Multi-reader, multi-writer parallel cuckoo hashing

John Siebenthaler

A Thesis in the Field of Mathematics and Computation

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

May 2019

Abstract

In 2001, Pagh and Rodler described a single-threaded hash table design called cuckoo hashing (R. Pagh & Rodler, 2001). Since then, the rise of multi-core processors and large datasets have motivated the development and refinement of concurrent hash tables. We (a) give a brief review of relevant theory of hash functions and hash algorithms; (b) describe a multi-reader, multi-writer parallel cuckoo hash table design; and (c) describe the results of experiments with changing parameters including number of threads, failure criteria, bucket selection strategies, lock granularity, and use of a stash. We suggest multiple areas for future work, varying the number of slots per bucket, and determining whether the depth-first approach of classic cuckoo hashing can have competitive performance under the proposed algorithm compared to a breadth-first approach.

1 Acknowledgements

Thanks to Michael Mitzenmacher for agreeing to be my thesis director and for all of the thought-provoking topics in his Algorithms and Algorithms at the End of the Wire courses that got me interested in cuckoo hashing (and many other topics). Thanks to my former coworker and friend BJ Singh for reviewing the thesis draft and providing valuable feedback. Thanks to Dr. Jeff Parker, who read at least two drafts of my thesis proposal and provided valuable feedback and advice. Thanks to my wife Chau for being patient through all the coursework and thesis work.

Contents

1 Acknowledgements **2**

2 Introduction **9**

2.1 Hash functions and hash tables 10

2.2 Collision resolution 11

2.3 Basic framework for hashing 13

2.4 Basic challenge of hash functions 14

2.5 Universal classes of hash functions 16

2.6 Limits of applicability of universality 20

2.7 Balanced allocations 21

2.8 Cuckoo hashing 22

2.9 Motivation for investigation of parallel cuckoo hashing 25

3 Related Work in concurrent cuckoo hashing **26**

3.1 Optimistic cuckoo hashing (Fan et al) 26

3.1.1 Tag-based Lookup/Insert 26

3.2 Li et al 27

4 Deadlock in naive parallel cuckoo hashing **27**

5 Hash table design **30**

5.1 Terms 30

5.1.1 Locks 31

5.1.2 Lock ordering 31

5.1.3 Thread 31

5.2 Operations 32

5.3 Memory layout 32

5.3.1 Multiple slots per bucket (blocked cuckoo hashing) 33
5.3.2 Lock ordering 34
5.4 Choice of Hash Function 35
5.5 Deadlock prevention 35
5.6 Cache-friendly discipline 36

6 Algorithm description **37**
6.1 Hierarchical organization and addressing 37
6.2 Locking 37
6.3 Version counters 38
6.3.1 In-flight insert problem 38
6.4 Insert 39
6.5 Delete 47
6.6 Lookup 47
6.7 Two levels of hash function 47
6.8 Abstraction of hash function interface 48

7 Experimental Results **48**
7.1 Workloads 48
7.2 Failure modes 49
7.3 First round of experiments 49
7.4 Choosing the least loaded bucket is better than randomly choosing a bucket 50
7.5 Retrying random slot ids improves inserts before failure 52
7.6 Decreasing maximum path length from 128 to 64 53
7.7 Decreasing thread count to 1 54
7.8 Experiments with high throughput workloads and a stash 56
7.8.1 Stash algorithm 56

7.8.2 Using feature elimination to investigate design parameters . . 58
7.8.3 Behavior with maximum inserts set to one million 58
7.8.4 Behavior with one million maximum inserts compared to four million 59
7.8.5 Results grouped by feature combinations 59
7.8.6 "Breaking" the 3,STASH,LL configuration at 98% occupancy 61
7.9 Effect of stash on inserts per second 62
7.10 Comparison with libcuckoo implementation of Goyal, Fan, Li, Anderson and Kaminsky 65

8 Conclusions and future work **65**

9 Notes **67**

References **67**

List of Figures

1 Density plots for the number of inserts before a failure with $1 \leq n \leq 6$ hash functions. All experiments were run on a hash table with 16,384 buckets, and one slot per bucket. An insert is considered failed if it does not succeed within 100 ejections. 24
2 Deadlock example with a naive implementation of parallel cuckoo hashing using per-bucket locks and one slot per bucket. Columns are threads of execution, and rows are discrete time steps. Gray items are deadlocked operations. The notation $a(0, 1)$ means key a can reside in bucket 0 or bucket 1. 28
3 Memory layout for the data store of the hash table. 32

4 Minimum number of successful inserts among 8 threads running the high contention workload. “LL” means least loaded bucket choice strategy, “R” means random. “LL.3” or “R.3” means probe up to three slots before considering the chosen bucket full. The median of each strategy’s readings is printed above the box plot. The suffixed number indicates the maximum number of random slot probes within the bucket before ejecting an occupant. If one of the probed slots is empty, it will be selected. If all retries are executed and no empty slot is probed, insert is recursively called with the occupant of the last slot probed as the ejectee. If an insert attempt requires more than 128 ejections, the thread exits. All experiments were run with two hash functions. 51

5 Maximum number of successful inserts among 8 threads running the high contention workload. “LL” means least loaded bucket choice strategy, “R” means random. The maximum number of successful inserts differs from the minimum number of successful inserts shown in figure 4 on page 51. All experiments were run with two hash functions. 52

6 Minimum number of successful inserts among 8 threads running the high contention workload with maximum path length of 64 (reduced from 128). 54

7 Number of successful inserts for 1 thread running the high contention workload with maximum path length of 64. 55

List of Tables

1 Some common notation used in the thesis. 10

2 Common top-level hash table parameters. For all of the experiments in this thesis, two hash functions were used ($d = 2$), along with 32 buckets per bucket zone ($n = 32$), and 32 slots per bucket ($q = 32$). The number of bucket zones per component table (m) is a function of these parameters along with the required storage capacity of the structure, which is passed as a parameter to the table creation code. . 30

3 Table summarizing results with one thread and 8 threads, indicating that thread count does not significantly affect table load. 55

4 Summaries of successful number of inserts per thread (count, mean, std, min) from the second round of experiments, with a high throughput workload (each thread inserting from a disjoint set of keys) under heavy occupancy (97% of slots occupied). Each entry contains the relevant data from the first round (one million inserts maximum), followed by the data from the second round in parentheses (four million inserts maximum). The count column indicates the number of samples, where one sample is a single thread's results from one run. The mean, std, and min columns represent the mean, standard deviation, and minimal value. The three table sections represent groupings according to the three parameters: retries per slot, presence or absence of stash, and bucket selection strategy. 60

5 Summaries of number of successful inserts from the second round of experiments (max, 25%, 50%, 75%). 60

6 Summaries of number of successful inserts from the second round of experiments (1%, 90%, 99%). 61

7 Summaries of successful inserts from from the 4 million max insert experiments (count, mean, std, min), grouped by every possible feature combination. The summaries are grouped by the configuration of the three design parameters, where for example "3,STASH,LL" means 3 retries per slot, with a stash and the least loaded bucket strategy. "2,NO_STASH,R" means 2 retries per slot, no stash and uniform random bucket select strategy. 62

8 Summaries of successful inserts from from the third round of experiments. These are the same experiments as the second round (high throughput workload with 4 million inserts max per thread), but with the target occupancy increased from 97% ($\frac{31}{32}$) to 98% ($\frac{63}{64}$) to induce failures in the 3,STASH,LL configuration. Results are presented as before (count, mean, std, min, grouped by every possible feature combination). The results indicate that the 3,STASH,LL starts to fail at 98% occupancy, with tests failing on average after 1,766,892 inserts (by comparison, all test runs completed all 4 million inserts in the second round of tests). 63

9 Sorted mean inserts per second from a high throughput workload with table capacity of 64 million and 8 million inserts per thread. In each experiment, 8 threads are launched at the same time and each is allowed to insert 8 million sequential integers, with each thread taking its integers from its own range disjoint from the other threads. Configurations represent presence or absence of stash (Y/N), target occupancy (the occupancy at which threads start deleting after every insert), and the number of buckets per zone. For example, "Y,3/4,16" means that the configuration had a stash, the target occupancy was 75%, and the lock granularity was sixteen buckets per zone (meaning that each lock is associated with sixteen buckets). Inserts per second are aggregated, meaning that the total number of inserts completed by all threads is divided by the elapsed time from start of threads to completion of all threads. All results were executed on a laptop equipped with a 8-core Intel Core i7-6700HQ CPU operating at 2.60GHz. 64

2 Introduction

Hashing is a broad term that refers to a variety of algorithms, techniques and data structures that involve the use of hash functions, and cuckoo hashing is a hashing algorithm which has grown to include a number of variations, including parallel versions.

We begin with an overview of the history and concepts of hashing that are relevant to our main topic of parallel cuckoo hashing. This overview includes definitions of hash table, hash function, and universal classes of hash functions, chaining, open addressing and linear probing. This discussion provides the context and background for a discussion of cuckoo hashing, which follows. We then briefly discuss the moti-

vation for investigating parallel cuckoo hashing, and discuss related work in parallel cuckoo hashing.

We then give a design of a parallel cuckoo hash table, and discuss some initial results from experiments run using this implementation. Finally, we conclude with a summary of findings and suggestions for future work. Table 2 on page 10 contains a reference for notation used in the thesis.

Expression	Meaning
$\ln x$	The natural logarithm of x
$\log x$	The logarithm base 10 of x
$\lg x$	The logarithm base 2 of x
$\log_a x$	The logarithm base a of x
$[n]$	The set $\{0, 1, 2, \ldots, n-1\}$ for some nonnegative integer n
$\mathcal{P}(S)$	The set of all possible subsets of the set S, including the empty set
$\lvert S \rvert$	The cardinality (number of elements) in the set S
$[a, b]$	$\{x : a \leq x \leq b\}$
$\mathbb{R}$	The real numbers

Table 1: Some common notation used in the thesis.

2.1 Hash functions and hash tables

All hashing involves the use of hash functions in some way, so we first define a hash function.

Definition 2.1 (Hash function)**.** Let $\mathcal{U}$ and $\mathcal{V}$ be two nonempty sets with $u = |\mathcal{U}|$ and $v = |\mathcal{V}|$. A hash function $h : \mathcal{U} \to \mathcal{V}$ is a mapping from $\mathcal{U}$ to $\mathcal{V}$. We often refer to $\mathcal{U}$ as the *universe* and $\mathcal{V}$ as the *index set*.

We write $[n]$ to indicate the set $\{0, 1, \ldots, n-1\}$ for some $n > 0$. A common hash function notation format that we will discuss is $h : \mathcal{U} \to [n]$ with $n > 0$.

Normally $|\mathcal{U}| \gg |\mathcal{V}|$ in practice, because the most common application of hash functions is to map a large universe (such as the set of all strings) into a smaller table (such as a contiguous array of memory locations in a computer system), but in the theory it is at times useful for $|\mathcal{U}| \leq |\mathcal{V}|$.

Hash functions find applications in several areas such as cryptography and data storage. The focus of this work is on their use in hash *tables*, which are data structures used for data storage and retrieval.

Definition 2.2 (Hash table)**.** A hash table is a data structure that supports three operations: *Lookup*(k), *Insert*(k) and *Delete*(k), where k is an element from $\mathcal{U}$ called the *key*. We define *Lookup*(k) to take key k and return *true* if k is present in the table, *false* otherwise. *Insert*(k) stores k in the table if it is not already present in the table. *Delete*(k) removes k from the table if present.

For simplicity of exposition, we only define operations of insert, lookup, and delete of keys here in order to focus on the core hashing algorithm. In practice, a hash table implementation usually associates user-provided satellite data with each key. Programming language implementations commonly have hash table implementations as part of the standard library (such as Java) or as part of the core langauge (such as Python).

The purpose of the hash function in a hash table is to map the large universe of keys $\mathcal{U}$ into a smaller set of indexes $\mathcal{V}$ whose elements identify locations of storage cells for keys. As a result, the expected time for all operations is $O(1)$. We often refer to the collection of one or more storage cells indicated by a particular hash value as a *bucket*, and the hash value itself as a *bucket address*.

2.2 Collision resolution

As mentioned above, normally $|\mathcal{U}| \gg |\mathcal{V}|$ in practice. Therefore a fundamental problem in hash tables is dealing with *collisions*, which we define here. Our definition requires some basic definitions from probability theory: discrete sample space, discrete random variable, and indicator random variable.

Definition 2.3 (Discrete sample space)**.** A *discrete sample space* is the set Ω of all

possible outcomes of a particular experiment, where $|\Omega|$ is finite. The sample space is associated with a function $Pr : \mathcal{P}(\Omega) \to [0, 1]$, such that $\Pr(\Omega) = 1$. We often refer to an arbitrary subset $E \subseteq \Omega$ as an *event*.

Definition 2.4 (Discrete random variable)**.** A *discrete random variable* is a function $X : \Omega \to \mathbb{R}$ that maps the events of a discrete sample space Ω to a real number.

Definition 2.5 (Indicator random variable)**.** An *indicator random variable* is a discrete random variable whose range is $\{0, 1\}$.

These definitions are similar to those of Bertsekas and Tsitsiklis (2008) and Mitzenmacher and Upfal (2017), adapted to the purposes of this thesis.

Definition 2.6 (Collision)**.** Let $\mathcal{U}$ and $\mathcal{V}$ be two sets, $h : \mathcal{U} \to \mathcal{V}$ be a hash function, and $x, y \in \mathcal{U}$.

$$\delta_h(x, y) = \begin{cases} 1 & \text{if } x \neq y \text{ and } h(x) = h(y) \\ 0 & \text{otherwise.} \end{cases}$$

For a given $x, y \in \mathcal{U}$, we say that x and y *collide* under h if $\delta_h(x, y) = 1$.

Our definition of collision very closely that of Carter and Wegman (1979), whose work we will discuss in detail further on. The purpose of giving definition 2.5 at this point in the discussion is to highlight the fact that collision can be thought of as an indicator random variable on the discrete sample space of pairs of keys $(x, y) \in \mathcal{U}^2$, which can help with the intuition around the role that probability theory plays in the theory of hashing. Hash functions can also be viewed as discrete random variables, as we will discuss below.

The idea of hashing originated with a 1953 IBM memorandum by H. P. Luhn, and independently about the same time by Gene M. Amdahl, Elaine M. Boehme, N. Rochester and Arthur L. Samuel (Knuth, 1998, p. 547). Both approaches used a hash function to compute a storage location from a key, but differed in the ways that

they resolved collisions. As we saw formally in definition 2.6, a collision occurs when two different keys are mapped to a single storage location by a hash function. This raises the problem of how to associate multiple items with a single address. The way that a hash table handles this problem is known as collision resolution.

Amdahl originated the idea of open addressing with linear probing to resolve collisions, whereas Luhn originated the idea of chaining (Knuth, 1998, p. 547). Open addressing is a collision resolution strategy whereby instead of associating an auxiliary structure such as a linked list with each bucket so that multiple keys can be stored in a bucket, each bucket addres defines a sequence of candidate locations in the hash table where the key may be stored. This sequence of candidate locations is known as a probe sequence. Open addressing was named in 1957 by W. W. Peterson (Knuth, 1998, p. 525). A simple open addressing scheme is linear probing described in Knuth (1998, p. 526), where the probe sequence is

$$h(k), h(k) - 1, \ldots, 0, v - 1, v - 2, \ldots, h(k) + 1.$$

Chaining is a collision resolution strategy whereby each bucket is associated with a linked list which is searched linearly during all hash operations, and which allows more than one item to be stored in a bucket.

2.3 Basic framework for hashing

In the literature of hash algorithms and hash functions, it can be helpful to think in terms of a small set of conceptual categories. Most of these terms either appear in or are closely related to the discussion in the "Concrete Schemes" section of (Pătraşcu & Thorup, 2015).

One category is specific implementation families of hash functions. Examples of hash function families are multiply-shift, polynomial, linear, or tabulation-based.

Hash function families correspond to Pătraşcu and Thorup (2015)'s "concrete schemes."

Another category is theoretical groups of hash functions that satisfy a combinatorial property such as universality. These are often called *classes*, as in "universal classes of hash functions."

Another category is abstract mathematical or probabilistic properties of hash functions or families of hash functions. 2-universal and strongly k-universal are two such properties which we define below. A particular concrete implementation may or may not possess specific mathematical properties.

Finally, applications are particular algorithms, patterns and techniques that *use* hash functions. Examples of applications are hash tables in general, linear probing, minwise independence, cuckoo hashing, and hashing with chaining. Some concrete implementation families of hash functions are appropriate for some applications but fail in other applications. This is true for cuckoo hashing, as we will see below.

2.4 Basic challenge of hash functions

We discuss the basic engineering challenge of hash function design and implementation. To do so, we require some more definitions from basic probability theory. We introduce the definitions of probability mass function and discrete uniform random variable, which follow closely those of (Bertsekas & Tsitsiklis, 2008, p. 72,74,89).

Definition 2.7 (Probability mass function)**.** A discrete random variable X has an associated function $p_X : \mathbb{R} \to [0, 1]$ such that $p_X(x) = \Pr(X = x)$. This function is called the *probability mass function (PMF)* of X.

It follows from this and the definition of probability space that $\sum_x p_X(x) = 1$.

Definition 2.8 (Discrete uniform random variable)**.** A discrete uniform random variable X is a discrete random variable that is uniformly distributed across a range of contiguous integer values. That is to say, for some integers a and b such that $a < b$,

the probability mass function for X is given by

$$p_X(k) = \begin{cases} \frac{1}{b-a+1} & \text{if } k \text{ is an integer in } [a, b], \\ 0 & \text{otherwise.} \end{cases}$$

We will also need the definition of mutual independence of discrete random variables, which follows closely that of Mitzenmacher and Upfal (2017, p. 24).

Definition 2.9 (Mutual independence of discrete random variables)**.** The discrete random variables $X_1, X_2, \ldots, X_k$ are *mutually independent* if for any subset $I \subseteq [1, k]$ and any values $x_i, i \in I$,

$$\Pr\big(\cap_{i \in I}(X_i = x_i)\big) = \prod_{i \in I} \Pr(X_i = x_i).$$

For the purposes of this work, we define the ideal hash function, which is sometimes called a "truly random" hash function (Thorup, 2015, p. 1).

Definition 2.10 (Truly random hash function)**.** A truly random hash function $h : \mathcal{U} \to [v]$ is a vector of mutually independent discrete random variables $(X_1, X_2, \ldots, X_{|\mathcal{U}|})$, where each X_k takes on each value in $[v]$ with probability $\frac{1}{v}$. Thus a truly random hash function is a $|\mathcal{U}|$-dimensional random variable.

Much of the early analysis of hash algorithms avoids the problem of hash function specification by assuming truly random hash functions. Since a hash function is a mapping from $\mathcal{U}$ to $\mathcal{V}$, it can be viewed as a vector $[v]^u$, so the assumption of truly random hash functions is equivalent to the assumption that a given hash function has been uniformly chosen from this vector space $[v]^u$. Specifying such a function would require $u \lg v$ random bits (Pătraşcu & Thorup, 2015, p. 1). To hash 64-bit keys into a 32-bit table, this would require 512 exabytes of information – one half of a billion terabytes – to specify a hash function.

In most cases in the application area of hash tables, in order for a hash function implementation to be practical, it must be fast, and it must not require too much memory to specify or execute. This motivates an engineering solution to the problem of ideal truly random hash functions that require 512 exabytes of information to specify. The fundamental engineering problem for hash functions is the tradeoff between the properties of a truly random hash function against computational and space complexity of hash function specification and computation.

2.5 Universal classes of hash functions

An important contribution to the study of hash functions was the definition by Carter and Wegman (1979) of *universal classes of hash functions*, which introduced a new way of classifying hash functions according to degrees and types of independence. The properties they discovered enabled the formation of standardized *classes* of hash functions from which individual hash functions could be randomly chosen by a program. This step allowed the authors to reason about performance of algorithms using functions from these classes without describing implementation details of the hash functions themselves.

As discussed in section 2.2 on page 11, Carter and Wegman (1979) formally defined the notion of *collision* of a pair of keys to be an indicator random variable on the probability space of pairs of elements from $\mathcal{U}$. They then used their definition of collision as the characteristic property in the key definition of their paper, which we will give shortly. First we make what may be an obvious observation in preparation for their definition.

Remark 2.1. Let $\mathcal{U}$ and $\mathcal{V}$ be two sets, let $\mathcal{H}$ be a set of hash functions $h : \mathcal{U} \to \mathcal{V}$, and let $x, y \in \mathcal{U}$. Then $\sum_{h \in \mathcal{H}} \delta_h(x, y) \leq |\mathcal{H}|$.

Proof. $\delta_h(x, y) \leq 1$, so $\sum_{h \in \mathcal{H}} \delta_h(x, y) \leq \sum_{h \in \mathcal{H}} 1 = |\mathcal{H}|$. □

Informally, if $\sum_{h \in \mathcal{H}} \delta_h(x, y) = |\mathcal{H}|$ for some $x, y \in \mathcal{U}$, then $\mathcal{H}$ is the worst possible hash function family from the perspective of collisions with x and y, because they collide under every single hash function in the family. We now give the key definition of Carter and Wegman (1979).

Definition 2.11 (2-universal)**.** Let $\mathcal{U}$ and $\mathcal{V}$ be two sets, and let $\mathcal{H}$ be a set of hash functions $h : \mathcal{U} \to \mathcal{V}$. We say that $\mathcal{H}$ is 2-universal if for all x, y in $\mathcal{U}$, $\sum_{h \in \mathcal{H}} \delta_h(x, y) \leq \frac{|\mathcal{H}|}{|\mathcal{V}|}$.

One way of thinking about this definition is as follows. Consider the worst possible hash function defined above for $x, y \in \mathcal{U}$. A hash family is 2-universal if, for all $x, y \in \mathcal{U}$, we divide that worst possible number of collisions by the size of $\mathcal{V}$. That is, *any* pair x, y is only permitted to collide in $\frac{1}{|\mathcal{V}|}$th of the functions in the family. This is the significance of the quantity $\frac{|\mathcal{H}|}{|\mathcal{V}|}$. This quantity allows for estimating the probability of collision if a hash function is sampled uniformly at random from the family. Therefore 2-universal can be thought of informally as a quality standard about collisions in a family of hash functions.

This definition assigns a name to the property of an upper bound on the number of collisions permitted for *any* pair of keys across an entire family of hash functions. This provides the basis for a theoretical framework that allows practitioners to prove that a particular implementation of a hash function family is 2-universal, and then take advantage of the rest of the theory built on that property.

There some variation in the names, definitions and terminology in the literature of hash function theory. For example, what Carter and Wegman (1979) define as *universal*$_2$ and what we define as 2-universal is defined by Thorup (2015) as simply "universal," with no adornment. This may be because the field of hash function theory is new compared to other branches of mathematics. In response to a question for clarification about these differences, Mikkel Thorup advised that "authors define

these things differently, so you have to read the definitions carefully in whatever paper you are reading."

Carter and Wegman (1979) then prove a result which shows that 2-universal is the best that we can do if $\mathcal{U}$ is much larger than $\mathcal{V}$, which is the normal case with hash functions. This is because for any family $\mathcal{H}$ of hash functions $h : \mathcal{U} \to \mathcal{V}$, there exists a pair $x, y \in \mathcal{U}$ such that the total number of collisions across all hash functions in $\mathcal{H}$ is bounded below, and for large $\mathcal{U}$ – the normal case – the bounds given by the definition of 2-universal are tight. We give the result here without the proof, which is provided in the original paper.

Proposition 2.1. Given any set $\mathcal{H}$ of hash functions $h : \mathcal{U} \to \mathcal{V}$, there exist $x, y \in \mathcal{U}$ such that

$$\sum_{h \in \mathcal{H}} \delta_h(x, y) \geq |\mathcal{H}| \left(\frac{1}{|\mathcal{V}|} - \frac{1}{|\mathcal{U}|} \right).$$

In the original paper, the inequality above is strict, but we relax it here to accomodate the case where $|\mathcal{U}| = |\mathcal{V}|$, where we have

$$\begin{aligned} \sum_{h \in \mathcal{H}} \delta_h(x, y) &\geq |\mathcal{H}| \left(\frac{1}{|\mathcal{V}|} - \frac{1}{|\mathcal{U}|} \right) \\ &\geq 0. \end{aligned}$$

To see why this inequality is not strict and may be zero, consider the hash family consisting of the $|\mathcal{V}|!$ bijective hash functions from $\mathcal{U}$ to $\mathcal{V}$ where $|\mathcal{U}| = |\mathcal{V}|$. None of these have any collisions, thus equality would hold.

Therefore if a family of hash functions is 2-universal, we have

$$\frac{|\mathcal{H}|}{|\mathcal{V}|} \geq \sum_{h \in \mathcal{H}} \delta_h(x, y) \geq |\mathcal{H}| \left(\frac{1}{|\mathcal{V}|} - \frac{1}{|\mathcal{U}|} \right),$$

where the left side of the inequality is due to the definition, and the right side of the

inequality is due to the lower bound result for pairwise collision. For $\mathcal{U} \gg \mathcal{V}$, which is the normal case, we have

$$\frac{|\mathcal{H}|}{|\mathcal{V}|} \geq \sum_{h \in \mathcal{H}} \delta_h(x, y) \geq |\mathcal{H}| \left(\frac{1}{|\mathcal{V}|} - \frac{1}{|\mathcal{U}|} \right) \approx \frac{|\mathcal{H}|}{|\mathcal{V}|}.$$

This inequality establishes a basic reasoning framework about pairwise collisions that continues to be used extensively.

It may seem counterintuitive to consider that this quantity increases with the number of hash functions in the family. But it starts to make sense when considering that the basic per-function unit of collision is given by the quantity $\frac{1}{|\mathcal{V}|}$, which is multiplied by the number of functions in the family: $\delta_h(x, y)$ is a quantity associated with a specific function, and $\sum_{h \in \mathcal{H}} \delta_h(x, y)$ is the sum over all functions in the family.

In a following paper, Wegman and Carter (1981) defined the concept of strong universality, which we give here in an updated form.

Definition 2.12 (Strongly universal)**.** Let $\mathcal{U}$ and $\mathcal{V}$ be two sets, and let $\mathcal{H}$ be a set of hash functions $h : \mathcal{U} \to \mathcal{V}$. We say that $\mathcal{H}$ is *strongly k-universal* if for any distinct $x_1, x_2, \ldots, x_k$ in $\mathcal{U}$, and any (not necessarily distinct) $y_1, y_2, \ldots, y_k$ in $\mathcal{V}$, and any h chosen uniformly at random from $\mathcal{H}$,

$$\Pr(h(x_1) = y_1 \cap h(x_2) = y_2 \cap \cdots \cap h(x_k) = y_k) = \frac{1}{|\mathcal{V}|^k}.$$

Strong universality implies uniformity, but universality does not. Kaser and Lemire (2012, p. 1) provide a simple proof of this fact. Consider the hash family $h : [0, 1) \to [0, 1)$, where h_1 is the identity function $h_1(x) = x$, and $h_2(x) = 0$. Then $\Pr(h_1(x) = h_2(x)) = \frac{1}{2}$, implying universality, but the family is not uniform because $\Pr(h(0) = 0) = 1$. Another way to think about this in terms of universality is that from the perspective of collisions, h_1 is the best possible hash function, whereas h_2

is the worst, and as a family they "balance each other out" and satisfy the definition of 2-universal. This family of hash functions is an example of a 2-universal family that would be terrible if used in an application, because half of the time, the randomly chosen function would be completely useless for hashing. k-independence is an equivalent definition to strongly k-universal that is often seen in papers about hash functions, for example in (Pătraşcu & Thorup, 2015).

2.6 Limits of applicability of universality

The paradigm of universal classes of hash functions could be viewed from a software engineering perspective as an *architectural* innovation, because it offers the promise of a *separation of concerns* or modularity: the implementation of specific concrete hash function families – in the sense discussed in section 2.3 – can be pursued separately from the problem of hash algorithms that use the hash functions. As long as the hash functions satisfy the requirements of the hash algorithm in terms of classes of hash functions, the larger algorithm will "work."

While 2-universal and strongly k-universal are important properties, there may be limits to how far independence and universality can be applied, and there may be cases where specific hash function families need to be analyzed for suitability in a particular hashing application (Pătraşcu & Thorup, 2015, p. 4). Different applications require different levels of independence. For hashing with chaining, 2-universality suffices (Pătraşcu & Thorup, 2015). For linear probing, the multiplicative and linear hash function families (which are strongly 2-universal) fail badly, and a minimum of strongly 5-universal is required (Pătraşcu & Thorup, 2015), (A. Pagh, Pagh, & Ruzic, 2009).

2.7 Balanced allocations

In 1988 Dietzfelbinger et al. (1988) described an algorithm with $O(1)$ worst-case lookup time and $O(1)$ expected amortized time for insertion and deletion (Dietzfelbinger et al., 1988), (Dietzfelbinger et al., 1994).

In 1995, Azar, Broder, Karlin, and Upfal (1999) discovered and named an important concept called *Balanced Allocations* which improves exponentially on a classical result of probability theory (Azar et al., 1999). Their discovery was a variation on the well-studied random process where n balls are sequentially placed into n bins one by one, with each box chosen uniformly at random. At the end of this process, the fullest box contains $\frac{(1+o(1))\ln n}{\ln\ln n}$ with high probability (approaching 1 for n sufficiently large). Their contribution introduced a variation of this process that results in an exponential improvement by changing the way each box is chosen. At each step, instead of choosing the next box uniformly at random, two boxes are chosen uniformly at random, and the ball is placed in the least loaded of the two boxes. With this modified process, the fullest box contains $\frac{\ln\ln n}{\ln 2} + O(1)$ balls.

Azar et al. (1999) also showed that if 2 boxes are chosen uniformly at random at each step, then the strategy of choosing the least loaded bin is the *best* strategy for minimizing the least loaded box – that is to say, if any other method is used to choose between the two boxes, then the fullest box will contain at least $\frac{\ln\ln n}{\ln 2} + O(1)$ balls.

The motivating application for their result was hashing, but their contribution is a basic result with wide application. In their paper, Azar et al. (1999) describe a hashing technique called *2-way chaining* where on insertion, the least loaded of two possible entries is chosen. Subsequent relevant work includes cuckoo hashing (R. Pagh & Rodler, 2001) and d-left hashing (Mitzenmacher, Richa, & Sitarman, 2001).

2.8 Cuckoo hashing

In 2004, Pagh and Rodler introduced an algorithm called *cuckoo hashing*, which achieved the same performance guarantees as (Dietzfelbinger et al., 1994), with a simplified design (R. Pagh & Rodler, 2001).

In (Kirsch, Mitzenmacher, & Wieder, 2009), the authors showed that the probability of failed inserts can be reduced by adding a *stash*, which is a small, constant-sized (≤ 4 in the original paper) number of extra storage locations designated for items for which inserts fail. The authors showed theoretically the benefits of using a stash, and showed experimentally that the stash, while not needed for approximately 99.3% of cases, successfully addressed the remaining cases. In particular, the largest stash required to successfully insert in all one million experiments was four. In addition, the authors considered the case of multiple slots per bucket ("blocked cuckoo hashing"), and found the stash to be useful in this case as well. We implemented and observed behavior with and without a stash in section 7 below. Blocked cuckoo hashing is described below in section 5.3.1.

The choice of hash functions is important in a cuckoo hashing implementation. Dietzfelbinger and Schellbach (2009) showed theoretically and experimentally that certain classes of hash functions result in high failure probability in cuckoo hash tables. Aumüller, Dietzfelbinger, and Woelfel (2012) describe a family of hash functions that use a combination of independent hash families along with lookup tables to augment the hash values. Aumüller, Dietzfelbinger, and Woelfel (2016) give a class of hash functions that "can replace other, less efficient constructions in cuckoo hashing (with and without stash)" among other purposes.

In cuckoo hashing as described in R. Pagh and Rodler (2001)'s original paper, a hash table is composed of two subtables T_1 and T_2, each with its own hash function h_1 and h_2 respectively, so that a given key has two possible locations. Pagh and Rodler's original algorithm is given in listing 1 on page 23.

Listing 1: The `insert()` operation defined in the original cuckoo hashing paper.

```
1  procedure insert(x)
2    if lookup(x) then return
3    loop MaxLoop times
4      if T1[h1(x)] is empty then { T1[h1(x)] = x; return }
5      swap(x, T1[h1(x)])
6      if T2[h2(x)] is empty then { T2[h2 (x)] = x; return }
7      swap(x, T2[h2(x)])
8    end loop
9    rehash(); insert(x)
10 end
```

The practical value of cuckoo hashing can be demonstrated experimentally. Figure 1 on page 24 summarizes results of six experiments where test code inserted random keys into cuckoo hash tables with a capacity of 16,384 buckets.[1] In these early tests, the hash functions were implemented as simple maps built with a pseudorandom number generator for ease of experimentation for the purpose of exploring the effects of number of hash functions on the number of inserts before failure. The bucket capacity of 16,384 buckets was an arbitrary choice to explore this behavior. The pseudocode for the insert function used in these experiments is given in listing 2 on page 25. Note that in these experiments, the d hash functions used a single subtable instead of one subtable per hash function.

In each experiment, random keys were inserted into a cuckoo hash table until an insert was not successful within 100 ejections. This number was chosen as an arbitrary invariant quantity to explore the effect of varying the number of hash functions on table occupancy, determined by number of inserts before failure. Each experiment was run 1000 times. The first experiment was run with one hash function (one choice), the second with two hash functions (two choices), and so on up to six hash functions. Note that hashing with only one hash function is not really cuckoo hashing, but serves as a useful point for comparison.

The results plot the number of insertions before failure for the six different hash

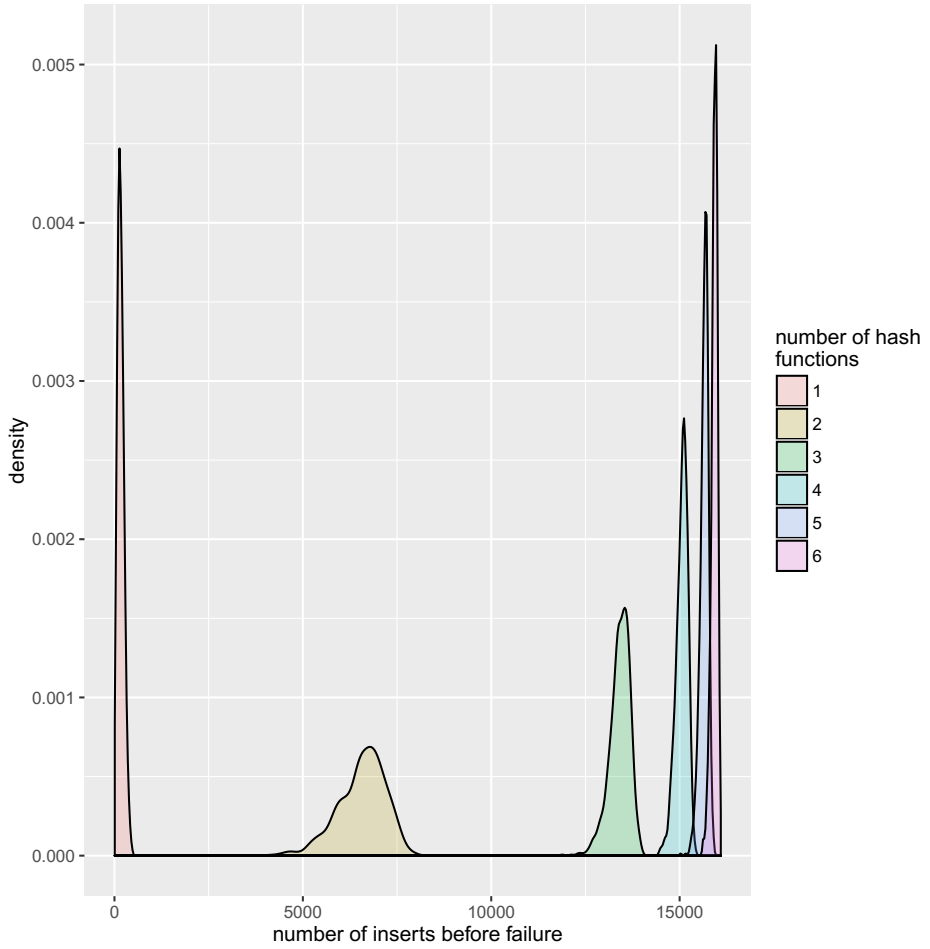

Figure 1: Density plots for the number of inserts before a failure with $1 \leq n \leq 6$ hash functions. All experiments were run on a hash table with 16,384 buckets, and one slot per bucket. An insert is considered failed if it does not succeed within 100 ejections.

function counts. A significant gain in memory utilization is realized when we go from one hash function to two. As we add hash functions, we get increasingly predictable hash table capacity, but the gains diminish with each additional function.

These experiments simplify implementation by avoiding the problem of selecting and using "real" (practical) hash functions, and instead use a relatively small key universe with tables mapping keys to pseudorandomly generated hash values.

Listing 2: The `insert()` operation used in early experiments.

```
1  procedure insert(x)
2    if lookup(x) then return
3    loop MaxLoop times
4      // Randomly shuffle the hash functions
5      random_shuffle(h)
6      for i in [0..d-1]
7        if T[h[i](x)] is empty then
8          T[h[i](x)] = x
9          return
10       end if
11     end for
12     swap(x, T[h[d-1](x)])
13   end loop
14   return FAIL
15 end
```

2.9 Motivation for investigation of parallel cuckoo hashing

All of the hashing techniques mentioned above are sequential (non-parallel). But at the time that Azar and Pagh and others were developing the area of balanced allocations and more efficient and predictable hashing, a shift in computer architecture was taking place. With the commoditization of multicore processors that began in the mid-2000's, symmetric multiprocessing systems are now the norm. Desktops and laptops are multicore, as are smart phones, and the number of cores is growing. The machine used to write this thesis (an Intel Core i7) has 8 cores. The technology Company Phytium announced a 64-core ARM processor on August 23, 2016.

The proliferation of multicore processor architectures motivate the development of parallel algorithms. Mitzenmacher suggests the design and analysis of efficient parallel cuckoo hash tables as a key open question about cuckoo hashing (Mitzenmacher, 2009). Subsequent work has been done in this area. Some of this work relies on specialized hardware features or platforms such as General Purpose Graphics Processing

Units (GPGPUs) and transactional memory.

The purpose of this thesis is to investigate and implement parallel cuckoo hashing algorithms that run on standard commodity processors such as x86 or ARM and do not require specialized hardware support. A secondary purpose is to select and use "real" (practical) hash functions in this implementation.

3 Related Work in concurrent cuckoo hashing

3.1 Optimistic cuckoo hashing (Fan et al)

Fan et al contributed *optimistic cuckoo hashing*, a design that supports multiple readers and a single writer (Fan, Anderson, & Kaminsky, 2013). They have continued this work in the form of a standalone library available at their GitHub repo to support multiple readers and multiple writers (Goyal, Fan, Li, & Andersen, 2013).

3.1.1 Tag-based Lookup/Insert

One of Fan et al's contributions is called *tag-based lookup/insert*. The goal of this technique is to reduce cache misses and support variable-length keys. Under tag-based lookup/insert, each slot contains a 1-byte *tag*, and a pointer to a *KV object*, which contains the key, the value, and the bucket metadata. The *tag* can be thought of as an 8-bit hash value for the key. Cache misses are reduced because the pointer to the *KV object* only needs to be followed if the tags match. For example, when searching for a key, if none of the tags match, none of the *KV object* locations need to be fetched. If tags match, then *KV object* needs to be dereferenced to determine if the keys actually match.

3.2 Li et al

Li et al give an implementation that supports multiple readers and multiple writers (Li, Andersen, Kaminsky, & Freedman, 2015). Their work describes algorithmic improvements to Fan et al as well as the use of hardware transactional memory (HTM) from Intel.

To reduce the time spent in critical sections, Li et al lock after discovering a cuckoo path. A *cuckoo path* is defined as the sequence of displaced keys in an Insert operation. We will define cuckoo path formally below. A writer first checks for an empty slot, and if one is discovered, locks the slot and inserts the item. If that step fails, then the writer repeatedly performs a search for a cuckoo path with an available slot, and on success, locks the whole table and attempts the insert on the found path. If any of the searches fail to find a cuckoo path to execute, the insert fails. They do not discuss failure modes of locking beyond mentioning the low probability of a livelock.

Li et al also use a version counter to avoid the need for read locks. Readers read the version counter, search for their element, read the version counter again and then check that it has not changed.

They also use breadth-first search to find a cuckoo path before locking. This differs from previous work ("basic" cuckoo hashing), which uses a greedy search. Under the greedy search algorithm, items are ejected until an empty slot is found.

4 Deadlock in naive parallel cuckoo hashing

A basic problem with implementing parallel cuckoo hashing can be illustrated with an example. Under this example we modify the original cuckoo hashing algorithm to use a single hash table but two hash functions. In order to insert, a thread must acquire the locks of all buckets in which an item could be hashed. It is easy enough to define

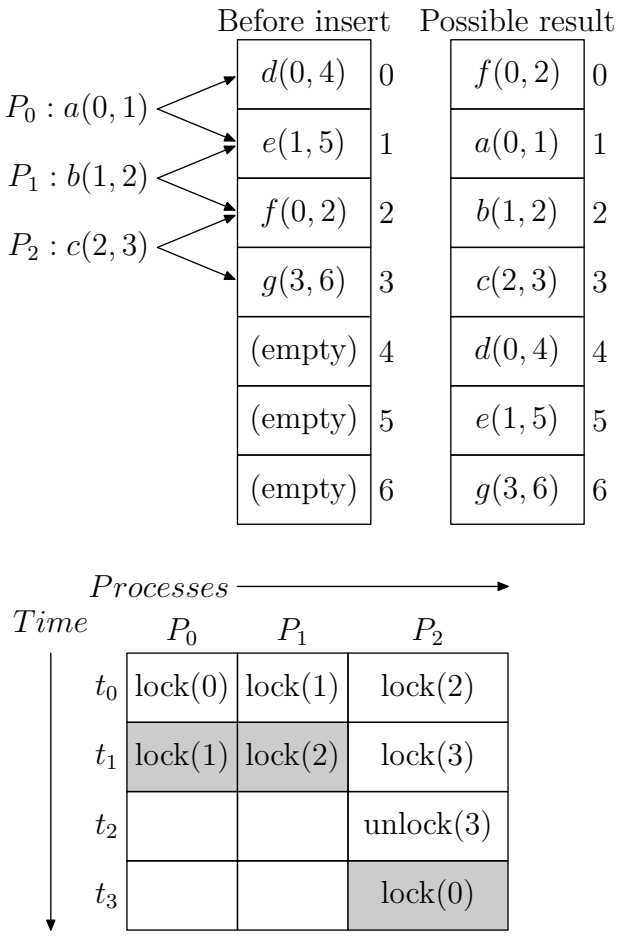

Figure 2: Deadlock example with a naive implementation of parallel cuckoo hashing using per-bucket locks and one slot per bucket. Columns are threads of execution, and rows are discrete time steps. Gray items are deadlocked operations. The notation $a(0,1)$ means key a can reside in bucket 0 or bucket 1.

a consistent ordering of locks to prevent this locking step from causing deadlock. For example, if storage locations are elements of an array, and each storage location is associated with a lock, then the storage location's index may be used as the sort criterion for a consistent ordering of the locks.

The problem comes when an item is ejected as part of the insert, resulting in an out-of-order lock acquisition sequence. Because ejection of existing elements is the fundamental distinguishing feature of cuckoo hashing, a solution to this problem is required.

Figure 2 on page 28 illustrates this problem. In the example, a cuckoo hashing table has two hash functions, say h_1 and h_2. Threads P_0, P_1 and P_2 attempt to insert an element at the same time. Each element has two possible buckets, one for each hash function (we ignore the case where $h_1(x) = h_2(x)$ here). With elements d, e, f and g already inserted, none of the candidate slots of the elements to be inserted are empty, so all of these insert operations will require an existing element to be ejected. In the illustration, the notation $e(1, 5)$ means that element e can reside either in location 1 or location 5. This is a notationally convenient way of indicating that $h_1(e) = 1$ and $h_2(e) = 5$. The notation $P_0 : e(0, 1)$ indicates that process 0 is attempting to perform an insert of element e, for which $h_1(e) = 0$ and $h_2(e) = 1$. The table can accomodate this workload, with one possible allocation of elements shown in the possible configuration under "Possible result".

The 4x3 matrix below the hash table illustrations indicates a potential sequencing of execution of the three threads (representing the three columns, with time ticks moving downwards) leading to deadlock. Rows indicate time steps, columns indicate threads of execution, and shaded slots indicate deadlocked lock attempts.

At step t_0, all three threads acquire the first bucket of their elements' bucket pair. At step t_1, all three threads attempt to lock the second bucket, but only P_2 succeeds. P_2 has randomly chosen cell 2 from candidate cells 2 and 3, and proceeds to eject occupant f from 2 by replacing f with c, and unlocks 3 because it will not eject g (the contents of 3). P_2 holds lock 2 and attempts to lock 0 to insert the ejected element f. Deadlock results because P_0 holds lock 0, and is part of a deadlock cycle leading back to P_2's attempt to acquire 0.

P_2 holds 2 while acquiring 0 to avoid data corruption. For example, if P_2 were to release 2 before acquiring 0, then another thread could search for f and fail to find it.

The key idea of this example is to show that if locks are acquired one by one as

ejections occur through the table, deadlock results.

5 Hash table design

The following section describes the design of a concurrent multi-reader, multi-writer cuckoo hash table implementation. Table 2 on page 30 shows some common parameters in the construction of the hash table structures. These will be discussed in detail in their relevant sections.

Expression	Meaning
d	The number of hash functions, hence the number of component tables
m	The number of bucket zones per component table
n	The number of buckets per bucket zone, determines lock granularity
q	The number of storage slots per bucket

Table 2: Common top-level hash table parameters. For all of the experiments in this thesis, two hash functions were used ($d = 2$), along with 32 buckets per bucket zone ($n = 32$), and 32 slots per bucket ($q = 32$). The number of bucket zones per component table (m) is a function of these parameters along with the required storage capacity of the structure, which is passed as a parameter to the table creation code.

5.1 Terms

A component hash table T is a vector of storage elements used to store keys and associated satellite data. A top-level hash table $\mathbf{T} = (T_0, T_1, \ldots, T_{d-1})$ is a vector of d such tables. A bucket is identified by its offset within a hash table. We define $\mathcal{U} = [2^w]$ to be the universe of keys for some integer w, and $\mathcal{V} = [2^l]$ to be the set of bucket locations for some integer l. Since we will have many occasions to make reference to the cardinality of these sets, we denote their sizes as $u = 2^w = |\mathcal{U}|$ and $v = 2^l = |\mathcal{V}|$. Normally $u \gg v$. We refer to a particular bucket within a hash table as $T[i]$ for some integer $i \in [v]$. The set of possible locations for a given key k is determined by the set $\mathbf{h}(k) = \{h_0(k), \ldots, h_{d-1}(k)\}$, where d is the number of hash functions and a hash

function $h_i : \mathcal{U} \to [v]$ for $i \in [d]$ pseudorandomly maps keys to locations. For a given key k, we denote the set of possible buckets $\{T_0[h_0(k)], \ldots, T_{d-1}[h_{d-1}(k)]\}$ where k could be located as $\mathbf{T}(k)$. We write $T_i(k)$ as shorthand for $T_i[h_i(k)]$ for some $i \in [d]$. A key k may reside in at most one location in the hash table. Each bucket has q slots. Every slot is either empty or contains a key. Therefore at any given time, a bucket contains 0 to q keys, and the top-level hash table holds 0 to dvq keys.

5.1.1 Locks

For the purposes of controlling concurrent access to the items stored in the top-level hash table, every component table T_i has an associated set of locks $\mathbf{L}_i$, and each bucket $T_i[j]$ has an associated lock $L_i[j]$ for $i \in [d]$, $j \in [v]$. For any $k \in [v]$, let $\mathbf{L}(k) = \{L_0[h_1(k)], \ldots, L_{d-1}[h_{d-1}(k)]\}$ denote the set of write locks for $\mathbf{T}(k)$, which we call the lockset for k.

5.1.2 Lock ordering

To avoid deadlock, we define an ordering for all locks $L_i[j]$ for $i \in [d]$, $j \in [v]$. For $s, t \in [d]$, $a, b \in [v]$, $L_s[a] < L_t[b]$ if $s < t$, or if $s = t$ and $a < b$. Let $L_i(k)$ for $i \in [d]$ denote the ith lock in the list of elements from $\mathbf{L}(k)$ so ordered. We say that we acquire $\mathbf{L}(k)$ as a convenient way of saying that we acquire $L_1(k), L_2(k), \ldots, L_d(k)$ in that order. We will discuss lock ordering in more detail below.

5.1.3 Thread

We use the term thread to refer to an independent thread of execution, independent of any particular package or implementation. We performed all experiments described in this thesis with an implementation in C using the pthreads library.

5.2 Operations

We define the operations *Lookup*(k), *Insert*(k) and *Delete*(k). We do not create a separate operation for update; if an element already exists in the table, *Insert*(k) will update it in place.

Lookup(k) takes a key k and returns *true* if k is present in the table, *false* otherwise. *Insert*(k) stores k in the table. If k is already present in the table, the operation is successful. *Delete*(k) removes k from the table.

For simplicity of exposition, we only define operations of insert, lookup, and delete of keys here in order to focus on the core hashing algorithm. The implementation stores user-provided satellite data with each key, so that the hash table implements a map of keys to objects rather than a set of keys.

5.3 Memory layout

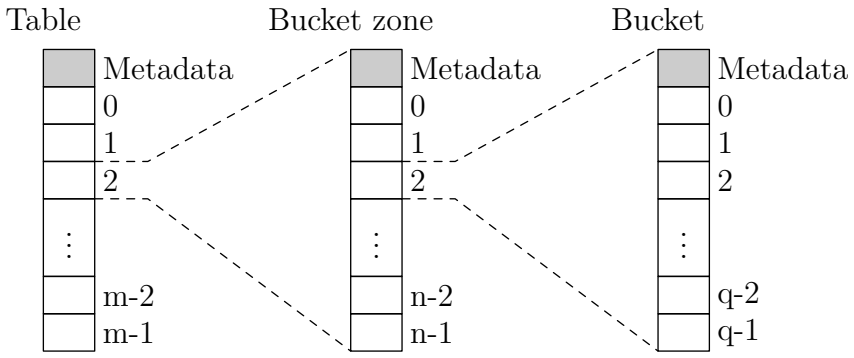

Figure 3: Memory layout for the data store of the hash table.

Figure 3 on page 32 shows the memory layout and hierarchical organization of the storage into component hash tables, bucket zones, buckets and slots. The top-level hash table structure has one component hash table per hash function, with d being the number of hash functions. Each component hash table contains m bucket zones, each of which contains n buckets, each of which contains q slots. Each level of the

hierarchy has a metadata element that is used to store information relevant to the level. The purpose of the bucket zone is to provide a mechanism for experimenting with lock granularity – because there is one lock per bucket zone, more buckets per zone results in a lower lock granularity. Each slot can hold one element, so the slot is the primitive level of element storage. The bucket metadata includes bucket-level metadata such as the slot valid vector. It follows from all of this that a top-level table can store a theoretical maximum of $dmnq$ entries. m and n are powers of two, so that hash function values can be partitioned into bucket zone and bucket offsets with shift and mask, simplifying addressing.

The concept of bucket zone was omitted from section 5.1 because it is a mechanism for experimenting with lock granularity, which would introduce unnecessary complexity to the formal model of the hash table. Grouping buckets into zones allows for breaking the one-to-one correspondence between buckets and locks, so that lock granularity can be adjusted from per-bucket (most granular/least coarse) up to per-table (least granular/most coarse) for the purpose of experimentation.

5.3.1 Multiple slots per bucket (blocked cuckoo hashing)

The hash table design given here allows for more than one slot per bucket, which is a generalization of cuckoo hashing described as "blocked cuckoo hashing" by (Dietzfelbinger & Weidling, 2007, p. 48). This differs from R. Pagh and Rodler (2001)'s original cuckoo hashing, where each bucket contains one slot. Instead, under this generalization, each bucket contains q slots. This approach was also used in Li et al. (2015). In all of the experiments performed in this thesis, there were 32 slots per bucket ($q = 32$).

5.3.2 Lock ordering

We discussed lock ordering more formally in section 5.1.2. Informally, the ordering of locks may be thought of from the perspective of the top level hash table as a contiguous region of memory. This memory is hierarchically organized into tables, bucket zones, buckets, and slots. Locks appear at the bucket zone level. Every bucket is associated with one and only one lock, but a lock is associated with one or more buckets, depending on the granularity of the hash table as determined by the parameter n. Because bucket zones have a "natural" ordering provided by their physical location in memory (their memory address), we use that ordering as the order that locks are acquired by a thread wishing to acquire any subset of locks.

For example, suppose a thread needs to acquire locks for the following buckets:

1. Table 1, zone 3, bucket 2
2. Table 1, zone 3, bucket 1
3. Table 1, zone 2, bucket 1
4. Table 1, zone 5, bucket 2
5. Table 0, zone 3, bucket 1
6. Table 0, zone 3, bucket 2

In this case, the lock acquisitions would proceed according to the following order: item 5, item 3, item 1, and finally item 4. Because items 1 and 2 belong to the same bucket zone, they have the same lock, so the corresponding lock would only be acquired once. The same applies to items 5 and 6.

When a thread locks a set of locks, the locks are sorted according to this order, and then acquired, with duplicates skipped as indicated in this example.

5.4 Choice of Hash Function

It has been shown that some hash functions do not perform well in cuckoo hash tables. This sensitivity of cuckoo hashing to hash function was observed experimentally in (R. Pagh & Rodler, 2001), and explained formally in (Dietzfelbinger & Schellbach, 2009).

Dietzfelbinger and Schellbach (2009) showed that under certain ratios of key set size to universe size, the *multiplicative* and *linear* families of hash functions do not perform well in cuckoo hashing due to the structure of the insert process known as the *cuckoo graph*. For the experiments described in this work, we used a hash function based on (Aumüller et al., 2012).

5.5 Deadlock prevention

Definition 5.1 (Cuckoo path)**.** A *cuckoo path* is a sequence of one or more 4-tuples, each of which has the form

$$(k, \mathbf{h}(k), \mathbf{v}(k), c).$$

Each tuple consists of a key k to be stored, a vector $\mathbf{h}(k) = (h_1(k), h_2(k), \ldots, h_d(k))$ of hash function values for key k that indicate the candidate bucket addresses for that element, a vector $\mathbf{v}(k) = (v(h_1(k)), v(h_2(k)), \ldots, v(h_d(k)))$ indicating the version information of each of the buckets recorded at the time the path entry was created, and the choice $c \in [1, d]$ representing the storage location chosen for e among the candidate bucket addresses $\mathbf{h}(k)$ of the step.

The first tuple of the sequence represents the item to be inserted into the table at step 1, and the last tuple of the sequence represents the last ejected item. For $i > 1$, the ith tuple of the path contains the item to be ejected at step $i - 1$ of the insert operation.

A cuckoo path is constructed during the first phase of the insert operation. On

successful construction, the cuckoo path structure is then used to provide the list of locks that need to be acquired, as well as all of the information needed to carry out the table modifications of the insert operation.

To avoid the deadlock problem discussed above and illustrated in figure 2 on page 28, we use a two-phase approach during insert. This consists of a *dry walk* to idenitfy a cuckoo path followed by a *lock walk* where the locks identified in the *dry walk* are acquired. The *dry walk* solves the problem of not knowing the entire cuckoo path as locks are acquired. By identifying the cuckoo path before modifications are made – which may be thought of as constructing a transaction plan or script – and then acquiring all locks in a consistent order during *lock walk*, deadlock is avoided. This approach is similar to the depth-first structure of original cuckoo hashing. We describe this approach in detail in section 6. We show that this approach avoids deadlock, but does not guarantee reader starvation-free operation.

5.6 Cache-friendly discipline

The design seeks to follow two principles of locality.

Write locality principle. Minimize the number of cache lines required for the writes involved in a destructive operation. If writing into a location, and other subsequent writes are anticipated to the same location, do them as closely together in time as possible.

Read locality principle. Design the layout to minimize the number of cache misses generated by fetches of the information required for an operation.

We locate the metadata containing bucket zone, bucket and slot information in the structures themselves to follow both the read and write locality principles, because reading the metadata associated with a bucket increases the probability that reading

the actual bucket data will result in a cache hit. When the bucket metadata is read, it forces the bucket's first cache line to be loaded if it is not already in the cache or if its cache line is invalid.

The diagram in figure 3 on page 32 shows the memory layout of the data store. A hash table has a theoretical capacity determined by several parameters, the number of functions d, the number of buckets per zone (n), and the number of slots per bucket (q), and the number of buckets per zone. Each bucket is divided into q slots of equal size.

The location of the metadata seeks to exploit cache locality. Because a fetch from DRAM can take on the order of 50 times longer than a fetch from a valid L1 cache line, minimizing cache misses can improve the performance of a design by a significant constant factor.

6 Algorithm description

6.1 Hierarchical organization and addressing

Recalling the context of figure 3 on page 32, a storage location can be thought of as a 4-tuple consisting of a table id (one of $\{0, 1, 2, \ldots, d-1\}$), bucket zone id (one of $\{0, 1, 2, \ldots, m-1\}$), bucket id (one of $\{0, 1, 2, \ldots, n-1\}$) , and slot id (one of $\{0, 1, 2, \ldots, q-1\}$). Every such storage location is therefore associated with a bucket zone, and every bucket zone has a lock. As a result of this hierarchical organization, every bucket zone contains nq different storage locations.

6.2 Locking

Buckets are divided into contiguous ranges called bucket zones, and each bucket zone is associated with a lock, which protects the contents of buckets within the bucket

zone from concurrent modification by independent threads of execution. For more details see section 5.1.2 on page 31 and section 5.3.2 on page 34.

6.3 Version counters

Each bucket has a version counter that is incremented twice by the writer thread each time the bucket's content is modified: once before modification, and again after. Therefore if a bucket's version counter is odd, it means that a writer thread owns a lock on that bucket's zone and that same writer thread is in the process of modifying the bucket's contents. Because insert and delete only modify buckets while a lock is held, this provides a way for `lookup` threads to determine whether data read is valid. The reader thread reads a bucket's version counter, then reads the relevant bucket data, and reads the version counter again. If the version counter is unchanged and even, then the bucket data is considered valid.

6.3.1 In-flight insert problem

Although this description addresses the issue of bucket consistency, it does not fully capture what is required to look up an item. During the execution of an insert, it is common and normal that several buckets are modified along the cuckoo path. At each point in the cuckoo path, an item is ejected, and then inserted into the next location on the path. In the time interval after ejection but before insertion into the next location, the relevant item does not exist in any bucket. We say that such an item is "in flight" when this occurs.

This issue was discovered by a correctness test that would execute a lookup after each insert, and the issue was observed very infrequently. To solve this problem, we introduced a "read barrier" operation into `lookup`, whereby the locks of the lockset of the relevant item are acquired and released one by one. Note that this is different from acquiring all locks and then releasing them, as is done for insert and delete,

because in this case only one lock is held at a time. Thus the description of the previous paragraph is augmented as follows. The reader thread reads a bucket's version counter, then reads the relevant bucket data, followed by a read barrier operation (acquiring and releasing all locks in the item's lockset in order), and reads the version counter again. If the version counter is unchanged, then the bucket data is considered valid.

6.4 Insert

The insert algorithm is the most complex of the operations, because it involves the identification of a cuckoo path (the dry walk), followed by a locking stage where all relevant buckets are locked in order and the cuckoo path is verified using the version counters, followed by the modification stage where the changes are committed.

The top level insert is little more than a dispatching retry loop that serves as the entry point to the insert operation. Listing 3 on page 39 shows the top-level insert function.

Listing 3: The top-level `insert()` operation.

```
1  static int pcuckoo_basic_insert(struct pcuckoo_table *table,
2                                  struct pcuckoo_item *item) {
3
4    int ret;
5    log_insert_start();
6    struct path *path = get_path();
7
8    struct slot *slot_entry = mk_slot_entry(table, item);
9    if (slot_entry == NULL) {
10     ret = ENOMEM;
11     goto ret1;
12   }
13
14   for_u(i, 0, MAX_PATH_LOCK_FAILURES) {
15     ret = pcuckoo_basic_insert_inner(table, slot_entry, path);
```

```
16      if (ret == E_OK) {
17        goto ret2;
18      } else if (ret == E_PATH_DEPTH_EXCEEDED) {
19        goto ret2;
20      } else if (ret == E_LOCK_FAILED) {
21        /* continue */
22      } else if (ret == E_CONCURRENT_MOD) {
23        /* continue */
24      } else {
25        assert(0);
26        goto ret2;
27      }
28    }
29
30  ret2:
31    free(slot_entry);
32
33  ret1:
34    put_path(path);
35    log_insert_end(ret);
36    return ret;
37  }
```

Listing 4 on page 40 shows the second level of the insert operation, which makes the first call to the recursive function `find_path` that builds the cuckoo path. If a cuckoo path is successfully constructed, all of the locks associated with the path are acquired, and the insert is committed to the table. This is performed in listing 5 on page 41.

In order to avoid complicating the description of locking, we have not yet mentioned how a stash is handled in the algorithm. The stash appears in listing 4 on page 40. We defer a detailed discussion of the stash algorithm until section 7.8.1 on page 56. The presence or absence of the stash is one of the design parameters explored in the later experiments described in section 7 on page 48.

Listing 4: The second-level `insert()` operation.

```
1  static int pcuckoo_basic_insert_inner(struct pcuckoo_table *table,
2                                        struct slot *entry, struct path
                                             *path) {
3    path_clear(path);
4    int ret = pcuckoo_basic_find_path(table, entry, path);
5    insert_func_t which_func;
6    if (ret == E_OK) {
7      if (path->flags & PATH_FLAG_STASH) {
8        which_func = insert_stash_existing;
9      } else {
10       which_func = insert_nostash;
11     }
12     return do_insert(table, entry, path, which_func);
13   } else {
14     assert(ret == E_PATH_DEPTH_EXCEEDED);
15
16     if (!table_has_stash(table)) {
17       return ret;
18     }
19     path->nelem = 1;
20     which_func = insert_stash_overflow;
21     return do_insert(table, entry, path, which_func);
22   }
23 }
```

Listing 5: Insert in the non-stash case.

```
1  static int insert_nostash(struct pcuckoo_table *table, struct path *
       path) {
2    int ret = path_lock(path, table);
3
4    if (ret != E_OK) {
5      /* unlock is already taken care of in the case of unsuccessful
           lock. */
6      assert(ret == E_CONCURRENT_MOD);
7      return ret;
8    } else {
9      /* Now that everything is verified, copy all of the entries
10      * through the graph. */
```

```
11      foreach_path_entry(eid, path) {
12        struct path_entry *curr_frame = path_eid2entry(path, eid);
13
14        assert_new_location_coherent(table, &curr_frame->slot,
              curr_frame);
15
16        assert(is_hashval_coherent(table, &curr_frame->slot));
17
18        copy_slot_to_bucket(table, frame2bucket(table, curr_frame),
19                            frame2table_slot(table, curr_frame),
20                            &curr_frame->slot, curr_frame->slot_id);
21      }
22    }
23    path_unlock(path, table);
24    return E_OK;
25  }
```

We commit the transaction by copying the contents of the cuckoo path in order into their new locations. This is accomplished in the non-stash cases in listing 5 on page 41, lines 11-21. The cuckoo path records each ejectee's slot entry into the path structure. In this way, the cuckoo path "shadows" the table data by making a "shadow copy" of each affected slot entry in the path, and encodes the state of the table as it would be during a classic cuckoo hashing insert. This is necessary because a classic cuckoo hashing insert alters the contents of the table as it proceeds, and planning a cuckoo path must account for these alterations. The copy step "commits" or "applies" the cuckoo path to the actual table. Before and after each copy, the relevant bucket's version data is incremented. The copy and version update are both performed by the `copy_slot_to_bucket` function invoked on line 18.

Listing 6 on page 43 recursively builds a cuckoo path through the hash table, terminating when it finds an empty slot or the maximum path length is reached. This could be viewed as similar to depth-first search for an empty slot. However, unlike the classic depth-first search as found in algorithms textbooks, the path finding algorithm does not "turn back" when it encounters a node already visited. The search only stops when a path length limit is exceeded or an empty slot is found. On successful

completion, an empty slot has been found, and a path has been constructed.

Listing 6: The recursive `find_path()` function.

```
1  static int pcuckoo_basic_find_path(struct pcuckoo_table *table,
2                                     struct slot *entry, struct path *
                                          path) {
3    struct pcuckoo_basic_table *desc = table2desc(table);
4    if (path_is_full(path)) {
5      inc_metrics(path_depth_exceeded);
6      return E_PATH_DEPTH_EXCEEDED;
7    }
8
9    struct path_entry *frame = item2frame(table, entry, path);
10   entry = &frame->slot;
11
12   /* This branch will only happen on the very first (non-recursive)
13    * call, to determine if the item is already in the table.  At the
14    * very first call, the slot entry cannot undergo concurrent
15    * modification because it is the entry of the item to be inserted
16    * into the table, the item from the top-level call.  If we find it
17    * in the table (which may include the stash), then indicate the
18    * item is in the table. */
19   if (path->nelem == 1) {
20     int do_copy = 0;
21     int ret = search_single_item(table, entry, path, do_copy);
22     if (ret == E_OK) {
23       return ret;
24     }
25   } else {
26     /* Having chosen our parameters, populate the version information
27      * for this frame.  The version information depends on the
28      * lockset, which depends on the contents of entry at the time
29      * item2frame was called.  So the version information is that of
30      * the randomly chosen bucket location from the hash of the item.
31      * If path->nelem == 1 we don't collect version information,
32      * because it was already done before, and doing so again would
33      * cause a race condition allowing duplicate entry insert (race
34      * event occurs when we find no duplicate, drop through to here,
35      * in the meantime another thread also finds no duplicate for
           same
36      * item, both insert successfully). */
37     collect_version_info(table, frame);
```

```
38    }
39
40    /* randomly choose the table */
41    frame->choice = pick_hashfunc(table, frame);
42
43    for_u(i, 0, desc->slot_select_num_tries) {
44      frame->slot_id = pick_slot_id(table, frame);
45      /* If this is a recursive call, then item may have undergone
46       * concurrent modification.  If that happens, this will be
             detected
47       * at path lock time, because path lock checks the version
48       * information of the frame, which is snapshotted below. */
49
50      /* After taking the snapshot of the version information, we check
             if
51       * the slot is empty.  If it is, we return OK, otherwise we
52       * try again. */
53      if (is_frame_loc_empty(table, frame)) {
54        inc_metrics(good_pathlen);
55        return E_OK;
56      }
57    }
58
59    /* Recursively call pcuckoo_basic_find with the ejectee as the new
60     * slot entry.  The path "shadows" the table in the sense that if a
61     * previous path step ejected an item, we want to retrieve it from
62     * the path, not from the current table value, because the path is
63     * like a script of changes that will be applied, a transaction. */
64    struct slot *new_entry = frame2table_slot_shadowed(table, frame,
          path);
65
66    return pcuckoo_basic_find_path(table, new_entry, path);
67  }
```

At each path step, it records the choice of hash function, bucket zone, bucket offset, slot, lockset and version of the path step's location. The boolean condition of line 19 is only true on the very first – hence non-recursive – call, and the associated branch is for the case where the item is already in the table. The reason `collect_version_info` does not occur in this branch is because it is performed as part of `search_single_item`.

Listing 7: The `path_lock()` function.

```
1  static int path_lock(struct path *path, struct pcuckoo_table *table)
       {
2    /* Sort the lockset in ascending order of lock id.  Locks will be
3     * acquired in the resulting order, preventing deadlock. */
4    path_sort_locks(path, table);
5
6    /* Acquire the locks.  If any lock fails, unlock all previous. */
7    unsigned path_locks = path_num_locks(path);
8    for_u(i, 0, path_locks) {
9      int ret = path_lock_entry(path, table, i);
10     if (ret != E_OK) {
11       path_lock_revert(path, table, path_locks);
12       return E_LOCK_FAILED;
13     }
14   }
15
16   /* With the lockset of the path acquired, check that the versions
         agree. */
17   foreach_path_entry(eid, path) {
18     struct path_entry *frame = path_eid2entry(path, eid);
19     if (!path_entry_version_agrees(table, frame)) {
20       path_lock_revert(path, table, path_locks);
21       return lock_fail_from_concurrent_mod();
22     }
23   }
24
25   return E_OK;
26 }
```

When an empty slot is found, the set of locks associated with all elements to be inserted along the path are locked in order. This is the *lock walk* described in 5.5, and shown in listing 7 on page 45. When all buckets are locked, the algorithm walks the path, checking for concurrent modification by comparing each step location's version to insure it is the same as that recorded at the time of first visit. If all of these operations are successful, then the path is walked again, and the insert and eject operations at each stage are executed.

The insert approach described here differs from the breadth-first search of (Li

et al., 2015), where possible cuckoo paths are explored with breadth-first search in order to discover a valid cuckoo path before proceeding with locking. Our insert follows the "greedy approach" of original cuckoo hashing, which can be viewed as a depth-first search for an empty slot. The purpose of the current work is not to develop a better, faster algorithm, but to experimentally explore the structure of this algorithm's behavior in order to contribute to the body of knowledge about concurrent cuckoo hashing algorithms.

Proposition 6.1. `insert(x)` is deadlock-free.

Proof. `insert(x)` sorts locks according to their physical location in the table, which is stable, and acquires them in order. Therefore any threads of execution that attempt to acquire common subsets of locks always do so according to the same stable order, preventing deadlock due to a cycle in the locking graph. □

Proposition 6.2. In `insert(x)`, `path` includes all locksets for every inserted element on the cuckoo path.

Proof. By induction. For the base case, the first time `find_path(x, path)` is called, `path` is empty. The lockset for `x` is determined by the potential locations for `x` and set by the call to `item2frame` in line 8 of listing 6 on page 43, which calls the d hash functions for `x` and stores the values in the frame entry for the current path node.

If the selected slot is empty, `x` is the only inserted element on the cuckoo path, and because its lockset is in the one element of `path`, `path` includes all locksets for every inserted element on the cuckoo path.

For the induction step, suppose the hypothesis is true for all elements on the cuckoo path up to a particular recursive call of `find_path(x, path)`. Because the lockset for `x` is determined by the potential locations for `x` and set by the call to `item2frame` in line 8 of listing 6 on page 43, and `x` is the last element on the cuckoo path, `path` includes all locksets for every inserted element on the cuckoo path, which

is what we wanted to show. □

Proposition 6.3. Writes to the hash table only occur after all locksets for every inserted element on the cuckoo path are acquired.

Proof. The only place where elements are written into the hash table is lines 19 of listing 4 on page 40. This occurs in the branch starting at line 10, which only occurs if all locks have been acquired successfully. □

6.5 Delete

The algorithm for *Delete*(k) is trivial, because there is no need to traverse a cuckoo path. To delete a key k, lock $\mathbf{L}(k)$. If k is found in $\mathbf{T}(k)$, it is deleted and success is returned. Otherwise failure is returned.

6.6 Lookup

Lookup(k) searches the contents of $\mathbf{L}(k)$ for k, and returns the satellite data if the version data of the bucket agrees before and after search. As described in section 6.3.1 on page 38, a read barrier operation (locking and releasing lock of $\mathbf{L}(k)$) is performed before the second read of the version counter to check for concurrent modification.

6.7 Two levels of hash function

A practical problem that arises in the implementation of hash functions that map objects to hash values for the purpose of implementing hash tables is that the choice of hash function depends on the particular type of object being hashed. This is because the set of all possible objects that may be used as keys is very large, with no set bit length, and no single appropriate way to hash the data. For example, a hash function that is appropriate for hashing 32-bit unsigned integers may not be appropriate for hashing an audio file.

Our cuckoo hash table design requires that key objects support two functions provided by the user: `hash()` and `equals()`. The `hash()` function takes the key object as input and returns a 32-bit unsigned hash value for the object. We call this function the *client* hash function – as described in the "Client vs. implementer" section of (Myers, 2008) – to distinguish this data type-specific hash function from the hash functions used by the cuckoo hash table implementation to identify candidate buckets. The client hash function's return value becomes the input to the hash functions of the cuckoo hash table. The `equals()` method of key objects is used during *Insert* and *Lookup* to determine whether two key objects whose `hash()` values are equal represent key object equality or collision.[1]

As a result of this design, the hash functions of the cuckoo hash table only need to support 32-bit unsigned integer inputs as keys. For the cuckoo table implementation, we used the simplified hash functions proposed in (Aumüller et al., 2012) and referenced in (Aumüller et al., 2016).

6.8 Abstraction of hash function interface

In order to support experimentation with different hash functions, the hash function implementation details were abstracted into the `struct hash_ops` interface defined in `hashfuncs.h`.

7 Experimental Results

7.1 Workloads

Unless otherwise noted, in all workloads, 8 threads are started and immediately start processing the workload. A supervisor thread waits for the worker threads to exit, and results are collected. The source code for the experiments is available at

http://pcuckoo.s3-website-us-east-1.amazonaws.com/pcuckoo-0.4.tar.gz.

Under the high contention workload, each thread attempts to insert the same sequence of keys. The keys are 8192 contiguous 64-bit unsigned integers starting from a random 64-bit unsigned integer. The purpose of having all threads attempt to insert the same sequence of keys is to induce contention. Using contiguous keys $(a, a+1, a+2, \ldots, a+8191)$ avoids assumptions on entropy of the input, which tests the quality of the hash functions. For values, each thread uses the sum of the key and the thread id, so that the last thread that inserted each key can be determined after the test is run for the purposes such as measuring fairness.

Under the high throughput workload, each thread inserts a range of keys. The ranges are disjoint so that no thread will attempt to insert the same value as another thread. The name "high throughput" reflects the assumption that if no two threads are inserting the same key, then theoretically there should be less contention because there is a better probability that the locksets will be disjoint.

7.2 Failure modes

The high contention workload intentionally tries to insert more items than the hash table can accomodate, so that all threads will eventually fail with a max path length exceeded condition. This failure mode occurs when the insert fails due to failure to identify an empty slot within a certain number of steps `MAX_PATHLEN`, which is equivalent to an insert process with a cuckoo path of at least `MAX_PATHLEN`. In our experiments, we configured `MAX_PATHLEN` initially to 128, and then reduced it to 64.

7.3 First round of experiments

In the first round of experiments, we measured the number of successful inserts among 8 threads under the high contention workload before a max path length exceeded error

occurs. Each thread exits when the first path length exceeded error occurs.

7.4 Choosing the least loaded bucket is better than randomly choosing a bucket

The first question we would like to answer is, "does it matter if we choose the least loaded of the available buckets for any item versus picking a bucket at random?" At each point in the cuckoo graph, there are n choices for the next node, where n is the number of hash functions in the cuckoo hash table implementation. In our experiments we used two hash functions ($n = 2$). Within each bucket there are q slots (32 in our first round of experiments). This differs from the original cuckoo hashing algorithm, in which each bucket may only contain one item. This multiple-slot approach has some similarity with linear probing, in that a bucket location represents multiple storage locations that are probed for empty space. Here the nearby storage locations are the slots in a bucket.

Each bucket contains zero to q items. The algorithm may choose randomly from the `NUM_FUNCS` buckets, or choose the bucket with the most empty spaces (the least loaded bucket). In the random selection strategy, at each point in the cuckoo graph, the bucket choice is made uniformly at random from both buckets indicated by the hash function of the key being inserted. In the least loaded strategy, the bucket with the least number of occupied slots is chosen. Under both strategies, once the bucket is chosen, the slot within the bucket is chosen uniformly at random, without regard to whether or not it is empty.

Figure 4 on page 51 and figure 5 on page 52 show that randomly choosing the bucket leads to a max path length failure condition faster than choosing the least loaded bucket. The difference between median samples of the two strategies is significant. In both cases, once the bucket is selected, a random choice of the bucket

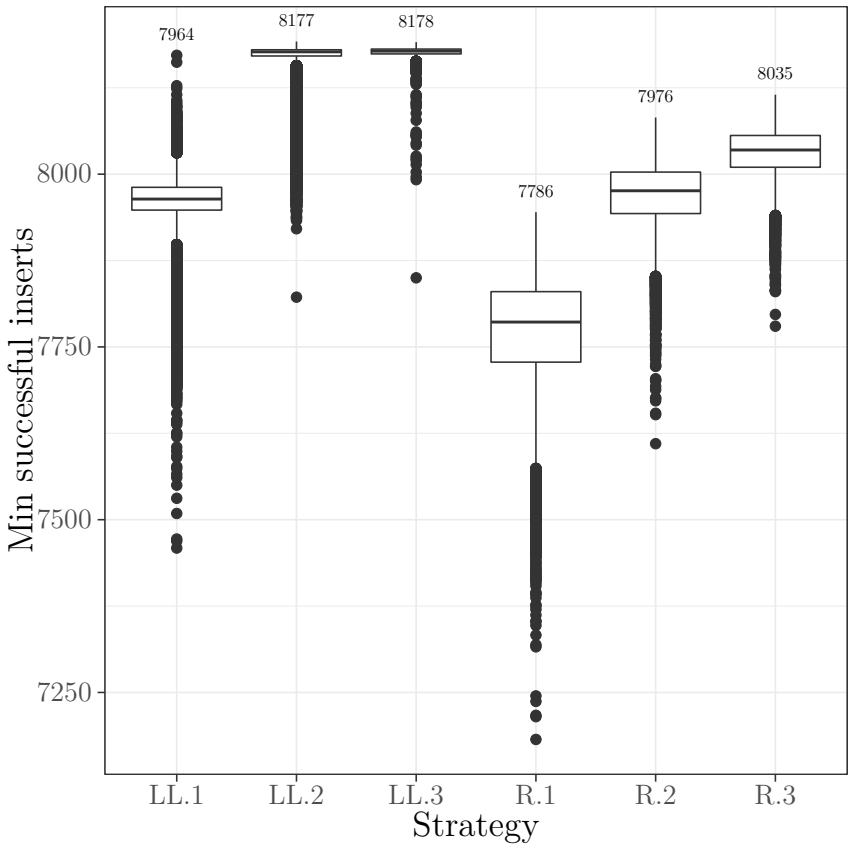

Figure 4: Minimum number of successful inserts among 8 threads running the high contention workload. "LL" means least loaded bucket choice strategy, "R" means random. "LL.3" or "R.3" means probe up to three slots before considering the chosen bucket full. The median of each strategy's readings is printed above the box plot. The suffixed number indicates the maximum number of random slot probes within the bucket before ejecting an occupant. If one of the probed slots is empty, it will be selected. If all retries are executed and no empty slot is probed, insert is recursively called with the occupant of the last slot probed as the ejectee. If an insert attempt requires more than 128 ejections, the thread exits. All experiments were run with two hash functions.

slots is made without regard to which slots are occupied. The data was generated by running the test program approximately ninety thousand times. Each run produces a new observation.

Both of these strategies differ from the insert algorithm of (R. Pagh & Rodler, 2001), which is deterministic, trying both possible buckets, and recursively calling insert with the last attempted bucket's ejectee if no empty bucket was found. Another

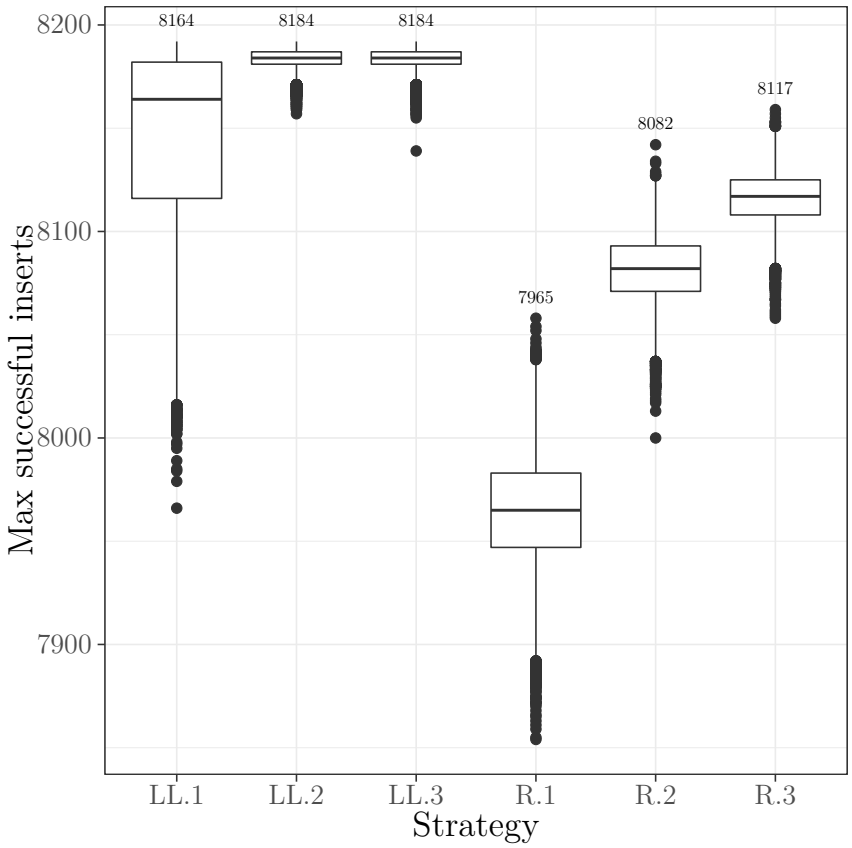

Figure 5: Maximum number of successful inserts among 8 threads running the high contention workload. "LL" means least loaded bucket choice strategy, "R" means random. The maximum number of successful inserts differs from the minimum number of successful inserts shown in figure 4 on page 51. All experiments were run with two hash functions.

difference is that the algorithm of (R. Pagh & Rodler, 2001) does not have multiple slots per bucket.

7.5 Retrying random slot ids improves inserts before failure

The next question we seek to answer is, "does it help to retry slot probes in the search for an empty slot before ejecting the occupant?" That is, does retrying slot probes defer the number of inserts before failure? The results of figure 4 on page 51 and figure 5 on page 52 show that in this round of experiments, retries of random slot id improve inserts before failure. The method of choosing a slot within the chosen

bucket varies randomly between zero, one or two retries. Listing 8 on page 53 shows the retry logic. If an empty slot is found within the retry loop, the function returns, otherwise it continues the search by returning the value of a recursive call, which means that the current node in the cuckoo graph will be an ejection.

Listing 8: Retry loop for slot probe.

```
1   for_u(i, 0, desc->slot_select_num_tries) {
2     frame->slot_id = pick_slot_id(table, frame);
3     if (is_frame_loc_empty(table, frame)) {
4       inc_metrics(good_pathlen);
5       return E_OK;
6     }
7   }
8
9   /* Recursively call insert_inner with the ejectee
10   * as the new slot entry. */
11  entry = frame2slot(table, frame);
12  return pcuckoo_basic_insert_inner(table, entry, path);
```

Retrying once was found to improve the number of inserts before failure. More than once was not as significant. However, since one retry pushed the table near capacity, the failure of multiple retries to improve time before failure leads to the question of path length. That is to say, if we change the maximum path length from 128 to 64, so that insert fails if a cuckoo path reaches length 64 rather than 128, how does it affect the number of successful inserts before failure?

7.6 Decreasing maximum path length from 128 to 64

This experiment addresses the question, “does changing the maximum path length from 128 to 64 affect time before failure?” Changing the maximum path length from 128 to 64 – so that an insert fails if the planned cuckoo path exceeds 64 entries – causes threads to fail approximately 5 to 7 per cent faster. Examining figures 6 and 4

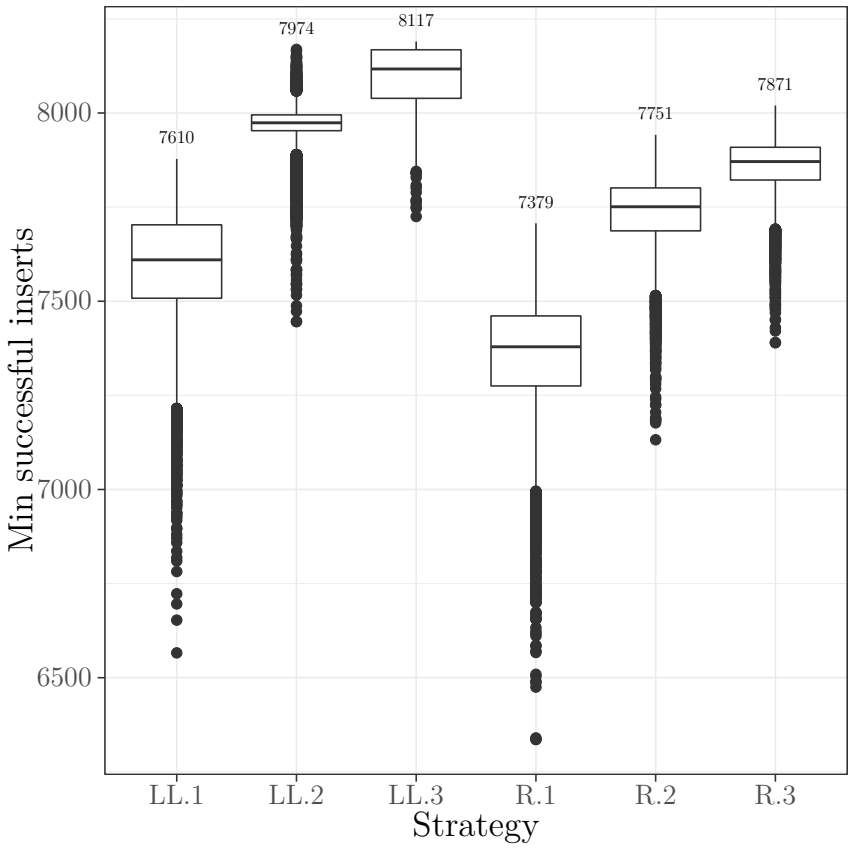

Figure 6: Minimum number of successful inserts among 8 threads running the high contention workload with maximum path length of 64 (reduced from 128).

suggest that the path length is important, but what is surprising is that even with a path length of 64, under the worst strategy (no retries, random bucket choice versus least loaded), the median fastest failing thread still manages to achieve a fairly high load before failure (median 7379 out of 8192 storage locations, approximately 90 per cent load). This leads to the question of what the significant contributors to this performance are. Candidates include the multiple slots per bucket structure, or some kind of "spreading" or "balancing" effect caused by the multiple threads.

7.7 Decreasing thread count to 1

Figure 7 on page 55 indicates that decreasing thread count to 1 does not decrease the median number of successful inserts. This suggests the need to identify some other

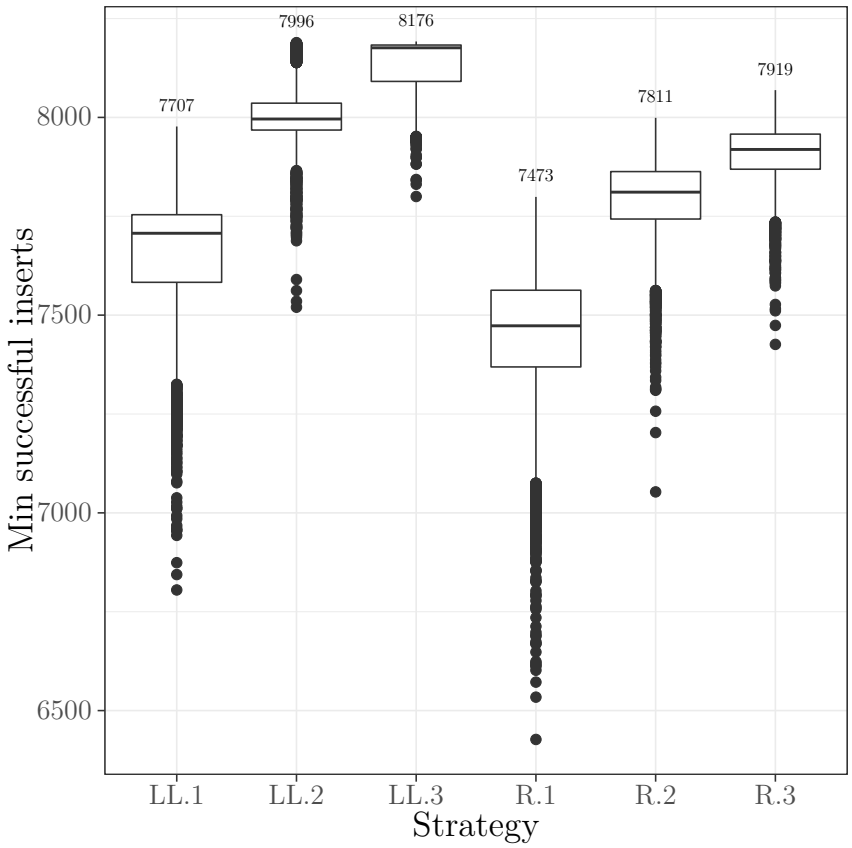

Figure 7: Number of successful inserts for 1 thread running the high contention workload with maximum path length of 64.

Strategy	Max path length	Inserts (1 thread)	Inserts (8 threads)
LL.1	64	7707	7610
LL.2	64	7996	7974
LL.3	64	8176	8117
R.1	64	7473	7379
R.2	64	7811	7751
R.3	64	7919	7871

Table 3: Table summarizing results with one thread and 8 threads, indicating that thread count does not significantly affect table load.

explanation for the high table occupancy. Table 3 on page 55 summarizes the results.

7.8 Experiments with high throughput workloads and a stash

The next round of experiments sought to explore the behavior of the hash table under a constant occupancy. All of these following tests used a table with a total capacity of 65536 entries. This number was chosen arbitrarily to lower the run time of experiments in order to gather a larger sample set. The range of 64-bit unsigned integers is partitioned among the threads (in this case 8 threads), and each thread inserts consecutive keys starting at a randomly chosen number within its range of values. Once each thread reaches its allocated occupancy, it starts deleting the oldest entry after each insert, so that after reaching the target occupancy, the occupancy remains constant for the remainder of the test.

Each experiment runs until all threads have exited, and each thread exits either when it encounters an insert failure (which always occurs due to path length exceeded), or it reaches an insert limit.

The target occupancy is 63,488, which is 97% ($\frac{31}{32}$) of the table's capacity. This load factor was chosen to observe the behavior of each configuration at or near the edge of its useful operation, so that the individual experiments can be expected to terminate in some reasonable time.

The goal of these experiments is to understand the specific effects of individual design parameters on the amount of time a table can be expected to run without failing, with the ideal amount of time being forever.

Each experiment randomly selects a value for the same configuration parameters as the other experiments (bucket select strategy and number of retries per slot), plus one more parameter indicating whether or not to use a stash.

7.8.1 Stash algorithm

We implemented the stash as a special "last bucket", a bucket zone consisting of one bucket. The stash therefore had 32 slots, just like any other bucket. To deal with

locking, we observed first that if the stash had to be locked during every destructive operation (insert or delete), then we would have degenerated to a single table lock, defeating one of the goals of using granularity to allow multiple independent inserts to occur concurrently if the locksets are disjoint.

The lock ordering scheme remains the same as described above, with the added property that the stash bucket's lock is the last one in any lock ordering. This allowed us to avoid locking the stash at every insert. We describe why this is so in the following paragraphs.

With a stash, insert has three cases. In the first case, insert without stash (the expected normal case), the item is inserted into a non-stash location. The stash bucket is not locked. In this case, an existing entry was not found in the stash with the key, and we check for concurrent modification by verifying that the bucket versions of all candidate locations have not changed after locks are acquired. This is exactly the same algorithm as in the case of a table without a stash.

In the second case, insert stash with existing item discovered, the lookup phase of $Insert(k)$ found that an item with key k exists in the stash. All of the locks in the lockset $\mathbf{L}(k)$ are acquired, with the addition of the stash lock. The entry is updated in the stash, and the version counters of all buckets in $\mathbf{L}(k)$ are incremented, along with the stash version counter.

In the third case, insert stash with overflow, a path depth failure has occurred (a would-be rehash condition), and the item is inserted into the stash. The locking proceeds as described in the second case.

Stating the above in a different way, in the second and third cases where the item is inserted into the stash, the bucket versions of the candidate buckets $\mathbf{L}(k)$ are incremented, so that a concurrent thread operating on an item with key k will observe version changes in the relevant buckets and avoid corruption by retrying. For example, if a concurrent insert was executing the first case (insert without stash)

on an item with the same key k, then it would observe the version number changes caused by the concurrent stash operation on the same key k and retry.

If the algorithm did not require that the stash-modifying insert thread increment the version counters of all buckets in the lockset, then in this case, the two threads attempting to insert the same item could both succeed, one in the stash and one through the normal non-stash path, resulting in two items present under the same key and thus corruption.

This approach avoids corruption while not requiring the stash to be locked at every insert.

7.8.2 Using feature elimination to investigate design parameters

Recall that there are three design parameters at play: bucket selection strategy (least loaded versus uniform random), slot probe retry count (the number of slots probed in the search for an empty slot at each step, either one, two or three), and whether or not a stash is used. Thus there are twelve different possible configurations.

The result of running experiments multiple times is a number of observations, each of which has a number of features. Because we are interested in the effect of our design parameters on performance, we can view our observations as a training set for an estimator, and use tools for feature elimination to help identify the design parameters (features) that are most relevant to our target feature, which is the number of inserts.

We used the recursive feature elimination (RFE) module of the scikit-learn machine learning library for this part of the investigation.

7.8.3 Behavior with maximum inserts set to one million

The first round of experiments had the maximum number of inserts set to one million, so that if a thread reaches a million inserts without failing, it exits. This was done

to bound the running time of each experiment to generate a reasonable number of observations with the various parameters.

Running the RFE procedure against the dataset indicated that the most determinative feature was the number of retries per slot, followed by the presence or absence of a stash, and finally the bucket select strategy. This information provided a starting point for ranking these features, but the most informative picture of the data emerged from a ranking of combinations of features, which we discuss in section 7.8.5 on page 59.

7.8.4 Behavior with one million maximum inserts compared to four million

To gather more data, a second round of experiments was initiated, but with a maximum of 4 million inserts. Table 4 on page 60, table 5 on page 60 and table 6 on page 61 summarize the number of inserts reached, grouped by the number of tries per slot, the presence or absence of a stash, and the bucket select strategy. For each cell, the first number is the result from the tests with one million maximum inserts, the second number in parentheses is the result from the tests with four million maximum inserts. The reason for including both numbers is to show that for some of the parameter combinations, the "clipping" effect of limiting the number of inserts for a run was significant.

7.8.5 Results grouped by feature combinations

The full picture of the various feature combinations becomes clear when every combination of features is inspected individually. Table **??** on page **??** shows the statistical summaries of results from the 4 million maximum inserts tests, grouped by every possible combination of features. The results are sorted by decreasing values of mean. They show a more complex picture than any one feature being dominant. For exam-

	count	mean	std	min
Tries				
1	2,064 (1,232)	7,729 (7,708)	271 (260)	6,455 (6,474)
2	1,632 (1,040)	248,760 (834,628)	437,756 (1,638,091)	6,982 (7,186)
3	1,664 (936)	292,848 (1,094,083)	424,954 (1,783,849)	7,611 (7,304)
Stash				
False	2,936 (1,728)	36,558 (34,514)	93,791 (84,465)	6,695 (6,474)
True	2,424 (1,480)	330,814 (1,244,548)	481,461 (1,893,277)	6,455 (7,025)
Strategy				
0	2,600 (1,744)	7,864 (7,849)	319 (341)	6,455 (6,474)
1	2,760 (1,464)	322,023 (1,289,536)	454,654 (1,880,508)	6,948 (6,925)

Table 4: Summaries of successful number of inserts per thread (count, mean, std, min) from the second round of experiments, with a high throughput workload (each thread inserting from a disjoint set of keys) under heavy occupancy (97% of slots occupied). Each entry contains the relevant data from the first round (one million inserts maximum), followed by the data from the second round in parentheses (four million inserts maximum). The count column indicates the number of samples, where one sample is a single thread's results from one run. The mean, std, and min columns represent the mean, standard deviation, and minimal value. The three table sections represent groupings according to the three parameters: retries per slot, presence or absence of stash, and bucket selection strategy.

	max	25%	50%	75%
Tries				
1	10,653 (8,597)	7,581 (7,556)	7,732 (7,722)	7,914 (7,903)
2	1,048,576 (4,194,304)	7,940 (7,886)	8,093 (8,038)	18,972 (14,398)
3	1,048,576 (4,194,304)	8,156 (8,218)	12,668 (35,766)	421,940 (721,272)
Stash				
False	1,048,576 (783,336)	7,682 (7,664)	7,902 (7,881)	9,570 (9,064)
True	1,048,576 (4,194,304)	7,927 (7,900)	8,060 (8,056)	1,048,576 (4,194,304)
Strategy				
0	9,618 (9,515)	7,693 (7,674)	7,844 (7,813)	8,009 (8,000)
1	1,048,576 (4,194,304)	7,949 (7,971)	11,512 (16,500)	1,048,576 (4,194,304)

Table 5: Summaries of number of successful inserts from the second round of experiments (max, 25%, 50%, 75%).

	1%	90%	99%
Tries			
1	7,088 (7,006)	8,000 (8,001)	8,263 (8,303)
2	7,543 (7,546)	1,048,576 (4,194,304)	1,048,576 (4,194,304)
3	7,732 (7,749)	1,048,576 (4,194,304)	1,048,576 (4,194,304)
Stash			
False	7,142 (7,036)	82,250 (94,871)	469,023 (459,112)
True	7,621 (7,583)	1,048,576 (4,194,304)	1,048,576 (4,194,304)
Strategy			
0	7,128 (7,046)	8,297 (8,289)	8,684 (8,926)
1	7,425 (7,442)	1,048,576 (4,194,304)	1,048,576 (4,194,304)

Table 6: Summaries of number of successful inserts from the second round of experiments (1%, 90%, 99%).

ple, a stash will not magically confer good performance upon a hash table that uses one retry per slot and a random bucket selection strategy. With three retries per slot, least loaded bucket strategy, and a stash, the threads consistently hit the maximum number of inserts, suggesting that the "clipping" problem remains even with 4 million maximum inserts when the configuration is 3,STASH,LL.

The top line of table 7 on page 62 indicates that this hash table design can be expected to run with 97% load for 4 million inserts. More experiments are needed to determine what the real "headroom" is here, either by increasing the occupancy or increasing the maximum inserts. Informally, we might say that under this round of experiments, the 3,STASH,LL configuration remains "unbroken", and we seek to understand the "breaking point" of this configuration.

7.8.6 "Breaking" the 3,STASH,LL configuration at 98% occupancy

In order to search for the "breaking point" for the 3,STASH,LL configuration in a third round of tests, we modify the experiment to increase the target occupancy to 64,512, which is 98% ($\frac{63}{64}$) of the table's total capacity. Recall that the target occupancy is

Combination	count	mean	std	min
3,STASH,LL	232	4,194,304	0	4,194,304
2,STASH,LL	216	3,984,290	623,125	1,133,463
3,NO_STASH,LL	272	174,432	148,715	7,958
2,NO_STASH,LL	248	11,484	4,365	7,729
3,STASH,R	208	8,388	227	8,057
2,STASH,R	232	8,024	208	7,186
3,NO_STASH,R	224	8,002	261	7,304
1,STASH,LL	264	7,984	106	7,425
2,NO_STASH,R	344	7,840	171	7,378
1,NO_STASH,LL	232	7,736	232	6,925
1,STASH,R	328	7,717	112	7,025
1,NO_STASH,R	408	7,505	255	6,474

Table 7: Summaries of successful inserts from from the 4 million max insert experiments (count, mean, std, min), grouped by every possible feature combination. The summaries are grouped by the configuration of the three design parameters, where for example "3,STASH,LL" means 3 retries per slot, with a stash and the least loaded bucket strategy. "2,NO_STASH,R" means 2 retries per slot, no stash and uniform random bucket select strategy.
w

the percentage of the table's slots that are full before the threads start deleting after insert. By comparison, the second round of tests were run with a target occupancy of 97% ($\frac{31}{32}$). Comparing the top lines of table 7 on page 62 and table 8 on page 63 show the difference in behavior. These results suggest that the 3,STASH,LL configuration is very stable up to somewhere around 97% total table occupancy, because that was the occupancy of the second round of experiments which had zero failures. Under 98% occupancy, the configuration starts to "break down" and experience failures.

7.9 Effect of stash on inserts per second

Having demonstrated the occupancy benefits of the 3,STASH,LL configuration, we explore the impact of the stash on hash table inserts per second. We modify our experiments to vary between the 3,STASH,LL and the 3,NO_STASH,LL configurations. We also vary the occupancy ratio – performing experiments with $\frac{3}{4}$ and $\frac{15}{16}$

Combination	count	mean	std	min
3,STASH,LL	1,536	1,766,892	1,392,837	14,602
3,NO_STASH,LL	1,664	11,102	4,690	7,431
2,STASH,LL	1,600	8,804	329	8,131
2,NO_STASH,LL	1,528	8,069	326	7,550
3,STASH,R	1,704	8,045	89	7,530
2,STASH,R	1,424	7,964	52	7,236
1,STASH,LL	1,616	7,948	69	7,244
3,NO_STASH,R	1,528	7,936	121	6,969
2,NO_STASH,R	1,384	7,830	143	7,084
1,NO_STASH,LL	1,624	7,730	209	6,590
1,STASH,R	1,560	7,722	71	6,657
1,NO_STASH,R	1,512	7,500	231	6,199

Table 8: Summaries of successful inserts from from the third round of experiments. These are the same experiments as the second round (high throughput workload with 4 million inserts max per thread), but with the target occupancy increased from 97% ($\frac{31}{32}$) to 98% ($\frac{63}{64}$) to induce failures in the 3,STASH,LL configuration. Results are presented as before (count, mean, std, min, grouped by every possible feature combination). The results indicate that the 3,STASH,LL starts to fail at 98% occupancy, with tests failing on average after 1,766,892 inserts (by comparison, all test runs completed all 4 million inserts in the second round of tests).

occupancy ratio – and vary the lock granularity. These tests were run with the high throughput workload, with 8 million inserts per thread and a table capacity of 64 million.

The results in table 9 on page 64 follow profiling of the code using the Callgrind and KCachegrind tools, eliminating unnecessary computations, inlining several functions, increasing the `gcc` optimization level to 3, and caching repeated operations such as fetching the thread-local storage area at api entry points. These results indicate that the fastest configuration is a 16% increase from the slowest configuration – significant and measurable, but not dramatic.

In these performance results, the lock granularity plays a significant role. This answers one of the original questions about the impact of lock granularity. In particular, lock granularity does appear to permit faster inserts per second in the high through-

Configuration	Inserts/sec
N,3/4,1	2,572,907
Y,3/4,1	2,553,468
N,15/16,1	2,505,206
Y,15/16,2	2,394,935
Y,15/16,4	2,366,985
N,3/4,2	2,361,870
N,15/16,4	2,356,475
N,15/16,16	2,333,771
Y,3/4,2	2,331,453
Y,15/16,8	2,326,557
N,15/16,8	2,319,221
N,3/4,4	2,313,731
N,3/4,8	2,309,036
Y,3/4,4	2,300,817
N,15/16,64	2,293,089
Y,15/16,16	2,291,533
N,3/4,16	2,291,261
Y,3/4,8	2,287,084
N,15/16,32	2,286,747
Y,15/16,32	2,275,409
Y,3/4,16	2,256,267
Y,15/16,64	2,248,077
Y,3/4,32	2,243,308
Y,3/4,64	2,223,511
N,3/4,64	2,217,431

Table 9: Sorted mean inserts per second from a high throughput workload with table capacity of 64 million and 8 million inserts per thread. In each experiment, 8 threads are launched at the same time and each is allowed to insert 8 million sequential integers, with each thread taking its integers from its own range disjoint from the other threads. Configurations represent presence or absence of stash (Y/N), target occupancy (the occupancy at which threads start deleting after every insert), and the number of buckets per zone. For example, "Y,3/4,16" means that the configuration had a stash, the target occupancy was 75%, and the lock granularity was sixteen buckets per zone (meaning that each lock is associated with sixteen buckets). Inserts per second are aggregated, meaning that the total number of inserts completed by all threads is divided by the elapsed time from start of threads to completion of all threads. All results were executed on a laptop equipped with a 8-core Intel Core i7-6700HQ CPU operating at 2.60GHz.

put workload. However, the results suggest that lock granularity may be viewed more as a tradeoff rather than a critically important requirement, because even with the

slowest configuration with 64 buckets per lock (hence 2,048 slots per lock due to the 32 slots per bucket), the performance gain achieved from moving to the highest lock granularity in this particular workload is only 15%.

These results suggest that because the stash does not appear to be correlated with slower insert performance, having a stash is almost certainly worthwhile in most applications due to the high occupancies it supports as demonstrated in section 7.8.6 on page 61.

7.10 Comparison with libcuckoo implementation of Goyal, Fan, Li, Anderson and Kaminsky

An ad-hoc test comparing the time required to insert 10 million items with 8 threads under the libcuckoo implementation of (Goyal et al., 2013) and our implementation indicated that the libcuckoo implementation was approximately 57% faster, with the libcuckoo implementation completing approximately 5.9 million inserts per second versus approximately 3.8 million inserts per second for our implementation. Because the libcuckoo implementation is more mature and has been developed for several years, this result suggests that the algorithm we have described in this thesis may merit further study.

8 Conclusions and future work

We have demonstrated a multi-reader, multi-writer parallel cuckoo hashing scheme with multiple slots per bucket. This scheme provides another platform for experiments with parallel cuckoo hashing. We found that the proposed path construction and locking algorithm successfully passes automated correctness tests under a high contention workload. We showed experimentally that the design parameters of retrying randomly chosen slots, use of a stash, and choosing the least loaded bucket all

improve the number of inserts before failure, and explored the observed results of combinations of these design parameters, which we called configurations. We showed that the configuration of 3,STASH,LL (maximum of 3 probe retries per bucket, stash enabled, and least-loaded bucket strategy) outperforms the others in terms of occupancy by a wide margin, and that it "breaks down" somewhere between 97% and 98% table occupancy.

We also showed experimental results suggesting that higher lock granularity does yield a performance improvement under the high throughput workload of multiple threads attempting to insert disjoint keys, but the performance increase was not dramatic (at most 16%). We also showed that under an initial ad-hoc insert test of 10 million items with default parameters, the more mature libcuckoo implementation of (Goyal et al., 2013) was approximately 57% faster than the one described here. This suggests that our implementation may merit further study.

There are several more areas of experimentation to explore, some of which we outline here.

1. Does replacing the depth-first (classic) path construction with breadth-first search as in Li et al. (2015) significantly improve performance? Is there a substantive difference between the insert algorithm of this thesis – what we call "depth-first (classic)" – and the random walk mentioned by Fotakis, Pagh, Sanders, and Spirakis (2005, p. 14)?

2. Does varying the slots per bucket (q) confirm the hypothesis that multiple slots per bucket is the dominant factor behind the improved occupancy observations?

3. For the underlying table hash functions, we implemented the functions proposed in (Aumüller et al., 2012) and referenced in (Aumüller et al., 2016). What effect does changing various parameters of these functions, or replacing them with other functions, have on the performance?

4. With more performance tuning, how does our implementation compare with other parallel cuckoo hashing implementations such as (Goyal et al., 2013)?

9 Notes

Section 2

1. The experiments were run as part of an assignment given in CSCIE-210 "Algorithms at the End of the Wire", fall 2012.

Section 6

1. This is a common solution to this problem. For example, the same approach is used by Java with the `hashCode()` and `equals()` methods of class `Object`, and Python with the `__hash__()` and `__eq__()` methods. For more information about the Java and Python methods, see (Oracle, 2018) and (The Python Software Foundation, 2018).

References

Aumüller, M., Dietzfelbinger, M., & Woelfel, P. (2012). Explicit and efficient hash families suffice for cuckoo hashing with a stash. *CoRR*, *abs/1204.4431*. Retrieved from `http://arxiv.org/abs/1204.4431`

Aumüller, M., Dietzfelbinger, M., & Woelfel, P. (2016). A simple hash class with strong randomness properties in graphs and hypergraphs. *CoRR*, *abs/1611.00029*. Retrieved from `http://arxiv.org/abs/1611.00029`

Azar, Y., Broder, A. Z., Karlin, A. R., & Upfal, E. (1999, September). Balanced allocations. *SIAM J. Comput.*, *29*(1), 180–200.

Bertsekas, D., & Tsitsiklis, J. (2008). *Introduction to probability, second edition* (2nd ed.). Athena Scientific.

Carter, J., & Wegman, M. N. (1979). Universal classes of hash functions. *Journal of Computer and System Sciences*, *18*(2), 143 - 154. Retrieved from `http://www.sciencedirect.com/science/article/pii/0022000079900448` doi:

https://doi.org/10.1016/0022-0000(79)90044-8

Dietzfelbinger, M., Karlin, A., Mehlhorn, K., auf der Heide, F. M., Rohnert, H., & Tarjan, R. E. (1988, Oct). Dynamic perfect hashing: upper and lower bounds. In *[proceedings 1988] 29th annual symposium on foundations of computer science* (p. 524-531). doi: 10.1109/SFCS.1988.21968

Dietzfelbinger, M., Karlin, A. R., Mehlhorn, K., Meyer auf der Heide, F., Rohnert, H., & Tarjan, R. E. (1994). Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, *23*(4), 738–761.

Dietzfelbinger, M., & Schellbach, U. (2009). On risks of using cuckoo hashing with simple universal hash classes. In *In proc. 20th acm/siam symposium on discrete algorithms (SODA)* (pp. 795–804).

Dietzfelbinger, M., & Weidling, C. (2007). Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, *380*(1), 47 - 68.

Fan, B., Anderson, D. G., & Kaminsky, M. (2013). Memc3: Compact and concurrent memcache with dumber caching and smarter hashing.

Fotakis, D., Pagh, R., Sanders, P., & Spirakis, P. (2005, 02). Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, *38*(2), 229.

Goyal, M., Fan, B., Li, X., & Andersen, M., David G.and Kaminsky. (2013). *libcuckoo.* `https://github.com/charlespwd/project-title`. GitHub.

Kaser, O., & Lemire, D. (2012). Strongly universal string hashing is fast. *CoRR*, *abs/1202.4961*.

Kirsch, A., Mitzenmacher, M., & Wieder, U. (2009). More robust hashing: Cuckoo hashing with a stash*. *SIAM Journal on Computing*, *39*(4), 1543-1561.

Knuth, D. E. (1998). *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc.

Li, X., Andersen, D. G., Kaminsky, M. K., & Freedman, M. J. (2015). Algorithmic improvements for fast concurrent cuckoo hashing.

Mitzenmacher, M. (2009). *Some Open Questions Related to Cuckoo Hashing.*

Mitzenmacher, M., Richa, A. W., & Sitarman, R. (2001). *The power of two random choices: A survey of techniques and results.*

Mitzenmacher, M., & Upfal, E. (2017). *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis* (2nd ed.). New

York, NY, USA: Cambridge University Press.

Myers, A. (2008). *Cs3110 lecture 21: Hash functions.* http://www.cs.cornell.edu/courses/cs3110/2008fa/lectures/lec21.html.

Oracle. (2018). *Object (java platform se 8).* https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html.

Pagh, A., Pagh, R., & Ruzic, M. (2009). Linear probing with constant independence*. *SIAM Journal on Computing*, *39*(3), 1107-1120. Retrieved from http://search.proquest.com.ezp-prod1.hul.harvard.edu/docview/880202424?accountid=11311 (Copyright - Copyright Society for Industrial and Applied Mathematics 2009; Document feature - Graphs; ; Last updated - 2012-02-21)

Pagh, R., & Rodler, F. F. (2001). Cuckoo hashing. In *9th annual european symposium on algorithms.*

Pătraşcu, M., & Thorup, M. (2015, November). On the k-independence required by linear probing and minwise independence. *ACM Trans. Algorithms*, *12*(1), 8:1–8:27. Retrieved from http://doi.acm.org.ezp-prod1.hul.harvard.edu/10.1145/2716317 doi: 10.1145/2716317

The Python Software Foundation. (2018). *The python language reference, version 3.7: Data model.* https://docs.python.org/3.7/reference/datamodel.html.

Thorup, M. (2015). High speed hashing for integers and strings. *CoRR*, *abs/1504.06804*. Retrieved from http://arxiv.org/abs/1504.06804

Wegman, M. N., & Carter, J. (1981). New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, *22*(3), 265 - 279. Retrieved from http://www.sciencedirect.com/science/article/pii/0022000081900337 doi: https://doi.org/10.1016/0022-0000(81)90033-7