Memory Reclamation Methods for Lock-Free Hash Tries

Pedro Carvalho Moreno

Mestrado em Ciência de Computadores
Departamento de Ciência de Computadores
2018

Orientador
Ricardo Jorge Gomes Lopes da Rocha, Professor Associado,
Faculdade de Ciências da Universidade do Porto

Coorientador
Miguel João Gonçalves Areias, Professor Auxiliar Convidado,
Faculdade de Ciências da Universidade do Porto

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

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

O Presidente do Júri,

Porto, ______/______/_________

Abstract

Data structures are a fundamental programming tool required by almost any program or algorithm. As such, data structures are a key target to ensure that some concurrency properties are inherited by the programs that use them. These properties range from scalability and progress guarantees to memory overheads and bounds.

In this thesis, we focus on studying a particular lock-free data structure, named *Lock-Free Hash Tries*, and on solving the problem of memory reclamation without loosing the lock-freedom property. To the best of our knowledge, outside garbage collected environments, there is no current implementation of hash maps that is both lock-free and able to reclaim memory. To achieve this goal, we review the current state-of-the-art memory reclamation methods for lock-free data structures and we go over the possibility of making Lock-Free Hash Tries compatible with such memory reclamation methods. We found such a task unfeasible to be done correctly without compromising performance. We thus propose an alternative approach for memory reclamation specific to Lock-Free Hash Tries that explores the characteristics of its structure in order to achieve efficient memory reclamation with low and well-defined memory bounds. This new memory reclamation method makes Lock-Free Hash Tries the first completely lock-free hash map implementation that works in a non garbage collected environment.

Experimental results show that our approach attains better results than the tested state-of-the-art memory reclamation methods and also provides a competitive and scalable thread safe hash map implementation, if compared to lock based implementations.

Keywords: Memory Reclamation, Lock-Freedom, Data Structures, Hash Maps

Resumo

As estruturas de dados são uma ferramenta fundamental de programação que servem de base a qualquer programa ou algoritmo. Por esse motivo, as estruturas de dados são um alvo chave para garantir propriedades de concorrência nos programas que as usam. Essas propriedades variam de garantias de progresso e escalabilidade a garantias de desempenho e limites de memória.

Nesta tese vamos nos focar em estudar a estrutura de dados *Lock-Free Hash Tries* e resolver o problema de reclamação de memória sem perder a propriedade de *lock-freedom.* Tanto quanto sabemos, fora de um ambiente *garbage collected*, não existe nenhuma implementação de *hash maps* que seja *lock-free* e capaz de reclamar memória ao mesmo tempo. Para alcançar esse objetivo, estudamos os métodos de reclamação de memória do estado da arte e exploramos a possibilidade de os aplicar às *Lock-Free Hash Tries.* No entanto, percebemos que essa tarefa não é possível de ser concretizada corretamente sem comprometer o desempenho. Assim, como alternativa, propomos um método de reclamação de memória específico para as *Lock-Free Hash Tries* que explora as características da sua estrutura de modo a não comprometer a eficiência e garantindo limites de memória bem definidos e baixos. Este novo método de reclamação de memória torna as *Lock-Free Hash Tries* na primeira implementação completamente *lock-free* de *hash maps* em ambientes sem *garbage collection.*

Resultados experimentais mostram que o nosso método alcança melhores resultados que os métodos estado-da-arte testados e resulta também numa implementação competitiva e escalável para ambientes concorrentes comparando com as implementações baseadas em *locks.*

Palavras-chave: Reclamação de Memória, *Lock-Freedom*, Estruturas de Dados, *Hash Maps*

Acknowledgements

I want to mainly thank my advisors Ricardo Rocha and Miguel Areias for all their support, ideas and countless hours spent helping me improve myself and this work as whole. They were simply the best mentors I could ever have to guide me on this journey.

I want to acknowledge the funding provided by the CRACS group of INESCTEC as it was of great relevance for the quality of this work.

A word of thanks to all my colleagues from the Masters program that were always there to help and provide a great work environment, specially to Ricardo Leite for all his constructive critique and discussions, which always pushed me on to better solutions.

To all my friends from Physics that accompanied me into countless long nights of study and research, and the very few I had of leisure. A very warm thank you to you all and I wish you the best for your futures. A special thank you to Carolina Afonso for always being there to hear me out and help me relax when I needed.

Finally I would like to say that all the support received from all my family was invaluable and what made this all possible. Thank you all for believing in me more than myself.

Dedicated to my parents and my grandfather

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Nowadays, processor technology has almost stagnated in core speed contrarily to what happened in the recent past. As a consequence, the main improvements seen recently focus mainly on increasing the core count and on improving the core interconnectivity. This paradigm shift increased the importance of programs and algorithms that are able to efficiently use as many processing cores as possible.

Data structures are a basic programming tool that finds its way on almost every computer program or algorithm. As such, data structures are a key building block to take advantage of multiple cores efficiently and to guarantee good progress, throughput and latency properties. In particular, guaranteeing progress in a *non-blocking* manner, can be defined as *wait-freedom*, *lock-freedom* or *obstruction-freedom* [9]. Wait-freedom is optimal from a latency and progress standpoint at the cost of throughput, obstruction-freedom is optimal in throughput at the cost of latency and progress, while lock-freedom brings a good balance between the three. Taking these characteristics in consideration, our focus will be on lock-freedom properties.

In the past years, several lock-free data structures have been developed but, rarely, include a built-in or associated method to reclaim memory. On the other hand, many memory reclamation schemes have been developed for general lock-free data structures [3, 5, 7, 8, 10, 13, 16, 18], but are not always compatible with every lock-free data structure. Sometimes, the memory reclamation schemes are not lock-free themselves, which removes the lock-freedom property from the combination, or are unable to exploit the specific characteristics of the data structures, due to their generalization.

In this thesis, we focus on extending the specific *Lock-Free Hash Tries* (LFHT) data structure, developed by Areias and Rocha [1, 2], to support an efficient memory reclamation in a lock-free manner. The LFHT data structure implements a hash map that settles for a hierarchy of hash tables instead of a monolithic expanding one, granting it good latency and throughput characteristics. To do memory reclamation, we started by exploring the state-of-the-art of the existing methods, namely the *pointer* and *time* based methods of memory reclamation for lock-free data structures. *Reference counting* based methods were not studied in such detail, as they are inefficient and, usually, either allow memory reutilization but not reclamation [14, 18] or require atomic instructions not widely supported by the current processor architectures, namely the double width compare and swap instruction [5].

As a first attempt, we thus tried to implement the existing memory reclamation methods in the LFHT data structure, but we found them all to be incompatible with the LFHT data structure due to its inherent procedure of delegation of removals under certain circumstances. Consequently we tried to make the LFHT data structure compatible with the existing memory reclamation methods, but our design was found to suffer from what is known as the *ABA problem* [4]. This occurred because the design required more space than a single atomic field can provide, for multiple threads to communicate through *compare and swap* (CAS) operations. However, in practice, we were not able to reproduce such ABA problem in our experiments, which somehow shows that the situations leading to the ABA problem are rare. As such, we decided to still implement some of the existing memory reclamation methods on top of our (incorrect) design, for the sake of benchmarking and comparison of results. In any case, the design was proved to be wrong and could at any time during execution lead to an inconsistent state of the data structure.

After this first attempt, it became obvious that adapting the data structure to be compatible with the existing memory reclamation methods was not feasible and, thus, we started exploring alternative designs for a memory reclamation method. This path allowed us to exploit specific characteristics of the data structure in order to attain better performance, guarantee memory bounds and take advantage of the memory reclamation design to enhance the performance of the data structure. Our new design is based on the idea of using a *hazard hash* to represent a path in the hierarchy of the data structure and a *hazard level* to represent a level of the hierarchy. This results in just a small and well-defined portion of the memory being blocked from reclamation by every thread, and a very small amount of updates required to such hazard hashes and levels during operation. The resulting lock-free memory reclamation method, which we named *hazard hash and level (HHL)*, achieves lower synchronization overhead than any of the other studied lock-free memory reclamation methods, while providing very well-defined and flexible memory bounds. However, it still has some minor limitations. It restricts the usage of some of the configuration parameters of the LFHT data structure and requires the underlying memory allocator to efficiently align the memory allocation requests.

With the LFHT data structure using the HHL memory reclamation method, we were able to provide a lock-free hash map implementation with well-defined memory bounds and achieve performance and scalability results surpassing the current lock based implementations of thread safe hash maps. Moreover, by comparing the HHL results with our incorrect design implementation, we also show that the current state-of-the-art time based methods, cause extreme performance degradation on high throughput data structures, such as LFHT. This is caused by their inherent need for global synchronization on every operation, which is not needed in the HHL method, thus making it immune to such extreme degradation.

The remainder of this thesis is organized as follows. First, we start by presenting the possible progress guarantees and by highlighting the relevance of lock-freedom in the context of lock-free data structures. Next, we introduce the state-of-the-art memory reclamation methods for lock-free data structures, by grouping them into basic time and pointer based methods, and methods built on top of the two basic ones. Then, we describe the LFHT data structure in full detail and our developments on accomplishing memory reclamation for it. Next, we present the results obtained with our work, both theoretical and experimental. We conclude this thesis with the final remarks and possible further work directions.

Chapter 2

Background

This chapter starts by presenting the progress guarantees for concurrent algorithms and by explaining the choice of lock-freedom. It then introduces a simple lock-free linked list as our working example and highlights the challenges of achieving memory reclamation while maintaining the lock-freedom property. Next, we present the two categories of the current state-of-the-art methods for memory reclamation on lock-free data structures: *basic methods* and *compound methods*. The basic methods cover the fundamental ideas that can be used for memory reclamation and the compound methods are built on top of the basic ones trying to balance their advantages and disadvantages.

2.1 Progress Guarantees

To implement synchronization in concurrent algorithms, we can use lock-based primitives which give exclusive access to sections of memory at a given time to a specific thread. In general, such primitives are effective and simple to use, but they come with some shortcomings. They do not give any progress guarantees if a thread T suspends, stops or fails after acquiring a lock, since no other thread waiting on the lock hold by thread T can make progress. Algorithms that rely on such methods are thus called *blocking*. The alternative to blocking algorithms are the *non-blocking* ones, which can be divided into three main categories: *wait-free*, *lock-free* and *obstruction-free* [9].

Wait-free algorithms offer the guarantee that any thread makes progress in a finite amount of steps independently of the steps executed by other threads. If the number of steps is dependent on the number of concurrently running threads, it is classified as a *bounded wait-free* algorithm, otherwise it is classified as a *population-oblivious wait-free* algorithm. On the other hand, lock-free algorithms ensure that in a certain amount of steps executed by all threads, at least one makes progress. Finally, obstruction-free algorithms ensure that a thread makes progress in a finite amount of steps if it is the only running thread during that period of time. We discuss lock-freedom in more detail next.

2.2 Lock-Freedom

Lock-freedom is a relevant property to concurrent algorithms as it gives strong progress guarantees while not being as hard to obtain as wait-freedom due to its more relaxed nature.

Lock-free algorithms ensure that in a certain amount of time the system progresses independently of the running state of particular threads, i.e., the system never ends up on deadlock or live-lock even on cases where thread failures occur and, usually, lock-free algorithms show better scalability when increasing the number of available cores.

To achieve lock-freedom, we must avoid any kind of lock in our algorithm, as a thread waiting on a lock does not make progress in any amount of time. This would not be a problem if we could control the scheduling of threads and make sure that at least one thread that is not waiting on a lock is running, but unfortunately that is not the case outside of the kernel space.

The solution is to take advantage of atomic directives that result in hardware specific instructions that operate atomically on memory locations. These directives can be simple atomic loads or stores, or more complex instructions like: (i) atomic exchanges, that change the value of a memory location and a register atomically; (ii) atomic arithmetic or bitwise operations, that take effect atomically on a memory location and return the result; or (iii) a compare and swap directive that takes a memory location, a desired value and a expected value as arguments and atomically changes the value in the memory location to the desired value if the expected value is present (it also returns true if it succeeds or false otherwise). The most relevant atomic directive is *compare and swap* (*CAS*), which is widely supported in modern architectures. Listing 2.1 shows the pseudo-code for the CAS instruction that would be executed atomically.

```
bool compare_and_swap (int *memory, int expected, int desired) {
   if (*memory == expected) {
      *memory = desired;
      return TRUE;
   }
   return FALSE;
}
```

Listing 2.1: CAS instruction pseudo-code.

2.2.1 ABA Problem

The CAS instruction is normally used as a mean to commit a change if no concurrent task has interfered with it in the meantime. Usually, in a lock-free environment, we start by reading some value, then by doing some task based on that value, and finally we try to commit the work done with a CAS instruction using the initial value read as the expected argument for the CAS.

However, the usage of the CAS instruction is not straightforward, as the semantic is not exactly this, because the same value does not mean that it remained unchanged between the first read and the final commit. A scenario can happen where a value A is first read, then concurrent operations update the value from A to B and then from B to A, allowing the CAS operation to succeed with an expected value of A even though the value was changed in the meantime. This is what is called an ABA problem [4].

The prevention of the ABA problem is usually done through the context of the algorithm or the use of other synchronization primitives. Preventing it through the context of the algorithm is ideal as it does not require hardware support for additional synchronization primitives. When such a solution is not possible, the most common way to avoid the problem is through the use of the double-width CAS, in which half of the atomic field is filled with a monotonically incremented value that is updated at every operation. This strategy does not fully prevent the ABA problem as the monotonically incremented value can overflow. However, the value is large enough, so we can usually prove that the overflow can only occur after an amount of time that we do not expect the whole execution to last, which is usually enough for a solution. Another practical problem is that the double-width CAS instruction is not widely available across all common architectures. An ideal solution would be the use of load-link and store-conditional (LL/SC) primitives. In theory, the LL/SC primitives have the exact semantic wanted to prevent the ABA problem, one can read with LL and then the write with SC only succeeds if the value remained unchanged. However, practical implementations are severely limited, as they usually do not allow memory access in between the LL and SC, as is the case for Alpha, ARM and MIPS. Even for other architectures that allow such memory access like PowerPC and RISC-V the SC can sporadically fail due to the granularity of the primitives (a write by any thread to memory location that is close to the location where the LL was previously done can cause the subsequent SC to fail). The last limitation is that there is no defined standard to use these primitives even in relatively low level languages like C or C++. As such, the LL/SC primitives are only used in practice to implement the CAS instruction.

2.2.2 Lock-Free Data Structures

Data structures are an obvious target to achieve lock-freedom. An algorithm can only truly achieve lock-freedom if all of its data structures are lock-free. There are already multiple implementations of lock-free data structures, but most of them are not entirely usable in a lock-free way, as they delegate the task of memory reclamation to an independent garbage collector. This is a problem as it limits the portability to environments where a garbage collector is not available, or is available but is not lock-free. This leads to the loss of the overall lock-freedom property, as one of the pieces it is relying upon does not have this property. The memory reclamation of removed elements on a lock-free data structure is not as simple as in a lock-based one. To ensure lock-freedom, we need to allow concurrent accesses to the elements on the data structure and, as such, we cannot guarantee that an element is not being accessed by other threads at the instant we remove it. Overcoming this limitation requires complex methods to postpone, delegate and determine when the reclamation should occur. These methods should also offer some guarantees on memory usage bounds and performance overhead while keeping the lock-freedom property.

2.2.3 Working Example

To better explain the reclamation methods presented next, we will use as our working example a lock-free linked list implementing a set to store key/contents pairs that works as follows. Each node in the list consists of a key, a reference to the contents stored on the node, a reference to the next node in the chain and an invalidity flag. The invalidity flag is considered to be embedded on the next node pointer (often the least significant bit of the pointer as it does not store any information due to memory alignments). A possible implementation is presented next in C-like pseudocode in Listing 2.2.

During its lifetime, a node can be in one of three states: *valid*, *invalid* or *unreachable*. Figure 2.1 shows a possible configuration illustrating these three states. Node $K1$ is considered *valid* since it has the invalidity flag not set and is reachable from the head of the data structure. Node $K3$ is considered *invalid* since it has the invalidity flag set but is still reachable from the head of the data structure. Finally, $K2$ is considered *unreachable* as it also has the invalidity flag set and is unreachable from the head of the data structure. Note that $K2$ may not have been reclaimed yet as other threads may still have local references to it. When it is determined that there are, no longer, any reference to an unreachable node by any thread then it is safe to reclaim the node's memory. A node in this state will be named as *reclaimable*.

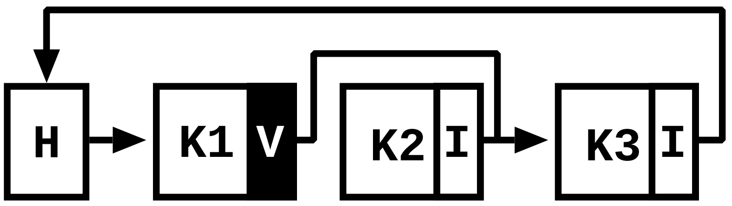

Figure 2.1: Node states.

Figure 2.2 shows the time intervals in which a node can attain these states during execution. A node can become invalid at any time during the remove operation, and unreachable in the same time period, but only after the node became invalid. When no other thread has a reference to the node, it becomes reclaimable. This can happen from when the remove operation finishes, until when the last operation, that started before the node was unreachable, finishes. Note that from this point onwards it is impossible for any other thread to have a reference to the node, making it certain to have become reclaimable at this point or before.

The list begins and ends with a special node H. It has a reference to the first node on the list, and the last node on the list references back to H. To manipulate a list three operations are possible: search for a node, insertion of a node and removal of a node.

A search starts in the reference of H and traverses the list until it either finds the node in a valid state or H again, meaning that the node does not exist. The search procedure is shown in

```
// node defenition
struct node {
   int key;
   void *contents;
   node_state state;                              // chain node and invalidity flag
}

// auxiliary function
#define FLAG_MASK 1

int get_invalidity_flag(node_state state) {
   return state & FLAG_MASK;
}

// interface functions
node *get_chain_node(node_state state) {
   return (node *) (state & ~FLAG_MASK);

int is_valid(node_state state) {
   return !get_invalidity_flag(state);
}

int is_invalid(node_state state) {
   return get_invalidity_flag(state);
}

node_state gen_state(node *ptr, enum flag) {
   if (flag == VALID)
      return (node_state) ptr;
   else
      return ((node_state) ptr) | FLAG_MASK;
}

node_state change_state(node_state state, enum flag) {
   return gen_state(get_chain_node(state), flag);
}
```

Listing 2.2: Node definition and interface functions for our working example.

Listing 2.3.

An insertion starts by traversing the list until it finds a node with the same key (in this situation it fails) or the last valid node before H where we will be inserting our new node N right after. To do that, we keep a reference to the *state* field L of the last valid node traversed and the value LV stored in L. Next, we assign H to the state field of node N. Then, we perform a CAS on L, trying to update LV to the address of node N. If the CAS succeeds, then node N is inserted. Otherwise, the operation has to be restarted, as the list was changed in the meantime by another thread and we need to recheck its state. The insert procedure is shown in Listing 2.4.

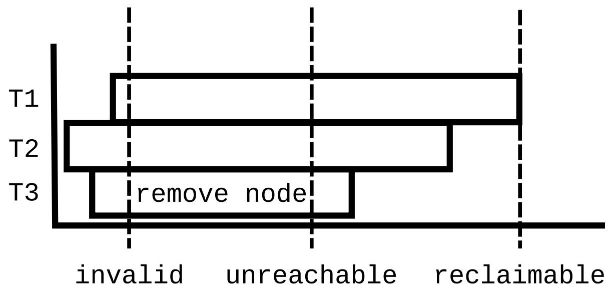

Figure 2.2: Representation of possible points in time where a node can become invalid, unreachable and reclaimable.

```
node *search(int key, node *H) {
   node *iter = get_chain_node(H->state);                          // traverse the list
   while (iter != H && !(is_valid(iter->state) && iter->key == key))
      iter = get_chain_node(iter->state);
   if (iter == H)                                                  // node not found
      return NULL;
   else                                                            // node found
      return iter;
}
```

Listing 2.3: Procedure to find a node based on its key.

The node N, the state field L and the value LV are represented in Listing 2.4, respectively, by the variables *new_node*, *valid_addr* and *valid_val*.

Finally, Listings 2.5 and 2.6 show the remove procedure. A removal starts by calling the *search()* procedure to traverse the list until it finds the node N which holds the key. If it is not found, nothing needs to be done, thus the procedure simply returns false. Otherwise, we mark N as invalid, by setting the invalidity flag, which stops other nodes from being concurrently inserted immediately after it. Afterwards, we try to make N unreachable. To make N unreachable, we start by finding the fist valid node A after N, then we need the reference to the state field L of the previous valid node before N in the chain, and the value LV stored in L. With that, we perform a CAS, similarly to the insertion case, but now with A as the desired value. If the CAS fails, we restart the process of making N unreachable. Otherwise, we successfully made the node unreachable, and we only need to reclaim its memory when it becomes reclaimable to complete the removal. The node A is represented in the Listing 2.6 by *valid_after*.

```
bool insert(node *H, node *new_node) {
   node_state *valid_addr = &(H->state);                        // find insertion point
   node_state valid_val = *valid_addr;
   node *iter = get_chain_node(valid_val);
   while (iter != H) {
      node_state tmp = iter->state;
      if (is_valid(tmp)) {
         if (iter->key == new_node->key)
            return FALSE;
         valid_addr = &(iter->state);
         valid_val = tmp;
      }
      iter = get_chain_node(tmp);
   }
   new_node->state = gen_state(H, VALID);
   if (CAS(valid_addr, valid_val, gen_state(new_node, VALID)))     // try to insert
      return TRUE;
   else                                                          // retry
      return insert(H, new_node);
}
```

Listing 2.4: Procedure to insert a node if a node with the same key does not exist already.

```
bool remove(node *H, int rm_key) {
   node *rm_node = search(H, rm_key);                                 // find node
   if (rm_node == NULL)                                          // node not found
      return FALSE;
   do {                                                        // make node invalid
      node_state tmp = rm_node->state;
      if (is_invalid(val))
         return FALSE;
   } while(!CAS(&(rm_node->state), tmp, change_state(tmp, INVALID)));
   make_unreachable(H, rm_node);                         // make node unreachable
   return TRUE;
}
```

Listing 2.5: Procedure to remove a node based on its key.

```
void make_unreachable(node *H, node *rm_node) {
   node *iter = get_chain_node(rm_node->state);            // find valid after rm_node
   while (iter != H) {
      node_state tmp = iter->state;
      if (is_valid(tmp))
         break;
      iter = get_chain_node(tmp);
   }
   node_state valid_after = gen_state(iter, VALID);
   node_state *valid_addr = &(H->state);                   // find valid before rm_node
   node_state valid_val = *valid_addr;
   iter = get_chain_node(valid_val);
   while (iter != H && iter != rm_node) {
      node_state tmp = iter->state;
      if (is_valid(tmp)) {
         valid_addr = &(iter->state);
         valid_val = tmp;
      }
      iter = get_chain_node(tmp);
   }
   if (iter == H)                                          // node not found (already unreachable)
      return;
   if (CAS(valid_addr, valid_val, valid_after))            // try to make unreachable
      return;
   else                                                    // retry
      return make_unreachable(H, rm_node);
}
```

Listing 2.6: Procedure to make an invalid node unreachable.

2.3 Memory Reclamation and Lock-Freedom

As seen before, to guarantee the lock-freedom of the data structure as a whole, memory reclamation also needs to be done in a lock-free manner. However, that is not enough to guarantee lock-freedom when combining the two components, as the synchronization between both requires that progress occurs in both components. For example, if the memory reclamation method is not able to progress (i.e., reclaim nodes) at the same rate as nodes are removed from the data structure, we may end up with unbounded memory consumption. This will exhaust the memory of the system leading to a logic lock on future memory allocations, as it is unrealistic to assume that we have infinite memory in a system. Another problem arises from the fact that reclamation of memory requires state information from all running threads, which unavoidably affects the potential scalability lock-freedom usually offers if nothing is done to minimize it.

In what follows, we describe the current state-of-the-art methods for memory reclamation on lock-free data structures. We present two basic methods, namely *grace periods* and *hazard pointers*, and two compound methods, namely *drop the anchor* and *hazard eras*, which are built on top of the basic ones.

2.3.1 Grace Period

Concurrent access to shared resources can lead to unexpected or inconsistent behavior. A *critical section* is a portion of code that has access to shared resources and therefore needs some sort of guarantees on protection from the other threads interacting with the same shared resources. In such scenario, a *quiescent state* is a moment in time where a thread is outside any critical section and a *grace period* is a period of time in which all threads have finished at least one critical section, i.e., have been in at least one quiescent state. Figure 2.3 shows the representation of three quiescent states leading to a grace period after thread $T3$ made a node unreachable.

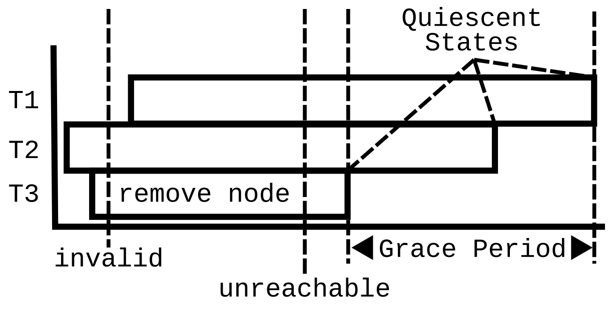

Figure 2.3: Representation of the quiescent states leading to a grace period.

When a node N is made unreachable with the goal of removing it, we can be certain that after a grace period has elapsed, no thread still references N, thus allowing us to safely reclaim its memory. Based on this method, various schemes for determining the relative temporal orders between events can be used to reclaim memory with very little synchronization and overhead [7, 8].

One way to establish such temporal order can be done by using the Lamport clocks method [11] or by defining global epochs [7]. The Lamport clocks method works as follows. Every thread has an internal clock that is readable by all threads but only writable by itself. To mark an event in time, a thread reads the clocks of all the threads and updates its clock with the maximum value read plus one. Since these clock values are relative to each other, a particular value tells us that all events marked with lower clock values happened before it. We can use this method to mark the time on a node, when we make it unreachable, and update the thread internal clocks on quiescent states. This information is enough to easily determine if a grace period has elapsed between the event of making a node unreachable and the moment we try to reclaim its memory, simply by checking if every thread has reported a quiescent state in this period. The declaration of any of these events however has a cost proportional to the number of running threads, and even thought we can freely control the frequency at which quiescent states are declared and at which we try to reclaim memory, this can only be done at the cost of more memory on the reclamation queue.

Another way to establish the temporal order is by defining global epochs where a global clock, used by every thread to update itself on quiescent states, is incremented atomically at events that need to be distinguished (e.g., making a node unreachable). This makes the cost of both events constant but makes them dependent on global state. An epoch change requires an atomic read-modify-write operation which can degrade performance. As with the Lamport clocks method, we can freely control both the frequency at which we declare quiescent states and at which we try to do reclamation at the cost of more memory in the reclamation queue.

Implementation-wise, a quiescent state can be declared explicitly or encapsulated in directives used to define the beginning and the ending of critical sections. Such directives would ease the usage of these methods for the programmer and would work by defining extended quiescent states when outside of all critical sections. This however adds some overhead to the procedure and offers less control over the frequency at which the quiescent states are declared.

Although these methods do not necessarily hurt the lock-freedom property of a concurrent data structure, they do so by delegating or deferring the reclamation and, as such, cannot ensure the lock-freedom of the system as whole. The event of waiting for the declaration of a quiescent state by a single thread, renders all other threads to be unable to reclaim memory of all future nodes made unreachable for removal. Per se, this does not block normal operation on the data structure, but does not allow any progress in memory reclamation, leading to unbounded memory consumption and a possible logic lock in memory allocation.

2.3.2 Hazard-Pointers

Hazard pointers [13] are shared variables that hold pointers to the elements concurrently in use by a thread. They serve the purpose of informing the other threads of what elements of a data structure cannot be reclaimed because they are being accessed by the thread that has references to them in its hazard pointers.

For the linked list example, we need two hazard pointers per thread, as we need to protect the current and the previous node a thread is traversing. Two hazard pointers are needed because, when a thread T reaches an invalid node I, it cannot be certain to find a valid node in the next reference field of I, as that field has become immutable from the point in time I was marked as invalid. Note that the node referenced in the next field of I is not yet protected by the hazard pointers of T and so it could have been already removed and reclaimed, rendering the reference unsafe to follow. The solution is to make node I unreachable, which can only be done by updating the previous node P, that is protected by the second hazard pointer. For that, we do a CAS trying to update the next reference of P to the node referenced in the next field of I. A CAS failure means that either I was already removed or P became invalid meanwhile and so we have insufficient information about the state of the nodes we are trying to follow. The only option is thus to restart the traversal from a known safe place (e.g., the head of the list). If the CAS succeeds, we can continue the traversal as usual, as we now know that the next node is referenced from a valid one and thus it cannot be already unreachable or reclaimed (note that this assumption is only possible because we know that every thread traverses the data structure in this way).

To remove a node, the removal procedure begins by marking the node invalid with a CAS, then makes it unreachable, in the same way as for the traversal, and ends by adding it to the local reclamation queue. The reclamation procedure starts when the amount of nodes in the reclamation queue reaches a certain threshold. That threshold can be tuned in order to exchange memory usage with performance. In particular, if we keep the size of the queue above $H + \Omega(H)$[1], being H the total number of hazard pointers, we can guarantee a constant expected amortized time complexity for the reclamation of a node. To reclaim nodes, we start by reading all hazard pointers of all threads and by comparing them against the nodes on the reclamation queue. A node not referenced by any hazard pointer can be safely reclaimed as no thread references it.

2.3.3 Drop the Anchor

The drop the anchor method [3] can be described as a grace period based method extended with a recovery mechanism based on hazard pointers that, despite being very expensive is, expected to run very rarely.

It works like the grace period method with a few extensions: (i) it adds a field to the nodes to mark the removal time; (ii) it uses a bit on the pointers to mark the ones that are frozen; (iii) and integrates a pointer and two additional bits into the per thread clock, to store an *anchor* and a *thread state*. The two additional bits serve the purpose of distinguishing between the states of *idle*, *running*, *stuck* and *recovered*, and the anchor works like a hazard pointer that is updated every time a predefined interval of nodes is traversed. With the additional bits, a thread can identify when another thread is idle, in order to not take its clock into account during

[1]This ensures that at least $\Omega(H)$ nodes are reclaimed at every time the reclamation procedure is called which has a base time complexity of $O(H)$ for reading all the hazard pointers.

reclamation, and can mark a thread as stuck, when it identifies that the thread is not making progress in a certain amount of time, in order to start a recovery procedure.

The goal of the recovery procedure is to recover the nodes blocked by the stuck thread, i.e., the nodes between the last anchor registered by the stuck thread and the point where it would update the anchor again. The recovery procedure replaces the existing nodes with new ones and marks the replaced nodes as frozen. This allows the remaining threads to be able to ignore the stuck threads and thus continue to reclaim memory normally. The replaced nodes remain frozen until the stuck thread recovers and takes care of them.

The recovery procedure starts at the anchor referenced by the stuck thread and marks each node as frozen and copies the valid ones to a new list until the next anchor point. Then, the node before the stuck thread anchor is reconnected to the new list and finally in a single atomic operation the thread clock is updated and the thread is marked as recovered. All threads, including the one marked as stuck, are able to assist in this procedure. This is essential to maintain the lock-freedom of the algorithm as a thread failure on this procedure would cause a lock otherwise.

Regarding the reclamation method, apart from needing to guarantee that the removal time of a node is inferior to the clock of all running threads, similarly to grace periods, we also need to ensure that it is greater than all recovered threads. This prevents nodes that are visible, to a thread which is either stuck or recovered, but were not frozen because they became unreachable before the recovery procedure had a chance to freeze them. The reclamation of this kind of frozen nodes can be left for the thread that caused their freeze, since when the thread recovers, it has all the information it needs to reclaim the nodes, and the exclusive access needed to do it.

Performance-wise, this method can approach the grace period method in normal operation and still has a bound on the memory usage which is proportional to the number of nodes between anchors in case of a thread failure. On the other hand, it may need double width CAS instruction support and the cost of the recovery procedure can lead to high latency when the recovery procedure is used [3].

2.3.4 Hazard Eras

The hazard Eras method [16] works similarly to grace periods, however it uses clocks not only for the removal time of every node, but also for the insert time, which allows to continue reclaiming memory on the event of a thread delay or failure. It uses a global clock that is incremented atomically at every removal and, similarly to the way hazard pointers are updated in the hazard pointers method, when a thread reads a new reference (within the data structure) it updates its local clock to the global clock.

The insert time of the nodes allows them to not be blocked from reclamation by threads that stalled or failed before such nodes were even inserted. Similarly to hazard pointers, we cannot follow references of invalid nodes while traversing the data structure as we have no guarantee that we will not find already reclaimed nodes if we follow such references.

Using this method, a delayed thread can block from reclamation the number of nodes the data structure has at the moment of the last update to its local clock. The amount of memory is thus bounded, but can still be very high for the performance and memory overhead it implies.

2.3.5 Comparison

All these methods have advantages and disadvantages, but it boils down to three main aspects: performance, memory usage and complexity. The grace periods method has optimal performance, as it is very simple, but the memory usage can explode rendering it to be unusable. The hazard pointers method has optimal bounds in memory, but has an extra cost of performance and it is slightly more complex to implement. Drop the anchor balances performance and memory very well, but pays the price in complexity. Hazard Eras end up balancing everything, but not excelling in any particular one. Table 2.1 shows in more detail the memory bounds and synchronization costs of each method.

Method	**Memory Bound**	**Synch. Operations per Node**
Grace Periods	unbounded	none
Hazard Pointers	$T^2 \times H$	2 loads + 1 store
Drop the Anchor	$T^2 \times A$	amortized
Hazard Eras	$T^2 \times N$	2 loads

Table 2.1: Comparison of the memory reclamation methods in terms of memory bounds and synchronization operations per node (T represents the number of threads, H the number of hazard pointers per thread, A the anchor interval and N the number of nodes that may be valid in the data structure at any given moment).

Chapter 3

Lock-Free Hash Tries

This chapter introduces the LFHT data structure in more detail, presents its inner structure and the different operations it supports, and describes the key algorithms used to support such operations.

3.1 Overview

Hash tries are a tree-based data structure with nearly ideal characteristics for a hash map implementation. We will base our work on the lock-free implementation of hash tries proposed by Areias and Rocha [1, 2]. The implementation has two kinds of nodes: *hash nodes* and *leaf nodes*. The leaf nodes store key/value pairs and the hash nodes implement a hierarchy of hash levels of fixed size 2^w. To map a key/value pair *(k,v)* into this hierarchy, we compute the hash value h for k and then use chunks of w bits from h to index the appropriate hash level, i.e., for each hash node H_i, we use the $w \times i$ least significant bits of h to index the entry in the appropriate bucket array of H_i. To deal with collisions, we let leaf nodes form a linked list in the respective bucket entry until a threshold is met and, in such case, we perform an expansion operation where the bucket entry is updated to a new hash node H_{i+1} containing the nodes in the linked list. Figure 3.1 shows a small example that illustrates how the insertion of nodes is done in a hash level.

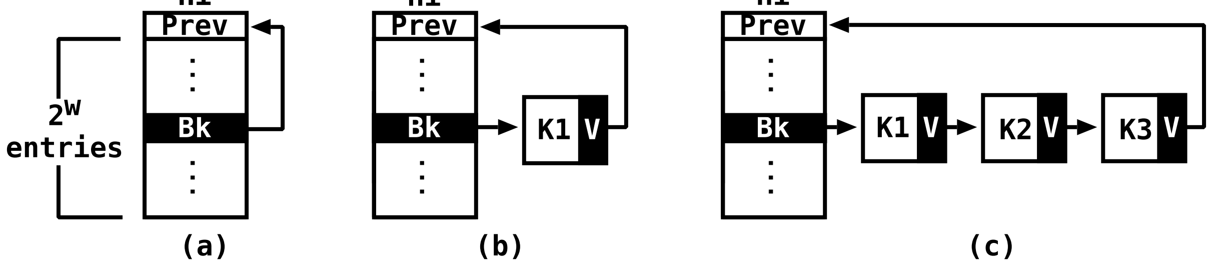

Figure 3.1: Insertion of nodes in a hash level.

Figure 3.1(a) shows the initial configuration for a hash level. Each hash level is formed by a hash node H_i, which includes a bucket array of 2^w entries and by a backward reference *prev* to the previous hash level or *NULL* in case of the first hash node, and by the corresponding chain of nodes per bucket entry. Initially, all bucket entries are empty. In Fig.3.1, B_k represents a particular bucket entry of H_i. A bucket entry stores either a reference to a hash node (initially the current hash node) or a reference to a separate chain of leaf nodes, corresponding to the hash collisions for that entry. Figure 3.1(b) shows the configuration after the insertion of node $K1$ on B_K and Fig. 3.1(c) shows the configuration after the insertion of nodes $K2$ and $K3$. A leaf node holds both a reference to a next-on-chain node and a flag with the condition of the node, which can be valid (V) or invalid (I). Note that the pair flag, reference to the next-on-chain becomes immutable when the flag turns invalid. Immutable fields are represented with a white background and mutable fields with a black background.

When the number of valid nodes in a chain reaches a given threshold, the next insertion causes the corresponding bucket entry to be expanded to a new hash level, so that the nodes in the chain are remapped in the new level, i.e., instead of growing a single monolithic hash table, the hash trie settles for a hierarchy of small hash tables of fixed size 2^w. Figure 3.2 shows the expansion of nodes from a full chain to the next hash level.

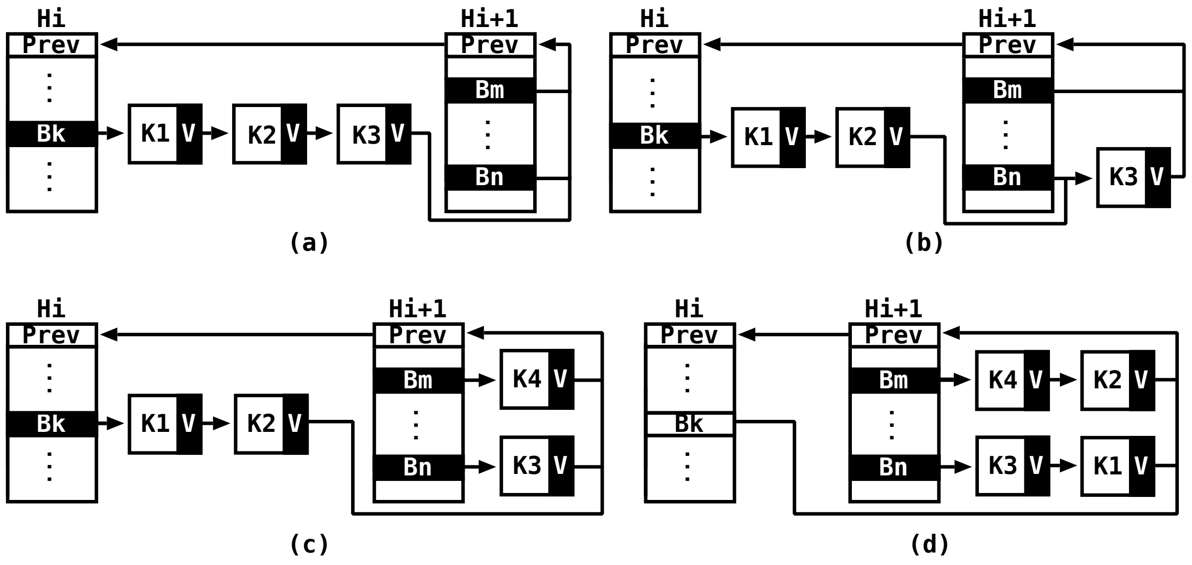

Figure 3.2: Expansion of nodes in a hash level.

The expansion operation starts by inserting a new hash node H_{i+1} at the end of the chain with all its bucket entries referencing H_{i+1} and the *prev* field referencing H_i (as shown in Fig. 3.2(a)). From this point on, new insertions will be done on the new level H_{i+1} and the chain of leaf nodes on H_i will be moved, one at a time, to the new level H_{i+1}. Figures 3.2(b) and 3.2(c) show how node $K3$ is first mapped in H_{i+1} (bucket B_n) and then moved from H_i (bucket B_k). It also shows a new node $K4$ being inserted simultaneously by another thread. When the last node is expanded the bucket entry in H_i references H_{i+1} and becomes immutable as we can see in Fig. 3.2(d).

Next, we describe how the removal of nodes is done. Figure 3.3 shows an example illustrating how a node is removed from a chain. The remove operation can be divided in two steps: (i) the invalidation of the node (as shown in Fig. 3.3(a)) and (ii) making the node unreachable (as shown in Fig. 3.3(b)). The invalidation step starts by finding the node N we want to remove and by changing its flag from valid (V) to invalid (I). If the value of the flag is already invalid, it means that another thread is also removing the node and, in such case, nothing else needs to be done. Next, to make the node unreachable, first we need to find the next valid node A on the chain (note that it can be the hash node corresponding to the level N is at). And then, we need to continue traversing the chain until we find a hash node H (if we have not yet).

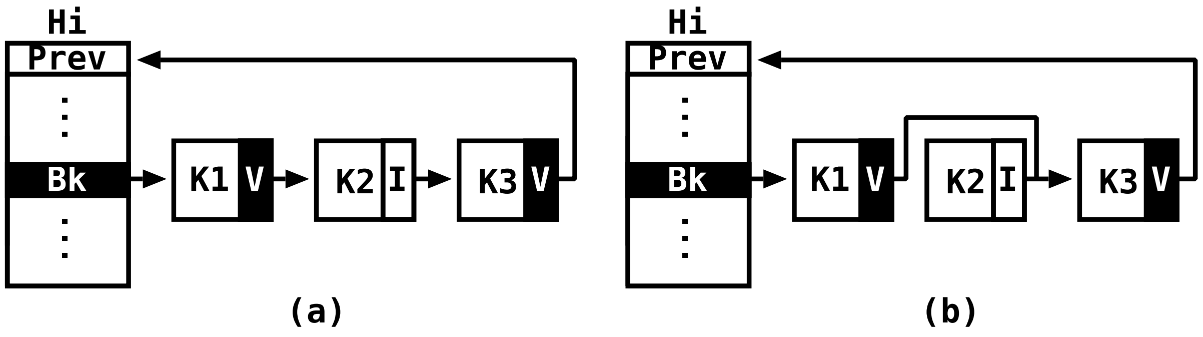

Figure 3.3: Removal of nodes in a hash level.

If H is the same hash node we have started from, we traverse again the chain until we find the last valid node B before N (or we consider the bucket entry if no valid node exists). If, while searching for B we do not find node N, it means that the node has already been made unreachable and our job is done. Otherwise, we just need to change the reference of B to A. This is the situation shown in Figure 3.3(b) where node $K1$ is made to point to $K3$.

If H is not the same hash node we have started from, this means that a concurrent expansion is happening simultaneously and we restart the process in the next level (note that node N could either have been expanded before we have invalidated it or is currently in the process of being expanded). In the case node N has been expanded before we made it invalid, we will be able to make it unreachable in the next level. Otherwise, if it is in the process of being expanded, we do not need to make the node unreachable, as the expanding thread will not expand it or will make it unreachable if it only sees it as invalid after completing its expansion (in this situation, the thread doing the expansion becomes responsible by making the node unreachable).

3.2 Algorithms

This subsection presents in more detail the algorithms that support our implementation of LFHT.

We start by introducing the *find_node()* procedure, shown in Listing 3.2, that is responsible for traversing the data structure looking for a particular node, while keeping auxiliary information about the traversal that will be useful for the other operations (the *insertion_info* structure as

```
struct insertion_info {
   node_state *valid_addr;
   node_state valid_val;
   int count;
}
```

Listing 3.1: Structure used to keep auxiliary insertion information about the traversal.

```
node *find_node(size_t hash_value, node **hash_node, insertion_info *args) {
   int pos = get_bucket(hash_value, *hash_node);
   args->valid_addr = &((*hash_node)->hash.array[pos]);
   args->valid_val = *(args->valid_addr);
   node *iter = get_chain_node(args->valid_val);
   args->count = 0;
   while (iter != *hash_node) {                              // traverse the chain
      if (iter->type == HASH_NODE) {                        // new hash node found
         while (iter->hash.prev != *hash_node)
            iter = iter->hash.prev;
         *hash_node = iter;                       // update hash_node to next level
         return find_node(hash_value, hash_node, args);
      }
      node_state tmp = iter->leaf.state;
      if (is_valid(tmp)) {                                    // found a valid node
         if (iter->leaf.hash_value == hash_value)
            return iter;
         args->valid_addr = &(iter->leaf.state);
         args->valid_val = tmp;
         (args->count)++;
      }
      iter = get_chain_node(tmp);
   }
   return NULL;
}
```

Listing 3.2: Procedure to find a node or where it would be inserted based on its hash value.

shown in Listing 3.1). The *find_node()* procedure receives three parameters, the first parameter is an input argument, the second parameter is an input and output argument, and the third parameter (the *insertion_info* structure) is an output argument. In more detail:

- *hash_value*: hash value of the node we are trying to find;
- *hash_node*: initially the pointer to the hash node we want to start the search at (for the entry call, it is the pointer to the root hash node), and at the end it stores the deepest hash node traversed;
- *args->valid_addr*: pointer to the atomic field of the last valid node found (initially the bucket entry and then the node's state fields);

- *args->valid_val*: value stored in the address pointed by *valid_addr*;
- *args->count*: counter used to keep the number of valid nodes found on the current chain.

The *find_node()* procedure starts by getting the bucket entry for the given hash value, which is stored in the *valid_addr* variable. The *valid_val* and *iter* variables are then updated accordingly to the first node on the chain, and the counter is also initialized before starting to traverse the chain (lines 2–6). The chain is then traversed until we find either: (i) the same hash node, case in which the node was not found and we return *NULL* (line 24); (ii) a different hash node, case in which we call the procedure recursively (line 12); or (iii) the node we are looking for, case in which we return it (line 17).

If a different hash node is found, we need to make sure that we do the recursion in the hash level following the current one. Note that during an expansion operation, we can follow a node leading to deeper hash levels. We fix that by following the *prev* fields in the hash nodes until reaching the hash node just one level after the current one (lines 9–10).

During the traversal, for each node, we read the atomic field of the node and check the flag to see if it is valid or invalid (line 15). If valid, we then check if it is the node we are looking for (line 16) and, if so, we return the node (line 17). Otherwise, this is not the node we are looking, but we update the *valid_addr*, *valid_val* and *count* variables accordingly (since this is the new last valid node visited) and we move to the next node in the chain. If the node is invalid, we just move to the next node in the chain without updating the three variables above (line 22).

With the *find_node()* procedure, the algorithm for searching for a node becomes trivial as we can see in Listing 3.3. If the node is found, we simply return its contents. Otherwise, we return *NULL*.

```
void *search_node(size_t hash_value, node *hash_node) {
   insertion_info args;
   node *found = find_node(hash_value, &hash_node, &args);
   if (found)
      return found->leaf.contents;
   else
      return NULL;
}
```

Listing 3.3: Procedure to find a node based on its hash value that returns its contents.

The procedure for inserting nodes is the *search_insert()* procedure, which inserts a node if it does not exist already in the data structure. The pseudo-code for the *search_insert()* procedure is shown in Listing 3.4.

The *search_insert()* procedure begins by calling *find_node()* to either find the node if it already exists or the potential insertion point for it[1]. If the node already exists, it just returns

[1]Note that the *hash_node* variable is updated to the deepest hash node traversed during the *find_node()*

```
 1 void *search_insert(size_t hash_value, node *hash_node, void *contents) {
 2    insertion_info args;
 3    node *found = find_node(hash_value, &hash_node, &args);
 4    if (found)                                              // node already exists
 5       return found->leaf.contents;
 6    if (args.count >= MAX_NODES) {                                  // chain is full
 7       node *new_hash = create_hash_node(hash_node->hash.level+1, hash_node);
 8       if (CAS(args.valid_addr, args.valid_val, new_hash)) {        // try to expand
 9          int pos = get_bucket(hash_value, hash_node);
10          adjust_chain_nodes(hash_node->hash.array[pos], new_hash);
11          hash_node->hash.array[pos] = gen_state(new_hash, VALID);
12          return search_insert(hash_value, new_hash, contents);
13       } else                                                    // expansion failed
14          free(new_hash);
15    } else {                                                        // try to insert
16       node *new_node = create_leaf_node(hash_value, contents, hash_node);
17       if (CAS(args.valid_addr, args.valid_val, gen_state(new_node, VALID)))
18          return new_node->leaf.contents;
19       else                                                      // insertion failed
20          free(new_node);
21    }
22    return search_insert(hash_value, hash_node, contents);                  // retry
23 }
```

Listing 3.4: Procedure to insert a node with a hash value/contents pair if a node with the same hash value does not exist.

its contents (line 5). Otherwise, it checks the *count* variable to verify if the chain of nodes is full. If it is full (lines 7–14), it tries to insert a new hash node after the last valid field reference, using for that a CAS operation on *valid_addr*. If successful, it calls the *adjust_chain_nodes()* procedure to expand the nodes from the current chain to the new hash node and then restarts the insertion procedure on the new hash node. If the chain is not full (lines 16–20), it tries to insert a new leaf node after the last valid field reference, using for that a CAS operation on *valid_addr*. If any of these two CAS operations fails, the corresponding newly created node is made free and the process is repeated from the beginning (line 22).

The *adjust_chain_nodes()* procedure is described in Listing 3.5 in more detail. The *adjust_chain_nodes()* procedure is very simple, it goes through the chain of nodes and calls *adjust_node()* for the valid nodes on the chain in order to expand them to the new level, one by one in reverse order.

The relevant work is thus done by the *adjust_node()* procedure shown in Listing 3.6. It begins by finding the expansion point (last valid node in the chain) on the new hash level by traversing the chain until reaching a hash node (lines 7–15). If it does not reach the given hash node, that means that another expansion is taking place simultaneously and, in that case, the process is

procedure.

```
void adjust_chain_nodes(node *leaf_node, node *hash_node) {
   if (leaf_node == hash_node)                          // no more nodes to expand
      return;
   node_state tmp = leaf_node->leaf.state;
   node *next = get_chain_node(tmp);
   adjust_chain_nodes(next, hash_node);                 // adjust the next node first
   if (is_valid(tmp))                                   // adjust the node if it is valid
      adjust_node(leaf_node, hash_node);
   return;
}
```

Listing 3.5: Procedure to ajust a chain of nodes into the next hash level.

restarted in the next level (lines 36–38). Otherwise, we verify the number of valid nodes on the chain and, if it is full (line 17), we try to insert a new hash node and initialize a second expansion operation before continuing the current one (lines 18–24). If there is available space, we make the state field of the node being expanded reference the given hash node (*force_cas()* operation at line 26) and then we try to insert the node in the chain (lines 28–32). The *force_cas()* operation only fails if the node becomes invalid meanwhile, case in which we do not need to expand it. After a successful expansion, we need to verify if the node is still valid, since a concurrent thread could have tried to remove it and would not be able to make it unreachable with the ongoing expansion. In this case, we need to make it unreachable by calling the *make_unreachable()* procedure before continuing. The pseudo-code for the *make_unreachable()* procedure is presented in Listing 3.7.

The *make_unreachable()* procedure starts by finding the first valid node (*valid_after*) after the node we are making unreachable (lines 3–9) and then by finding the hash node that ends the chain (lines 10–11). Then, if that hash node is the given hash node (*hash_node*), we know that no expansions are being done concurrently, and we start a second traversal trying to find the last valid node (*valid_addr*) before the node we are making unreachable (lines 13–24) and we also keep the value of its state field (*valid_val*). If in that second traversal, we do not find our node (lines 30–31), it means that it has been already made unreachable by another thread and we simply return. Otherwise, we try to connect *valid_addr* to *valid_after* through a CAS operation (line 26). If the CAS operation succeeds, we have successfully made the node unreachable. If not, we restart the whole process (line 29).

If a different hash node is found (lines 32–34), we check its level and if it is greater than the current level, we restart the process in that hash node. Otherwise, it means that we have already tried the *make_unreachable()* procedure in a level after the one in which the node was already made unreachable and we can safely terminate.

Finally, to remove a node from the data structure, we use the *search_remove()* procedure as shown in Listing 3.8. The *search_remove()* procedure begins by calling *find_node()* to find if the node with the given hash value exists. If not, nothing needs to be done. Otherwise, it tries to make the node invalid by calling *mark_invalid()* and, if it succeeds, it calls the

```
void adjust_node(node *leaf_node, node *hash_node) {
   int pos = get_bucket(leaf_node->leaf.hash_value, hash_node);
   node_state *valid_addr = &(hash_node->hash.array[pos]);
   node_state valid_val = *valid_addr;
   node *iter = get_chain_node(valid_val);
   int count = 0;
   while (iter->type == LEAF_NODE) {                          // find the expansion point
      node_state tmp = iter->leaf.state;
      if (is_valid(tmp)) {
         valid_addr = &(iter->leaf.state);
         valid_val = tmp;
         count++;
      }
      iter = get_chain_node(tmp);
   }
   if (iter == hash_node) {                                   // expansion point found
      if (count >= MAX_NODES) {                                        // chain is full
         node *new_hash = create_hash_node(hash_node->hash.level+1, hash_node);
         if (CAS(valid_addr, valid_val, new_hash)) {              // try new expansion
            adjust_chain_nodes(hash_node->hash.array[pos], new_hash);
            hash_node->hash.array[pos] = gen_state(new_hash, VALID);
            return adjust_node(leaf_node, new_hash);
         } else                                                // new expansion failed
            free(new_hash);
      } else {                                                       // try to expand
         if (!force_cas(leaf_node, hash_node))
            return;
         if (CAS(valid_addr, valid_val, gen_state(leaf_node, VALID))) {
            if (is_invalid(leaf_node->leaf.state))
               make_unreachable(leaf_node, hash_node);
            return;
         }
      }
      return adjust_node(leaf_node, hash_node);                                // retry
   }
   while (iter->hash.prev != hash_node)                       // retry on next level
      iter = iter->hash.prev;
   return adjust_node(leaf_node, iter);
}
```

Listing 3.6: Procedure to adjust a single node into the next hash level.

make_unreachable() procedure in the continuation. The *mark_invalid()* repeats a CAS operation until either it is able to succeed by making the node invalid or, if the node becomes invalid by the action of other thread, it fails. Thus, in case of failure, that means that another thread is also doing the removal and we can safely return. Otherwise, *make_unreachable()* turns the node unreachable as explained above.

```
void make_unreachable(node *leaf_node, node *hash_node) {
   node *iter = get_chain_node(leaf_node->leaf.state);
   while (iter->type == LEAF_NODE) {                              // find valid_after
      node_state tmp = iter->leaf.state;
      if (is_valid(tmp))
         break;
      iter = get_chain_node(tmp);
   }
   node_state valid_after = gen_state(iter, VALID);
   while (iter->type != HASH_NODE)                                // find end of chain
      iter = get_chain_node(iter->leaf.state);
   if (iter == hash_node) {                                       // chain ends in same level
      int pos = get_bucket(leaf_node->leaf.hash_value, hash_node);
      node_state *valid_addr = &(hash_node->hash.array[pos]),
      node_state valid_val = *valid_addr;
      iter = get_chain_node(valid_val);
      while (iter->type == LEAF_NODE && iter != leaf_node) {      // find node again
         node_state tmp = iter->leaf.state;
         if (is_valid(tmp)) {
            valid_addr = &(iter->leaf.state);
            valid_val = tmp;
         }
         iter = get_chain_node(tmp);
      }
      if (iter == leaf_node) {                                    // try to make unreachable
         if (CAS(valid_addr, valid_val, valid_after))
            return;
         else                                                     // failed, so retry
            return make_unreachable(leaf_node, hash_node);
      } else if (iter == hash_node)                  // node not found (already unreachable)
         return;
   } else if (iter->hash.level < hash_node->hash.level)
      return;                                 // node was never in this level (already unreachable)
   make_unreachable(leaf_node, iter);               // expansion in course, try on next level
}
```

Listing 3.7: Procedure to make an invalid node unreachable if it is not being concurrently adjusted to the next level.

```
bool search_remove(size_t hash_value, node *hash_node) {
   insertion_info args;
   node *found = find_node(hash_value, &hash_node, &args);
   if (found && mark_invalid(found)) {
      make_unreachable(found, hash_node);
      return TRUE;
   }
   return FALSE;
}
```

Listing 3.8: Procedure to remove a node based on its hash value.

Chapter 4

Extending Lock-Free Hash Tries with Memory Reclamation

In this chapter, we start by describing our initial attempt of applying the current state-of-the-art memory reclamation methods to the LFHT data structure. Then, we discuss why the current state-of-the-art methods are not applicable to LFHT. Next, we present the new memory reclamation method specific to LFHT. This new method is closely integrated with the previous implementation and exploits the LFHT structure to achieve optimal memory bounds, low synchronization overhead and even enhance the data structure itself.

4.1 State-of-the-Art Memory Reclamation Methods

All the current state-of-the-art memory reclamation methods rely on the remove operation in order to ensure that the element being removed is left in an unreachable state when they terminate. This guarantees that all elements added to a thread local memory reclamation queue have been deemed as unreachable and thus no other thread will acquire a reference to them.

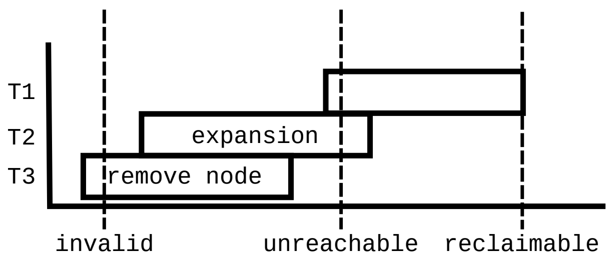

Figure 4.1: Representation of how the node states can change in LFHT.

In the original design of the LFHT data structure, a node is not guaranteed to be unreachable at the end of the remove operation. The reason for this is that, if the node to be removed is in a chain that is being expanded by a thread, the task of making the node unreachable can be delegated to the thread doing the expansion operation. Figure 4.1 shows an example of how a node can become reclaimable later than what would be expected if the delegation did not happen. Avoiding this delegation mechanism is not a viable solution since the expanding thread can always reinsert the node in the new level before realising that it was marked as invalid and made unreachable. Figure 4.2 illustrates the situation.

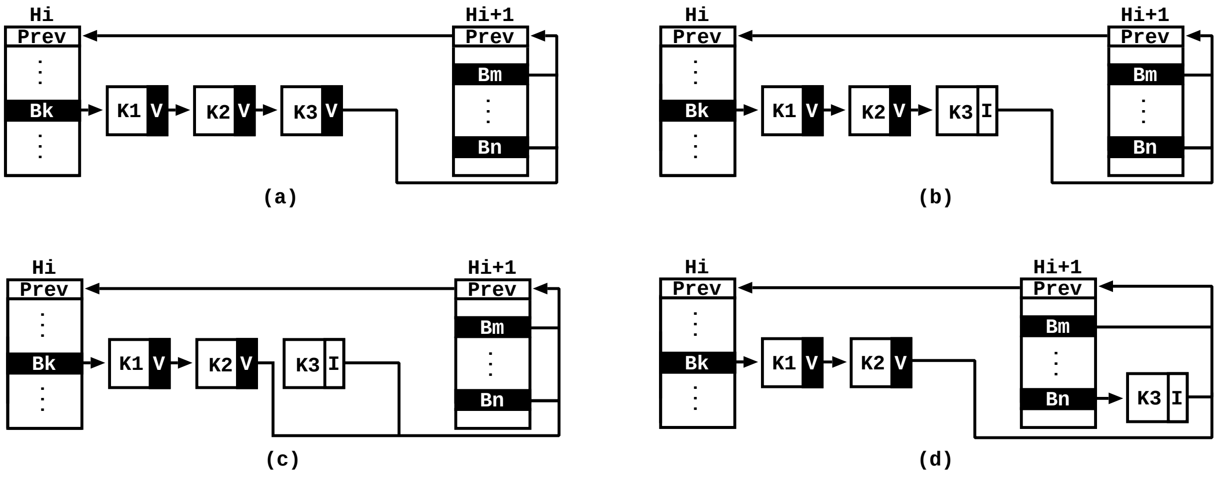

Figure 4.2: Reinsertion of an invalid node during an expansion operation.

The problem resides exclusively on the case where a thread T_1, doing an expansion, reads a valid node $K3$ and before changing the corresponding bucket reference in the new level H_{i+1} in order to expand it (Fig. 4.2(a)), another thread T_2 is able to invalidate $K3$ (Fig. 4.2(b)) and make it unreachable (Fig. 4.2(c)). As the removing thread T_2 does not interfere with the reference in H_{i+1}, the expanding thread T_1 can succeed in updating the bucket reference Bn in H_{i+1} to $K3$ and effectively reinsert $K3$ making it reachable again (Fig. 4.2(d)).

4.1.1 Attempted Solution

To apply the current state-of-the-art methods to the LFHT data structure we need to be able to adapt the design to either: (i) know when a node becomes (permanently) unreachable; or (ii) adjust the remove operation to provide that guarantee. Option one requires keeping track of what nodes become unreachable at every reference update and also requires a close integration of the reclamation method into the data structure as we need to be able to do memory reclamation operations at every reference update. Some methods already require a similar level of integration, but others do not require integration at all. An example is the grace period based methods that under normal circumstances only need to take action at the start or at the end of the data structure operations. Taking that into account, our focus was on the second option.

Our proposal is to adjust the remove operation in the situation where a node is being marked as invalid in a chain that is being expanded (i.e., before making the node unreachable). The idea is to search for the spot where the node would be expanded to, in the new level, and mark that spot with a special tag. That tag would cause the CAS done by the expanding thread to fail and thus avoid the node from being reinserted. The expanding thread would then verify that the node was made invalid in the meantime, after the CAS failure, and would skip its expansion.

To implement this solution, we will take advantage of the remaining unused bits in the state field that are known to be always zero due to the memory alignment (recall that, one bit is already being used for the invalidity flag). We also need to adjust all operations to ignore the new tag bits when dereferencing these memory addresses and to keep the tag unchanged when the state field is updated with a CAS. The remove operation is then adjusted as follows. When the remove operation detects that a node N being invalidated is in a chain referencing a hash level H different from the one it was initially on, it searches for the node N in the new level H. If N is found in H, it means that it was already expanded and we can proceed as usual making N unreachable. Otherwise, we just increment the tag of the reference to where N would be expanded to, and only then, we try to make N unreachable. This action prevents the reinsertion of the node as in the example of Fig. 4.2.

This method was implemented and tested extensively without showing any wrong results. However, there is a critical flaw that, under very specific circumstances, can lead to nodes being reinserted after being made unreachable. The problem arises from the fact that multiple expansions can occur simultaneously in different hash levels of the same path, i.e., multiple expansions can be trying to expand different nodes into the same point and thus overflow the tag and make the reinsertion of an invalid node possible again. This tag overflow reflects what is called the ABA problem [4].

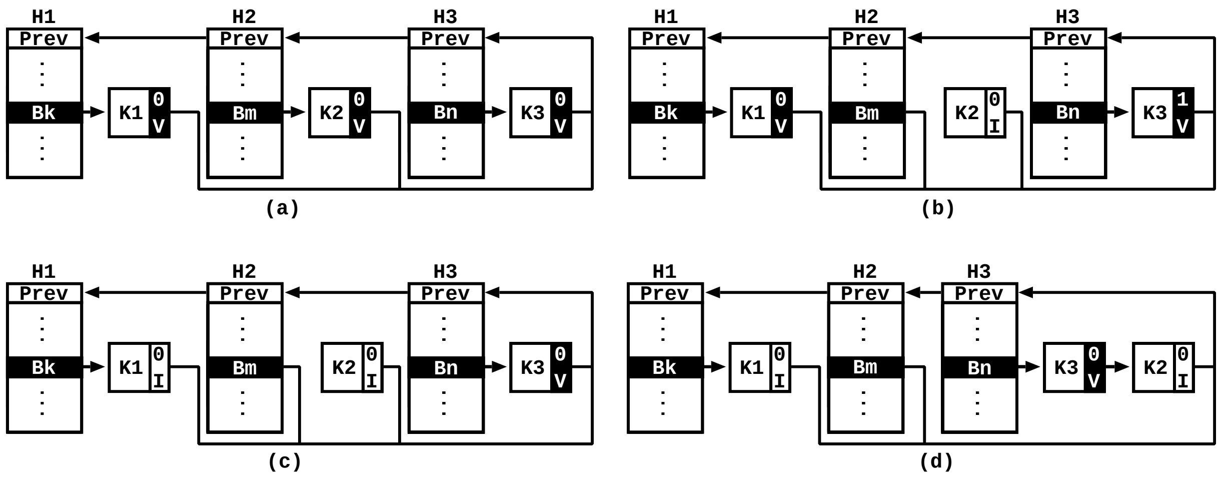

Figure 4.3: Reinsertion of an invalid node due to a tag overflow.

Figure 4.3 shows how the ABA problem can occur in our case. We consider a 1-bit tag and two expansion occurring simultaneously as shown in Fig. 4.3(a). Thread T_1 is expanding the level H_1 (node $K1$) and thread T_2 is expanding the level H_2 (node $K2$) and both nodes ($K1$ and $K2$) are to be expanded to bucket Bn in H_3, i.e., after node $K3$. Now consider that before performing the CAS to move $K2$ into H_3 after $K3$ (the state of $K3$ becomes the first A in ABA), another thread T_3 invalidates $K2$ and, as it detects an ongoing expansion, it increments the tag of $K3$ (becomes the B in ABA) and only then makes $K2$ unreachable (Fig. 4.3(b)). Then T_3

invalidates $K1$ and, as it detects an ongoing expansion, it increments the tag on the expansion point which is again $K3$ (Fig. 4.3(c)). As the tag in this example only has 1 bit, it now overflows and becomes 0 again (becomes the second A in ABA). Finally, in Fig. 4.3(d), T_2 resumes and performs the CAS on $K3$ that wrongfully succeeds due to the tag overflow, thus reinserting the unreachable node $K2$. Note that in order to simplify this example, the tag field of the bucket entries (Bk, Bm and Bn) were omitted from the figure, as they were not needed in this example.

This scenario could still happen even if more bits were available for the tag as multiple nodes could be removed while being expanded in H_1, and more expansions could be happening in levels behind H_1. To fully prevent this scenario from happening, the tag would need to have at least the number of bits given by Eq. 4.1 where L represents the maximum number of levels and C the maximum number of valid nodes in a chain.

$$\lceil log_2((L-2) \times C + 1) \rceil \tag{4.1}$$

This requires too many bits in the tag to prevent the ABA problem. Note that on a 64bit architecture, with aligned allocations, we would have at most 4 available bits for the tag, which is not enough even with aggressive parameters such as a value of 3 for C and a value of 8 for L (64 bits $\div$ 8 levels = 256 buckets per level). Since we were unable to reproduced the ABA problem in our experiments and this does not impact the performance in any meaningful way, we still used this method for benchmarking purposes. In what follows, we present the algorithms that support this implementation.

4.1.2 Algorithms

Since most of the algorithms remain identical to the ones described in the previous chapter, we will only show the ones that were significantly changed. To keep the tags unchanged when updating the node states we introduce a wrapper for the CAS operations that ensures that the tag value remains unchanged. The procedures not shown next only have the default CAS operations replaced by the new *CAS_keep_tag()* as shown in Listing 4.2. The new auxiliary functions to manage the tag are presented in Listing 4.1.

The main changes were introduced in the *make_unreachable()* procedure that has been split in two functions, shown in Listings 4.3 and 4.4. Together, these two functions do exactly the same as before with the exception that if an ongoing expansion is detected we now call *tag_expansion_point()* and only then make the node unreachable.

The *tag_expansion_point()* procedure returns a reference to the hash node where the node (argument *leaf_node*) we are trying to make unreachable is found, which means that *leaf_node* has already been expanded and thus the procedure of making it unreachable can safely be restarted from that hash node. On the other hand, if the *tag_expansion_point()* function does not find *leaf_node*, it tags the expansion point in order to prevent the expansion from occurring

```
#define TAG_POS 1
#define TAG_MASK ((1 << TAG_BITS) - 1) << TAG_POS

node *get_chain_node(node_state state) {
   return state & ~(TAG_MASK | FLAG_MASK)

int get_tag(node_state state) {
   return (state & TAG_MASK) >> TAG_POS;
}

node_state gen_state(node *ptr, int tag, enum flag) {
   if (flag == VALID)
      return ((node_state) ptr) | ((tag << TAG_POS) & TAG_MASK);
   else
      return ((node_state) ptr) | ((tag << TAG_POS) & TAG_MASK) | FLAG_MASK;
}

node_state change_state(node_state state, int tag, enum flag) {
   return gen_state(get_chain_node(state), tag, flag);
}

node_state increment_tag(node_state state) {
   return change_state(state, get_tag(state) + 1, VALID);
}
```

Listing 4.1: Interface functions for tag management.

```
bool CAS_keep_tag(void *addr, node_state expected, node_state desired) {
   return CAS(addr, expected, change_state(desired, get_tag(expected), VALID));
}
```

Listing 4.2: CAS wrapper to ensure that tags remain unchanged.

and then we can safely make the node unreachable in the chain being expanded by calling the *reconnect_chain()* procedure.

The details of the *tag_expansion_point()* procedure are described in Listing 4.5. The procedure traverses the data structure (lines 2–13 and 16–20) until it either finds *leaf_node* (line 14), in which case it returns the given hash node, or it finds the expansion point (lines 21–24), in which case it tries to tag it. If it is able to tag the expansion point, it returns *NULL* in order to tell *make_unreachable()* that it is safe to make the node unreachable in the chain that is being expanded. Otherwise, it restarts the procedure until it is either able to tag the expansion point or find *leaf_node*.

The other relevant changes were introduced in the *adjust_node()* procedure, as presented in Listing 4.6. As we can see, the main differences range from lines 26 to 34, where we no longer verify if a node is invalid after a successful expansion but, instead, we verify if the CAS failed

```
void make_unreachable(node *leaf_node, node *hash_node) {
    node *iter = get_chain_node(leaf_node->leaf.state);
    while (iter->type == LEAF_NODE) {                          // find valid_after
        node_state tmp = iter->leaf.state;
        if (is_valid(tmp))
            break;
        iter = get_chain_node(tmp);
    }
    node_state valid_after = gen_state(iter, VALID);
    while (iter->type != HASH_NODE)                            // find end of chain
        iter = get_chain_node(iter->leaf.state);
    if (iter != hash_node) {                                   // expansion in course
        while (iter->hash.prev != hash_node)
            iter = iter->hash.prev;
        iter = tag_expansion_point(iter, leaf_node);
        if (iter)
            return make_unreachable(leaf_node, iter);
    }
    return reconnect_chain(leaf_node, hash_node, valid_after);
}
```

Listing 4.3: Modified version of the procedure to make a node unreachable in order to be compatible with the state-of-the-art memory reclamation methods.

because the node was invalidated and return in that case (line 33). Note that the *force_cas* procedure fails if *leaf_node* is invalid (line 26), which means that we can avoid the extra check to verify if the node is still valid after finding *valid_addr* and *valid_val*. Otherwise, we could have the tag already incremented by another thread in *valid_val* and succeed the expansion with an invalid or unreachable node.

```
void reconnect_chain(node *leaf_node, node *hash_node, node_state valid_after) {
   int pos = get_bucket(leaf_node->leaf.hash_value, hash_node);
   node_state *valid_addr = &(hash_node->hash.array[pos]);
   node_state valid_val = *valid_addr;
   node *iter = get_chain_node(valid_val);
   while (iter != leaf_node && iter->type == LEAF_NODE) {          // find node again
      node_state tmp = iter->leaf.state;
      if (is_valid(tmp)) {
         valid_addr = &(iter->leaf.state);
         valid_val = tmp;
      }
      iter = get_chain_node(tmp);
   }
   if (iter == leaf_node) {                                  // try to make unreachable
      if (CAS_keep_tag(valid_addr, valid_val, valid_after))
         return;
      else                                                     // failed so try again
         return make_unreachable(leaf_node, hash_node);
   } else if (iter == hash_node)                   // node not found (already unreachable)
      return;
   else {                    // expansion happened before invalidation, try on next level
      while (iter->hash.prev != hash_node)
         iter = iter->hash.prev;
      return reconnect_chain(leaf_node, iter, valid_after);
   }
}
```

Listing 4.4: Procedure to reconect a chain of nodes in order to leave a node unreachable.

```
node *tag_expansion_point(node *hash_node, node *leaf_node) {
   int pos = get_bucket(leaf_node->leaf.hash_value, hash_node);
   node_state *valid_addr = &(hash_node->hash.array[pos]);
   node_state valid_val = *valid_addr;
   node *iter = get_chain_node(valid_val);
   while (iter != leaf_node && iter->type == LEAF_NODE) {       // traverse the chain
      node_state tmp = iter->leaf.state;
      if (is_valid(tmp)) {
         valid_addr = &(iter->leaf.state);
         valid_val = tmp;
      }
      iter = get_chain_node(tmp);
   }
   if (iter == leaf_node)                                        // node found
      return hash_node;
   if (iter != hash_node) {                   // expansion in course, try on next level
      while (iter->hash.prev != hash_node)
         iter = iter->hash.prev;
      return tag_expansion_point(iter, leaf_node);
   }
   if (CAS(valid_addr, valid_val, increment_tag(valid_val))            // try to tag
      return NULL;
   else                                                          // failed so try again
      return tag_expansion_point(hash_node, leaf_node);
}
```

Listing 4.5: Procedure to increment a tag of an expansion point in order to prevent the expansion of an invalid node.

```
void adjust_node(node *leaf_node, node *hash_node) {
   int pos = get_bucket(leaf_node->leaf.hash_value, hash_node);
   node_state *valid_addr = &(hash_node->hash.array[pos]);
   node_state valid_val = clear_tag(*valid_addr);
   node *iter = get_chain_node(valid_val);
   int count = 0;
   while (iter->type == LEAF_NODE) {                      // find the expansion point
      node_state tmp = iter->leaf.state;
      if (is_valid(tmp)) {
         valid_addr = &(iter->leaf.state);
         valid_val = tmp;
         count++;
      }
      iter = get_chain_node(tmp);
   }
   if (iter == hash_node) {                               // expansion point found
      if (count >= MAX_NODES) {                                   // chain is full
         node *new_hash = create_hash_node(hash_node->hash.level+1, hash_node);
         if (CAS(valid_addr, valid_val, new_hash)) {          // try new expansion
            adjust_chain_nodes(hash_node->hash.array[pos], new_hash);
            hash_node->hash.array[pos] = gen_state(new_hash, VALID);
            return adjust_node(leaf_node, new_hash);
         } else                                           // new expansion failed
            free(new_hash);
      } else {                                                    // try to expand
         if (!force_cas(leaf_node, hash_node))
            return;
         if (CAS_keep_tag(valid_addr, valid_val, gen_state(leaf_node, VALID))) {
            return;
         }
      }
      if (is_invalid(leaf_node->leaf.state)               // node was invalidated
         return;
      return adjust_node(leaf_node, hash_node);                           // retry
   }
   while (iter->hash.prev != hash_node)                      // find on next level
      iter = iter->hash.prev;
   return adjust_node(leaf_node, iter);
}
```

Listing 4.6: Modified version of the procedure to adjust a node into a new level in order to be compatible with the state-of-the-art memory reclamation methods.

4.2 New Memory Reclamation Method

As our first attempt to make the LFHT data structure compatible with the existing memory reclamation methods did not succeed, in what follows, we focus on an alternative approach that we took to create novel reclamation method that is integrated in the data structure. By following this approach, we also seek the opportunity to exploit the characteristics of the data structure in order to improve its performance.

4.2.1 Overview

Hazard pointers have optimal memory bounds in memory reclamation, but they rely on global synchronization for every node being traversed by using sequentially consistent atomic writes. Reducing this synchronization overhead without letting the memory bounds explode is a difficult task because, usually, there is not a good way to merge nodes in well-defined groups and protect them with a single hazard pointer.

However, in LFHT, the nodes are already grouped into buckets that have a well-defined maximum size, and we can use a pair <hash, level> to refer to a specific chain of nodes contained in a bucket. The hash argument represents a path in the structure and the level argument represents a portion of this path. With this strategy, we can have a *hazard pair <HH, HL>* formed by a *hazard hash (HH)* and a *hazard level (HL)* to protect a well-defined group of nodes instead of using a hazard pointer to protect a single node, thus reducing the synchronization overhead without letting the memory bounds explode.

4.2.2 Changes to Lock-Free Hash Tries

To successfully implement this idea of reducing synchronization to a minimum while preserving the memory bounds, some changes were made to the data structure and algorithms. The changes were the following:

- Bucket entries were extended to include a *hash flag* indicating if the stored reference is for a next level hash node. The new flag is part of the atomic field, which includes also the reference. As before, whenever the flag is set, the entire atomic field becomes immutable.
- Leaf nodes were extended to include a *generation field* indicating the hash level where the node was first inserted on.
- A *level tag* indicating the hash level where a leaf node is in at any given moment was added as part of the state field, which still includes the next-on-chain node reference and the invalidity flag. Whenever the tag is updated, the entire atomic state field is updated.
- Threads now collaborate to finish all ongoing expansions in a path before inserting new

nodes in such path. This ensures that no more than one expansion will be occurring in a path at any given moment.

- If the number of nodes blocked from reclamation by a thread's hazard pair exceeds a given threshold, an expansion operation is forced on that chain. As we will see, this prevents a single thread from blocking a potentially unlimited number of nodes from being reclaimed.
- The traversal of the data structure has been changed to ensure that a node is blocked by a hazard pair before accessing it. The extra information added to the atomic fields also allows for a more efficient traversal.
- The list of hazard pairs needs to be read twice in the same order when doing reclamation, as a node can be added to the reclamation list before becoming unreachable due to the delegation process.

4.2.3 Main Idea

In general, to reclaim memory, we need to be sure that there is no thread with a reference to the node we are reclaiming. As we have seen, each thread will block a chain of nodes of a bucket entry therefore a leaf node can only be reclaimed if: (i) it is not in a blocked chain; and (ii) it has never been in the past, as a thread could have *seen* it there and, in the meantime, the leaf node could have been expanded to a further level.

To guarantee these restrictions, we need the following information: (i) the hash value of the node defines the path where the node might ever be; (ii) the generation field defines the first level the node was in; and (iii) the level tag, that becomes immutable as the node is invalidated, defines the level in which the node was removed. Together, this information holds the set of chains where the node has been during its life time.

To ensure that each thread blocks the correct chain of nodes from reclamation, when traversing it, we will make use of the hash flag in the bucket entries, the level tag in the leaf nodes and the knowledge that no node is inserted in a path being expanded. The diagram in Fig. 4.4 summarizes the traversal procedure when a thread is looking for a node represented by a given *Hash Value*. When traversing a hash node (HN), the hash flag determines if the reference in a bucket entry is for the same level (SAME_LEVEL), case in which we update the hazard level (HL) before following the bucket entry. When traversing a chain, we rely on the level tag to know if an expansion is happening concurrently. If we find a level tag that is greater than the current hazard level plus one, then we know that the level in which we started the traversal of the chain has already been completely expanded. This means that we can reread the hash node where we started the current traversal in order to move to the next level (as we know it will be now referencing a new hash node). If we find a level tag that is equal to the hazard level plus one, we know that either the current level is being expanded or was already expanded, so we reread the hash flag on the bucket entry and determine in which situation we are in. If the hash flag is set (NEW_LEVEL), we know that the last hash node now references a new one and we can just continue by following that reference. If the hash flag is not set (SAME_LEVEL), that means the expansion is still in course and the reference we have read with a level tag equal to the hazard level plus one references a leaf node that has already been in the previous level. This means that the current hazard level, although lower than the level where the node is, it is still blocking that node and we can safely read the node without updating the hazard level.

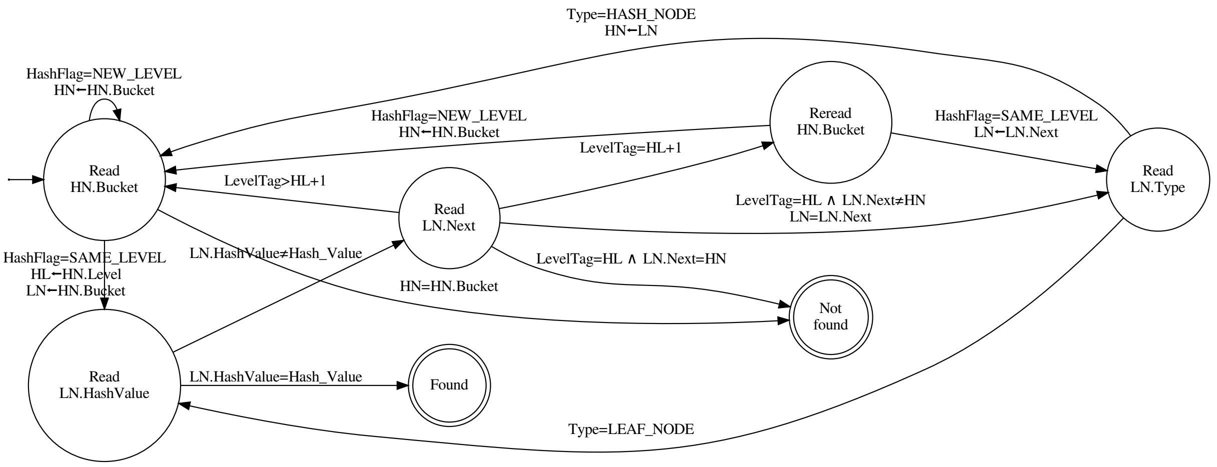

Figure 4.4: Diagram representing the traversal procedure.

The guarantee that no node can be inserted while there is an expansion in course, is only possible in a lock-free manner (i.e., without locks), if threads trying to insert nodes can collaborate in the expansion. For this collaboration to work, apart from having to continuously check if the expansion has already finished as shown before, we need to verify if the node N we are trying to expand has not been expanded yet. This is done by checking if any of the nodes traversed in the new level is N while finding the point where N will be expanded to, and then, doing an additional check to verify if it is still valid and referencing the hash node it is being expanded to. Doing this additional check prevents the problem exemplified in Fig. 4.5 where thread T_1 starts expanding the node $K3$ (Fig. 4.5(a)) but before T_1 starts traversing the new level H_{i+1}, a thread T_2 expands nodes $K3$ and $K2$ (Fig. 4.5(b)) and then another thread T_3 removes the node $K3$ (Fig. 4.5(c)). When T_1 resumes and expands node $K3$ it creates a loop as shown in Figure 4.5(d).

Regarding the removal operation, it remains mostly the same, but now we can detect earlier if an expansion has started by just seeing a level tag or a hash level with a greater value than the level tag of the node after being invalidated and thus delegate the task of making it unreachable sooner. At the end of the removal operation, even if the task of making the node unreachable was delegated, the node is added to the thread's local queue for reclamation and, at every N additions to the reclamation queue, with N being an arbitrary threshold, a reclamation procedure is triggered.

The reclamation procedure starts by reading the list of hazard pairs belonging to all the running threads and by storing them in local storage, much like as in the hazard pointers method. However, here we need to do this process twice (one after the other) and use the two copies for the memory reclamation process. This additional read of the list of hazard pairs is needed due

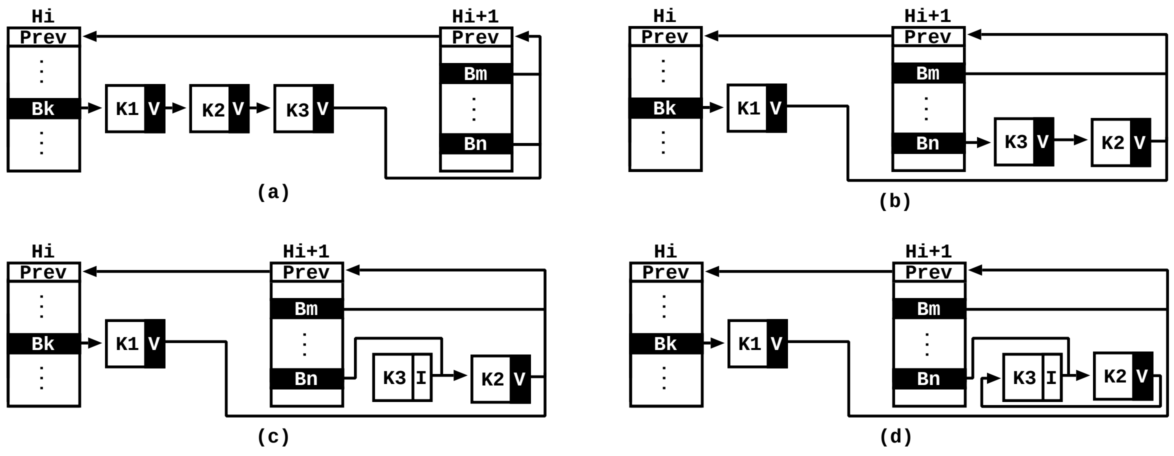

Figure 4.5: Problem caused by the lack of the extra check during an expansion.

to the delegation of the task of making the node unreachable during expansions. What could happen is that, when we start the reclamation process of a node N, it might be reachable from the head of the data structure (when a delegation happens) but blocked by the thread doing the respective expansion. In this situation, when we are reading the list of hazard pairs it could happen that, we read the entry from thread T_1 while it is still doing an unrelated operation, but then, before we read the entry from T_2, that is doing the expansion along with the delegated process of making N unreachable. In this situation T_1 could reach N and then T_2 could finish the expansion, leaving N unreachable. In this scenario, both T_1 and T_2 hazard pairs that we have read are not blocking the node N and, as such, we would consider N to be safe to reclaim, which was not the case as T_1 has a reference to it. The second read of the list of hazard pairs prevents this scenario from happening because if the thread T_2 finishes before we finished the first read of the list, from that point onwards the node N is certain to be unreachable and thus impossible for any thread to acquire a reference to it. Note that a node not blocked by any thread in the first read is, by default, unreachable but it is not certain to be reclaimable. As such, a node not blocked by any thread in both reads is certainly unreachable and reclaimable.

After reading twice the list of hazard pairs, we go through each node on the reclamation queue and check if it is being blocked by any hazard pair, i.e., if the hazard level is between the generation and the level tag and if the hazard hash equals the hash value of the node up to the hazard level. If we find such a pair, we skip the node. Otherwise, we remove it from the queue and reclaim its memory. Note that the time complexity of this operation can be reduced in similar ways to the ones described in the hazard pointers paper [13].

Everything discussed so far is enough to make this reclamation method to work. However, it does not ensure a finite memory bound. The reason is that an infinite number of nodes can be inserted and removed from a specific chain without ever triggering an expansion. This is rather unrealistic but nonetheless possible. And so, a thread that suspends or fails in such a chain or a group of threads that are continuously working on a chain in such a way that at any given time at least one is traversing it, would prevent infinite amounts of memory from being reclaimed. This issue can be solved by counting how many nodes each thread T is blocking from reclamation during the reclamation procedure, and if one of the counters reaches a predefined threshold, an expansion operation is forced on that specific chain. This would not guarantee the reclamation of the nodes which are causing the problem, but prevents thread T from blocking more new nodes from being reclaimed until it progresses. Later, when T progresses, the previously blocked nodes would be reclaimable again as no thread can acquire a hazard pair for a chain that has already been expanded. This not only provides us with a well-defined and flexible memory bound but can also improve performance, as an expansion would likely divide the multiple threads concurrently working on a specific chain between multiple chains, thus reducing contention in that section.

The fact that we cannot force an expansion on the last level is not a problem. In the last level, the solution is to traverse the chain exactly as in the hazard pointers method [13], using the hazard level to inform the reclamation procedure that the hazard pointers method should be used to reclaim such nodes. Since the last level cannot be expanded, delegations cannot happen either and thus the hazard pointers method works here as intended. Note that this situation is extremely rare, as the data structure needs to be almost full and the hash function not be doing a very good job.

4.2.4 Limitations

Since this memory reclamation method is tightly integrated with the LFHT, it could be adapted to work with similar data structures but not with the generality of lock-free data structures.

The usage of the alignment bits to store the invalidity flag and the level tag limits the amount of information that we can store on the level tag, which in turn limits the maximum amount of levels that the data structure can have and consequently the minimum size of the bucket arrays. For example, assuming A as the address size in bits of the architecture in question (e.g., 32 or 64 bit size addresses) and the minimum addressable size of 8 bits, i.e., 1 byte, then $\frac{A}{8}$ is the number of possible addressable positions in a address sized value (e.g., in a 32 bit value we have $\frac{32}{8} = 4$ possible addressable positions). Since a leaf node has a size of 4 addresses: one for the hash value; one for the state field; one for the generation field and node type; and one for the reference to whatever the node is storing. The number of addressable positions in a node is thus $\frac{4A}{8}$. Therefore, the amount of bits available for the level tag (excluding the invalidity flag) is given by Eq. 4.2.

$$log_2\left(\frac{4A}{8}\right) - 1 = log_2\left(\frac{A}{4}\right) \tag{4.2}$$

This means that the maximum amount of levels is defined by Eq. 4.3

$$2^{log_2(\frac{A}{4})} = \frac{A}{4} \tag{4.3}$$

And consequently, the minimum amount of bits consumed per level, assuming all levels of the same size, is 4 as we can see in Eq. 4.4

$$\frac{A}{\frac{A}{4}} = 4 \tag{4.4}$$

Also, the minimum size of the bucket array is 16 entries, as shown in Eq. 4.5

$$2^4 = 16 \tag{4.5}$$

As we can see, these limits are all reasonable, so they ended up as being more a concern to the definition of the parameters than anything else.

To actually use all the alignment bits to store the level tag and the invalidity flag, we need to allocate memory in an aligned way. This is done with the *aligned_alloc()* function that has the limitation of only allowing allocation sizes multiple of the alignment, wich is not a problem since the size of the leaf and hash node is already a multiple of the alignment, thus allowing the memory allocator to perform these allocations as efficiently as normal allocations, as in the case of the lock-free memory allocator LRMalloc by Leite and Rocha [12].

4.2.5 Guarantees

In addition to guarantee lock-freedom, this memory reclamation method has well-defined memory bounds and low synchronization overhead. The memory bounds are achieved through the use of the hazard pairs and the forced expansions, which define a very fine control over the maximum amount of unreclaimed memory we want to allow. To adjust this limit, we have four main variables: (i) the number T of threads, (ii) the frequency F at which we do the reclamation procedure, i.e., the reclamation procedure is done every F removals done by a thread; (iii) the threshold E of blocked nodes by a single thread that triggers an expansion; and (iv) the maximum number C of nodes in a chain. Given these parameters, the memory bound is given by Eq. 4.6.

$$T^2 \times (E + F + C) + T \times F \tag{4.6}$$

The $E + F + C$ factor comes from the fact that in one reclamation procedure a thread can see $E - 1$ nodes blocked by a thread and in the next iteration this number could have been increased by F, as F new removals could have occurred in the same chain. Then a maximum of C nodes can still be present and then removed from the blocked chain. The multiplying factor T^2 is due to the fact that for a specific thread this can happen once for every other existing thread ($T - 1$ occurrences per thread), and every thread can be in such a situation ($T \times (T - 1)$). The factor $T \times F$ is justified by every thread being in the state right before starting the reclamation procedure, so they might have F more new nodes to reclaim since the last time.

Regarding the synchronization overhead of the method, on average, it is expected to be just two atomic writes per operation, one for the hazard hash and another for the hazard level, as in the most common case only these two writes are required. In the case where we are traversing a chain while an expansion is occurring, we need an extra atomic read per node traversed and one atomic write to the hazard level if the expansion finishes during the traversal. Note that this is an uncommon situation, which in a lock based approach would require a lock. In a worst case scenario, where we have multiple hash collisions until the last level, we fall back to a linked list with hazard pointers as the reclamation method. That makes the data structure equivalent to a lock-free linked list in performance. This fallback method was implemented mainly as a safety measure to allow such collisions to occur without failure and to keep the memory bound from ever being exceeded even under such extreme circumstances.

4.2.6 Algorithms

This subsection presents in detail the algorithms that implement the modified version of LFHT to include the new memory reclamation method, which we named Hazard Hash and Level (HHL). We start by introducing the new auxiliary functions to manage the hash flag and level tag in Listing 4.8. Next, we present in Listing 4.9 the modified version of the *find_node()* procedure that now includes the extra ability of finding invalid nodes. The *find_node()* procedure will be used in three main scenarios: (i) to find a valid node based on its hash value, or its insertion point if it does not exist; (ii) to find an invalid node and its preceding *valid_addr* and *valid_val*; and (iii) to find the expansion point for a given hash value. Listing 4.7 shows the prototypes for this use cases.

The modified version of the *find_node()* procedure receives two extra arguments:

- *invalid*: the pointer to an invalid node we want to find again; the value *NULL* is used if a valid node is meant to be found.
- *prev_hash*: the address of a pointer to the last hash node we have updated the hazard level at; it references *NULL* if we have not updated the hazard level in the previous or current level (note that *prev_hash* can be updated on every call to *find_node()*).

The *invalid* argument is an input argument and the *prev_hash* is both an input and an output argument.

When we start a traversal in a bucket entry, we can find ourselves in one of three situations. In the first situation, the hash flag is set (i.e., it references a new hash level), thus we can restart the traversal in the new level and set *prev_hash* to *NULL*, as we have not updated the hazard level in the current level (lines 11–14). In the second situation, the flag is not set (i.e., it references a leaf node or itself) and *prev_hash* is either *NULL* or references the previous hash node, which in turn references the current hash node directly (this means that there is no expansion occurring in the previous level). In such situation, we update the *prev_hash* and the hazard level with

```
node *find_valid_node(size_t hash_value,
                      node **hash_node,
                      node **prev_hash,
                      insertion_info args) {
   return find_node(hash_value, NULL, hash_node, prev_hash, args);
}

node *find_invalid_node(node *invalid,
                        node **hash_node,
                        node **prev_hash,
                        insertion_info args) {
   return find_node(invalid->leaf.hash_value, invalid, hash_node, prev_hash,
       args);
}

node *find_expansion(size_t hash_value,
                     node **hash_node,
                     node **prev_hash,
                     insertion_info args) {
   return find_node(hash_value, DUMMY_NODE_PTR, hash_node, prev_hash, args);
}
```

Listing 4.7: Use cases for the *find_node()* procedure.

update_hazard_level and restart the procedure in the same level (lines 15–19). In the third situation, the hash flag is not set and *prev_hash* references the current hash node or references the previous hash node and it does not have the hash flag set, which means an expansion is occurring in that level. Thus, in this situation, we continue traversing the chain (lines 21–49).

The procedure to continue traversing the chain works similarly to the original *find_node()* except that we now can find an invalid node (line 37), and that we need to keep track of the tags to be sure we only dereference leaf nodes protected by our hazard pair (lines 39–47).

The *search_node()* procedure presented in Listing 4.10 is almost the same as the original except that it now starts by updating the hazard hash to the hash value of the node it is searching for with the *update_hazard_hash()* procedure.

The *search_insert()* procedure remains mostly the same as shown in Listing 4.11. The main changes are the need to help finishing the expansions in course before inserting a new node (lines 8–12) and setting the correct tags and generations (lines 18, 28 and 32).

The *expand_hash()* procedure just wraps the extra steps of verifying if the expansion has already finished before starting (lines 2–4) and setting the bucket entry to the new hash node now with the hash flag set (line 9) as shown in Listing 4.12. Remember that we do not need to keep level tags in the bucket entries.

The modified version of *adjust_chain_nodes()* procedure is presented in Listing 4.13. The

```
#define ADDRESS_SIZE 64
#define TAG_POS 1
#define TAG_MASK (((ADDRESS_SIZE/4)-1) << TAG_POS)

// auxiliary functions

int get_level_tag(node_state state) {
   return state & TAG_MASK;
}

int get_hash_flag(node_state state) {
   return state & FLAG_MASK;
}

// interface functions

node *get_chain_node(node_state state) {
   return (node *) (state & ~(TAG_MASK | FLAG_MASK));

int is_new_level(node_state state) {
   return get_hash_flag(state);
}

int is_same_level(node_state state) {
   return !get_hash_flag(state);
}

node_state gen_state(node *ptr, int tag, enum flag) {
   if (flag == INVALID || flag == NEW_LEVEL)
      return ((node_state) ptr) | (tag << TAG_POS) | FLAG_MASK;
   else
      return ((node_state) ptr) | (tag << TAG_POS);
}

node_state change_state(node_state state, int tag, enum flag) {
   return gen_state(get_chain_node(state), tag, flag);
}
```

Listing 4.8: Interface functions for the management of the new hash flag and level tag.

adjust_chain_nodes() procedure now receives one extra argument, *prev_bucket*, that is a pointer to the bucket entry being expanded. The *prev_bucket* will have its hash flag set when the expansion finishes and will serve to guarantee that we do not dereference a node that is not blocked by the hazard pair. The main difference to the original *adjust_chain_nodes()*, is again the checks on the level tag and hash flag to guarantee that no unprotected node is dereferenced. If any of these checks succeeds then it does not only mean that the node we were trying to dereference may not be protected but also that the expansion was finished by another thread and we can return.

```
node *find_node(size_t hash_value,
                node *invalid,
                node **hash_node,
                node **prev_hash,
                insertion_info *args) {
   int pos = get_bucket(hash_value, *hash_node);
   node_state tmp = (*hash_node)->hash.array[pos];
   node *iter = get_chain_node(tmp);
   if (*prev_hash)
      int prev_pos = get_bucket(hash_value, *prev_hash);
   if (is_new_level(tmp)) {                                  // new hash level
      *hash_node = iter;
      *prev_hash = NULL;
      return find_node(hash_value, invalid, hash_node, prev_hash, args);
   } else if (!*prev_hash || (*prev_hash != *hash_node &&
                 is_new_level((*prev_hash)->hash.array[prev_pos]))) {
      *prev_hash = *hash_node;
      update_hazard_level((*hash_node)->hash.level);
      return find_node(hash_value, invalid, hash_node, prev_hash, args);
   }
   args->valid_addr = &((*hash_node)->hash.array[pos]);
   args->valid_val = tmp;
   args->count = 0;
   while (iter != *hash_node) {                              // traverse the chain
      if (iter->type == HASH_NODE) {                         // new hash node found
         *hash_node = iter;
         return find_node(hash_value, invalid, hash_node, prev_hash, args);
      }
      tmp = iter->leaf.state;
      if (is_valid(tmp)) {                                   // found a valid node
         if (!invalid && iter->leaf.hash_value == hash_value) {
            return iter;
         }
         args->valid_addr = &(iter->leaf.state);
         args->valid_val = tmp;
         (args->count)++;
      } else if (iter == invalid)                            // invalid node found
         return iter;
      if (get_level_tag(tmp) > ((*prev_hash)->hash.level+1)) {      // 2 expansions
         *prev_hash = NULL;
         return find_node(hash_value, invalid, hash_node, prev_hash, args);
      } else if (get_level_tag(tmp) > (*prev_hash)->hash.level &&     // expansion
                 is_new_level((*prev_hash)->hash.array[prev_pos])) {
         *hash_node = get_chain_node((*prev_hash)->hash.array[prev_pos]);
         *prev_hash = NULL;
         return find_node(hash_value, invald, hash_node, prev_hash, args);
      }
      iter = get_chain_node(tmp);
   }
   return NULL;
}
```

Listing 4.9: Procedure to find a node or where the node would be inserted based on its hash value or address, extended to support the HHL memory reclamation method.

```
void *search_node(size_t *hash_value, node *hash_node) {
   update_hazard_hash(hash_value);
   node *prev_hash = NULL;
   insertion_info args;
   node *found = find_valid_node(hash_value, &hash_node, &prev_hash, &args);
   if (found)
      return found->leaf.contents;
   else
      return NULL;
}
```

Listing 4.10: Procedure to find a node based on its hash value that returns its contents, extended to support the HHL memory reclamation method.

In the *adjust_node()* procedure presented in Listing 4.14, apart from the usual hash flag and level tag checking, we have an extra verification to prevent the problem exemplified in Fig. 4.5 (lines 23–25).

The modified version of the *make_unreachable()* procedure is presented in Listing 4.15. The *make_unreachable()* procedure now can terminate earlier, when trying to find the *valid_after* node, if it finds that an expansion is occurring in the chain where the node *leaf_node* was made invalid at (lines 11–12). It also needs to do the additional checks, if an expansion is occurring in the level before the one that the node was made unreachable at, and if such an expansion terminates it needs to update the hazard level (lines 13–20). To find *valid_addr* and *valid_val* it now uses the *find_node()* procedure as explained before.

The pseudo-code for the modified *search_remove()* procedure is presented in Listing 4.16. The differences from the original version are: the call to *update_hazard_hash* at the start to update the hazard hash (line 2); after finding the node and marking it as invalid (line 8) it needs to update the *hash_node* and hazard level as the node could have been expanded between finding it and marking it invalid; and, finally, after making the node unreachable we call *reclaim_node()* to reclaim it (13–14).

The *update_hazard_hash()* and *update_hazard_level()* procedures are presented in Listing 4.17 and the *reclaim_node()* procedure in Listing 4.18.

The *update_hazard_hash()* and *update_hazard_level()* procedures just update the corresponding entry in the hazard pairs array. The *reclaim_node()* procedure starts by adding the node to be reclaimed to a thread's local list and by incrementing a counter (lines 5–9). When the counter reaches a threshold value *MR_THRESHOLD*, the reclamation procedure is started (line 10). The reclamation procedure starts by resetting the counter and by allocating a local array *ha* with twice the size needed to hold the hazard pairs of all threads and a counter (lines 11–13). Then it reads the array of hazard pairs two times in the same order and stores them on the local array (lines 14–18). Next every node on the local list is compared to every entry of the local array, and if it is protected by one pair the respective counter is increased and the

```
void *search_insert(size_t hash_value, node *hash_node, void *contents) {
   update_hazard_hash(hash_value);
   node *prev_hash = NULL;
   insertion_info args;
   node *leaf_node = find_valid_node(hash_value, &hash_node, &prev_hash, &args);
   if (leaf_node)                                               // node already exists
      return leaf_node->leaf.contents;
   if (prev_hash != hash_node) {                              // help expansion in course
      expand_hash(hash_value, prev_hash, hash_node);
      update_hazard_level(hash_node->hash.level);
      prev_hash = hash_node;
   }
   if (args.count >= MAX_NODES) {                                       // chain is full
      node *new_hash = create_hash_node(hash_node->hash.level+1, hash_node);
      if (CAS(                                                          // try to expand
            args.valid_addr,
            args.valid_val,
            gen_state(new_hash, new_hash->hash.level, VALID))) {
         expand_hash(hash_value, hash_node, new_hash);
         return search_insert(hash_value, new_hash, contents);
      } else                                                        // expansion failed
         free(new_hash);
   } else {                                                             // try to insert
      node *new_node = create_ans_node(
            hash_value,
            contents,
            hash_node->hash.level,
            gen_state(hash_node, hash_node->hash.level, VALID));
      if (CAS(
            args.valid_addr,
            args.valid_val,
            gen_state(new_node, get_level_tag(args.valid_val), VALID)))
         return new_node->leaf.contents;
      else                                                          // insertion failed
         free(new_node);
   }
   return search_insert(hash_value, hash_node, contents);                      // retry
}
```

Listing 4.11: Procedure to insert a node with a hash value/contents pair, extended to support the HHL memory reclamation method.

node is skipped (lines 27–32). Otherwise, the node is freed and removed from the local list (lines 34–38). Finally, we check all the counters, and for every one that reaches a threshold value *FE_THRESHOLD*, we call the *force_expansion()* procedure in order to force an expansion on that spot, thus preventing more nodes from being protected by the corresponding hazard pair.

Finally we present the *force_expansion()* procedure in Listing 4.19. The *force_expansion()* procedure just finds the insertion point for the new hash node through the *find_node()* procedure.

```
 1 void expand_hash(size_t hash_value, node *prev_hash, node *hash_node) {
 2    int pos = get_bucket(hash_value, prev_hash);
 3    node_state tmp = prev_hash->hash.array[pos];
 4    if (is_same_level(tmp)) {
 5       adjust_chain_nodes(
 6             get_chain_node(tmp),
 7             &(prev_hash->hash.array[pos]),
 8             hash_node);
 9       prev_hash->hash.array[pos] = gen_state(hash_node, 0, NEW_LEVEL);
10    }
11 }
```

Listing 4.12: Procedure to expand a bucket entry into a new level, extended to support the HHL memory reclamation method.

```
 1 void adjust_chain_nodes(node *leaf_node,
 2                         node_state *prev_bucket,
 3                         node *hash_node) {
 4    node_state tmp = leaf_node->leaf.state;
 5    node *next = get_chain_node(tmp);
 6    if ((get_level_tag(tmp) >= hash_node->hash.level && next != hash_node) ||
 7        is_new_level(*prev_bucket))
 8       return;                               // expansion alredy compleated
 9    if (next != hash_node)                    // adjust in reverse order
10       adjust_chain_nodes(next, prev_bucket, hash_node);
11    if (is_valid(tmp)) {                      // adjust node only if valid
12       if (is_new_level(*prev_bucket))
13          return;
14       if (!force_cas(
15             leaf_node,
16             gen_state(hash_node, hash_node->hash.level, VALID)))
17          return;
18       adjust_node(leaf_node, prev_bucket, hash_node);
19    }
20    return;
21 }
```

Listing 4.13: Procedure to adjust a chain of nodes into a new level, extended to support the HHL memory reclamation method.

Then it verifies if the insertion point is in the given level (line 6) and, if so, it tries to insert the new hash node and execute the expansion (line 20). If it fails to insert the new hash node it restarts the procedure.

The procedure for traversing a chain of nodes in the last level is not presented here as it would be exactly the same as the implementation of linked lists in the hazard pointers method [13].

```
void adjust_node(node *leaf_node, node_state *prev_bucket, node *hash_node) {
   int count = 0;                                                    // find expansion point
   int pos = get_bucket(leaf_node->leaf.hash, hash_node);
   node_state *valid_addr = &(hash_node->hash.array[pos]);
   node_state valid_val = *valid_addr;
   node *iter = get_chain_node(valid_val);
   if (is_new_level(valid_val) || is_new_level(*prev_bucket))
      return;
   while (iter != leaf_node && iter->type == LEAF_NODE) {
      node_state tmp = iter->leaf.state;
      if ((get_level_tag(tmp) != hash_node->hash.level) ||
          is_new_level(*prev_bucket))
         return;
      if (is_valid(tmp)) {
         valid_addr = &(iter->leaf.state);
         valid_val = tmp;
         count++;
      }
      iter = get_chain_node(tmp);
   }
   if (iter == leaf_node)
      return;
   if (leaf_node->leaf.state !=                                      // check to prevent loop
       gen_state(hash_node, hash_node->hash.level, VALID))
      return;
   if (CAS(                                                          // try to expand
         valid_addr,
         valid_val,
         gen_state(leaf_node, get_level_tag(valid_val), VALID))) {
      if (is_invalid(leaf_node->leaf.state) &&
          get_chain_node(leaf_node->leaf.state) == hash_node) {
         if (is_new_level(*prev_bucket)) {
            update_hazard_level(hash_node->hash.level);
            make_unreachable(leaf_node, hash_node, hash_node);
         } else {
            make_unreachable(leaf_node, hash_node, hash_node->hash.prev);
         }
      }
      return;
   }
   return adjust_node(leaf_node, prev_bucket, hash_node);            // retry
}
```

Listing 4.14: Procedure to adjust a single node into the next level, extended to support the HHL memory reclamation method.

```
void make_unreachable(node *leaf_node, node *hash_node, node *prev_hash) {
   node_state tmp = leaf_node->leaf.state;
   node *iter = leaf_node;
   node *valid_after = NULL;
   node_state check;
   unsigned node_tag = get_level_tag(tmp);
   int pos;
   pos = get_bucket(leaf_node->leaf.hash_value, prev_hash);
   while (iter->type == LEAF_NODE) {                          // find valid_after
      tmp = iter->leaf.state;
      if (get_level_tag(tmp) > node_tag)
         return;
      if (get_level_tag(tmp) > prev_hash->hash.level &&
          is_new_level(check = prev_hash->hash.array[pos])) {
         hash_node = get_chain_node(check);
         update_hazard_level(hash_node->hash.level);
         prev_hash = hash_node;
         pos = get_bucket(leaf_node->leaf.hash_value,hash_node);
         continue;
      }
      if (!valid_after && is_valid(tmp))
         valid_after = iter;
      iter = get_chain_node(tmp);
   }
   if (!valid_after)
      valid_after = iter;
   insertion_info args;                                       // find node again
   if (find_invalid_node(leaf_node, &iter, &prev_hash, &args)) {
      if (CAS(                                         // try to make unreachable
            args.valid_addr,
            args.valid_val,
            gen_state(valid_after, get_level_tag(args.valid_val), VALID)))
         return;
      else                                                   // failed so retry
         return make_unreachable(leaf_node, iter, prev_hash);
   }
   return;
}
```

Listing 4.15: Procedure to make an invalid node unreachable if it is not being concurrently adjusted, extended to support the HHL memory reclamation method.

```
void search_remove(size_t hash_value, node *hash_node) {
   update_hazard_hash(hash_value);
   insertion_info args;
   node *prev_hash = NULL;
   node *found = find_valid_node(hash_value, &hash_node, &prev_hash, &args);
   node *next;
   if (found) {
      if (next = mark_invalid(found)) {
         if (!find_invalid_node(found, &hash_node, &prev_hash, &args))
            return;
         if (get_level_tag(next) < hash_node->hash.level)
            return;
         make_unreachable(found, hash_node, prev_hash);
         reclaim_node(hash_node, found);
      }
   }
}
```

Listing 4.16: Procedure to remove a node based on its hash value, extended to support the HHL memory reclamation method.

```
void update_hazard_level(int level) {
   mr_entry *array = get_hazard_array();
   int thread_id = get_thread_id();
   array[thread_id].hazard_level = level;
}

void update_hazard_hash(size_t hash_value) {
   mr_entry *array = get_hazard_array();
   int thread_id = get_thread_id();
   array[thread_id].hazard_hash = hash_value;
}
```

Listing 4.17: Procedures to update the hazard level and hazard hash.

```
void reclaim_node(node *hash_node, node *leaf_node) {
   mr_entry *array = get_hazard_array();
   int thread_id = get_thread_id();
   int max_threads = get_total_threads();
   reclamation_list *new = malloc(sizeof(reclamation_list));
   new->leaf_node = leaf_node;                          // add node to reclamation queue
   new->next = array[thread_id].thread_list;
   array[thread_id].thread_list = new;
   array[thread_id].count++;
   if (array[thread_id].count >= MR_THRESHOLD) {                    // start reclamation
      array[thread_id].count = 0;
      hazard_array *ha;
      ha = alloca(2 * max_threads * sizeof(hazard_array));
      for (int i = 0; i < 2 * max_threads; i++) {               // read hazard pairs 2x
         ha[i].hash_value = array[i % max_threads].hazard_hash;
         ha[i].level = array[i % max_threads].hazard_level;
         ha[i].count = 0;
      }
      reclamation_list **ptr = &(array[thread_id].thread_list);
      while (*ptr) {                                      // go through each node on queue
         unsigned gen = (*ptr)->leaf_node->gen;
         unsigned tag = get_level_tag((*ptr)->leaf_node->leaf.state);
         size_t hash_value = (*ptr)->leaf_node->leaf.hash_value;
         reclamation_list *tmp;
         int reclaim = TRUE;
         for (int i = 0; i < 2 * max_threads; i++) {
            if (gen <= ha[i].level && tag >= ha[i].level &&
                match_hash(ha[i].level, ha[i].hash_value, hash_value)) {
               reclaim = FALSE;
               ha[i].count++;
               break;
            }
         }
         if (reclaim) {
            tmp = *ptr;
            *ptr = (*ptr)->next;
            free(tmp->leaf_node);
            free(tmp);
         } else
            ptr = &((*ptr)->next);
      }
      for (int i = 0; i < 2 * max_threads; i++) {           // check threshold to expand
         if (ha[i].count > FE_THRESHOLD &&
             ha[i].level < MAX_LEVEL)
            force_expansion(hash_node, ha[i].hash_value, ha[i].level);
      }
   }
}
```

Listing 4.18: Procedure to reclaim nodes.

```
void force_expansion(size_t hash_value, node *hash_node, int level) {
   update_hazard_hash(hash_value);
   insertion_info args;
   node *prev_hash = NULL;
   find_expansion(hash_value, &hash_node, &prev_hash, &args);
   if (hash_node->hash.level == level) {
      node *new_hash = create_hash_node(hash_node->hash.level+1, hash_node);
      int pos = get_bucket(hash_value, hash_node);
      if (args.valid_addr == &(hash_node->hash.array[pos])) {
         if (!CAS(
               args.valid_addr,
               args.valid_val,
               gen_state(new_hash, new_hash->hash.level, NEW_LEVEL)))
            return force_expansion(hash_value, hash_node, level);
      } else {
         if (CAS(
               args.valid_addr,
               args.valid_val,
               gen_state(new_hash, new_hash->hash.level, VALID)))
            expand_hash(hash_value, hash_node, new_hash);
         else
            return force_expansion(hash_value, hash_node, level);
      }
   }
}
```

Listing 4.19: Procedure to force an expansion on a specific chain.

Chapter 5

Experimental Results

In this chapter, we begin by introducing our benchmarking methodology and by presenting the versions of the data structure that we considered relevant for the experiments. Next, we present the results obtained along with a discussion for their possible explanation.

5.1 Methodology

To run our experiments, we have developed a benchmarking tool which compiles the version of the data structure we want to use, together with a small module that controls the execution. Our benchmarking tool receives as input 5 parameters: (i) the number T of threads to be considered; (ii) the number N of total operations to be executed; (iii) the percentage Pi of insert operations; (iv) the percentage Pr of remove operations; and (v) the percentage Ps of search operations. Figure 5.1 shows a visual representation of the benchmarking tool, in which the controller receives the 5 parameters and communicates with the data structure through the specific API that was created to run the insert, remove and search operations.

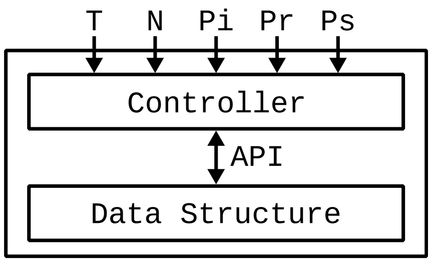

Figure 5.1: Benchmarking tool diagram.

The benchmarking tool has three stages. In the initial stage, the tool prepares the environment of the run. The tool starts by launching the given number of threads T and by setting up the data structure with each thread inserting the values that will be searched and removed. The number of operations is divided equally among the threads and each thread receives a pre-defined seed to be used by the pseudo-random number generator (PRNG). The PRNG is then used to generate keys, and the range of the PRNG is divided according with the given percentages to decide if a key corresponds to a insert, a remove or a search operation.

In the second stage, the tool executes the benchmarks. It starts by resetting the seed of each thread to its default value and marking the execution time. Then, each thread performs its set of operations according with the values given by the PRNG. When all threads finish the execution of their set of operations, the benchmark is considered done and the execution time is presented. In an optional third stage we do a verification of the consistency of the structure. It starts by resetting the seeds again and then every thread does a search for every value provided by the PRNG, in order to verify if the values in the insertion and search ranges are present and the values in the remove range are missing. Figure 5.1 shows how the keys are distributed among the threads and the insert, remove and search operations.

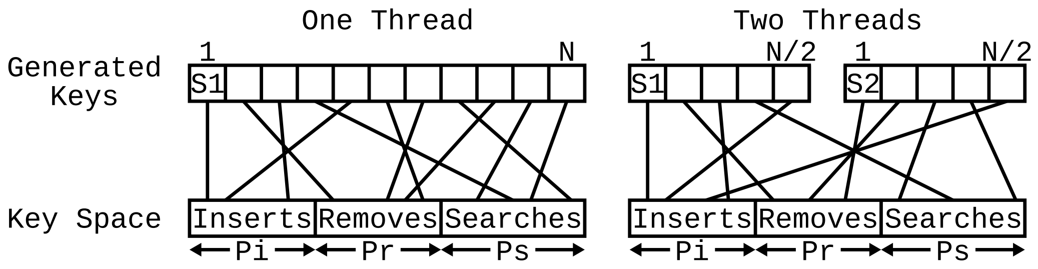

Figure 5.2: Representation of how the keys are divided among the threads and operations.

The PRNG used was the *nrand48_r()* function from the standard library, which uses the linear congruential algorithm that is given by Eq. 5.1 with a value a of 25214903917, a value c of 11 and a value m of 2^{48}. From the values generated the 32 higher order bits are extracted and used as described for the benchmark.

$$f(n+1) = (a \times f(n) + c) \mod m \tag{5.1}$$

The nature of the random generated values implies that the percentage of inserts, removes and searches may not be exactly the same as specified in the input parameters, but as the PRNG used has good properties and the number of operations is large enough, the actual deviation can be considered negligible[1].

For the experiments, we used several versions of the LFHT data structure, namely:

- *NF* (No Free): this version is the direct implementation of the original design [1, 2] where no memory is reclaimed.
- *OF* (optimistic free): this version takes an optimistic approach to reclaim memory. Each thread has a big enough reclamation ring buffer that it fills with the nodes being removed and reclaims their memory each time it goes around the buffer and refills it with new

[1]In the experimental results, we confirmed that the deviation was in fact negligible

removed nodes. This version is incorrect by definition but will serve as our base line as it represents the best possible performance achievable for memory reclamation.

- *GPE* (Grace Periods based on Eras) : this version implements a grace period method based on eras on top of our compatible implementation. The grace period method uses a global clock that is atomically incremented at every removal and a local clock that every thread updates to the global clock at a quiescent state, which is declared at every operation.
- *GPL* (Grace Periods based on Lamport clocks): this version also implements a grace period method on top of our compatible implementation, but it is based on the Lamport clocks method. At every quiescent state, each thread reads all the other threads' clocks and updates its own with the maximum value read plus one. Again, a quiescent state is declared at every operation.
- *HHL* (Hazard Hash and Level): this version is a prototype implementation of the memory reclamation method based on hazard hashes and levels. Due to lack of time, some of the features were not implemented and tested completely. It lacks the implementation of the forced expansions strategy and the hazard pointer traversal in the last level. However, we argue that the experiments will not suffer any meaningful impact without such support, as these features serve the main purpose of guaranteeing the memory bounds and allow extreme cases of key collision.

All these versions were tested in two configurations: (i) with a maximum chain size of 3, hash nodes with 2^4 bucket entries and a threshold for reclamation of 2^8 nodes, and (ii) with a maximum chain size of 3, hash nodes with 2^8 bucket entries and a threshold for reclamation of 2^8 nodes. In what follows, we name these configurations as *4* and *8* when both are compared in the same graph.

To put the results in perspective, we also used the concurrent hash maps data structure from the Intel Thread Building Blocks [17]library. We were not able to find any other reliable hash map data structures capable of running concurrent modification operations and able to reclaim memory outside a garbage collected environment.

All versions were executed with a hash function equivalent to the identity ($h(x) = x$) in order to minimize possible differentiating factors and reduce the overall overhead[2]. All experiments were run 5 times and the results presented are the average of such runs.

5.2 Results and Discussion

The environment for our experiments was a NUMA machine with two AMD Opteron Processor 6274 and 32GiB of ECC RAM. The memory allocator used was jemalloc [6] version 5.0 as it showed good results and was able to scale with all the cores used without generating contention

[2]Note that the keys are already random and the way the key space is divided already prevents interference.

in the kernel. Ideally, we would use a lock-free memory allocator, however, in the task at hand, eliminating noise caused by overhead is the main priority. The experiments shown next use a fixed size of 10^7 operations but with a varying percentage of insert, remove and search operations. In Appendix A we show the data for all the experiments done.

5.2.1 Baseline

We start by presenting, in Fig. 5.3 and Fig. 5.4, the comparison between the *NF* and the *OF* versions in the 2^4 bucket entries configuration (recall that *OF* will be used as baseline in future comparisons). We show the results in terms of throughput in order to better show the normal behavior of the data structure with a varying number of threads. The results in Fig. 5.3 only include remove operations, which somehow shows the overhead caused by the memory allocator in the *free()* procedure (the main difference between the two versions). The results in Fig. 5.4, include half insert and half remove operations. Both figures show almost linear scalability of the data structure, with a slight degradation from around the 16 threads mark onwards, which we believe could be caused by the NUMA characteristics of the hardware used. In these experiments, the overhead of the memory allocator becomes minimal, which is justified by the usage of thread local caches that modern memory allocators like jemalloc use. Local caches allow the memory allocator to require minimal synchronization between threads, as long as the amount of allocations and deallocations is identical and they are interleaved enough, which is the situation in our benchmarks.

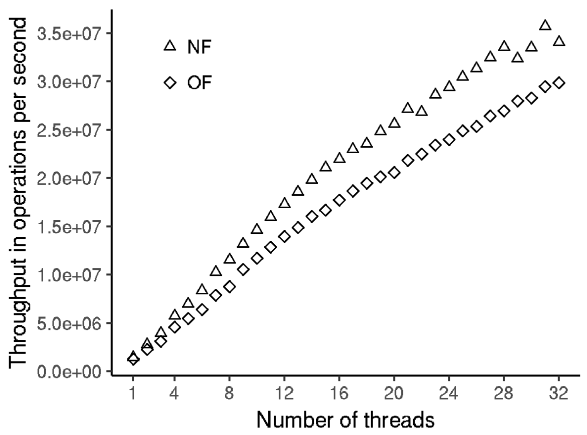

Figure 5.3: Throughput of NF and OF with 100% removes (higher is better).

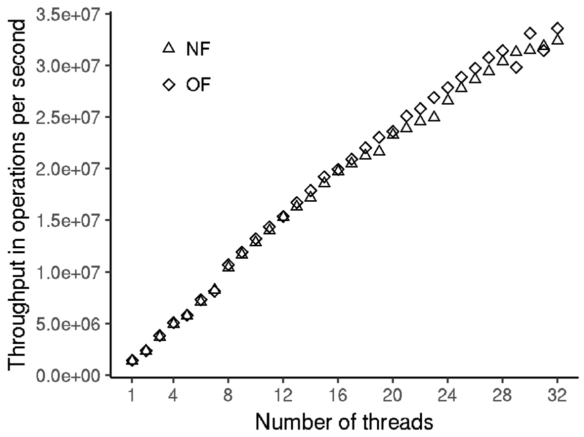

Figure 5.4: Throughput of NF and OF with 50% inserts and 50% removes (higher is better).

5.2.2 Memory Reclamation Impact

This subsection compares the *OF*, *GPE*, *GPL* and *HHL* versions with the 2^4 bucket entries configuration. To simplify the presentation, all results are normalized to the *OF* version as it better shows the overhead implied by each specific memory reclamation method. Figures 5.5 to

5.10 show results for different ratios of insert, remove and search operations.

With inserts and searches only (Fig. 5.5 and Fig. 5.6), the *GPE* version behaves very closely to ideal, as the global clock is never updated resulting in almost no synchronization for the memory reclamation done in practice. The same happens for the *HLL* version and, as no reclamation is done, the hazard pairs are never synchronized between threads. However, on average, *HHL* has one more atomic write per operation than *GPE*.

On the remaining results (Figs. 5.7–5.10) one can observe a heavy degradation on both grace period based methods. This can by explained by the synchronization required every time a quiescent state is declared, which happens at every operation. Declaring the quiescent states less often could improve the scalability of these methods, but would also increase the memory usage. One of the reasons why we have implemented these two grace periods based memory reclamation methods was the fact that they map exactly to the state-of-the-art drop the anchor and hazard eras methods. The drop the anchor method does the same thing as our *GPL* method for its clock management but adds the anchor maintenance and recovery procedures to be able to achieve a finite memory bound. Similarly, the Hazard Eras method does the same thing that our *GPE* method does for its clock management, but instead of doing the equivalent of a quiescent state at every operation, it does so at every node traversed in order to achieve a memory bound. As such, if either of these methods were fully implemented with the LFHT data structure, they would have, at best, a similar performance to the grace period methods we are showing. Moreover, we would expect that any method based on grace periods that requires, at least one quiescent state per operation, would not be competitive with our *HHL* method in workloads that require a non trivial amount of remove operations. As one can see, in Fig. 5.8, just 5% of insertions and 5% of removals is enough to more than double the execution time with 32 threads comparing the *HHL* version to the best *GP* method.

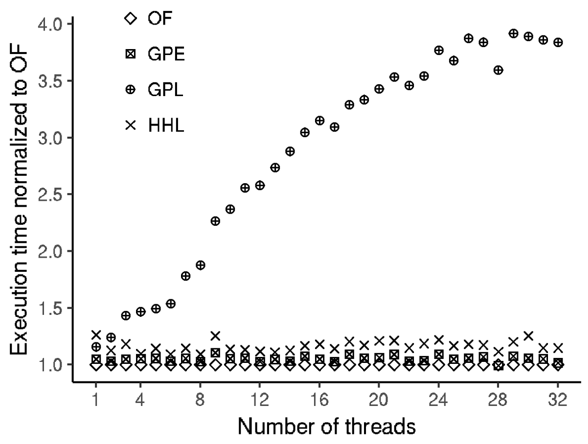

Figure 5.5: Execution time of OF, GPE, GPL and HHL normalized to OF with 100% inserts (lower is better).

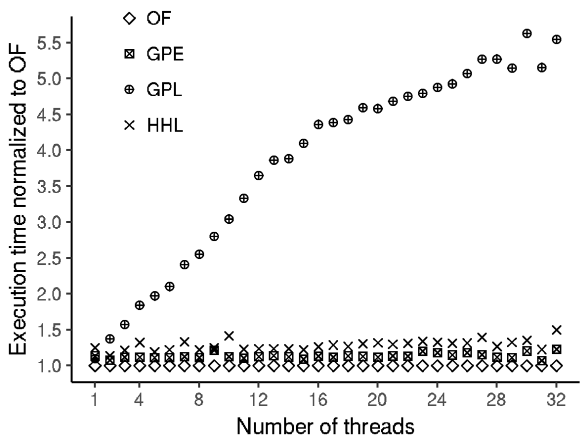

Figure 5.6: Execution time of OF, GPE, GPL and HHL normalized to OF with 100% searches (lower is better).

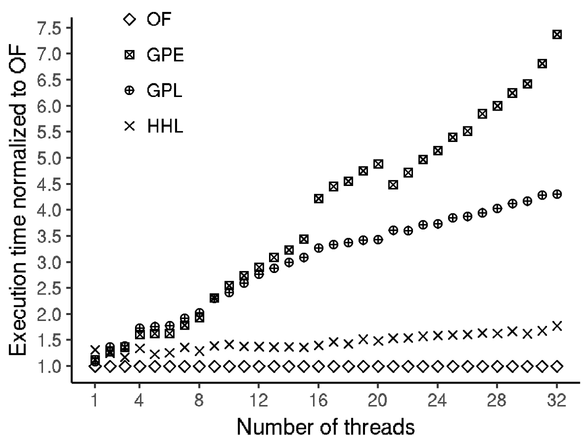

Figure 5.7: Execution time of OF, GPE, GPL and HHL normalized to OF with 100% removes (lower is better).

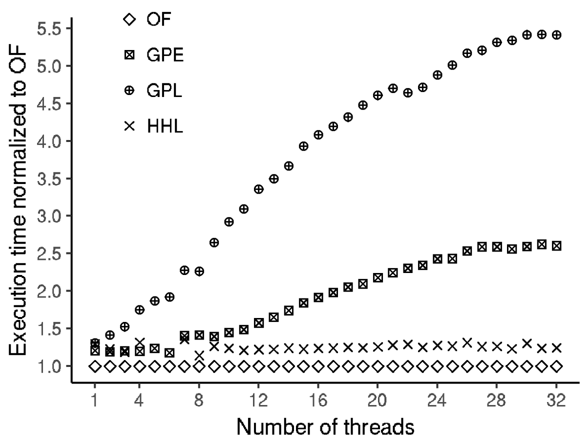

Figure 5.8: Execution time of OF, GPE, GPL and HHL normalized to OF with 5% inserts, 5% removes and 90% searches (lower is better).

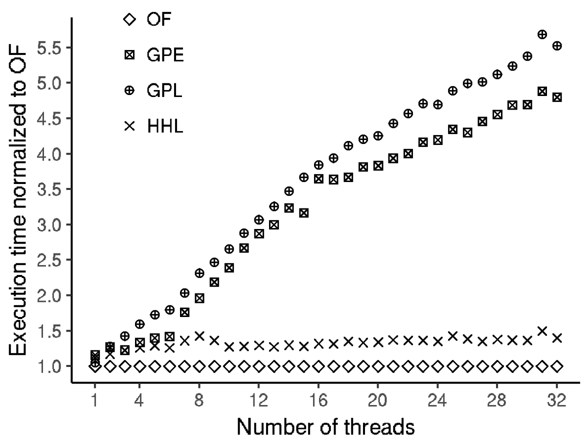

Figure 5.9: Execution time of OF, GPE, GPL and HHL normalized to OF with 25% inserts, 25% removes and 50% searches (lower is better).

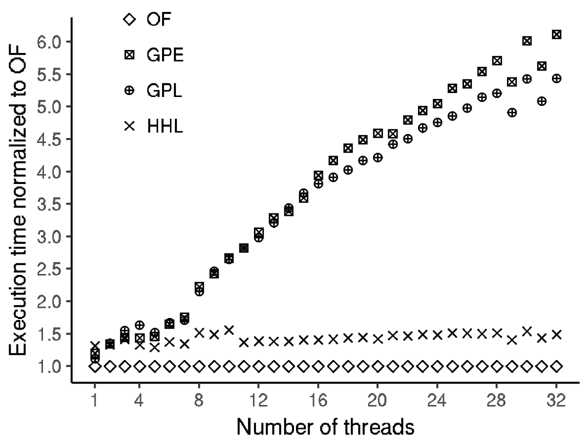

Figure 5.10: Execution time of OF, GPE, GPL and HHL normalized to OF with 50% inserts and 50% removes (lower is better).

5.2.3 Comparison with Lock Based Data Structures

Finally, we present the comparison between our lock-free *HLL* memory reclamation method and the concurrent hash maps from the Intel Thread Building Blocks library (*TBB*), which is lock based. These results are again shown in terms of throughput in order to better reveal the scalability of each data structure. As one can observe, in Fig. 5.11 and Fig. 5.12, LFHT with our *HHL* memory reclamation method is very competitive when compared against TBB concurrent hash maps in benchmarks with heavy search workloads. Even in the search only benchmark (Fig. 5.11), *HHL* is still better with the 2^8 bucket entries configuration, but this configuration may not be ideal in terms of memory overhead. Figures 5.13 through 5.18 show that the concurrent hash maps from the TBB library is incapable of dealing with more than around 5×10^6 modification operations (inserts and removes) per second independently of the number of threads used, while LFHT with our *HHL* memory reclamation method is able to scale almost linearly with any kind of operation, being able to produce about 5 times the throughput for a workload of only modification operations with 32 threads, as one can observe in Fig. 5.16. These results clearly show the impact of the value of a lock based approach compared to lock-free.

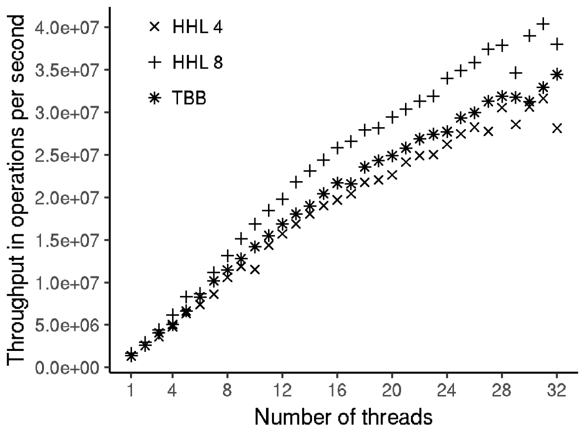

Figure 5.11: Throughput of HHL and TBB with 100% searches (higher is better).

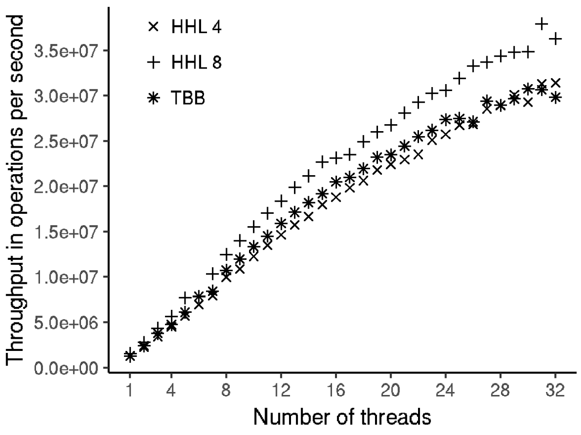

Figure 5.12: Throughput of HHL and TBB with 5% inserts, 5% removes and 90% searches (higher is better).

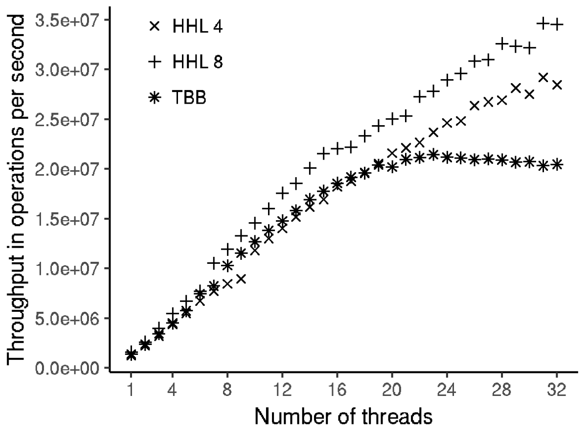

Figure 5.13: Throughput of HHL and TBB with 10% inserts, 10% removes and 80% searches (higher is better).

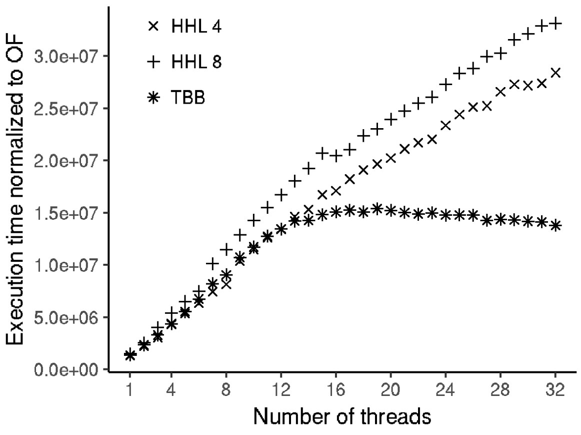

Figure 5.14: Throughput of HHL and TBB with 15% inserts, 15% removes and 70% searches (higher is better).

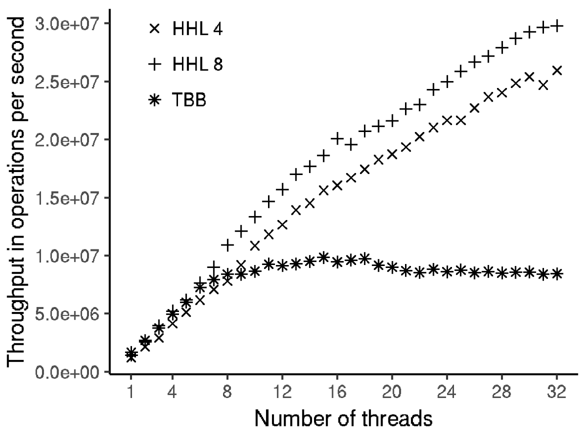

Figure 5.15: Throughput of HHL and TBB with 25% inserts, 25% removes and 50% searches (higher is better).

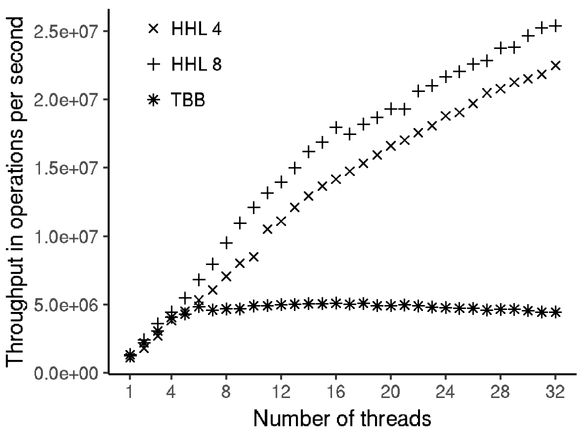

Figure 5.16: Throughput of HHL and TBB with 50% inserts and 50% removes (higher is better).

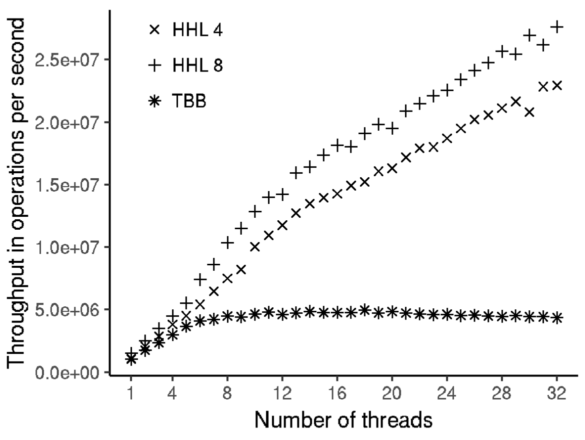

Figure 5.17: Throughput of HHL and TBB with 100% inserts (higher is better).

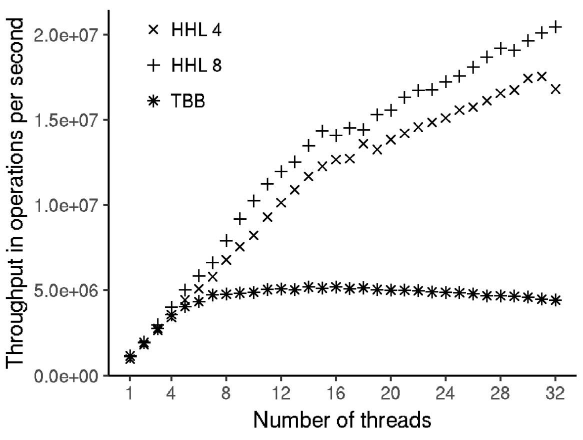

Figure 5.18: Throughput of HHL and TBB with 100% removes (higher is better).

Chapter 6

Conclusion

This chapter summarizes the work and contributions of the thesis and proposes some directions for further work. The thesis started with the goal of reclaiming memory from the original LFHT data structure, as proposed by Areias and Rocha [1, 2]. To achieve this goal, we started by studying the current state-of-the-art memory reclamation methods and the data structure itself. After studying both, we found that they were incompatible due to the delegation process of remove operations inherent to LFHT, which may leave a removed node reachable, and to the common assumption by all state-of-the-art memory reclamation methods that a node is unreachable at the end of a remove operation.

Our first attempt to solve the problem, and make the data structure compatible with the current state-of-the-art memory reclamation methods, was to eliminate the delegation process from the LFHT data structure. We were able to implement a working method to do so, but after further revision it was determined that it was affected by an ABA problem. Since such problem was not reproducible in real experiments, we took some time in detecting it and in finding a valid alternative.

As a consequence, we decided instead to create a novel dedicated memory reclamation method to the LFHT data structure that would not only draw from the knowledge gained from studying the state-of-the-art methods but would exploit the characteristics of the data structure to achieve the best possible performance and memory bounds. We accomplished this goal by creating a memory reclamation method based on the idea of using a path and a level of the structure to protect a specific and well-defined region, which we named hazard hash and level (HHL) method. This method is able to achieve good performance while providing low and well-defined memory bounds.

By comparing the HHL memory reclamation method with two grace period implementations built on top of our first attempt, we were able to show that those grace periods based methods that require at least one quiescent state per operation, are not able to achieve the performance of the HHL method. This includes modern methods like drop the anchor by Braginsky et al. [3] and hazard eras by Ramalhete and Correia [16]. Our experiments seem to indicate that this kind of grace period based methods, which are known to be among the most performant methods, are probably not viable in terms of performance for modern high throughput concurrent data structures, like the LFHT.

We also compared the LFHT data structure with the HHL memory reclamation method against the only reliable implementation of a concurrent hash map that does not need a garbage collector for memory reclamation. This implementation is the concurrent hash maps from the Intel TBB library that has a lock based approach. Our experiments showed that our approach of the LFHT data structure with the HHL memory reclamation method not only has the advantage of being lock-free but is also competitive in heavy read workloads, showing better scalability in all other workload scenarios.

We hope that the work in this thesis will be a basis to further improvements and research in this area. We next suggest some topics for further work.

More experimentation. For future work we could make a deeper analysis on the practical memory usage and requirements of the LFHT data structure and find further optimizations that can be done to the structure or the reclamation method.

Memory reclamation of hash nodes. Further work could be done on the developed approach in order to allow it to remove and reclaim hash nodes which would likely grant a great reduction in memory usage.

Ading features. Common features to modern data structures could be added to our prototype in order to make it a competitive alternative in a production environment.

Extension to similar data structures. Further study could also be done in applying the HHL memory reclamation method to other similar tree data structures as it showed very promising results with the LFHT data structure.

This work developed during this thesis resulted in a publication in a national event [15].

Appendix A

Tables

In this appendix we show the execution time in seconds for every configuration tested. The execution times shown are the average of five runs.

Table A.1: 10^6 operations, 2^4 bucket entries, 100% inserts, 0% removes and 0% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.4738974	0.4486556	0.515284	0.5161362	0.590777	0.832854
2	0.27976	0.2861502	0.3053098	0.4067292	0.3511796	0.4927194
3	0.2020094	0.2032052	0.220095	0.3287516	0.2483364	0.366295
4	0.1600376	0.1585484	0.1719914	0.2788804	0.1926818	0.2948452
5	0.1313622	0.1307392	0.1412954	0.249618	0.158021	0.2525986
6	0.1130836	0.1127396	0.1222284	0.2253444	0.1377766	0.224962
7	0.0983884	0.0986598	0.1074146	0.2095294	0.1201454	0.2129166
8	0.089914	0.0879456	0.097686	0.2004848	0.1044218	0.205574
9	0.080603	0.0796718	0.0873032	0.1950864	0.100016	0.2009218
10	0.075425	0.0745222	0.0808556	0.1823582	0.0901786	0.2066424
11	0.0718214	0.0706136	0.0758186	0.1861306	0.0875594	0.2063814
12	0.064152	0.0648146	0.070289	0.1804466	0.082883	0.203865
13	0.0611018	0.0619902	0.0686924	0.1694062	0.0770108	0.2048602
14	0.0615542	0.0594512	0.063671	0.1623952	0.0731406	0.202378
15	0.0580726	0.0557514	0.0650376	0.1698162	0.0685074	0.2043542
16	0.0526856	0.0538344	0.0579638	0.1656494	0.0672624	0.2011576
17	0.0531536	0.051651	0.0571672	0.1627282	0.0661224	0.208966
18	0.0512614	0.0514752	0.0546624	0.159009	0.0655014	0.2046524
19	0.0490136	0.0489362	0.0519874	0.1529112	0.0594168	0.2023552
20	0.0472942	0.048744	0.0583648	0.1545104	0.0600858	0.2007668
21	0.0529168	0.0495778	0.055006	0.1499692	0.055473	0.207839
22	0.0471882	0.047015	0.0486738	0.1537166	0.0573132	0.2023664
23	0.046452	0.0496118	0.0513006	0.1463874	0.0597466	0.2121548
24	0.0517568	0.0450726	0.0506888	0.1458754	0.0543788	0.2109314
25	0.0485688	0.049643	0.0521128	0.1492082	0.051493	0.2171484
26	0.0440634	0.0499108	0.0475602	0.1456894	0.0563958	0.2206604
27	0.0514674	0.0459644	0.0515002	0.143496	0.0532636	0.2198694
28	0.0475368	0.0513232	0.0537366	0.144887	0.0623142	0.2194724
29	0.0465048	0.0458334	0.05154	0.1433084	0.0558976	0.2243618
30	0.0555616	0.0513888	0.0565296	0.1429142	0.0567302	0.2215868
31	0.0546824	0.0490518	0.060575	0.1423012	0.062192	0.2248406
32	0.0535846	0.0537544	0.0544314	0.1430638	0.061888	0.2279118

Table A.2: 10^6 operations, 2^4 bucket entries, 0% inserts, 100% removes and 0% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.382505	0.548311	0.610269	0.5386682	0.7188426	0.6713892
2	0.2228764	0.304937	0.4254838	0.4538056	0.407142	0.4100402
3	0.153998	0.2106962	0.3510188	0.3790104	0.283573	0.302919
4	0.120683	0.1618132	0.3128856	0.345947	0.2200936	0.2423038
5	0.0993686	0.1315056	0.2576902	0.2806744	0.1830682	0.2379318
6	0.0846334	0.114036	0.236251	0.2411822	0.1547384	0.2200158
7	0.0732362	0.0963426	0.2190512	0.2252286	0.1357772	0.2099412
8	0.0650372	0.0857522	0.2138874	0.2261096	0.1207668	0.2036324
9	0.0585118	0.0770986	0.2060422	0.2149752	0.1115202	0.2019124
10	0.053957	0.0718026	0.2000326	0.2009758	0.1063614	0.1975216
11	0.049662	0.0653728	0.1958626	0.1939544	0.1027444	0.1943572
12	0.0467676	0.0613322	0.1906346	0.1918396	0.0925532	0.1941342
13	0.0437262	0.056422	0.19693	0.1814292	0.087294	0.1919418
14	0.0405382	0.0517312	0.191919	0.182352	0.0827792	0.1947664
15	0.0391108	0.051228	0.190917	0.1785756	0.084582	0.1942088
16	0.0414266	0.0493242	0.2383776	0.1751642	0.0779904	0.1910746
17	0.0363234	0.0470614	0.2396728	0.1726278	0.0750744	0.1925208
18	0.0334662	0.0447338	0.2369058	0.174238	0.0741888	0.1959584
19	0.0333998	0.0435596	0.2374612	0.1716612	0.0703238	0.2020424
20	0.0354496	0.041676	0.2337734	0.1619322	0.068621	0.1988042
21	0.0325138	0.039754	0.1960838	0.1649442	0.0669028	0.2024918
22	0.0317504	0.0428214	0.1978804	0.1558276	0.0661172	0.1966988
23	0.0326562	0.0375276	0.2025214	0.156665	0.0672482	0.2011392
24	0.0321882	0.036912	0.2025942	0.154909	0.0647938	0.2070122
25	0.0310394	0.036964	0.2129104	0.1520424	0.0666626	0.2009812
26	0.0286792	0.0397186	0.2059236	0.1502788	0.0663832	0.206756
27	0.0333098	0.0343472	0.2088246	0.149969	0.0601982	0.1982072
28	0.0273732	0.0371814	0.2097612	0.146309	0.0644718	0.203428
29	0.027137	0.0388168	0.2094264	0.1548452	0.0570692	0.2056164
30	0.0302326	0.032518	0.2139214	0.1498402	0.0657106	0.2073898
31	0.0297892	0.041162	0.220639	0.1509242	0.0703846	0.2108364
32	0.0313052	0.0367638	0.2339144	0.151806	0.0663718	0.2076626

Table A.3: 10^6 operations, 2^4 bucket entries, 0% inserts, 0% removes and 100% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.306464	0.294602	0.419657	0.3776086	0.4481186	0.4404996
2	0.175625	0.1742862	0.2108096	0.3279684	0.2482396	0.2584118
3	0.116561	0.1170844	0.1422822	0.2622914	0.1660156	0.178313
4	0.0878036	0.0880212	0.1074814	0.230902	0.1264198	0.1371504
5	0.071759	0.0712156	0.0883458	0.209709	0.1019884	0.1113862
6	0.0603624	0.0619932	0.0738998	0.1969438	0.0859986	0.094915
7	0.0537368	0.0519772	0.0643346	0.1867838	0.074148	0.0820576
8	0.0471268	0.0467098	0.0560562	0.1828364	0.0658238	0.073619
9	0.0420938	0.0414374	0.0519824	0.1766392	0.0597488	0.0661494
10	0.038416	0.0386404	0.0465592	0.175924	0.0530894	0.0610006
11	0.0356334	0.0358282	0.0423376	0.1706972	0.0524896	0.0556888
12	0.0322934	0.0323508	0.0408146	0.1726212	0.0462652	0.0528194
13	0.0312272	0.0309212	0.0380108	0.1660646	0.04547	0.0498456
14	0.0288284	0.0306914	0.0359636	0.1661764	0.0412714	0.0469
15	0.0272136	0.0293026	0.032643	0.1624638	0.040318	0.044757
16	0.029462	0.0264416	0.0324444	0.1626308	0.0418578	0.0487978
17	0.02473	0.0246756	0.0326708	0.1606282	0.0367866	0.0418924
18	0.0245764	0.0262058	0.0306584	0.1518324	0.0374186	0.0385802
19	0.0252674	0.0232266	0.030278	0.1511014	0.0355408	0.0359094
20	0.0250274	0.0244418	0.0279668	0.1460804	0.0348164	0.0370916
21	0.0240252	0.0247716	0.0276192	0.1437702	0.0365674	0.0390338
22	0.0222694	0.0210948	0.029259	0.1439028	0.0319754	0.0378044
23	0.0242714	0.0224234	0.0276888	0.1407292	0.0343186	0.040462
24	0.0199678	0.0247882	0.028028	0.137897	0.0339896	0.0353256
25	0.0203242	0.02456	0.0247286	0.1347732	0.0323798	0.0362436
26	0.0236066	0.0238904	0.0271016	0.1333324	0.0307278	0.0340918
27	0.0252742	0.0213234	0.0287898	0.1316834	0.0342748	0.0394166
28	0.0206206	0.0228018	0.0299014	0.130626	0.0286082	0.0332864
29	0.0231164	0.024864	0.0262878	0.1280216	0.0350204	0.0335118
30	0.0231954	0.0221082	0.0286696	0.1252166	0.0313472	0.0347814
31	0.0232966	0.0242412	0.0255942	0.126959	0.03092	0.031302
32	0.0230528	0.0237264	0.028138	0.1246498	0.0320526	0.0358696

Table A.4: 10^6 operations, 2^4 bucket entries, 50% inserts, 50% removes and 0% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.4487758	0.420862	0.5566154	0.512601	0.632776	0.5994604
2	0.2493062	0.2434982	0.3640998	0.3874542	0.3599168	0.3832336
3	0.172839	0.1668322	0.2742538	0.2999512	0.2500902	0.287331
4	0.1331958	0.1301614	0.2318754	0.2545842	0.1936844	0.2367216
5	0.110403	0.1080088	0.2049648	0.2245666	0.1601572	0.225169
6	0.0937304	0.091242	0.1940196	0.2075372	0.1352218	0.213578
7	0.0817998	0.0788768	0.1886992	0.1940726	0.1171434	0.2066438
8	0.072618	0.071307	0.1870984	0.191436	0.1055646	0.2068536
9	0.066557	0.0634846	0.180589	0.188418	0.0972748	0.2036736
10	0.0595308	0.059187	0.181974	0.1775906	0.0878182	0.2041424
11	0.0561052	0.054754	0.1811314	0.1831334	0.083449	0.1918108
12	0.0529536	0.0516038	0.181276	0.169936	0.0800442	0.196648
13	0.0491856	0.046965	0.1763296	0.177694	0.0725138	0.1982822
14	0.0456912	0.0452336	0.1745666	0.183154	0.0695532	0.1956646
15	0.0442516	0.0428766	0.1701862	0.179197	0.0681196	0.1967674
16	0.0424604	0.0402698	0.1922818	0.1675706	0.064792	0.1960966
17	0.0407912	0.0414726	0.1867238	0.1706384	0.0609696	0.1970666
18	0.0399156	0.0395542	0.1849746	0.1685492	0.0627654	0.1939006
19	0.038079	0.0367936	0.18427	0.1672744	0.0574986	0.199899
20	0.036295	0.0384906	0.18293	0.1624516	0.0584254	0.1991862
21	0.0353912	0.0377392	0.1737554	0.1646704	0.0563378	0.1963306
22	0.0331728	0.0333594	0.1732406	0.1589648	0.0533598	0.2015892
23	0.03338	0.0376212	0.1726282	0.1590962	0.051033	0.2071408
24	0.0334242	0.0333608	0.1737758	0.158125	0.050352	0.2035772
25	0.0338346	0.0331142	0.1718904	0.156975	0.049736	0.2068292
26	0.0345434	0.0337962	0.1695288	0.1576484	0.0530282	0.2066548
27	0.0328034	0.0320004	0.16865	0.152744	0.0486622	0.2023506
28	0.031924	0.0335968	0.171371	0.1555306	0.0504028	0.2070804
29	0.032895	0.0302786	0.172987	0.1492512	0.0541108	0.2094464
30	0.031637	0.0320178	0.1695376	0.1488308	0.0520058	0.2125064
31	0.031802	0.03239	0.170128	0.1536314	0.0517562	0.211919
32	0.0370128	0.0359178	0.1773006	0.1458862	0.0562942	0.2141714

Table A.5: 10^6 operations, 2^4 bucket entries, 5% inserts, 5% removes and 90% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.3296798	0.3830454	0.4401374	0.4494676	0.4687254	0.4843558
2	0.1883376	0.1918534	0.247344	0.342479	0.2695492	0.2727634
3	0.1289606	0.1294334	0.1699452	0.2694778	0.1805658	0.1884106
4	0.0967412	0.0983852	0.132224	0.2323728	0.1378066	0.1448556
5	0.07854	0.0788898	0.1124622	0.2087992	0.1121554	0.119502
6	0.066931	0.0672178	0.098738	0.1941004	0.0935802	0.1023092
7	0.0588584	0.0586096	0.0892908	0.1849336	0.082811	0.0880734
8	0.052218	0.0522	0.0830408	0.1807828	0.0719134	0.0784824
9	0.0473642	0.0471678	0.079727	0.1789886	0.0661526	0.0723246
10	0.0425746	0.0424254	0.0782992	0.1782982	0.0601584	0.06615
11	0.0393016	0.0387164	0.0749246	0.1797062	0.054908	0.0609974
12	0.036677	0.037162	0.0737704	0.165962	0.0515556	0.0564958
13	0.0335534	0.0347282	0.07287	0.1686442	0.0489338	0.0534584
14	0.0350518	0.0331354	0.072278	0.1620398	0.0487278	0.0495876
15	0.0296666	0.0310092	0.0703686	0.163229	0.0434472	0.0478484
16	0.0304804	0.0302032	0.0708296	0.1634666	0.0419182	0.0445832
17	0.027788	0.030359	0.0703512	0.1620482	0.0415688	0.044514
18	0.0268514	0.0288442	0.0705468	0.154029	0.040002	0.0452156
19	0.0268368	0.029772	0.0694026	0.1582504	0.0392178	0.0434196
20	0.032347	0.0264724	0.0693754	0.1538816	0.0373052	0.0410692
21	0.024126	0.0288454	0.0691108	0.149121	0.0373954	0.0431992
22	0.0270122	0.0264826	0.0674252	0.149015	0.0359222	0.0411042
23	0.0265806	0.0237054	0.0670766	0.145916	0.0386376	0.039239
24	0.0229552	0.0264638	0.0666054	0.1431098	0.0336332	0.0400892
25	0.0260794	0.0301124	0.0671046	0.1404214	0.0380094	0.0377668
26	0.0269654	0.0238014	0.0650712	0.137237	0.0384522	0.0385632
27	0.027163	0.022991	0.065443	0.1353282	0.0324972	0.042442
28	0.0253324	0.023853	0.0658062	0.1353164	0.0394324	0.0376254
29	0.0241228	0.025178	0.0688286	0.1330138	0.0369766	0.0384806
30	0.025675	0.0275644	0.0644386	0.133765	0.0357492	0.0418346
31	0.0237118	0.0261194	0.0649322	0.1309298	0.0376666	0.0377162
32	0.0271912	0.0281008	0.0654722	0.1286136	0.033962	0.0402214

Table A.6: 10^6 operations, 2^4 bucket entries, 10% inserts, 10% removes and 80% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.3675044	0.3340276	0.44693	0.4655312	0.5223158	0.5276
2	0.203011	0.2070302	0.271928	0.3445826	0.2842208	0.2916758
3	0.1364474	0.1387186	0.1876704	0.2703304	0.1916034	0.200147
4	0.1031282	0.1034038	0.149546	0.2348462	0.1450958	0.1540032
5	0.0840178	0.086128	0.1268774	0.2117066	0.1202042	0.125236
6	0.0708194	0.0719838	0.1137972	0.1982378	0.0999388	0.1076214
7	0.0610142	0.0629874	0.1067456	0.1868356	0.0876534	0.0948036
8	0.0539026	0.0564464	0.1025938	0.1836692	0.0769764	0.084185
9	0.0490824	0.0497868	0.100097	0.1821584	0.0708432	0.077347
10	0.0448002	0.045479	0.0997496	0.1733886	0.0629112	0.0708688
11	0.0422706	0.0432554	0.098952	0.1814306	0.0581682	0.0667482
12	0.0375842	0.0394034	0.0973558	0.1773466	0.056666	0.0631278
13	0.0373038	0.0369332	0.0962916	0.1733378	0.0514714	0.0600524
14	0.0338692	0.0351832	0.0949056	0.1704932	0.0487922	0.0578044
15	0.0328626	0.0327546	0.0969838	0.1678558	0.0489506	0.0544678
16	0.030105	0.033042	0.0954746	0.1690818	0.0457626	0.0528172
17	0.0297578	0.0294114	0.096297	0.1646738	0.0436802	0.0519766
18	0.0279204	0.029486	0.09564	0.1633556	0.0410894	0.052157
19	0.028451	0.0290042	0.094471	0.155422	0.0419138	0.0511034
20	0.026773	0.0296152	0.096644	0.1561952	0.0437484	0.0505018
21	0.0278114	0.0278582	0.0925248	0.148129	0.040341	0.0532624
22	0.0255438	0.0269426	0.0928146	0.1521884	0.0378718	0.0533682
23	0.0306202	0.0252146	0.0917894	0.1494274	0.0370716	0.0534528
24	0.0260058	0.0281284	0.0901694	0.146545	0.0335742	0.051859
25	0.0231804	0.0309744	0.0913588	0.1435452	0.0355914	0.0522182
26	0.0252662	0.0272786	0.0900478	0.1411006	0.032808	0.05086
27	0.028313	0.0289116	0.0898372	0.140543	0.0382668	0.051707
28	0.026642	0.0285358	0.087253	0.1386078	0.0381168	0.0525682
29	0.02859	0.0274668	0.087364	0.1367844	0.0352078	0.050824
30	0.0289506	0.0255324	0.087623	0.1345026	0.0341974	0.0529818
31	0.029934	0.033283	0.0860072	0.1336098	0.0377298	0.0533272
32	0.0318492	0.0290436	0.0868376	0.1330308	0.0411074	0.0526452

Table A.7: 10^6 operations, 2^4 bucket entries, 15% inserts, 15% removes and 70% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.4030936	0.384907	0.5034134	0.4962936	0.5657092	0.508507
2	0.210892	0.2149394	0.2855928	0.3541092	0.2963078	0.304138
3	0.1425484	0.1429534	0.201344	0.273939	0.2017718	0.2097976
4	0.1097022	0.10975	0.1625974	0.236128	0.1528814	0.1640498
5	0.0869374	0.0904298	0.1408746	0.2131782	0.1264254	0.1344666
6	0.0752176	0.076801	0.1278284	0.2034792	0.1061072	0.11545
7	0.0642328	0.065646	0.1233224	0.19152	0.0954502	0.101298
8	0.0579642	0.0582656	0.1177182	0.184192	0.081586	0.0935718
9	0.0510662	0.0521014	0.1162858	0.1837172	0.0744796	0.0845976
10	0.0470408	0.0491934	0.1179196	0.183588	0.0690496	0.080502
11	0.0430254	0.0448936	0.1144774	0.1765618	0.061448	0.07528
12	0.0414522	0.0425528	0.117733	0.1813516	0.0605758	0.0737774
13	0.0379908	0.0387534	0.1168222	0.1747146	0.0552796	0.0718832
14	0.0369768	0.0379608	0.1114964	0.1743734	0.0501578	0.0691198
15	0.0346362	0.0355992	0.1116114	0.1737484	0.0513962	0.0694338
16	0.0326904	0.0360244	0.1127432	0.1782018	0.0482666	0.0691878
17	0.0310632	0.0326526	0.1141956	0.1617808	0.0464146	0.068692
18	0.0340784	0.0317172	0.113403	0.1626726	0.0444724	0.0681496
19	0.0295172	0.030774	0.1135038	0.1551286	0.0453928	0.0695104
20	0.0294636	0.0301664	0.1121858	0.1585694	0.046392	0.0708184
21	0.0269844	0.034389	0.1095792	0.1566564	0.0411642	0.0699532
22	0.0303596	0.0286932	0.107292	0.1538152	0.041034	0.0703594
23	0.0263156	0.0277838	0.1082986	0.1506576	0.0440564	0.0700916
24	0.0324594	0.0310452	0.1080002	0.1501034	0.0407424	0.0729366
25	0.030635	0.0319956	0.1060434	0.1465708	0.0388284	0.071191
26	0.0269406	0.0278838	0.1055888	0.1447046	0.0392162	0.0704882
27	0.0319758	0.024886	0.1091898	0.1430446	0.037761	0.0714196
28	0.0259842	0.0248608	0.1032528	0.1420272	0.0380712	0.0707272
29	0.0296196	0.0260216	0.1040368	0.1398146	0.0359866	0.07153
30	0.0322806	0.0291456	0.1035104	0.1406506	0.037155	0.0725642
31	0.030539	0.034074	0.1073352	0.1362262	0.0400236	0.0723948
32	0.0284438	0.0355752	0.1014494	0.1404762	0.0402674	0.0729414

Table A.8: 10^6 operations, 2^4 bucket entries, 25% inserts, 25% removes and 50% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.3869272	0.3925338	0.5165832	0.4580332	0.5509448	0.5659378
2	0.2269	0.230245	0.3145204	0.3620438	0.3159762	0.3286644
3	0.1528662	0.1553118	0.2253506	0.2805248	0.2171186	0.2346246
4	0.117624	0.1175366	0.1826846	0.237292	0.1670106	0.1860698
5	0.0973076	0.097364	0.1607546	0.2148036	0.1358742	0.1587232
6	0.0813058	0.0829774	0.1493166	0.2001986	0.1157948	0.1412396
7	0.070413	0.0710742	0.1469814	0.1903828	0.0994556	0.1281638
8	0.0625866	0.0623594	0.1442758	0.1873844	0.0903876	0.1188894
9	0.0561276	0.0575994	0.1447632	0.1849586	0.0804844	0.1143058
10	0.0512682	0.0521966	0.142728	0.1838042	0.0742214	0.1084386
11	0.046879	0.048799	0.140482	0.1819104	0.0695614	0.1083534
12	0.0444664	0.0458438	0.1426528	0.1709128	0.0628816	0.1066118
13	0.0414404	0.041529	0.140418	0.1783092	0.0622028	0.104972
14	0.0390298	0.0391414	0.1396128	0.1700362	0.0573902	0.1055902
15	0.0390202	0.038265	0.137573	0.1739928	0.0579362	0.1054548
16	0.036809	0.0366448	0.1444204	0.1702982	0.0578508	0.1039564
17	0.038133	0.0349948	0.1418038	0.1653014	0.0526232	0.105588
18	0.0328682	0.0343916	0.1406924	0.1668044	0.0493758	0.1061424
19	0.0330102	0.0348044	0.1388496	0.1604956	0.050114	0.1077076
20	0.0318446	0.031546	0.1388978	0.1571	0.046422	0.1073704
21	0.0296892	0.0311694	0.1343344	0.1593842	0.0480874	0.1078732
22	0.0336316	0.030187	0.135872	0.1567516	0.0460634	0.1073268
23	0.0288792	0.0288626	0.1361688	0.1556868	0.0434118	0.1097316
24	0.0275228	0.0310552	0.132485	0.153694	0.0428932	0.1089148
25	0.0312258	0.0354884	0.1335836	0.151401	0.0414386	0.1107822
26	0.0291574	0.0284816	0.1314042	0.1515942	0.0450116	0.11057
27	0.0302794	0.0294336	0.130566	0.1462564	0.0442514	0.1104188
28	0.031268	0.0315014	0.1337984	0.145504	0.043455	0.1132992
29	0.0338422	0.0291276	0.1269608	0.1431704	0.0432596	0.1141138
30	0.0304082	0.0304478	0.13107	0.1451976	0.0446004	0.111359
31	0.0324356	0.0298542	0.128384	0.1439738	0.0423216	0.1135402
32	0.0284236	0.0347798	0.1290764	0.1420776	0.0538066	0.1164374

Table A.9: 10^6 operations, 2^4 bucket entries, 40% inserts, 40% removes and 20% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.4342622	0.4325958	0.5147644	0.506824	0.609691	0.6351868
2	0.2411372	0.2401818	0.3458406	0.3776108	0.3463486	0.3806386
3	0.1658816	0.164766	0.2541822	0.286879	0.237363	0.2772358
4	0.1269454	0.1272514	0.2109982	0.2447698	0.1832314	0.2286758
5	0.1050266	0.1040096	0.1882764	0.2166466	0.150281	0.1982936
6	0.0891824	0.0883272	0.1775	0.2011778	0.1271394	0.1804248
7	0.077706	0.0764922	0.1734866	0.1936876	0.1147486	0.1749708
8	0.0698024	0.0678262	0.1736022	0.1826472	0.1013126	0.1641678
9	0.061257	0.0608758	0.170344	0.1866234	0.092102	0.1671936
10	0.0573248	0.056574	0.1719404	0.188489	0.0830778	0.1643722
11	0.052355	0.0512006	0.16974	0.1803848	0.0799756	0.1640218
12	0.0493212	0.0479916	0.1690976	0.183572	0.0738784	0.1612024
13	0.0455044	0.0451186	0.1625492	0.1738764	0.0735676	0.1586986
14	0.0442506	0.0429922	0.1637874	0.1771324	0.0662128	0.1579268
15	0.0412022	0.0417416	0.1650952	0.1757828	0.0629522	0.1545302
16	0.038926	0.0390528	0.172256	0.1664178	0.0614446	0.1600402
17	0.0381764	0.0380868	0.168496	0.1668982	0.0598104	0.1543378
18	0.0360842	0.0364766	0.1667952	0.16717	0.0568278	0.1596866
19	0.0356768	0.0373738	0.1696198	0.1639858	0.0548702	0.1644768
20	0.0336472	0.0340088	0.168322	0.1642838	0.0557492	0.1591206
21	0.0369226	0.032798	0.1598978	0.164379	0.0577502	0.1635916
22	0.0307104	0.0359306	0.1592234	0.1604054	0.049276	0.163325
23	0.0315214	0.0333874	0.1585232	0.1580098	0.0507606	0.1661676
24	0.0344544	0.0284858	0.1583526	0.1563498	0.0464082	0.1646404
25	0.0345368	0.0324884	0.1602148	0.1585262	0.0526324	0.1718028
26	0.034245	0.0337474	0.1569976	0.1528482	0.04638	0.165025
27	0.0340414	0.0286068	0.157723	0.1525912	0.0483308	0.1683328
28	0.032592	0.0338346	0.1540496	0.1509968	0.0476832	0.1722578
29	0.0353542	0.039267	0.1553828	0.1483702	0.0474412	0.1726868
30	0.0327834	0.0296134	0.1519322	0.1483444	0.0479046	0.175919
31	0.0353366	0.03405	0.156025	0.1487726	0.05001	0.1782274
32	0.0366216	0.033727	0.1576122	0.1561926	0.05147	0.1751

Table A.10: 10^7 operations, 2^4 bucket entries, 100% inserts, 0% removes and 0% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	7.8294242	7.2378316	7.591064	8.3671688	9.14943	9.530383
2	4.1904468	4.305692	4.4426366	5.335624	4.847277	5.6668086
3	2.9522742	2.9473834	3.0913862	4.2237176	3.4850108	4.216107
4	2.3996278	2.3932628	2.5187704	3.5078794	2.6220828	3.3333674
5	1.9204404	1.9351874	2.0436534	2.8919374	2.2120418	2.7249544
6	1.6870686	1.6846756	1.7400948	2.5876426	1.839812	2.432537
7	1.3512366	1.3513394	1.426069	2.4074456	1.5461354	2.3572212
8	1.2117618	1.2161886	1.249893	2.2811428	1.329116	2.226948
9	0.967872	0.9695442	1.072364	2.1961586	1.216805	2.254451
10	0.87952	0.8776796	0.9255362	2.0802384	0.9969138	2.148221
11	0.8078048	0.8074712	0.8527666	2.0645352	0.9137838	2.0809474
12	0.7538664	0.7612896	0.782664	1.9649608	0.8496182	2.163218
13	0.7145206	0.7072072	0.7403326	1.9345698	0.785773	2.1019218
14	0.6546904	0.659155	0.6799648	1.897447	0.742005	2.0564954
15	0.6161976	0.6140502	0.660356	1.8715804	0.7163284	2.0877114
16	0.599563	0.5932394	0.6215128	1.8688658	0.69988	2.088803
17	0.5710592	0.5864262	0.602426	1.8135998	0.6694988	2.0894508
18	0.5603458	0.5456344	0.59566	1.7957092	0.6568806	2.0063398
19	0.5246614	0.5304342	0.561472	1.7693924	0.6217766	2.101988
20	0.5051266	0.5059714	0.5375088	1.7346674	0.612539	2.0639748
21	0.4887656	0.4798568	0.5229318	1.6963234	0.5818242	2.1111432
22	0.4679624	0.485854	0.5010874	1.6809902	0.5575446	2.1413642
23	0.4524	0.4675852	0.4844466	1.656726	0.5547144	2.1688352
24	0.4440474	0.4376392	0.4778688	1.6501026	0.5338852	2.156899
25	0.4504032	0.439456	0.4605648	1.6171292	0.5126036	2.2053844
26	0.4370866	0.418806	0.4423756	1.6234132	0.4946314	2.1901296
27	0.409942	0.4145312	0.44273	1.5924648	0.4862164	2.2216012
28	0.405788	0.424677	0.4211124	1.527472	0.473464	2.241313
29	0.3881334	0.3839864	0.4123774	1.504642	0.461528	2.2073292
30	0.3991768	0.383688	0.4059242	1.4935094	0.4806018	2.2492446
31	0.3913682	0.3813812	0.400992	1.4729058	0.4374332	2.2516356
32	0.3749012	0.379789	0.3859628	1.4584962	0.4354406	2.2925594

Table A.11: 10^7 operations, 2^4 bucket entries, 0% inserts, 100% removes and 0% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	6.8812982	7.8430362	8.8421006	8.5578352	10.2980494	8.583846
2	3.5635302	4.4131124	5.6235252	6.0617406	5.5163316	5.1361156
3	2.5141924	3.2192762	4.4048036	4.4823416	3.7683608	3.6136262
4	1.7304168	2.1824622	3.5094002	3.7784168	2.9242216	2.792842
5	1.4307994	1.825946	2.9722448	3.2228108	2.2494024	2.4605984
6	1.1942018	1.5605268	2.5450402	2.7772436	1.9585748	2.2997844
7	0.9705018	1.2656748	2.265806	2.4292636	1.7210044	2.1053422
8	0.862871	1.1394652	2.2025462	2.309931	1.4699556	2.0910472
9	0.7567302	0.9473558	2.1907518	2.1807798	1.3194612	2.0609448
10	0.6829118	0.8538414	2.17627	2.062641	1.2143884	2.0459562
11	0.6260398	0.7771766	2.1281186	2.0166438	1.0748276	1.9762048
12	0.5782414	0.7159918	2.0748108	1.9815066	0.9856402	1.9625592
13	0.5376408	0.6712854	2.0727638	1.9349816	0.9172348	1.9829144
14	0.5044636	0.6231854	2.0138292	1.8641382	0.8551994	1.9216212
15	0.4737982	0.5986696	2.0601884	1.8481768	0.8139482	1.9441902
16	0.4544842	0.5636084	2.3774572	1.8453308	0.7885608	1.9184374
17	0.4344242	0.5353358	2.3839368	1.7855868	0.7852988	1.9565408
18	0.4239234	0.513295	2.3358284	1.7340356	0.7342008	1.938565
19	0.4022004	0.4960364	2.3561344	1.6962554	0.7529502	1.9779838
20	0.3901102	0.4855032	2.370821	1.6655002	0.7214756	1.9900728
21	0.367844	0.4577696	2.05498	1.6518502	0.7036812	1.9926022
22	0.3726512	0.4442418	2.0963734	1.6026844	0.6856752	2.0041162
23	0.348812	0.426671	2.120201	1.5849998	0.6728632	2.035564
24	0.3398956	0.4167086	2.1435074	1.5553032	0.6614158	2.045794
25	0.3280438	0.4017096	2.1678052	1.5454	0.6419684	2.050755
26	0.318817	0.3941502	2.1745292	1.527275	0.6341758	2.0804418
27	0.3077382	0.3782912	2.2125044	1.4939306	0.6198888	2.1347416
28	0.297869	0.370607	2.2243042	1.493658	0.603639	2.130905
29	0.3087554	0.3570922	2.2308772	1.471858	0.5964414	2.1433366
30	0.2980678	0.35311	2.2673822	1.472082	0.5733462	2.1665016
31	0.2799242	0.3390992	2.3095434	1.4546016	0.5694584	2.2242588
32	0.293362	0.3347774	2.4688738	1.4425964	0.5945352	2.2528734

Table A.12: 10^7 operations, 2^4 bucket entries, 0% inserts, 0% removes and 100% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	6.1955522	5.9216218	6.7187546	6.5028254	7.3852124	7.1489118
2	3.205799	3.3706582	3.630035	4.6284244	3.8372764	3.8236002
3	2.2388036	2.2479152	2.5146164	3.5322266	2.727542	2.447922
4	1.562576	1.5699808	1.7556772	2.8937298	2.075711	1.9730712
5	1.3128652	1.3162706	1.460066	2.5943402	1.5679986	1.497821
6	1.106364	1.0994192	1.2217136	2.3112474	1.3451598	1.2031562
7	0.8629088	0.8667542	0.9764926	2.0893386	1.1567646	0.9810962
8	0.7646058	0.7694786	0.8538358	1.9660746	0.9403744	0.870567
9	0.674458	0.672866	0.818077	1.8858934	0.841323	0.7809748
10	0.614682	0.6123596	0.6882008	1.865076	0.867207	0.7039308
11	0.5592162	0.5644494	0.6246078	1.8804454	0.6935452	0.6437482
12	0.5151886	0.5149842	0.5799238	1.8794174	0.636211	0.5920012
13	0.4771454	0.478412	0.5446352	1.8490792	0.5918498	0.5529574
14	0.4473272	0.4480538	0.5026232	1.7409602	0.5534964	0.5256702
15	0.4216606	0.4286572	0.4675992	1.7566838	0.5244152	0.4887466
16	0.3986932	0.400634	0.4533934	1.7483262	0.50681	0.4606682
17	0.3823786	0.379786	0.4251996	1.6675114	0.4887764	0.4625274
18	0.3735882	0.3623148	0.4111882	1.6057126	0.458558	0.424019
19	0.3494772	0.3474004	0.3923368	1.5962594	0.4530532	0.411066
20	0.3324244	0.334242	0.3731362	1.5324412	0.4411578	0.4012246
21	0.3197594	0.3184412	0.3618386	1.4918454	0.4133088	0.3871194
22	0.3314816	0.3061792	0.3466288	1.4554336	0.4011286	0.3716124
23	0.294727	0.297822	0.358181	1.4284208	0.3993638	0.3643392
24	0.2884612	0.2871278	0.3382352	1.4006002	0.3804126	0.3603592
25	0.2789724	0.277989	0.319694	1.3704774	0.3639382	0.3404722
26	0.2753048	0.2687712	0.3167466	1.363718	0.3536182	0.3335638
27	0.283095	0.2583074	0.2974724	1.361724	0.359909	0.3193706
28	0.2481748	0.258202	0.28897	1.3620514	0.3268858	0.3133342
29	0.2653504	0.2638252	0.2925796	1.3585412	0.349546	0.3144976
30	0.2595054	0.2410446	0.2905018	1.3573984	0.3261268	0.3198854
31	0.249227	0.257801	0.2760116	1.3294966	0.3158112	0.3033988
32	0.2444006	0.2371026	0.2911276	1.3158324	0.3551418	0.289982

Table A.13: 10^7 operations, 2^4 bucket entries, 50% inserts, 50% removes and 0% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	7.1878502	6.8536578	8.2242548	7.6357698	8.9893516	7.4302
2	4.2113768	4.1699696	5.5968426	5.6505084	5.5420442	4.5252766
3	2.6734786	2.592255	3.7367484	4.0145522	3.6733194	3.2584746
4	2.0121334	1.9602236	2.8191832	3.2003844	2.6036852	2.4579136
5	1.7271518	1.7176444	2.5069252	2.6086672	2.2150618	2.3184416
6	1.4081368	1.3634826	2.2516972	2.2762538	1.8756052	2.057885
7	1.207113	1.2284588	2.1515158	2.1010434	1.6481828	2.1692626
8	0.9607024	0.9328704	2.0759074	2.0094032	1.4129914	2.1332908
9	0.8552854	0.8368242	2.0284236	2.0621922	1.2457026	2.1289714
10	0.7754828	0.755261	2.0146396	1.9979032	1.176199	2.0354912
11	0.7131326	0.695317	1.963518	1.9610366	0.9505146	2.0301344
12	0.652658	0.6492456	1.9902554	1.9387466	0.9002422	2.0026262
13	0.6125756	0.597928	1.9630104	1.9181796	0.8249902	1.9903556
14	0.5817396	0.5586674	1.8939298	1.9201102	0.7712354	1.9717686
15	0.5374266	0.5200274	1.8699682	1.9092602	0.7313208	1.9721744
16	0.5067448	0.5012284	1.975945	1.9106808	0.7053796	1.9574486
17	0.4874866	0.477592	1.99341	1.8694522	0.6770304	1.9888696
18	0.469916	0.4538232	1.9787174	1.8281304	0.6516358	1.9643578
19	0.4617936	0.4341214	1.9503356	1.8112624	0.627115	2.0280984
20	0.429776	0.4233944	1.9431818	1.7851848	0.6012836	2.031365
21	0.4181758	0.3984446	1.8259402	1.7634758	0.5870146	2.0104366
22	0.4069986	0.3872658	1.8586894	1.745577	0.5687354	2.0400812
23	0.4002352	0.371735	1.8368242	1.7364992	0.553063	2.0743626
24	0.376199	0.3590114	1.8132524	1.7090502	0.5315978	2.0937096
25	0.3598232	0.3463276	1.8308842	1.682736	0.5243666	2.1143966
26	0.3488036	0.3362198	1.800265	1.6741388	0.5072542	2.114483
27	0.3400714	0.3250836	1.8027226	1.6733348	0.4878618	2.1713292
28	0.3292914	0.3181428	1.816209	1.6578132	0.480719	2.1450096
29	0.3193614	0.3351458	1.8040038	1.6455424	0.4701208	2.1459342
30	0.317793	0.3020292	1.8165996	1.6396216	0.4645596	2.1929566
31	0.3138922	0.3181668	1.7909714	1.6185486	0.4575514	2.242732
32	0.308456	0.2977056	1.820843	1.6180288	0.4440004	2.2432474

Table A.14: 10^7 operations, 2^4 bucket entries, 5% inserts, 5% removes and 90% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	6.6618044	5.8426522	7.0585868	7.6412248	7.5869668	7.8181554
2	3.5026518	3.569091	4.26106	5.0493786	4.3856246	4.1382576
3	2.417571	2.4432082	2.9437296	3.7248188	2.9030144	2.6320722
4	1.6540424	1.6808038	2.0115986	2.9426514	2.2160152	2.1018806
5	1.3759086	1.4084572	1.7456922	2.634624	1.7460588	1.6323226
6	1.1986676	1.2178668	1.4329392	2.3394574	1.43345	1.2712506
7	0.9152522	0.9273148	1.3039718	2.1125578	1.2569412	1.1831474
8	0.8689684	0.8741082	1.238534	1.9774816	0.999523	0.9295846
9	0.7134926	0.7253078	1.0115492	1.9215726	0.9170144	0.8337904
10	0.6446174	0.657323	0.9526968	1.9231668	0.8137892	0.7482908
11	0.5949758	0.6115292	0.9103362	1.892449	0.7395714	0.6883092
12	0.5449498	0.5581982	0.8803536	1.8735584	0.6823336	0.6284774
13	0.5043012	0.5183054	0.8566598	1.8122904	0.634811	0.5830152
14	0.4798904	0.4824806	0.8405334	1.7688246	0.5988778	0.5485222
15	0.449603	0.4526012	0.8344706	1.7803472	0.5559346	0.5207712
16	0.4177642	0.4295314	0.8225728	1.7542054	0.5310158	0.4876018
17	0.4025268	0.4056894	0.8047672	1.702749	0.5038202	0.4761186
18	0.3788158	0.3874534	0.796962	1.6725608	0.4841564	0.4553346
19	0.3699196	0.3689812	0.7737024	1.653149	0.4584778	0.430647
20	0.3537596	0.3534906	0.7701694	1.62858	0.4453058	0.4256396
21	0.3418128	0.3397292	0.762994	1.5969964	0.4354212	0.4087148
22	0.3236224	0.329408	0.759011	1.5286972	0.4247072	0.3921516
23	0.312281	0.3183642	0.7463664	1.5010196	0.3979146	0.3816926
24	0.3056398	0.3037742	0.7383068	1.4829048	0.38851	0.3651922
25	0.290378	0.2948834	0.7175124	1.4782466	0.3736888	0.3634052
26	0.2800352	0.283642	0.7188656	1.4660902	0.3724824	0.3684014
27	0.294893	0.2778578	0.7201228	1.4466312	0.350149	0.3399742
28	0.2683034	0.2734374	0.7083312	1.4533618	0.3451952	0.3454148
29	0.256481	0.2697606	0.6905966	1.4406584	0.3321366	0.3371124
30	0.2600226	0.2622228	0.6805968	1.4202582	0.3413178	0.3249306
31	0.253869	0.2577558	0.6764812	1.3971532	0.3193674	0.32602
32	0.250501	0.2557254	0.665973	1.3845338	0.3179562	0.3351478

Table A.15: 10^7 operations, 2^4 bucket entries, 10% inserts, 10% removes and 80% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	5.616183	6.3295304	7.89534	7.1698894	8.0931242	7.2139578
2	3.5375812	3.692621	4.2386894	5.039195	4.4347994	4.2103282
3	2.458792	2.503677	3.1228582	3.72118	3.1199632	2.8997158
4	1.7479426	1.8076948	2.1833132	3.073573	2.2742978	2.2187274
5	1.4033528	1.452472	1.8562478	2.6508008	1.8183806	1.737475
6	1.2160894	1.232011	1.6355104	2.3373652	1.4832586	1.3363194
7	0.9392172	0.9598908	1.4288742	2.0800732	1.2983964	1.2131894
8	0.815512	0.8370908	1.3919952	2.0407708	1.1803762	0.9711248
9	0.7334264	0.7527654	1.2888466	1.9731734	1.1166198	0.8666862
10	0.6636462	0.6830526	1.2194828	1.9271454	0.846148	0.7874214
11	0.6049328	0.62025	1.1921408	1.9029556	0.7680222	0.723857
12	0.5625144	0.5754754	1.1892574	1.8708398	0.7125916	0.6757818
13	0.518224	0.53537	1.1558762	1.853464	0.658697	0.6321782
14	0.487559	0.4982732	1.1566008	1.8138674	0.6178658	0.5901242
15	0.4564466	0.469416	1.1552876	1.784647	0.5900044	0.5624244
16	0.4341608	0.4421492	1.1311372	1.761244	0.5486288	0.5385284
17	0.4199138	0.4218532	1.1297674	1.7559532	0.5321236	0.5212102
18	0.387935	0.3989468	1.1519698	1.715212	0.5082794	0.5103068
19	0.3755792	0.3922932	1.090952	1.679069	0.4865702	0.4901278
20	0.3605638	0.3671612	1.0997552	1.6499016	0.4624974	0.4943576
21	0.3464368	0.3502378	1.0122596	1.62004	0.4518162	0.4766996
22	0.3333628	0.348234	1.0257838	1.5898248	0.4409202	0.4725944
23	0.3222506	0.3262378	1.005856	1.5613064	0.422022	0.4656732
24	0.31123	0.3137804	0.9708764	1.5345332	0.4057932	0.4719726
25	0.300627	0.3091374	0.962059	1.5137128	0.4024592	0.473401
26	0.2869288	0.2989532	0.9527734	1.507717	0.3789256	0.4772222
27	0.283988	0.289095	0.94511	1.4853162	0.37375	0.4756676
28	0.2752872	0.310424	0.931748	1.4774842	0.3712596	0.47769
29	0.2698736	0.2806118	0.9142746	1.4792944	0.3551602	0.4832972
30	0.2645752	0.268264	0.9035652	1.461191	0.3632068	0.4821788
31	0.2608278	0.2878808	0.8901014	1.4441642	0.3421688	0.4915404
32	0.258304	0.2690542	0.8826842	1.4147348	0.3512908	0.4881862

Table A.16: 10^7 operations, 2^4 bucket entries, 15% inserts, 15% removes and 70% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	7.2247424	6.7795164	7.9958868	7.6008696	7.5635044	7.1980538
2	3.7458446	3.8233164	4.4174412	5.1959028	4.4808708	4.2524636
3	2.5181912	2.5830386	3.258622	3.7031998	3.2648762	3.0296402
4	1.8623816	1.8971996	2.2939932	3.0661762	2.3204944	2.281632
5	1.4526112	1.4743236	1.947862	2.6372754	1.865459	1.8030146
6	1.2335808	1.2574698	1.6634806	2.3036214	1.571652	1.479494
7	1.0067372	1.0429628	1.539152	2.1264918	1.3398644	1.2171022
8	0.8420362	0.8611808	1.439083	2.056209	1.2272892	1.1032202
9	0.7605922	0.7685736	1.4192824	1.96624	0.9626732	0.9312064
10	0.6785908	0.693254	1.3535834	1.9169356	0.869832	0.8521464
11	0.6201244	0.6346844	1.3779764	1.9042886	0.7907934	0.7829028
12	0.5761464	0.5867388	1.3250786	1.8834708	0.7426116	0.7436276
13	0.5361588	0.5443104	1.2981428	1.8493842	0.683115	0.702975
14	0.5122626	0.5108322	1.3484852	1.837967	0.6532512	0.6997614
15	0.4736834	0.481129	1.3020834	1.8299662	0.596897	0.6746704
16	0.4434796	0.457936	1.2892652	1.825271	0.5843678	0.6629478
17	0.4212494	0.4327976	1.301372	1.7646604	0.5482076	0.6555488
18	0.4030988	0.4105648	1.2884552	1.7418022	0.5236222	0.6627292
19	0.3895896	0.4073724	1.2849678	1.6843486	0.5078266	0.6493154
20	0.3734346	0.3789842	1.2544104	1.6697426	0.4936068	0.6565188
21	0.356498	0.3649602	1.2291896	1.6439688	0.4733968	0.6648526
22	0.3425602	0.3478946	1.2391302	1.6233488	0.4603774	0.6714984
23	0.3271452	0.3472186	1.2416586	1.5950408	0.4534706	0.666235
24	0.3157004	0.326526	1.2033668	1.5748602	0.4279498	0.6764782
25	0.307647	0.3119022	1.2041374	1.5600622	0.409408	0.675592
26	0.2997122	0.3072498	1.191846	1.5322554	0.3976202	0.676397
27	0.2975106	0.2950086	1.1609576	1.5253506	0.3958508	0.6996596
28	0.2857964	0.292869	1.1494	1.518086	0.3756158	0.6947276
29	0.2739286	0.2757106	1.1611062	1.5067548	0.365996	0.6980466
30	0.2710608	0.2724396	1.182984	1.4973716	0.3677226	0.7040684
31	0.2692334	0.2676732	1.1603912	1.4677614	0.3646032	0.7066814
32	0.2733008	0.2663002	1.1092598	1.4656886	0.3516268	0.7240268

Table A.17: 10^7 operations, 2^4 bucket entries, 25% inserts, 25% removes and 50% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	6.9278462	7.3547598	8.5238536	7.7138896	8.2256678	5.9906252
2	3.9024626	3.9608358	5.0270722	5.0763442	4.64307	3.7057574
3	2.6398802	2.7906442	3.4262516	3.9733078	3.4131212	2.6504388
4	1.9269284	1.9139394	2.5571078	3.0478282	2.4095812	2.0053192
5	1.4927244	1.5098086	2.1087574	2.6058286	1.9477398	1.6667838
6	1.2829428	1.2853434	1.823535	2.3093736	1.6176044	1.3720812
7	0.9961258	1.0359202	1.8259996	2.1064846	1.4099212	1.2597464
8	0.8762166	0.8940988	1.7532402	2.0712514	1.2763416	1.1886624
9	0.7828996	0.7964272	1.7447224	1.9648128	1.0858538	1.188798
10	0.7033218	0.7224338	1.7276408	1.9172284	0.920035	1.1525734
11	0.6460918	0.6579022	1.758417	1.8944572	0.8438438	1.077934
12	0.6062284	0.6082852	1.746489	1.868225	0.7878126	1.0918312
13	0.5665844	0.562572	1.687595	1.833971	0.7176644	1.0745442
14	0.5183292	0.5297194	1.7140302	1.8403686	0.687857	1.0509282
15	0.4886808	0.5000728	1.5835246	1.8344198	0.639375	1.0143444
16	0.4635162	0.4722312	1.7238602	1.8154312	0.622407	1.0564774
17	0.4484196	0.4546926	1.6539898	1.790848	0.5973278	1.0378774
18	0.4286484	0.4252084	1.5611736	1.7508902	0.5734368	1.0248308
19	0.4019574	0.4103994	1.565648	1.7266058	0.546905	1.090063
20	0.389767	0.3978012	1.5247538	1.6922432	0.5331158	1.1095572
21	0.3862698	0.3771752	1.4850994	1.6712386	0.5164952	1.1462752
22	0.3603184	0.3612626	1.446544	1.6508718	0.4934456	1.1672022
23	0.3608942	0.3484642	1.4514968	1.6423034	0.4750994	1.130784
24	0.335909	0.3418298	1.4341724	1.6062824	0.461249	1.158781
25	0.3277432	0.3237068	1.4066532	1.5833584	0.46173	1.1395602
26	0.3130218	0.3168434	1.3628596	1.5821702	0.4395558	1.1716566
27	0.3043524	0.312929	1.3956646	1.5695422	0.4219584	1.1559342
28	0.3000988	0.302201	1.3774004	1.548825	0.4159044	1.1739274
29	0.2952774	0.2939462	1.3785106	1.5408118	0.4020682	1.1645884
30	0.285722	0.2885024	1.3540628	1.55211	0.3933058	1.162408
31	0.2741278	0.2698042	1.3176198	1.5347364	0.4044896	1.1883218
32	0.2788526	0.2755542	1.3232354	1.5223352	0.385158	1.1833446

Table A.18: 10^7 operations, 2^4 bucket entries, 40% inserts, 40% removes and 20% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	6.6683776	7.0203348	8.0923912	7.5763826	8.9921174	6.9939452
2	4.0312042	4.1552818	5.2196206	5.421916	5.4782502	4.2707554
3	2.6125732	2.5852966	3.5663394	3.9449846	3.5333086	3.0321874
4	1.883727	1.862032	2.7935816	3.1622008	2.5233898	2.3364484
5	1.68543	1.6363882	2.2591398	2.5878512	1.987481	1.9032574
6	1.375495	1.3537038	2.1481094	2.2620638	1.8087606	1.922864
7	1.1627166	1.2006864	1.9958338	2.0582366	1.581675	1.833678
8	0.9181736	0.908848	1.9577094	1.9570334	1.3251018	1.763282
9	0.826519	0.823202	1.9158534	1.942055	1.2156666	1.7598122
10	0.7448156	0.749215	1.9373622	1.9039426	0.9843624	1.6770264
11	0.6847204	0.6769112	1.842083	1.8928648	0.9074318	1.6934878
12	0.6263854	0.6273536	1.8608298	1.8800644	0.8361546	1.6791586
13	0.5837886	0.5859294	1.8500608	1.8721408	0.7757812	1.6358228
14	0.5450558	0.5469122	1.8126318	1.834016	0.7291146	1.6205802
15	0.5290838	0.5111036	1.8130194	1.8473986	0.697002	1.6035698
16	0.4910276	0.4922434	1.8522438	1.8180216	0.6603522	1.6090074
17	0.4677288	0.4623128	1.8356172	1.8158716	0.6401336	1.6024186
18	0.44684	0.4388972	1.7957562	1.8142114	0.616676	1.6272776
19	0.4282494	0.424376	1.8031308	1.8037018	0.5979404	1.6499248
20	0.4182838	0.4017684	1.7820528	1.7853986	0.5797914	1.6835978
21	0.3935272	0.3857676	1.729647	1.7738268	0.5504394	1.6631294
22	0.3798464	0.3735528	1.7166322	1.7429024	0.5389072	1.679022
23	0.3708122	0.3709028	1.7259814	1.7238798	0.5156414	1.6990006
24	0.3575976	0.3496004	1.7195632	1.7085146	0.5049122	1.6842054
25	0.353944	0.3385424	1.7199978	1.6983092	0.486594	1.7127142
26	0.3375784	0.3357524	1.7069462	1.6763664	0.472257	1.7451548
27	0.3273048	0.3163636	1.6820036	1.662445	0.4567748	1.7554628
28	0.3220018	0.3186534	1.7027932	1.6605204	0.4513328	1.7440672
29	0.308376	0.3064686	1.706102	1.6427126	0.4421898	1.8095746
30	0.3070204	0.2946986	1.6789794	1.6356824	0.4271082	1.7796302
31	0.2937892	0.2849258	1.665067	1.6171748	0.4238832	1.7992566
32	0.295722	0.288473	1.668579	1.6076626	0.4185	1.8503998

Table A.19: 10^6 operations, 2^8 bucket entries, 100% inserts, 0% removes and 0% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.5906976	0.6511014	0.6874956	0.6413496	0.7520938	0.832854
2	0.3546038	0.3498034	0.3769506	0.4593806	0.426301	0.4927194
3	0.2485768	0.2474478	0.2637992	0.361919	0.2953016	0.366295
4	0.1904768	0.19294	0.2043836	0.3038756	0.2272822	0.2948452
5	0.1566948	0.157772	0.1671122	0.2702668	0.1845818	0.2525986
6	0.1358456	0.1350954	0.143034	0.2420108	0.1567724	0.224962
7	0.1208134	0.1184862	0.1280592	0.223376	0.1421348	0.2129166
8	0.1095816	0.1076566	0.1165266	0.2104528	0.1261444	0.205574
9	0.098098	0.0998974	0.1037326	0.1961636	0.1217278	0.2009218
10	0.0943436	0.093046	0.09464	0.1897038	0.106411	0.2066424
11	0.086016	0.086409	0.09206	0.1839058	0.1130968	0.2063814
12	0.0834682	0.0838958	0.0858916	0.1873858	0.0985868	0.203865
13	0.0777344	0.0777006	0.0847268	0.1806814	0.0926126	0.2048602
14	0.0782454	0.0758794	0.0821282	0.1674702	0.0905724	0.202378
15	0.0714328	0.075653	0.0771624	0.1666676	0.0852724	0.2043542
16	0.0759666	0.0745144	0.0771082	0.1639568	0.0835938	0.2011576
17	0.069866	0.067634	0.0784442	0.1665182	0.0830864	0.208966
18	0.0681618	0.0721888	0.0748472	0.1629592	0.0824948	0.2046524
19	0.0689356	0.069457	0.0718178	0.164781	0.0794772	0.2023552
20	0.0730092	0.065255	0.0708494	0.1603992	0.0847878	0.2007668
21	0.0691504	0.0645064	0.0722648	0.16149	0.0745204	0.207839
22	0.06419	0.0671522	0.0699936	0.1559158	0.076659	0.2023664
23	0.069437	0.0636294	0.067292	0.1574766	0.0737244	0.2121548
24	0.0698698	0.0690676	0.0675406	0.157979	0.0753296	0.2109314
25	0.063967	0.076895	0.0685826	0.156293	0.0768868	0.2171484
26	0.0692754	0.0665946	0.0799874	0.1586528	0.084809	0.2206604
27	0.072424	0.0657362	0.0696716	0.1535814	0.0786792	0.2198694
28	0.0679432	0.083378	0.0744864	0.1541164	0.0745948	0.2194724
29	0.0722892	0.0752648	0.069124	0.159572	0.0801786	0.2243618
30	0.0791296	0.0668854	0.0725376	0.1508734	0.0753808	0.2215868
31	0.0771774	0.073271	0.0777502	0.1508774	0.0844256	0.2248406
32	0.0771206	0.0716872	0.074022	0.1520412	0.0823888	0.2279118

Table A.20: 10^6 operations, 2^8 bucket entries, 0% inserts, 100% removes and 0% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.406863	0.5738512	0.5485404	0.5561504	0.687524	0.6713892
2	0.2316444	0.3014178	0.4199926	0.454319	0.3940652	0.4100402
3	0.1535838	0.202452	0.3335342	0.367855	0.2716148	0.302919
4	0.116265	0.152862	0.2966268	0.341643	0.206917	0.2423038
5	0.093695	0.1242384	0.2586996	0.2827872	0.1702802	0.2379318
6	0.0785296	0.1059686	0.2187616	0.2396202	0.1433178	0.2200158
7	0.0681944	0.0895368	0.2104658	0.2184482	0.1245708	0.2099412
8	0.0615108	0.0781654	0.2082968	0.2126332	0.110913	0.2036324
9	0.0537794	0.06939	0.2081564	0.2101608	0.101201	0.2019124
10	0.0485506	0.062585	0.2006578	0.2017836	0.0939044	0.1975216
11	0.0458706	0.0581326	0.1954912	0.1918554	0.0908804	0.1943572
12	0.0419028	0.0545228	0.1904462	0.1886714	0.0837486	0.1941342
13	0.039999	0.0513054	0.1904818	0.1835558	0.081154	0.1919418
14	0.0363788	0.049541	0.1904548	0.1808136	0.0792794	0.1947664
15	0.035925	0.0460858	0.193819	0.1777474	0.0717454	0.1942088
16	0.0336364	0.0466256	0.235153	0.1775956	0.0710868	0.1910746
17	0.0336166	0.0436282	0.2368404	0.1726618	0.0727964	0.1925208
18	0.034608	0.0441672	0.2372142	0.1649302	0.0708322	0.1959584
19	0.0293082	0.038017	0.2345584	0.1671546	0.0686086	0.2020424
20	0.0286528	0.041055	0.2268834	0.1635638	0.0656076	0.1988042
21	0.0309802	0.0362382	0.1966294	0.1610012	0.0685676	0.2024918
22	0.0303984	0.0379916	0.1977892	0.1597018	0.0669662	0.1966988
23	0.0265784	0.0354712	0.1996458	0.1559176	0.0657362	0.2011392
24	0.0312404	0.034651	0.2155586	0.1535812	0.062522	0.2070122
25	0.0261892	0.0349576	0.2003238	0.15769	0.061688	0.2009812
26	0.0288348	0.0354434	0.2074564	0.1488808	0.0607318	0.206756
27	0.0268004	0.0307924	0.2084232	0.1560098	0.0619758	0.1982072
28	0.0266916	0.0376112	0.2048714	0.1561362	0.0634302	0.203428
29	0.0280832	0.0357066	0.2046082	0.1445158	0.063704	0.2056164
30	0.0303338	0.0340684	0.2137854	0.1531286	0.0576168	0.2073898
31	0.0275432	0.0347034	0.2140796	0.1473432	0.062118	0.2108364
32	0.0285812	0.0358754	0.2460816	0.147557	0.066762	0.2076626

Table A.21: 10^6 operations, 2^8 bucket entries, 0% inserts, 0% removes and 100% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.3424866	0.318266	0.4050034	0.3830582	0.4219192	0.4404996
2	0.1903312	0.187992	0.2278528	0.3510742	0.2516852	0.2584118
3	0.1297244	0.1297354	0.1509782	0.268915	0.1735672	0.178313
4	0.0967062	0.0963302	0.1124558	0.2330922	0.1314736	0.1371504
5	0.078709	0.079058	0.0911458	0.2106692	0.1071536	0.1113862
6	0.0656122	0.0653978	0.0755802	0.198855	0.0885344	0.094915
7	0.0568032	0.0564784	0.0668914	0.1884934	0.0770686	0.0820576
8	0.0494162	0.050573	0.0582442	0.1812196	0.0674976	0.073619
9	0.0441154	0.0453868	0.0522784	0.1719214	0.0620766	0.0661494
10	0.0411802	0.0398804	0.047574	0.1776606	0.0563068	0.0610006
11	0.0387468	0.0385318	0.04417	0.1771054	0.052782	0.0556888
12	0.0342886	0.0374394	0.0392176	0.1774792	0.0508216	0.0528194
13	0.032557	0.0323778	0.0397648	0.16712	0.0450406	0.0498456
14	0.030148	0.0302948	0.0348596	0.1651974	0.0418588	0.0469
15	0.0300366	0.0289582	0.0331838	0.158981	0.0406468	0.044757
16	0.0277206	0.0273166	0.0328954	0.1624864	0.0399896	0.0487978
17	0.0258742	0.0308568	0.0305364	0.1610804	0.0396696	0.0418924
18	0.0260948	0.0239608	0.0318462	0.1560118	0.0376338	0.0385802
19	0.0267372	0.02626	0.0305186	0.1527844	0.0329924	0.0359094
20	0.0234374	0.0229562	0.0273494	0.144111	0.0363204	0.0370916
21	0.0271342	0.0241672	0.0309344	0.1442734	0.0343348	0.0390338
22	0.0236144	0.0243856	0.0243942	0.139813	0.035285	0.0378044
23	0.0207026	0.0247944	0.031176	0.1386078	0.0315694	0.040462
24	0.0224946	0.0210868	0.0261396	0.136493	0.0338572	0.0353256
25	0.0254986	0.0242474	0.0288908	0.1324688	0.0294396	0.0362436
26	0.0250674	0.0252174	0.0255508	0.1296084	0.0369664	0.0340918
27	0.0242314	0.0242288	0.0276266	0.1290182	0.0328102	0.0394166
28	0.0265512	0.0196834	0.0280624	0.1257786	0.0300414	0.0332864
29	0.02059	0.0212554	0.0234186	0.1253458	0.0298394	0.0335118
30	0.022865	0.0234506	0.0285548	0.1255454	0.0290418	0.0347814
31	0.0247336	0.024289	0.02732	0.124716	0.0307374	0.031302
32	0.023638	0.022438	0.0265606	0.1243122	0.0327114	0.0358696

Table A.22: 10^6 operations, 2^8 bucket entries, 50% inserts, 50% removes and 0% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.4885462	0.4494234	0.504547	0.5424176	0.6221652	0.5994604
2	0.249931	0.2438412	0.3558372	0.3816752	0.3492664	0.3832336
3	0.1679106	0.1628224	0.261732	0.289661	0.239971	0.287331
4	0.1284048	0.1236182	0.2212582	0.250212	0.1822986	0.2367216
5	0.1035134	0.1012534	0.1938606	0.2174498	0.149056	0.225169
6	0.0859114	0.084721	0.1851774	0.1997618	0.1248954	0.213578
7	0.0751824	0.0736516	0.1831954	0.194998	0.1080036	0.2066438
8	0.066205	0.0661712	0.1816666	0.186423	0.0961452	0.2068536
9	0.0606568	0.0590088	0.1773822	0.1843134	0.0903064	0.2036736
10	0.0540196	0.0547966	0.1740762	0.1794016	0.0803848	0.2041424
11	0.0500652	0.0485092	0.1756524	0.1789836	0.0806312	0.1918108
12	0.0471926	0.046169	0.177058	0.1764112	0.07854	0.196648
13	0.0442906	0.0471852	0.1723782	0.1730946	0.0676778	0.1982822
14	0.0440176	0.0449942	0.1732174	0.168623	0.066183	0.1956646
15	0.0393378	0.038379	0.1722672	0.1807124	0.0609134	0.1967674
16	0.0393144	0.0407556	0.1845004	0.169312	0.0637808	0.1960966
17	0.0385322	0.0375366	0.186369	0.1652432	0.0615362	0.1970666
18	0.0391344	0.0374672	0.1842338	0.1670234	0.0603538	0.1939006
19	0.0371866	0.0363482	0.1840318	0.1609356	0.0557842	0.199899
20	0.0348036	0.0332472	0.1802306	0.160836	0.0589732	0.1991862
21	0.0344088	0.037021	0.1682244	0.1592464	0.056678	0.1963306
22	0.034195	0.0310324	0.170289	0.1601012	0.0520332	0.2015892
23	0.0317122	0.031368	0.1676848	0.1564408	0.0506654	0.2071408
24	0.0316426	0.0322732	0.1724408	0.1574366	0.0500654	0.2035772
25	0.0316034	0.0315796	0.169095	0.1536658	0.0498462	0.2068292
26	0.030871	0.0283224	0.1642892	0.152165	0.0492854	0.2066548
27	0.0316458	0.0329006	0.1643372	0.1494918	0.051529	0.2023506
28	0.0302642	0.0288078	0.1636162	0.1487296	0.0460088	0.2070804
29	0.0359792	0.0325042	0.1674608	0.1479826	0.051404	0.2094464
30	0.027527	0.027671	0.1652708	0.1464534	0.0459286	0.2125064
31	0.0271866	0.033307	0.165846	0.1450012	0.046765	0.211919
32	0.0347978	0.0349322	0.1701576	0.1514764	0.0542344	0.2141714

Table A.23: 10^6 operations, 2^8 bucket entries, 5% inserts, 5% removes and 90% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.3470976	0.3547338	0.4558894	0.4080542	0.4940288	0.4843558
2	0.200511	0.2053488	0.2473468	0.3508526	0.275404	0.2727634
3	0.1340614	0.136393	0.1721526	0.268278	0.1847444	0.1884106
4	0.1006236	0.102078	0.1339202	0.2349856	0.1395872	0.1448556
5	0.0809622	0.0825868	0.1118698	0.2106622	0.1132242	0.119502
6	0.06796	0.069203	0.0980958	0.191289	0.0950072	0.1023092
7	0.059397	0.0606348	0.0901318	0.1815756	0.083146	0.0880734
8	0.0518614	0.052587	0.0841676	0.1823636	0.0739044	0.0784824
9	0.0477004	0.0477876	0.0789508	0.1780492	0.0664972	0.0723246
10	0.0428592	0.042604	0.0770216	0.1758184	0.0600598	0.06615
11	0.0393602	0.0400138	0.0738454	0.1758788	0.053675	0.0609974
12	0.0357214	0.036792	0.0736364	0.1773328	0.0530344	0.0564958
13	0.0345364	0.0349178	0.0722162	0.1712368	0.051504	0.0534584
14	0.0322822	0.0330958	0.0719314	0.1626314	0.0442052	0.0495876
15	0.0314452	0.031501	0.0722336	0.1680446	0.04236	0.0478484
16	0.030471	0.0291808	0.0715134	0.1639588	0.0443898	0.0445832
17	0.0292094	0.0276058	0.0706926	0.157427	0.0432054	0.044514
18	0.0272668	0.0287534	0.0690542	0.1589412	0.0415218	0.0452156
19	0.0260562	0.0262776	0.0687524	0.1536276	0.0387036	0.0434196
20	0.0272018	0.0276644	0.0678866	0.1535372	0.0402758	0.0410692
21	0.0303498	0.0294652	0.0682924	0.1475814	0.0396754	0.0431992
22	0.0255132	0.0261872	0.0679988	0.144232	0.0364734	0.0411042
23	0.0233104	0.0262628	0.0681746	0.1428122	0.03621	0.039239
24	0.026559	0.022241	0.0660328	0.1403718	0.0344354	0.0400892
25	0.0278304	0.0253882	0.0667868	0.137888	0.035184	0.0377668
26	0.0257502	0.024394	0.0645694	0.1358344	0.0346884	0.0385632
27	0.0240286	0.0270542	0.0647552	0.1341322	0.0328782	0.042442
28	0.0269544	0.0229184	0.0638324	0.1315002	0.0328948	0.0376254
29	0.0254856	0.0228584	0.0646978	0.1311228	0.0311174	0.0384806
30	0.0233864	0.0240222	0.0649664	0.1296328	0.0334562	0.0418346
31	0.024274	0.0259686	0.0647266	0.127635	0.0412514	0.0377162
32	0.0254268	0.0270078	0.066717	0.1283436	0.0392334	0.0402214

Table A.24: 10^6 operations, 2^8 bucket entries, 10% inserts, 10% removes and 80% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.3184314	0.3460454	0.4316666	0.4217398	0.5151132	0.5276
2	0.2051252	0.2095042	0.2682578	0.354059	0.2863022	0.2916758
3	0.1362924	0.1383536	0.186244	0.2698396	0.1917986	0.200147
4	0.1025132	0.104395	0.1465296	0.2315304	0.1450196	0.1540032
5	0.0833416	0.0845396	0.1243682	0.2100484	0.1187724	0.125236
6	0.0695078	0.0708832	0.1132614	0.1897324	0.0996112	0.1076214
7	0.0608042	0.0617438	0.1052208	0.1860626	0.0853716	0.0948036
8	0.052485	0.0547482	0.1021608	0.1818792	0.075846	0.084185
9	0.0485792	0.0484418	0.099682	0.1775974	0.0682336	0.077347
10	0.043881	0.0451616	0.0998218	0.1776312	0.0623408	0.0708688
11	0.0401098	0.041168	0.0962668	0.1787214	0.0613336	0.0667482
12	0.036438	0.0383304	0.0957982	0.1730876	0.0541856	0.0631278
13	0.0348668	0.0364868	0.0945434	0.1687396	0.0522488	0.0600524
14	0.0328648	0.0337024	0.094422	0.1661566	0.0502866	0.0578044
15	0.03377	0.0333638	0.0960728	0.1719122	0.0508996	0.0544678
16	0.0294178	0.0307774	0.095483	0.1563258	0.0451844	0.0528172
17	0.0299484	0.0313354	0.0965444	0.1591486	0.0438696	0.0519766
18	0.0279668	0.0307398	0.0950142	0.158602	0.048853	0.052157
19	0.027512	0.027283	0.0923578	0.155736	0.0417708	0.0511034
20	0.0255124	0.0273414	0.091494	0.1522462	0.0408072	0.0505018
21	0.0254366	0.0262446	0.0923762	0.1520964	0.0387482	0.0532624
22	0.0291392	0.0260536	0.0918576	0.1480298	0.0442336	0.0533682
23	0.0270498	0.0246632	0.0898086	0.144604	0.0391956	0.0534528
24	0.0253474	0.025211	0.0891586	0.1438182	0.0354524	0.051859
25	0.0255664	0.0254472	0.0891886	0.1398922	0.0375846	0.0522182
26	0.0274258	0.0311918	0.0893104	0.1381396	0.03599	0.05086
27	0.0268904	0.028528	0.0886806	0.1370536	0.0384346	0.051707
28	0.0211978	0.0286612	0.0880866	0.1348602	0.03506	0.0525682
29	0.0265718	0.0252162	0.090009	0.1338078	0.0373278	0.050824
30	0.0239258	0.0278908	0.0867002	0.1321838	0.0393784	0.0529818
31	0.023834	0.0279398	0.0862764	0.131959	0.034795	0.0533272
32	0.024222	0.026614	0.0887268	0.1344476	0.037574	0.0526452

Table A.25: 10^6 operations, 2^8 bucket entries, 15% inserts, 15% removes and 70% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.3656458	0.4181394	0.5018748	0.4710684	0.5305014	0.508507
2	0.210281	0.2084432	0.2778488	0.3497814	0.2980054	0.304138
3	0.138729	0.1430732	0.199157	0.2685652	0.1995262	0.2097976
4	0.1055896	0.107966	0.158616	0.2322378	0.150146	0.1640498
5	0.0849078	0.0878478	0.135668	0.2083486	0.1218632	0.1344666
6	0.0709358	0.0733678	0.1238426	0.1944744	0.1029536	0.11545
7	0.0610628	0.0636578	0.119416	0.1827696	0.0900034	0.101298
8	0.0543296	0.0561088	0.1169358	0.1824824	0.0785066	0.0935718
9	0.0491608	0.0503092	0.115284	0.1743646	0.073061	0.0845976
10	0.045841	0.0452284	0.1117898	0.1696564	0.066646	0.080502
11	0.0418252	0.0424944	0.115483	0.1775564	0.0594284	0.07528
12	0.0389018	0.0414998	0.1105196	0.166708	0.0562218	0.0737774
13	0.0369132	0.0364722	0.1132372	0.1718492	0.050593	0.0718832
14	0.034345	0.0345966	0.1109488	0.1648806	0.0512188	0.0691198
15	0.0327408	0.0325066	0.1126478	0.1697742	0.048102	0.0694338
16	0.030804	0.0311014	0.1157334	0.1664386	0.0521428	0.0691878
17	0.0297346	0.031843	0.1120298	0.1621468	0.047171	0.068692
18	0.0293004	0.0304178	0.112539	0.160202	0.0460146	0.0681496
19	0.0303142	0.0278254	0.1122734	0.1578136	0.04683	0.0695104
20	0.0284042	0.0266174	0.1116518	0.1551526	0.0416182	0.0708184
21	0.0316732	0.0281226	0.107626	0.1521922	0.0404322	0.0699532
22	0.0283524	0.0262566	0.1098912	0.1515746	0.042154	0.0703594
23	0.0244466	0.026942	0.1087782	0.1477428	0.0421558	0.0700916
24	0.0276886	0.0268068	0.1077414	0.147198	0.0380206	0.0729366
25	0.0261778	0.025924	0.1063066	0.1444526	0.03734	0.071191
26	0.0255556	0.0241322	0.1056374	0.1407664	0.0390534	0.0704882
27	0.0222542	0.027494	0.1034622	0.140943	0.0361248	0.0714196
28	0.0254316	0.0242142	0.1014822	0.1392382	0.0417278	0.0707272
29	0.0245046	0.0247264	0.1023106	0.1355868	0.033794	0.07153
30	0.0247534	0.0235772	0.1016344	0.139559	0.0377998	0.0725642
31	0.0277746	0.026477	0.1015564	0.1360668	0.0376892	0.0723948
32	0.0282468	0.0315476	0.103495	0.1359994	0.0382976	0.0729414

Table A.26: 10^6 operations, 2^8 bucket entries, 25% inserts, 25% removes and 50% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.369485	0.391072	0.5057136	0.448458	0.583119	0.5659378
2	0.2152242	0.214202	0.3042486	0.3552504	0.3106148	0.3286644
3	0.1469048	0.1485076	0.2175176	0.2700298	0.2101926	0.2346246
4	0.110596	0.112051	0.1771348	0.2317696	0.1596796	0.1860698
5	0.0890416	0.0913046	0.1553688	0.2050692	0.1303862	0.1587232
6	0.0754016	0.0758604	0.1442106	0.1937654	0.1093346	0.1412396
7	0.0655946	0.0663104	0.1480276	0.185853	0.0940766	0.1281638
8	0.0566446	0.0583332	0.1414876	0.1829482	0.0857596	0.1188894
9	0.0519382	0.052336	0.1374112	0.1810778	0.0760538	0.1143058
10	0.0476006	0.0479426	0.1412494	0.1708224	0.0688902	0.1084386
11	0.044267	0.0449696	0.1364196	0.1785588	0.0671594	0.1083534
12	0.0424476	0.0413532	0.1395602	0.1671194	0.0690374	0.1066118
13	0.0380616	0.0426182	0.1393896	0.1647502	0.056682	0.104972
14	0.036568	0.0376136	0.1324348	0.1695392	0.0588974	0.1055902
15	0.0355324	0.0348826	0.135569	0.1643332	0.0549952	0.1054548
16	0.0336874	0.0333336	0.1418858	0.1672242	0.053971	0.1039564
17	0.0327872	0.0330368	0.1394768	0.161066	0.0489826	0.105588
18	0.0306052	0.0305834	0.1366966	0.160275	0.050046	0.1061424
19	0.03299	0.0287304	0.1384208	0.1634142	0.047765	0.1077076
20	0.0322454	0.0293332	0.1357484	0.1594382	0.0509394	0.1073704
21	0.027135	0.0323286	0.1309338	0.1572344	0.045229	0.1078732
22	0.0285744	0.0306686	0.1316272	0.1580758	0.048493	0.1073268
23	0.0263962	0.0265092	0.1320516	0.1516012	0.043152	0.1097316
24	0.024604	0.0279616	0.1308134	0.1506622	0.0439422	0.1089148
25	0.0303596	0.0248052	0.1301448	0.149518	0.0392658	0.1107822
26	0.0255222	0.0297482	0.1297374	0.1465234	0.0407958	0.11057
27	0.0256848	0.0295558	0.1242782	0.1449766	0.03914	0.1104188
28	0.027129	0.0283294	0.1257574	0.1425748	0.0429216	0.1132992
29	0.0243224	0.0286844	0.1231236	0.1416312	0.0381872	0.1141138
30	0.0229284	0.024446	0.1224016	0.1406134	0.0460908	0.111359
31	0.0269496	0.0272464	0.1223638	0.1417618	0.0438026	0.1135402
32	0.0246212	0.0324616	0.1282622	0.1413832	0.0452048	0.1164374

Table A.27: 10^6 operations, 2^8 bucket entries, 40% inserts, 40% removes and 20% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	0.4191788	0.3960116	0.4758428	0.5017942	0.6248074	0.6351868
2	0.2356902	0.2376704	0.3307564	0.369282	0.3348088	0.3806386
3	0.157723	0.1581056	0.242929	0.2789474	0.2292606	0.2772358
4	0.1197714	0.1175494	0.2008044	0.2367654	0.1743712	0.2286758
5	0.097042	0.09555	0.1781038	0.2124574	0.1406908	0.1982936
6	0.081124	0.0804176	0.1726664	0.1979288	0.1186236	0.1804248
7	0.0703216	0.0692676	0.1674194	0.1889606	0.1025816	0.1749708
8	0.0617508	0.0613688	0.170541	0.1805114	0.0900328	0.1641678
9	0.056609	0.0561076	0.1655766	0.1812546	0.0827474	0.1671936
10	0.050639	0.0502446	0.1644378	0.1731744	0.0744114	0.1643722
11	0.0482358	0.0471216	0.1662418	0.1758554	0.0735732	0.1640218
12	0.0445594	0.042794	0.164134	0.1731228	0.0721956	0.1612024
13	0.040895	0.042134	0.1613076	0.166195	0.0696484	0.1586986
14	0.0387622	0.0390574	0.1639286	0.1749646	0.0591078	0.1579268
15	0.0358144	0.0370056	0.1608458	0.173852	0.056932	0.1545302
16	0.0369456	0.0363752	0.170151	0.1683942	0.0598726	0.1600402
17	0.0362812	0.0345488	0.1718098	0.16908	0.0561684	0.1543378
18	0.0335882	0.0317548	0.1652748	0.1646122	0.0542344	0.1596866
19	0.031229	0.031988	0.164626	0.1644742	0.0538116	0.1644768
20	0.0347534	0.032001	0.1637768	0.1581954	0.054206	0.1591206
21	0.0339738	0.0327044	0.1554526	0.1581062	0.0510942	0.1635916
22	0.0296452	0.0293424	0.1547958	0.1585992	0.0479602	0.163325
23	0.0313726	0.031377	0.153875	0.1558888	0.0491106	0.1661676
24	0.0288844	0.029081	0.156669	0.1542716	0.0470352	0.1646404
25	0.0269082	0.0307198	0.1528558	0.152317	0.0467974	0.1718028
26	0.0280322	0.0280792	0.154327	0.1502962	0.043791	0.165025
27	0.0297414	0.0276096	0.1522576	0.1484058	0.0512868	0.1683328
28	0.0268886	0.0284232	0.1520746	0.1481726	0.0430682	0.1722578
29	0.0284136	0.0283996	0.149058	0.1457088	0.0525446	0.1726868
30	0.0270996	0.0272784	0.1482634	0.1438838	0.045795	0.175919
31	0.0330798	0.031444	0.147624	0.1463824	0.0470594	0.1782274
32	0.0297884	0.0331044	0.1512282	0.1467782	0.0470884	0.1751

Table A.28: 10^7 operations, 2^8 bucket entries, 100% inserts, 0% removes and 0% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	5.6450852	5.4401996	6.191636	5.923075	6.5706784	9.530383
2	3.122895	3.2999138	3.4570618	4.2127614	3.9937538	5.6668086
3	2.37241	2.3932254	2.505429	3.6002296	2.8566718	4.216107
4	1.8543092	1.8533606	1.9547012	2.9761448	2.2147372	3.3333674
5	1.3678622	1.3760366	1.4692058	2.6754208	1.8060942	2.7249544
6	1.0969472	1.1071098	1.209761	2.3900234	1.3462126	2.432537
7	0.900525	0.9086064	0.9709052	2.2267608	1.1581098	2.3572212
8	0.7952838	0.7944728	0.8575142	2.127614	0.9641964	2.226948
9	0.7131664	0.7196848	0.76754	2.0445918	0.8686836	2.254451
10	0.642977	0.6532486	0.696872	1.9680922	0.7771996	2.148221
11	0.5902794	0.5957286	0.637279	1.9412754	0.7149916	2.0809474
12	0.5532402	0.5530046	0.5857066	1.910924	0.7030884	2.163218
13	0.5367902	0.522001	0.5525302	1.8789258	0.626758	2.1019218
14	0.5152752	0.5021318	0.5429908	1.844539	0.6089306	2.0564954
15	0.4524466	0.4718012	0.4980956	1.7893392	0.5755858	2.0877114
16	0.4586376	0.4486186	0.4869196	1.7916168	0.5505548	2.088803
17	0.4415734	0.4421956	0.473369	1.7838348	0.5540868	2.0894508
18	0.4196198	0.4108074	0.467159	1.7010672	0.5235822	2.0063398
19	0.4149454	0.3960646	0.4418886	1.714438	0.5042416	2.101988
20	0.4095278	0.397004	0.4346842	1.6754694	0.5124962	2.0639748
21	0.3907044	0.3745188	0.4134128	1.6282178	0.4782274	2.1111432
22	0.3668358	0.370432	0.4066906	1.6117516	0.4652574	2.1413642
23	0.3649074	0.3714792	0.3958558	1.5903572	0.4522434	2.1688352
24	0.3510516	0.3465274	0.378777	1.4852824	0.4430652	2.156899
25	0.3451032	0.3447414	0.365607	1.4699282	0.4268418	2.2053844
26	0.3347702	0.3373988	0.3688608	1.4634178	0.4138708	2.1901296
27	0.327047	0.3223882	0.3589958	1.4529758	0.4037746	2.2216012
28	0.3271114	0.323104	0.362738	1.4297064	0.389462	2.241313
29	0.308618	0.3166708	0.338918	1.4190064	0.3929122	2.2073292
30	0.3120784	0.3209746	0.3413908	1.4073266	0.370965	2.2492446
31	0.311332	0.311861	0.3225616	1.37341	0.381842	2.2516356
32	0.3027128	0.3004748	0.3239834	1.3572028	0.3619012	2.2925594

Table A.29: 10^7 operations, 2^8 bucket entries, 0% inserts, 100% removes and 0% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	5.4889444	6.3863794	7.5199092	7.0785974	8.8438902	8.583846
2	2.7533268	3.7539766	4.9845444	5.2391272	5.0642192	5.1361156
3	1.9607556	2.5214982	3.8336064	4.1143894	3.342549	3.6136262
4	1.3680676	1.8510012	3.327924	3.6194114	2.4892214	2.792842
5	1.2292998	1.5488552	2.820784	3.0075842	1.9868692	2.4605984
6	0.9548414	1.2206098	2.472514	2.6276602	1.7059586	2.2997844
7	0.7524706	0.9710052	2.3902726	2.3476702	1.5090868	2.1053422
8	0.653574	0.8517362	2.2611418	2.2901406	1.2610482	2.0910472
9	0.5866288	0.754785	2.1864988	2.1557588	1.0873024	2.0609448
10	0.5285836	0.6913018	2.1071898	2.0708654	0.9744472	2.0459562
11	0.4870232	0.6272218	2.0930486	1.9801632	0.8891996	1.9762048
12	0.451612	0.5715994	2.0155874	1.9472966	0.8344934	1.9625592
13	0.409312	0.5271242	2.1009594	1.8799272	0.7968718	1.9829144
14	0.3871086	0.4925794	2.0354764	1.8571986	0.7408304	1.9216212
15	0.3574118	0.4832596	1.981525	1.887231	0.6966386	1.9441902
16	0.354711	0.449648	2.3885334	1.828362	0.7094306	1.9184374
17	0.3352846	0.4318398	2.3528074	1.7753272	0.688129	1.9565408
18	0.320403	0.4051608	2.3650226	1.7270044	0.6927162	1.938565
19	0.3183108	0.4011926	2.3210392	1.7193484	0.6526612	1.9779838
20	0.2952788	0.3809542	2.2782356	1.665855	0.6416178	1.9900728
21	0.2888724	0.361476	1.98976	1.6268238	0.6122674	1.9926022
22	0.2764348	0.3531848	2.0350914	1.6244152	0.5973442	2.0041162
23	0.2789092	0.3426494	2.03938	1.604221	0.5958616	2.035564
24	0.2606976	0.3291934	2.166809	1.5995712	0.5796992	2.045794
25	0.2669818	0.3175452	2.1081618	1.5850644	0.5685688	2.050755
26	0.2414916	0.307373	2.0959006	1.5582326	0.5524338	2.0804418
27	0.2317778	0.2956596	2.1145428	1.5332468	0.5345862	2.1347416
28	0.2275064	0.2951502	2.1379902	1.5049788	0.5205008	2.130905
29	0.23516	0.281901	2.1260208	1.5078184	0.5238424	2.1433366
30	0.2336718	0.2731862	2.1428462	1.4842608	0.5086586	2.1665016
31	0.22012	0.2656596	2.1541886	1.4626708	0.497016	2.2242588
32	0.2093992	0.2639582	2.4417332	1.4654176	0.4885264	2.2528734

Table A.30: 10^7 operations, 2^8 bucket entries, 0% inserts, 0% removes and 100% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	4.5772592	4.0488638	5.6370254	4.6548524	5.852426	7.1489118
2	2.5601264	2.429729	2.9349054	4.1427674	3.351208	3.8236002
3	1.6411912	1.6287468	1.93922	3.15822	2.2415256	2.447922
4	1.1838028	1.2146692	1.4935144	2.6297398	1.6075396	1.9730712
5	0.9543654	0.9919622	1.2971172	2.3279972	1.1991542	1.497821
6	0.8325898	0.824498	0.9192612	2.154941	1.1485204	1.2031562
7	0.6571154	0.6568348	0.7489578	1.9763464	0.895077	0.9810962
8	0.5716486	0.5736472	0.6542262	1.909797	0.7597422	0.870567
9	0.5135218	0.5083886	0.580954	1.8876916	0.6590368	0.7809748
10	0.4618636	0.4580626	0.5214498	1.8390128	0.5924678	0.7039308
11	0.4183378	0.417298	0.4775348	1.852643	0.5407974	0.6437482
12	0.3898262	0.3830604	0.43834	1.8338782	0.5040472	0.5920012
13	0.3537232	0.3547286	0.4102242	1.7994504	0.458047	0.5529574
14	0.3345018	0.3396058	0.3780182	1.739708	0.432038	0.5256702
15	0.3114302	0.3090636	0.3578022	1.7399658	0.4094724	0.4887466
16	0.2906754	0.2958748	0.343951	1.7515874	0.3869462	0.4606682
17	0.280328	0.2806424	0.325971	1.650514	0.37591	0.4625274
18	0.2624058	0.2627616	0.3139486	1.592483	0.3576702	0.424019
19	0.25853	0.252478	0.292354	1.57071	0.3550004	0.411066
20	0.2419656	0.2513054	0.2842492	1.537908	0.339588	0.4012246
21	0.2336992	0.2319212	0.2782018	1.4922162	0.3289102	0.3871194
22	0.2245984	0.2336334	0.2627862	1.4840672	0.319113	0.3716124
23	0.2133252	0.2361162	0.2548642	1.447625	0.3131068	0.3643392
24	0.2151512	0.2224544	0.2597424	1.4600176	0.2938638	0.3603592
25	0.2007524	0.205868	0.2383268	1.4421126	0.286191	0.3404722
26	0.2130076	0.1924586	0.2323634	1.420635	0.278734	0.3335638
27	0.2052732	0.1915632	0.244126	1.4038612	0.2669356	0.3193706
28	0.2090106	0.1818658	0.2317544	1.3972124	0.263842	0.3133342
29	0.1896358	0.1820438	0.2178696	1.3731114	0.2883738	0.3144976
30	0.1860376	0.1833444	0.2063696	1.3697604	0.2561444	0.3198854
31	0.1831202	0.1870304	0.209288	1.329057	0.2473898	0.3033988
32	0.1921538	0.1867248	0.2103922	1.3251844	0.2628878	0.289982

Table A.31: 10^7 operations, 2^8 bucket entries, 50% inserts, 50% removes and 0% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	5.0903864	5.2521086	6.5075196	5.995897	7.9253262	7.4302
2	3.009908	2.8677168	4.1282452	4.6428042	4.1179642	4.5252766
3	2.1765802	2.0793166	2.9903898	3.2747668	2.7706418	3.2584746
4	1.5464024	1.6342806	2.5357084	2.701502	2.241641	2.4579136
5	1.2397858	1.2531034	2.2439292	2.3014416	1.81928	2.3184416
6	0.9327896	0.9071378	2.043087	2.10445	1.463915	2.057885
7	0.8011694	0.7795096	1.9844562	2.0315326	1.2556964	2.1692626
8	0.7055928	0.681681	1.991382	2.007777	1.050863	2.1332908
9	0.6306938	0.6132146	1.9471802	1.9507166	0.911884	2.1289714
10	0.5742098	0.546382	1.9213328	1.9154616	0.8253548	2.0354912
11	0.5243874	0.506904	1.8702408	1.8659112	0.7589418	2.0301344
12	0.4847516	0.4703272	1.8894798	1.8670564	0.7155514	2.0026262
13	0.4468602	0.4332952	1.8636426	1.8640398	0.6662324	1.9903556
14	0.4144458	0.4042942	1.8066988	1.8737004	0.6170376	1.9717686
15	0.399362	0.3844598	1.8401354	1.8714498	0.5919864	1.9721744
16	0.3839966	0.3546206	1.976232	1.85366	0.5565186	1.9574486
17	0.3686786	0.3506766	1.9117402	1.8113822	0.5718156	1.9888696
18	0.3477888	0.332143	1.9158078	1.777699	0.5491784	1.9643578
19	0.3315024	0.327887	1.9047846	1.7624954	0.534747	2.0280984
20	0.3236618	0.3024626	1.896956	1.7254202	0.5174476	2.031365
21	0.3134794	0.2927672	1.7772524	1.7094142	0.5179274	2.0104366
22	0.299831	0.2799498	1.780096	1.693863	0.4844736	2.0400812
23	0.2900898	0.2735904	1.7423228	1.6849452	0.4752642	2.0743626
24	0.278718	0.2611016	1.8037318	1.6703374	0.4611574	2.0937096
25	0.274498	0.2560018	1.7691968	1.6515366	0.4530486	2.1143966
26	0.2641778	0.253709	1.7534058	1.6368734	0.4424514	2.114483
27	0.2616826	0.243542	1.74388	1.6109402	0.4375422	2.1713292
28	0.2605416	0.2349886	1.7594884	1.6037134	0.4203536	2.1450096
29	0.2461656	0.2269458	1.7335528	1.5892806	0.4193988	2.1459342
30	0.2397274	0.2248112	1.7500146	1.5702672	0.4049244	2.1929566
31	0.2327598	0.2198704	1.7406994	1.5633682	0.3956296	2.242732
32	0.2289896	0.2315526	1.7445658	1.4843254	0.393552	2.2432474

Table A.32: 10^7 operations, 2^8 bucket entries, 5% inserts, 5% removes and 90% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	4.236591	4.666774	5.7743806	5.0938962	6.3241098	7.8181554
2	2.683191	2.7051032	3.5036422	4.2641076	3.5642772	4.1382576
3	1.7632468	1.8474818	2.2196766	3.3246126	2.2959336	2.6320722
4	1.2460878	1.2536554	1.6967326	2.7807324	1.766227	2.1018806
5	1.0036766	1.104708	1.4562608	2.3153116	1.2954056	1.6323226
6	0.8162168	0.810319	1.2951472	2.121417	1.2705098	1.2712506
7	0.6794082	0.6887072	0.9641928	1.969569	0.9649684	1.1831474
8	0.596615	0.6025892	0.8979816	1.945255	0.8000866	0.9295846
9	0.5281688	0.5394254	0.8407482	1.8747694	0.7127166	0.8337904
10	0.4774874	0.4859856	0.8015744	1.8277804	0.642713	0.7482908
11	0.4348552	0.4410286	0.775421	1.8336744	0.5862846	0.6883092
12	0.4001752	0.4053394	0.75833	1.8123562	0.5438212	0.6284774
13	0.3720224	0.3755046	0.7340308	1.7464454	0.502861	0.5830152
14	0.3442628	0.3524866	0.7404246	1.7685526	0.4724538	0.5485222
15	0.327992	0.3308376	0.7137836	1.7413994	0.440273	0.5207712
16	0.3133768	0.3136496	0.7186182	1.7417976	0.4319516	0.4876018
17	0.294942	0.2995812	0.7142582	1.6787486	0.4249836	0.4761186
18	0.2809248	0.280359	0.7123462	1.6411454	0.4008822	0.4553346
19	0.265821	0.2719024	0.7056906	1.6019522	0.3844662	0.430647
20	0.262209	0.2615732	0.6932406	1.5796754	0.3734068	0.4256396
21	0.2449668	0.2472628	0.6817942	1.5665672	0.3557432	0.4087148
22	0.2356122	0.2378458	0.674813	1.543967	0.3414854	0.3921516
23	0.224961	0.2329924	0.674183	1.5331968	0.3302754	0.3816926
24	0.2176874	0.2264866	0.666337	1.5037722	0.3268528	0.3651922
25	0.2103768	0.2129356	0.6686194	1.4842492	0.3131662	0.3634052
26	0.2023738	0.2093966	0.6623828	1.4682726	0.3002166	0.3684014
27	0.1976618	0.2075146	0.6471186	1.467264	0.2966484	0.3399742
28	0.1994124	0.1965436	0.6411948	1.4499196	0.2907746	0.3454148
29	0.202436	0.1905942	0.6326472	1.4396186	0.28732	0.3371124
30	0.1836852	0.1949472	0.6266964	1.4274728	0.2868504	0.3249306
31	0.1857666	0.1897704	0.6157852	1.4205434	0.263591	0.32602
32	0.1907254	0.200546	0.6258258	1.4027988	0.2753438	0.3351478

Table A.33: 10^7 operations, 2^8 bucket entries, 10% inserts, 10% removes and 80% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	4.1680936	4.8522406	5.9875466	5.3022574	6.2823802	7.2139578
2	2.6783816	2.8135436	3.6189888	4.352719	3.858052	4.2103282
3	1.8424566	1.8794098	2.4253736	3.3266602	2.507453	2.8997158
4	1.409	1.4513716	1.8255684	2.6968546	1.827799	2.2187274
5	1.209006	1.2656446	1.566704	2.307283	1.4919254	1.737475
6	0.817812	0.833815	1.5116278	2.1406898	1.2944888	1.3363194
7	0.700748	0.7137122	1.166611	1.9980524	0.9477314	1.2131894
8	0.6098842	0.6252488	1.1214842	1.937992	0.838135	0.9711248
9	0.5439328	0.5608836	1.0474494	1.8498622	0.7518034	0.8666862
10	0.495814	0.5030062	0.9920086	1.8339656	0.6864238	0.7874214
11	0.4495006	0.4618338	0.9886536	1.8229052	0.6241174	0.723857
12	0.4152814	0.4312942	0.9703472	1.8183532	0.5686362	0.6757818
13	0.3817974	0.3996598	0.9820184	1.7762132	0.538811	0.6321782
14	0.3573956	0.3797572	0.9671558	1.756362	0.4973568	0.5901242
15	0.3365634	0.3443568	0.9646648	1.7706326	0.4634376	0.5624244
16	0.3229232	0.3283732	0.9549168	1.7384246	0.4529616	0.5385284
17	0.3047842	0.3064596	0.945652	1.7154026	0.4501256	0.5212102
18	0.2884532	0.2990398	0.9558872	1.7040628	0.4286058	0.5103068
19	0.2736332	0.2805918	0.9478494	1.6459862	0.4107028	0.4901278
20	0.2616858	0.2866534	0.9269404	1.6304816	0.398997	0.4943576
21	0.2565642	0.2578864	0.9408326	1.5952444	0.394423	0.4766996
22	0.241454	0.2511982	0.9114076	1.5783566	0.3663372	0.4725944
23	0.2327484	0.2407424	0.9288372	1.5451394	0.359146	0.4656732
24	0.2323934	0.2352196	0.9184332	1.5372852	0.345407	0.4719726
25	0.2230992	0.2280782	0.893406	1.5222764	0.3376392	0.473401
26	0.214245	0.2222124	0.8868284	1.510456	0.324133	0.4772222
27	0.2135196	0.2123928	0.8771998	1.5019798	0.3224162	0.4756676
28	0.2047424	0.201811	0.8717022	1.4917388	0.3065542	0.47769
29	0.1990828	0.1999778	0.8583688	1.4869384	0.3090064	0.4832972
30	0.1907216	0.1996806	0.8541928	1.4688918	0.3106006	0.4821788
31	0.2037882	0.2008578	0.8386178	1.4474192	0.2885412	0.4915404
32	0.1915256	0.1959522	0.8401958	1.4499408	0.289404	0.4881862

Table A.34: 10^7 operations, 2^8 bucket entries, 15% inserts, 15% removes and 70% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	5.0121898	5.412314	6.4602348	6.304394	6.660301	7.1980538
2	2.7875036	2.8444228	3.7414574	4.3606366	3.9233442	4.2524636
3	1.8950978	1.8695798	2.5178818	3.2681962	2.4897188	3.0296402
4	1.4295082	1.4686788	1.929539	2.6632388	1.8461232	2.281632
5	1.2813514	1.3109	1.6816778	2.3125106	1.5341162	1.8030146
6	0.8376798	0.8557966	1.6017444	2.1696586	1.3382828	1.479494
7	0.7163232	0.7317552	1.4338244	1.9431246	0.9855744	1.2171022
8	0.6263422	0.6418266	1.245221	1.90936	0.870352	1.1032202
9	0.5612874	0.5801364	1.2475194	1.8457562	0.7760948	0.9312064
10	0.5087478	0.5155272	1.2034256	1.8284268	0.7001154	0.8521464
11	0.4582996	0.4738984	1.2082502	1.804733	0.6443784	0.7829028
12	0.424554	0.433709	1.1760366	1.8028908	0.5981062	0.7436276
13	0.3940844	0.4087044	1.184322	1.7898642	0.5539284	0.702975
14	0.3687136	0.377	1.192696	1.782814	0.5190904	0.6997614
15	0.3516422	0.3613048	1.1819906	1.7754106	0.4826072	0.6746704
16	0.3280154	0.3328906	1.1778494	1.7953924	0.488288	0.6629478
17	0.3084282	0.3179214	1.20268	1.745064	0.4746352	0.6555488
18	0.296521	0.3072486	1.1744856	1.7133934	0.4463634	0.6627292
19	0.285778	0.2964374	1.1641042	1.665603	0.43426	0.6493154
20	0.276188	0.2793276	1.1698046	1.6265674	0.4174786	0.6565188
21	0.2699854	0.2766154	1.127184	1.610307	0.4037476	0.6648526
22	0.2540226	0.2544476	1.1291476	1.6041964	0.3919304	0.6714984
23	0.2441688	0.2487634	1.1185728	1.5824838	0.3831048	0.666235
24	0.2313876	0.2394942	1.1080274	1.5686148	0.3663428	0.6764782
25	0.2277858	0.2330984	1.1176488	1.5481432	0.3523514	0.675592
26	0.2249978	0.2249556	1.0782134	1.5238016	0.3467132	0.676397
27	0.2164452	0.2183546	1.0597994	1.519123	0.3334624	0.6996596
28	0.2087312	0.217731	1.0192798	1.5113436	0.3300454	0.6947276
29	0.2082154	0.2108966	1.0279824	1.5119986	0.316538	0.6980466
30	0.2137676	0.2071282	1.010386	1.5021656	0.3109098	0.7040684
31	0.1969468	0.2097122	0.9835992	1.4839542	0.3037568	0.7066814
32	0.196695	0.2055382	1.0033698	1.46953	0.3015884	0.7240268

Table A.35: 10^7 operations, 2^8 bucket entries, 25% inserts, 25% removes and 50% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	5.2789698	5.3551584	5.9334414	6.0255854	6.9459346	5.9906252
2	2.8488682	2.8526178	3.7242116	4.1372904	3.847759	3.7057574
3	1.9078266	1.942189	2.6044062	3.242065	2.5097204	2.6504388
4	1.5035692	1.5059244	2.13294	2.560327	1.9129158	2.0053192
5	1.117523	1.1439104	1.9016504	2.2569796	1.6151886	1.6667838
6	0.8774806	0.8866224	1.7653334	2.0641714	1.3055834	1.3720812
7	0.749185	0.7606902	1.6951292	1.9415906	1.1064728	1.2597464
8	0.6589918	0.6605006	1.630717	1.904835	0.915603	1.1886624
9	0.593633	0.5921806	1.529957	1.8994742	0.8255436	1.188798
10	0.528517	0.53578	1.5790286	1.8367464	0.7494512	1.1525734
11	0.4818678	0.4907444	1.450986	1.848086	0.6816822	1.077934
12	0.4451904	0.4496146	1.4986616	1.8047918	0.6359534	1.0918312
13	0.4098212	0.4213054	1.4640712	1.81853	0.5877026	1.0745442
14	0.3906612	0.3875866	1.4549254	1.789924	0.5650112	1.0509282
15	0.3658964	0.3725498	1.421006	1.7832776	0.5365526	1.0143444
16	0.3485334	0.3529778	1.5352216	1.7851454	0.4974274	1.0564774
17	0.3336554	0.3306348	1.4380654	1.7589736	0.510591	1.0378774
18	0.319338	0.3191458	1.4370146	1.7286888	0.4818126	1.0248308
19	0.29969	0.3085812	1.4209004	1.686716	0.4723144	1.090063
20	0.2920898	0.2907126	1.4135532	1.652137	0.4623372	1.1095572
21	0.2782776	0.282925	1.37455	1.6411474	0.4410716	1.1462752
22	0.2752412	0.2710506	1.3464548	1.6320182	0.4345232	1.1672022
23	0.2603344	0.270452	1.335922	1.6070922	0.4111582	1.130784
24	0.2574426	0.2542314	1.3361686	1.5968448	0.4000448	1.158781
25	0.2428232	0.2457572	1.3299968	1.583376	0.3860598	1.1395602
26	0.259785	0.2377836	1.313072	1.5802018	0.3745358	1.1716566
27	0.2323668	0.2312714	1.3097562	1.5670764	0.3677228	1.1559342
28	0.2237974	0.2276104	1.2967542	1.5490138	0.357778	1.1739274
29	0.21714	0.2187402	1.2669894	1.5348416	0.348102	1.1645884
30	0.211119	0.2230086	1.26199	1.4736854	0.3413944	1.162408
31	0.2148396	0.2118816	1.2516998	1.4550038	0.336792	1.1883218
32	0.2063044	0.214057	1.2455002	1.4501678	0.3354282	1.1833446

Table A.36: 10^7 operations, 2^8 bucket entries, 40% inserts, 40% removes and 20% searches

Threads	NF	OF	GPE	GPL	HHL	TBB
1	5.0213982	5.2663626	6.66529	6.2213748	7.06788	6.9939452
2	2.9129168	2.9286874	4.002558	4.3294136	4.1209506	4.2707554
3	2.0377632	1.971099	2.8877462	3.2991534	2.7816858	3.0321874
4	1.527124	1.5031714	2.4259212	2.6732542	2.0447932	2.3364484
5	1.2199698	1.197353	2.1031072	2.2504474	1.8123486	1.9032574
6	0.9189756	0.8946014	1.930646	2.1063584	1.3946622	1.922864
7	0.7890986	0.7734072	1.890995	1.9541122	1.212552	1.833678
8	0.6914098	0.6765556	1.866009	1.9219796	0.9945184	1.763282
9	0.6154144	0.6078512	1.8496218	1.866002	0.8797496	1.7598122
10	0.556837	0.5461914	1.8017968	1.8350188	0.7932732	1.6770264
11	0.5120384	0.5002228	1.7968712	1.8474322	0.7390274	1.6934878
12	0.4638336	0.4614386	1.8256684	1.830645	0.6737796	1.6791586
13	0.433111	0.4294	1.7842436	1.8370118	0.633372	1.6358228
14	0.4088638	0.3977422	1.7702866	1.8316332	0.5955722	1.6205802
15	0.3834396	0.3795878	1.7499298	1.8254744	0.5682956	1.6035698
16	0.3687942	0.364516	1.8328922	1.8297998	0.5609984	1.6090074
17	0.3534868	0.3357274	1.7861664	1.8098488	0.5438646	1.6024186
18	0.34262	0.3291274	1.7877928	1.7911712	0.5187684	1.6272776
19	0.3391016	0.3159988	1.7822836	1.7572192	0.504459	1.6499248
20	0.3147528	0.2984346	1.7409528	1.7355196	0.4976124	1.6835978
21	0.2982768	0.288546	1.7013634	1.7138268	0.4840456	1.6631294
22	0.2911766	0.2808176	1.7157932	1.7007072	0.4631522	1.679022
23	0.2764028	0.2688988	1.7060212	1.6779126	0.4532084	1.6990006
24	0.2721342	0.2655315	1.7010464	1.6618928	0.4403062	1.6842054
25	0.2611182	0.2607986	1.6804272	1.6405656	0.428054	1.7127142
26	0.2554958	0.246703	1.656121	1.6339596	0.418303	1.7451548
27	0.2478576	0.2441786	1.6620452	1.6033112	0.4046122	1.7554628
28	0.2466966	0.23157	1.6329274	1.6144392	0.4040752	1.7440672
29	0.2383152	0.2331128	1.6133238	1.5997136	0.3898418	1.8095746
30	0.2291792	0.2326542	1.5277332	1.5547482	0.3812344	1.7796302
31	0.227491	0.2177254	1.5106514	1.4976342	0.3799156	1.7992566
32	0.2251064	0.2274206	1.5599702	1.4894496	0.3723416	1.8503998

Bibliography

[1] Miguel Areias and Ricardo Rocha. A Lock-Free Hash Trie Design for Concurrent Tabled Logic Programs. *International Journal of Parallel Programming*, 44(3):386–406, June 2016.

[2] Miguel Areias and Ricardo Rocha. Towards a Lock-Free, Fixed Size and Persistent Hash Map Design. In *International Symposium on Computer Architecture and High Performance Computing*, pages 145–152. IEEE, 2017.

[3] Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the Anchor: Lightweight Memory Management for Non-blocking Data Structures. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 33–42. ACM, 2013.

[4] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In *IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 185–192. IEEE, 2010.

[5] Detlefs, David L. and Martin, Paul A. and Moir, Mark and Steele,Jr., Guy L. Lock-free reference counting. In *ACM Symposium on Principles of Distributed Computing*, pages 190–199. ACM, 2001.

[6] Jason Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *BSDCan Conference*, 2006.

[7] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004.

[8] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[9] Maurice Herlihy and Nir Shavit. On the Nature of Progress. In *Principles of Distributed Systems*, pages 313–328. Springer, 2011.

[10] Maurice Herlihy, Victor Luchangco, and Mark Moir. The Repeat Offender Problem: A Mechanism for Supporting Dynamic-sized Lock-free Data Structures. Technical report, 2002.

[11] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[12] Ricardo Leite and Ricardo Rocha. LRMalloc: a Modern and Competitive Lock-Free Dynamic Memory Allocator. In *International Meeting on High Performance Computing for Computational Science*, September 2018.

[13] Maged M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[14] Maged M. Michael and Michael L. Scott. Correction of a Memory Management Method for Lock-Free Data Structures. Technical report, Rochester University, NY, Department of Computer Science, 1995.

[15] Pedro Moreno and Ricardo Rocha. Reclaiming Memory from Lock-Free Hash Tries. In *10th INForum - Simpósio de Informática*, 2018.

[16] Pedro Ramalhete and Andreia Correia. Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 367–369. ACM, 2017.

[17] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[18] John D. Valois. Lock-free Linked Lists Using Compare-and-swap. In *ACM Symposium on Principles of Distributed Computing*, pages 214–222. ACM, 1995.