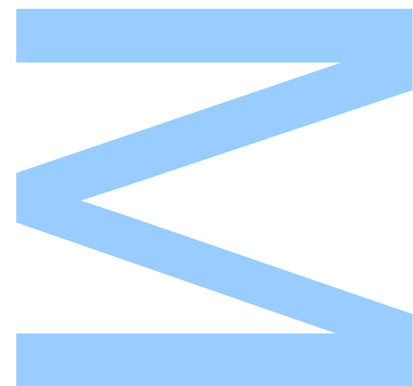
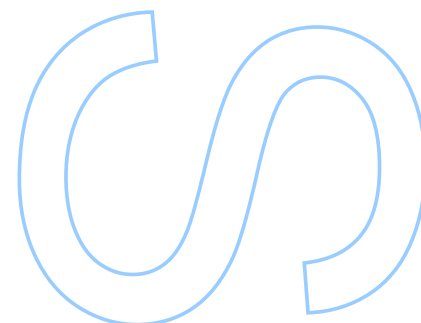
Memory Reclamation for an Elastic Lock-free Hash Trie Map

João Miguel Chamiça Pereira
Masters in Computer Science
Departamento de Ciência de Computadores
2022

Orientador
Ricardo Jorge Gomes Lopes da Rocha, Faculdade de Ciências

Supervisor
Pedro Carvalho Moreno, Faculdade de Ciências





U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ______/______/________

MSC

DECLARAÇÃO DE HONRA

Eu, **João Miguel Chamiça Pereira**, natural de **Portugal**, residente em **Portugal**, nacional de **Portugal**, portador (a) do Cartão de Cidadão nº **15981523**, inscrito(a) no Mestrado em **Mestrado em Ciência de Computadores** da Faculdade de Ciências da Universidade do Porto declaro, nos termos do disposto na alínea a) do artigo 14.º do Código Ético de Conduta Académica da U.Porto, que o conteúdo da presente dissertação *Memory Reclamation for an Elastic Lock-free Hash Trie Map* reflete as perspetivas, o trabalho de investigação e as minhas interpretações no momento da sua entrega.

Ao entregar esta dissertação *Memory Reclamation for an Elastic Lock-free Hash Trie Map*, declaro, ainda, que a mesma é resultado do meu próprio trabalho de investigação e contém contributos que não foram utilizados previamente noutros trabalhos apresentados a esta ou outra instituição.

Mais declaro que todas as referências a outros autores respeitam escrupulosamente as regras da atribuição, encontrando-se devidamente citadas no corpo do texto e identificadas na secção de referências bibliográficas. Não são divulgados na presente dissertação *Memory Reclamation for an Elastic Lock-free Hash Trie Map* quaisquer conteúdos cuja reprodução esteja vedada por direitos de autor.

Tenho consciência de que a prática de plágio e auto-plágio constitui um ilícito académico.

Assinatura do Autor

Data

26/09/2022

Acknowledgements

This thesis would not have been possible if not for the support and guideance of both my supervisors Pedro Moreno and Ricardo Rocha.

I would additionally like to thank INESC-TEC for providing financial support during the INFORUM 2022 conference, in which we published and presented a paper.

UNIVERSIDADE DO PORTO

Abstract

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

MSc. Computer Science

Memory Reclamation for an Elastic Lock-free Hash Trie Map

by João Miguel Chamiça Pereira

A hash map is elastic if it can expand and compress. Hash maps expand in order to reduce collisions and compress in order to reduce depth and memory usage. Starting from a particular elastic lock-free hash map design, called the lock-free hash trie map, which implements expansion and compression in constant time while maintaining the high throughput of lock-freedom, we focus on solving the problem of memory reclamation outside garbage collected environments without losing the lock-freedom property. We propose a lock-free and safe memory reclamation method using hazard pointers that is compatible with the compression mechanism of this data structure. Experiments show that our approach obtains results on par with the best state-of-the-art memory reclamation methods, in execution time. On the other hand, our proposed method is capable of maintaining lower memory consumption than the alternative methods.

Contents

Acknowledgements **D**

Abstract **E**

Contents **G**

List of Figures **I**

List of Tables **K**

1 Introduction **1**

2 Background **3**
2.1 Hash tries 5
2.2 State-of-the-Art Comparison 7

3 Lock-free Hash Trie Map **9**
3.1 Lookup 9
3.1.1 Average Path Length 10
3.1.2 Algorithm 10
3.2 Insertion 13
3.2.1 Dealing With Collisions 13
3.2.2 Algorithm 14
3.3 Expansion 15
3.3.1 Algorithm 16
3.4 Removal 18
3.4.1 Invalidation Step 18
3.4.2 Memory Reclamation Problem 19
3.4.3 Delegation Problem 20
3.4.4 Algorithm 21
3.5 Compression 21
3.5.1 Freezing 22
3.5.2 Counter 25
3.6 Cost Analysis 28
3.6.1 Lookup Cost 28
3.6.2 Insertion Cost 30
3.6.3 Removal Cost 30

3.6.4 Expansion Cost . . . 30
3.6.5 Freeze Compression Cost . . . 30
3.6.6 Counter Compression Cost . . . 30
3.6.7 The Cost of Synchronization . . . 31

4 Memory Reclamation 33
4.1 The Cost Of Synchronization . . . 34
4.1.1 The Cache . . . 34
4.1.2 Memory Barriers . . . 37
4.1.3 Summary . . . 40
4.2 Memory Life Cycle . . . 40
4.3 Memory Reclamation Methods . . . 41
4.3.1 Hazard Pointers . . . 41
4.3.2 Hazard Hash And Level . . . 45
4.4 Our Contribution . . . 46
4.4.1 Number of Hazard Pointers . . . 50
4.4.2 Other Important Changes . . . 52
4.4.3 Delegation Problem . . . 54

5 Experiments 57
5.1 Benchmark Program . . . 58
5.1.1 Metrics . . . 58
5.2 Machine Specifications . . . 59
5.3 Hash Map Parameters . . . 60
5.3.1 Chain Length . . . 61
5.3.2 Chunk Size . . . 61
5.3.3 Memory Allocator . . . 64
5.4 Compression . . . 65
5.5 Memory Reclamation . . . 68

6 Conclusions 73

A SNF - benchmark data 75

B FNF - benchmark data 77

C CNF - benchmark data 79

D SNF (empty map) - benchmark data 81

E FNF (empty map) - benchmark data 83

F CNF (empty map) - benchmark data 85

G FHP - benchmark data 87

H FHPA - benchmark data 89

I HHL - benchmark data 91

List of Figures

2.1 Hash trie map 5
2.2 Comparison of state-of-the-art designs 7

3.1 Lookup demonstration for different map configurations 11
3.2 Inserting nodes 13
3.3 Expanding to reduce the average path length 15
3.4 Demonstration of the complete expansion of a collision chain 17
3.5 Removing node K_1 18
3.6 Conflicting operations when removing a node 19
3.7 The delegation problem 20
3.8 Conflicting operations when compressing a hash node 22
3.9 Compression using a freeze node 23
3.10 Aborting the compression operation 24
3.11 Compression using a counter field 26
3.12 Hash trie map with a perfect hash function 29

4.1 Typical multi-core UMA architecture 35
4.2 Unnecessary CPU stalls due to cache synchronization 37
4.3 The memory reclamation problem 41
4.4 Solution to the memory reclamation problem using hazard pointers 42
4.5 Hazard pointers applied to the traversal of the LFHT tree 47
4.6 Hazard pointers applied to the traversal of LFHT collision chains 48
4.7 Protecting nodes during lookup (dotted lines represent HP protections) . . 49
4.8 Infinite collision chain traversal 52
4.9 Not enough hazard pointers when expanding 53
4.10 Unable to protect a parent hash node 53
4.11 LFHT's delegation problem 54

5.1 Overhead caused by gathering additional statistics 59
5.2 Architecture visual description of our benchmark machine 60
5.3 Variable chain length benchmark - 2^{24} nodes; average of 10 samples; chunk size 4 62
5.4 Variable chunk size benchmark - 2^{24} nodes; average of 10 samples; chain length 4 63
5.5 Comparing the original memory allocator with *jemalloc* 65
5.6 Compression benefits - 2^{24} nodes, average of 10 samples 66
5.7 Compression overheads - 2^{24} nodes, average of 10 samples, chain length and chunk size of 4 67

5.8 Cache misses - 25% of insertions; 25% of removals; 50% of searches 68
5.9 Memory consumption - 25% of insertions; 25% of removals; 50% of searches 69
5.10 Memory reclamation method throughput benchmark - 2^{24} nodes, average of 10 samples 71

List of Tables

4.1 Demonstrating the effects of the delegation problem in the reclamation procedure . . . 55

A.1 SNF - 25% removals, 25% insertions, 50% searches . . . 75
A.2 SNF - 100% insertions . . . 75
A.3 SNF - 100% removals . . . 75
A.4 SNF - 100% searches of inserted keys . . . 76
A.5 SNF - 100% searches of keys not inserted on the map . . . 76

B.1 FNF - 25% removals, 25% insertions, 50% searches . . . 77
B.2 FNF - 100% insertions . . . 77
B.3 FNF - 100% removals . . . 77
B.4 FNF - 100% searches of inserted keys . . . 78
B.5 FNF - 100% searches of keys not inserted on the map . . . 78

C.1 CNF - 25% removals, 25% insertions, 50% searches . . . 79
C.2 CNF - 100% insertions . . . 79
C.3 CNF - 100% removals . . . 79
C.4 CNF - 100% searches of inserted keys . . . 80
C.5 CNF - 100% searches of keys not inserted on the map . . . 80

D.1 SNF (empty map) - 25% removals, 25% insertions, 50% searches . . . 81
D.2 SNF (empty map) - 100% insertions . . . 81
D.3 SNF (empty map) - 100% removals . . . 81
D.4 SNF (empty map) - 100% searches of inserted keys . . . 82
D.5 SNF (empty map) - 100% searches of keys not inserted on the map . . . 82

E.1 FNF (empty map) - 25% removals, 25% insertions, 50% searches . . . 83
E.2 FNF (empty map) - 100% insertions . . . 83
E.3 FNF (empty map) - 100% removals . . . 83
E.4 FNF (empty map) - 100% searches of inserted keys . . . 84
E.5 FNF (empty map) - 100% searches of keys not inserted on the map . . . 84

F.1 CNF (empty map) - 25% removals, 25% insertions, 50% searches . . . 85
F.2 CNF (empty map) - 100% insertions . . . 85
F.3 CNF (empty map) - 100% removals . . . 85
F.4 CNF (empty map) - 100% searches of inserted keys . . . 86
F.5 CNF (empty map) - 100% searches of keys not inserted on the map . . . 86

G.1 FHP - 25% removals, 25% insertions, 50% searches 87
G.2 FHP - 100% insertions 87
G.3 FHP - 100% removals 87
G.4 FHP - 100% searches of inserted keys 88
G.5 FHP - 100% searches of keys not inserted on the map 88

H.1 FHPA - 25% removals, 25% insertions, 50% searches 89
H.2 FHPA - 100% insertions 89
H.3 FHPA - 100% removals 89
H.4 FHPA - 100% searches of inserted keys 90
H.5 FHPA - 100% searches of keys not inserted on the map 90

I.1 HHL - 25% removals, 25% insertions, 50% searches 91
I.2 HHL - 100% insertions 91
I.3 HHL - 100% removals 91
I.4 HHL - 100% searches of inserted keys 92
I.5 HHL - 100% searches of keys not inserted on the map 92

Chapter 1

Introduction

In recent years concurrent programming has been increasing in relevance as processors became closer and closer to the physical boundaries of single threaded performance. The focus was, thus, shifted towards increasing core counts, distributed computing, concurrent algorithms, memory hierarchies and parallelism. However, because concurrent access to memory can break the semantics of algorithms, we need synchronization mechanisms to ensure correctness.

One of the most popular, studied and intuitive approaches to synchronization was through the use of locks. On the other hand, locks halt the execution of other threads preventing parallel algorithms from fully exploiting the maximum capacity of the computational resources available. Furthermore, the operating system can halt the progress of one thread which may prolong the time other threads remain waiting. Lock-free programming is a possible alternative that guarantees progress even in the event of one thread halting. Lock-freedom is a *non-blocking* progress guarantee, as stated by Herlihy and Shavit [23]. What distinguishes lock-freedom from other non-blocking progress guarantees is the fact that, in lock-freedom, threads can obstruct each other's progress. This means that a thread can get stuck retrying a CAS operation over and over indefinitely. Even though this thread is not blocked, it could still have its progress obstructed.

For a program to truly be non-blocking, all its component parts must also be non-blocking. This includes: memory allocator, the operating system kernel functions, the standard library functions, and the imported data structures. Unfortunately, a lot of these tools only have blocking implementations. If we want to design and implement a non-blocking data structure which uses, as an example, a blocking memory allocator, our data structure cannot truly be non-blocking because thread progression cannot be guaranteed

when using this memory allocator. If we implement non-blocking versions of these already existing tools, we can contribute to the performance of novel non-blocking algorithms.

Our goal is to design a lock-free hash trie map capable of dynamically expanding and compressing using fixed size nodes and persistent memory references. The hash trie map is a map whose bucket arrays form a tree. To search for keys on a hash trie map we use the partitioned hashing strategy which will help reduce the overhead of map expansion and compression. Our design, based on the *single width compare and swap* (CAS) operation, is the continuation of the work done by Areias, Rocha and Moreno [37, 38]. It is implemented in the C language and must be capable of reclaiming memory.

Our main contributions are as follows:

- The rank of concurrent hash map designs, in Fig. 2.2.
- The compression algorithm, in section 3.5.2.
- The memory reclamation method support for the compression mechanism, in chapter 4.
- Section 5.4 measures the compression mechanisms, now in the C language. In 2021, Areias and Rocha [37] ran similar tests implemented in Java. We compare the results of both tests.
- Section 5.5 compares different reclamation methods, applied to the data structure.

In what follows, chapter 2 summarizes the literature related to our work. In chapter 3, we describe the algorithms for every major operation of the map and analyze the asymptotic cost in memory consumption and execution time. Next, in chapter 4, we propose methods for reclaiming memory in the absence of garbage collectors. Chapter 5 is where we compare the memory footprint and running time of the newly introduced algorithms against the original data structure. Finally, chapter 6 contains a summary of the document as well as proposals for further work.

Chapter 2

Background

Lock-free data structures have a number of advantages over lock based ones. On one hand, management of locks is unnecessary which prevents deadlocks and livelocks. Furthermore, lock-free data structures are non-blocking meaning that the suspension of one thread won't prevent others from progressing, making them suitable for asynchronous systems and real-time applications, as demonstrated by Herlihy [5, 23].

Without locks, we can reduce the amount of context switches and waiting queues. In particular, context switches force a processor pipeline to flush, reload TLB entries, save processor registers, and force the OS scheduler to execute. Context switches also degrade cache performance, whose lines are invalidated by other threads, as demonstrated by Mogul and Borg [6] and Li et al. [20]

The main drawback of lockless programming is that it is generally more difficult to design algorithms and debug. Because there are no critical sections in a lock-free program, when a thread resumes execution after preemption by the OS scheduler, the state of the shared object may have changed in the meantime. In fact, the shared object may change at any point in time. To aggravate this issue, compilers and CPUs reorder instructions presenting yet another challenge. Instruction reordering is intended to reduce single threaded execution times, but results in inconsistent concurrent access to a shared object. Thus, when synchronizing threads, it may be necessary to force a specific order of instructions to solve inconsistent access through the use of *memory barriers* and *synchronization primitives*. Knowing when and what barriers to use presents another challenge to lockless programming. McKenney [31] compiled a list of information regarding computer architectures and parallel programming.

One of the most used synchronization primitives in lock-free programming is the compare-and-swap (CAS) instruction, supported by most modern CPUs, having first appeared in the IBM System/370 architecture [3]. Algorithm 1 shows the implicit sequence of instructions executed atomically by a CAS operation.

Algorithm 1 Compare-and-swap($address\ M, value\ E, value\ N$)

1: **if** $Value(M) = E$ **then**
2: $\quad Value(M) \leftarrow N$
3: $\quad$ **return** True
4: **return** False

The CAS operation is usually used in a loop as a way to allow for long transactions that span multiple instructions. We first observe the state of the shared object and write all changes to local memory. Then, one single CAS instruction commits such changes to shared memory, but only if the global state is still coherent with what we initially observed. If the CAS fails, we try the same procedure again.

There are two main issues with CAS-based lock-free implementations. First, the memory reclamation problem. After removing data from a shared data structure we may want to free the memory allocated for such data. The process of freeing memory is called *memory reclamation*. However, in lock-free programming, it is possible that other threads still hold a reference to this data even after we remove it from the shared data structure. If we free it directly, we risk a use-after-free by this other thread. To solve this issue, we need to design a mechanism for detecting when memory blocks are no longer referenced by any thread so that they may be reclaimed. In a lock based design, we prevent multiple threads from using the same memory references which makes this a non issue.

Secondly, the ABA problem. A successful CAS instruction will change the state of the shared data structure. If a CAS instruction changes a value A to a value B, the shared data structure transitioned to a second state. Changing the value again from B to A will change the shared data structure to a third state. Although the shared data structure is at the third state, a CAS instruction may assume the data structure is still at its first state, because it observes the initial value: A. Another thread may execute a CAS changing, for example, A to C without taking into consideration the changes made from the first to the third state. This problem can break the semantics of an algorithm. Therefore, we must design a lock-free algorithm with the ABA problem in mind. Typically, a solution to the memory reclamation problem is sufficient to solve the ABA problem because removed

objects will never have their memory reclaimed when used by multiple threads and, thus, the change of value from B to A is impossible.

2.1 Hash tries

A *trie*, as described by Bentley, Knuth and McIlroy [4], represents a set of words and all prefixes of those words. The method for searching tries uses characters of a string in succession to select a direction in each level of a *k-ary* tree hierarchy. Tries have been used in IP routing protocols [8], peer-to-peer distributed hash tables [12] and scanners for language parsers [4].

The trie data structure was first introduced in 1959 [1] as a string searching data structure, baptized by Fredkin [2] (*trie* as in, "information re**trie**val") and later implemented by Bentley et al. [7], Nilsson et al. [8], and Bagwell [9]. In particular, Bagwell's implementation, the *array mapped tries*, demonstrated higher performance and space efficiency. Later, Bagwell [10] extended it to support bit strings as opposed to character strings and renamed it to *hash array mapped tries*, which our own implementation is based on. The name *hash trie map* is used as a synonym of hash array mapped trie, throughout the literature.

Figure 2.1 illustrates a hash trie map. The intermediate nodes that form a trie are called *hash nodes* (H_i), each having a fixed number of buckets (B_i). *Leaf nodes* (K_i) contain the key-value pairs.

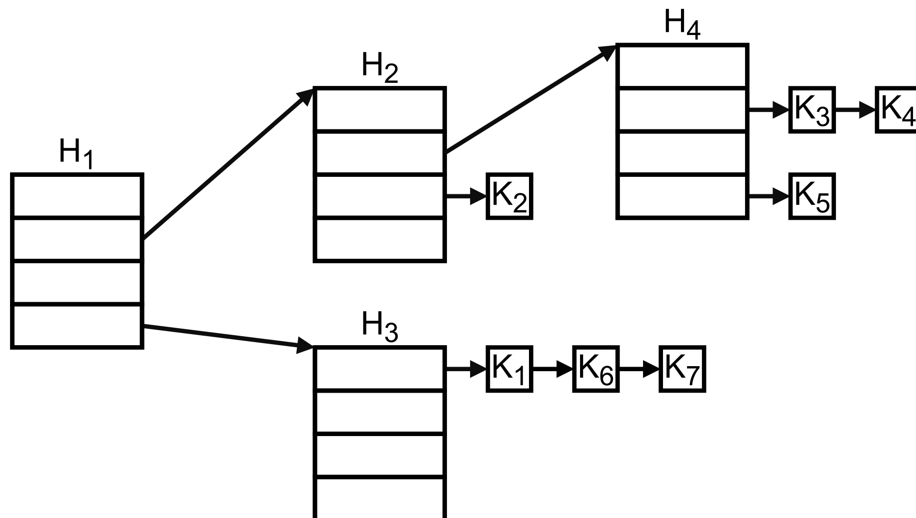


FIGURE 2.1: Hash trie map

As keys are inserted on the hash map, hashes collide forming linked lists of keys. Consequently, it will take more iterations to search for keys on the map, but we can expand

the map in size in order to reduce hash collisions. A classic hash map design will use a single monolithic array to store all keys, and, to expand it, we need to allocate a bigger array and move every key, one by one, from the old array. However, the bigger the hash map, the longer it will take to expand. An expansion will force all threads to cooperate on this potentially long operation, hindering their progress. On the other hand, on a *hash trie map*, hash nodes have a fixed size allowing expansion to finish in constant time, as stated by Bagwell [9]. To expand, we add a new fixed size hash node in front of a collision chain, and move all nodes of the chain to this new hash node, one by one. Moreover, because expansion will not involve all nodes of the map, only those of a collision chain, threads can operate on other locations of the map independently without obstruction.

In the context of lockless programming, Harris [11] introduced the first lock-free linked list based on the single width CAS operation which was improved upon by Michael [13] to support lock-free memory management methods with bounded memory. Michael also proposed a lock-free hash table using the aforementioned lists as collision chains. A classic single array hash table design is used, not a hash trie map. The proposed design, thus, does not support expansion, because it would hinder progress guarantees. The author [15] later added support for safe memory reclamation in environments without garbage collection, using hazard pointers. Then, Shavit and Shalev [19] added support for table expansion.

Prokopec et al. [25] presented a lock-free version of Bagwell's *hash array mapped trie*, using single-width CAS, called *Ctries*. Ctries will only allocate memory for populated buckets to prevent empty buckets from taking space. To add or remove nodes: we clone the contents of a hash node to local memory, apply all changes to the local copy, and replace the original with the cloned hash node in one CAS instruction. Not only is recursive helping unnecessary, but the number of synchronizations is low compared to our own design. Moreover, map operations are limited to one single synchronization instruction making it easier to verify algorithm correctness. The main disadvantage is the overhead caused by the frequent cloning of hash nodes. Ctries also support lock-free snapshots [25]. On a later date, Prokopek [35] modified this design so as to run operations in expected constant time, naming this newly designed data structure: Cache Tries. Cache Tries use an auxiliar data structure other than the hash map in order to speed up map traversal and other operations.

Areias and Rocha [30] formulated a lock-free version of Bagwell's hash trie map. Like

with Ctries, the proposed design is also capable of performing lock-free expansion, to reduce collisions. Unlike with Ctries, the CAS instructions are applied directly onto each bucket as opposed to replacing an entire hash node. Then, the authors [32] added support for sorted keys and, after that, compression of hash nodes [37]. A custom memory reclamation method with little overhead was proposed by Moreno et al. [38] This is the data structure we study throughout this document.

2.2 State-of-the-Art Comparison

There have been many concurrent hash map designs over the years, but how do they compare in performance?

In Fig. 2.2, each implementation is ordered by relative performance in execution time. A relation $A \xrightarrow{P} B$ means that B was reported to outperform A, in paper P. The citation to the paper is attached to the arrow. The only paper with artifact checks is the one on *cache tries* [35].

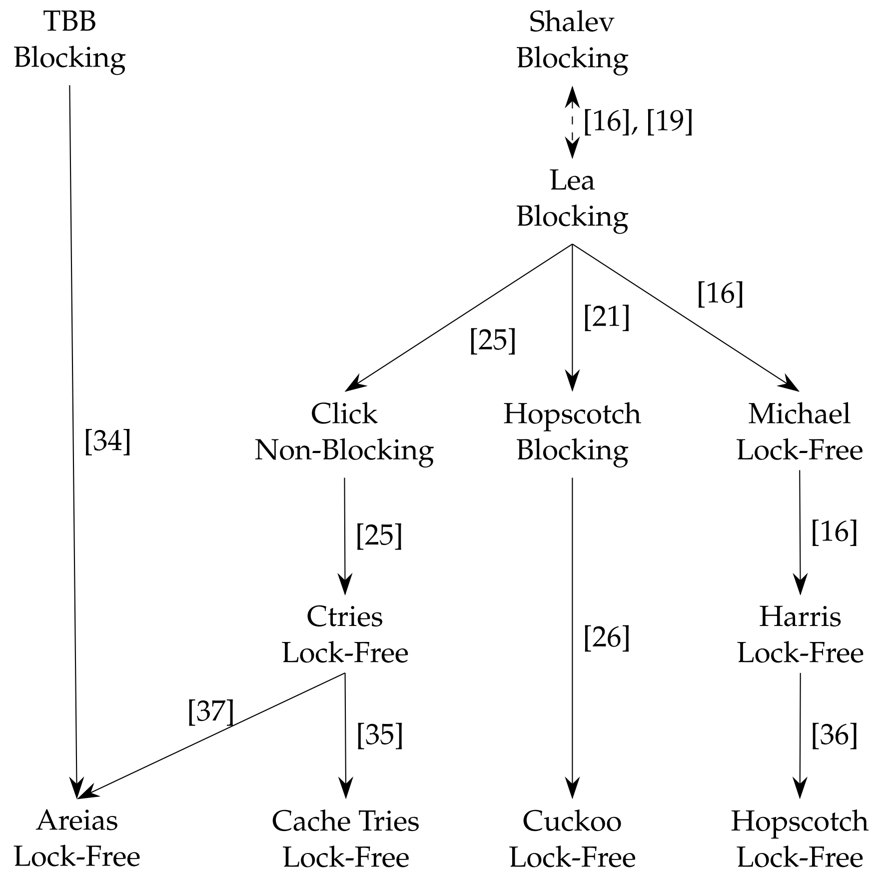


FIGURE 2.2: Comparison of state-of-the-art designs

As we will see in chapter 5.5, implementation decisions such as memory reclamation methods and cache-awareness can turn what was previously a costly design into a faster implementation. Notably, the *Harris Map* [16] data structure does not require any memory reclamation method in order to free memory in an unmanaged language. The *Java Virtual Machine* (JVM) applies the same memory reclamation method to all programs. One of these designs implemented for use with the JVM could outperform the Harris map with a different reclamation method. Thus, it is not always clear which design is faster, as it may depend heavily on the execution environment. The dashed line between Shavit and Lea illustrates this same issue. In Shalev and Shavit [19], the authors demonstrate how the new design outperforming Lea's, running in the JVM. Later, Purcell and Harris reimplemented both solutions in C, using epoch based memory reclamation, which resulted in Lea's design being faster.

TBB, Lea, Click, Shalev, Michael and Harris use the classic linear hashing approach. Ctries, Cache tries, and Areias designs are based on Bagwell's hash array mapped tries, which uses partition hashing. Cuckoo uses cuckoo hashing and Hopscotch mixes cuckoo hashing, chaining and linear probing.

Although lock-freedom guarantees progress, some threads may still be stuck in a CAS-loop obstructed by a series of conflicting operations. A common technique to mitigate this issue is having obstructed threads assist others on completing their tasks. This is called *recursive helping* by Fraser [14], and by Herlihy and Shavit [23]. A non-blocking algorithm in which there is no obstruction of threads is called *wait-free*. A wait-free data structure, according to Herlihy and Shavit [23], is a stronger progress guarantee than lock-freedom, in which every thread makes progress in a finite amount of steps. We have included the recursive helping technique in our new hash map design. However, we have not validated whether or not the data structure is wait-free as a whole.

Chapter 3

Lock-free Hash Trie Map

A lock-free hash trie map (LFHT) is accessed through the functions: **INSERT**, **REMOVE** and **LOOKUP** while the expansion and compression functions are never called directly by a user. This chapter is dedicated to the analysis of each function: **INSERT**, **REMOVE**, **LOOKUP**, **EXPAND** and **COMPRESS**. Each algorithm is explained before the analysis of its asymptotic cost. Two alternative strategies are presented for the compression mechanism.

In a LFHT, there are two types of nodes: leaf and hash nodes. Leaf nodes contain key-value pairs whereas hash nodes are used for traversal in a trie like manner. Each hash node contains a header and an array of buckets, of *fixed size* 2^W. The chunk size (W) determines the number of bucket entries of each and every hash node as well as the number of bits extracted from a hash string for indexing purposes. The chunk of bits extracted from a hash string will be used to select the appropriate bucket of a hash node. Additionally to an indicator of the current tree level, all headers may contain a reference pointing back to their parent hash node. Further, headers may have additional fields depending on the implementation.

3.1 Lookup

To traverse the LFHT, subsequent chunks of a hash are used in a hashing strategy called *partitioned hashing*. This is achieved through bit-wise shifts and masks. The traversal is done depth-first so each hash node we advance to belongs to a deeper level of the hierarchy. The level *field* of the header selects the correct chunk from the hash which is, in

turn, used to index the hash node's bucket array. An empty bucket points back to its own hash node.

In Fig. 3.1, the **LOOKUP** function is demonstrated for two different map configurations. A small hash size is used for simplification. If the total length of the hash in bits is not divisible by the chunk size, the last chunk will be zero-filled. Note, however, that the hierarchy will only be built after expansion to reduce collisions of keys.

Every map operation depends on the **LOOKUP** function. To add a node, we first need to check if it is already present, so we call **LOOKUP**. The same is done when removing or finding a node. Thus, the performance of this operation has a significant impact in all others.

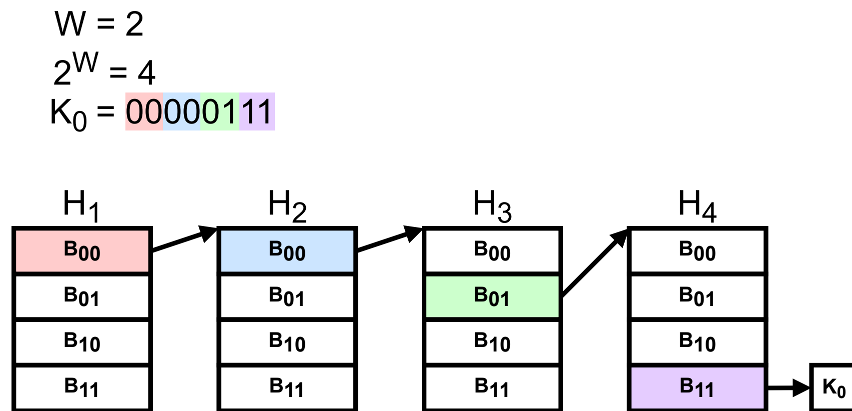
3.1.1 Average Path Length

A crucial part in the analysis of the **LOOKUP** function is understanding the impact of the *average path length*. Traversing the tree requires a number of successive *hops* before reaching the leaves. Moreover, as we will see in the next section, leaves can form linked lists which will further increase the number of hops.

The metric of average path length denotes how many hops a **LOOKUP** function performed on average. A high average path length can be one of the main causes for slowdowns. Conversely, reducing the average path length will lead to positive performance results as will be shown and discussed in later experiments. However, memory accesses with high latency due to synchronization are the biggest issue when it comes to performance in throughput, as we will explore in greater detail later.

3.1.2 Algorithm

The **LOOKUP**(H, h) function, as illustrated by Alg. 2, traverses the tree starting from the hash node H and returns the node associated with the hash h, if it exists. The traversal begins at the root of the tree hierarchy. The chunk size (W) is a preemptively configured global variable and does not change at runtime. The array notation $H[i]$ denotes the content of the ith bucket of hash node H. This bucket may either point to a leaf node, to another hash node, or, if it's empty, back to hash node H. The function returns: *iter*, which is the node with hash h, or *nil* if it is not present on the map; the parent hash node H of the collision chain where the key-value pair would be inserted; the node before *iter*,



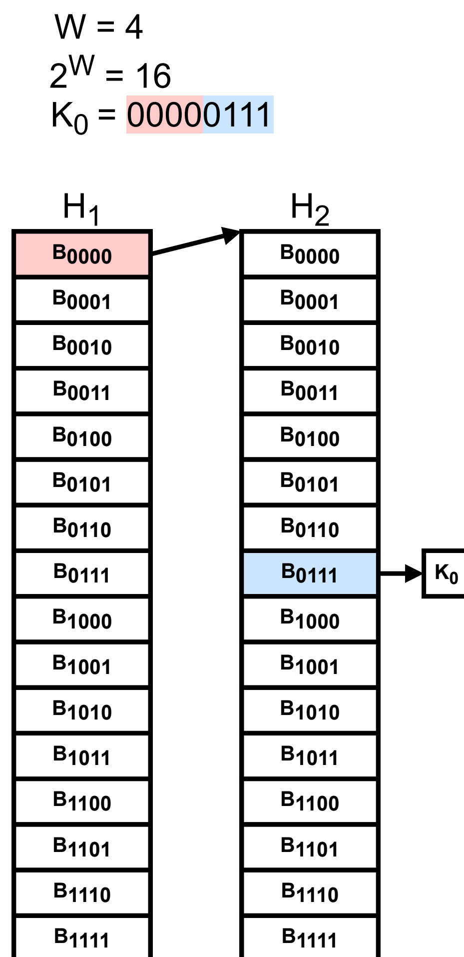


FIGURE 3.1: Lookup demonstration for different map configurations

or the last node of the collision chain if *iter* is not present; the length of the collision chain, up until *iter*.

Algorithm 2 LOOKUP($HashNode\ H, KeyHash\ h$)

```
1:  i ← GetChunk(h, H.level, W)
2:  iter ← H[i]
3:  if iter is a freeze/unfreeze node then
4:      iter ← iter.next
5:  prev ← H
6:  len ← 0
7:  while iter ≠ H do
8:      if iter is a hash node then
9:          return Lookup(iter, h)
10:     len ← len + 1
11:     nxt ← iter.next
12:     if iter is valid then
13:         if iter.hash = h then
14:             return ⟨iter, H, prev, len⟩
15:         prev ← iter
16:     iter ← nxt
17: return ⟨nil, H, prev, len⟩
```

From lines 1-2, the bucket is selected using bit-wise shifts and masks. The chunk of W bits is selected from the hash string h, which indexes the bucket array of our current hash node H.

Lines 3-4 are only necessary when implementing the compression mechanism using *freeze* nodes. Two adjacent hash nodes may have a freeze node in between. This node is skipped at line 4 when traversing the tree.

During traversal, we keep track of additional variables necessary for the insertion and removal functions. The *prev* variable keeps track of the node directly before our iterator. The *prev* variable will have the value of the last node of a collision chain, if the key we are looking for (h) is not present on the map. *len* keeps track of the length of the collision chain, which is useful, for example, when keeping track of hazard pointers.

At the beginning of the loop at line 7, *prev* points to the parent hash node, while our iterator *iter* points to a leaf node in bucket i. The loop terminates when our iterator points back to its parent hash node.

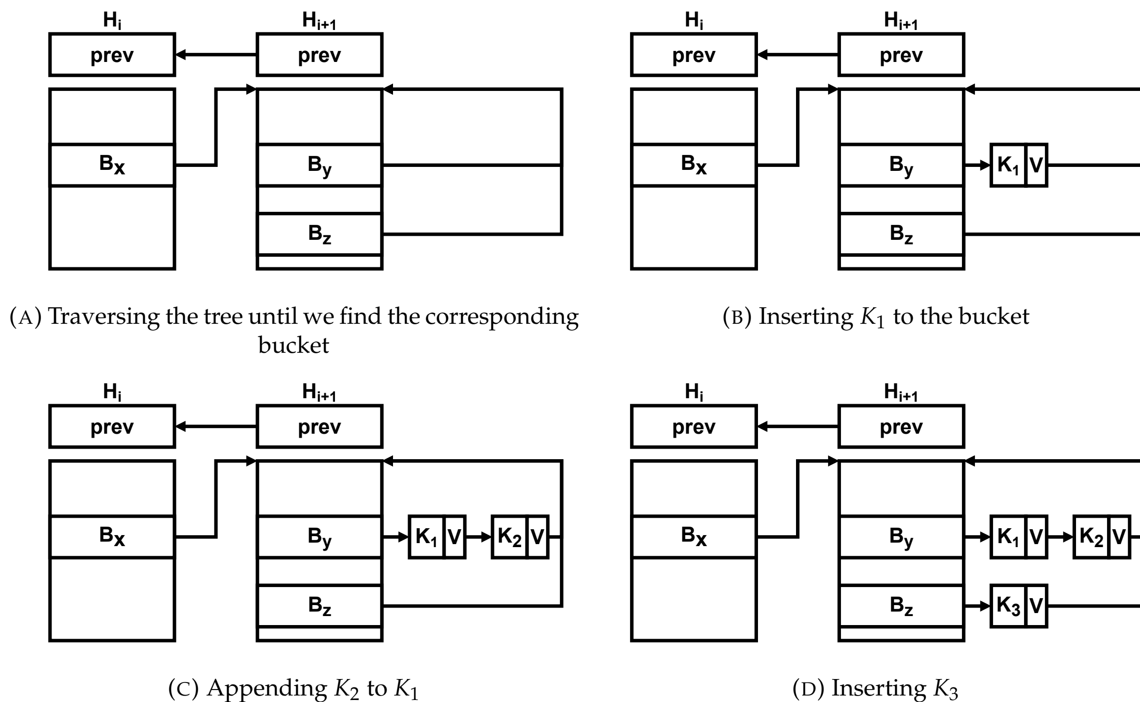
On lines 8-9, we move to a new hash node down the tree hierarchy. The remaining lines of the loop, 10-16 are dedicated to verifying our iterator and advancing to the next node on the collision chain. We do not keep track of invalid nodes (line 12). If *iter* has

the hash we were looking for (h), the **LOOKUP** function returns its value. If the node we were looking for does not exist, we reach the end of the collision chain and return at line 17.

3.2 Insertion

To add a new key-value pair we first need to locate the target bucket with **LOOKUP**. Then, the reference to the new leaf node is written directly to the bucket through the use of compare-and-swap, as demonstrated in Fig. 3.2. Again, if this CAS operation fails, the process is restarted.

The field V will be mentioned and explained in section 3.4, which documents the **REMOVE** function.



(A) Traversing the tree until we find the corresponding bucket

(B) Inserting K_1 to the bucket

(C) Appending K_2 to K_1

(D) Inserting K_3

FIGURE 3.2: Inserting nodes

This version does not support updates of already inserted keys. Thus, if the target key is already inserted, the operation will return the already inserted value.

3.2.1 Dealing With Collisions

Leaves will collide when inserting in an already populated bucket. Collisions are tolerated to a certain degree by chaining multiple leaves together in a Harris [11] singly linked

list. The **LOOKUP** function will traverse collision chains until the tail end is reached to which we may append the new nodes. The nodes at the tail end of the list point back to their parent hash node. As the number of inserted nodes increases, however, so do the number of collisions and so does the average path length. Section 3.3 explains how to mitigate this issue.

If chains increase the average path length, why tolerate collisions in the first place? In short, expansion comes with a cost. Expanding the map right as two hashes collide would increase the overhead of the average insert operation, as demonstrated in later section 5.3.1. In fact, some versions of the hash trie map do not tolerate collisions such as Prokopek's *ctries* [25] and *cache tries* [35] (although newer versions do support them [25]).

3.2.2 Algorithm

The **INSERT**(H, h, v) function, illustrated by Alg. 3, appends the key-value pair $\langle h, v\rangle$ to the tail end of the tree, starting from hash node H.

Algorithm 3 INSERT($HashNode\ H, KeyHash\ h, Value\ v$)

1: $\langle node, H, prev, len\rangle \leftarrow$ LOOKUP(H, h)
2: **if** $node \neq nil$ **then**
3:     **return**
4: **if** $prev$ is freeze/unfreeze node and UNFREEZE(H, h) fails **then**
5:     **return** INSERT$(H.prev, h, v)$
6: **if** $len = EXPANSION_THRESHOLD$ **then**
7:     $H \leftarrow$ EXPAND$(H, h, prev)$
8:     **return** INSERT(H, h, v)
9: $N \leftarrow$ NEWLEAFNODE(H, h, v)
10: **if** CAS$(prev.next, H, N)$ fails **then**
11:     **return** INSERT(H, h, v)

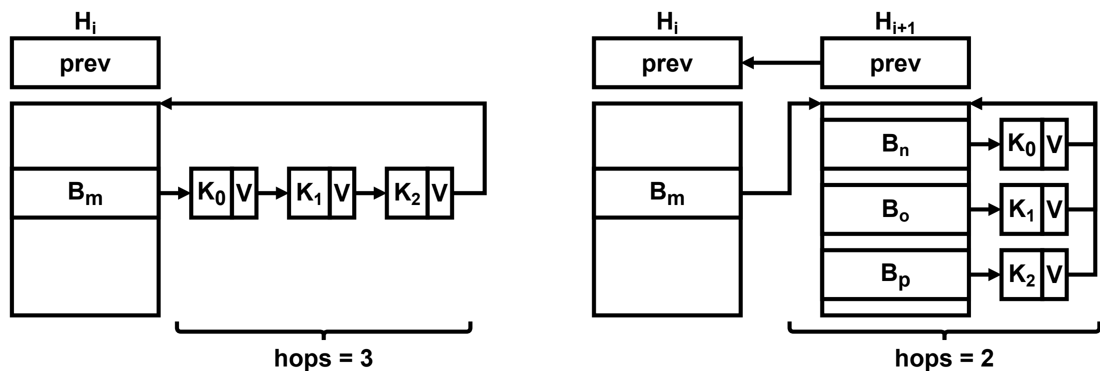
The **LOOKUP** call at line 1 has multiple purposes. The first is to detect whether a leaf node with h is already inserted (lines 2-3). The second is to get the tail end of the target collision chain, i.e. the one we want to append the new node to.

Lines 4-5 make use of *prev*, also returned by **LOOKUP**, which is the node before the tail. Lines 4-5 deal with the compression mechanism discussed later on in this chapter. If *prev* is a *compression node*, then we try to cancel the ongoing compression with the **UNFREEZE** function. This is required for compatibility with the compression operation, which we will explain in section 3.5.1. Finally, the length of the target collision chain, *len*, is later used in lines 6-8 to trigger an expansion, if the expansion threshold was reached.

The remaining lines create a new leaf node, N, and an attempt is made at appending this node to the collision chain. If the attempt fails, we retry the **INSERT** function from the start.

3.3 Expansion

As nodes are inserted, collision chains get longer, increasing the average number of hops performed per **LOOKUP**, i.e. average path length. To reduce the average path length, nodes of a chain are distributed through a new hash node. This process is called expansion. The advantage of expansion is demonstrated by Fig. 3.3. Note, however, that the distribution of nodes is not necessarily even. In the average case, the number of hops decreases, but nodes may still collide when expanding to a new hash node. This does not prove to be advantageous when reaching for K_0 and K_1. However, if we do not impose a maximum length to collision chains, the average path length would be bigger.



(A) The longer the collision chain, the larger the average path length

(B) New levels reduce the average length of collision chains

FIGURE 3.3: Expanding to reduce the average path length

To uphold the *lock free* property, the expansion process must discard locks completely and we must take into consideration that other threads can interfere with the procedure at any point. Figure 3.4 illustrates the whole process. The expansion procedure will start due to an insertion of a key in an already saturated collision chain. Let's assume our chain threshold is 3. The expansion procedure starts by appending a new hash node to the tail of the collision chain. From here on out, threads will travel down the chain and insert new nodes to the new hash node, H_{i+1}, instead of appending nodes to the chain.

To complete the expansion of a collision chain, we move each leaf node of the collision chain to the new hash node. This process is called *rehashing*, and is illustrated in Fig. 3.4c-3.4e. Nodes of a collision chain must be rehashed in reverse order, i.e. from the tail to the head. If we rehash node K_0 before K_1 in Fig. 3.4c, K_1 will be temporarily unreachable from the hash map. Because this is a lock-free algorithm, threads could bump against this collision chain at any point in time, even during an expansion. Thus, in order to guarantee the correctness of the algorithm, we must make sure nodes are reachable at all times and, to do so during an expansion, we must rehash chains from the tail to the head.

3.3.1 Algorithm

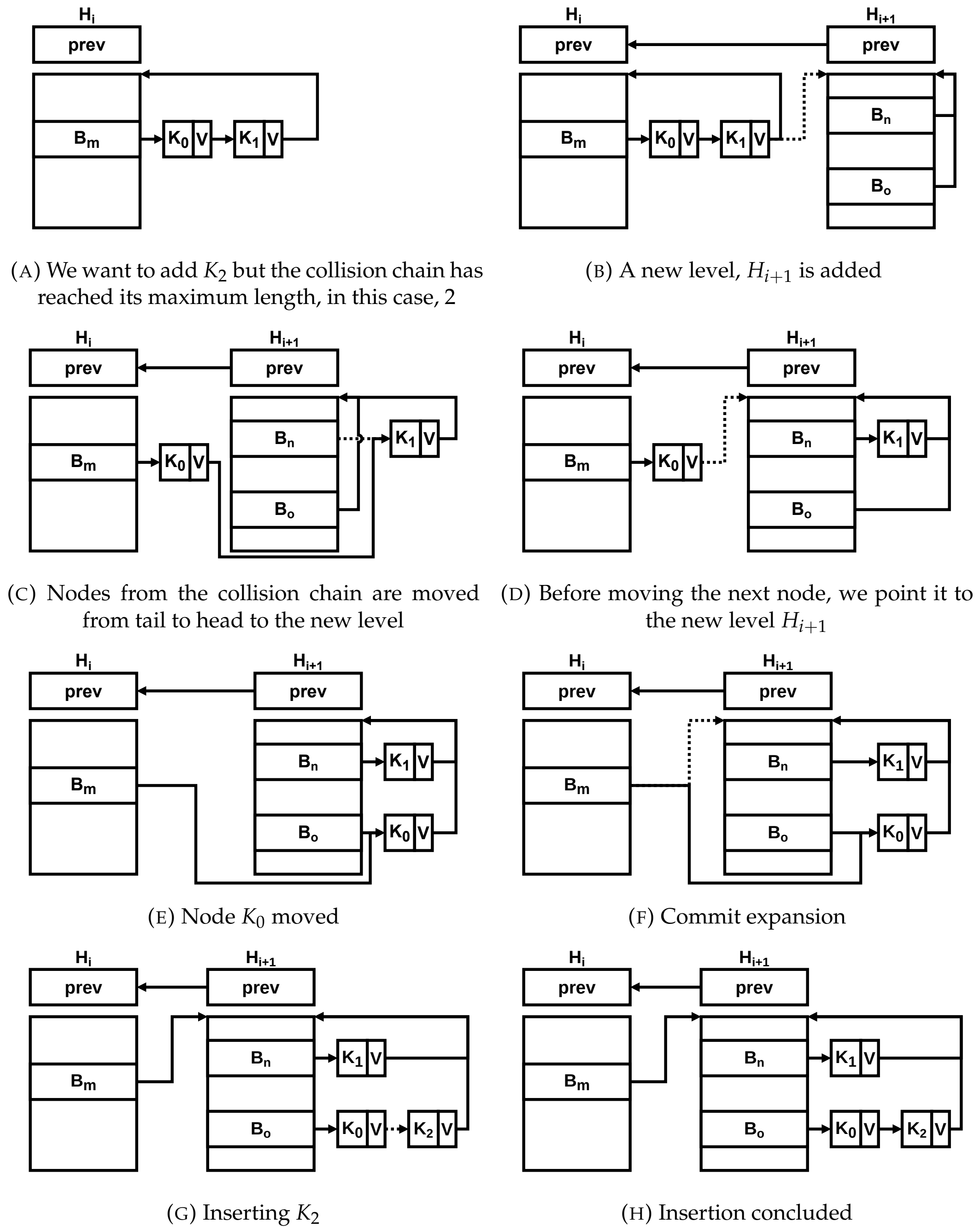
The **EXPAND**$(H, h, tail)$ function, illustrated in Alg. 4, adds a new hash node to a path, in front of H. The new hash node H' will be appended to the tail end of the chain, $tail$, such that all nodes of the chain remain visible throughout the expansion process. Finally, each leaf node of the chain will be adjusted to the new level.

Algorithm 4 EXPAND(*HashNode H*, *KeyHash h*, *Value tail*)

1: $H' \leftarrow$ CREATENEWHASHNODE(H)
2: **if** CAS($tail.next, H, H'$) **then**
3:     **return** HELPEXPAND(H, h, H')
4: $seen \leftarrow tail.next$
5: **if** $seen$ is not a hash node or $seen = H$ or $seen$ is invalid **then**
6:     **return** H
7: **return** HELPEXPAND($H, h, seen$)

If the CAS at line 2 is successful, the new hash node H' was appended to the target collision chain. The **HELPEXPAND** function (lines 3 and 7) will perform the expansion of the nodes on the collision chain to the new hash node (H'). The chunk size (W) is a preemptively configured global variable and does not change at runtime. If the CAS at line 2 fails, it must be because some other thread tempered with the tail of the chain. In line 4 we load the reference next to the tail and proceed to verify whether or not it is a new hash node, at line 5. Line 6 will occur if the conditions for a new expansion have not been fulfilled. Line 7 starts the **HELPEXPAND** procedure with the new hash node, *seen*, created by another thread. Threads that reach line 7 will help the expansion started by the thread whose CAS at line 2 was successful.

First, we load the head of the collision chain in lines 1-2. If *head* is not a leaf node, then it is either a hash node or a compression node. If *head* is not a leaf node, the expansion



(A) We want to add K_2 but the collision chain has reached its maximum length, in this case, 2

(B) A new level, H_{i+1} is added

(C) Nodes from the collision chain are moved from tail to head to the new level

(D) Before moving the next node, we point it to the new level H_{i+1}

(E) Node K_0 moved

(F) Commit expansion

(G) Inserting K_2

(H) Insertion concluded

FIGURE 3.4: Demonstration of the complete expansion of a collision chain

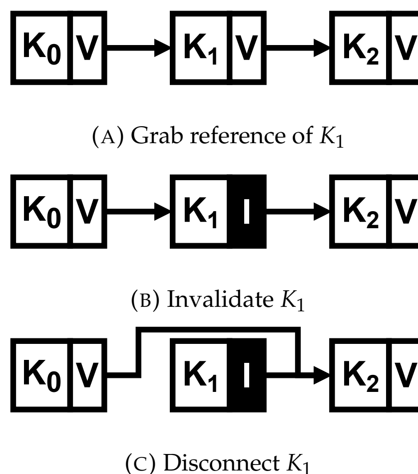
process must have already terminated (lines 3-4). Line 5 is where each leaf node of the collision chain, from tail to head, one by one, is moved to the new hash node H'. After **REHASHNODES** terminates, the bucket $H[i]$ points to the *head* of the collision chain we expanded. To commit the expansion, we set the bucket's reference to the new hash node H' (line 6).

Algorithm 5 HELPEXPAND(*HashNode H*, *KeyHash h*, *HashNode H'*)

1: $i \leftarrow$ GETCHUNK$(h, H.level, W)$
2: $head \leftarrow H[i]$
3: **if** *head* is not a leaf node **then**
4:     **return** H
5: REHASHNODES$(H', head)$
6: **if** CAS$(H[i], head, H')$ **then**
7:     **return** H'
8: **return** H

3.4 Removal

To remove a node, we must first locate it, again, using the **LOOKUP** function. Harris [11] designed the removal operation in a lock free list as a two step process. First, we mark the target node as *invalid* (represented by the I field in Fig. 3.5) as a way to warn other threads of its eventual removal. Only then can we disconnect it from the list, as illustrated in Fig. 3.5, effectively making it *unreachable*.



(A) Grab reference of K_1

(B) Invalidate K_1

(C) Disconnect K_1

FIGURE 3.5: Removing node K_1

Why is the first step necessary? Is there an issue with detaching the node, right away? The remainder of this section answers these questions.

3.4.1 Invalidation Step

The first step is necessary in order to prevent insertions of new nodes in front of already removed ones. Consider the following history of conflicting operations, illustrated by Fig. 3.6:

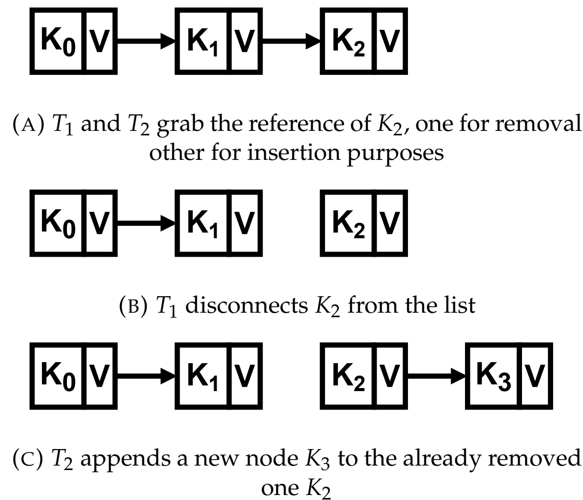
1. Thread T_1 wants to remove K_2 from the list;

2. Thread T_2 wants to add node K_3 after the list's tail, K_2;

3. Thread T_1 disconnects K_2 from the list;

4. Thread T_2 adds K_3 in front of K_2.

As a result, the new node is also disconnected from the list.



(A) T_1 and T_2 grab the reference of K_2, one for removal other for insertion purposes

(B) T_1 disconnects K_2 from the list

(C) T_2 appends a new node K_3 to the already removed one K_2

FIGURE 3.6: Conflicting operations when removing a node

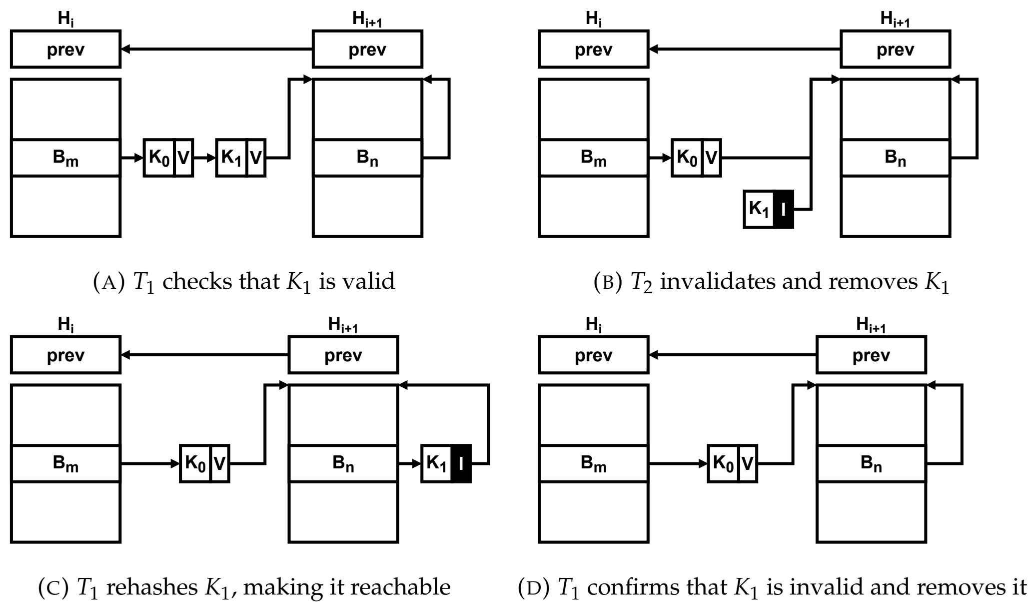
Likewise, removing two adjacent nodes may also cause one of the removed nodes to be reinserted due to a race.

To prevent any further insertions, removed nodes must be made immutable. One possible way of implementing this is to set the lowest significant bit of the *.next* field, forcing foreign CAS operations to fail. Since allocated structures are memory aligned, the lowest significant bit of a pointer is always zero. The *.next* field of an invalid node is still addressable, nonetheless, but cannot be overwritten.

Any CAS instruction applied to an immutable field will fail, making invalid nodes suitable for removal without conflicts. When encountering invalid nodes, threads must either skip them by traversing to the next neighbor or attempt to remove them (helping). The latter case is only necessary in some cases. For example, as will be mentioned later on, when implementing a *hazard pointer* based memory reclamation method.

3.4.2 Memory Reclamation Problem

Even after a node is made unreachable, it may still have its reference visible to other threads. Because of this, we are unable to free its memory after removal, unlike with lock



(A) T_1 checks that K_1 is valid

(B) T_2 invalidates and removes K_1

(C) T_1 rehashes K_1, making it reachable

(D) T_1 confirms that K_1 is invalid and removes it

FIGURE 3.7: The delegation problem

based implementations. The act of safely freeing memory is called memory reclamation. This was not a problem in a garbage collected environment such as the JVM which has its own ways of detecting exactly when an object can be freed. When implementing the LFHT in an unmanaged language, such as C, we must implement our own memory reclamation mechanism, which we will discuss in great length in Chapter 4.

3.4.3 Delegation Problem

During the expansion procedure, nodes of a collision chain are moved to a new hash node, one by one, from tail to head. But what if nodes are removed from the collision chain during an expansion? An ongoing expansion will check if a node is invalid before moving it to a newer level. However, there is a race condition in which the node will be invalidated and removed right after it being confirmed valid, by the expanding thread, and right before it is moved by an expansion. We call this the *delegation problem*. This problem is illustrated in Fig. 3.7.

One of the early solutions to this problem forced threads which were performing expansions to double check whether the moved nodes were valid or not. These nodes would then be removed a second time from the map, as illustrated by Fig. 3.7d. The name delegation refers to the fact that the removal of a node during an expansion would be delegated to the thread responsible for expansion. Hence, the name, *delegation problem*. However,

this solution will lead to concurrency hazards when trying to implement a memory reclamation method. We will look at this problem's impact on memory reclamation in a later chapter, and propose another solution.

3.4.4 Algorithm

The **REMOVE**(H, h) function, illustrated by Alg. 6, removes the node with hash h from the tree and attempts to compress its parent hash node.

Algorithm 6 REMOVE($HashNode\ H, KeyHash\ h$)

1: $\langle node, prev, len \rangle \leftarrow$ LOOKUP(H, h)
2: **if** $node = nil$ **then**
3:     **return**
4: **if** *node* is valid and *node.next* is a hash node and $node.next \neq H$ **then**
5:     $H \leftarrow$ HELPEXPAND$(H, h, node.next)$
6:     **return** REMOVE(H, h)
7: **if** TRYINVALIDATE$(node)$ fails and *node* is valid **then**
8:     **return** REMOVE(H, h)
9: **if** CAS$(prev, node, node.next)$ succeeds **then**
10:     COMPRESS(H, h)

The first line searches for the node matching the target hash h, the one we wish to remove. If the node does not exist we return from the function (lines 2-3).

Lines 4-6 prevent the *delegation problem*. The **TRYINVALIDATE** function at line 7 will perform a CAS to the *node* in order to invalidate it. Therefore, we try to set the least significant bit of its *.next* field to 1. This masked pointer will still point to the same node in memory. If the node invalidation step fails and the node is still valid, it must mean that the next neighbor of our target *node* changed in the meantime and we must retry the removal operation from the beginning (lines 7-8).

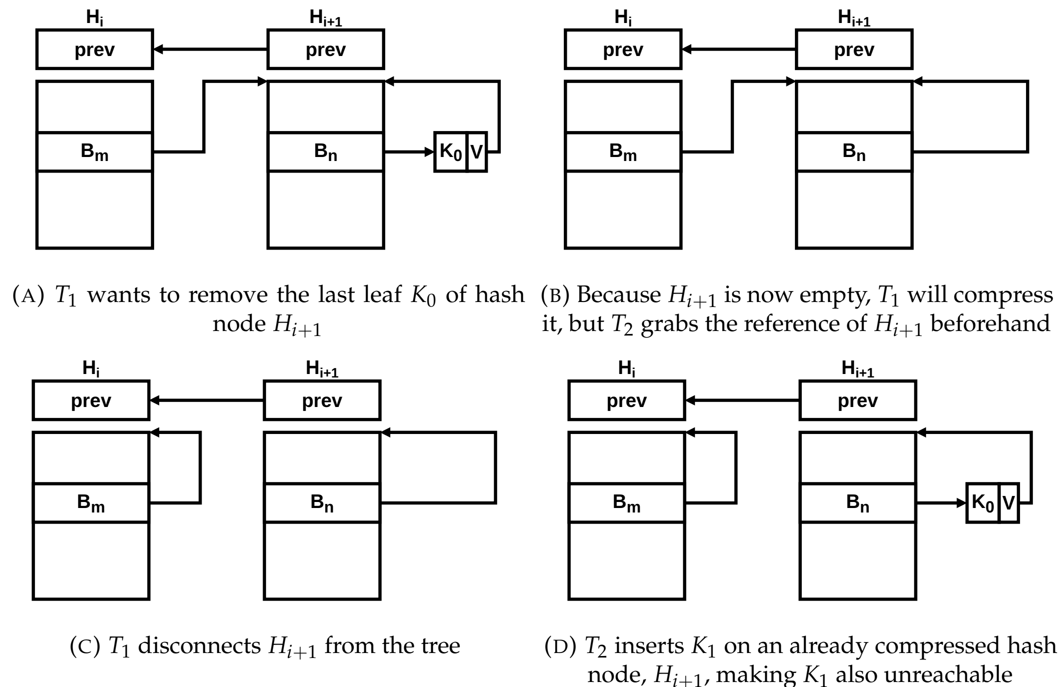
Note how multiple threads may try to remove the same node after it has been invalidated. Whoever can detach it from the collision chain at line 9 is the one responsible for compressing the hash node (line 10). The compression operation is discussed in the next few sections.

3.5 Compression

After the removal of leaf nodes, a hash node may become empty. In that case, the hash node is not only taking up unnecessary space in memory but also adding an additional

level on the tree hierarchy, causing longer **LOOKUP** times.

Compression is the mechanism for removing hash nodes. However, this process is not as simple as detaching the hash node in one step, since other threads may attempt to insert keys to the already detached hash node. Figure 3.8 illustrates a history of conflicting instructions which we need to take into consideration when designing the lock-free compression algorithm.



(A) T_1 wants to remove the last leaf K_0 of hash node H_{i+1}

(B) Because H_{i+1} is now empty, T_1 will compress it, but T_2 grabs the reference of H_{i+1} beforehand

(C) T_1 disconnects H_{i+1} from the tree

(D) T_2 inserts K_1 on an already compressed hash node, H_{i+1}, making K_1 also unreachable

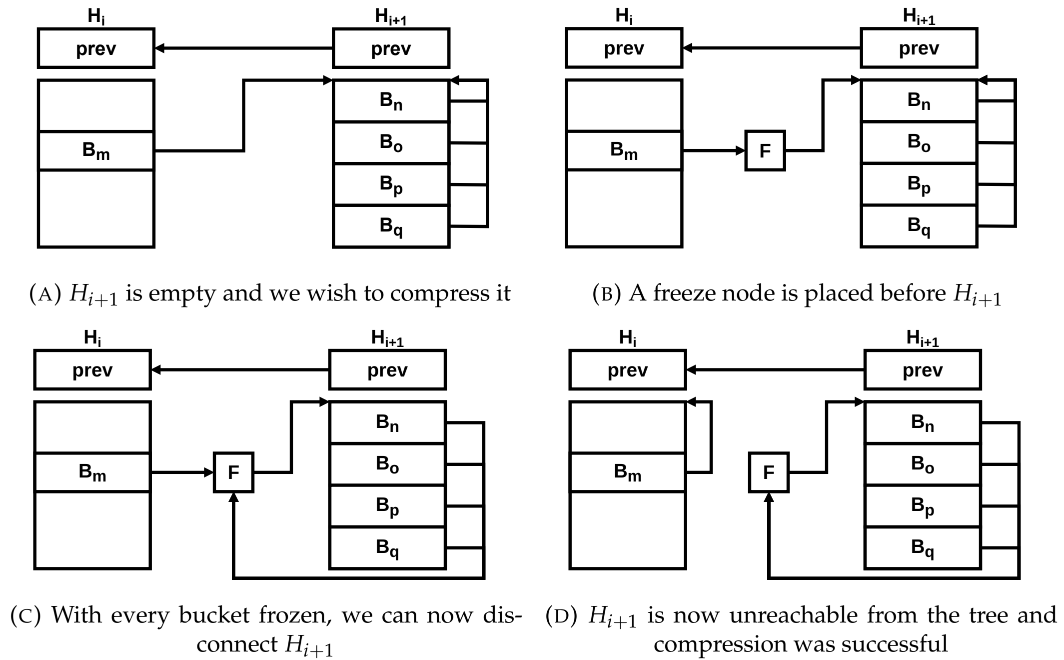
FIGURE 3.8: Conflicting operations when compressing a hash node

This problem is identical to the one explored in the removal operation. Thus, a possible solution is to mark the hash node as immutable, preventing further modifications. Unlike with leaf nodes, hash nodes have multiple bucket entries which may be overwritten at any point in time. The next sections explore different ways to issue a hash node for removal.

3.5.1 Freezing

In section 3.4, we discussed the removal operation which is capable of marking a leaf node as invalid, effectively making it immutable. We can borrow this concept to mark each bucket of the hash node, one by one, as immutable with a series of CAS calls. In the following example, Fig. 3.9, one by one, each bucket is redirected to a special *freeze*

type node. Once neighbor threads detect this node, they should skip it and attempt to complete their work concurrently.



(A) H_{i+1} is empty and we wish to compress it

(B) A freeze node is placed before H_{i+1}

(C) With every bucket frozen, we can now disconnect H_{i+1}

(D) H_{i+1} is now unreachable from the tree and compression was successful

FIGURE 3.9: Compression using a freeze node

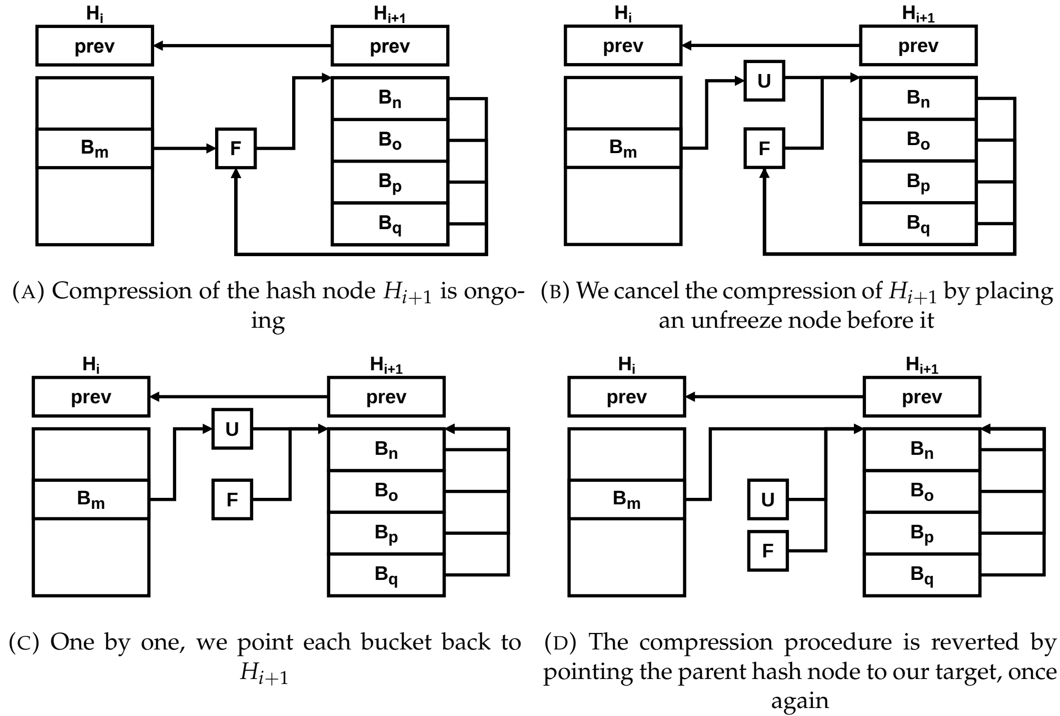
From here on out, the word "freezing" refers to the successful replacement of a field, using *compare-and-swap*, to a special *freeze* type node. Likewise, "unfreezing" refers to the successful replacement of a field to an *unfreeze* type node.

Only empty hash nodes can be compressed. The compression procedure will be reverted if one of the buckets is populated in the meantime.

When inserting a node we can cancel an ongoing compression to the target hash node. To cancel the compression, we must unfreeze the bucket pointing to the target hash node, as illustrated in Fig. 3.10. If the replacement of the *freeze* node is successful, it must mean that the compression process was canceled before terminating. In practice, this case rarely occurs. While gathering benchmarks, unfreezing would only take place during tests with high degrees of contention which hash functions prevent.

3.5.1.1 Algorithm

The **COMPRESS**(H, h) function removes hash H, detaching it from its parent hash H'. H' is at level l. The **INSERT**, **LOOKUP** and **REMOVE** functions must be changed to detect and skip freeze F or unfreeze U nodes at the start of any chain. The chunk size (W) is a



(A) Compression of the hash node H_{i+1} is ongoing

(B) We cancel the compression of H_{i+1} by placing an unfreeze node before it

(C) One by one, we point each bucket back to H_{i+1}

(D) The compression procedure is reverted by pointing the parent hash node to our target, once again

FIGURE 3.10: Aborting the compression operation

preemptively configured global variable and does not change at runtime. Moreover, if the **INSERT** function detects a freeze node F it must try to unfreeze the hash node H.

Algorithm 7 COMPRESS($HashNode\ H, KeyHash\ h$)

1: $prev \leftarrow H.prev$
2: **if** $prev = nil$ or ISNOTEMPTY(H) **then**
3:     **return**
4: $F \leftarrow$ NEWFREEZENODE(H)
5: $i \leftarrow$ GETCHUNK(h, prev.level, W)
6: **if** CAS($prev[i], H, F$) fails **then**
7:     **return**
8: **for** $j \leftarrow 0; j < W; j \leftarrow j+1$ **do**
9:     **if** CAS($H[j], H, F$) fails **then**
10:         **return**
11: $H.prev \leftarrow nil$
12: **if** CAS($prev[i], F, prev$) **then**
13:     **return** COMPRESS($prev, h$)

Algorithm 7 illustrates the compression procedure. First, we verify if H is the root hash node. The root hash node is the entry point of our map, so we must not remove it (lines 2-3). The root hash node is the only one with no previous neighbors (its *prev* field is set to *nil*).

The **ISNOTEMPTY** function iterates all buckets of our target node H. If one such bucket is populated with any node, we abort the compression procedure (line 3) because we only wish to compress empty hash nodes.

At line 4, **NEWFREEZENODE** allocates a *freeze* node pointing to our target H. This freeze node is placed in between H and its parent hash node (*prev*) at line 6.

From lines 7 to 10, each bucket of H is pointed to the freeze node, one by one. Line 11 is for memory reclamation purposes. Setting the *prev* field of a hash node prevents a memory reclamation conflict which will be discussed next chapter. Finally, line 12 will commit the compression by completely detaching H from the tree. Hash node H was removed from its parent, *prev*, possibly leaving it empty. Therefore, we must also try to compress the parent.

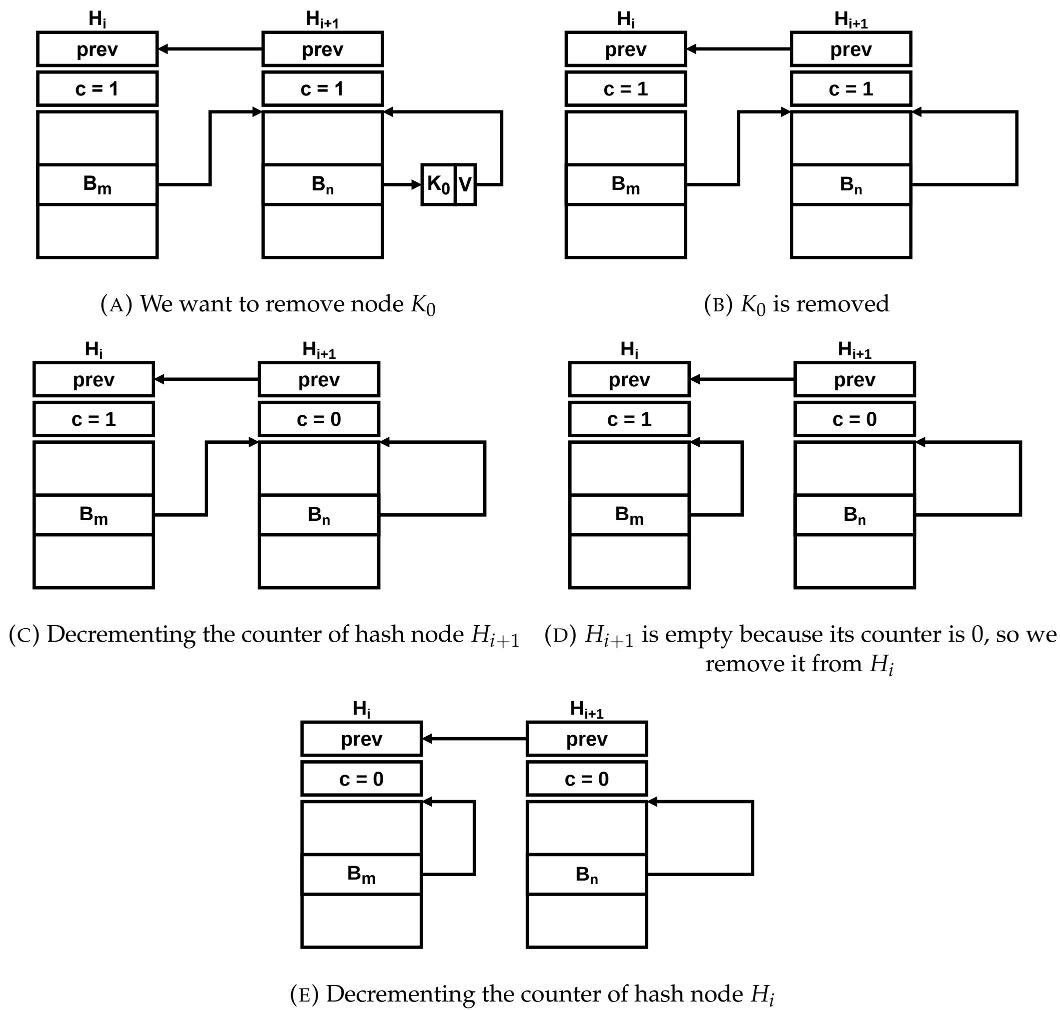
Every removal operation emptying a bucket will trigger the compression operation. After removing the last node of a chain, a thread must check if all buckets of the hash node are empty. Only then can the compression procedure begin. We can reduce the frequency of verification by marking one of the hash's buckets. A thread who empties the chain of a marked bucket is the only one responsible for compressing the hash node.

3.5.2 Counter

A second method of marking a hash node as immutable is to keep a counter of populated buckets updated in each hash node. Figure 3.11 illustrates this idea. This version requires changes to the insertion procedure, adding extra synchronization to it. When removing a node which leaves the bucket empty, the counter must be decremented. If the counter reaches 0, no thread can add further nodes, effectively preventing further modifications to the hash node.

Note that the counter is incremented before the insertion of a node, using a CAS operation. This prevents any thread from adding a node to an already detached hash, and prevents the following conflict of operations:

1. T_1 adds node K_0 to an empty bucket B_x.
2. T_1 is interrupted by the operating system scheduler.
3. T_2 appends node K_1 to K_0.
4. Because K_0 was already in B_x, T_2 is not responsible for increasing the counter of the hash node, but T_1 is still asleep.



(A) We want to remove node K_0

(B) K_0 is removed

(C) Decrementing the counter of hash node H_{i+1}

(D) H_{i+1} is empty because its counter is 0, so we remove it from H_i

(E) Decrementing the counter of hash node H_i

FIGURE 3.11: Compression using a counter field

5. T_3 compresses the hash node, even though B_x is not empty, because T_1 has yet to increase the counter.

6. Nodes K_0 and K_1 are now unreachable.

Before inserting a node to an empty bucket, we increase the counter. If multiple threads try to insert to the same empty bucket, only one of them will succeed. This means a counter may, in fact, exceed the actual number of non-empty buckets. A thread which fails the insertion procedure must decrement the counter, balancing its value. In other words, during the insertion procedure, we increment the counter in order to prevent the compression of the hash node.

3.5.2.1 Algorithm

The **COMPRESS**(H, h) function removes hash H, detaching it from its parent hash H'. Algorithm 8 illustrates the alternative compression function, using counters. H' is at level l. The chunk size (W) is a preemptively configured global variable and does not change at runtime. Changes to both **INSERT** and **REMOVE** procedures must also be made. In fact, **INSERT** suffers a significant cost penalty due to the additional synchronization required when updating a counter.

Algorithm 8 COMPRESSION(*HashNode H*, *KeyHash h*)

```
1: prev ← H.prev
2: if prev = nil or H.counter ≠ 0 then
3:     return
4: i ← GETCHUNK(h, prev.level, W)
5: H.prev ← nil
6: if CAS(prev[i], H, prev) fails then
7:     return
8: if prev.prev = nil then
9:     return
10: count ← ATOMICDECREMENT(prev.counter)
11: if count ≠ 0 then
12:     return
13: return COMPRESSION(prev, h)
```

Lines 2-3 return if the target hash node H is the root or if it is not empty, indicated by the greater than zero counter. Like previously mentioned, line 5 prevents threads from attempting to compress a hash node with its memory already reclaimed. Line 6 will detach hash node H from the tree. From line 8 to 10, the thread which successfully compressed H will have to decrement its parent's counter. If this counter is equal to 0, we must try to compress the parent hash node *prev* (lines 11-13).

Algorithm 9 shows the changes applied to the aforementioned **INSERT** function to ensure the correctness of this compression method, using a counter. **INSERT**(H, h, v) must preemptively increment the counter of hash node H to prevent its compression during an insertion. *counter* is the number of populated buckets of H.

The insert function is similar to the original one, from lines 1 to 6, with the exception of the freeze/unfreeze node handling which is now unnecessary. In lines 7-12, we check whether the hash node H has already been issued for compression, by verifying if its counter is 0 (all hash nodes start with counters greater than 0). Lines 13-15 we append our new node N to the target collision chain. The counter represents the number of populated

Algorithm 9 INSERT(*HashNode H*, *KeyHash h*, *Value v*)

```
1:  ⟨node, H, prev, len⟩ ← LOOKUP(H, h)
2:  if node ≠ nil then
3:      return
4:  if len = EXPANSION_THRESHOLD then
5:      H ← EXPAND(H, h, prev)
6:      return INSERT(H, h, v)
7:  observed ← H.counter
8:  if observed = 0 then
9:      COMPRESS(H, h)
10:     return INSERT(RootHashNode, h, v)
11: if prev = H then
12:     while CAS(H.counter, observed, observed + 1) fails do
13:         if H.counter = 0 then
14:             COMPRESS(H, h)
15:             return INSERT(RootHashNode, h, v)
16: N ← NEWLEAFNODE(H, h, v)
17: if CAS(prev.next, H, N) fails then
18:     ATOMICDECREMENT(prev.counter)
19:     return INSERT(H, h, v)
```

buckets of H. Therefore, if the tail of the chain is a leaf node, then the bucket is already occupied and the counter is not incremented. In the remaining lines we try to increment the counter. If the CAS at line 18 fails, it means some other thread tempered with the counter and we must check if it was left at 0 or not. If that's the case, we must reinsert the node N (lines 19-23).

We must change the **REMOVE** function to perform **ATOMICDECREMENT**(*counter*) after detaching a node and leaving its chain empty. The **ATOMICDECREMENT** function will decrement the given field, by one, and return the new value of the field atomically.

3.6 Cost Analysis

3.6.1 Lookup Cost

The time complexity of the **LOOKUP** procedure is determined by the maximum height L of the tree and the maximum length C of collision chains. In other words, L is the maximum number of recursive calls to **LOOKUP** and C is the number of maximum iterations of the collision chain traversal loop. The maximum height is determined by the chunk

size, W, and the maximum length of the hash, h.

$$max(L) = \lceil \frac{\#h}{W} \rceil$$

Thus, a path to a leaf can be shorter but no longer than $L + C$, making the worst case $O(L + C)$.

On an x64 machine, the length of a hash would be 64 bits, resulting in a constant maximum height of the tree. For example, the longest path possible on an x64 machine, using chunks of 4 bits, would be $max(L) = \frac{64}{4} = 16$. For any theoretical machine, however, it could be much longer ($\#h < +\infty$).

Consider a hypothetical ideal hash function which avoids collisions whenever possible. Figure 3.12 illustrates this case. The letter n represents the amount of keys currently inserted on the map. For every n inserted keys, the height of the tree is at most $\log_W(n)$. This is the conclusion Bagwell [9] reached, when proposing the data structure, and which constitutes the best case for cost. The hash functions we use in practice cannot consistently perform this perfectly even distribution of keys on hash nodes. However, the cost of traversing the tree is still logarithmic on average. Although the **LOOKUP** function of hash trie map is more costly than that of a classic hash map, the trade-off is a lower cost of expansion and compression. And, on average, the asymptotic cost of **LOOKUP** is still constant.

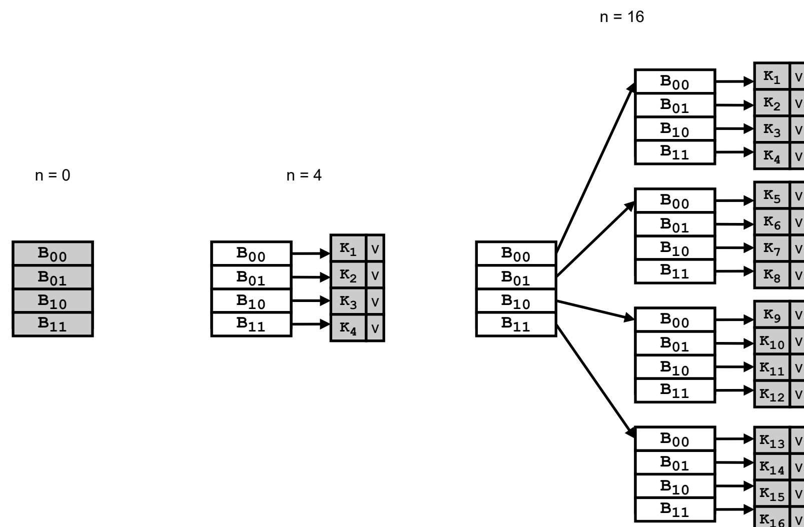


FIGURE 3.12: Hash trie map with a perfect hash function

In Chapter 4, we will see that the **LOOKUP** algorithm will suffer a number of costly modifications.

3.6.2 Insertion Cost

In regards to space and time complexity, the cost of the algorithm is determined by the cost of the **EXPAND** and **LOOKUP** functions.

It is possible, although very unlikely, that a thread could loop infinitely in a CAS-loop. This is caused by a form of starvation, where threads keep conflicting with one another and, as a result, one of the threads gets stuck in retrial. However, a good hash function will prevent retrials, by spreading keys evenly throughout the map, separating points of conflict. Despite all this, the average case is equal to that of the **LOOKUP** function.

3.6.3 Removal Cost

The asymptotic analysis in execution time of the removal function is identical to that of the insert function, described in section 3.6.2.

3.6.4 Expansion Cost

To expand, a traditional hash map allocates an array with double the entries to which all inserted nodes are moved to. Thus, typically, the cost for expansion is $\Theta(n)$.

The main advantage of the hash trie map is that expansions will only involve a fixed number of nodes. Each expansion is targeted at a specific collision chain. All nodes of the chain will be moved to a new hash node. Thus, the cost of expansion is $\Theta(C) = \Theta(1)$, C being the constant length of a collision chain.

3.6.5 Freeze Compression Cost

In theory, the iteration of all buckets gives a cost of $\Theta(2^W) = \Theta(1)$. In practice, however, 2^W writes to shared memory are required to successfully freeze a hash node. Thus, this compression function is slower than the one presented in section 3.5.2.

3.6.6 Counter Compression Cost

Only two writes to shared memory are required to successfully compress a hash node. On the other hand, the **INSERT** function suffers a higher number of CAS instructions for

every retrial. Additionally, the counter is another contention point. Thus, the synchronization load of the compression mechanism is moved to the much more frequent insert operation.

3.6.7 The Cost of Synchronization

We have studied the asymptotic cost of algorithms based on the amount of iterations performed. However, on modern CPUs, the cache takes a big load off a portion of these algorithm iterations, namely while traversing lists or a tree.

The main culprit behind slowdowns is *synchronization* because it forces CPUs to halt progress. Cache synchronization protocols, the flushing of write buffers and accesses to a higher level in the memory hierarchy are examples of mechanisms which can halt progress. Frequently relying on synchronization decreases the efficiency of the algorithm and worsens performance. Unfortunately, synchronization is necessary for ensuring algorithm correctness in concurrent programming. If we are able to reduce synchronization as much as possible while maintaining correctness, we can greatly improve the performance of our algorithm.

Memory reclamation methods typically demand more frequent synchronization. Therefore, memory reclamation methods are an integral part of algorithm optimization. Sometimes we can exploit certain characteristics of the algorithm to reduce the amount of synchronization needed to reclaim memory. Some algorithms may not even require memory reclamation methods, namely Harris and Purcell's hash map [16]. In garbage collected environments every process gets the same reclamation method which hinders the performance of potentially faster algorithms, like the aforementioned hash map.

Next chapter is dedicated to the implementation and analysis of memory reclamation methods to the LFHT.

Chapter 4

Memory Reclamation

Outside garbage collected environments, we may not be able to free memory right after removal of nodes from the LFHT data structure. After removing a leaf node we are not sure whether some other thread has a local reference to it. In the worst case, if we free the memory of this leaf node, we risk a *use after free* problem like the following:

1. Thread T_1 stops amidst of traversing through node N_1.
2. T_2 removes and frees the memory of N_1.
3. T_2 allocates a new node N_2 calling *malloc*. The reference given by the allocator points to the exact same memory block of the previously deleted node N_1. N_2 is inserted in a completely different place on the hash map.
4. T_1 traverses through N_2, which has the same reference as the deleted N_1. T_1 has moved to another location in the structure.

Therefore, we need to manually implement a mechanism capable of determining when a pointer is no longer in use by any thread. This is called a *memory reclamation method* and there are a number of alternatives, some of which were experimented by Moreno et al. [34] for the LFHT data structure. In the end, a custom memory reclamation method was designed exclusively for use with LFHT called *hazard hash and level* (HHL) which was able to outperform other state of the art methods. However, the HHL is not compatible with the compression mechanism of LFHT. The *ABA problem* can be an additional problem associated with CAS-based lock-free programs.

Our end goal is to implement a memory reclamation method for LFHT compatible with the compression mechanism. In what follows, we discuss in detail the impact of

synchronization primitives and memory barriers in the performance of any program. McKenney [31] is the main source which supports these first sections. An in-depth understanding of synchronization will be necessary in the interpretation of the results given later in Chapter 5. Next, we describe in detail two already existing memory reclamation methods (Section 4.3), namely the *Hazard Pointers* method and the afore mentioned HHL. Moreover, in sub-section 4.1.2.1, the concept of *asymmetric memory barriers* is introduced which will be pivotal in optimizing our memory reclamation method, later on.

4.1 The Cost Of Synchronization

Synchronization primitives, atomic operations and memory barriers are some of the more costly operations. If we understand in detail how synchronization is implemented in a shared-memory multi-core system (SMP), we are able to optimize its use and speedup our programs.

In the first few sections, we will go over the architecture of an SMP. In the last few sections we will describe how memory barriers work, the different types of barriers and their cost.

4.1.1 The Cache

Overtime, manufactured CPUs have become faster while memory units more spacious. However, memory access latency has not been able to keep up with the execution speed of CPUs. Thus, every time a CPU performs an access to memory it must stall its execution during which it could have already executed hundreds of instructions. CPU stall hinders the progress guarantees of a lock-free algorithm.

It is important to define what characteristics contribute to higher latency of RAM access. They are as follows:

- Access must go through the slow system bus;
- RAM uses cheaper (albeit slower) registers than other memory units;
- Virtual memory address translation.

First, RAM and CPU must communicate through the slowest bus on the system. They share this interconnect with other I/O devices and must synchronize its access through a controller, one at a time.

Secondly, because manufacturers focus on producing spacious RAMs, they need to take into consideration the higher cost of manufacturing. After all, more transistors are required to build a larger memory. To reduce the manufacturing costs, each RAM register must contain less transistors than a CPU or cache register, as explained by Null et al. [27]. Registers with a higher number of transistors are faster than RAM registers, which further contributes to the disparity of latency.

Finally, as evidenced by Aiken et al. [17], virtual memory address translation does have an impact on access latency. After all, performing one read of RAM requires multiple memory accesses to manage and inspect *translation lookaside buffer* entries. However, it is hard to determine exactly how much virtual memory contributes to access latency.

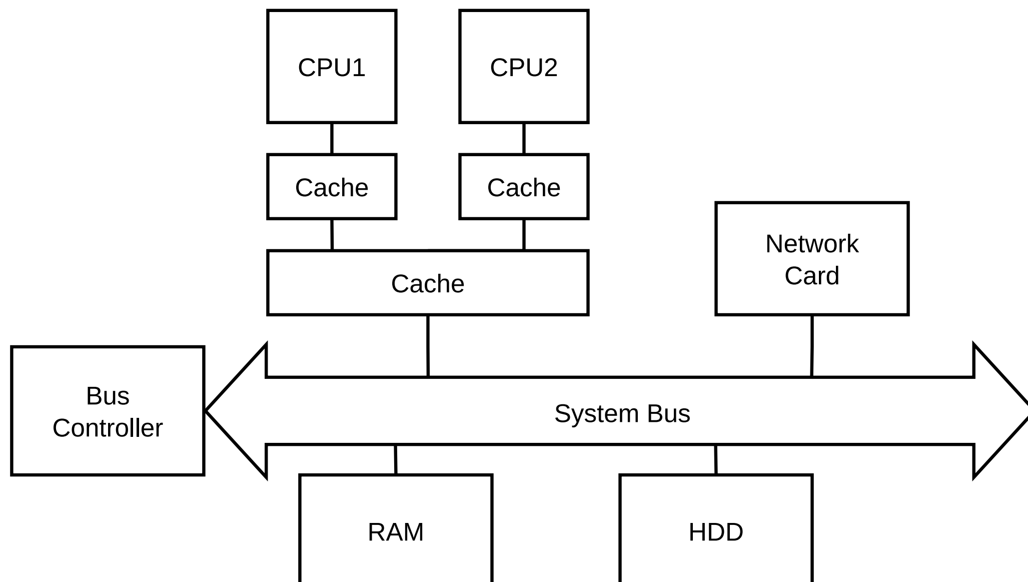


FIGURE 4.1: Typical multi-core UMA architecture

Caches are smaller and faster memory units which mitigate these latency issues and minimize CPU stall. They share a private interconnect with CPUs and have faster registers. Figure 4.1 illustrates an example architecture of a shared memory multi-core system (SMP). Each CPU has a cache memory which it has exclusive access to. Caches form a hierarchy in which the smaller caches are closer to CPUs and larger caches closer to RAM. When the CPU reads from memory, caches fetch multiple contiguous bytes of memory from RAM at once. This group of bytes is called a *cache line* or *cache block*. Although fetching from RAM is slow, subsequent accesses to the cached blocks of memory will be retrieved without needing to access RAM, preventing the CPU from stalling.

In the context of the LFHT data structure, once the tree hierarchy is built, many hash nodes will be cached. Thus, traversing the tree from root to leaves will have a negligible overhead since it will require little to no RAM access. What really hinders the performance of our concurrent data structure is *cache synchronization* and an abundance of *cache misses*, due to the synchronization between threads.

4.1.1.1 Cache Miss

The *cache miss* is an expensive event performed whenever the CPU requests to read a memory location not present in cache memory. It is expensive because it stalls the CPU while a number of contiguous memory blocks of RAM are loaded into cache. One way to reduce the amount of cache misses, we must reduce the number of writes to memory locations shared by multiple threads.

4.1.1.2 Cache Coherence Protocol

A low number of cache miss is not necessarily an indicator of higher throughput. Sometimes, the CPU may stall due to cache synchronization which may not involve high numbers of cache misses.

CPUs write to their own caches concurrently, which may cause concurrency hazards. For example, if two CPUs write to the same shared memory location on their respective caches, we are unsure which of the two values to push to memory. To ensure the correctness of our programs, caches must always have the most up to date versions of shared memory blocks. Cache coherence protocols give this guarantee, but with a cost. Caches communicate with message passing through the system interconnects in order to inform others of modified blocks. Caches send requests and wait for other caches to respond. This period of waiting for a response will stall the CPU, yet again, but will not register as a cache miss since no new memory blocks are cached.

Before writing to a block, a cache must guarantee exclusive rights to modify it. Figure 4.2 illustrates the exchanged messages in this case, where both CPUs 0 and 1 execute instructions from top to bottom. CPU 0 sends invalidate requests for other caches to discard the target block and waits for their responses, during which it cannot execute further instructions. The cache hierarchy was built in order to prevent the CPU from stalling, but, at the end of the day, the cache synchronization protocol will force CPUs to do just that anyway.

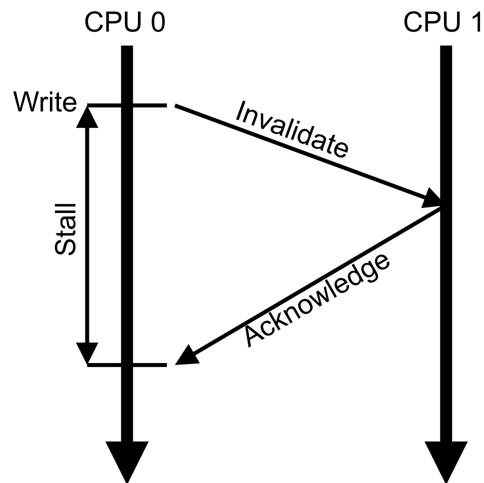


FIGURE 4.2: Unnecessary CPU stalls due to cache synchronization

To mitigate this issue, *store buffers* and *invalidate queues* are attached to each individual CPU. Before modifying a block, a CPU sends invalidate requests and pushes the modified block to the store buffer. Then, the CPU may progress even before receiving all acknowledgements, after which it pushes the changes of the buffer to the cache. The CPU may queue up multiple blocks on the buffer and may read directly from it to ensure it gets the most up to date blocks.

If the store buffer is full, the CPU must flush it by attending to each modification, waiting for the acknowledgements and pushing the changes onto the cache. That said, because store buffers have little space, CPU stalls will still be commonplace.

When a CPU receives an *invalidate* request, it tries to invalidate the cache block. However, because the cache may be busy, it may take long before it can send an acknowledgement. The delayed arrival of acknowledgements may cause store buffers to fill quickly. To mitigate this issue, therefore preventing constant store buffer flushing, caches place invalidate requests in the invalidate queues of other caches. This way, we do not need to wait for acknowledgements since we guarantee the block will eventually be discarded by others.

4.1.2 Memory Barriers

Cache coherence protocols ensure us that any write to cache on a shared memory multiprocessing system is visible to all CPUs and, ultimately, by RAM. However, store buffers

and invalidate queues make it such that modifications to memory may appear out of order. Furthermore, compilers and CPUs may perform deliberate reordering of instructions to save clock cycles and speed up execution.

Memory barriers are directives that inform compilers, CPUs and caches to guarantee a specific order of modifications to memory in a concurrent program. For caches, *read memory barriers* flush the invalidate queues while *write memory barriers* flush store buffers which will force the CPU to stall. Another term used for memory barriers is: *memory fences*. We will use the word "barriers" interchangeably with memory barriers.

Additionally, barriers prevent compilers and CPUs from reordering instructions passed that barrier, yet another reason to use barriers sparingly.

Some operations require both types of barriers to ensure correctness, such as the CAS synchronization primitive. When a single barrier uses both read barrier and write barrier semantics, we call it a *strong memory barrier*. Thus, we must minimize the amount of CAS calls to prevent CPU stall.

4.1.2.1 Asymmetric Memory Barriers

Sometimes, it is necessary to use strong barriers in frequently called functions (common code paths) to prevent concurrency hazards. However, this will have a negative impact on the program's performance, in throughput.

Asymmetric barriers mitigate this issue by replacing strong barriers, in common code paths, with *weaker* barriers. We use the adjectives *weak* and *strong* to refer to barriers which, respectively, cause less CPU stall or cause more CPU stall. In other words, a stronger barrier will force CPUs to stall for longer periods of time, but will limit the possible order of instructions which, in turn, may help prevent concurrency conflicts. In exchange, an asymmetric barrier is used in an uncommon code path. Asymmetric barriers are stronger than strong symmetric barriers, so they should only be used if there is a disparity of execution time between functions.

```
1 _Atomic(int) c;
2
3 A(int tid) {
4   if (tid == 0) atomic_store(&c, *c + 1); // strong barrier
5
6   work();
7 }
```

```
8
9 B(int tid) {
10   int obs = atomic_load(&c); // strong barrier
11
12   doOtherWork(obs);
13 }
```

LISTING 4.1: Program which can be optimized using asymmetric barriers

A potential use case of asymmetric barriers is shown in Listing 4.1. Variable *c* is a single-writer multiple-reader location only modified by thread with *tid* of 0, read by other threads at line 10. If most of the time is spent calling function **A** (common code path), and only rarely function **B**, T_0 is suffering the consequences of the strong barrier at line 4 needlessly. Note that the memory barriers at lines 4 and 10 ensure that the most up-to-date value of *c* is visible to all threads. We are able to mitigate this issue using asymmetric barriers, as shown in Listing 4.2.

```
1 _Atomic(int) c;
2
3 A() {
4   // weak barrier
5   if (tid == 0) atomic_store_explicit(&c, *c + 1, memory_order_relaxed);
6
7   work();
8 }
9
10 B() {
11   asymmetric_barrier();
12   int obs = atomic_load(&c);
13
14   doOtherWork(obs);
15 }
```

LISTING 4.2: Using asymmetric barriers for optimization

At line 5, thread T_0 now uses a relaxed memory barrier preventing the CPU and compiler from reordering instructions, but does not force its cache buffers to flush. In exchange, threads at line 11 will have to execute an expensive barrier. The trade-off is worth it because function **B** is rarely executed.

Upon encountering memory barriers, processors are forced to flush their own private store buffers and invalidate queues. Thus, stalling the CPU while guaranteeing that the

thread's changes are made available to other processors. An asymmetric barrier, on the other hand, will send signals to all other processors forcing them to flush their buffers. This way, caches are synchronized on demand. However, an asymmetric barrier is heavier than strong memory barriers, so it should only be used in uncommon code paths. With asymmetric barriers we prevent the CPU from stalling frequently in common code paths.

4.1.3 Summary

In sum, the more modifications we make to shared memory locations the higher the amount of cache misses, because modifying a cache block will force other caches to invalidate this same block, specially if this is a frequently read block by all caches.

On the other hand, the use of memory barriers hinders the performance of our concurrent programs. Frequent use of memory barriers will force cores to flush store buffers and invalidate queues which will stall the CPU and prevent compilers and CPUs from optimizing instruction order. Frequent use of memory barriers prevents us from taking advantage of the hardware optimizations available to our architecture. Asymmetric memory barriers can mitigate the effects of frequently used barriers, only in specific situations.

An optimal parallel program must: minimize the number of writes to memory; use memory barriers sparingly; and use the right types of barriers, all of this while guaranteeing algorithm correctness.

4.2 Memory Life Cycle

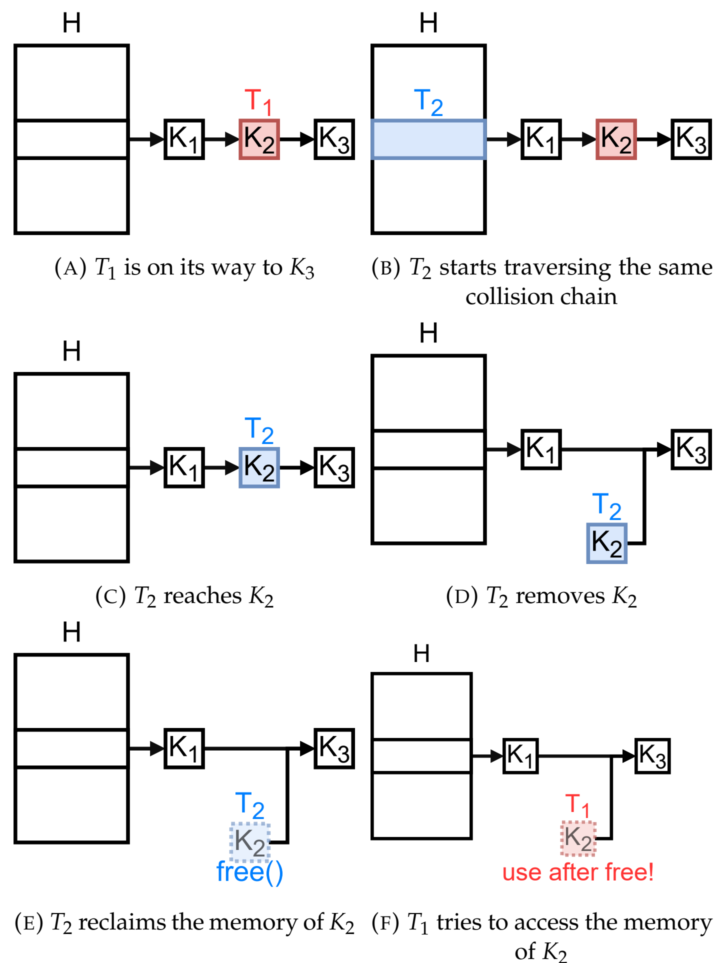
Before delving into concrete reclamation methods it is useful to describe the life cycle of allocated memory. We will refer to allocated memory blocks we want to free as "nodes".

A node is **unallocated** before its creation. After being assigned a memory location a node becomes **allocated** and eventually becomes **reachable** once added to the data structure. Logically removing a node will make it **unreachable**.

This node is then placed in a *reclamation queue* where references of nodes are maintained until they can be freed. A node is **retired** after it has been logically removed from the data structure and placed in the reclamation queue. The reclamation function consists of iterating the queue and identifying which nodes may be reclaimed. A node may only be reclaim if no other thread possesses a reference to such node. After freeing a node's memory, it becomes **reclaimed**.

4.3 Memory Reclamation Methods

Figure 4.3 illustrates an example of the memory reclamation problem, applied to our hash map. Thread T_1 is traversing a collision chain, and is on its way to node K_3, as it is put to sleep by the operating system scheduler. Thread T_2, then, removes K_2. Ordinarily, this would have been fine, since T_1 can get back on track to the collision chain by following the **.next** reference of K_2. However, T_2 reclaims the memory of K_2 and, so, T_1 will eventually try to access an already reclaimed memory block.



(A) T_1 is on its way to K_3

(B) T_2 starts traversing the same collision chain

(C) T_2 reaches K_2

(D) T_2 removes K_2

(E) T_2 reclaims the memory of K_2

(F) T_1 tries to access the memory of K_2

FIGURE 4.3: The memory reclamation problem

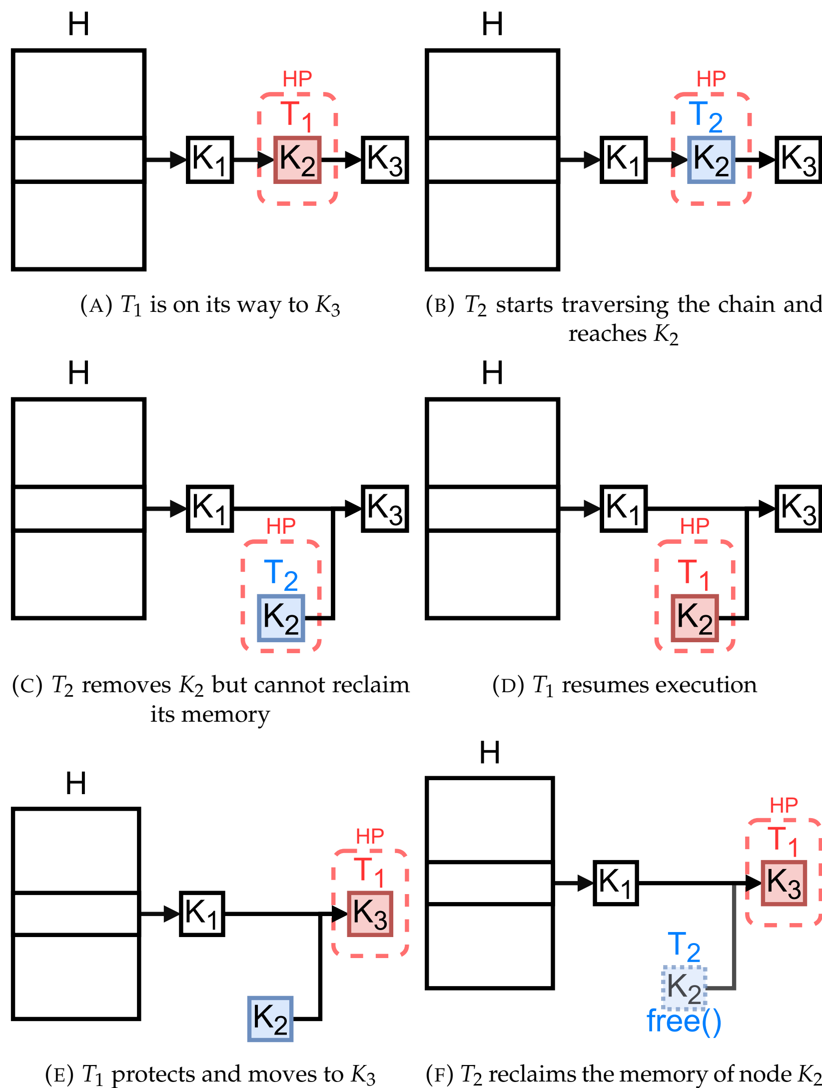
In this section we describe two memory reclamation methods applied to the LFHT: *Hazard Pointers* and *Hazard Hash and Level*, which can solve the problem described above.

4.3.1 Hazard Pointers

In the *hazard pointers* method, proposed by Michael [15], each thread maintains a number of single-writer multiple-reader pointers, called hazard pointers, which are used to

prevent objects from having their memory reclaimed.

First, we need to identify what objects of our shared data structure can be removed and reclaimed. Then, before accessing the memory of one such object, we store its reference in one of the hazard pointers. This way, we prevent its memory from being reclaimed while our thread is using it. We say that a hazard pointer is protecting an object, when the reference of the object is stored in the hazard pointer. The thread will eventually stop using the object and, as such, will discard its hazard pointer protection. An object can have its memory reclaimed if and only if no hazard pointer is protecting it.



(A) T_1 is on its way to K_3

(B) T_2 starts traversing the chain and reaches K_2

(C) T_2 removes K_2 but cannot reclaim its memory

(D) T_1 resumes execution

(E) T_1 protects and moves to K_3

(F) T_2 reclaims the memory of node K_2

FIGURE 4.4: Solution to the memory reclamation problem using hazard pointers

Figure 4.4 illustrates the solution to the problem previously shown in Fig. 4.3. If thread T_1 protects K_2 before accessing it, using a hazard pointer, T_2 will not be able to free the

memory of K_2. Only after T_1 moves the hazard pointer to the next node on the collision chain (K_3) can T_2 reclaim the memory of the now unprotected K_2.

Protecting an object with a hazard pointer is a bit more complicated than merely setting one field. The code Listing 4.3 describes all necessary steps in order to protect a node.

```
_Atomic(void*)* hazard_pointers;

int hp_protect(int tid, void* node, _Atomic(void*) location) {
  // Get this thread's own hazard pointer.
  _Atomic(void*) hp = hazard_pointers[tid];

  // hp = node;
  atomic_store(&hp, node); // strong barrier

  // Was "node" removed in the meantime?
  void* observed = atomic_load(&location);

  // The protection of "node" will only succeed, if the
  // node remains in place during this function.
  //
  // If observed != node, someone has removed "node" in
  // the meantime, and could have already reclaimed its
  // memory
  return observed == node;
}
```

LISTING 4.3: Hazard pointer protection

The variable **location** is the memory location where **node** is attached to on the map. For example, it could be the **.next** field of the previous node on a collision chain, or it could be a bucket slot of a hash node. After setting the reference of the node to a hazard pointer owned by our thread (line 8), we must perform an additional safety check to make sure **node** was not removed and reclaimed in the meantime (lines 11 and 19). We read the value on **location** and check if it still points to our target node. Otherwise, we risk suffering the consequences of a race condition wherein **node** is removed and reclaimed before our protection is made visible, at line 8.

Listing 4.4 illustrates the memory reclamation function. When an object is removed by a thread, it is first placed into the thread's private *reclamation list* (lines 6-7). When this list reaches a certain length (line 10), the thread tries to free the memory of each node of

the list (lines 14-24). A node can only have its memory reclaimed if and only if it is not protected by any hazard pointer (line 15).

```
_Atomic(void*)* hazard_pointers;
LinkedList** reclamation_list;

void hp_reclaim(int tid, void* node) {
  // Get this thread's own reclamation list.
  LinkedList* list = reclamation_list[tid];
  list_add(list, node);

  // Reclamation threshold can be adjusted.
  // In our case, we use 10000.
  if (list_length(list) < RECLAMATION_THRESHOLD) {
    return;
  }

  atomic_thread_fence(memory_order_acquire);

  for_each(void* elem: list) {

    if(is_node_protected(hazard_pointers, elem)) {
      // Node is currently in use.
      //
      // Node will remain on reclamation list
      // and its reclamation will be postponed.
      continue;
    }
    list_remove(list, elem);
    free(elem);
  }
}
```

LISTING 4.4: Hazard pointer reclamation function

After setting the hazard pointer field, we must call a strong memory barrier to make sure the most up-to-date version of all hazard pointers are propagated to other chaches (line 8 of Listing 4.3). Without this barrier, other threads could remove and reclaim the **node** even after our thread successfully returns from the function **hp_protect**. Without barriers, the instructions at line 15 of Listing 4.4, during the reclamation function, could

observe outdated values in cache of other thread's hazard pointers and, consequently, reclaim a protected node.

The overhead caused by these barriers can be mitigated using asymmetric barriers, as described in Section 4.1.2.1. As a matter of fact, Dice et al. [29] applied asymmetric barriers to the hazard pointers memory reclamation method and obtained significant performance improvements. As discussed previously, the common code path will be adapted to use weak memory barriers instead. In this case, the common code path is the function **hp_protect** as illustrated in line 8 of Listing 4.3. As a result, the reclamation function will need to call an asymmetric barrier before checking any hazard pointers, in function **hp_reclaim** (line 13). This asymmetric barrier will force other threads to propagate their most up-to-date values of their respective hazard pointers to the calling thread. Asymmetric barriers are heavy operations which stall CPUs for long, but, in this case, they will only be called after a number of removed nodes indicated by the **RECLAMATION_THRESHOLD** (we used a value of 10000 in our implementation).

4.3.2 Hazard Hash And Level

In the hazard pointers method, we protect one node at a time with one synchronized write to memory. On the other hand, the *Hazard Hash and Level (HHL)* method, proposed by Moreno et al. [38], is a specific memory reclamation method capable of protecting entire collision chains with only two synchronized writes to memory. However, this method is not compatible with the compression of hash nodes, meaning only leaf nodes may have their memory reclaimed and hash nodes will be left intact.

Each thread maintains a pair of fields called a *hazard pair* made of a *hazard hash* and a *hazard level*. The hazard hash identifies a path, and the hazard level selects a collision chain from that path. The hazard hash is set to the hash of the node currently being targeted. The hash of a node represents the path taken from the root of the tree to the node itself. Thus, with the hazard hash we can identify the path containing the group of leaf nodes we wish to protect. In a single path, though, there could be multiple collision chains due to an ongoing expansion. Therefore, the hazard level selects the appropriate chain from the path we want to protect. Leaf nodes will have their memory reclaimed if and only if they do not belong to a collision chain currently under protection. We must mark leaf nodes with the level they were first inserted, since nodes are moved to newer collision chains after expansion.

After a number of retired leaf nodes exceed a given threshold, the reclamation function starts. First, the hazard pairs of all threads are loaded into local memory. Then, we check, for every node in the reclamation queue, if the node was not removed from a collision chain currently under protection.

This memory reclamation method is able to exploit the specific characteristics of the hash map to obtain better performance, in throughput, compared to the hazard pointers method.

4.4 Our Contribution

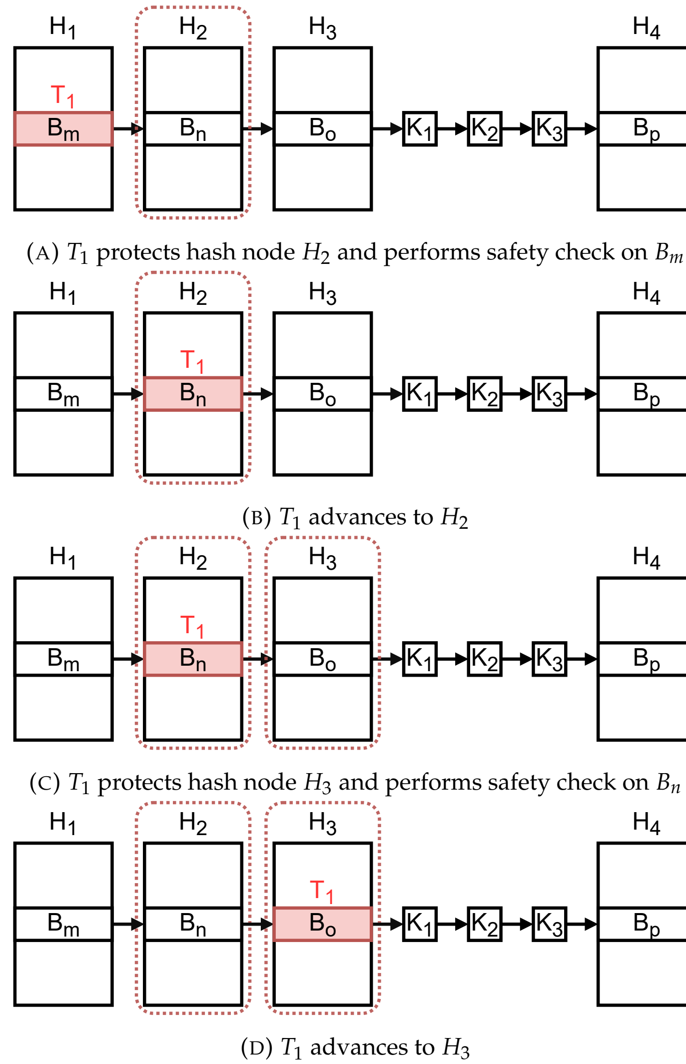
To remedy HHL's lack of support for map compression, we propose a lock-free and safe memory reclamation method based on hazard pointers that is compatible with the compression mechanism. The key goals of our approach are to:

- Fully support compression, which includes the ability to reclaim memory from removed hash nodes and removed leaf nodes;
- Maintain the lock-freedom property;
- Guarantee fixed memory bounds.

In our approach, a hazard pointer can be used to protect the individual reference of either a hash node or a leaf node. Before traversing through a hash or leaf node, we must protect its reference. This is because leaf nodes can be removed directly by threads, and hash nodes can be removed due to compression.

All necessary hazard pointer protections will be performed during the **LOOKUP** function. Thus, while traversing the tree, we protect each hash node on the hierarchy or leaf node on a collision chain before accessing it. Figure 4.5 illustrates the steps needed to traverse the tree hierarchy.

The root hash node, H_1, is never compressed. Thus, we do not need to protect it beforehand. Every hash node henceforth needs to be protected with a hazard pointer. After we select the appropriate bucket, in the case of Fig. 4.5a, B_m, we protect the hash node H_2 by setting its reference to a hazard pointer, illustrated by the dotted lines surrounding the whole memory block. As we have mentioned in previous section 4.3.1, we must perform an additional safety check to determine if H_2 was removed and reclaimed in the meantime. Because B_m still points to H_2, the protection was successful, and we may proceed



(A) T_1 protects hash node H_2 and performs safety check on B_m

(B) T_1 advances to H_2

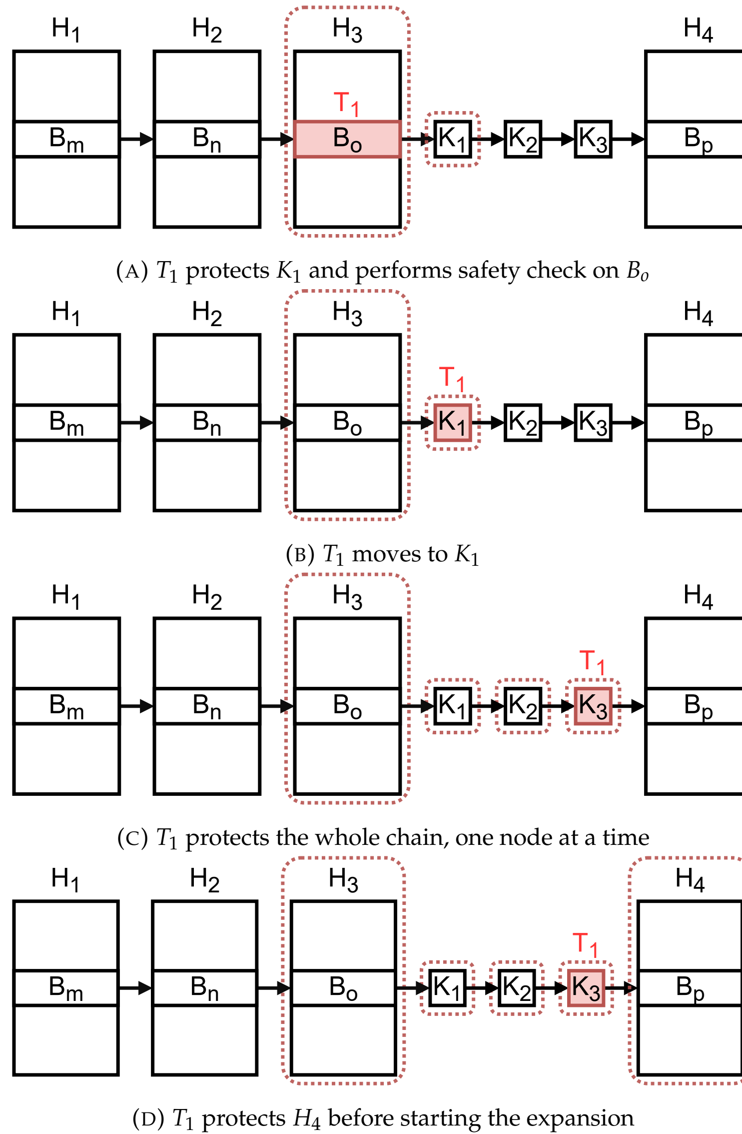
(C) T_1 protects hash node H_3 and performs safety check on B_n

(D) T_1 advances to H_3

FIGURE 4.5: Hazard pointers applied to the traversal of the LFHT tree

with the traversal (Fig. 4.5b). Now, to protect H_3 we need a separate hazard pointer (Fig. 4.5c). If we reuse the hazard pointer currently protecting H_2, we risk having the memory of hash node H_2 reclaimed during our additional safety check on bucket B_n. Thus, we need at least two separate hazard pointer fields per thread in order to traverse the tree hierarchy.

The protection of leaf nodes is similar to that of hash nodes. This is illustrated in Fig. 4.6. The main difference is that all leaf nodes of the collision chain are protected by a separate hazard pointer. We want to prevent any leaf node from having its memory reclaimed while moving the collision chain to a new level, during an expansion, such as the one illustrated in the figure, represented by H_4 on the tail of the chain. We have omitted the intermediate steps between Fig. 4.6b and 4.6c. The omitted steps are similar to those first illustrated in Fig. 4.6a and 4.6b, for each node on the chain, one at a time.



(A) T_1 protects K_1 and performs safety check on B_o

(B) T_1 moves to K_1

(C) T_1 protects the whole chain, one node at a time

(D) T_1 protects H_4 before starting the expansion

FIGURE 4.6: Hazard pointers applied to the traversal of LFHT collision chains

Figure 4.7 represents a particular case in which the removal of nodes on a collision chain may conflict with hazard pointer protection. Consider that thread T_1 is traversing the collision chain starting from bucket B_p, manages to protect the nodes K_0 and K_1, and is now trying to follow the chain from K_1 to K_2. Consider also that, concurrently, thread T_2 removes K_1 and K_2 from the hash map. We know that K_1 will not have its memory reclaimed because it is protected by T_1. However, since K_2 is unprotected, it can be reclaimed by any thread. Hence, to not risk accessing the potentially freed memory block of K_2, T_1 must restart traversing from bucket B_p. Whenever threads fail to protect nodes on a collision chain, the lookup procedure must be restarted from the last bucket.

Algorithm 10 summarizes the use of hazard pointers for the **LOOKUP** procedure,

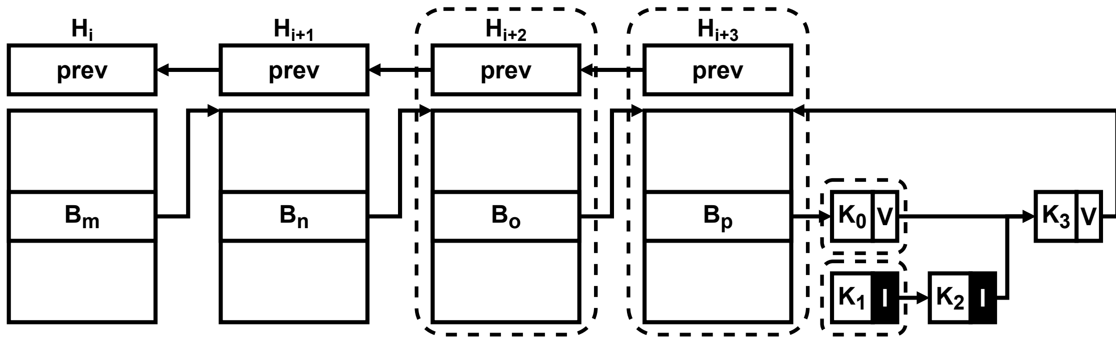


FIGURE 4.7: Protecting nodes during lookup (dotted lines represent HP protections)

given a hash node H and a hash h. The chunk size (W) is a preemptively configured global variable and does not change at runtime. At the start, we determine the *index* of the bucket at level $H.level$, using bit-wise shifts and masks to select a chunk of W bits from the hash h (line 1). Then, we load the contents of the bucket of hash node H, using the *index*, and try to protect the observed reference read from this bucket (lines 2–3). The **HP-PROTECT** procedure sets the reference of the first argument to one of the thread's hazard pointers and performs the protection safety check by re-reading the second argument. If the protection safety check fails, **HPPROTECT** will return false, and we should restart the algorithm from the beginning (line 4).

Next, if the bucket points to a freeze node, we try to skip it and move to the next node (lines 5–11). Then, if the freeze node or the bucket point to a new hash node, we traverse down a level of the tree (lines 14–15) performing **LOOKUP** recursively on this new found hash node. Otherwise, we traverse through the collision chain (lines 16–25). We must always try to protect the reference of the next node (lines 23–24) before moving to it (line 25) and, if we find an invalid leaf node, we must attempt to remove it using the **REMOVE** procedure (line 17). We will explain in more detail this last step on section 4.4.2. If the removal is successful, the removed node goes to the thread's reclamation list and we may proceed. If the removal fails, we must restart the lookup (line 18). Finally, if we find the node with the given hash h, we return successfully (lines 21–22).

Algorithm 10 Lookup(H, h)

```
1: index ← GetChunk(h, H.level, W)
2: iter ← H[index]
3: if HPPROTECT(iter, H[index]) fails then
4:     return LOOKUP(H, h)
5: if iter is a freeze node then
6:     nxt ← iter.next
7:     if HPPROTECT(nxt, iter.next) fails then
8:         return LOOKUP(H, h)
9:     if nxt = H then
10:        return LOOKUP(RootH, h)
11:    iter ← nxt
12: prev ← H
13: while iter ≠ H do
14:    if iter is a hash node then
15:        return LOOKUP(iter, h)
16:    nxt ← iter.next
17:    if iter is invalid and REMOVE(iter, h) fails then
18:        return LOOKUP(H, h)
19:    else if iter is valid then
20:        prev ← iter
21:        if iter.hash = h then
22:            return TRUE
23:    if HPPROTECT(nxt, prev.next) fails then
24:        return LOOKUP(H, h)
25:    iter ← nxt
26: return FALSE
```

4.4.1 Number of Hazard Pointers

As mentioned previously, each thread requires at least two hazard pointers to protect pairs of subsequent hash nodes, plus as many hazard pointers as the maximum length of

a collision chain, to protect leaf nodes. In what follows, we give more details about the reason for these numbers.

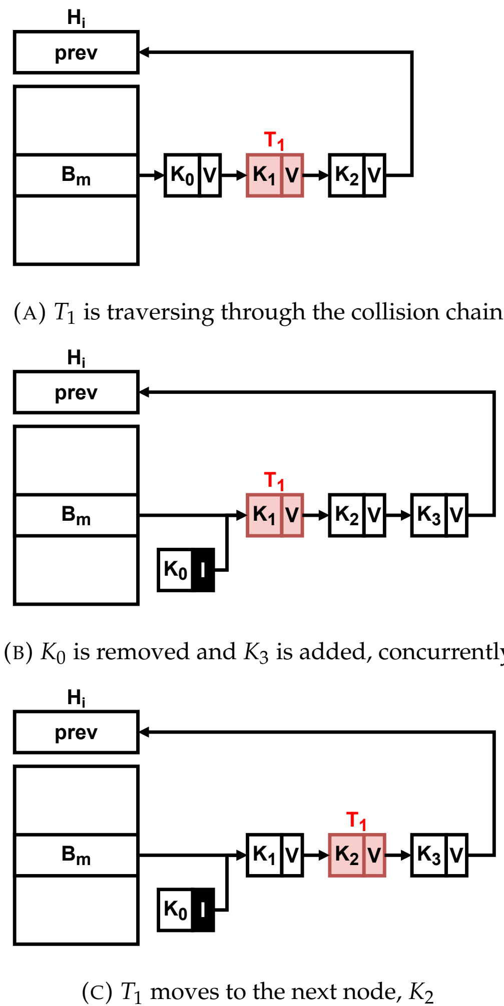
First, why do we need to protect more than one hash node at a time? After setting a hazard pointer to protect a hash node, the node can still be removed by another thread just before we are able to protect it. To prevent this, we must perform an additional protection safety check to verify if the protection was successful and, if the safety check fails, we must restart the operation. To perform a safety check, we need one hazard pointer to protect the parent hash node and another to protect the child. This is necessary to prevent the parent hash node from having its memory reclaimed during the protection of the child hash node. Thus, to traverse any list or tree, we need at least two hazard pointers.

For the leaf nodes, however, due to the expansion procedure, we must protect every node of the collision chain with a separate hazard pointer. This is necessary because, when rehashing nodes to a new level (expanding), we rehash each leaf node from the tail to the head of the collision chain. However, since leaf nodes lack a reference to the previous node on the collision chain, we cannot protect leaf nodes during expansion and we must protect all nodes before starting the expansion.

During lookup, a thread may need to traverse an infinite amount of leaf nodes. This is illustrated in Fig. 4.8 where a previous node on the chain, K_0, is concurrently removed and a new node, K_3, is appended to the chain. To solve this issue, we must count the number of hops when traversing collision chains. If the number of hops exceeds the maximum length of a collision chain, we must go back to the beginning of the chain.[1]

In conclusion, we need at most $2 + L$ hazard pointers per thread (L being the maximum length of the collision chain). Bounded memory is guaranteed because only at most $(2 + L) \times T$ references (T being the maximum number of threads in execution) can be protected at any instant.

[1] For the sake of simplicity, this is omitted in Algorithm 10



(A) T_1 is traversing through the collision chain

(B) K_0 is removed and K_3 is added, concurrently

(C) T_1 moves to the next node, K_2

FIGURE 4.8: Infinite collision chain traversal

4.4.2 Other Important Changes

To completely ensure the correctness of our approach, the following changes were also applied (we discuss each one in more detail next):

- Threads must remove invalid nodes during lookup;
- Threads must cooperate in an ongoing expansion before proceeding;
- After compression, the *prev* field of a hash node must made invalid.

In Fig. 4.7, we have already illustrated an example where a thread needs to restart traversing from the head of a collision chain if an invalid node N is found since, otherwise, it would not be possible to protect the next node on the chain. However, if the thread responsible for removing N is blocked for long, other threads will continuously loop back

to the head of the chain. Therefore, to prevent obstruction, threads must detach invalid nodes from the collision chain.

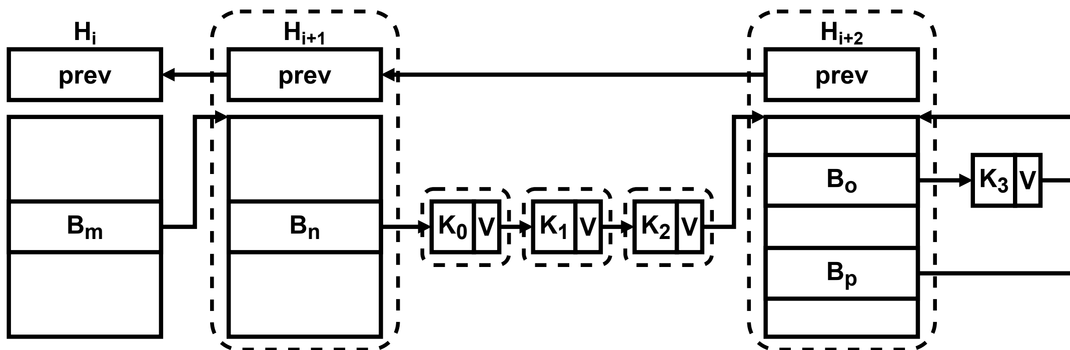


FIGURE 4.9: Not enough hazard pointers when expanding

During expansion, leaf nodes are moved to a new level L. If other threads insert new nodes in L concurrently, the thread responsible for expanding may need to protect these newly inserted nodes but may not have more hazard pointers available. This situation is illustrated in Fig. 4.9. Dotted lines represent the nodes protected with hazard pointers by the expanding thread T and node K_3 represents a node inserted by another thread. Because T has no more available hazard pointers, it would need to discard one already in use in order to protect K_3, which may not be possible. To prevent this kind of situation, threads cooperate in an ongoing expansion, before adding new nodes. With this, K_3 would never be inserted before all nodes were rehashed.

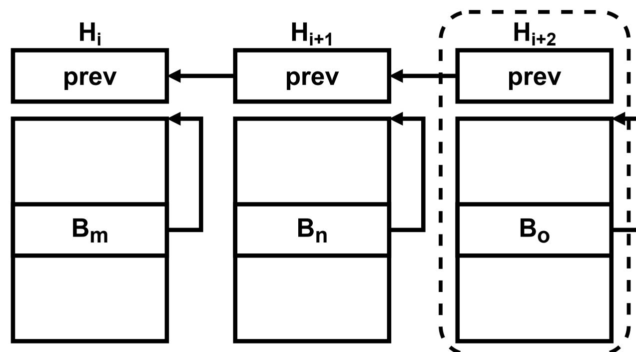


FIGURE 4.10: Unable to protect a parent hash node

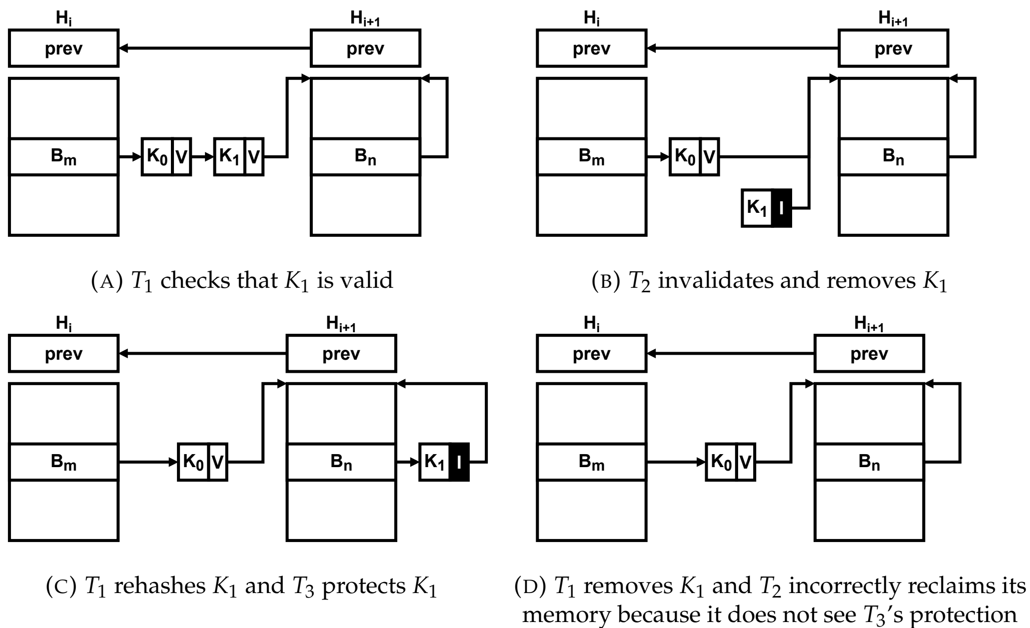
Every hash node has a reference to its parent in a field called *prev*. This field is an important part of the compression mechanism since, when a hash node is successfully compressed, we follow *prev*'s reference to also attempt to compress the parent hash node. However, since the *prev* field never changes its value, the hazard pointer safety check

is useless. The *prev* field will still point to the parent even after the parent has been reclaimed. In Fig. 4.10, thread T compresses H_{i+2} and then tries to protect H_{i+1} before starting its compression. However, in the meantime, some other thread completely removes H_{i+1} and frees its memory. Because the *prev* field of H_{i+2} remains unchanged, T cannot use it as a way to protect H_{i+1}. To make it possible to protect the parent hash node from the *prev* field, we must therefore change its value to *null* before removing the child hash node. And before modifying it to *null*, we protect the parent hash node.

4.4.3 Delegation Problem

During expansion, if a thread T_1 only sees a node N as invalid after moving it to the next hash level (because concurrently another thread T_2 marked N as invalid), T_1 will become responsible for making N unreachable. The process of transferring this responsibility to the expanding thread is called *delegation*. One flaw with this process, called the *delegation problem*, is illustrated in Fig. 4.11.

When rehashing a collision chain into a new hash node, if the tail of the collision chain is removed concurrently, the tail may be reinserted on the new level during the ongoing expansion. This is because of a race condition where the expanding thread captures the reference of the tail and, before rehashing it, other thread invalidates and detaches the node from the collision chain.



(A) T_1 checks that K_1 is valid

(B) T_2 invalidates and removes K_1

(C) T_1 rehashes K_1 and T_3 protects K_1

(D) T_1 removes K_1 and T_2 incorrectly reclaims its memory because it does not see T_3's protection

FIGURE 4.11: LFHT's delegation problem

Table 4.1 demonstrates how just one check of all hazard pointers leads to the illegal reclamation of a node, in this case, K_1. Thread T_2, the one responsible for the removal of K_1, is looping through all thread's hazard pointers, checking if any pointer matches node K_1, which T_2 wishes to reclaim. During the first iteration, $t = 1$, T_2 checks for T_3's hazard pointer, which is not protecting K_1. On the last iteration, $t = 3$, T_3 has protected K_1 due to the reinsertion and T_1 has released its protection. Thus, T_2 will reclaim the memory of the node, even though it is in possession of T_3. By performing a second pass through all thread's hazard pointers, this problem is solved. This is the solution implemented in the HHL method.

	T_3	T_2	T_1
$t = 1$	-	-	K_1
$t = 2$	K_1	-	K_1
$t = 3$	K_1	-	-

TABLE 4.1: Demonstrating the effects of the delegation problem in the reclamation procedure

The solution implemented in the HHL method for the delegation problem is to check all hazard pointers twice during the reclamation procedure [38].

Notice how only nodes at the tail of a collision chain can suffer from this problem. If we invalidate a node before the tail, this invalidation will be visible to the expansion procedure when time comes to move this node to the new level. Therefore, our approach to solving this problem is to verify if the node we wish to remove is pointing to a new hash node. Before we remove a node, we must assist with an ongoing expansion. Once the expansion is finished, we may remove the node. This way, we do not need to check all hazard pointers twice to ensure correctness.

Chapter 5

Experiments

In this chapter, we compare the performance, in throughput, of the proposed features of LFHT. First, we compare both compression designs documented in the previous chapters, one of which uses a counter and another which uses *freeze* control nodes. Then, we measure the overhead of the memory reclamation method relative to implementations with no support for memory reclamation.

Bellow is the list of implementations we want to compare and their acronyms:

- **SNF**, static no free, which does not implement compression and does not reclaim memory;
- **FNF**, elastic no free, with compression using freeze control nodes but no memory reclamation (presented in Section 3.5.1);
- **CNF**, elastic no free, with compression using counter fields but no memory reclamation (presented in Section 3.5.2);
- **FHP**, elastic with hazard pointers, an elastic version using freeze nodes with memory reclamation using hazard pointers;
- **FHPA**, same as above but uses asymmetric barriers with hazard pointers instead;
- **HHL**, static with the hazard hash and level memory reclamation method, as proposed by Moreno et al. [38].

We start the chapter with a brief explanation of the program used to perform the benchmarks (section 5.1). Then, we describe the specifications of the server hardware used (section 5.2). Next, we determine the optimal configuration values for the LFHT,

such as: the length of collision chains, or the memory allocator to use (section 5.3). In the two last sections, 5.4 and 5.5, we compare each version of LFHT, with and without compression and memory reclamation.

5.1 Benchmark Program

We have implemented benchmarks in order to compare the performance in throughput of the different versions of LFHT. The program will gather a number of measurements throughout the experiments, such as: the number of hash nodes traversed, or the number of collision chains expanded.

Before each test begins, the hash map is filled with a number of keys. This is done to make sure that each function call successfully modifies the map. For example, it is impossible to remove keys, if the map is empty at the start of the benchmark. To mitigate this issue, we preemptively add all keys which will be issued for removal during the benchmark. The same thing is done for the keys we want to search. To guarantee that threads insert keys successfully we must make sure the map does not contain any keys which we plan on inserting, during the benchmark. During the test, threads will concurrently call a combination of the following map functions: insert, remove and search for keys on the map. The test terminates when all threads successfully called a predetermined number of map functions.

Keys are generated using a *pseudorandom number generator* (PRNG). Using a PRNG as a key generator will make it such that threads often operate in different locations on the map simultaneously, reducing the contention and improving throughput, but will not prevent threads from generating the same keys from time to time.

We want to understand how the throughput will grow when increasing the number of cooperating threads. A test will be partitioned into even work loads each attributed to a thread. Thus, the more threads we add, the more we can divide the total work of the test to be performed in parallel and, thus, finish each test more quickly.

5.1.1 Metrics

Each thread has a set of local counters which it updates during the experiments. Each counter can contain useful information in the analysis of the performance of the different implementations. Each thread keeps track of the number of expansions performed during

the experiments, the average path length, the maximum depth of the map traversed, the number of retrials due to a CAS failure, the number of hash nodes compressed, and the number of compressions aborted.

Because each thread has its own private ownership to a counter, no synchronization is needed to update measurements. However, if counters are placed in a shared cache line (in this case, the same 64 byte block) along with hash node references, it may cause a false sharing problem. That is, when a thread updates a counter it will affect other's caches needlessly. To prevent this, counters should be allocated in their own contiguous memory block and aligned to the size of a cache line, one per thread.

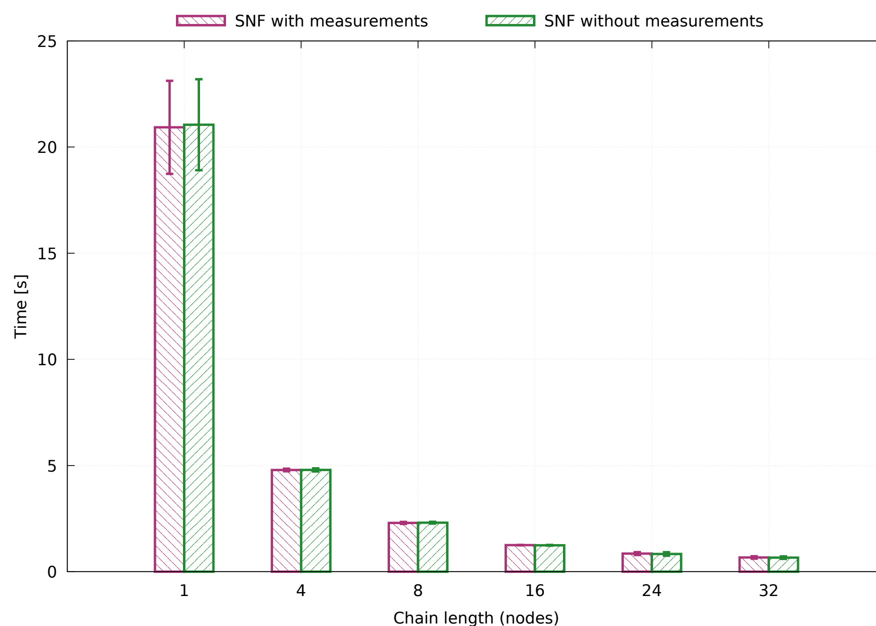
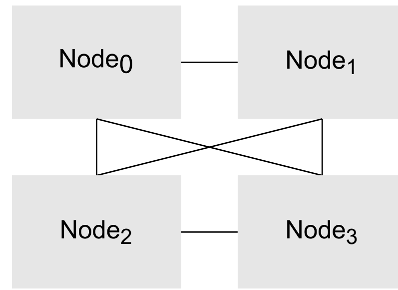


FIGURE 5.1: Overhead caused by gathering additional statistics

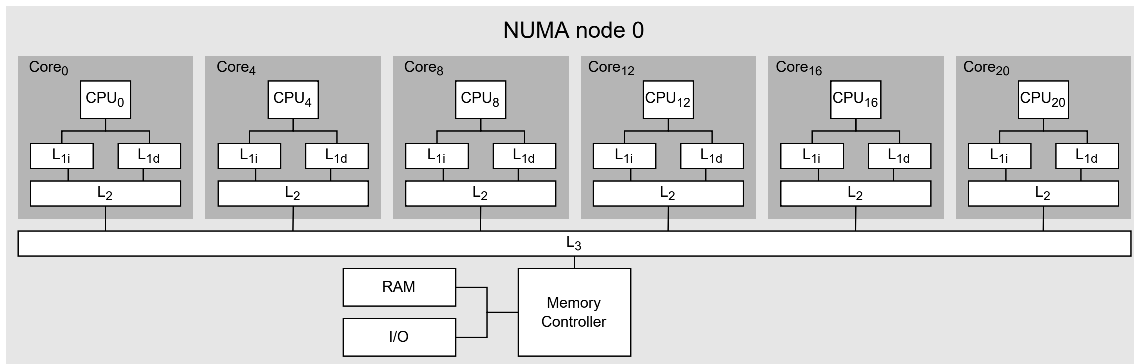
Figure 5.1 compares the map library with and without gathering additional metrics. The resulting overhead is negligible.

5.2 Machine Specifications

The benchmarks were performed on a machine with the architecture: Six-Core AMD Opteron Processor 8425 HE, which is illustrated in Fig. 5.2.



(A) Topology of the NUMA nodes



(B) Internal architecture of (NUMA) node 0

FIGURE 5.2: Architecture visual description of our benchmark machine

The L_1 data cache has $1.5MiB$ of size, L_2 has $12MiB$, and L_3 has $20MiB$. Each CPU has its own L_1 and L_2 caches. The L_1 cache is separated into two caches, one dedicated to instructions and other dedicated to data.

The machine has 4 NUMA nodes, with 6 cores each, so the message latency is much higher when cores from different nodes have to synchronize. For this reason, when we increase the number of cores to 7, 13, and 19, the number of cache misses may increase.

The CPU numbers of each node are not arranged in numeric order. Node 0 has CPUs 0, 4, 8, 12, 16, and 20; node 1 has CPUs 1, 5, 9, 13, 17, and 21; node 2 has CPUs 2, 6, 10, 14, 18, and 22; node 3 has CPUs 3, 7, 11, 15, 19, and 23.

5.3 Hash Map Parameters

Before comparing implementations of the hash trie map, we need to determine the optimal map configurations. For example, the chunk size to use throughout the testing phase.

5.3.1 Chain Length

Different keys can collide in the same bucket entry, forming a collision chain. When collision chains get to a certain length, expansion takes place to reduce the average path length. The maximum length of the chain can be configured, which determines how frequent expansions will be.

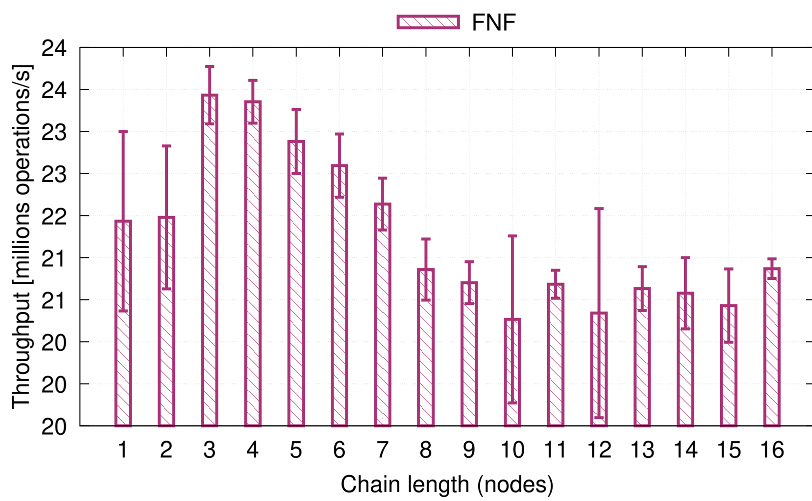
Figure 5.3 illustrates the results, in throughput, when varying the chain length. With lengths of 1 to 3 nodes, the sheer number of expansions and high average path length contribute to lower throughput, as illustrated in Fig. 5.3b and 5.3c. With chains longer than 5 nodes, however, it is the high average path length that causes the slowdown, as demonstrated in Fig. 5.3c. The better value seems to be a size of 4, which shows the highest throughput of all samples, with the smallest error, and low average path length.

5.3.2 Chunk Size

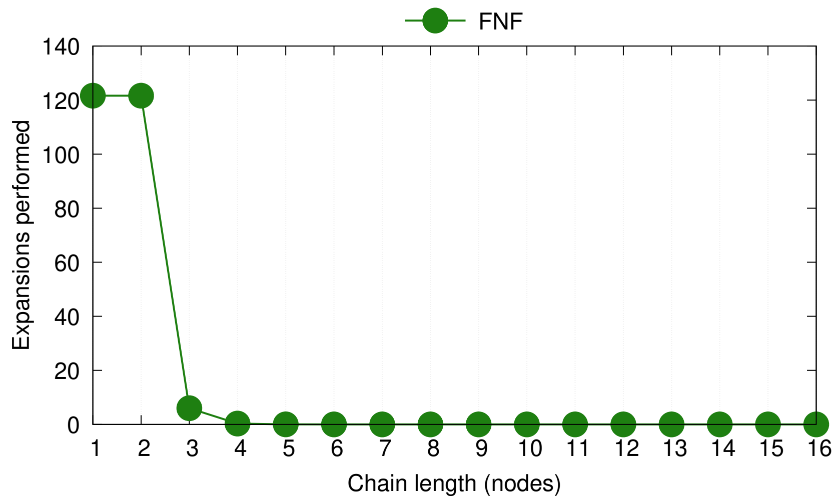
The chunk size determines the size of the hash nodes. Remember, 2^W is the number of buckets in a hash node where W is the chunk size.

A small chunk size will increase the number of expansions, compressions, hash collisions, and average path length. This overhead can be observed in samples using chunk sizes of 1 to 3 bits, as illustrated in Fig. 5.4a. On the other hand, increasing the chunk size will increase the memory consumption, as can be observed in Fig. 5.4b. Too big of a chunk size will also cause lower throughput because it will take multiple cache misses to load the buckets of a single hash node.

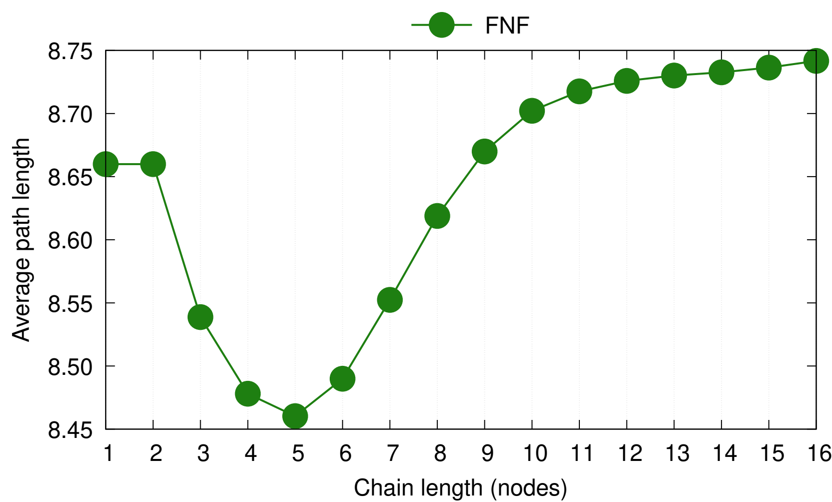
With the given results, there is no clear optimal chunk size value. However, we want to prevent low throughput and high memory consumption. Thus, chunk sizes of either 4, 5 or 7 may be the best option. We shall use chunk sizes of 5 from here on out. Thus, hash nodes will have $2^5 = 32$ bucket entries each, since it will force a higher number of expansions and compressions which are the main metrics we want to study.



(A) Throughput

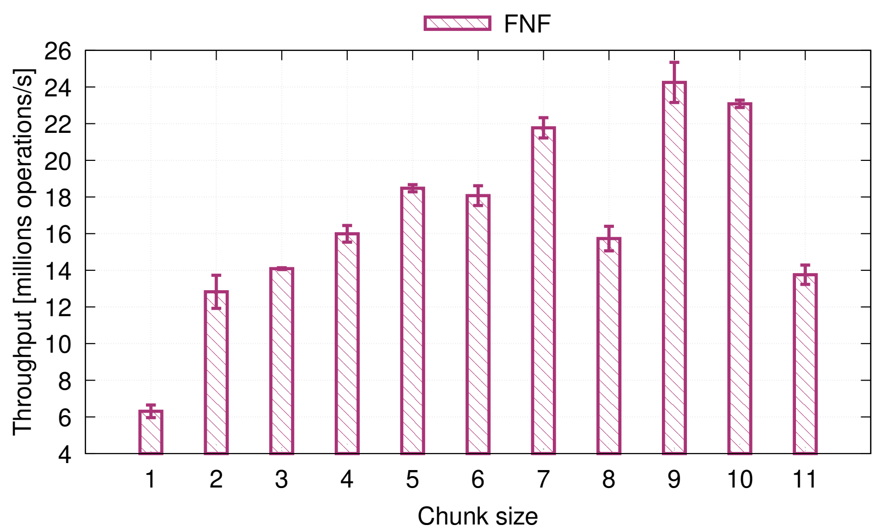


(B) Expansions performed

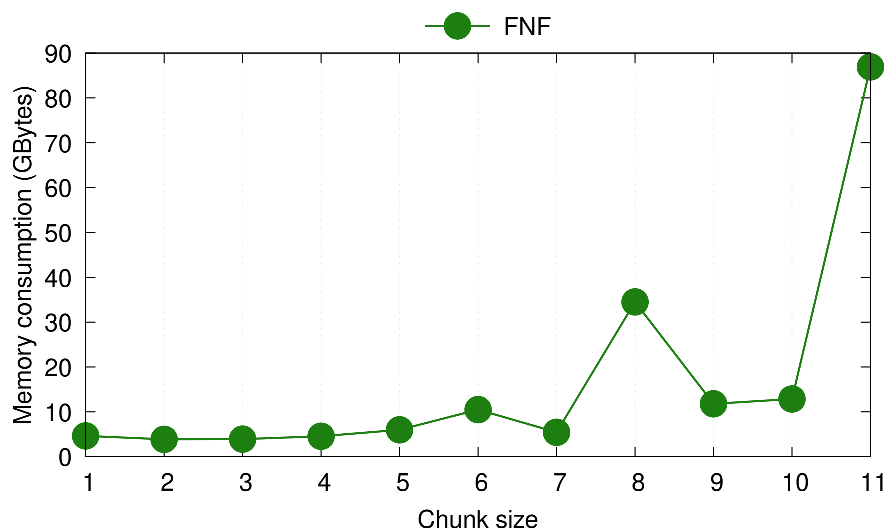


(C) Average path length

FIGURE 5.3: Variable chain length benchmark - 2^{24} nodes; average of 10 samples; chunk size 4



(A) Throughput



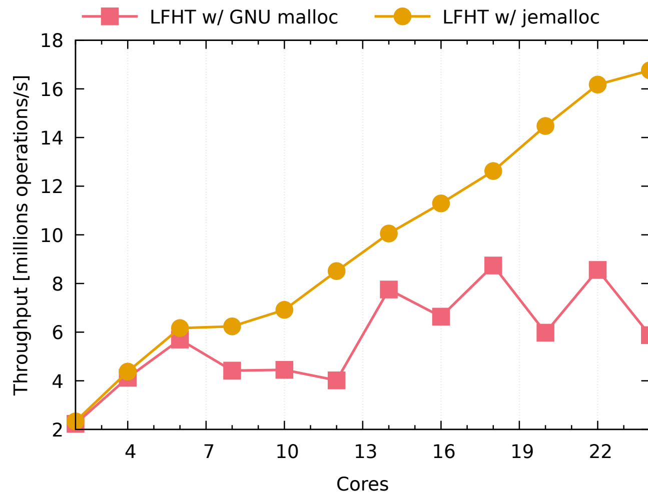
(B) Maximum memory consumption, in gigabytes

FIGURE 5.4: Variable chunk size benchmark - 2^{24} nodes; average of 10 samples; chain length 4

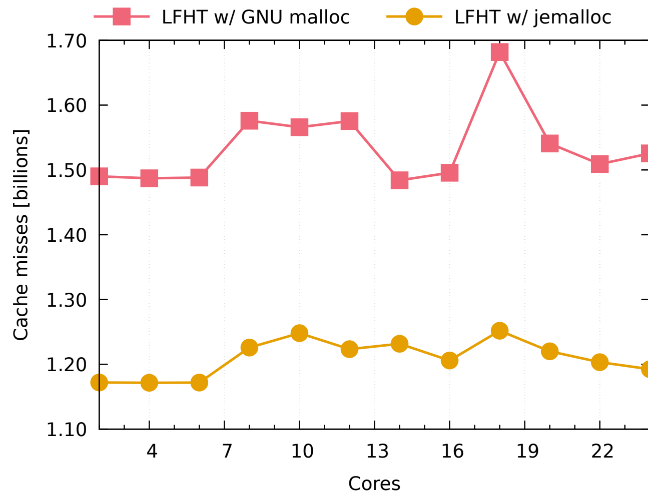
5.3.3 Memory Allocator

The standard library *malloc* function on Linux systems is **blocking**, meaning it will halt the progress of multiple processors concurrently requesting memory blocks. Because the memory allocator is frequently called during the execution of our hash map, our algorithm will not be able to scale as we increase the number of cores. To prevent this problem we used another *malloc* implementation called *jemalloc*, as documented by Evans [18].

The *jemalloc* project was created with the intention of making a multiprocessor-scalable memory allocator for the FreeBSD operating system. Although *jemalloc* is also blocking, we see significant improvement in throughput as shown in Fig. 5.5a. As seen in Fig. 5.5b, the new memory allocator helps reduce the number of cache misses. More importantly, the new allocator implements a fine-grained locking mechanism to prevent threads from blocking. When using the original allocator, we will not be able to exploit parallelism since threads will continuously queue up when calling the allocator.



(A) Throughput measurement



(B) Cache misses measurement

FIGURE 5.5: Comparing the original memory allocator with *jemalloc*

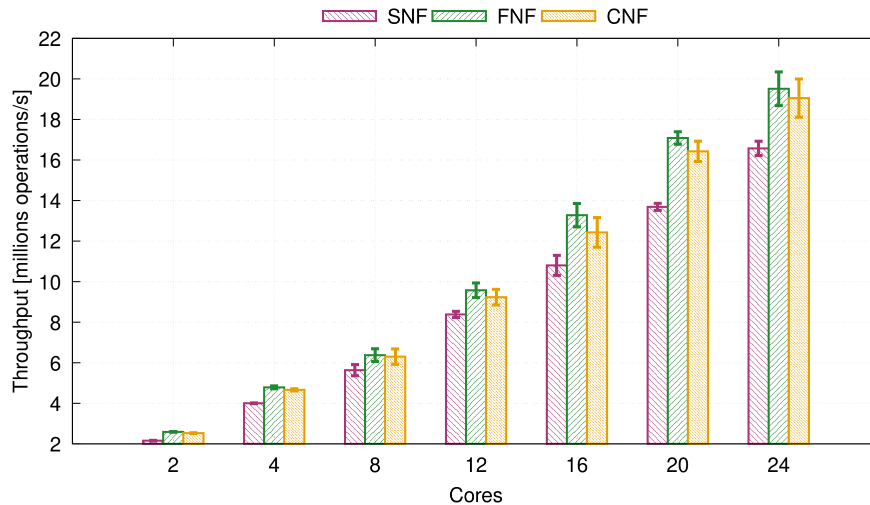
5.4 Compression

In this section we study the impact of compression on throughput.

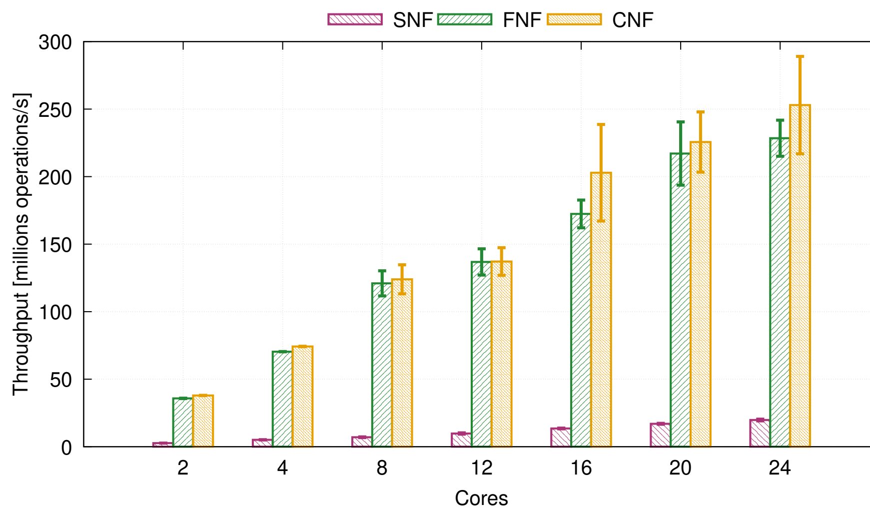
To aggravate the differences between the elastic and static versions, we fill the map with keys and completely empty it before starting the benchmarks illustrated in Fig. 5.6. This approach to testing is the one used in the original paper by Areias and Rocha [37], which we would like to closely follow.

The presented histograms indicate the throughput, i.e. the number of operations performed per second. The higher the value, the less time it took to fully complete the benchmark. The results obtained in Fig. 5.6 are similar to the ones shown in the paper [37].

An additional benefit to compression, which is not apparent in these benchmarks, is the fact that its memory consumption can be potentially lower. When a memory reclamation method is implemented, hash nodes may also be reclaimed, therefore, saving space.



(A) Reinserting nodes after emptying the map

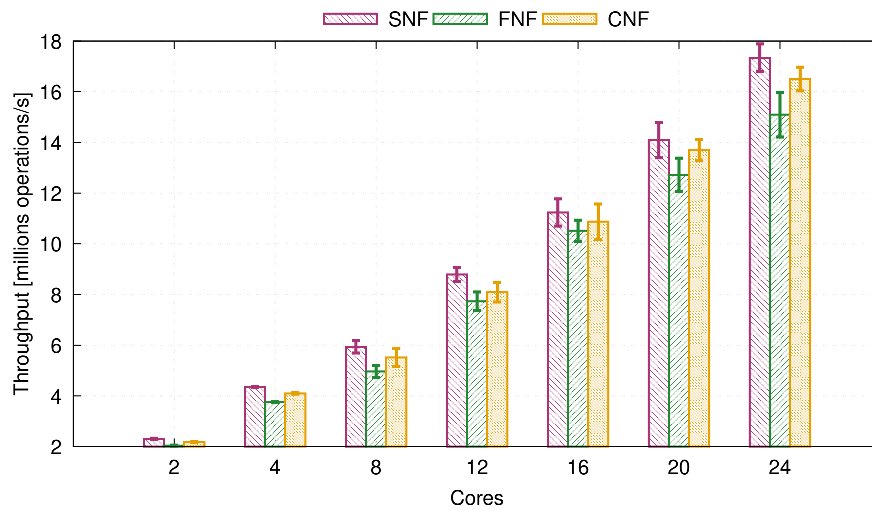


(B) Searching for nodes after emptying the map

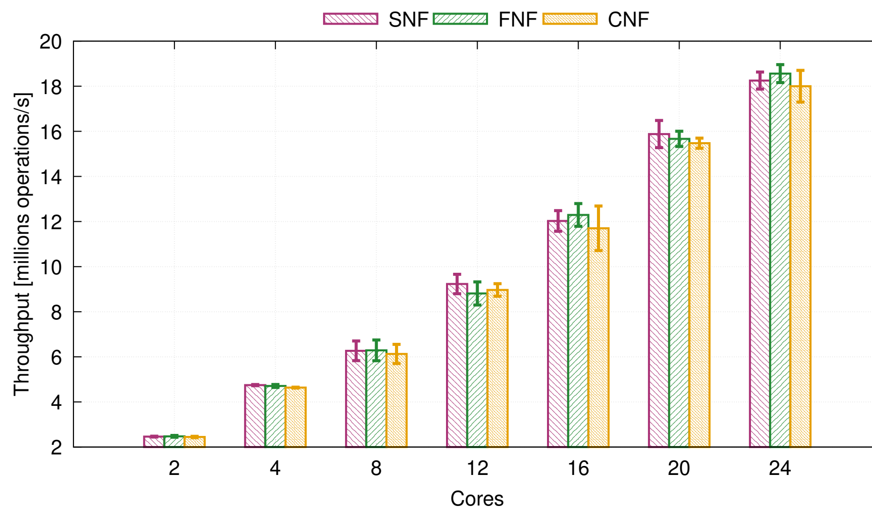
FIGURE 5.6: Compression benefits - 2^{24} nodes, average of 10 samples

Before the test begins, the map is filled with a number of nodes and soon completely emptied out. For the static version (**SNF**), all leaf nodes will be removed but the hash nodes will remain. Thus, the tree will be left intact. For this reason, the elastic maps (**FNF** and **CNF**) have an advantage right at the start which results in a lower average path length all around. The average path length is an important metric for determining the performance of the lookup function during the benchmark program.

When reinserting all nodes (Fig. 5.6a), the static map (**SNF**) will need to traverse an already built hierarchy of hash nodes, resulting in a higher average path length, making it slower in comparison to the elastic implementations (**FNF** and **CNF**). Since the hierarchy is completely built, the static map does not expand any nodes during the benchmark. The results show us that even with the added overhead of expansion, by keeping a low average path length, the elastic versions are faster. Finally, when searching for keys in an empty map, the average path length of the elastic map is zero since only the hash root node is checked. On the other hand, the static map needs to traverse the complete hierarchy tree for every search. Figure 5.6b illustrates the poor performance of the static map demonstrating the importance of keeping a low average path length.



(A) Removing nodes in a filled map



(B) Searching for nodes in filled map

FIGURE 5.7: Compression overheads - 2^{24} nodes, average of 10 samples, chain length and chunk size of 4

Figure 5.7 shows the overheads of compression when the map is filled with a number of keys. As expected, the compression mechanism introduces additional work and synchronization to the removal operation, which is illustrated in Fig. 5.7a, where the static version (**SNF**) outperforms both elastic versions. On the other hand, the test of Fig. 5.7b shows no differences between versions because the same nodes are present in the map before the start of the test.

5.5 Memory Reclamation

We can lower the memory consumption of our hash map by implementing a memory reclamation method. However, it is not clear how much of an impact to speedup this memory reclamation method is. In this section, we analyse the results of the benchmark program comparing our proposed memory reclamation method, based on hazard pointers (**FHP** and **FHPA**), with the base hash map implementation (**SNF**) which does not reclaim memory. We also include an alternative memory reclamation method (**HHL**), designed by Moreno et al. [38] and described in section 4.3.2. Again, this memory reclamation method does support map compression and, therefore, does not free the memory of hash nodes.

We could have implemented the hazard pointers method to the version of the map capable of compressing using a counter. However, the higher number of cache misses in comparison with the version using *freeze nodes*, illustrated in Fig. 5.8, would aggravate the overhead of the hazard pointers method.

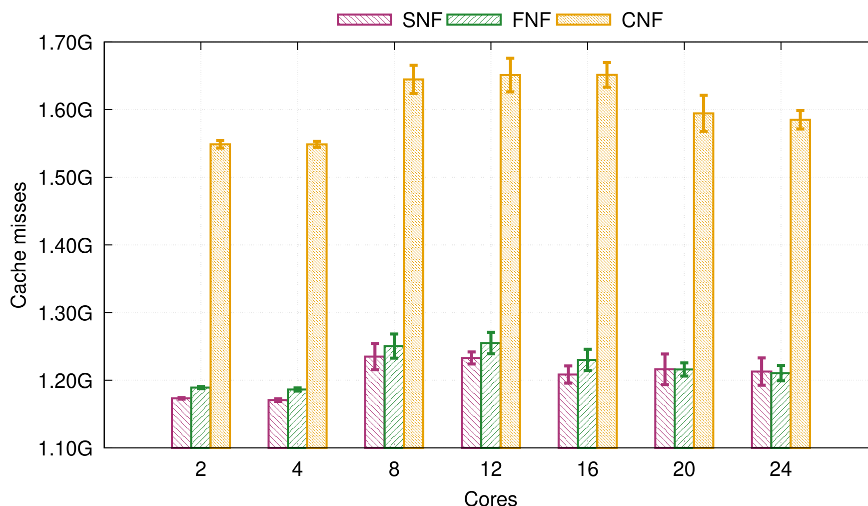


FIGURE 5.8: Cache misses - 25% of insertions; 25% of removals; 50% of searches

The main benefit of implementing a hazard pointers memory reclamation method is that hash nodes can have their memory reclaimed. Therefore, the memory consumption of our version is lower than previous versions, as illustrated in Fig. 5.9. We measured memory consumption by counting the allocated memory at the end of the test. Versions which are capable of reclaiming memory will have less memory allocated by the end of the test.

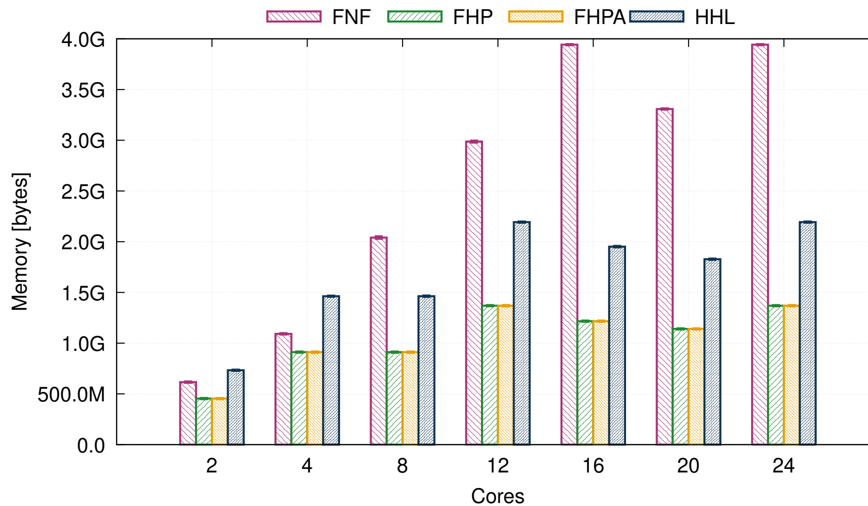


FIGURE 5.9: Memory consumption - 25% of insertions; 25% of removals; 50% of searches

Figure 5.10 shows the results of the benchmark program when comparing the four memory reclamation methods.

HHL has a lower number of cache misses compared to **FHPA**. However, the lower average path length of our version makes it the version with lower throughput.

HHL applies, on average, two protections per lookup, a contrast to our method which protects every node traversed through. To reiterate, each protection is done with a strong memory barrier in the original hazard pointers implementation (**FHP**). The system call *mem_barrier* of the Linux kernel helps mitigate this issue. The system call implements an asymmetric barrier, which **FHPA** uses. With this function, every hazard pointer protection has the lowest memory ordering constraints, although a compiler barrier must still be used.

With asymmetric barriers, the number of cache misses does not change. However, during the memory reclamation procedure, which occurs much less frequently, caches will be forced to synchronize using asymmetric barriers, i.e. CPUs will be forced to flush their store buffers and invalidate queues. For both our implementations using the hazard pointers method, it is only after removing 10000 nodes that the reclamation procedure

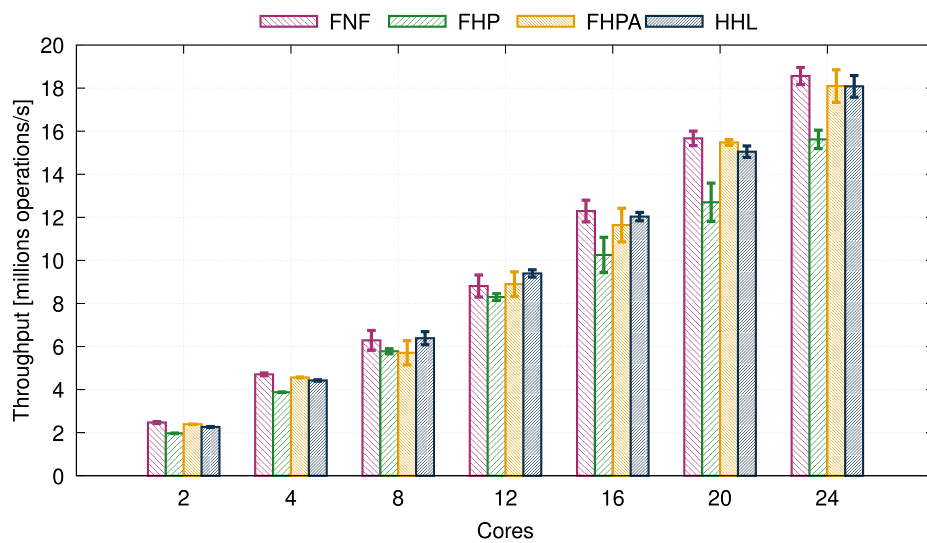
will be called. Otherwise, the hazard pointer protection used in the common code path of the lookup function will use weak memory barriers.

During lookup, each thread will push changes onto their own store buffers, but the progress of traversing the tree will not be hindered. With symmetric barriers, on the other hand, every node traversed will force the store buffers and invalidate queues to flush, frequently stalling the CPU. As a demonstration, the version **FHP**, which does not use this system call, has the same number of cache misses as **FHPA**, but the lowest throughput of all versions.

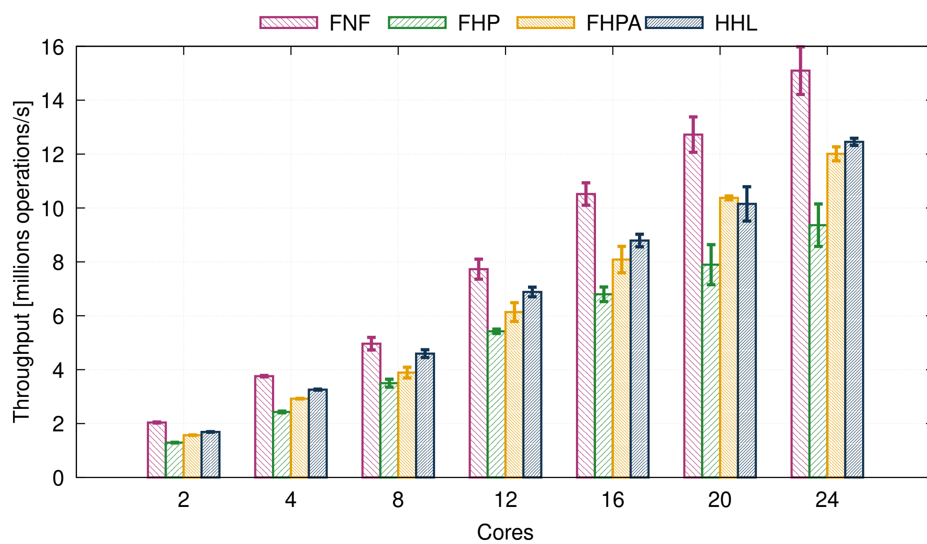
The removal operation (Fig. 5.10b) shows a significant slowdown of the elastic versions, especially the ones using hazard pointers. In most cases, however, the number of search operations outnumbers insertions and removals, and a certain number of removals implies that at least the same number of insertions occurred previously.

However, during the test where we simultaneously insert and search for keys, as shown in Fig. 5.10a, **FHPA** has lower throughput compared to both **FHP** and **HHL**.

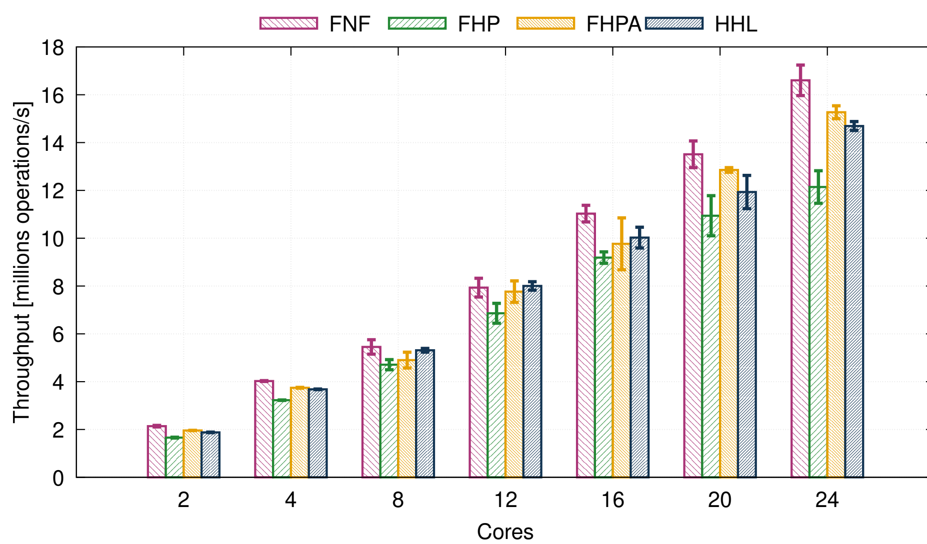
In Fig. 5.10c we perform mostly searches while simultaneously inserting and removing nodes. The results show **FHPA** just barely outperforming **HHL**, in throughput. In addition to this, **FHPA** is capable of reclaiming the memory of hash nodes giving it also an advantage in memory consumption compared to the other versions, as illustrated in Fig. 5.9.



(A) 100% of searches



(B) 100% of removals



(C) 25% of insertions; 25% of removals; 50% of searches

FIGURE 5.10: Memory reclamation method throughput benchmark - 2^{24} nodes, average of 10 samples

Chapter 6

Conclusions

We have proposed a memory reclamation method for a lock-free hash map data structure named LFHT. To the best of our knowledge, this is the first implementation of a memory reclamation method compatible with LFHT's compression operation, thus, being able to reclaim memory for both types of nodes: key-value pair nodes, or internal tree hash nodes. On top of having lower memory footprint, our design showed similar throughput when compared to the state-of-the-art method (*HHL*), and can even outperform it in some scenarios, despite the fact that *HHL* does not support compression of the hash map.

We have designed our own benchmark suite in order to determine whether or not the proposed method was more efficient in execution time and memory consumption. This way, we could gather detailed statistics of the map's behavior throughout the tests, such as: the number of expansions, or the number of nodes reclaimed. These metrics allowed us to reach conclusions regarding the results of the benchmarks, such as how the original hazard pointers method contributed to an increase of cache misses and cache synchronization which was detrimental to the performance of the map, in throughput.

The machine used to run the benchmarks uses processors which follow the *Intel x86* memory model, which is one of the most strict compared to other architectures. In fact, Intel's manual of the architecture [22] provides the following details:

1. When reordering instructions, the CPU cannot reorder **loads** with other **loads**;
2. When reordering instructions, the CPU cannot reorder **stores** with other **stores**;
3. **Stores** to the same memory location have total order.

On the other hand, architectures such as *ARM* and *IBM POWER* have more relaxed memory models compared to *x86*, as described by Maranget et al. [24]. Running the benchmarks and tests on a machine with *ARM* or *IBM POWER* architectures would likely give higher throughput results of our implementation.

For future work, we plan on extending our reclamation method to support back expansion, as proposed by Areias and Rocha [37]. We also plan to compare our memory reclamation method against the *Automatic Optimistic Access* [28] and the *Free Access* [33] methods. Additionally, we could try and apply *asymmetric barriers* not just to hazard pointers, but to other operations as well.

Appendix A

SNF - benchmark data

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2155945.7208566	608207315.2000000	1173206076.5999999	1.0000000	6.4533541
4	4101745.0954660	1081161545.5999999	1170689265.3000000	1.9024146	6.4520786
8	5604737.4135745	2030616806.4000001	1234913768.0999999	2.5969620	6.4367994
12	8098146.3260336	2977520220.8000002	1232882041.2000000	3.7479937	6.4532223
16	10766903.6961805	3925356732.8000002	1208404407.0999999	4.9740594	6.4292695
20	13360564.6082560	3293017344.0000000	1216128106.7000000	6.1772027	6.4528194
24	16403496.4465333	3925359788.8000002	1212777383.9000001	7.6017116	6.4288990

TABLE A.1: SNF - 25% removals, 25% insertions, 50% searches

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	1985968.4820043	1008242902.4000000	1189207521.0000000	1.0000000	6.1135070
4	3712412.2392347	1482040809.5999999	1190396601.2000000	1.8692210	6.1135100
8	5000059.4559875	2428228524.8000002	1248908281.0999999	2.5048484	6.1135030
12	7193376.8445235	3376742393.5999999	1255147046.0999999	3.6004772	6.1135068
16	9827147.3762232	4326053286.3999996	1227174630.3000000	4.9185357	6.1135013
20	12640886.7700754	3693429561.5999999	1220200615.0999999	6.3351736	6.1135068
24	14788371.0491096	4326054374.3999996	1208543326.4000001	7.4338257	6.1135030

TABLE A.2: SNF - 100% insertions

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2305738.4521411	472561536.0000000	1161987284.8000000	1.0000000	6.7729508
4	4352309.6917897	946996752.0000000	1159689019.5000000	1.8877840	6.7699055
8	5935052.6380330	1895471715.2000000	1227909170.5999999	2.5700874	6.7674759
12	8790220.1904819	2840964054.4000001	1216025239.2000000	3.8093017	6.7760619
16	11236605.5981433	3791268934.4000001	1202394752.7000000	4.8625924	6.7520755
20	14092595.6503559	3158792044.8000002	1193127471.5000000	6.0977252	6.7752835
24	17338062.8121213	3791269920.0000000	1185678346.0999999	7.5127596	6.7515025

TABLE A.3: SNF - 100% removals

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2462864.5784391	474406806.4000000	1168739278.7000000	1.0000000	6.9478699
4	4746500.5805565	946668355.2000000	1160292703.4000001	1.9272867	6.9477171
8	6266840.3464215	1897344585.5999999	1229306384.4000001	2.5305188	6.9463145
12	9229922.0692209	2843555196.8000002	1223470830.7000000	3.7392229	6.9462622
16	12025433.5735942	3791268294.4000001	1199468265.0000000	4.8755376	6.9474273
20	15878376.4684238	3159078617.5999999	1193898921.7000000	6.4373617	6.9476326
24	18251010.8240982	3791269433.5999999	1186133603.2000000	7.4077072	6.9477791

TABLE A.4: SNF - 100% searches of inserted keys

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2410214.4007048	473627750.4000000	1167473994.3000000	1.0000000	6.0373670
4	4614330.4527202	947489788.8000000	1164207285.0000000	1.9144644	6.0373670
8	5983346.2790048	1894587683.2000000	1224389270.5999999	2.4735290	6.0373670
12	8872086.5915894	2847594291.1999998	1220360288.2000000	3.6649910	6.0373670
16	12192435.1421641	3791268320.0000000	1202169357.8000000	5.0568507	6.0373670
20	15241081.9284931	3158861302.4000001	1197566042.0999999	6.3046558	6.0373670
24	18189678.8976500	3791269651.1999998	1190935640.8000000	7.5448443	6.0373670

TABLE A.5: SNF - 100% searches of keys not inserted on the map

Appendix B

FNF - benchmark data

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2138837.0748578	615879337.6000000	1189112802.9000001	0.9918524	6.4298037
4	4025820.8766979	1091716329.5999999	1186240249.2000000	1.8672930	6.4282263
8	5452748.0374609	2040785929.5999999	1250449164.5999999	2.5211742	6.4125009
12	7933197.1420155	2986908614.4000001	1254941468.7000000	3.6706111	6.4301125
16	11028463.7623749	3942269392.0000000	1230080675.0000000	5.1103367	6.4064406
20	13510006.0891823	3308020137.5999999	1215852546.0000000	6.2548967	6.4295906
24	16604149.7999531	3942271449.5999999	1210463250.5000000	7.6895304	6.4059393

TABLE B.1: FNF - 25% removals, 25% insertions, 50% searches

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	1980013.2308766	1015089987.2000000	1202352207.8000000	0.9970254	6.0897752
4	3722212.9269475	1492213504.0000000	1199405473.0999999	1.8740938	6.0897764
8	4955256.2903333	2443353772.8000002	1260161464.7000000	2.4748760	6.0897712
12	7300228.3319084	3390547552.0000000	1260009081.9000001	3.6628436	6.0897695
16	10354377.4398093	4344033862.3999996	1242472882.7000000	5.2128964	6.0897697
20	11863996.8236285	3710952588.8000002	1231650318.2000000	5.8916720	6.0897715
24	14604592.4687550	4344034976.0000000	1228782464.8000000	7.3342416	6.0897651

TABLE B.2: FNF - 100% insertions

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2038391.0892221	534974748.8000000	1189735260.0000000	0.8840782	6.7698220
4	3758246.9892912	1009610147.2000000	1187152658.5999999	1.6300752	6.7673469
8	4963786.5062881	1960617267.2000000	1241884292.0999999	2.1482770	6.7661820
12	7730133.1270698	2910480918.4000001	1250841149.7000000	3.3443506	6.7742787
16	10517731.1839474	3862178675.1999998	1226543944.4000001	4.5542645	6.7487237
20	12723145.6808034	3227434592.0000000	1216160460.2000000	5.5030955	6.7731124
24	15096881.7952073	3862180032.0000000	1212615797.4000001	6.5245739	6.7489602

TABLE B.3: FNF - 100% removals

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2474213.4314676	475079619.2000000	1176188011.2000000	1.0044058	6.9470988
4	4707994.7036259	949886006.4000000	1173834713.5999999	1.9113364	6.9471412
8	6287490.3862582	1902209804.8000000	1235708318.5999999	2.5380421	6.9445874
12	8810513.3863472	2852268732.8000002	1226197098.7000000	3.5646458	6.9447086
16	12291831.7060692	3802080102.4000001	1208745518.5999999	4.9821850	6.9468168
20	15665663.1459981	3167928048.0000000	1204680667.0000000	6.3580916	6.9470391
24	18559779.3794144	3802081689.5999999	1202238771.0000000	7.5327596	6.9470470

TABLE B.4: FNF - 100% searches of inserted keys

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2427391.6091418	475741222.4000000	1176824273.2000000	1.0071786	5.9914980
4	4638111.4850845	948079392.0000000	1177248301.0000000	1.9245281	5.9914980
8	6154495.7317285	1899473539.2000000	1240523045.3000000	2.5447544	5.9914980
12	9066694.9264710	2852927804.8000002	1238809858.0999999	3.7562278	5.9914980
16	12052226.6161647	3802080243.1999998	1209692314.3000000	4.9909536	5.9914980
20	15580937.7064202	3167631561.5999999	1203707092.3000000	6.4583765	5.9914980
24	18610686.2312442	3802081356.8000002	1201493949.4000001	7.7116580	5.9914980

TABLE B.5: FNF - 100% searches of keys not inserted on the map

Appendix C

CNF - benchmark data

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2116309.6306887	607951036.8000000	1548712707.5000000	0.9816180	6.4301675
4	3987396.2274189	1080655529.5999999	1548607114.3000000	1.8494815	6.4284317
8	5284251.1264904	2030471305.5999999	1644651443.5000000	2.4415748	6.4154294
12	7771265.2430460	2977175155.1999998	1651077634.5000000	3.5981193	6.4295480
16	10293041.7977205	3925410147.1999998	1651359135.7000000	4.7447129	6.4061379
20	13141405.3755566	3293103494.4000001	1594444468.7000000	6.0793871	6.4295301
24	15746159.1784841	3925411523.1999998	1584953153.9000001	7.2877606	6.4059123

TABLE C.1: CNF - 25% removals, 25% insertions, 50% searches

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	1952382.9074307	1015100505.6000000	1599201469.0000000	0.9830314	6.0897752
4	3614950.8361104	1488815411.2000000	1596312755.0999999	1.8201222	6.0897767
8	4902162.4979232	2437957369.5999999	1695686902.4000001	2.4557827	6.0897709
12	7344791.7190484	3387241692.8000002	1703230164.5000000	3.6924345	6.0897696
16	9720283.1918723	4333221670.3999996	1705248756.0999999	4.8810719	6.0897708
20	12512449.8947217	3700949827.1999998	1641726954.0999999	6.2927582	6.0897724
24	14487713.6385671	4333222720.0000000	1644843960.3000000	7.2833001	6.0897659

TABLE C.2: CNF - 100% insertions

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2187602.6669837	473940371.2000000	1286241156.0000000	0.9488037	6.7699850
4	4096401.6268813	947967011.2000000	1278940392.2000000	1.7767589	6.7675148
8	5516482.5378918	1893141062.4000001	1367780760.5999999	2.3827428	6.7654358
12	8094804.4393454	2847673340.8000002	1366095107.4000001	3.5020794	6.7740716
16	10875527.8130425	3791268281.5999999	1365226785.0000000	4.6962125	6.7505205
20	13693096.6773185	3159473241.5999999	1310814300.0999999	5.9336232	6.7731290
24	16502329.0787444	3791269126.4000001	1313481672.8000000	7.1521482	6.7489764

TABLE C.3: CNF - 100% removals

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2447200.1641759	474083139.2000000	1282509735.7000000	0.9935557	6.9470216
4	4636075.7567094	945997324.8000000	1282246130.7000000	1.8824702	6.9470734
8	6127701.7191544	1893506134.4000001	1371505371.2000000	2.4758146	6.9452007
12	8965706.5706265	2843074739.1999998	1362403028.5000000	3.6369570	6.9448319
16	11700780.2187571	3791268524.8000002	1369738586.4000001	4.7125169	6.9466064
20	15473879.0871998	3158756982.4000001	1311906638.9000001	6.2819229	6.9470926
24	18000268.7729077	3791269600.0000000	1317223879.7000000	7.2972105	6.9470650

TABLE C.4: CNF - 100% searches of inserted keys

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2397040.1218851	473771936.0000000	1443174372.5000000	0.9946337	5.9914980
4	4585420.7068965	947924969.6000000	1445526300.7000000	1.9026706	5.9914980
8	6091837.8602861	1897397654.4000001	1535300754.5999999	2.5116518	5.9914980
12	8616530.9868526	2842210041.5999999	1540986394.8000000	3.5492182	5.9914980
16	11612291.5228447	3791268102.4000001	1549311400.5999999	4.8098885	5.9914980
20	15140433.7175006	3159215987.1999998	1480286996.0000000	6.2701782	5.9914980
24	18184146.6831650	3791269331.1999998	1492474737.5999999	7.5378925	5.9914980

TABLE C.5: CNF - 100% searches of keys not inserted on the map

Appendix D

SNF (empty map) - benchmark data

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2472154.0542228	607709417.6000000	1173206076.5999999	1.1466541	5.7725164
4	4660449.3666358	1081021820.8000000	1170689265.3000000	2.1616815	5.7725509
8	6295631.0328006	2027526953.5999999	1234913768.0999999	2.9125916	5.7725653
12	9440626.4636015	2976557558.4000001	1232882041.2000000	4.3775883	5.7726117
16	12760306.1432050	3925356796.8000002	1208404407.0999999	5.9165868	5.7727128
20	15925617.4937686	3293410508.8000002	1216128106.7000000	7.3809413	5.7726205
24	18726256.6522171	3925359379.1999998	1212777383.9000001	8.6796138	5.7725184

TABLE D.1: SNF (empty map) - 25% removals, 25% insertions, 50% searches

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2157581.1880885	1007913836.8000000	1189207521.0000000	1.0863792	5.8295280
4	4003806.7265529	1482767955.2000000	1190396601.2000000	2.0160767	5.8295304
8	5634179.2012418	2433536636.8000002	1248908281.0999999	2.8301066	5.8295237
12	8383412.5718636	3378394329.5999999	1255147046.0999999	4.2200756	5.8295276
16	10803681.3135824	4326053427.1999998	1227174630.3000000	5.4287362	5.8295221
20	13688941.6730192	3694602310.4000001	1220200615.0999999	6.8919814	5.8295274
24	16572776.1532599	4326054054.3999996	1208543326.4000001	8.3413914	5.8295237

TABLE D.2: SNF (empty map) - 100% insertions

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2729172.3413901	475032598.4000000	1161987284.8000000	1.1837659	5.7534510
4	5212589.8171618	946725984.0000000	1159689019.5000000	2.2608063	5.7534510
8	7056679.9483363	1895060051.2000000	1227909170.5999999	3.0571124	5.7534510
12	10325925.7367631	2841390944.0000000	1216025239.2000000	4.4754179	5.7534510
16	13706767.6255312	3791268652.8000002	1202394752.7000000	5.9439002	5.7534510
20	17412404.6609458	3159428240.0000000	1193127471.5000000	7.5498172	5.7534510
24	20413252.3126707	3791269907.1999998	1185678346.0999999	8.8491900	5.7534510

TABLE D.3: SNF (empty map) - 100% removals

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2766133.6091897	472491270.4000000	1168739278.7000000	1.1231580	5.7535350
4	5240530.0231329	946995075.2000000	1160292703.4000001	2.1277645	5.7535350
8	7000787.1112052	1899416163.2000000	1229306384.4000001	2.8343898	5.7535350
12	10274397.2342338	2845149078.4000001	1223470830.7000000	4.1674319	5.7535350
16	13865745.0258394	3791268396.8000002	1199468265.0000000	5.6273349	5.7535350
20	17796245.0219758	3159828284.8000002	1193898921.7000000	7.2254018	5.7535350
24	20382778.4843585	3791269868.8000002	1186133603.2000000	8.2631696	5.7535350

TABLE D.4: SNF (empty map) - 100% searches of inserted keys

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2673948.8880152	474144886.4000000	1167473994.3000000	1.1095670	5.7533900
4	5099880.4609351	946109987.2000000	1164207285.0000000	2.1162418	5.7533900
8	7017936.3456658	1897477545.5999999	1224389270.5999999	2.9078194	5.7533900
12	9783942.8083798	2843371008.0000000	1220360288.2000000	4.0492049	5.7533900
16	13513525.1924591	3791268422.4000001	1202169357.8000000	5.6051462	5.7533900
20	16961235.5756235	3159675824.0000000	1197566042.0999999	7.0353471	5.7533900
24	19773745.1139139	3791269420.8000002	1190935640.8000000	8.1968093	5.7533900

TABLE D.5: SNF (empty map) - 100% searches of keys not inserted on the map

Appendix E

FNF (empty map) - benchmark data

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	3637022.2005159	753626995.2000000	1189112802.9000001	1.6869287	5.1196338
4	6978829.5662838	1267728880.0000000	1186240249.2000000	3.2369559	5.1194623
8	9154368.9843462	2313464844.8000002	1250449164.5999999	4.2437040	5.1194692
12	14026987.9421681	3340018745.5999999	1254941468.7000000	6.4918888	5.1194428
16	18851716.8165846	4367191414.3999996	1230080675.0000000	8.7286617	5.1187329
20	24188570.3342797	3678250985.5999999	1215852546.0000000	11.2014247	5.1194162
24	27295258.4596901	4367180153.6000004	1210463250.5000000	12.5945751	5.1197993

TABLE E.1: FNF (empty map) - 25% removals, 25% insertions, 50% searches

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2590755.1126384	1278986428.8000000	1202352207.8000000	1.3045144	5.1700047
4	4787698.7167970	1793832272.0000000	1199405473.0999999	2.4103100	5.1699805
8	6374209.2808208	2824552627.1999998	1260161464.7000000	3.2018091	5.1699328
12	9574715.7522217	3877584963.1999998	1260009081.9000001	4.8144056	5.1700007
16	13278584.7528223	4889398060.8000002	1242472882.7000000	6.6736735	5.1699628
20	17084854.2139310	4200441123.1999998	1231650318.2000000	8.6002350	5.1699881
24	19513267.7026835	4889400646.3999996	1228782464.8000000	9.8078429	5.1699687

TABLE E.2: FNF (empty map) - 100% insertions

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	36229317.6012761	512948867.2000000	1189735260.0000000	15.7145845	0.0000000
4	70950335.6789131	1029572467.2000000	1187152658.5999999	30.7742310	0.0000000
8	124189708.3583228	2065927107.2000000	1241884292.0999999	53.6552546	0.0000000
12	130068586.8970316	3085380784.0000000	1250841149.7000000	55.6663704	0.0000000
16	169855649.9427042	4126671929.5999999	1226543944.4000001	71.8986254	0.0000000
20	225264862.1299986	3437925011.1999998	1216160460.2000000	96.2260093	0.0000000
24	233095963.8286940	4126672672.0000000	1212615797.4000001	100.8810781	0.0000000

TABLE E.3: FNF (empty map) - 100% removals

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	40903612.9260384	513261737.6000000	1176188011.2000000	16.6038928	0.0000000
4	80444643.4341095	1032435264.0000000	1173834713.5999999	32.6643538	0.0000000
8	132939355.8708106	2060219737.5999999	1235708318.5999999	53.5472207	0.0000000
12	165599138.3083206	3096193216.0000000	1226197098.7000000	66.2155498	0.0000000
16	196353391.0362703	4126672006.4000001	1208745518.5999999	78.1891615	0.0000000
20	244003946.7063327	3436423945.5999999	1204680667.0000000	98.0677313	0.0000000
24	303807832.6474321	4126672774.4000001	1202238771.0000000	118.6297748	0.0000000

TABLE E.4: FNF (empty map) - 100% searches of inserted keys

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	35791197.5319518	516396281.6000000	1176824273.2000000	14.8517551	0.0000000
4	70364264.9737898	1030768758.4000000	1177248301.0000000	29.1984861	0.0000000
8	120972120.5727924	2063855072.0000000	1240523045.3000000	49.8799958	0.0000000
12	136824210.9520378	3100698249.5999999	1238809858.0999999	56.4816343	0.0000000
16	172374271.8916164	4126671788.8000002	1209692314.3000000	71.2819283	0.0000000
20	217106952.0666831	3437840256.0000000	1203707092.3000000	89.1749062	0.0000000
24	228450839.7452757	4126672915.1999998	1201493949.4000001	94.4843757	0.0000000

TABLE E.5: FNF (empty map) - 100% searches of keys not inserted on the map

Appendix F

CNF (empty map) - benchmark data

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	3597997.1174121	714123129.6000000	1548712707.5000000	1.6688236	5.1196444
4	6901484.1592387	1187247433.5999999	1548607114.3000000	3.2010334	5.1194502
8	9007432.6574521	2135302441.5999999	1644651443.5000000	4.1733169	5.1194560
12	13066943.4160703	3083787212.8000002	1651077634.5000000	6.0280559	5.1194756
16	18502708.2007002	4031788908.8000002	1651359135.7000000	8.5683085	5.1187236
20	23851868.3915671	3398689059.1999998	1594444468.7000000	11.0527674	5.1193659
24	27091431.3973140	4031776390.4000001	1584953153.9000001	12.5124505	5.1197941

TABLE F.1: CNF (empty map) - 25% removals, 25% insertions, 50% searches

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2530231.7852714	1236094332.8000000	1599201469.0000000	1.2740099	5.1700052
4	4665607.3059632	1708741686.4000001	1596312755.0999999	2.3490737	5.1699822
8	6301301.9843819	2657584057.5999999	1695686902.4000001	3.1612707	5.1699266
12	9236085.3327985	3605957145.5999999	1703230164.5000000	4.6423494	5.1700067
16	12430078.2131891	4553994528.0000000	1705248756.0999999	6.2370758	5.1699583
20	16427084.2501999	3922403856.0000000	1641726954.0999999	8.2640411	5.1699921
24	19053200.6084416	4553996960.0000000	1644843960.3000000	9.5687796	5.1699640

TABLE F.2: CNF (empty map) - 100% insertions

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	35711026.6342340	473734851.2000000	1286241156.0000000	15.4897965	0.0000000
4	70039571.3916017	946542816.0000000	1278940392.2000000	30.3797715	0.0000000
8	119647519.4334662	1894315737.5999999	1367780760.5999999	51.7714563	0.0000000
12	136358432.0094794	2843642662.4000001	1366095107.4000001	58.6638027	0.0000000
16	165919021.7471161	3791268332.8000002	1365226785.0000000	71.7628488	0.0000000
20	206940369.4005714	3159212780.8000002	1310814300.0999999	89.6289411	0.0000000
24	242437928.5468839	3791269651.1999998	1313481672.8000000	103.5223647	0.0000000

TABLE F.3: CNF (empty map) - 100% removals

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	42584163.6160411	472458572.8000000	1282509735.7000000	17.2462501	0.0000000
4	83843604.9168358	947701862.4000000	1282246130.7000000	33.9875808	0.0000000
8	140049447.2812652	1896487766.4000001	1371505371.2000000	56.5450594	0.0000000
12	166676620.7120861	2845398108.8000002	1362403028.5000000	66.0530549	0.0000000
16	188962665.9557773	3791268588.8000002	1369738586.4000001	76.4101101	0.0000000
20	248534046.6100695	3158781049.5999999	1311906638.9000001	99.4328248	0.0000000
24	272232303.8098804	3791269203.1999998	1317223879.7000000	109.8119514	0.0000000

TABLE F.4: CNF (empty map) - 100% searches of inserted keys

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	37946568.0231866	473808457.6000000	1443174372.5000000	15.7463239	0.0000000
4	74229375.6452983	948586188.8000000	1445526300.7000000	30.8022734	0.0000000
8	123959534.8990024	1895863286.4000001	1535300754.5999999	51.0311204	0.0000000
12	137127383.7873024	2843206057.5999999	1540986394.8000000	56.5991859	0.0000000
16	202880989.3158082	3791268320.0000000	1549311400.5999999	81.8886668	0.0000000
20	225608275.2627168	3159257209.5999999	1480286996.0000000	92.6585755	0.0000000
24	252980229.7142133	3791269728.0000000	1492474737.5999999	103.2922840	0.0000000

TABLE F.5: CNF (empty map) - 100% searches of keys not inserted on the map

Appendix G

FHP - benchmark data

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	1958165.6801031	453742028.8000000	1212223774.0000000	0.9082650	5.3581074
4	3743719.6594435	911489209.6000000	1210794325.2000000	1.7364429	5.3643905
8	4901822.0847252	911467782.4000000	1268922969.8000000	2.2633694	5.3645993
12	7766023.5933490	1369360227.2000000	1248175477.0000000	3.5883701	5.3400016
16	9765465.4006277	1216598160.0000000	1264232781.3000000	4.4630121	5.3704923
20	12854521.3452280	1140411094.4000001	1272845057.2000000	5.9620774	5.3697346
24	15269815.6561299	1369353811.2000000	1274985416.2000000	7.0804777	5.3412626

TABLE G.1: FHP - 25% removals, 25% insertions, 50% searches

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	1903693.0592635	994000934.4000000	1224710814.5999999	0.9585987	6.0897748
4	3586297.1114616	1451815116.8000000	1222374409.4000001	1.8058503	6.0897768
8	4772945.4319700	1451843715.2000000	1269736424.2000000	2.3780333	6.0897713
12	7477467.2010966	1909638966.4000001	1276388591.3000000	3.7632874	6.0897688
16	9237975.4073428	1756962793.5999999	1278265831.9000001	4.5816492	6.0897672
20	12914719.4435473	1680708006.4000001	1303015622.5000000	6.5029070	6.0897732
24	14481174.1606153	1909643500.8000000	1285091758.5000000	7.2678551	6.0897805

TABLE G.2: FHP - 100% insertions

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	1568189.8305500	184462410.0000000	1224574809.2000000	0.6801846	3.9235337
4	2924110.4187425	329052080.0000000	1224552530.8000000	1.2683279	3.9287411
8	3891078.6216237	328965123.2000000	1302632874.0000000	1.6834472	3.9294088
12	6137349.8109924	786862912.0000000	1288031426.0999999	2.6528553	3.9042455
16	8085632.6251631	634218140.8000000	1289937692.5000000	3.4921084	3.9341331
20	10374621.8951761	557970147.2000000	1310991295.2000000	4.4998291	3.9333813
24	12009524.4434678	786914768.0000000	1298620673.8000000	5.2067260	3.9090142

TABLE G.3: FHP - 100% removals

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2390641.1871109	457782678.4000000	1197938229.5000000	0.9707180	6.9477041
4	4561836.9964745	915574310.4000000	1196715932.0000000	1.8523113	6.9477049
8	5710213.5091007	915584713.6000000	1257897846.7000000	2.2965375	6.9475890
12	8897769.0718481	1373420128.0000000	1253618978.8000000	3.5966549	6.9475943
16	11640138.3997152	1220835545.5999999	1257647977.0999999	4.7011048	6.9470674
20	15476101.9806022	1144439504.0000000	1227603143.4000001	6.2836699	6.9472160
24	18092911.5741567	1373433068.8000000	1263598302.0000000	7.3325588	6.9472256

TABLE G.4: FHP - 100% searches of inserted keys

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2379298.1256871	457778099.2000000	1198783037.3000000	0.9873194	5.9914980
4	4531192.6951648	915578915.2000000	1197896140.4000001	1.8801634	5.9914980
8	6244943.4520223	915679513.6000000	1249070437.5000000	2.5839414	5.9914980
12	9247010.1275261	1373422614.4000001	1250414644.2000000	3.8354617	5.9915002
16	11626297.4298238	1220696425.5999999	1253009762.7000000	4.7985199	5.9915059
20	15444290.0123333	1144545459.2000000	1238460157.5000000	6.4079796	5.9915302
24	18513288.2428718	1373418966.4000001	1245017057.5999999	7.6808090	5.9914982

TABLE G.5: FHP - 100% searches of keys not inserted on the map

Appendix H

FHPA - benchmark data

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	1659691.1793200	453848630.4000000	1228159576.5000000	0.7697383	5.3572757
4	3225149.4888122	911565123.2000000	1226258343.0999999	1.4959336	5.3642677
8	4710003.2396420	911495603.2000000	1285600310.5000000	2.1799911	5.3645341
12	6860553.1650186	1369357260.8000000	1282184397.0000000	3.1697543	5.3399990
16	9188724.3093974	1216693574.4000001	1284698051.3000000	4.2591307	5.3705799
20	10940343.4070893	1140372601.5999999	1266548184.5999999	5.0408052	5.3700362
24	12141959.5355984	1369361734.4000001	1257149671.0999999	5.6142991	5.3413796

TABLE H.1: FHPA - 25% removals, 25% insertions, 50% searches

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	1675556.4675230	994091897.6000000	1243009297.3000000	0.8437048	6.0897750
4	3271771.2529591	1451876246.4000001	1240047120.4000001	1.6472156	6.0897768
8	4864718.2569200	1451769014.4000001	1288254510.2000000	2.4387399	6.0897717
12	7146225.6193981	1909636028.8000000	1301164139.4000001	3.5976099	6.0897697
16	8903546.0565352	1756943222.4000001	1304550277.5000000	4.4502005	6.0897714
20	11113767.3469573	1680685420.8000000	1288127181.7000000	5.5151860	6.0897769
24	12962511.0166798	1909638665.5999999	1274373488.4000001	6.5115570	6.0897725

TABLE H.2: FHPA - 100% insertions

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	1291515.8395841	184467440.0000000	1239528483.0000000	0.5601439	3.9235742
4	2429623.3729017	329099804.8000000	1241340017.8000000	1.0536891	3.9285618
8	3498688.9000276	329034009.6000000	1320780175.2000000	1.5148636	3.9288309
12	5423956.7066668	786869494.4000000	1304033111.3000000	2.3521350	3.9042319
16	6797975.2154481	634222934.4000000	1299016231.0000000	2.9437943	3.9339861
20	7895762.4328200	557918361.6000000	1303098379.8000000	3.3942450	3.9333539
24	9362098.4600946	786872496.0000000	1282611053.2000000	4.0313446	3.9078022

TABLE H.3: FHPA - 100% removals

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	1970133.9733243	457877036.8000000	1212821262.3000000	0.7999455	6.9476836
4	3876908.0901136	915662284.8000000	1212246097.3000000	1.5742164	6.9476801
8	5779041.8597701	915634048.0000000	1253952034.4000001	2.3456027	6.9476262
12	8296240.7920270	1373418838.4000001	1270811921.5999999	3.3675109	6.9476580
16	10252993.0033164	1220751913.5999999	1286862131.0000000	4.1352135	6.9471150
20	12697170.0965559	1144447059.2000000	1243073360.5000000	5.1301089	6.9472353
24	15617705.5339504	1373418124.8000000	1254812349.3000000	6.3368756	6.9472392

TABLE H.4: FHPA - 100% searches of inserted keys

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2004747.6199358	457869971.2000000	1215605602.4000001	0.8318918	5.9914980
4	3947961.9820388	915639427.2000000	1214249474.8000000	1.6382404	5.9914980
8	5904032.1449687	915649596.8000000	1258093872.2000000	2.4490883	5.9914981
12	8450306.0144158	1373417088.0000000	1263437257.4000001	3.5059405	5.9914980
16	10821262.7745216	1220854560.0000000	1277592367.5999999	4.4746780	5.9915165
20	13000072.1025862	1144463968.0000000	1256423235.3000000	5.3724111	5.9915028
24	15948339.7577081	1373422643.2000000	1262371945.2000000	6.6099770	5.9915192

TABLE H.5: FHPA - 100% searches of keys not inserted on the map

Appendix I

HHL - benchmark data

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	1878672.7707736	732939532.8000000	1222116370.8000000	0.8713639	6.7051144
4	3679494.5934059	1462684080.0000000	1222124614.8000000	1.7066682	6.7212879
8	5312633.2509869	1462937161.5999999	1297767526.0999999	2.4636670	6.7220617
12	8003524.7553944	2193070396.8000002	1280506882.0000000	3.7104467	6.6575300
16	10025315.6209682	1951288217.5999999	1279121423.8000000	4.6409903	6.7378125
20	11932148.6504466	1827544729.5999999	1255742653.7000000	5.5149214	6.7348869
24	14692509.9369307	2193083660.8000002	1257343282.2000000	6.8138364	6.6617525

TABLE I.1: HHL - 25% removals, 25% insertions, 50% searches

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	1756659.7385608	1265727126.4000001	1239646946.9000001	0.8845543	6.2683532
4	3395895.0454699	1995613465.5999999	1238759294.3000000	1.7099208	6.2683591
8	4993584.6624879	1994736873.5999999	1309655604.7000000	2.5121339	6.2683459
12	7231573.2814440	2725892304.0000000	1301018043.8000000	3.6346012	6.2683522
16	9737846.4800093	2481920659.1999998	1285708225.3000000	4.9027876	6.2683480
20	12226011.6133358	2360129523.1999998	1269622148.7000000	6.1561520	6.2683535
24	13985854.2097001	2725894624.0000000	1261929509.5000000	7.0347828	6.2683482

TABLE I.2: HHL - 100% insertions

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	1688345.3287437	204417033.6000000	1215090874.5000000	0.7322945	7.6281307
4	3258175.3999522	934438745.6000000	1214485214.8000000	1.4132005	7.6488386
8	4594957.6197266	934051952.0000000	1289453502.2000000	1.9910822	7.6519079
12	6883434.0726443	1664651712.0000000	1276300701.3000000	2.9837017	7.5513267
16	8791685.2607180	1421068102.4000001	1265550764.5000000	3.8105721	7.6701733
20	10151748.2899126	1299576086.4000001	1261771515.9000001	4.3849229	7.6684752
24	12454197.1395703	1664653888.0000000	1253882161.3000000	5.4014586	7.5674504

TABLE I.3: HHL - 100% removals

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2266995.9892553	730877977.6000000	1206811790.0000000	0.9204592	6.9576341
4	4424848.1286437	1460847808.0000000	1206136563.8000000	1.7966445	6.9576344
8	6385423.0221683	1460475881.5999999	1277430564.8000000	2.5862546	6.9576389
12	9393187.6520703	2191107376.0000000	1267008786.9000001	3.8129008	6.9576099
16	12033315.0812436	1947448121.5999999	1275811538.8000000	4.8849041	6.9562831
20	15047664.3090629	1826606569.5999999	1227210455.4000001	6.1082701	6.9564898
24	18080487.4006526	2191109440.0000000	1234704574.0000000	7.3359461	6.9563194

TABLE I.4: HHL - 100% searches of inserted keys

Cores	Throughput	Memory	Cache Misses	Speedup	APL
2	2160592.1858132	730827996.8000000	1214338842.0999999	0.8965596	6.1195780
4	4264545.2627437	1460797078.4000001	1213589912.0000000	1.7695245	6.1195780
8	6230745.3947817	1461051619.2000000	1275314829.0999999	2.5841695	6.1195780
12	8999779.0721615	2191107456.0000000	1272112121.8000000	3.7324628	6.1195780
16	11754834.3992749	1947138425.5999999	1272358895.0999999	4.8763415	6.1195780
20	14514242.6027871	1826239280.0000000	1238108204.8000000	6.0168488	6.1195780
24	17561214.1200988	2191109648.0000000	1232102639.7000000	7.2841472	6.1195780

TABLE I.5: HHL - 100% searches of keys not inserted on the map

Bibliography

[1] Rene De La Briandais. "File searching using variable length keys". In: *Papers presented at the the March 3-5, 1959, western joint computer conference*. 1959, pp. 295–298.

[2] Edward Fredkin. "Trie memory". In: *Communications of the ACM* 3.9 (1960), pp. 490–499.

[3] IBM. "System/370 principles of operation". In: *Order Number GA22-7000*. 1970.

[4] Jon Bentley, Don Knuth, and Doug McIlroy. "Programming pearls: A literate program". In: *Communications of the ACM* 29.6 (1986), pp. 471–483.

[5] Maurice Herlihy. "A methodology for implementing highly concurrent data structures". In: *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*. 1990, pp. 197–206.

[6] Jeffrey C Mogul and Anita Borg. "The effect of context switches on cache performance". In: *ACM SIGPLAN Notices* 26.4 (1991), pp. 75–84.

[7] Jon L Bentley and Robert Sedgewick. "Fast algorithms for sorting and searching strings". In: *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*. 1997, pp. 360–369.

[8] Stefan Nilsson and Gunnar Karlsson. "Fast address lookup for Internet routers". In: (1998).

[9] Phil Bagwell. *Fast and space efficient trie searches*. Tech. rep. 2000.

[10] Phil Bagwell. *Ideal hash trees*. Tech. rep. 2001.

[11] Timothy L Harris. "A pragmatic implementation of non-blocking linked-lists". In: *International Symposium on Distributed Computing*. Springer. 2001, pp. 300–314.

[12] Petar Maymounkov and David Mazieres. "Kademlia: A peer-to-peer information system based on the xor metric". In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 53–65.

[13] Maged M Michael. "High performance dynamic lock-free hash tables and list-based sets". In: *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. 2002, pp. 73–82.

[14] Keir Fraser. *Practical lock-freedom*. Tech. rep. University of Cambridge, Computer Laboratory, 2004.

[15] Maged M Michael. "Hazard pointers: Safe memory reclamation for lock-free objects". In: *IEEE Transactions on Parallel and Distributed Systems* 15.6 (2004), pp. 491–504.

[16] Chris Purcell and Tim Harris. "Non-blocking hashtables with open addressing". In: *International Symposium on Distributed Computing*. Springer. 2005, pp. 108–121.

[17] Mark Aiken et al. "Deconstructing process isolation". In: *Proceedings of the 2006 workshop on Memory system performance and correctness*. 2006, pp. 1–10.

[18] Jason Evans. "A scalable concurrent malloc (3) implementation for FreeBSD". In: *Proc. of the bsdcan conference, ottawa, canada*. 2006.

[19] Ori Shalev and Nir Shavit. "Split-ordered lists: Lock-free extensible hash tables". In: *Journal of the ACM (JACM)* 53.3 (2006), pp. 379–405.

[20] Chuanpeng Li, Chen Ding, and Kai Shen. "Quantifying the cost of context switch". In: *Proceedings of the 2007 workshop on Experimental computer science*. 2007, 2–es.

[21] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. "Hopscotch hashing". In: *International Symposium on Distributed Computing*. Springer. 2008, pp. 350–364.

[22] Part Guide. "Intel® 64 and ia-32 architectures software developer's manual". In: *Volume 3B: System programming Guide, Part* 2.11 (2011).

[23] Maurice Herlihy and Nir Shavit. "On the nature of progress". In: *International Conference On Principles Of Distributed Systems*. Springer. 2011, pp. 313–328.

[24] Luc Maranget, Susmit Sarkar, and Peter Sewell. "A tutorial introduction to the ARM and POWER relaxed memory models". In: *Draft available from http://www. cl. cam. ac. uk/~ pes20/ppc-supplemental/test7. pdf* (2012).

[25] Aleksandar Prokopec et al. "Concurrent tries with efficient non-blocking snapshots". In: *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. 2012, pp. 151–160.

[26] Nhan Nguyen and Philippas Tsigas. "Lock-free cuckoo hashing". In: *2014 IEEE 34th international conference on distributed computing systems*. IEEE. 2014, pp. 627–636.

[27] Linda Null and Julia Lobur. *Essentials of Computer Organization and Architecture*. Jones & Bartlett Publishers, 2014.

[28] N. Cohen and E. Petrank. "Automatic memory reclamation for lock-free data structures". In: *ACM SIGPLAN Notices* 50.10 (2015), pp. 260–279.

[29] D. Dice, M. Herlihy, and A. Kogan. "Fast Non-Intrusive Memory Reclamation for Highly-Concurrent Data Structures". In: *International Symposium on Memory Management*. 2016, pp. 36–45.

[30] M. Areias and R. Rocha. "Towards a Lock-Free, Fixed Size and Persistent Hash Map Design". In: *Proceedings of the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2017)*. Ed. by M. Valero and A. Melo. Campinas, Brazil: IEEE Computer Society, 2017, pp. 145–152.

[31] Paul E McKenney. "Is parallel programming hard, and, if so, what can you do about it?" In: *arXiv preprint arXiv:1701.00854* (2017).

[32] Miguel Areias and Ricardo Rocha. "On Extending a Fixed Size, Persistent and Lock-Free Hash Map Design to Store Sorted Keys". In: *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*. IEEE. 2018, pp. 415–422.

[33] Nachshon Cohen. "Every data structure deserves lock-free memory reclamation". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–24.

[34] P. Moreno. "Memory Reclamation Methods for Lock-Free Hash Tries". MSc Thesis. Portugal: University of Porto, 2018.

[35] Aleksandar Prokopec. "Cache-tries: Concurrent lock-free hash tries with constant-time operations". In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2018, pp. 137–151.

[36] Robert Kelly, Barak A Pearlmutter, and Phil Maguire. "Lock-free hopscotch hashing". In: *Symposium on Algorithmic Principles of Computer Systems*. SIAM. 2020, pp. 45–59.

[37] M. Areias and R. Rocha. "Towards an Elastic Lock-Free Hash Trie Design". In: *Proceedings of the 20th International Symposium on Parallel and Distributed Computing (ISPDC 2021)*. Ed. by B. Iancu and Ralf-Peter Mundani. Cluj-Napoca, Romania (online event): IEEE Computer Society, 2021, pp. –.

[38] P. Moreno, M. Areias, and R. Rocha. "On the Implementation of Memory Reclamation Methods in a Lock-Free Hash Trie Design". In: *Journal of Parallel and Distributed Computing* 155 (2021), pp. 1–13.