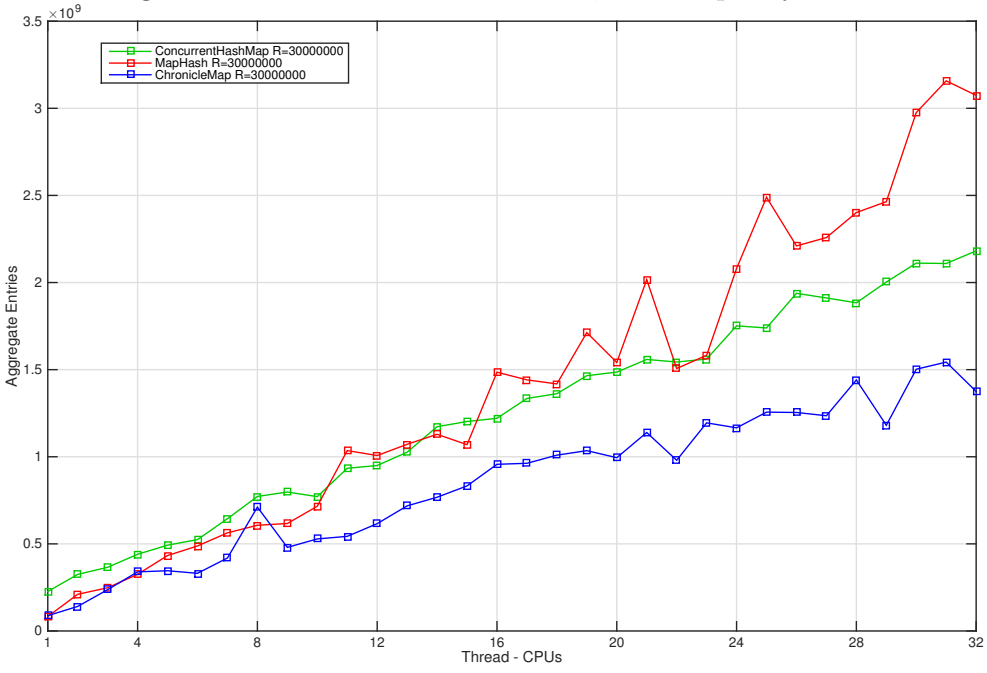

Figure 4.4: Placement Performance, default policy, Intel

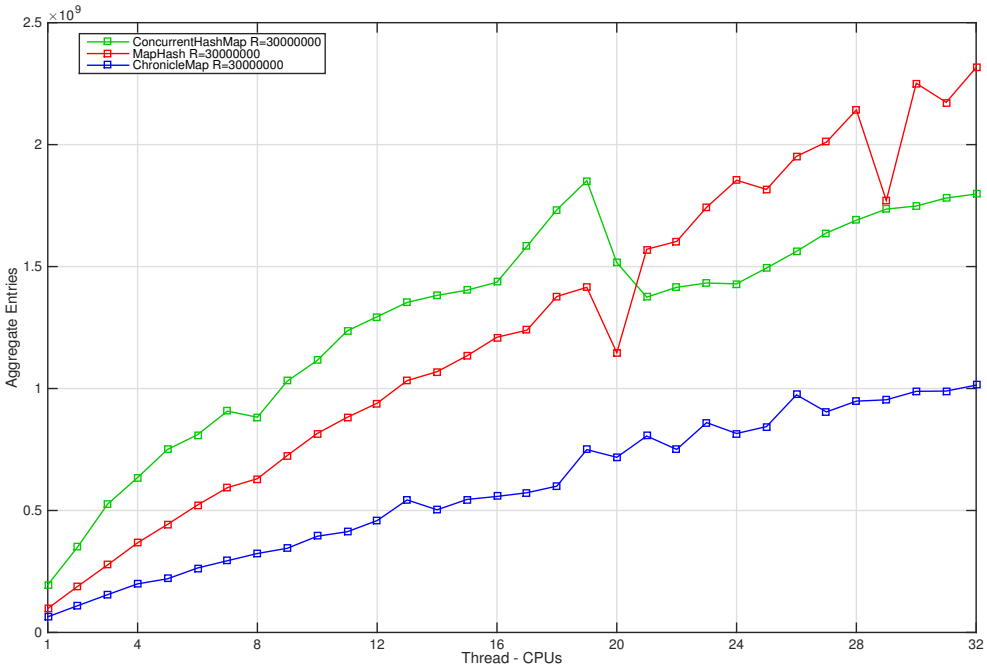

Figure 4.5: Placement Performance, interleaving, AMD

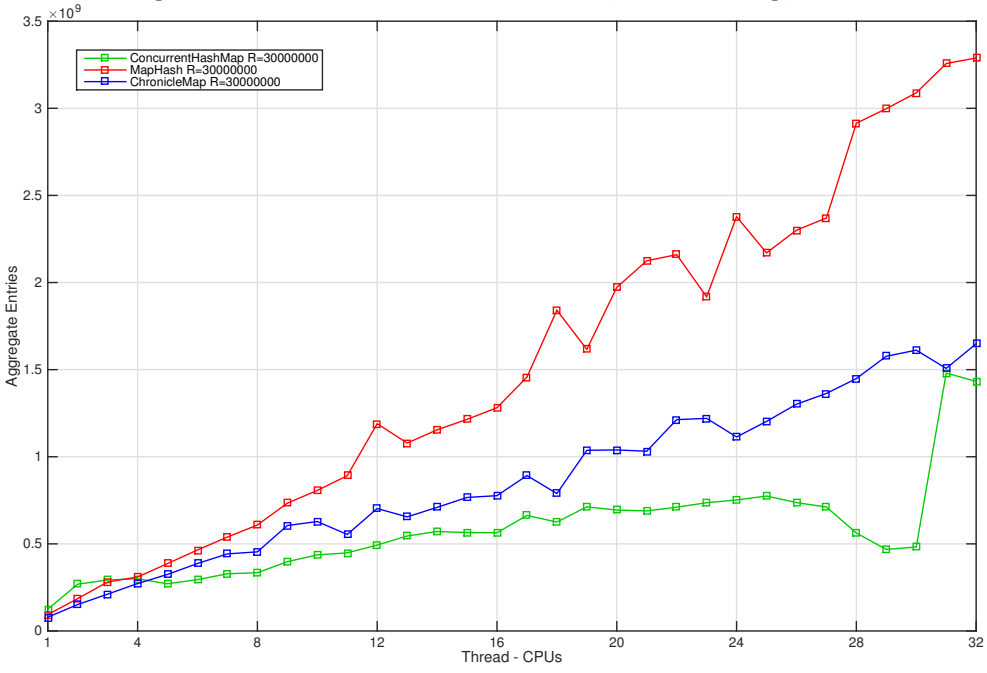

Figure 4.6: Placement Performance, interleaving, Intel

4.8 MapHash Eviction Experiment

MapHash implements record eviction because it is designed to be used in a restricted-sized data cache. Nevertheless, evictions should be rare because eviction cost is significantly higher than a normal insert operation. A high eviction rate indicates the hash table needs to be extended, which is supported by MapHash. It is difficult to draw any quantitative results about an event that is not supposed to occur.

The following experiment shows that a small eviction rate does not significantly affect performance, but as evictions increase, there is a noticeable decrease in performance. The experiment loads a hash table with 15M records and fixes the maximum size at 4GBs, which just hold the records (15M $\times$ (256-byte records + 16-byte hash buckets), approximately 4GB). The operation traces are varied by 15M, 30M, 60M, to 120M keys. For records equal to trace length (15M), there are no evictions. The larger traces contain keys that are not initially loaded into the hash table, and these key cause evicts in the full hash table for insert operations. As the trace lengthens, there are more keys that cause evictions so the eviction rate increases. Because there are only 15% insertions, the evictions occur randomly along the trace. Figures 4.7 and 4.8 show for a 30M trace, with a small number of evictions, performance is similar to the no evict case (15M). As the trace doubles, the number of evictions double, and performance begins to drop significantly. Performance plateaus when virtually all the inserts cause evicts. MapHash is designed to work in a data cache where it is unlikely the number of cached records completely exceeds the allowed space, so the large numbers of evictions are not expected. The results already show that the performance is not seriously influenced by a small number of evictions.

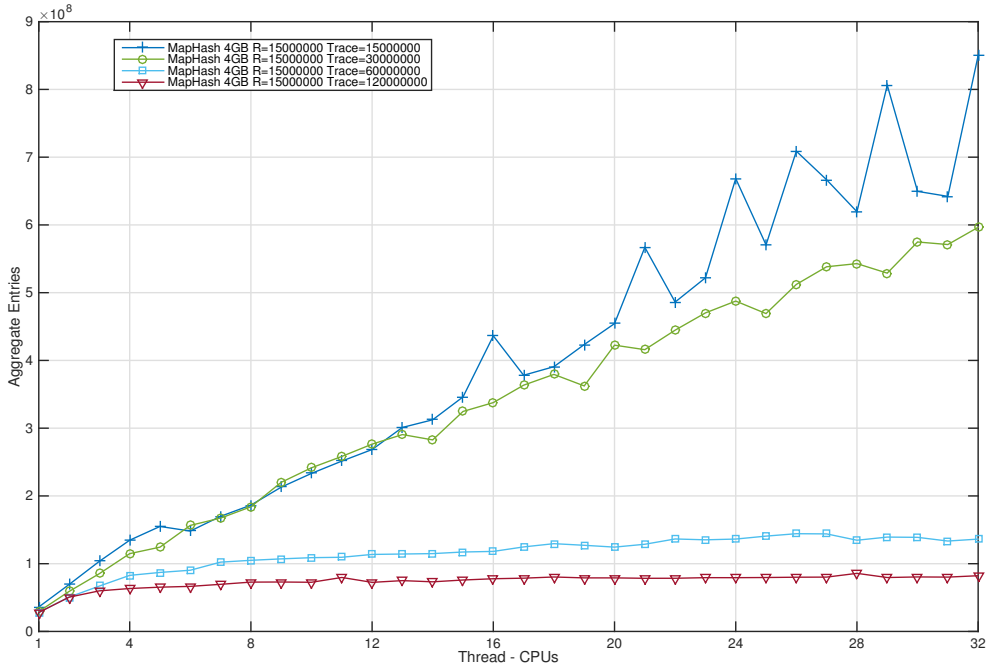

Figure 4.7: Eviction Performance With Different Trace Size, AMD

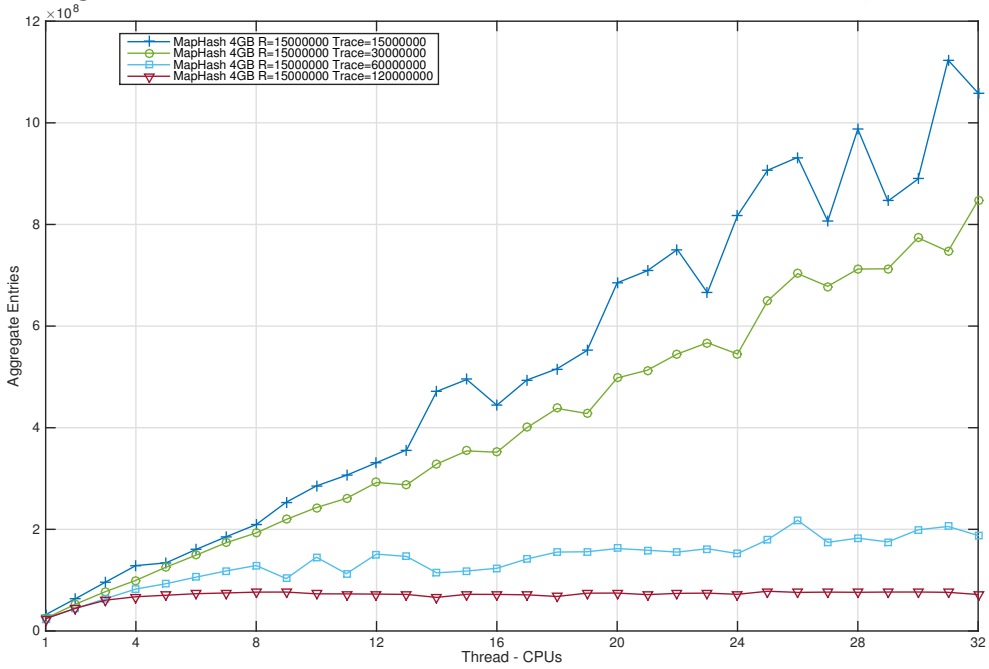

Figure 4.8: Eviction Performance With Different Trace Size, Intel

ChronicleMap does not automatically evict when a threshold is reached. It does provide a mechanism for user code to implement eviction using the map.remove functionality. Otherwise, ChronicleMap keeps maximums set at configuration (number of entries and average record size), and fails if the maximums are exceeded.

4.9 MapHash Multiple JVM

MapHash is targeted to support a multiple JVM scenario. The following experiment checks if using multiple JVMs with multiple threads attached to the hash table is different from a single JVM attached to the hash table with an equivalent number of threads. Only MapHash and ChronicleMap support multiple JVMs, so only these two hash tables are tested. The experimental setup populates a hash table, and leaves the hash table for the second part of the experiment. Then 4 JVMs are started, each varying the number of worker threads from 1 to 8, performing the normal throughput experiment of get/put/remove operations on a hash table with 30M records. The number of workers in each JVM is always the same, so the total number of workers are 4, 8, 12, 16, 20, 24, 28 and 32. The JVMs are not synchronized so one may finish before another and press on to the next number of threads. The results are shown in Figures 4.9 and 4.10 with the corresponding experiment with a single JVM and 30M records. There is no noticeable performance difference between a single and multiple JVMs, which is acceptable.

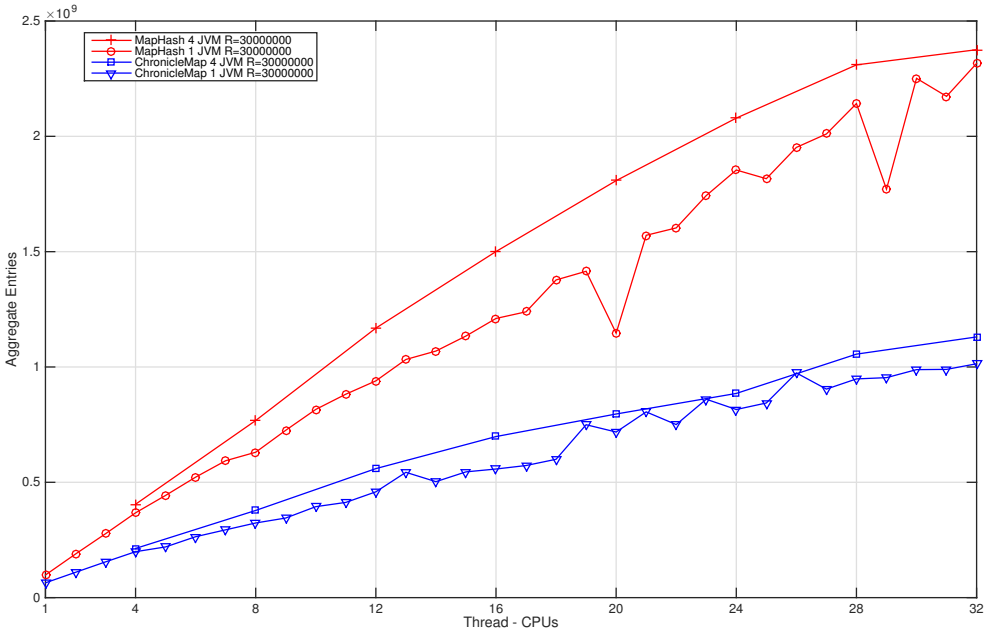

Figure 4.9: Multiple JVM Performance, AMD

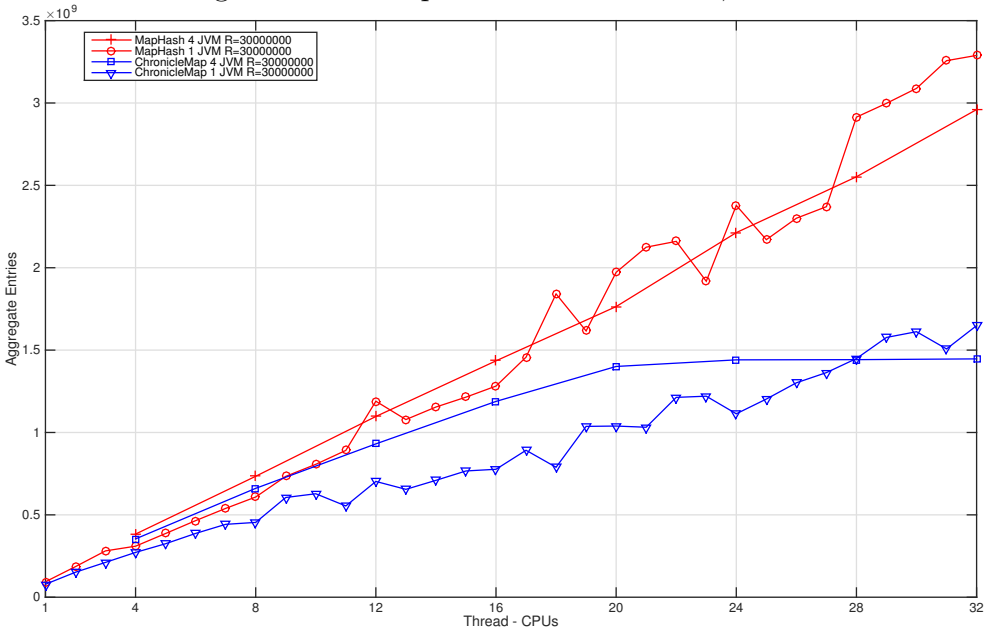

Figure 4.10: Multiple JVM Performance, Intel

4.10 Scaling Experiment

One of the most important goals for MapHash is to scale to large hash tables. The following experiment ramps up the size of the hash table through the following sizes 4GB, 8GB, 16GB, 32GB, 64GB, and 128GB. The number of hash buckets is equal to the number of records, there are no evicts, and the hash table fits in memory. Figures 4.11 and 4.12 show the results of the scaling experiments.

Performance is in three groups. First is ConcurrentHashMap, which is normally greater than or equal to MapHash because of its ability to perform lock-free reads. However, ConcurrentHashMap has a large memory footprint to achieve lock-free reads, and hence, its performance begins to drop or the experiments time out (missing data points). In all experiments, ConcurrentHashMap is given the maximum memory available on the computer which is 2 to 4 times the maximum size of the hash table. Second is MapHash, which is usually twice as fast as third place ChronicleMap.

For each hash table, results only diminish slightly as the size increases. The reason is that each experiment is run for 60 seconds, and the number of operations during that period is relatively constant, independent of the size. The only difference is the spatial access to data for the operations. For small hash tables, less unique memory is accessed and more memory is accessed multiple times because the trace is repeated. For large hash tables, more memory is accessed and less memory is accessed multiple times because the trace may not even repeat. This difference in spatial access is seen as a slight decrease in performance as the hash table increases in size for each hash table. The decease likely comes from the fixed-sized TLB, which spills more frequently as the number of unique

memory accesses increases. The experiments show the big advantage MapHash has when dealing with large scale datasets.

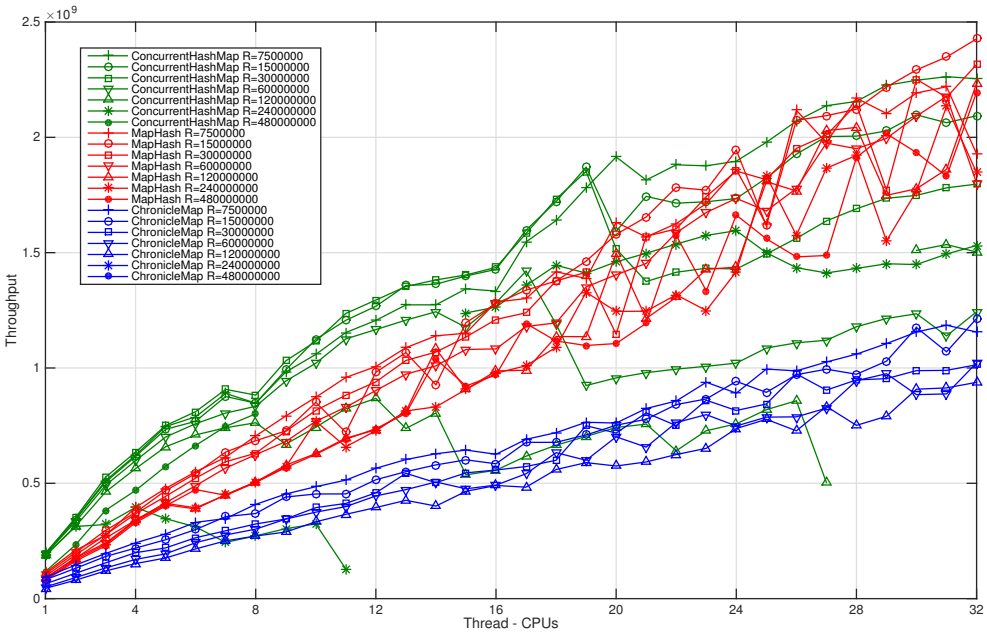

Figure 4.11: Scaling Performance, AMD

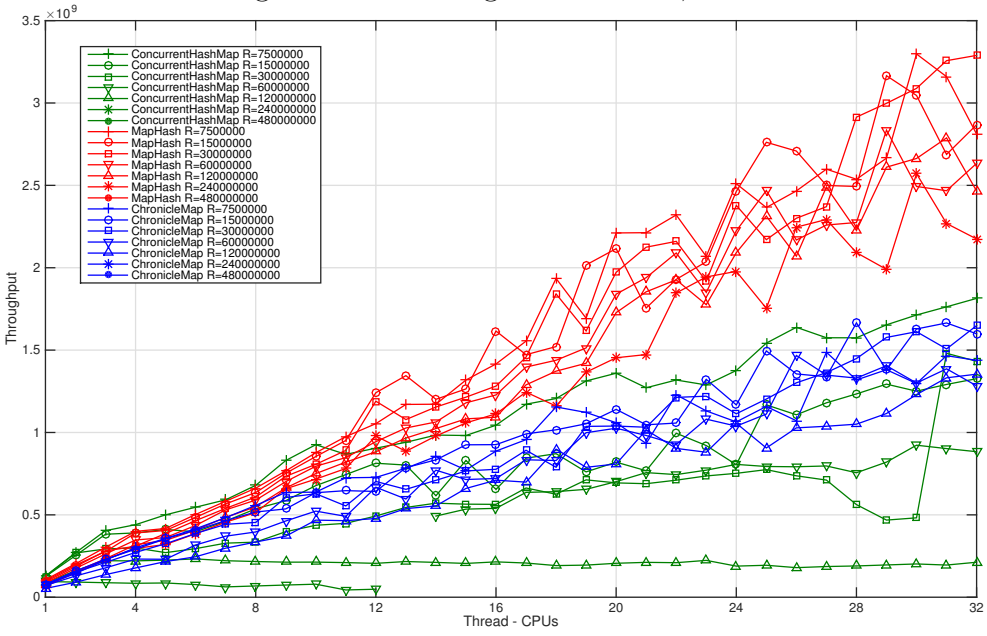

Figure 4.12: Scaling Performance, Intel

Chapter 5

Conclusion

MapHash achieves all the initial goals of providing a high-performance off-heap concurrent hash table, which is twice as fast as the best commercial product available for high-performance financial trading. MapHash consists of only 500 lines of Java code and 40 lines of C code. The experimental harness for testing all the hash tables is 400 lines of code.

MapHash achieves some of its simplicity because of two relaxations: fixed-length records and fixed-size hash bucket array, which allows straightforward memory management in the off-heap storage. Fixed-sized records are seldom a problem even when there are different sized records, because the records do not vary in size significantly; hence, having to always store the maximum record size only wastes a small amount of storage. Not expanding the hash-bucket array dynamically is not a problem in trading scenarios because the initial number of hash buckets can be large, and the number of new insert keys after start up is

usually small. MapHash does allow dynamic extension of records, which is more important in trading, and MapHash has been scaled to 400GB by FINANCIALOGIX on a large Amazon cloud-server. As a precaution against overflow, MapHash also supports eviction of records at low cost. Finally, like Chronicle, MapHash is garbage collection free, but uses less overall memory and loads significantly faster.

MapHash also achieves it simplicity and performance by using very simple implementation techniques for concurrency and storage management. While more complex mechanisms were examined and tested, the simplest implementations always gave the best performance, possibly because the workload is straightforward, so handling complex special cases was unnecessary.

5.1 Future Work

MapHash does not attempt to shrink storage that becomes free at the end of the storage area or have a mechanism to compact the storage area. Both of these operations are possible because there are no reference given out to data in the storage area. Nevertheless, storage recovery can only occur off-line.

While MapHash supports multiple JVMs on the same server, it does not provide replication to multiple JVMs across a distributed system, as is available in ChronicleMap. Some form of proxy mechanism would have to be created on the client to access the MapHash data cache.

Further testing of locking versus lock-free for the hash chains is needed to see if pre-

emption of the kernel thread holding the lock occurs in practice.

References

[1] SC t Abrahams and ET Keve. Normal probability plot analysis of error in measured and derived quantities and standard deviations. *Acta Crystallographica Section A: Crystal Physics, Diffraction, Theoretical and General Crystallography*, 27(2):157–165, 1971.

[2] Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. 1(1):6–16, January 1990.

[3] Woody Canaday. Global foreign exchange services. Technical report, Greenwich Associates, Stamford Connecticut, U.S.A., March 2014. https://www.greenwich.com/~/media/Files/Greenwich%20Rankings/Awards/2014/Mar/FES-2014-QSL.pdf.

[4] Richard J Cichelli. Minimal perfect hash functions made simple. *Communications of the ACM*, 23(1):17–19, 1980.

[5] N. Dale. *C++ Plus Data Structures*. Jones & Bartlett Learning, 2003.

[6] Xstream trading suite. http://www.financialogixgroup.com, 2015.

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

[8] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. *Hopscotch Hashing*. 2008.

[9] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-018. March 2009.

[10] Peter Lawrey. Chronicle map. http://chronicle.software/products/chronicle-map/.

[11] Doug Lea. Hash table util.concurrent.ConcurrentHashMap, revision 1.3, in JSR-166, the proposed Java Concurrency Package.

[12] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, April 2014. USENIX Association.

[13] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[14] Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6):491–504, 2004.

[15] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'96, pages 267–275, New York, NY, USA, 1996. ACM.

[16] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51:122–144, 2004.

[17] RedisLabs. How fast is Redis? http://redis.io/topics/benchmarks.

[18] RedisLabs. Redis Homepage. http://redis.io.

[19] P. Westfall and K.S.S. Henning. *Understanding Advanced Statistical Methods*. Chapman & Hall/CRC Texts in Statistical Science. CRC Press, 2013.