U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

PhD

3rd CYCLE

FCUP
2024

U.PORTO

Generic Lock-Free Memory Reclamation

Pedro Carvalho Moreno

FC

Generic Lock-Free Memory Reclamation

Pedro Carvalho Moreno
Doctoral Program in Computer Science
Department of Computer Science
Faculty of Sciences of the University of Porto
2024

U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

PhD

Generic Lock-Free Memory Reclamation

Pedro Carvalho Moreno

Thesis carried out as part of the Doctoral Program in Computer Science

Department of Computer Science

2024

Supervisor

Ricardo Jorge Gomes Lopes da Rocha, Associate Professor,
Faculty of Sciences of the University of Porto

Acknowledgements

I am extremely grateful to my advisor Ricardo Rocha for his continued assistance, guidance and encouragement. He was always available to help me solve problems and discuss new ideas, and always pointed me in the best possible direction when I was unsure of what path to take. He is not only the best advisor I could ask for, but also a very good friend I enjoy spending time with.

I am also very grateful to Miguel Areias for always being available to help, for the multiple contributions he made to this work, and for being a good friend.

I am grateful to Nachshon Cohen and Erez Petrank for providing the source code to their work that allowed me to build my work upon it, to Ricardo Leite for reviewing my contributions to his memory allocator, and to all the anonymous reviewers that reviewed my papers and provided constructive feedback.

I want to acknowledge the Fundação para a Ciência e Tecnologia (FCT) for funding this work, during 4 years, within the grant SFRH/BD/143261/2019, and the Center for Research and Advanced Computing Systems (CRACS) group of INESC TEC for the travel funding.

A warm thank you to all my colleagues from the lab that were always there to help and provide a great work environment, I wish you all the best.

To all my friends that helped me take my mind of work and have very enjoyable times, thank you very much to you all.

Finally, I am deeply thankful to my parents for all the help, love and support they always provided.

Thank you all for all the support.

Abstract

With the current tendency for CPUs to have increasingly more cores in each generation, being able to take advantage of such large amounts of computing resources becomes increasingly important. Data structures are a fundamental building block for most programs, and their properties are inherited by the programs that use them. Lock-freedom is an important property, that not only ensures immunity to deadlocks, live-locks and priority inversion, but also allows high efficiency, minimal synchronization and great scalability. However, lock-free data structures require additional constructs to manage memory, that are known as *Safe Memory Reclamation* (SMR) methods. It is key that these methods are as efficient as possible, both in performance and in memory usage, in order for programs to take advantage of the available computing resources.

In this thesis, we review how synchronization between cores and memory management happens from the CPU architecture level to the data structure level, going over the language constructs, operating system, memory allocator, SMR methods and progress guarantees. In particular, we focus on a lock-free data structure, called *Lock-Free Hash Tries* (LFHT), and we explore the viability of applying the LFHT data structure to real world programs and we show how to make it scalable to larger amounts of data. We also propose a redesign of the LFHT data structure that makes it simpler, compatible with more SMR methods and more cache friendly. As the new design puts additional pressure on the SMR method, we propose a novel memory allocator extension that allows to both simplify and improve the memory management capabilities of the state-of-the-art *Optimistic Access* (OA) SMR method, which can be generally applied to other lock-free data structures. To conclude, we apply our improved OA method to our new design of the LFHT data structure.

Experimental results, show that our contributions, either increase performance across the board or give new general capabilities to the system at no significant performance cost. And, in most cases, we are able to both increase performance and practicality.

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

Resumo

Com a tendência atual de CPUs terem cada vez mais núcleos a cada geração, ser capaz de aproveitar quantidades tão grandes de recursos computacionais torna-se cada vez mais importante. As estruturas de dados são um elemento fundamental para a maioria dos programas e, como tal, as suas propriedades são herdadas pelos programas que as utilizam. *Lock-freedom* é uma propriedade importante, que não só garante imunidade a *deadlocks*, *live-locks* e inversão de prioridade, mas também permite alta eficiência, sincronização mínima e boa escalabilidade. No entanto, estruturas de dados *lock-free* requerem construções adicionais para gerenciar a memória, que são conhecidas como métodos *Safe Memory Reclamation* (SMR). É fundamental que estes métodos sejam tão eficientes quanto possível, tanto no desempenho como no uso de memória, para que os programas aproveitem os recursos computacionais disponíveis.

Nesta tese, analisamos como a sincronização entre núcleos e a gestão de memória acontece desde o nível da arquitetura do CPU até o nível da estrutura de dados, passando pelas construções da linguagem, sistema operativo, alocador de memória, métodos SMR e garantias de progresso. Em particular, nos concentramo-nos numa estrutura de dados *lock-free*, chamada *Lock-Free Hash Tries* (LFHT), e exploramos a viabilidade de aplicar a estrutura de dados LFHT a programas do mundo real e mostramos como torná-la escalável para maiores quantidades de dados. Também propomos um redesenho da estrutura de dados LFHT que a torna mais simples, compatível com mais métodos SMR e mais amigável à *cache*. Como o novo design coloca pressão adicional no método SMR, propomos uma nova extensão ao alocador de memória que permite simplificar, e melhorar a gestão de memória do método SMR estado da arte *Optimistic Access* (OA), o qual pode ser aplicado a outras estruturas de dados *lock-free*. Para concluir, aplicamos o nosso método OA aprimorado ao nosso novo design da estrutura de dados LFHT.

Os resultados experimentais mostram que as nossas contribuições aumentam o desempenho de forma geral ou garantem novas capacidades ao sistema sem nenhum custo significativo de desempenho. Na maioria dos casos conseguimos aumentar o desempenho e a praticabilidade.

Palavras-chave: Recuperação de Memoria, Lock-Freedom, Estruturas de Dados Concorrentes, Hash Maps, Alocação de Memoria, Memoria Virtual

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

Most developments in computer hardware in modern times come increasingly from multithreading capabilities and less from single-threaded performance gains. To take advantage of multithreading, software needs to be designed with such capabilities in mind, starting from the basic building blocks of every program, the data structures. Most data structures used today in multithreaded systems rely on locking primitives in order to synchronize access, such locking primitives can make the process of converting a single-threaded data structure to a multithreaded one easier, but have multiple disadvantages, such as, creating live-locks, deadlocks or priority inversion [30]. Lock-free data structures and algorithms prevent such disadvantages [38], but are required to be designed from the ground up to be lock-free, and require additional support in order to reclaim memory [46], namely, *Safe Memory Reclamation* (SMR) methods.

There are multiple implementations of lock-free data structures, but most of them are not entirely usable in a lock-free manner, as they delegate the task of reclaiming memory to a garbage collector. This is a problem as it avoids portability to environments where a garbage collector is not available or, if available, it is not lock-free [72]. This leads to the loss of the overall lock-freedom property, as one of the pieces does not have the property. On the other hand, many memory reclamation schemes were developed for general lock-free data structures. However, some are not compatible with all lock-free data structures [14, 39, 56, 74, 83], or not lock-free themselves [3, 36], or depend on specific operating system or hardware implementations [16, 23, 27].

The memory reclamation of removed elements on a lock-free data structure is not as simple as in a lock-based data structure. To ensure lock-freedom, we need to allow concurrent access 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 moment it is being removed. Overcoming this limitation requires sophisticated methods to postpone, delegate and determine when the reclamation may occur. These methods should also offer some guarantees on performance and memory usage bounds while keeping the lock-freedom property. For example, if the memory reclamation method is not able to reclaim the removed elements (i.e., progress) at the same rate as they are removed from the data structure, we may end up with unbounded memory consumption. Another desirable property is being easy to apply, but as proved by Shefi and Petrank, an SMR method can not achieve

bounded memory usage (robustness), compatibility with most data structures (applicability) and ease of use at the same time. Only two of these three properties are possible at once for an SMR method [80].

The *Lock-Free Hash Tries* (LFHT) data structure is a hash trie data structure that implements a hash map in a lock-free manner. It was originally developed in Java, relying on the Java garbage collector in order to reclaim memory [6, 7]. As the LFHT data structure is incompatible with most standard SMR methods, in previous work, we have developed an SMR method, named *Hazard Hash and Level* (HHL) that allows the LFHT data structure to reclaim memory independently of the garbage collector [62].

Growing hash tables can have an advantage by always retaining a constant depth, however outside of scenarios mainly dominated by read operations, the growth of a hash table in a lock-free manner can be a severe bottleneck that destroys any performance advantage it has in relation to hash tries. Although the LFHT data structure is nearly ideal for a lock-free hash map implementation, it still retains some crucial flaws. The original LFHT implementation lacks the support for removing empty hash nodes, this support was added later by Areias and Rocha [8], but is incompatible with the HHL SMR method. Another flaw is its complexity, which is exacerbated by the added features like the removal of hash nodes. This complexity can be a barrier for its practical use in real world scenarios, as it requires a lot of care to implement correctly, and can be hard to maintain or add features.

The main purpose of this thesis is to continue to make lock-free data structures the standard tool for further concurrent applications. We will use the LFHT data structure as a case study in order to show that most of the problems can be solved or mitigated, and that lock-free data structures can be made desirable for wide adoption in multithreaded applications. A fundamental barrier to successfully achieve this goal is SMR, that although computer scientists have been working on a solution for more than 40 years, no ideal solution has been found yet, or is even possible [80]. As such, we strive for the best compromises in order to achieve a good practical solution.

1.1 Thesis Roadmap

We next describe the flow of the work done during this thesis and the resulting main contributions.

As our implementation of the LFHT data structure with the HHL SMR method showed great results [62], we chose to apply it to a real world application. For this we used the YAP Prolog system, as it is a language runtime with multithreading capabilities [66]. Multithreaded Prolog is able to perform concurrent computations, in which each thread runs independently but shares the program clauses. However, threads still require access to global data structures for symbol management. So, we replace the original atom table implementation in the YAP system with the LFHT implementation in conjunction with the HHL SMR method, in order to investigate whether an efficient and scalable symbol management can make a difference in a

multithreaded environment. Performance results showed that the new implementation has better results in single threaded execution and much better scalability in multithreaded execution than the original atom table.

The LFHT data structure has one important disadvantage. As the amount of data stored increases so does the depth of the internal hash levels, which reduces its performance. The size of the hash nodes that represent the hash levels can be increased in order to reduce the depth, but that comes at a cost of a relatively large memory overhead when it is used with small amounts of data. If the size of hash nodes is decreased however, with large amounts of data we see an increase in depth and consequential performance degradation at relatively minimal memory savings. To make the data structure dynamically adaptable, we developed a compression mechanism that allows the data structure to start with a small hash node size, and then dynamically combine multiple smaller hash nodes into larger ones, thus reducing the depth of the data structure. This happens when a hash node is fully populated by other hash nodes. In such a case, we combine the fully populated hash node and all its children into a single larger hash node and thus reduce the depth by one level. This allows the data structure to remain as efficient as possible both in memory usage and performance in scenarios with small and large amounts of data.

Another disadvantage of the LFHT data structure is the complexity of its expansion operation. This is not only the main cause behind its incompatibility with most SMR methods, but also hinders the development of features, such as, the compression mechanism mentioned above and an empty hash node removal mechanism developed by Areias and Rocha [8]. In order to simplify the expansion mechanism, we redesigned the LFHT data structure so that collisions on hash nodes do not form a linked list, but are instead stored in a simple array. This design simplification improves its performance, makes the data structure more cache friendly, by turning memory accesses more linear instead of random, and makes it compatible with most SMR methods.

The new LFHT design although more performant, places an additional burden on the SMR method. As it has to replace the whole array of collisions when performing an insertion or removal, it generates more garbage than the previous design. Thus, the efficiency of the SMR method becomes increasingly important. The state-of-the-art *Optimistic Access* (OA) SMR method provides high efficiency by the nature of being optimistic, but that means that it also needs to be able to read memory even after such memory was reclaimed. To be able to do so, the original OA design implements a recycling mechanism that allows memory to be reused, but never released to the memory allocator or operating system. Our proposal is to extend LRMalloc [48], a lock-free general purpose memory allocator, in such a way that we can guarantee memory allocations to be readable even after we free such allocations. We start by solving the problem at the memory allocator level, by adapting LRMalloc such that it does not release memory used by the OA method back to the operating system. This allows us to simplify the implementation of the OA memory reclamation method as we no longer need a recycling mechanism in order to manage the distribution of memory between threads. This task is now covered by the memory allocator as it was designed for this task in a general sense. We also gain the ability to reuse memory reclaimed by the OA method across the whole process. Then, to complete our solution,

and have the ability to release the memory used by the OA method to the operating system, we exploit how current operating systems/hardware use virtual memory in order to free the physical memory while keeping the virtual addresses valid. These novel extensions allow to both simplify and improve the memory management capabilities of the state-of-the-art OA method, which can be generally adopted and applied to support memory reclamation in other lock-free data structures.

To complete our work, we apply the OA method to the new design of the LFHT data structure. However, the new LFHT design shows some performance bottlenecks on the LRMalloc memory allocator. We solve these bottlenecks by reducing the thread cache size and at the same time implementing a raw memory cache and by applying a new addressing mode to the internal structures that prevents LRMalloc from touching memory before it is allocated by the user. These changes allow for the new design of the LFHT data structure in combination with the OA method using the allocator extension to outperform the original design with the HHL SMR method in most scenarios.

1.2 Thesis Outline

The remainder of this thesis is organized as follows.

In Chapter 2, we introduce the relevant background to this thesis, and review the state-of-the-art in SMR techniques.

In Chapter 3, we describe the LFHT data structure and the HHL method and then present all our work done on top of the LFHT data structure, such as, the application to YAP Prolog, the compression mechanism and the new simplified design.

In Chapter 4, we introduce the concepts of virtual memory and memory allocation, along with a more in depth description of the LRMalloc memory allocator and the OA SMR method. Then, we present our palloc extension to the LRMalloc memory allocator that allows our OA SMR method simplification, and we apply this simplified OA to the new LFHT design together with key improvements to the LRMalloc memory allocator.

Finally, in Chapter 5, we conclude the thesis and present future work directions.

1.3 Thesis Publications

The work done during this thesis resulted in the following publications. An extended version of our previous work on the HHL method was published in the *Journal of Parallel and Distributed Computing* (JPDC) [64]. Our work on applying the LFHT data structure to the YAP Prolog system was published in the proceedings of the 2023 *International Symposium on High-Level Parallel Programming and Applications* (HLPP) [65], and is currently submitted to the special

issue of the *International Journal of Parallel Programming* (IJPP). The compression mechanism for the LFHT data structure resulted in a publication in the proceedings of the 2020 *International European Conference on Parallel and Distributed Computing* (Euro-Par) [63]. The OA SMR method simplification with the palloc LRMalloc extension resulted in a publication in the proceedings of the 2023 *Symposium on Parallelism in algorithms and Architectures* (SPAA) [61]. In the near future, we plan to publish our new simplified LFHT design and a survey about the state-of-the-art SMR methods.

Chapter 2

Background

In this Chapter, we start by introducing the possible progress guarantees and what they ensure. Next, we present the most common atomic primitives and how they are implemented. Then, we discuss memory ordering in software and hardware by showing how modern compilers and CPUs can cause memory accesses to appear out of order, and how the C11 standard [42] can control such order. Finally, we will go over the ABA problem and show possible ways to avoid it.

2.1 Progress Guarantees

Concurrent algorithms can behave in different ways depending on the environment in which they are executed. In order to have some insight over what progress conditions we can expect from a given concurrent algorithm, classifications have been devised for them. These classifications allow us to easily grasp what kind of progress guarantees we can expect for each operation or for the algorithm as a whole depending on the scheduling environment in which they are executed. In what follows, we will start by defining minimal and maximal progress and scheduler requirements, and then we will show the different progress guarantee categories as defined by Herlihy and Shavit [38].

2.1.1 Minimal and Maximal Progress

For a concurrent algorithm to be useful it needs to ensure some kind of progress. For example, an algorithm that enters a deadlock state is not useful as it will never be able to finish its task. Similarly, an algorithm in which some threads are never able to make any progress, such threads are just as useless. This might not be a problem in the case of a thread pool, in which we only care about the progress of the computation as a whole and not which thread contributed to it. But in the example of a server where each thread is handling a client, we want all threads to make progress in a timely manner.

Minimal progress

Minimal progress is defined as if the system takes a sufficiently long finite number of steps, at least some threads have made progress [38]. This is useful if we need to ensure that the system makes progress as a whole but the individual progress from specific threads in a timely manner is not important.

Maximal progress

Maximal progress ensures that if the system takes a sufficiently long finite number of steps, every thread makes some progress [38]. This is the ideal and strongest guarantee that can be required when the progress of every individual thread in a timely manner is required.

2.1.2 Scheduler Dependence

The scheduler is the entity that determines in what order and for how long a thread executes, and as such it plays a major role in how the system progresses. Note that under a *benevolent scheduler*, that by definition, an algorithm that only provides minimal progress, in practice provide maximal progress. This is the reason why most implementations in the real world tend to only strive for minimal progress as the available schedulers tend to be benevolent.

A scheduler can also be considered *fair* if it ensures that every thread executes an infinite number of concrete steps [38]. This means that every thread is given the opportunity to do some work. A scheduler can further be considered *uniformly isolating* if any thread that takes an infinite number of steps, some of these steps are contiguous and not interleaved with any other thread [38]. Such a scheduling strategy can be achieved, at least with high probability, with an exponential back-off strategy [1], in which threads back-off until all but one are inactive.

2.1.3 Progress Categories

Now we will describe the different categories of progress, what they require and what they provide. We start by describing the blocking categories, namely deadlock-freedom and starvation-freedom, followed by the non-blocking properties clash-freedom, obstruction-freedom, lock-freedom and wait-freedom.

Deadlock-freedom

Deadlock-freedom ensures minimal progress in every execution under a fair scheduler, and maximal progress in some executions under a fair scheduler. This means that, at any time, there is at least one thread that can make progress, i.e., it is not blocked, and thus, a fair scheduler ensures that such thread will eventually execute and thus make progress [38]. Consider the

example in which all threads are contending on a single lock. Deadlock-freedom ensures that there is at least one thread that is not waiting on the lock, and that it is possible for any thread to acquire the lock, but it does not ensure that a thread that tries to acquire the lock will eventually succeed.

Starvation-freedom

Starvation-freedom ensures maximal progress in every execution under a fair scheduler [38]. This means that, at any point, there is at least one thread that can make progress and also means that every thread will reach a state in which it will be able to make progress. Considering again the example above, but implemented with a waiting queue, which would ensure that a thread that tries to acquire the lock will eventually be able to do so after every thread that arrived there earlier completed its critical section.

Clash-freedom

Clash-freedom ensures minimal progress in every execution under a uniformly isolating scheduler, and maximal progress in some executions under a uniformly isolating scheduler [38]. This property is mostly theoretical as it does not commonly happen in reality. It can be seen as a weaker version of obstruction-freedom.

Obstruction-freedom

Obstruction-freedom ensures maximal progress in every execution under a uniformly isolating scheduler [38]. This means that any thread is able to make progress as long as it is able to run without interference for sufficiently long. By the nature of being non-blocking, it also means that a stalled thread does not impede the progress of the system as a whole. An example of an obstruction-free algorithm would be one in which when two threads are performing a conflicting operation and end up being interleaved, both threads can fail and have to retry the operation, but when either of them is able to run in isolation, the operation succeeds.

Lock-freedom

Lock-freedom ensures minimal progress in every execution, and maximal progress in some executions [38]. This means that, independently of the scheduling used, it ensures that at least one thread makes progress in the system as a whole as long it is executed for sufficiently long, or that some threads failing to complete an operation means that there is at least one thread that succeeded.

Wait-freedom

Wait-freedom ensures maximal progress in every execution [38]. That means that independently of how threads are scheduled any thread is able to make progress if it is executed for sufficiently long. As such, wait-freedom is the strongest progress guarantee an algorithm or method can have.

2.1.4 Relations Between Categories

	Scheduler independent Non-blocking	Scheduler dependent Non-blocking	Scheduler dependent Blocking
Maximal progress	Wait-free	Obstruction-free	Starvation-free
Minimal progress	Lock-free	Clash-free	Deadlock-free

Table 2.1: Periodic table of progress

Table 2.1 shows all the progress guarantees in a periodic table like form. It divides rows into maximal and minimal progress, and columns based on scheduler dependence and blocking/non-blocking.

Any algorithm that only ensures minimal progress can easily be upgraded to its maximal progress counterpart either by relying on a benevolent scheduler or by employing a helping mechanism. As most OS schedulers are benevolent, the majority of algorithms in the real world tend to only ensure minimal progress e.g., the usage of simple locks over waiting queue locks, lock-free algorithms over wait-free ones, etc. Although, due to the benevolent schedulers they are executed under, they end up achieving maximal progress. A helping mechanism is another way to achieve maximal progress, in which every thread starts publishing the operation it is trying to accomplish and then helping other threads complete their operations before trying to complete its own.

A system having a certain progress guarantee does not mean that all its individual methods have the same or stronger progress guarantee. As an example, a lock-free system can be composed by only obstruction-free methods, as each method by itself can fail to progress forever as long as it happens due to the success of methods being executed by other threads. More specifically, in a lock-free data structure, the removal of a node can fail due to a concurrent insertion, and such a scenario can repeat indefinitely with the remove operation always failing.

2.2 Atomic Primitives

In order for CPU cores to synchronize between themselves in a shared memory environment they rely on memory operations. However, simple loads and stores are not always enough to perform more complex synchronization operations and, as a result, more powerful primitives were

developed. Their main characteristic is being atomic, which means that from the perspective of any other thread the primitives effects can only be observed as if the primitive was not executed at all or was completely executed from start to finish. Such primitives, that both read and write to memory, are known as *Read-Modify-Write* (RMW) primitives. Next, we will describe the most common atomic primitives used for implementing concurrent algorithms, along with the hardware instructions currently available to implement such primitives. Note that atomic instructions can take an additional memory ordering parameter that specifies the ordering restrictions for the instruction (memory ordering will be further explained in Section 2.3).

2.2.1 Load/Store

Atomic load and store operations are the most basic atomic operations that allow us to read and write respectively to an atomic variable. However, we still need to specify at the language level that such operations are atomic, as otherwise the compiler might for example split a load or store into multiple smaller loads or stores, e.g., load/store a 64 bit variable in 2 blocks of 32 bits, or even completely elide the operation if it determines that it is unnecessary from a sequential execution perspective. Listing 2.1 and 2.2 show how the atomic load and store primitives can be implemented assuming they are executed atomically. Note that the following Listings will use the types: (i) A for atomic types, (ii) C for the corresponding non-atomic type and (iii) M for the other argument in arithmetic operations, which corresponds to C for integer types and *ptrdiff_t* for atomic pointer types.

```
1 C atomic_load(A* obj){
2     return *obj;
3 }
```

Listing 2.1: Load assuming atomic execution

```
1 void atomic_store(A* obj, C desired){
2     *obj = desired;
3 }
```

Listing 2.2: Store assuming atomic execution

2.2.2 Load-Link/Store-Conditional

The *Load-Link/Store-Conditional* (LL/SC) is one of the most powerful primitives. A LL works like a regular atomic load but also marks the read location so that the next SC on the same location can only succeed if the value in the location never changed in between the two operations, in which case it works like an atomic store that returns true, otherwise does nothing while returning false. The interface is also sometimes extended with a *Validate* (VL) operation that verifies if a location marked by LL is still unchanged. The strongest version of LL/SC supports nesting

LL/SC operations with any other operation to be executed in between (including other LL/SC operations), which allows it to avoid the ABA problem (the ABA problem will be further discussed in Section 2.4). However, such a strong LL/SC implementation is unavailable in hardware and the alternative software implementations have a time and memory overhead [12, 15, 43]. In most RISC ISAs, a very weak version of LL/SC is implemented in hardware, but these implementations have some important limitations: they do not allow nesting; they heavily restrict the type and/or number of operations that can be done between the LL and SC; and operations on adjacent memory locations spuriously break the link between the LL and SC. These limitations render the LL/SC pair so weak that it becomes vulnerable to the ABA problem and is only useful for the sake of implementing the compare and swap operation described next.

2.2.3 Compare and Swap

The atomic *Compare And Swap* (CAS) operation replaces a value in memory with a desired value if it was equal to a provided expected value, in which case it returns true, and if the value in memory differs from the expected value it does nothing while returning false. Listing 2.3 shows how the compare and swap operation could be implemented if it was executed atomically.

```
1 bool atomic_compare_and_swap(A* obj, C expected, C desired){
2     if(*obj == expected){
3         *obj = desired;
4         return true;
5     }
6     return false;
7 }
```

Listing 2.3: Compare and Swap assuming atomic execution

However, in practice an atomic *Compare And Exchange* (CAE) is what is commonly used, which can be seen as a slightly better interface for CAS. It differs from CAS in the fact that the expected value is passed as a reference and in the case of failure it is updated to the value present in the target memory location. Listing 2.4 shows how it could be implemented if executed atomically.

```
1 bool atomic_compare_and_exchange(A* obj, C* expected, C desired){
2     if(*obj == *expected){
3         *obj = desired;
4         return true;
5     }
6     *expected = *obj;
7     return false;
8 }
```

Listing 2.4: Compare and Exchange assuming atomic execution

From this point onwards, we will use the term CAS to mean either of the implementations as they are equivalent in almost all scenarios, and we will use the term CAE only when the usage of the CAE implementation is relevant. CAE actually has two versions, a strong and a weak one. The weak version is allowed to spuriously fail (fail even if the target memory location contained a value equal to the desired value). This version is meant to be used only when the operation is being used in a retry loop and in such situations can provide better performance in some implementations as we will see next.

CAS is usually implemented using one of two ways: (i) usually in CISC ISA's by using an instruction that performs equivalently to CAS, or (ii) usually in RISC ISA's by using LL/SC instructions, that start by performing a LL on the target memory and comparing the loaded value with the expected value and if they are equal trying to SC the desired value to the memory location. In case of failure, the strong versions of CAE retry if after an SC failure the memory value is still equal to the expected value.

```
1  bool atomic_compare_and_exchange_strong(A* obj, C* expected, C desired){
2      do{
3          C tmp = LL(obj);
4          if(tmp == *expected && SC(obj, desired)){
5              return true;
6          }
7          tmp = atomic_load(obj);
8      }while(*expected == tmp);
9      *expected = tmp;
10     return false;
11 }
```

Listing 2.5: Strong Compare and Exchange implemented with LL/SC

Listing 2.5 shows how the strong version of CAE can be implemented with LL/SC. The retry is necessary because the value being the same does not imply that it remained unchanged since the LL was performed as it could be a value A during the LL then change by another thread from A to B and from B to A before the SC operation. This is one example of an ABA problem that we will go into more detail in Section 2.4.

```
1  bool atomic_compare_and_exchange_strong(A* obj, C* expected, C desired){
2      C tmp = LL(obj);
3      if(tmp == *expected && SC(obj, desired)){
4          return true;
5      }
6      *expected = tmp;
7      return false;
8  }
```

Listing 2.6: Weak Compare and Exchange implemented with LL/SC

Listing 2.6 shows how the weak version of CAE can be implemented with LL/SC. The weak version of CAE can fail spuriously and as such is intended to replace the strong version when used in a retry loop and the spurious failures are of no consequence. It results in better performance in architectures that rely on hardware LL/SC instead of CAS.

2.2.4 Double Width Compare and Swap

The *Double Width Compare and Swap* (DWCAS) operation works exactly as the regular CAS operation, but can operate on two consecutive and aligned machine words, while the regular CAS can only operate on a single machine word. Its main use case is ABA prevention by using the extra space for a monotonic tag, although at a cost of extra memory usage. It is usually only available in CISC architectures, such as x86, that directly implement it in hardware, however ARM implements a double width version of LL/SC that allows the implementation of DWCAS.

2.2.5 Double/Multi Compare and Swap

Double Compare and Swap (DCAS), not to be confused with DWCAS, allows us to perform a CAS operation on two arbitrary memory locations, i.e., it only succeeds if both locations contain the respective expected values, and when it succeeds both locations are changed atomically. There are some hardware implementations of DCAS, such as m68k, however they are exceedingly rare and there is no modern/mainstream ISA that implements this primitive. The *Multi Compare and Swap* (MCAS) is a more generic version of DCAS that can operate on more than two arbitrary locations, but unlike DCAS there are no hardware implementations of MCAS. Currently, the usage of either DCAS or MCAS need to rely on software implementations that like LL/SC require a time and memory overhead.

2.2.6 Exchange

The atomic exchange operation performs an atomic store while also returning the value that the store replaced. Listing 2.7 shows how it could be implemented if executed atomically.

```
1 C atomic_exchange(A* obj, C desired){
2     C tmp = *obj;
3     *obj = desired;
4     return tmp;
5 }
```

Listing 2.7: Exchange assuming atomic execution

The implementation of exchange can either be done through an exchange hardware instruction, implemented with CAS, or directly with LL/SC. Listing 2.8 shows how exchange can be implemented with CAE and Listing 2.9 shows how it can be implemented with LL/SC.

```
C atomic_exchange(A* obj, C desired){
    C tmp = atomic_load(obj);
    while(!atomic_compare_and_exchange_weak(obj, &tmp, desired)){}
    return tmp;
}
```

Listing 2.8: Exchange implemeted with CAE

```
C atomic_exchange(A* obj, C desired){
    C tmp;
    do{
        tmp = LL(obj);
    }while(!SC(obj, desired);
    return tmp;
}
```

Listing 2.9: Exchange implemeted with LL/SC

2.2.7 Fetch and Op

Atomic Fetch and Op operations are a group of operations that perform an operation directly on a memory location and return the previous value, for example, *Fetch And Add* (FAA) atomically adds a number to a memory location returning the number in memory before the addition. Listing 2.10 shows how FAA could be implemented if executed atomically.

```
C atomic_fetch_add(A* obj, M arg){
    C tmp = *obj;
    *obj += arg;
    return tmp;
}
```

Listing 2.10: FAA assuming atomic execution

These operations are implemented in hardware in the most popular architectures, but they can also be implemented using CAE as shown in Listing 2.11 or LL/SC as shown in Listing 2.12.

```
C atomic_fetch_add(A* obj, M arg){
    C new,
      old = atomic_load(obj);
    do{
        new = old + arg;
    while(!atomic_compare_and_exchange_weak(obj, &old, new));
    return old;
}
```

Listing 2.11: FAA implemeted with CAE

```
1 C atomic_fetch_add(A* obj, M arg){
2     C new,
3       old;
4     do{
5         old = LL(obj);
6         new = old + arg;
7     }while(!SC(obj, new));
8     return old;
9 }
```

Listing 2.12: FAA implemeted with LL/SC

2.3 Memory Ordering

Throughout most of the history of computing CPUs were single threaded, and even today most of the code executed is still single threaded. As such, both compilers and CPUs are extremely aggressively optimized for this use case. This means that both compilers and CPUs, unless told otherwise, take the program that is given and heavily modify it to achieve the best possible performance with the only restriction that the actual execution produces the same result that the code specifies assuming a single threaded execution. Even though modern CPUs are able to execute multiple instructions in a single thread per clock cycle, memory access is still in the order of 100's of clock cycles. As such, optimizations around memory operations are extremely important, both the compiler and the CPU try to elide as much memory operations as possible and even reorder them, so the CPU can keep doing work while waiting for memory or cache. In what follows, we will explore how compiler level optimizations can foil expectations in parallel environments and then how CPU level optimizations can cause memory accesses to appear out of order.

2.3.1 Compiler Level Reordering

First, we will describe the optimizations a compiler can make to memory accesses by assuming single threaded execution.

Access Tear

Access tear is when a memory access of a given size is divided in multiple accesses of smaller size. An example would be the replacement of a 64 bit load with 8 loads of 8 bits each. Such a replacement would not alter the behavior of the program in a single threaded scenario, thus the compiler is allowed to do so if not explicitly told not to.

Load Fuse

A load can be fused when a compiler still has the result from a previous load of the same address, and if in between the two loads there are no operations that modify the address it can assume the second load is unnecessary and remove it. Note that it does not take into account that another thread might have modified the address concurrently as it assumes single threaded execution.

Store Fuse

Similarly to load fuse, when there are two stores to the same address with no loads in between the compiler can eliminate the first store as it will be overwritten anyway.

Access Reorder

The compiler is also free to reorder any memory accesses as long as it keeps the dependencies between accesses ordered, but when there are no dependencies between two memory accesses they can always be reordered.

Invented Loads

The compiler is also allowed to invent loads. For example, this can happen if under register pressure it has to lose a previously loaded value that it will need in the future. In such a case, instead of having to store and load it back from the stack, the compiler might just load the value a second time (the invented load) from where it was previously loaded.

Invented Stores

Like loads, stores can also be invented, as a location that will be the target of a store in the future could be used as temporary storage until then. As in the previous example, it could be used to temporarily store the value of a register when under pressure.

2.3.2 CPU Level Reordering

Here, we will show the high level design of modern multicore CPUs and how certain design decisions can lead to memory accesses being perceived out of order.

Cache Structure

As stated before, main memory accesses can take 100's of CPU cycles while multiple instructions can execute in a single clock cycle. One of the first improvements to this problem was the usage of caches that constitute very small amounts of memory that can be accessed in a single clock cycle or less. Such caches can store copies of data that is frequently loaded from memory in order to vastly speed up such loads, and they can also keep the data stored to memory while it is being propagated to memory so that the CPU can continue execution without having to wait for the store to complete. As increasing the size of the cache tends to make its access slower, modern CPUs employ multiple levels of cache, usually a level 1 (L1) cache that serves a single core and has the lowest access time (usually 1 clock cycle or less) but is also the smallest, and then as we go further in the levels L2, L3 or even L4 they tend to be shared among more cores and have higher access times, but also be much larger. In a CPU with cache, the communication with memory is delegated to the cache, and happens in fixed size blocks, called cache lines, that can range from 16 to 256 bytes. Figure 2.1 shows a simple view of the architecture of a dual-core CPU with only two L1 caches, one per core, that we will use as further reference.

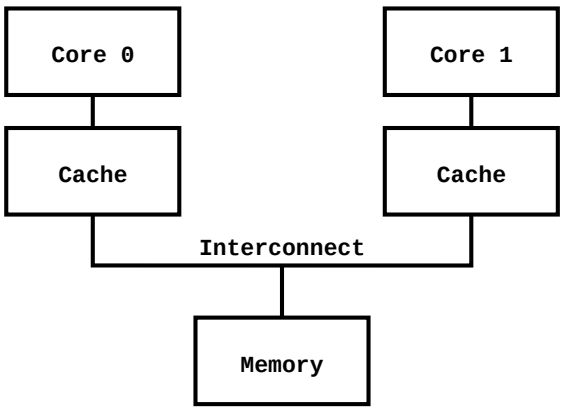


Figure 2.1: Cache structure

MESI Protocol

The main problem arises when we have multiple cores and need to keep their caches somewhat synchronized. To solve this problem a communication protocol needs to be used. These protocols can be extremely complex, but for the sake of simplicity and understanding of the problem, we will see how the MESI protocol [37] works. MESI stands for the 4 possible states a cache line can be, which are: *modified*, *exclusive*, *shared* and *invalid*.

A cache line is in a *modified* state when it has been written to by the core and is the only cache that has the most up-to-date value. This means that no other cache line can have a valid copy of that line, and when that cache line needs to be replaced it has to be written back to memory first.

The *exclusive* state, like the *modified* state implies that no other thread can have a valid copy, but the contents of the cache line are the same as memory. When a cache line is loaded from memory, it usually starts in this state as it is requested by a core. As a line in this state is up-to-date with memory it might at any point be discarded without having to write back to memory.

When a line is in a *shared* state, it might be replicated in another core cache, so at this point the core is permitted to read it but needs to change it to exclusive before performing any write. As with the *exclusive* state, the line is up-to-date with memory, so it can be discarded without writing it back to memory.

An *invalid* state means that the data in the cache line can no longer be considered valid, and as such the line can be seen as empty. Cache lines in an *invalid* state are the ideal target to be used for new data from memory as nothing will have to be discarded to do so.

In order to change between the four states, the caches and memory need to communicate, and to do so they use the following messages:

Read The read message contains the address the cache line intends to obtain the value of.

Read Response the read response message contains the data requested by a previous read message, and can originate from either memory or another cache.

Invalidate The invalidate message contains an address for a cache line that all other threads must invalidate.

Invalidate Acknowledge The invalidate acknowledge message confirms to the other cache that the requested invalidate was performed.

Read Invalidate The read invalidate message is a combination of the read and invalidate messages, that requests the data of an address and at the same time asks all other caches to invalidate their entry.

Writeback The writeback message contains the address and the data that is to be written to memory and is used to free space in modified lines.

Now we will describe all the state transitions between the MESI states:

Modified to Exclusive The cache sends a writeback but keeps the cache line.

Modified to Shared Another core attempts to read the line, so this cache receives a read message which causes it to send a writeback and a read response.

Modified to Invalid Another core attempts to write to this line so this cache receives a read invalidate, to which it has to send a writeback, a read response and an invalidate acknowledge.

Exclusive to Modified The core writes to the cache it already had exclusive access to, so no messages are traded.

Exclusive to Shared Another core attempts to read the line, so this cache receives a read message to which it responds with a read response making the line available in more than one cache.

Exclusive to Invalid Another core attempts to write to this line so this cache receives a read invalidate, to which it responds with both a read response and invalidate acknowledge.

Shared to Modified The core attempts to write to the cache line, but first the cache needs to send an invalidate and wait for the invalidate acknowledge responses.

Shared to Exclusive The core will soon perform a write to the cache line, so it sends an invalidate, and after receiving all invalidate acknowledge messages the line becomes exclusive.

Shared to Invalid Another core attempts to write to this line, so this core receives an invalidate message to which it responds with an invalidate acknowledge.

Invalid to Shared The core attempts to read the line, so the cache sends a read message to which another cache responds with a read response.

Invalid to Exclusive The core attempts to read the line, so the cache sends a read message to which only the memory responds leaving this cache as the only owner.

Invalid to Modified The core attempts to write the line, so the cache sends a read invalidate message to which all other threads must respond with a read response if they have the value and always with an invalidate acknowledge.

Store Buffers

One of the main issues with the MESI protocol is that when a store is performed and the cache line is not in the exclusive state, it requires the cache to send an invalidate or read invalidate message and wait for all the invalidate acknowledge before proceeding with the store. The adopted solution to this problem is the addition of store buffers, which allows the core to continue execution while the cache is waiting for all the invalidate acknowledges. Figure 2.2 shows the new layout of the CPU.

However, the simple addition of store buffers is not enough as it would foil even sequential execution. Take as an example the following scenario in Listing 2.13 in which a and b start as 0. The store in line 1 would be buffered and when a in line 2 is read it would obtain the previous value of a still in the cache (1) resulting in the final value of b being 1 instead of 2.

This would be incorrect even in single threaded execution, which would not be acceptable. As such store forwarding needs to be performed, in which the core checks its own store buffer before consulting the cache as we show in Fig. 2.3.

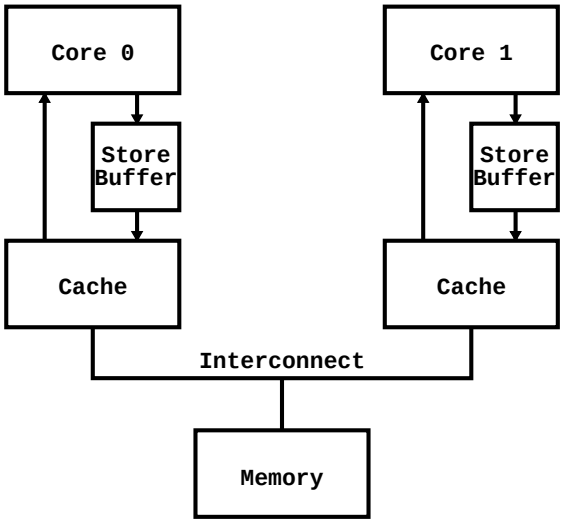


Figure 2.2: Caches with store buffers

```
1 int a = 1;
2 int b = a + 1;
3 assert(b == 2);
```

Listing 2.13: Store Buffer problem

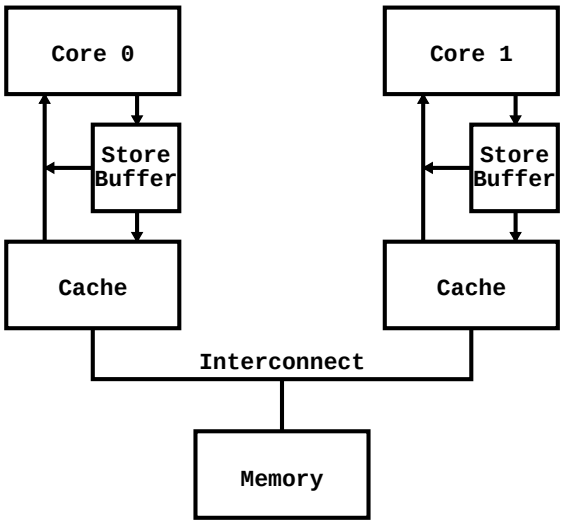


Figure 2.3: Caches with store forwarding

Store forwarding allows for the single threaded execution to work as expected, but in a multithreaded scenario memory accesses can appear to occur out of order. Take the code in Listing 2.14 for example, both cores perform the store that remains in their store buffer, and then perform the load from cache not seeing each other's stores as they remain in the buffer, resulting in both loads getting the value of 0. This requires an instruction to be introduced after the store in order to tell the core to flush its store buffer when we want to observe the behavior of the core as sequentially consistent. Such an instruction is usually known as a full memory barrier. In summary, store buffers allow for stores to be reordered after loads as shown in Table 2.2. Note that in the following litmus test listings we will use the explicit versions of *atomic_load* and *atomic_store* with *memory_order_relaxed* in order to specify no memory ordering requirements but still ensure the memory accesses happen and happen atomically. In Section 2.3.3, we will describe the explicit versions of the atomic primitives and memory model of the C11 standard [42]. As here we want to show the effects of the hardware optimizations, we assume the compiler performs no memory access reordering in the following litmus tests. The *exists* clause is meant to show correct but unexpected behavior caused by the hardware optimizations. Memory barriers can be used to prevent such unexpected behavior.

```
1  int x = 0;
2  int y = 0;
3
4  P0(int *x, *y){
5      atomic_store_explicit(y, 1, memory_order_relaxed);
6      r1 = atomic_load_explicit(x, memory_order_relaxed);
7  }
8
9  P1(int *x, *y){
10     atomic_store_explicit(x, 1, memory_order_relaxed);
11     r2 = atomic_load_explicit(y, memory_order_relaxed);
12 }
13
14 exists(r1==0 && r2==0);
```

Listing 2.14: Litmus test: store buffer

A core might also be able to bypass the store buffer if the cache line is in the exclusive or modified state as we show in Fig. 2.4, this removes some pressure from the store buffer, but allows for stores to be reordered between themselves as shown in Listing 2.15. In this example, in $P0$, x goes through the write buffer while y is directly written to the cache, and then $P1$ immediately sees the new value 1 of y and then sees the old value 0 of x because it still resides in the store buffer in $P0$. The solution in such a case is a lighter memory barrier, also known as a write memory barrier, that only ensures ordering between stores, and to do so, forces all the stores to go through the buffer until all stores that preceded the barrier have left the store buffer. In summary, store buffers bypass allow for stores to be reordered after stores as shown in Table 2.2.

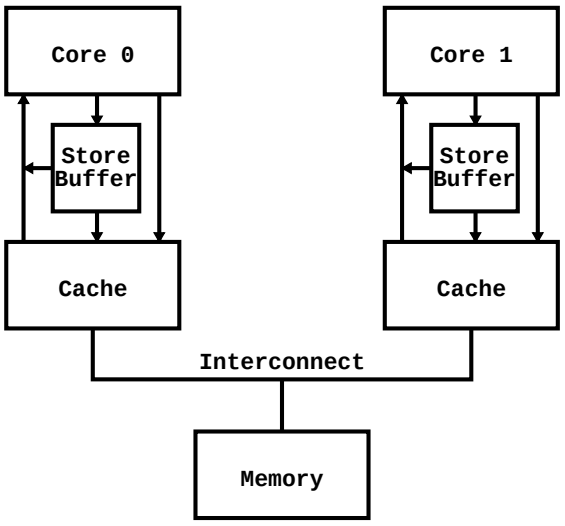


Figure 2.4: Caches with store buffer bypass

```
1  int x = 0;
2  int y = 0;
3
4  P0(int *x, *y){
5      atomic_store_explicit(x, 1, memory_order_relaxed);
6      atomic_store_explicit(y, 1, memory_order_relaxed);
7  }
8
9  P1(int *x, *y){
10     r1 = atomic_load_explicit(y, memory_order_relaxed);
11     r2 = atomic_load_explicit(x, memory_order_relaxed);
12 }
13
14 exists(r1==1 && r2==0);
```

Listing 2.15: Litmus test: store buffer bypass

In conclusion, the store buffers allow the core to perform stores without a synchronization cost as long as such stores do not have ordering requirements with subsequent loads, and the store buffer is not completely filled, in which case it needs to revert to wait for invalidation acknowledge messages.

Invalidate Queues

Another way to improve the time it takes from sending an invalidate or read invalidate to receiving an invalidate acknowledge from all caches is to have the caches respond faster to the invalidate messages. This is not easy because a cache might receive invalidates from all other caches at the same time, and it still needs to process the requests from its core while responding to the

invalidate requests that require it to make sure the cache line is invalidated before responding.

One way to make this improvement possible is to use invalidate queues. With a invalidate queue, a cache can immediately respond with an invalidate acknowledge as soon as it places the invalidate message in the queue. Putting the invalidate message in the queue serves as a promise that the cache will not send any MESI protocol messages related to the addresses in the queue until the entry in the queue has been processed. Figure 2.5 shows the cache architecture overview with invalidate queues.

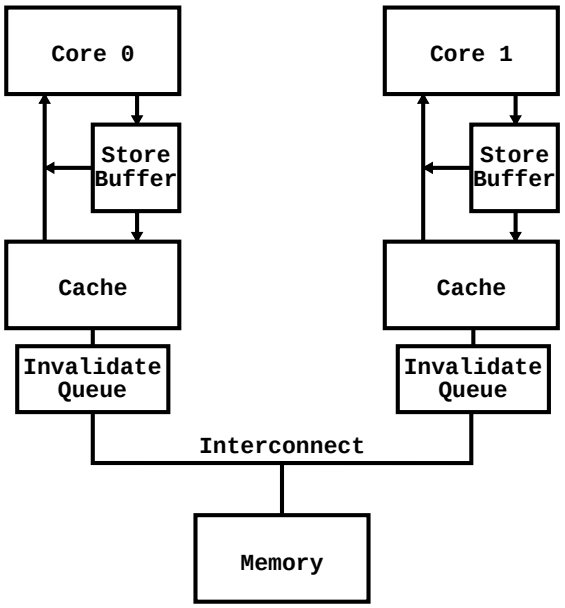


Figure 2.5: Caches with invalidate queues

Even thought the invalidate queue prevents messages to be sent related to the addresses in the queue, it does not prevent the core from loading from cache lines that have unprocessed invalidation messages in the queue. As we can see in Listing 2.16 this leads to memory reordering. In this case, $P1$ could have x in its cache and when $P0$ writes 1 to x it sends an invalidate that $P1$ puts in the queue and responds with an invalidate acknowledge. $P0$ then proceeds to flush its write buffer and write 1 to y that could be in its cache as exclusive already. Finally, $P1$ sends a read message for y and receives a read response with the value 1 and then unaware that an invalidate message for x is still in the queue it reads the old value 0 from its cache for x.

In order to ensure loads are performed in the expected order, another memory barrier needs to be used, that will make sure all the invalidates in the queue when the memory barrier is executed are processed before the next instruction, this memory barrier is usually known as a read memory barrier. In summary, invalidate queues allow for loads to be reordered after loads as shown in Table 2.2.

```
1  int x = 0;
2  int y = 0;
3
4  P0(int *x, *y){
5      atomic_store_explicit(x, 1, memory_order_relaxed);
6      write_memory_barrier();
7      atomic_store_explicit(y, 1, memory_order_relaxed);
8  }
9
10 P1(int *x, *y){
11     r1 = atomic_load_explicit(y, memory_order_relaxed);
12     r2 = atomic_load_explicit(x, memory_order_relaxed);
13 }
14
15 exists(r1==1 && r2==0);
```

Listing 2.16: Litmus test: invalidate queue

Cache Coherence

A system can be considered cache coherent if all its threads observe all changes to a location in the same order. Note that being cache coherent applies to changes/accesses, of the same size and aligned, to a single location, as observing changes in the same order to different locations, even if overlapping, is a different property, and can lead to different results on a cache coherent architecture [28]. In listing 2.17, we show a possible behavior in a non-cache coherent architecture. Note that this can only happen with very aggressive hardware optimizations, such as load reordering at the decoding stage in the CPU effectively having the CPU executing the loads in a different order.

While most architectures are cache coherent, for example the Itanium architecture does not have this property and the exists clause on the litmus test could trigger[1]. The terms *single-variable sequential consistency* and *single-copy atomicity* can also be used to describe cache coherence.

Multicopy Atomicity

Multicopy atomicity is the extension of cache coherence to multiple locations, so in a fully multicopy atomic system all threads see all stores, even if to different locations, in the same order. Unlike cache coherence, the simple addition of store buffers to a hardware implementation can be enough to lose the multicopy atomicity property, which makes it a very hard property to achieve while attaining good performance. As such, implementations that strive for multicopy atomicity usually achieve a weaker version called other-multicopy atomicity, that excludes the

[1]This is actually prevented in the C11 standard, and thus the litmus test, on Itanium, because even a relaxed atomic load, causes a memory barrier to be emitted preventing the loads from being reordered.

```
1  int x = 0;
2
3  P0(int *x){
4      atomic_store_explicit(x, 1, memory_order_relaxed);
5  }
6
7  P1(int *x){
8      atomic_store_explicit(x, 2, memory_order_relaxed);
9  }
10
11 P2(int *x){
12     r1 = atomic_load_explicit(x, memory_order_relaxed);
13     r2 = atomic_load_explicit(x, memory_order_relaxed);
14 }
15
16 P3(int *x){
17     r3 = atomic_load_explicit(x, memory_order_relaxed);
18     r4 = atomic_load_explicit(x, memory_order_relaxed);
19 }
20
21 exists(r1==1 && r2==2 && r3==2 && r4==1);
```

Listing 2.17: Litmus test: non-cache coherence

threads doing the stores, meaning that every other thread apart from the ones doing the stores, observe the stores in the same order. The weaker other-multicopy atomicity has the benefit of allowing store buffers, as only the thread doing a store can see it in a different order due to store forwarding. Listing 2.18 shows an example where the exists clause can trigger in non multicopy atomic systems.

One example where the exists clause may trigger is if for example some threads share a cache and write buffer, as exemplified in Fig. 2.6. Consider that $P0$, $P1$ and $P2$ in Listing 2.18 are mapped to core 0, core 1 and core 2 in Fig. 2.6.

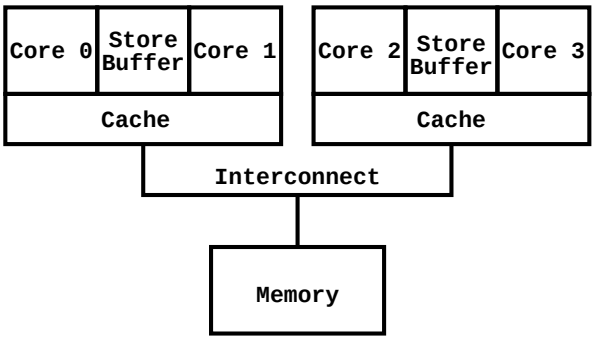


Figure 2.6: Shared store buffers

In such an example, the write by $P0$ goes through the write buffer as x is not in the cache.

```
1  int x = 0;
2  int y = 0;
3
4  P0(int *x){
5      atomic_store_explicit(x, 1, memory_order_relaxed);
6  }
7
8  P1(int *x, int *y){
9      r1 = atomic_load_explicit(x, memory_order_relaxed);
10     atomic_store_explicit(y, r1, memory_order_relaxed);
11 }
12
13 P2(int *x, int *y){
14     r2 = atomic_load_explicit(y, memory_order_relaxed);
15     read_memory_barrier();
16     r3 = atomic_load_explicit(x, memory_order_relaxed);
17 }
18
19 exists(r1==1 && r2==1 && r3==0);
```

Listing 2.18: Litmus test: non-multicopy atomicity

Then, $P1$ sees the store of 1 to x done by its sibling $P0$ earlier by checking their shared store buffer, and as such, sees the new value 1 of x. Next, $P1$ stores 1 to y by bypassing the store buffer by the nature of y being in the cache, $P2$ then reads the new value 1 of y and the old value 0 of x because the write of 1 to x is still in the shared store buffer of $P0$ and $P1$. Note that flushing the invalidate queue in $P2$ can have no effect as the store to x might still be in the store buffer and as such not yet have triggered the invalidate message to be sent. In order for the previous example to never trigger the exists clause even in non-multicopy atomic architectures the following barriers shown in Listing 2.19 would have to be used. They work because they are cumulative, which means that the release in $P1$ makes the write to x visible if it sees it, and the acquire in $P2$ ensures that the second load is ordered after it. In Section 2.3.3, we will go into more detail on acquire and release semantics.

Architecture Overview

As seen until now modern CPU's can employ a variety of strategies to improve their performance at the cost of reordering memory operations, however the kind of reordering they are allowed to do needs to be specified by the ISA, which ensures consistency between different CPU implementations. Note that even if the ISA allows it, a specific implementation may not perform some reordering, in other words an implementation may be stricter than what the specification allows. Table 2.2 adapted from the work by McKenney [52], shows what reordering some of the most common ISA's allow.

```
1  int x = 0;
2  int y = 0;
3
4  P0(int *x){
5      atomic_store_explicit(x, 1, memory_order_relaxed);
6  }
7
8  P1(int *x, int *y){
9      r1 = atomic_load_explicit(x, memory_order_relaxed);
10     atomic_store_explicit(y, r1, memory_order_release);
11 }
12
13 P2(int *x, int *y){
14     r2 = atomic_load_explicit(y, memory_order_acquire);
15     r3 = atomic_load_explicit(x, memory_order_relaxed);
16 }
17
18 assert(r2==0 || r3==1);
```

Listing 2.19: Litmus test: non-multicopy atomicity fixed

	Alpha	ARMv7	ARMv8	Itanium	MIPS	POWER	x86	z
Loads Reordered After Loads	Y	Y	Y	Y	Y	Y		
Loads Reordered After Stores	Y	Y	Y	Y	Y	Y		
Stores Reordered After Stores	Y	Y	Y	Y	Y	Y		
Stores Reordered After Loads	Y	Y	Y	Y	Y	Y	Y	Y
Atomics Reordered With Loads/Stores	Y	Y	Y		Y	Y		
Dependent Loads Reordered	Y							
Dependent Stores Reordered								
Non-Sequentially Consistent	Y	Y	Y	Y	Y	Y	Y	Y
Non-Multicopy Atomic	Y	Y	Y	Y	Y	Y	Y	
Non-Other-Multicopy Atomic	Y	Y		Y	Y	Y		
Non-Cache Coherent				Y				

Table 2.2: Memory ordering by ISA

2.3.3 C11 Memory Model

The C11 standard [42] introduces concurrency to the language and along with thread manipulation routines that somewhat replace their POSIX counterpart, it also introduces atomic operations and memory ordering semantics that previously had to rely on compiler intrinsics or assembly implementations. In the default configuration of the atomic operations introduced in the C11 standard, a memory ordering does not need to be specified and defaults to sequential consistency, which keeps code easy to reason about as it ensures total ordering between all atomic instructions used in the default configuration. However, as demonstrated before, ensuring total ordering can have a significant performance cost, as it requires the usage of memory barriers, that force store buffers and invalidate queues to constantly flush. To allow the mitigation of this cost, the C11 standard also includes explicit variants of all the atomic operations in which the memory ordering requirements can be specified, and also provides a fence function ($atomic\_thread\_fence()$) that provides a memory barrier without an associated atomic operation. Next we will describe the acquire and release, and sequential consistency semantics and then all the possible orderings that can be specified in the C11 standard from the weakest to the strongest.

Acquire and Release Semantics

An atomic operation can have acquire semantics if it is an atomic operation that performs a load from memory and possibly something more, e.g. a RMW operation, and an atomic operation can have release semantics if it is an atomic operation that performs a store to memory and possibly something more. So, with a release operation A that writes to a memory location M and an acquire operation B that reads from memory location M, we can ensure that all memory operations ordered before A become visible to operation ordered after B, if B reads the value written by A. In Listing 2.20 we show an example of an acquire release ordering. Note that the store and load to y do not even need to be atomic.

```
1  int x = 0;
2  int y = 0;
3
4  P0(int *x, *y){
5      *y = 1;
6      atomic_store_explicit(x, 1, memory_order_release);
7  }
8
9  P1(int *x, int *y){
10     r1 = atomic_load_explicit(x, memory_order_acquire);
11     r2 = *y;
12 }
13
14 assert(r1 == 0 || r2 == 1);
```

Listing 2.20: Acquire release usage

Sequential Consistency Semantics

Sequential consistency ensures total ordering and is strictly stronger than acquire release semantics. So with a sequentially consistent operation A that writes to memory location M and a sequentially consistent operation B that reads from memory location M we can ensure that all memory operations ordered before A become visible to operation ordered after B if B reads the value written by A, and the memory operations that are ordered after A happen after the memory operations ordered before B if B sees a value of M before the write by A. Listing 2.21 shows an example of where sequentially consistent order usage is required.

```
1  int x = 0;
2  int y = 0;
3
4  P0(int *x, *y){
5      atomic_store_explicit(x, 1, memory_order_seq_cst);
6      r1 = *y;
7  }
8
9  P1(int *x, *y){
10     *y = 1;
11     r2 = atomic_load_explicit(x, memory_order_seq_cst);
12 }
13
14 assert(r1 == 1 || r2 == 1);
```

Listing 2.21: Sequential consistency usage

Relaxed Ordering

The relaxed ordering provides no ordering guarantees[2] and thus has minimal performance impact, it is meant to be used when ordering is not a concern, but the operation still needs to happen atomically and thus a non-atomic operation is not sufficient. Note that it still ensures some properties, such as, cache coherence for example, and obviously, does not allow any reordering that would make single threaded execution incorrect. It can be used by passing the *memory_order_relaxed* value as the argument for an explicit atomic operation. As this option does not ensure any ordering, it is meaningless to pass it as an argument to an explicit fence and doing so results in a no-op.

Consume Ordering

Consume ordering needs to be paired with a release operation and has semantics similar to acquire but weaker, in that it only ensures that operations dependent on the consume load can

[2]An exception being loads on Itanium, which result in an acquire load being emitted, otherwise non cache coherent behavior could be observed.

view the operations ordered before the release. However, the definition of dependency can be tricky, as the compiler is sometimes able to eliminate some dependencies during optimization. As a consequence the value of *memory_order_consume* is always regarded as an acquire in practice. Most architectures do not reorder dependent loads, which means that a consume operation would not need to emit any memory barrier instruction in such architectures. Listing 2.22 shows an example of the usage of consume, in which the load of the value depends on the consume load of a pointer to it.

```
1  int *x = NULL;
2  int y = 0;
3
4  P0(int **x, *y){
5      *y = 1;
6      atomic_store_explicit(x, y, memory_order_release);
7  }
8
9  P1(int **x){
10     r2 = atomic_load_explicit(x, memory_order_consume);
11     if(r2 != NULL){
12         r3 = *r2;
13     }
14 }
15
16 assert(r2 == NULL || r3 == 1);
```

Listing 2.22: Consume usage

Acquire Ordering

The acquire ordering is available through the *memory_order_acquire* value and provides acquire semantics. As such it can only be paired with operations that read memory. When used in a fence it should be placed after the acquiring operation and before the operations that need to happen after the ones that are ordered before the releasing operation. Another way to look at the fence is that it prevents loads and stores after the fence to be reordered before atomic loads ordered before the fence. In the example hardware implementation we showed previously, it would be equivalent to an invalidation queue flush. Note that the previously shown hardware optimizations do not allow loads to be reordered after stores, however, it can be allowed by other hardware optimizations and the compiler, nonetheless this fence prevents such reordering as well. Listing 2.23 shows such an example.

Release Ordering

The release ordering provides release semantics and is available through the *memory_order_release* value. And as such can only be paired with operations that write to memory. When used in a

fence it should be placed before the releasing operation and before the operations that need to happen before the operations that are ordered after the acquiring operation. Another way to look at the fence is that it prevents loads and stores before the fence to be reordered after atomic stores ordered after the fence. In the example hardware implementation we showed previously, it would be equivalent to a store buffer bypass prevention until all stores previously in the buffer are completed. Note that the previously shown hardware optimizations do not allow loads to be reordered after stores, however, it can be allowed by other hardware optimizations and the compiler, nonetheless this fence prevents such reordering as well. Listing 2.23 shows such an example.

```
1  int x = 0;
2  int y = 0;
3
4  P0(int *x, *y){
5      r1 = *y;
6      atomic_store_explicit(x, 1, memory_order_release);
7  }
8
9  P1(int *x, *y){
10     r2 = atomic_load_explicit(x, memory_order_acquire);
11     *y = 1;
12 }
13
14 assert(r1 == 0 || r2 == 0);
```

Listing 2.23: Load reordered after store prevention

Acquire Release Ordering

The acquire release ordering is available through the *memory_order_acq_rel* value and provides both acquire and release semantics, and as such is only applicable to operations that both read and write to memory such as RMW operations or fences. An acquire release fence is equivalent to a pair of acquire and release fences, but note that it does not prevent stores before the fence from being reordered with loads after the fence, as the acquire fence does not prevent stores before the fence from being reordered after the fence, and the release fence does not prevent loads ordered after the fence from being reordered before the fence.

Sequentially Consistent Ordering

The sequentially consistent ordering provides sequentially consistent semantics and is available through the *memory_order_seq_cst* value. This ordering can be applied to any kind of operation and can mach an acquire or release fence in order to provide acquire release semantics. When used in a fence, it prevents any reordering between operations ordered before the fence

and after the fence. In the example hardware implementation shown previously, it would require the disabling of all optimizations, including a store buffer flush which is the optimization that leads to the most gains.

2.4 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 what is required, because the presence of 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 back 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 [21].

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. The most common strategy is to apply the CAS to references instead of values, as in the presence of a garbage collector or memory reclamation scheme, the recycling of memory is prevented until it is no longer being used, including by the thread trying to commit the operation. In other words it would prevent the A in ABA from reappearing until the operation finishes and it is no longer a problem. When such a solution is not possible, as is often the case when implementing the memory reclamation scheme itself, the most common way to avoid the problem is through the use of the DWCAS, 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 can be large enough, so it can be proven that the overflow can only occur after an amount of time that the execution is not expected to last. Another practical problem is that the DWCAS instruction is not widely available across all common architectures. An ideal solution would be the use of 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, as discussed before, software implementations of LL/SC have a significant overhead compared to the CAS family of instructions and hardware implementations are too limited to be useful in such scenarios. As the ABA problem is only a problem if the program state in the first A is different from the program state in the second A, DCAS can also be a solution. If the second memory location DCAS can act on can be used to verify if the program state has

changed[3], we can ensure the change on the first location only occurs if the program state is the same, and thus if an ABA did occur it would be harmless. However, DCAS is rarely available in hardware (not supported by any current/mainstream ISA), and software implementations suffer from the same issues as LL/SC.

2.5 Safe Memory Reclamation

Safe memory reclamation (SMR) on lock-free data structures is a much harder problem to solve, compared to their lock based counterparts. In a lock based data structure one can easily have exclusive access to any region of memory, and as such, memory can safely be freed. However, in a lock/wait-free environment, exclusive access to any region of the data structure can not be expected without violating the lock/wait-free properties. To reclaim memory extra care needs to be taken to ensure that such memory will not be used again after being reclaimed by a thread that possesses an old reference to it, at the consequence of reading invalid memory, corrupting the process memory or triggering an access violation from the operating system.

In a lock-free data structure a node goes through a total of 4 stages from a memory management standpoint. A node starts as **unallocated**, and when it is received from the allocator the node becomes **local** and is populated with the appropriate information. Then the node can be inserted in the data structure and becomes **shared**, meaning it can now be accessed by any thread traversing the structure. Finally, during removal the node is disconnected from the data structure and becomes unreachable at which point other threads can no longer acquire a reference to it. However, threads can still possess references to the node that have been acquired before the node became unreachable. At this point the node can be **retired**, and when no other thread can access the node, it can be reclaimed, at which point it becomes unallocated again, and its memory can either be reused or returned to the operating system. At this point, another node using the same address can be regarded as an entirely different node.

The main function of an SMR method is to determine when a node that is retired can be reclaimed and become unallocated. To achieve this, a retired node is placed in a *limbo list* until it can be determined that the node can be reclaimed. The limbo list can either be private to the thread, shared between all threads or a combination of both. The retire operation is usually done explicitly by the programmer, but SMR methods can also include memory management techniques that allow the programmer to forego the explicit insertion of the retire calls.

Determining when a node becomes reclaimable can be done fully passively by just waiting for every thread to lose their references to the node, or a more proactive approach can be taken by changing the execution in order to force threads to lose the references to the node.

[3]For the second memory location a desired value equal to the expected value would be used.

2.5.1 Strategies

The usual methods for memory reclamation can be based of reference counting [23, 60, 86], quiescent states [29, 35], forced quiescent states [81], hazard pointers [41, 44, 58] or a combination of these strategies [16, 19, 74, 88]. Next we will describe each of these 4 strategies.

Reference Counting

Lock-free reference counting [86], as the name implies, each node of the data structure has an associated reference counting field that counts the number of references that exist to the node. As such, each thread tries to increment the reference counting field before accessing the node, and decrements after it completes its access. When the reference counter of a node reaches 0, the node can safely be reclaimed. The main advantage of these methods is that they not only provide memory reclamation but can also help in memory management.

However, this strategy also suffers from the same downside as its classic counterpart, in that circular references can happen in some contexts leading to groups of nodes that have their counter never reach 0. An increment and decrement to the reference counter can also be a significant overhead during operation leading to poor performance, and a single stopped thread accessing a node can prevent not only that node from having its reference counter reach zero, but also an infinite chain of nodes reachable from that node, leading to unbounded memory usage.

Quiescent States

Quiescent states [35, 46] based methods rely on the basic idea that after some time has elapsed following a node becoming unreachable, all threads lose their reference to such node and the ability to acquire new references to it, and as such it becomes reclaimable.

During normal operation there are time periods in which threads access memory shared among multiple threads, these time periods are called *critical sections*. The periods of time outside such critical sections, in which threads lose all references to shared memory, are called *quiescent states*. A time period in which all threads go through at least one quiescent state, is called a *grace period* and ensures that any node that was unreachable before the start of the grace period can be reclaimed after it has elapsed.

Figure 2.7 shows an example timeline with three threads performing operations in a data structure. Thread T_3 is performing a remove operation that starts by invalidating the node and then makes it unreachable. After the node is unreachable, waiting for every thread to finish its current operation and go through a quiescent state, which corresponds to a grace period, is enough to ensure that the node is reclaimable after the grace period has elapsed.

In order to keep track of quiescent states to determine when a grace period has elapsed, and thus safely reclaim memory, different mechanisms can be employed. However, independently of

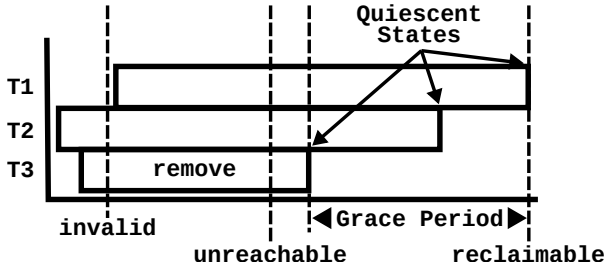


Figure 2.7: Grace period

how the grace periods are tracked, their natural occurrence can not be guaranteed in a lock-free environment, as a single thread not making progress and thus, not going through quiescent states, prevents the occurrence of all future grace periods. Consequently, this strategy does not actually ensure that memory will be reclaimed and as such is not robust.

Usually there are two different ways to interact with a quiescent state based method: (i) by declaring the beginning and end of critical sections, which is easier for the programmer, and allows for periods outside critical sections to be regarded as extended quiescent states, or (ii) periodically declare an instantaneous quiescent state, this usually allows better performance but can consume more memory, and the programmer is responsible to make sure quiescent states are declared sufficiently often.

Forced Quiescent States

A new alternative to classic quiescent state methods is to force the quiescent states to happen [19]. This is accomplished by forcing threads to abort and restart its current operation, which solves the unbounded memory problem, but the forced quiescent states can also limit the progress guarantees of every operation of the data structure to lock-freedom. A thread can now starve by being constantly forced to restart an operation without ever being able to complete it, but note that every restart implies progress has been made by another thread.

This strategy can be somewhat easily applied to *access-aware* data structures, which are data structures that can be separated into read-only and write phases [82], otherwise specific abortion recovery mechanisms need to be devised to achieve compatibility.

Pointer Based

In this strategy each thread is in charge of informing all the other threads of the node it is accessing, making it possible for each thread to know exactly what nodes are unsafe to reclaim. This strategy is more complex than it seems because after a thread informs the other threads that a node is unsafe to reclaim, it needs to ensure that the node is still reachable, otherwise it might have already been reclaimed before it was able to protect it from reclamation by informing the other threads.

This requirement ensures well-defined memory bounds but also imposes some overhead and can limit compatibility with some data structures as unreachable nodes can not safely be used for traversal.

2.5.2 Properties

SMR methods have a multitude of desired properties, but there cannot be a single method that achieves all the desired properties at the same time [80]. As such, each solution constitutes a trade-off between these properties that we will present/define next.

Robustness

The term robustness has been used to describe SMR methods by multiple authors, but only formalized by Sheffi and Petrank [80]. An SMR method is robust if it can have a bound on the maximum amount of nodes that can be in the retired state and have not yet been reclaimed, i.e. have not reached the unallocated state after being retired. Sheffi and Petrank [80] define *Robustness* as a bound on the retired nodes that is asymptotically smaller than the size of the data structure at any moment multiplied by the number of threads, and *Weak Robustness* as a bound that is polynomial on the size of the data structure at any moment multiplied by the number of threads.

Applicability

For the applicability property, the definition by Sheffi and Petrank is also followed [80], which ensures that the integration of the SMR method with the data structure is correct, memory safe and retains the progress guarantees of the data structure. Note that some data-structures have different progress guarantees for different operations but here only the progress guarantee of the data structure as a whole is taken into account (e.g., having wait-free searches and lock-free inserts and removes, renders the data structure as lock-free). It further defines *Strong Applicability* if the SMR method is applicable to any data structure and *Wide Applicability* if the method is applicable to any access-aware data structure, which are data structures that can be separated into read-only and write phases [82].

Ease of Use

For an SMR method to be easy to integrate in any data structure [80], it needs to: (i) be provided as an object with a uniform API that can be used across any implementation; (ii) only implement or require calls for the start and end of an operation, allocation and retire, and instrument primitive memory access operations; (iii) instrumentation of primitive memory operations should be linearisable implementations of such operations; (iv) the implementation

should be well-formed, so control flow is not moved arbitrarily between the SMR and data structure, such as, the forced restarts in forced quiescent states methods; and (v) the SMR method may add new fields to the data structure nodes, but not access or modify any fields apart from the ones it adds.

2.5.3 Methods

The first description of a memory reclamation method was done by Kung et al. [46] in its description of a binary search tree in which each modifying operation locks a constant number of nodes, and read only operations can execute in a non-blocking fashion. However, no clear method to track grace periods is provided, just an idea of using a log system to keep track of entries and exits of critical sections, and the use of such a log to identify grace periods. There is no formalization nor implementation of such a log system to determine the occurrence of grace periods.

This memory reclamation method, as proposed by Kung et al., is later referenced as the solution to memory reclamation in other concurrent data structures [50, 51, 73], but no additional contribution to it is made, including to the problem of tracking grace periods.

Valois Lock-free Reference Counting

The first fully functional SMR was described by Valois [86] and further corrected by Michael and Scott [60]. The implementation is based on reference counting. To access a node, it starts by trying to increment its reference count, and then validate if the reference used to access the node remains unchanged. At the end of an access, the reference count is decremented and, if it causes it to reach zero, the thread tries to claim the node for retirement by marking a claim flag. A thread that succeeds in marking the claim flag retires the node. In this form, the method can not guarantee robustness, as a single stopped thread holding a local reference to a node can prevent not only that node from being reclaimed but also an infinite chain of nodes that starts in that node. Another limitation is that reclaimed nodes can only be reused for the same purpose and never returned to the operating system, as the reference counting field of the node needs to remain valid throughout the lifetime of the data structure. Note that a node could be reclaimed after a thread reads a reference to it, but before it increments its reference counter.

Detlefs Lock-free Reference Counting

In 2001, Detlefs et al. proposed a new method for lock-free reference counting [23], in which by using the Double CAS (DCAS) primitive that can operate on 2 non-adjacent memory locations simultaneously, it gains the ability to arbitrarily reuse the reclaimed memory. By using DCAS it is possible to atomically verify if a node is valid and, only if it is valid, simultaneously increment its reference count. This is done by using a DCAS operation to act on both the reference to the

node that is present on the previous node and the reference count field of the node, but only modify the reference count field. The DCAS instruction however, is rarely available in hardware, which extremely limits the portability of this method.

Read Copy Update

Motivated by the lack of scalability and priority inversion issues associated with readers-writer locks, operating system programmers start to adopt the quiescent states method in order to create the *read-copy update* (RCU) API [5, 31, 34, 53–55, 78]. RCU is defined as a mechanism in which threads make changes to a data structure by first reading and copying the portion that needs to be changed, then performing changes locally in the copy, and finally updating a reference in the data structure to replace the old portion with the changed one. However, the techniques used for tracking grace periods heavily rely on the control over the scheduler and performance counters only available at a kernel level.

Userspace RCU

In 2012, Desnoyers et al. brought the RCU API that was heavily developed by operating system programmers, into userspace [22]. However, the methods proposed for tracking grace periods are done in a blocking manner, and mostly following the ideas proposed by Harris [35] and Fraser [29].

Quiescent States Based Reclamation

In 2001, Harris starting from his lock-free implementation of a linked list, proposed a method to determine the occurrence of grace periods [35]. Each thread copies the value of a global clock to a local clock on a quiescent state. To determine if a grace period has elapsed since a node was made unreachable, it only needs to verify if all threads have a local clock time more recent than the time at which the node has been retired. To improve performance a batching technique is also used and, instead of tracking grace periods per node, it groups nodes in lists with equivalent retire times of the last retired node, and when a grace period elapses for the list, all nodes it contains become reclaimable.

Epoch Based Reclamation

In 2004, Fraser proposes another method to track grace periods called epochs, that is also commonly known as *Epoch Based Reclamation* (EBR) [29]. This method has a global epoch that every thread updates to at a quiescent state, and this global epoch is incremented when all threads reach the current global epoch. As such, when all threads reach the current epoch, the nodes made unreachable 2 epochs ago become reclaimable. Thus, only 3 distinct values

for epochs are needed and the same number of lists to batch unreachable nodes. The main idea is that as a thread makes a node unreachable at an epoch e, it can be sure that all other threads are either at epoch $e-1$, e or $e+1$, so when every thread reaches $e+2$ (the moment the global epoch will be increased to $e+3$) it can be sure that even the threads that were at $e+1$, when the node was made unreachable, have been through a quiescent state, and as such a grace period has elapsed. The 3 limbo lists are global, which improves memory usage, but also increases synchronization and thus can impact performance. Overall, the method is easy to use and strongly applicable but not robust.

Hazard Pointers

The hazard pointers strategy was proposed by Michael [56, 58]. This strategy uses a global list to store a number of hazard pointers per thread, corresponding to the nodes each thread currently has access to. The list is readable by all threads, but each thread can only write in its associated entries.

When a thread traverses a data structure, it starts by reading a valid reference to a node, then it updates one of its hazard pointers to that reference, and finally it needs to verify if the reference remains valid, as otherwise, the node it is trying to dereference might have been removed and reclaimed concurrently to the assignment of its hazard pointer.

Note that in order to ensure that a reference is valid, the node that contains it must be reachable. As an example Michael suggests a variation of Harris linked lists in which the validity flag that is used to facilitate removals is also used as an over-approximation of unreachability. As nodes can only be accessed while protected by a hazard pointer, in a data structure like a linked list, at least 2 hazard pointers per thread are required as a thread can only protect the next node by accessing its predecessor, and its predecessor needs to remain protected for the duration of the protection and validation of the next node. When none of the protected nodes can be proven to be reachable from the root, the thread is forced to restart the operation as it is unable to protect any further nodes. When a node is removed and becomes unreachable, it is stored in a thread local limbo list. The full list of hazard pointers is then periodically read and every node in the local list that is not referenced in any hazard pointer is freed. This allows for a tight memory bounds as long as an unbounded number of hazard pointers per thread are not required.

The inability to traverse invalid nodes that are still reachable renders this method not applicable to some data structures. Note that the protection validation requires a sequentially consistent memory ordering, which implies at least a store buffer flush in almost all architectures. The asymmetric memory barrier optimization described by Dice et al. [27] somewhat reduces this performance impact but comes with its own limitations and disadvantages.

Pass the Buck

Herlihy et al. developed a solution named *Pass The Buck* (PTB) [40] that shares the same underlying idea of Michael's hazard pointers. The main difference is the usage of a global hand off structure, that has one entry per hazard pointer. The hand off structure serves the purpose of holding nodes that cannot be reclaimed due to being protected by a hazard pointer. So, during the reclamation phase, a thread instead of keeping in the limbo list a node that cannot be reclaimed, it stores it in the respective global hand off entry. When the entry is already populated by another node, it means that the respective hazard pointer has changed since that node was handed off, so, the node being reclaimed is switched with the one in the hand off entry. Note that the retrieved node from the hand off entry can end up in another hand off entry if there is another hazard pointer protecting it. In other words, it takes advantage of the fact that each hazard pointer can only protect one node to have an entry per hazard pointer to store nodes that fail to be reclaimed due to being protected. This hand off structure can have a negative impact on performance, but reduces memory usage.

Single-word Lock-Free Reference Counting

Herlihy et al., also suggests the combination of the PTB method with Detlefs lock-free reference counting in order to create the *Single-word Lock-Free Reference Counting* (SLFRC) [39, 41] method. The idea is to use the PTB method to protect references while the reference count is incremented. By combining both strategies it is able to perform lock-free reference counting with only resorting to single-word CAS instead of using DCAS, it is able to free nodes, and it is able to perform validation of reachability by relying on the reference counting method. The usage of lock-free reference counting can be used to achieve strong applicability but at the cost of performance and robustness.

Blelloch's Hazard Pointers

Blelloch and Wei improves on hazard pointers by making the protection of nodes constant time instead of lock-free [13]. This is achieved by using swcopy [12], a primitive that atomically copies a memory location to a single-writer memory location that can be implemented in constant time using only single-width CAS. As hazard pointers are single-writer pointers, the swcopy primitive allows it to forego the need to verify that the protection was successful and thus possibly have to retry indefinitely. Another key feature is that the API is changed so that nodes can be retired more than once and an eject call incrementally reclaims nodes and returns them, which can happen as many times as the number of times the node was retired. The improvement to hazard pointers keep the same strong robustness and lack of applicability as the original method.

Blelloch's Reference Counting

Blelloch and Wei also show that the new hazard pointer API can be used to implement reference counting [13] by protecting reference counting changes and by retiring the object on each reference destruction that will then have its reference count decremented when it is processed by eject. This method of operation allows all the changes to the reference counter to be done in constant time with the use of atomic_fetch_add instead of having to use CAS for increments due to the counters being sticky. This reference counting method gains applicability at the cost of robustness compared to the hazard pointer implementation.

Deferred Reference Counting

Anderson et al. improves on the reference counting method by Blelloch et al. [13] by introducing snapshot pointers to develop *Deferred Reference Counting* (DRC) [4]. The key idea is that during a traversal of the data structure, constantly updating the reference count on every node traversed can be a significant bottleneck, and as such, it augments the method by allowing references to be obtained by snapshot pointers that instead of incrementing the reference count they use a hazard pointer buffer that prevents the reference count from being decremented. If the buffer of hazard pointers is ever exhausted, the method falls back to incrementing the reference count. When the snapshot pointer is destroyed the method checks if the pointer is in the hazard pointer buffer in order to decide if it needs to decrement the reference count. The method achieves the same properties as the reference counting method it is based on.

Pass The Pointer

Correia et al. developed the Pass the Pointer (PTP) method [20] which is similar to the hazard pointers method, but further lowers the memory bound. In this method, instead of a limbo list it has a handover array that has one entry per hazard pointer. As such, when retiring a node the thread goes through the hazard pointers array until it finds one protecting the node, and at that point it replaces the corresponding entry in the handover array with the node. If another node is present in the entry, it continues the search in the hazard pointer array, now searching for the replaced node as before. When the end of the hazard pointer array is reached, the node can be freed as there can be no hazard pointer protecting it. In other words, the retired nodes trickle down the hazard pointers until there is no hazard pointer left to protect it. Note that a hazard pointer can at most protect one node. Based on this manual method, Correia et al. also present an automatic method called OrgGC that uses reference counting for shared references and the PTP method for local references.

HP++

The HP++ method, proposed by Jung et al. [44] improves on hazard pointers by achieving applicability while retaining the robustness of hazard pointers. The hazard pointers method uses the logical deletion of nodes (by marking the least significant bit of the next pointer) to over-approximate unreachability, which makes it unable to follow references in invalid nodes and thus incompatible with some data structures. The HP++ method makes invalidation an under-approximation of unreachability by having nodes marked as invalid only after being made unreachable. In order to do so correctly, it also protects the node that follows the node or group of nodes that were made unreachable with a hazard pointer until the unreachable node or nodes have been invalidated. This makes it possible to traverse nodes that were logically removed but are still safe to traverse, i.e., nodes that were not made unreachable yet at that point. With these changes, the method becomes applicable to any data structure, such as Harris lists that rely on optimistic traversal, i.e., traverse logically deleted nodes without removing them during search operations.

Wait-free Reference Counting

A wait-free version of Valois [60, 86] reference counting method was proposed by Sundell [84], in which he develops helping mechanisms is order to transform the protected read and allocation of nodes, that are the only procedures that might have to retry indefinitely, from lock-free to wait-free. In order to achieve wait-freedom when a thread attempts to traverse a node, it first publicizes the reference it is trying to follow in a hazard-pointer like manner so that any thread that changes such a reference can help it find a valid next node. Then it increments the reference counter of the current node pointed by the reference, and checks if the reference has been changed. If it remains the same it can safely traverse to the node, otherwise it undoes the increment and uses the answer by a helping thread to the publicized reference. When a thread changes a reference in the data structure it now also needs to check if there is any other thread attempting to follow that reference, and if there is, it needs to help by providing the new value to that thread. As a node can be retired between a thread acquiring its reference and incrementing its reference counter, nodes can not be freed, only reused. A single stopped thread preventing a reference counter from reaching zero can prevent an unbounded number of nodes reachable from such node from being retired, rendering the method not robust.

BEWARE&CLEANUP

Following the SLFRC method by Herlihy et al. [39, 41] Gidenstam et al. proposed a lock-free reference counting method called *BEWARE&CLEANUP* (B&C) [32] that uses Hazard Pointers for memory reclamation and reference counting for memory management, so the reference counts only count global references and local references are managed by hazard pointers. But the main improvement proposed is a way to guarantee bounded memory by being able to shorten the

chains of removed nodes that were previously impossible to reclaim due to a thread being stuck at the beginning of such a chain. With the use of hazard pointers, the method is able to identify where a thread is stuck at, and remove the invalid nodes between the thread and the first valid node, making it also possible for the stuck thread to continue traversing the data structure when it wakes up without needing to restart the traversal. This allows the method to be both strongly applicable and strongly robust.

StackTrack

In 2014, Alistarh et al. developed *StackTrack* [2] by exploiting *Hardware Transactional Memory* (HTM). By dynamically splitting each operation into transactions that publish the threads registers and stack before ending the transaction, each thread is then able to verify at any moment which nodes can not be reclaimed by checking the published information. This simulates the ideal scenario where the whole operation is a single transaction, but accomplishes it with smaller transactions that are actually supported by hardware. Publishing the stack before the end of the transaction, works as a checkpoint to prevent other threads from reclaiming memory that is in use in the unsafe moments between transactions. But as the current HTM implementations do not ensure progress, a "slow path" alternative needs to be used when transactions consistently fail, which in this case a similar implementation to hazard pointers is used. This method is implemented at the compiler level, making it easier to adopt, but also limiting compiler choice.

ThreadScan

Alistarh et al. developed *ThreadScan* [3] that uses operating system signals to force threads to analyze their stack and registers for possible references to nodes. When the limbo list is full, the reclamation procedure is started by sending a signal to every other thread, that in response mark the nodes in the limbo list that are referenced in its stack or registers. The thread sending the signals also marks the nodes in the limbo list in the same way, and then waits for an acknowledgement from every other thread that it has completed the task, at which point it can free every non-marked node from the limbo list. This strategy is very efficient as it has no overhead outside the reclamation procedure, and the usage of signals in conjunction with the bounded number of steps required for the reclamation procedure provides some progress guarantee under a fair scheduler. However, the method is still blocking which causes the whole system to lose its lock-freedom or wait-freedom properties.

Drop The Anchor

Combining the advantages of the quiescent states and hazard pointers methods, Braginsky et al. developed the *Drop The Anchor* (DTA) the method [14]. The method relies primarily on quiescent states for reclamation but also periodically updates a hazard pointer (anchor) for a fixed interval of traversals. This strategy allows other threads to identify when a thread is taking

too long to declare a quiescent state and mark it as stuck and execute a recovery procedure so that such a thread does not prevent further memory reclamations. The anchor of a stuck thread is used to find what possible nodes the thread could be accessing (the nodes between the anchor and the next anchor point), and such nodes are frozen and replaced by new copies, at this point the thread is marked as recovered, which can be considered as an extended quiescent state that no longer prevents memory reclamation. However, nodes that were retired before a thread was marked as stuck or recovered cannot be reclaimed, as they could still be accessed by such a thread, and the recovery procedure might not have found them. The usage of such a recovery mechanism allows the method to become robust while amortizing the cost of hazard pointer updates, but requires a DWCAS to update the anchor and time stamp pair, requires nodes to store their insertion time which consumes more memory and is only applicable to linked list like structures.

Qsense

Balmau et al. developed a hybrid strategy between quiescent states and pointer based strategies called *QSense* [9]. This method relies mainly on quiescent state based reclamation in order to achieve the best performance possible, but can fall back to hazard-pointer based reclamation when long delays are detected in other threads. In order to be able to transition between quiescent states and hazard pointers, it needs to assign hazard pointers even when running in quiescent state mode. This would usually lead to a high overhead due to the memory barriers required by hazard pointer assignment, but QSense avoids the direct use of such barriers by relying on rooster processes. Roster processes are a kind of dummy processes set to run periodically, one in each hardware thread, causing a context switch in the hardware thread. Since a context switch requires a memory barrier, we can be sure that if we wait for a period of time that at least one context switch has occurred in every thread, we can reclaim nodes older than that period in a hazard pointer way even if we do not use memory barriers during hazard pointer assignment.

Hazard Eras

Ramalhete and Correia in 2017 were able to bound memory usage in a quiescent state based method, called Hazard Eras (HE) [74], by allowing the retirement of nodes that were inserted and removed after a thread stopped, thus achieving weak robustness. In order to do so, it uses a global monotonic era clock, nodes are annotated with the era they were inserted in and the era in which they were removed, and each thread has an array of hazard eras that stores the eras in which it acquired hazardous references in. To acquire a valid reference in this method and ensure it remains valid, we have to first read the reference while ensuring it is still reachable (requires a Harris-Michael list like traversal), then ensure we have the current global era in one of our hazard era array entries, and finally verify that the reference has not changed in the meanwhile. In a sense, this mode of operation is very similar to the hazard pointers method despite relying on a different strategy. To retire a node, we first annotate the node with the global era it was removed

on, then we add the node to the limbo list and increment the global era, and finally we can retire every node in the limbo list for which there is no hazard era *he* that for the node creation era *ce* and deletion era *de* we have, $ce \leq he \leq de$. The main advantage of this method in comparison to hazard pointers, is that for most accesses we already have a valid hazard era from the previous which allows us to forego updating the hazard era and also forego the memory barrier associated with it. Even though this method is able to achieve weak robustness, by requiring a hazard pointer like traversal and hazard pointer like API, it loses the applicability usually provided by quiescent state based reclamation.

Interval Based Reclamation

Wen et al. developed Interval Based Reclamation (IBR) [88] that is similar to the HE method but differs in two key aspects. Instead of an array of hazard eras that protect nodes created before and removed after such eras, it has an interval that protects all nodes that have a lifetime (interval between creation and a deletion era) that intersects with the thread's interval. And during traversal, instead of reading the node reference then updating the protection (if needed) and then verifying if the reference remains the same, it starts by updating the interval, then reads the reference and then verifies that the global epoch remains the same. The lower bound of the interval is set to the current era when the operation starts and the upper bound is updated as nodes are read. This methodology leads to more memory being irreclaimable, but keeps it bounded and allows the pointers to nodes to be tagged with the creation era of the node it is pointing, which allows the method to forego reading the global era clock during traversal and consequently reduce synchronization at the cost of some memory overhead. These tags can be relaxed, so they can be higher than the creation date of the following node, which removes the need for the tag and pointer to be atomically changed.

Margin Pointers

Solomon and Morrison developed the Margin Pointers (MP) method [83] which amortizes the cost of hazard pointers by having a margin pointer protecting an interval of nodes instead of a single node. This makes the method limited to ordered data structures and requires every node to be assigned an index that is also strictly ordered, which is achieved by giving a node an index that falls exactly in between the previous and the next node. In order for a node to be protected before being accessed, its index needs to be stored as a tag in every pointer that points to the node. The method then stores an index as a margin pointer that protects all the nodes in a predetermined interval centered around the protected index. To deal with collisions in indexes, a node that cannot be given a unique index, it is given a special index that informs the system that hazard pointers should be used for such a node. As multiple nodes can end up with the same index, when the previous ones have already been removed, a strategy similar to hazard eras is used to prevent unbounded memory. This makes the method robust, but neither applicable nor easy to use.

DEBRA+

Brown developed DEBRA [16] as an improved version of epoch based reclamation (EBR). DEBRA uses a private limbo list per thread instead of a global one and uses global and local memory pools with efficient movement of memory between them to minimize utilization of the memory allocator. These modifications to EBR minimize synchronization thus improving performance, but keeps the same limitations of EBR. Additionally, DEBRA+ was also proposed, which adds robustness to DEBRA by adding the ability to *neutralize* crashed/delayed threads forcing them into a quiescent state. This neutralization mechanism is achieved by the usage of signals and siglongjmp, so when a thread does not advance in its epoch for sufficiently long, it is sent a signal that causes it to execute an idempotent recovery mechanism. As it assumes a thread after receiving a signal cannot execute any further steps before executing the signal handler, this mechanism effectively forces the thread into a quiescent state as long as the recovery mechanism makes it lose all its local references. For modification operations, hazard pointers are used to protect the affected nodes by the modification, so the modification part of the operation can be done in a quiescent state.

Optimistic Access

Cohen and Petrank proposed *Optimistic Access* [19] in 2015 as a way to materialize the strategy that we describe as forced quiescent states. As the name implies, this method takes an optimistic approach to memory reclamation, such that threads can read nodes that might be already reclaimed but check afterward if the read was valid and ignore the result if it was not. It still requires hazard pointers to protect all nodes involved in a modification operation, as modifying reclaimed nodes could leave the data structure in an inconsistent state, even if only temporarily. In order to force quiescent states, it implements a shared list with one *warning flag* per thread that is used to inform threads of a quiescent state. Before performing reclamation threads inform every other thread by marking their warning flags, which every thread checks after every shared read in order to verify if the read was valid. So, when a thread performs a read on shared memory and after verifies that its warning flag is set, it immediately restarts the operation and resets its warning flag. The possibility of reading reclaimed memory however, prevents the method from using the memory allocator to release memory, as an access to an unmapped address would cause a segmentation violation. As a solution, the method uses a memory pool and performs memory reclamation only when such memory pool is exhausted. Threads start by storing retired nodes in a private list in order to minimize synchronization, and when a threshold is reached they are moved to a global retire pool all at once. When a reclamation is triggered all threads are warned through the warning bit and all nodes are moved from the retire pool to a processing pool, in which all nodes not referenced by any hazard pointer are moved to the memory pool and nodes referenced by a hazard pointer are moved back to the retire pool. The main advantage of this optimistic approach is that for the validity check only an acquire memory barrier is required and for modification operations only a single sequentially consistent barrier is required for multiple

hazard pointers. The method still achieves robustness and wide applicability.

Automatic Optimistic Access

Cohen and Petrank then improve on Optimistic Access with Automatic Optimistic Access [18]. This method foregoes the user from having to explicitly retire nodes, and instead uses a mark and sweep mechanism to mark all nodes that are reachable and then reclaim all unmarked nodes. Nodes can be considered reachable if they are either reachable from the global roots of the data structure or from a hazard pointer. When reclamation is triggered by an exhaustion of the memory pool, a new mark and sweep phase begins, in which all threads try to collaborate. When a thread crashes or is delayed during the mark phase, its work is picked up by another thread. The memory pool is implemented as a lock-free stack in which the current mark and sweep phase is also stored in its head field requiring a DWCAS to update. This ensures the synchronization of phases as a new one cannot start before the previous one has finished, and allows threads to begin collaboration as soon as they try to allocate a new node. This method keeps the robustness and wide applicability properties of the Optimistic Access method it is based on.

Free Access

Cohen improves on AOA with Free access [17] by foregoing the need for the data structure to be written in a normalized form [85], only needing the data structure functions to be annotated and then relying on a compiler pass to divide the program into read-only and write-only sections. It automatically registers local roots at the start of each section in hazard pointer frames that are then used for the mark and sweep collection of unreachable nodes.

Version Based Reclamation

Sheffi et al. improves on the optimistic access method by allowing modification operations to also be done optimistically instead of relying on hazard pointers [81]. This is achieved by changing the warning method from one warning bit per thread to a global monotonic clock. The global clock is incremented in order to send a warning to all threads, and by having every thread have a local copy of the last value seen in the global clock, which allows threads to verify the existence of a warning by comparing their local copy against the global clock. This warning method allows the use of the value of the global clock as a tag in every atomic reference that is then updated with a DWCAS, which ensures that no ABA problem can occur and that reclaimed nodes are not changed as the DWCAS is ensured to fail in such cases. The method achieves the same applicability and robustness as the optimistic access method.

Neutralization Based Reclamation

Singh et al. follows the forced quiescent state strategy in order to develop neutralization based reclamation (NBR) [82]. NBR requires the data structure to be divided into read and write phases, in which read phases are restartable and write phases only begin after every node that is going to be used during the phase has been protected in a hazard pointer like fashion. At the start of the read phase the thread saves its state through the use of sigsetjmp and sets a restartable flag to true. At the start of the write phase, it protects all the references it will need during the phase in a hazard pointer like fashion and then sets the restartable flag to false. When performing reclamation, a thread starts by sending a signal to every other thread, which causes all threads in the read phase to restart, and then it can reclaim every node that is not protected by a hazard pointer. This can be done because the signal acts as a forced quiescent state for threads in the read phase and nodes used by threads in the write phase are protected by hazard pointers. The method is then improved upon by NBR+ which allows threads to piggyback on other threads forced quiescent states in order to perform reclamation without having to force all readers to restart, which effectively reduces the amount of signals and restarts. This method while not easy to use is widely applicable and robust.

Pointer and Epoch Based Reclamation

Kang and Jung developed *Pointer and Epoch Based Reclamation* (PEBR) [45] by combining epoch based reclamation and hazard pointer based reclamation. During normal operation it works similarly to EBR, but hazard pointers are also used to protect every reference followed. In addition, when a thread detects that another thread is not progressing its local epoch, it is able to eject such a thread, forcing it into a quiescent state. This is done by checking the threads hazard pointers and by creating a bloom-filter of hazard pointers in order to help other threads doing reclamation. In order for the ejection to be possible, every thread also needs to check if it has been ejected on every hazard pointer protection, and in case it has been, the protection fails and may cause the thread to restart its operation. In order to keep the method performant, it uses asymmetric memory barriers as proposed by Dice et al. [27]. While the method makes extensive use of Rust language features like RAII, the type system and borrow checker to make the method as easy to use as possible, it is not able to fully achieve that goal, however the method is applicable and robust.

Hayline

Nikolaev and Ravindran developed Hayline [68] which takes a novel approach to reference counting. The main idea is that instead of keeping a reference count in the node when it is being used, the reference count is only given to the node when it is retired and with an initial value of the number of active threads at that moment. Then, when threads finish an operation, they decrement the reference count of every node that was retired during that operation and they free

the ones which the counter reaches zero. This is implemented by having a counter in the head of the global limbo list, which is incremented when threads start an operation and decremented when they finish. Threads also check the last inserted node when they enter so that they know, when they finish an operation, until which node they need to traverse the list decrementing the reference counter. When a node is retired it is added to the retirement list and set with the value of the active threads counter as a reference counter. To amortize synchronization costs, multiple limbo lists are used (not necessarily in 1:1 mapping with threads), and retired nodes are inserted in batches in all lists.

As Hayline is not robust, Nikolaev and Ravindran also develop Hayline-S, which adds eras in order to prevent stopped threads from protecting nodes that were inserted before they stopped. This is done similarly to HE or IBR, as we have a global monotonic clock that limbo lists are updated to during traversal of the data structure as in the IBR method. This allows limbo lists to be skipped when inserting a retired node with a creation time newer than the list. One of the advantages of these methods is two additional properties: (i) transparency, which the authors define as the ability for threads to be created and destroyed seamlessly as they do not possess associated structures like limbo lists or hazard pointers/eras; (ii) snapshot-freedom, which the authors define as not needing to take a snapshot of the state of all threads in order to perform reclamation.

2.5.4 Comparison

Table 2.3 compares all the previous SMR methods in terms of what strategies they are based on, what properties they have, and their APIs. For the 4 strategies, we use Y to indicate if the method is based on Reference Counting, Quiescent States, Forced Quiescent States and Pointer Based. For the 3 properties, we use Y to indicate if the method is easy to use, for robustness, we use W for weak robustness and S for strong robustness, and for applicability, we use W for wide applicability and S for strong applicability. For the API we use W if the method requires double-width CAS, we use D if it requires double CAS and H if it requires hardware transactional memory, and Y if the method uses the free call from the memory allocator.

	Strategies				Properties			API	
	RC	QS	FQS	PB	Easy	Robust	Applicable	Primitive	Free
Valois's LFRC [86]	Y				Y		S		
Detlefs's LFRC [23]	Y				Y		S	D	Y
QSBR [35]		Y			Y		S		Y
EBR [29]		Y			Y		S		Y
HP [56]				Y	Y	S			Y
PTB [39]				Y	Y	S			Y
SLFRC [40]	Y			Y	Y		S		Y
Blelloch's HP [13]				Y	Y	S			Y
Blelloch's RC [13]	Y			Y	Y		S		Y
DRC [4]	Y			Y	Y		S		Y
PTP [20]				Y	Y	S			Y
OrcGC [20]	Y			Y	Y		S		Y
HP++ [44]				Y		S	S		Y
WFRC [84]	Y			Y			S		
B&C [32]	Y			Y		S	S		Y
StackTrack [2]				Y		S		H	Y
ThreadScan [3]				Y		S			Y
DTA [14]		Y		Y		S		W	Y
QSense [9]		Y		Y		S			Y
HE [74]		Y			Y	W			Y
IBR [88]		Y			Y	W			Y
MP [83]		Y		Y		W			Y
DEBRA [16]		Y			Y		S		Y
DEBRA+ [16]		Y	Y	Y		S	W		Y
OA [19]			Y	Y		S	W		
AOA [18]			Y	Y		S	W	W	
FreeAccess [17]			Y	Y		S	W		
VBR [81]			Y			S	W	W	
NBR [82]			Y	Y		S	W		Y
PEBR [45]		Y	Y	Y		S	S		Y
Hyaline [68]	Y	Y			Y		S	W	Y
Hyaline-S [68]	Y	Y			Y	W		W	Y

Table 2.3: SMR strategies and properties

Chapter 3

Lock-free Hash Tries

Hash tries are a tree-based data structure with nearly ideal characteristics for a hash map implementation. Areias and Rocha developed an efficient lock-free design of a hash trie named Lock-free Hash Tries (LFHT) [6, 7]. It was implemented in the Java language, and relied on the Java garbage collector to ensure memory reclamation. In previous work to this thesis, we developed a custom memory reclamation method to the LFHT data structure named *Hazard Hash and Level* (HHL) [62, 64]. The motivation for such work was the fact that the LFHT data structure is incompatible with most SMR methods available. This incompatibility is due to the fact that, during the growth of the data structure, nodes being removed by a thread can be momentarily reinserted by another thread, which can be a problem since most SMR methods rely on the fact that a node is permanently unreachable after it is removed. We call the issue caused by the temporary reinsertion of nodes the delegation problem.

Most lock-free data structures are purely academic works and are rarely used in practical scenarios, even though they often achieve competitive performance in comparison to their lock-based counterparts [7]. In this work, we started by exploring how the LFHT data structure can be used in real world scenarios, and how effective it can be. So, we applied the LFHT data structure to the YAP Prolog system. The YAP Prolog system is known for its performance and, since it supports multithreading, it requires internal data structures that are performant in multithreaded scenarios. We replaced YAP's atom table implementation, which was originally implemented as n hash table with readers-writer locks, with the LFHT data structure and measured the performance impact of the change.

Concurrently, and motivated to improve the performance and scalability to larger amounts of data, we developed a compression mechanism for the LFHT data structure that attempts to reduce its depth, resulting in less memory accesses for every operation [63]. This mechanism takes advantage of the fact, that, when a hash node is fully populated by other hash nodes, we can replace the 2 levels of hash nodes by a single hash node, effectively reducing the depth of the hash trie by one level.

The implementation of the compression mechanism showed that expansions in the LFHT

data structure are complex, which is exacerbated by the compression mechanism. As such, we decided to redesign the LFHT data structure in order to simplify how expansions are performed and at the same time solve the delegation problem in order to make it compatible with most SMR methods. We also chose a design that is more cache friendly in order to improve its performance that we then evaluate against the original design.

In what follows, we start by introducing the Lock-free Hash Tries (LFHT) data structure, as originally developed by Areias and Rocha [6, 7], then we show how memory reclamation was added to it, starting by introducing the delegation problem and then by presenting the HHL method [62, 64] as a solution. Next, we present our work in showing it being used in a real world application, namely the YAP Prolog system in order to replace its atom table implementation [65]. Next, we present our work on a compression mechanism to the LFHT data structure [63] in order to improve its performance and scalability to large amounts of data. Finally, we present a new LFHT design that simplifies the implementation and solves the delegation problem.

3.1 Introduction to LFHT

The LFHT data structure 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, each node with a fixed size bucket array of 2^w entries. To map a key/value pair (k,v) into this hierarchy, a hash value h is computed for k and then chunks of w bits from h are used to index the appropriate hash node, i.e., for each hash level H_i, the i^{th} group of w bits of h are used to index the entry in the appropriate bucket array of H_i. To deal with collisions, the leaf nodes form a linked list in the respective bucket entry until a threshold is met and, in such case, an expansion operation updates the nodes in the linked list to a new hash level H_{i+1}, 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 3.1 shows how the insertion of nodes is done in a hash level.

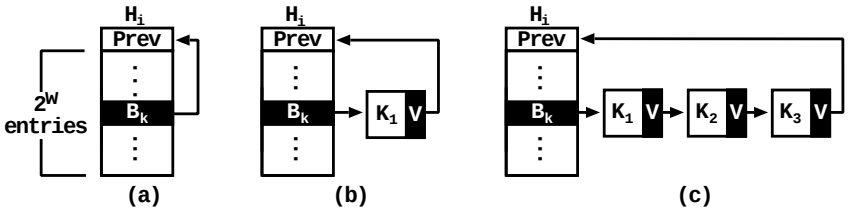


Figure 3.1: Insertion of nodes in a hash level

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 a backward reference *Prev* to the previous hash level, 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 K_1 on B_k and Fig. 3.1(c) shows the configuration after the insertion of nodes K_2 and K_3. 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). The last leaf node in the chain references back to the current hash node.

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. Figure 3.2 shows how nodes are remapped in the new level.

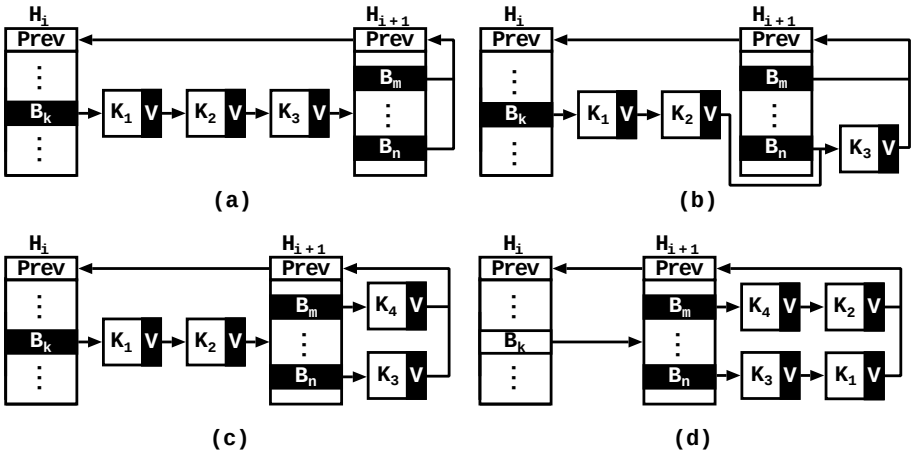


Figure 3.2: Expansion of nodes in a hash level

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 H_{i+1}. Figure 3.2(b) and Fig. 3.2(c) show how node K_3 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 K_4 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 (Fig. 3.2(d)). Immutable fields are represented with a white background.

Next, Fig. 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 (shown in Fig. 3.3(a)) and (ii) making the node unreachable (shown in Fig. 3.3(b)).

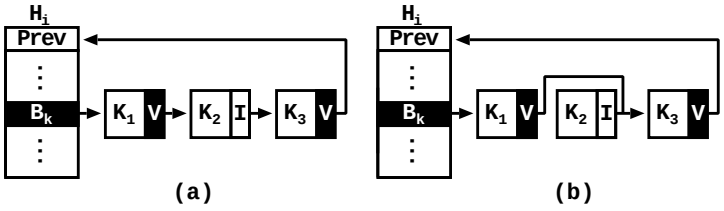


Figure 3.3: Removal of nodes in a hash level

The invalidation step starts by finding the node N (node K_2 in Fig. 3.3) to be removed and by changing its flag from valid (V) to invalid (I). If 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 as the thread making the node invalid becomes responsible for it. Next, to make the node unreachable, first the next valid node A (node K_3 in Fig. 3.3) is found on the chain (note that it can be the hash node H_i corresponding to the level N is at). Then, by continuing to traversing the chain the hash node H_i is found (if it was not yet found). If H_i is the same hash node that the traversal of the chain started from, the chain is traversed again until the last valid node B (node K_1 in Fig. 3.3) before N is found (or we consider the bucket entry if no valid node exists before N). If, while searching for B node N is not found, it means that N has already been made unreachable and the removal is complete. Otherwise, we just need to change the reference of B to A. This is shown in Fig. 3.3(b), where K_1 refers to K_3.

If H_i is not the same hash node the traversal of the chain started from, this means that a concurrent expansion is happening simultaneously and the process is restarted in the next level (note that node N could either have been expanded before it was invalidated or it is currently in the process of being expanded). In the case N has been expanded before it was made invalid, it will be possible to make it unreachable in the next level. Otherwise, if N is in the process of being expanded, the thread doing the removal does not need to make the node unreachable, as the expanding thread doing the expansion will not expand it or will make it unreachable if it only sees N as invalid after completing its expansion. In this situation, the thread doing the expansion becomes responsible for making the node unreachable. The process of transferring this responsibility to the expanding thread is called *delegation* [62, 64].

3.2 Memory Reclamation

In this Section, we describe our previous work on implementing SMR in the LFHT data structure. We start by describing in more detail the delegation problem that affects the LFHT data structure and makes it incompatible with most SMR methods, and then we describe de Hazard Hash and Level (HHL) method that takes advantage of the internal structure of the LFHT design in order to provide efficient SMR while avoiding the delegation problem.

3.2.1 Delegation Problem

By default, all the state-of-the-art memory reclamation methods rely on the fact that an element being removed from a data structure is left in an unreachable state when the remove operation terminates and the retire procedure is called. However, in the original design of the LFHT data structure, a node may still be reachable at the end of the remove operation, if a concurrent expansion is happening simultaneously and the task of making the node unreachable was *delegated* to the expanding thread.

Figure 3.4 illustrates how an expansion operation can change the moment where a node is considered unreachable and reclaimable. In particular, the assumption that a thread starting

after the end of the remove operation cannot obtain a reference to the removed node is not valid anymore. This is the case of thread T_4 in Fig. 3.4, which started its operation after thread T_1 finished the remove operation, but before the node was made unreachable by the expanding thread T_2. In this scenario, a node can become reclaimable only after T_4 finished its operation and, thus, later than what would be expected if no delegation happened.

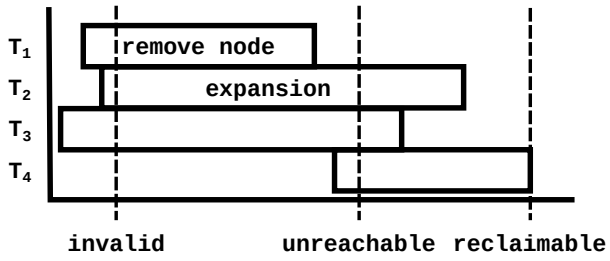


Figure 3.4: Node states during expansion

Avoiding this delegation mechanism is not possible since T_2 can always reinsert the node in the new hash level before realizing that it was marked as invalid and made unreachable. Figure 3.5 illustrates this situation in more detail. The problem resides exclusively in the case where a thread T_2, doing an expansion, reads a valid node K_3 and, before changing the corresponding bucket reference in the new level H_{i+1} in order to expand K_3 (Fig. 3.5(a)), another thread T_1 is able to invalidate K_3 (Fig. 3.5(b)) and make it unreachable (Fig. 3.5(c)). As the removing thread T_1 does not interfere with the reference in H_{i+1}, the expanding thread T_2 can succeed in updating the bucket reference B_n in H_{i+1} to K_3 and effectively reinsert K_3 making it reachable again (Fig. 3.5(d)). Note that the delegation problem happens exclusively when the last node on the chain is being removed, as otherwise, the expanding thread will realize the node is invalid when changing its next reference to the new hash node, and thus, will not insert it in the new hash node.

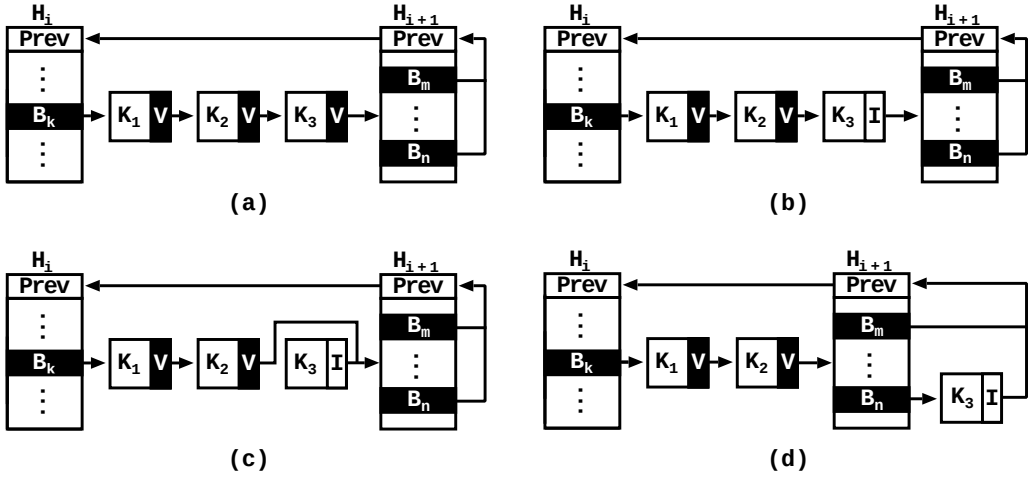


Figure 3.5: Reinsertion of an invalid node during expansion

3.2.2 Hazard Hash and Level

Hazard pointers have good memory bounds in memory reclamation, however they rely on thread synchronization based in performing sequentially consistent atomic writes on every node being traversed. Reducing this synchronization overhead, while keeping good memory bounds, is a difficult task and, a good approach to accomplish it, is to merge nodes in well-defined groups and protect them with a single hazard pointer. Such a strategy was previously used by the DTA method [14] and more recently by the MP method [83].

An interesting characteristic of LFHT is that leaf nodes are already grouped in chains that have a well-defined maximum size. Thus, instead of having a single hazard reference to protect a single node, the *Hazard Hash and Level* (HHL) approach protects a well-defined group of leaf nodes. In this approach, each thread maintains a special *hazard pair* <*HH, HL*>, formed by a *Hazard Hash* (HH) and a *Hazard Level* (HL), to indicate in which part of the data structure it is positioned. HH represents a path in the LFHT data structure and HL represents a portion of this path. In what follows, we describe in detail the HHL approach, its guarantees and limitations.

Properties and Key Ideas

In the HHL approach, the original LFHT's algorithms and data structures were extended to ensure that a thread cannot have access to nodes outside the portion of the path defined by its current hazard pair. It ensures the following properties: (i) threads recovering from preemption must progress to a valid data structure (hash node or leaf node) within the same path; and (ii) no new nodes are inserted in a path with an expansion in course. In the original design, threads can be moved to a different path and recover by moving in that path. In the new design, if a thread is moved to a different path, it now returns immediately to the last known hash node and recovers from that point. Also, in the original LFHT design, the insert and expand operations have the same priority, which means that they could be performed concurrently in the same path. In the new design, it is given a higher priority to the expand operation, such that threads must collaborate to finish the undergoing expansions in a path, before inserting new nodes.

To implement these properties, the following changes were made to the LFHT data structure: (i) a bucket entry includes a *hash flag* to indicate if it stores a reference to a next hash level (the hash flag is part of the atomic field that includes the reference); and (ii) a leaf node includes a *generation field*, indicating the hash level where it was first inserted, and a *level tag*, indicating the hash level where it is at the moment (the level tag is part of the atomic field that also includes the validity flag and the reference to the next-on-chain node). This means that the state information of a leaf node is now given by a generation field G_i and by an atomic tuple with three arguments <*NextNode, LevelTag, ValFlag*>. For example, in Fig. 3.6(c), the value of the generation field for node K_1 is G_1, meaning that it was inserted in the hash level H_1, and the value of the atomic tuple is <K_3, *2, V*>, meaning that it is referring to node K_3 (1st argument), it is in the hash level H_2 (2nd argument) and it holds a valid key (3rd argument).

The key idea behind the HHL approach is the fact that each thread executing on the LFHT data structure protects from reclamation a single and well-defined chain of leaf nodes. Therefore, a leaf node N can only be reclaimed if: (i) N is not in a protected chain; and (ii) N has never been ina protected chain in the past, as a thread could have acquired a reference to it there and still be holding it, despite the fact that, in the meantime, N could have been expanded to a deeper level. The example in Fig. 3.6 illustrates the key ideas of a safe traversal in the HHL approach.

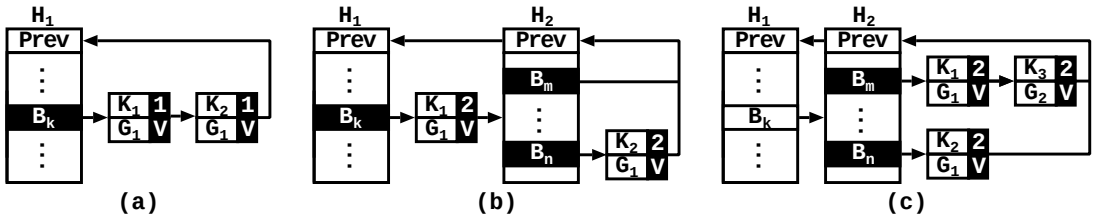


Figure 3.6: Safe traversal of nodes in the HHL approach

Figure 3.6(a) shows the initial state. Assume that a thread T reached the hash level H_1 and has updated its hazard level HL to refer to H_1 (level 1). Assume also that T is searching for node k_2, and was preempted in node K_1 before reading the next-on-chain reference to K_2. While preempted, the configuration of the chain may change due to a concurrent expansion. Later, to guarantee that when T resumes, it can safely follow the reference in K_1, one must ensure that the reference is protected by HL. The three situations that can occur once T resumes from preemption are discussed next.

The first situation is the case where the reference in K_1 still refers to K_2 as shown in Fig. 3.6(a). Since the level tag in K_1 is the same as HL (1=1), T can safely follow the next-on-chain reference to K_2.

The second situation is the case where the reference in K_1 changed due to a concurrent expansion and it refers now to the hash node H_2 as shown in Fig. 3.6(b). Since the level tag in K_1 is now higher than HL (2>1), T is able to detect the concurrent expansion. T then rereads the reference in the bucket entry B_k in order to check the hash flag and understand if the expansion has already finished. As B_k is still referring to the same level (the hash flag is not set), T knows that the expansion is still undergoing and, as no new nodes can be inserted during an expansion, T can safely follow the reference in K_1 to H_2.

The third and last situation is the case where the reference in K_1 also changed due to a concurrent expansion, and it refers now to a different node K_3 as shown in Fig. 3.6(c). Since the level tag in K_1 is again higher than HL (2>1), T rereads the reference in B_k. However, in this scenario, B_k refers to the next level H_2 (the hash flag is set), thus it is not safe to follow its reference, since T can reach a node not being protected by HL, and end in a path where it can miss node K_2. T then restarts the traversal from the reference in B_k instead of following the reference in K_1.

In summary, when traversing a chain, T relies on the level tag to know if an expansion

is happening concurrently. If T finds a level tag that is higher than the current hazard level under protection, and it knows that the level in which it started the traversal has already been completely expanded, then T should not follow any reference because it can reach a node N not being protected by its hazard level.

Solution to the Delegation Problem

A node N being removed is added to the thread's local limbo list at the end of the corresponding remove operation. N is invalidated during the remove operation, but it is uncertain if it is left in an unreachable state, since this process could have been delegated to a thread doing a concurrent expansion. As such, a delayed delegation can further postpone the moment where N can be considered reclaimable. To guarantee that the reclamation of N is safe, the following information is required: (i) the hash value corresponding to the key stored in N, which defines the path where N could have been; (ii) the generation field, which defines the entry point in that path; and (iii) the level tag, that becomes immutable when N is invalidated and thus defines the last hash level where N was in.

The reclamation process is then triggered when a thread's local limbo list reaches a pre-defined threshold number of nodes. The reclamation procedure begins by reading the list of hazard pairs of all threads and by copying them in a local data structure, much like as in the hazard pointers method [58]. However, for the HHL method, this reading needs to be done twice and use the two copies of the hazard pairs before performing any reclamation of memory for a node. With only one read we cannot avoid the situation where a thread T_1 is not protecting N, when its hazard pair is read, and then T_1 accesses N before a second thread T_2, performing the delegation process, turns N unreachable. If the hazard pair for T_2 is read next, then it can happen that T_2 is not protecting N either by that time.

By doing two reads we can determine if N is unreachable in the first read, and if it is unreachable we can determine if it is reclaimable in the second read. So after reading twice the list of hazard pairs, a node N in the limbo list cannot be reclaimed if it is protected by any hazard pair $<HH, HL>$, i.e., if HH equals the hash value of N up to the hazard level HL and if HL is between the generation and the level tag of N. If such hazard pair exists, then the node is kept in the limbo list. Otherwise, the thread removes the node from its local limbo list and reclaims its memory.

Memory Bounds

Everything discussed so far is enough to make the memory reclamation method work. However, it does not ensure a finite memory bound if an infinite number of nodes is inserted and removed from a specific chain without ever triggering an expansion. The reason is that if a thread T suspends or fails in such a chain or if there is a group of threads continuously working on that chain in such a way that at any given time at least one is traversing it, this would prevent the

nodes removed from that chain from ever being reclaimed. To solve this problem, an expansion is forced on a chain if there are too many nodes that cannot be reclaimed and were removed from that chain. It does not guarantee the reclamation of the previously removed nodes, but prevents T from protecting more nodes. Later, when T progresses, the previously removed nodes would be made reclaimable as no thread can acquire a hazard pair for a chain that has already been expanded.

This solution not only provides 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 hash level is not a problem. In the last level, the solution is to traverse the chain exactly as in the hazard pointers method [58], 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. In conclusion the method achieves compatibility with the delegation problem present in the LFHT data structure while providing strong robustness, even if at the cost of ease of use and applicability.

3.3 LFHT on YAP

Prolog is the most popular and powerful logic programming language. Prolog gained its popularity mostly because of the success of the sophisticated compilation technique and abstract machine known as the Warren's Abstract Machine (WAM) presented by David H.D. Warren in 1983 [87]. Nowadays, it is widely used in multiple domains, such as, machine learning [70], program analysis [10], natural language analysis [69], bioinformatics [67] and semantic web [24]. Prolog systems represent data as terms, that can be number, strings, or atoms, or a composition of terms. Prolog atoms are particularly important, as they are both used as symbols and as a convenient representation of strings. In this work, we focus on the *Atom Table* used for atom management, and we investigate whether the traditional design can still be a good solution for recent challenges Prolog systems face.

The atom table is shared by all threads and, thus, needs to be highly efficient and scalable in order to ensure Prolog programs can take full advantage of multicore systems. As such, they are an ideal target to test the LFHT data structure in a real world scenario. We thus propose using the LFHT data structure to replace one of these internal data structures, namely the atom table that is used for symbol management, and verify if it is able to adequately perform in a real world use case.

In this Section, we start by introducing the YAP Prolog system focusing on the most relevant parts. Then, we will describe our work on replacing the lock-based implementation of the YAP atom table with the LFHT data structure, and finally we will present and discuss the obtained

results.

3.3.1 The YAP Prolog System

Prolog with multithreading has the ability to perform concurrent computations, in which each thread runs independently but shares the program clauses [66]. Almost all Prolog systems support some sort of multithreading. In particular, the multithreading library in the YAP Prolog system [76] can be seen as a high-level interface to the POSIX threads library, where each thread runs on a separate data area, but shares access to the global data structures (code area, atom table and predicate table). As each thread operates its own execution stack, it is natural to associate each thread with an independent computation that can run in parallel as threads already include all the machinery to support shared access and updates to the global data structures and input/output structures.

Figure 3.7 presents a high-level picture of the YAP system. The system is written in C and Prolog. Interaction with the system always starts through the top-level Prolog library. Eventually, the top-level refers to the core C libraries. The main functionality of the core C libraries includes starting the Prolog engine, calling the Prolog clause compiler, and maintaining the Prolog internal database. The engine may also call the just-in-time indexer (JITI) [77]. Both the compiler and the JITI rely on an assembler to generate code that is stored in the internal database. The C-core libraries further include the parser and several built-ins (not shown in Fig. 3.7). An SWI-Prolog compatible threads library [89] provides support to thread creation and termination, and access to locking.

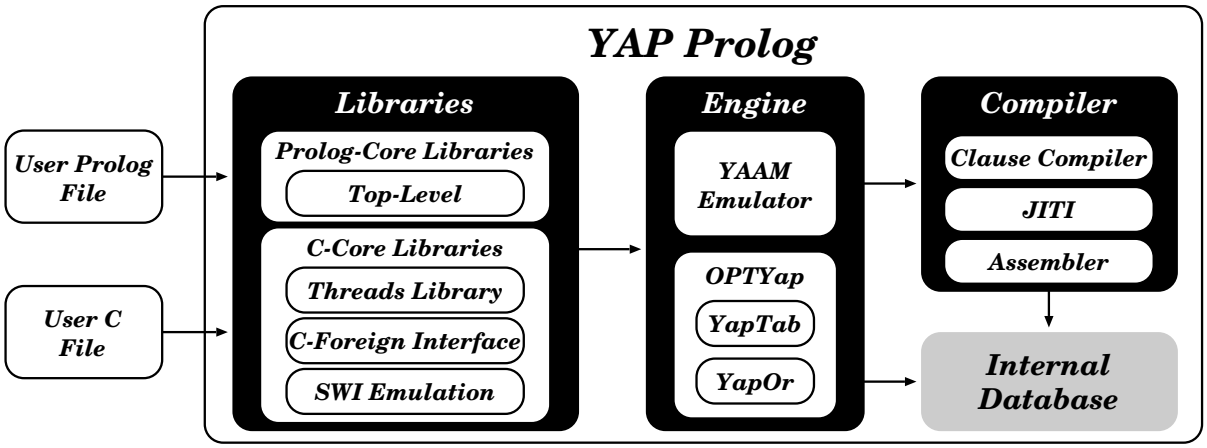


Figure 3.7: The YAP Prolog system

YAP includes two main components, the *Engine* and the *Database*. The Engine maintains the abstract machine internal state, such as abstract registers, stack pointers, and active exceptions. The Database maintains the root pointers to the internal database, which includes the *Atom Table* and the *Predicate Table*. In order to support multithreading, YAP's data structures are organized as follows:

- The `GLOBAL` structure is available to all threads and references the global data structures, locks should protect access to these data structures.
- The `LOCAL` structure is a per-thread array referencing the thread's local data structures, e.g., the engine abstract registers, internal exceptions, and thread specific predicates. The data is accessible through the thread's `LOCAL` structure, whose address is available from thread-local storage.

Figure 3.8 presents in more detail YAP's internal data structures with particular emphasis on the atom table. It assumes support for two threads, hence it requires two `LOCAL` structures, each containing a copy of the corresponding WAM registers.

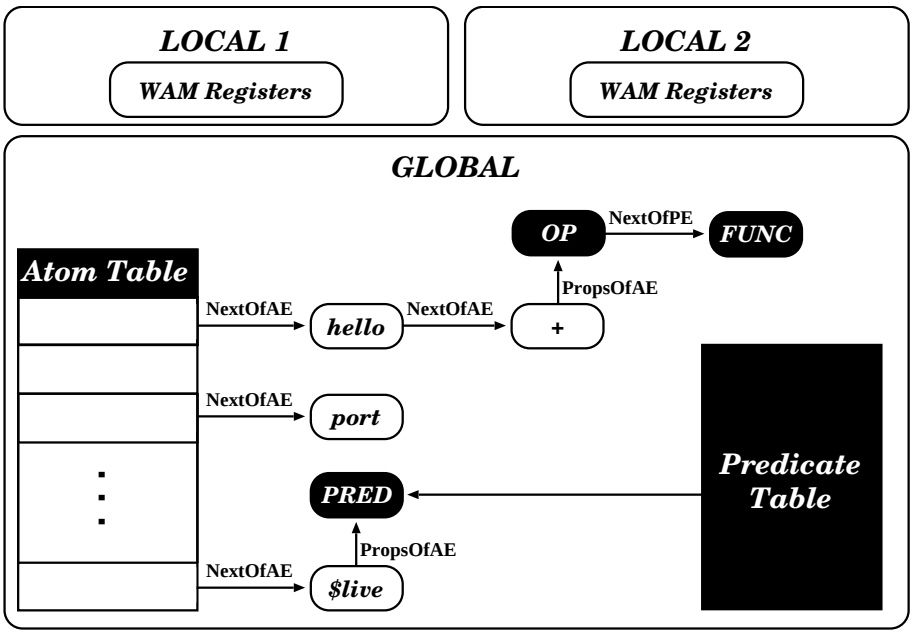


Figure 3.8: YAP's internal data structures

The main structure inside `GLOBAL` is the *Atom Table*, which contains objects of the abstract type `Atom`. As discussed above, atoms are used to represent symbols and text. The latter usage stems because the same text can appear in different parts of a program. Storing text as atoms can save both space and time, once to compare two segments one just has to compare atoms, e.g., two text segments match if and only if they are the same atom, that is, if they have the same entry in the atom table. At the implementation level, the atoms are stored in a linked list and each node within that linked list has a reference to a secondary linked list, that holds the properties of the atoms. Predicates with atoms as name are also stored in the atom table. Predicates are also often present in a Prolog program and there might exist several predicates with the same name (but with a different arity or belonging to different modules), and in such situations, there is a direct hash-table for them.

The abstract type `Atom` has a single concrete type, `AtomEntry`. Thus, the atom table is implemented as a single-level bucket array hash table with a separate chaining mechanism,

implemented as linked lists, to support collisions among `AtomEntry` objects. Once the bucket array data structure is saturated, the hash table triples its size, and the `AtomEntry` objects are placed in the newly created data structure. Each `AtomEntry` contains:

1. `StrOfAE`: a C representation of the atom's string;
2. `NextOfAE`: a pointer to the next atom in the linked list for this hash entry;
3. `PropsOfAE`: a pointer to a linked list of atom properties;
4. `ARMLOCK`: a readers-writer lock that serializes access to the atom.

The `Prop` type abstracts objects that we refer to by the atom's name. Example subtypes of `Prop` include functors, modules, operators, global variables, blackboard entries, and predicates. All of them are available by looking up an atom and following the linked list of `Prop` objects.

Figure 3.8 shows an atom table with four atoms: `hello`, `+`, `port`, and `$live`. Notice that only atoms + and `$live` have associated properties. The atom + has two properties, one of type `op` and another of type `functor`, and the atom `$live` has a property of type `predicate`. In practice, most atoms do not have properties. Every concrete type of `Prop` implements two fields:

1. `KindOfPE` gives the type of property;
2. `NextOfPE` allows organizing properties for the same atom as a linked list.

Each property extends the abstract property in its own way. As an example, *functors* add three extra fields: a back pointer to the atom, the functor's arity, and a list of predicates that share the same name and arity, but belong to different modules.

This design is based on LISP implementations, and has been remarkably stable throughout the history of the system.

3.3.2 Replacing the Atom Table

This section describes our proposal to improve the performance of YAP's atom table in concurrent environments. For that, we replaced the original version of the atom table, based in single level hashing, by the LFHT design in such a way that, instead of having a specialized version of a concurrent hash table implementing the atom table, we can simply use the general purpose LFHT design and allow it to manage everything, which goes from managing the concurrent accesses, to indexing the atoms for a faster access and handling atom collisions through a highly efficient chaining mechanisms. Moreover, to free memory from the atom table, we also take advantage of LFHT's memory reclamation mechanism described in Section 3.2, which will automatically handle the physical removal of atoms and corresponding internal data structures.

In what follows, we show in more detail how the LFHT data structure was integrated into the YAP system. To make the integration as smooth as possible, we need to understand all the details regarding YAP's internal database and how it is accessible from all internal and external libraries and data structures. Figure 3.9 presents the new organization of YAP's internal data structures based in the LFHT design (for comparison with Fig. 3.8, we left in gray the parts that were not changed from the original design). For the sake of presentation, the LFHT hash levels shown at the left of the figure are presented in a compact way as a single level, representing the initial configuration, which will be expanded during execution to multiple levels as described in the previous section.

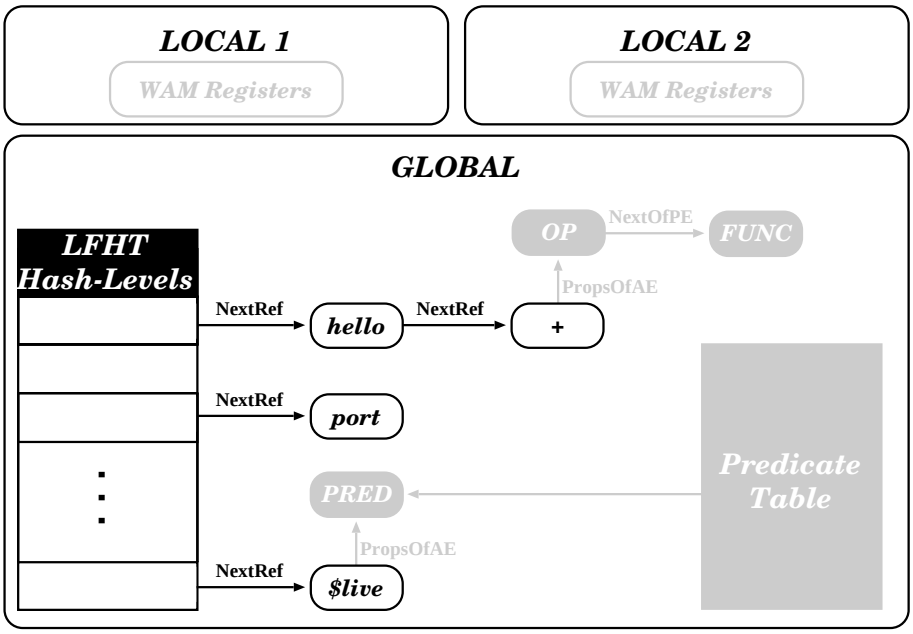


Figure 3.9: The new organization of YAP's internal data structures

When comparing the new organization in Fig. 3.9 with the previous one in Fig. 3.8, one can observe two main modifications. The original `NextOfAE` field was removed, since the chaining mechanism will be managed by LFHT's design, and the readers-writer lock `ARWLock`, used to serialize the access to the atoms in the original version of the atom table, was also removed, since now the LFHT design only uses CAS operations.

Using CAS operations instead of readers-writer locks has some advantages. It can significantly reduce the number of write operations done in memory during the execution of a program. At the implementation level, a readers-writer lock requires writing operations even when threads are only reading information from a protected memory region. This happens because readers-writer locks need to keep track of the number of threads that are in a protected memory region and, to do so, they use standard atomic counters. Moreover, these writing operations also require memory barriers to ensure the consistency of memory operations.

Note that LFHT does not completely avoid memory barriers, but requires fewer and weaker ones. The main gain comes from the fact that the design is lock-free, which means that reading

operations do not require any write operations.

The remaining data structures and references are unchanged. This is the case of the `PropsOfAE` pointer to the atom's *properties* and the `StrOfAE` representation of the atom's string, therefore allowing the other YAP's data structures, such as the Predicate Table, to still access the atoms' information as they do in the original design.

In order to fully replace YAP's atom table with LFHT's design, some additional extensions were required to ensure full compatibility with the original design. These extensions include: (i) support for arbitrary keys and full-hashing collisions; and (ii) an iteration mechanism capable of traversing all keys stored in the atom table in a given instant of time. In the following subsections, we discuss how these extensions were implemented.

Arbitrary Keys

By default, the LFHT implementation assumes that the hash function is good enough to avoid key collisions, meaning that it relies only on the generated hash value to find a key, thus not considering the case of two keys generating the same hash value. To also consider this situation, when searching for a key K, we still use the hash value h to move through the hash levels but, when a node N corresponding to h is found, we need to confirm that N holds K. And, if this is not the case, we keep searching for the next node corresponding to h that may hold K.

YAP's atom table uses strings as keys, and although we could add support for strings to LFHT's design, we decided to implement a more general solution independent of the type of the key. During LFHT's initialization, now we must give the following parameters: (i) a key comparison function; (ii) a hash function; and (iii) a key destructor function. The key comparison function should implement the comparison of keys to be used in the hash value searching mechanism. The hash function allows to simplify the API, since now we only need the key as argument to the LFHT operations instead of both the key and the hash value. The key destructor functions allows to free memory used by the key when we remove a node. We also allow for any of these parameters to be undefined, and in such case we disable the associated feature. For example, if no hash function is defined, we assume that the given key is the hash itself, if no key comparison function is passed, we assume that the user knows that hash values will not collide, and if no key destructor is passed, we assume that the key does not need a destructor. Listing 3.1 shows the new C language high-level API of the LFHT data structure.

The Iteration Procedure

During the execution of a program, a Prolog system might be required to iterate over all atoms present in the atom table. YAP is no exception, thus LFHT data structure was extended to support this additional operation. In a nutshell, the iterator of LFHT data structure presents atoms by the natural order that their hash value appears in the data structure for collision free

```
1  // Initializes the data structure and returns a handler
2  struct lfht_head *init_lfht(size_t (*hash_func)(void *),
3         int (*key_cmp)(void *, void *), void (*key_free)(void *));
4
5  // Returns the value associated with the key if it exists
6  void *lfht_search(struct lfht_head *head, void *key);
7
8  // Returns the value associated with the key if it exists,
9  // otherwise inserts the key with the provided value
10 void *lfht_insert(struct lfht_head *head, void *key, void *value);
11
12 // Removes the key and returns the associated value
13 void *lfht_remove(struct lfht_head *head, void *key);
14
15 // Returns the next key in hash/key order
16 void *lfht_next_key(struct lfht_head *head, void *key);
```

Listing 3.1: C language high-level API of the LFHT data structure

atoms, otherwise, the LFHT data structure consumes the atoms by the natural order of their keys.

At the implementation level, the iterator begins by presenting the atom with the lowest hash value. To present the next atom, it starts from the previously presented atom, and the process continues until there are no more atoms to be presented. If there are atoms with the same hash value, it presents the next smallest key with the same hash value. Otherwise, returns the smallest key of the next available smallest hash. By iterating this way, it ensures that iteration is done over all keys that were present when the iteration began and that were not removed during the iteration process. Keys that are inserted concurrently during an iteration might not be presented, this will happen if the iterator is iterating over a hash value which is higher than the hash value of the key that was inserted.

Algorithm 1 shows how the iteration process is done over the hash nodes, in order to find the next key. Note that we changed the order in which we read the hash value to be from the most significant bits to the least significant bits from the first level to the last level, so that we can have the property that nodes in a bucket $B[i]$ always have smaller hash values than nodes in a bucket $B[k]$ in the same hash node (for $i < k$). To find the first key we pass the $Null$ key to the *Iterator* function which lets us start at the bucket entry corresponding to the hash with value 0. Otherwise, we compute the hash value from the key and start iterating from the corresponding bucket. We begin in the root hash node and, if in the corresponding bucket we find a new hash node, we try to recursively find a next key in such hash node. If the bucket contains leaf nodes we call the *IterateChain()* function described in Algorithm 2 in order to find a next key in the chain. In both situations, if we find such a key we return it, otherwise we continue searching in the next bucket. If we reach the end of the hash node without finding a key, we return $Null$ in order to indicate no key was found.

Algorithm 1 *Iterate*(*Key k*, *Node Hn*)

```
1: if k = Null then
2:    h ⇐ 0
3: else
4:    h ⇐ Hash(k)
5: for i ⇐ Index(Hn, h) to Hn.size do
6:    if Hn.array[i].type = HASHNODE and Hn.array[i] ≠ Hn then
7:       R ⇐ Iterate(k, Hn.array[i])
8:    else if Hn.array[i].type = LEAFNODE then
9:       R ⇐ IterateChain(k, h, Hn.array[i])
10:   if R ≠ Null then
11:      return R
12: return Null
```

Algorithm 2 shows how we find the next node in a chain. We need to iterate over the whole chain as the nodes are unordered in the chain. We start by filtering the nodes that are actually ordered after the key provided, then we start by assigning the 1st node to N and replace it if we find a node that is ordered before it[1].

Algorithm 2 *IterateChain*(*Key k*, *Hash h*, *Node Ln*)

```
1: N ⇐ Null
2: while Ln.type = LEAFNODE do
3:    if Ln.hash > h or (Ln.hash = h and (k = Null or Ln.key > k)) then
4:       if N = Null or Ln.hash < N.hash or (Ln.hash = N.hash and Ln.key < N.key)
         then
5:          N ⇐ Ln
6:    Ln ⇐ Ln.next
7: return N
```

3.3.3 Experimental Results

In order to evaluate the impact of our proposal, we next show experimental results comparing the original and new versions of YAP's atom table. To put the results in perspective, we also compare both YAP's implementations with SWI-Prolog, a well-known and popular Prolog system that also implements concurrent support for the atom table in a lock-free fashion [90]. SWI-Prolog uses a single-level hash design to implement the atom table with lock-free operations, except for the resizing of the hash table, which is not lock-free because it uses a standard readers-writer locking scheme. This happens because while the resize is in progress, the next pointers linking atoms in the same bucket are generally incorrect, and dealing with this incorrectness is not a trivial task, which is solved with a standard readers-writer lock.

[1]Note that, for the sake of simplicity, we are omitting how the iterator proceeds when a concurrent expansion of hash nodes and the memory reclamation related operations occur.

The hardware used was a machine with 4 AMD Opteron™ Processor 8425 HE with 6 cores each, 64 KiB of L1 cache per core, 512 KiB of L2 cache per core and 5 MiB of usable shared L3 cache per CPU. It had a total of 128 GiB of DDR3 memory. The machine was running the Ubuntu 22.04 operating system with Linux kernel version 5.15.0-69. The results shown in the following figures were obtained by taking the mean of 10 benchmark runs.

Benchmark

We describe next the benchmark used to evaluate the performance of our implementation. In a nutshell, the benchmark will generate a huge stress over the Prolog's atom table, by inserting an enormous amount of atoms in a multithreaded fashion. Although it is an artificial benchmark, it is designed to expose all the potential bottlenecks in the atom table, allowing a deeper study about using the LFHT design in YAP. Next, we show the pipeline of predicates used in the benchmark.

```
1  % compile the generation sequences
2  :- compile('seq.pl').
3
4  % top query call
5  benchmark(WO, T):-
6     atom_dataset(DS),
7     % mark the inital time
8     statistics(walltime,[InitTime,_]),
9     % create and join threads
10    findall(Id, (between(1, T,_),
11                 thread_create(worker(DS, WO),Id)), Ids),
12    forall(member(I,Ids), thread_join(I,_)),
13    % mark the final time
14    statistics(walltime,[EndTime,_]),
15    Time is EndTime - InitTime,
16    % show the execution time
17    write('Time: '), write(Time).
```

Listing 3.2: Initial setup and top query call

We begin with Listing 3.2 showing the Prolog code for the initial setup of the benchmark and the *benchmark/2* predicate, which is the top predicate to be called. We start by compiling an initial set (file `seq.pl`) of 240,000 different sequences that will be used as base sequences to generate a combination of multiple atoms to be inserted in the atom table. The *benchmark/2* predicate is then used to mark the initial and final times, create and join threads, and to show the execution time. It receives two arguments, the worker offset WO, used to batch a set of sequences from the initial set that will be used to create the combination of atoms, and the total number of threads T to be executed. For this benchmark, we used a batch of 2,000 sequences of work to be done.

```
1  % setup scheduler
2  :- dynamic qsize/1.
3  :- mutex_create(qlock).
4  qsize(0).
5
6  % manage the working queue
7  worker(DS, WO) :-
8     mutex_lock(qlock),
9     % get work from queue
10    qsize(I),
11    (I =< DS -> % thread got work W
12       % setup next work
13       retract(qsize(I)),
14       IL is I + WO, assert(qsize(IL)),
15       mutex_unlock(qlock),
16       % compute work W
17       compute(I, IL),
18       % get more work
19       worker(DS, WO);
20       % no more work to be done
21       mutex_unlock(qlock)).
```

Listing 3.3: The naive parallel scheduler

The second stage of the pipeline is the scheduler. Listing 3.3 shows the code that implements the naive parallel scheduler used in the benchmark. It uses a dynamic predicate *qsize/1* to mark the number of the next sequence from the initial set that is available to be used for the generation of atoms and a standard lock named *qlock* to synchronize threads when they are getting work. To get work, a thread T begins by gaining access to the lock, then it reads the next sequence I stored in *qsize/1* and, if there is work to be done, T prepares the queue with the next available sequence IL, releases the lock and goes to executing work. Otherwise, there is no more work to be done, thus T keeps *qsize/1* in the same state, releases the lock, and proceeds to the thread join predicate.

The third and final stage of the pipeline implements the process of generating atoms to be inserted and stored in the atom table. Listing 3.4 shows both *compute/2* and *combine_atoms/2* predicates. For each batch of work, a thread uses the *compute/2* predicate to get the corresponding sequences from the initial set, and, for each sequence, it calls the *combine_atoms/2* predicate to generate all possible combination of atoms from the sequence. Each generated atom is then automatically inserted by the Prolog system in the atom table.

Results

To analyze the behavior caused by having different ratios of searches and inserts in the benchmark, we experimented it by preloading the atom table with a varying portion of the dataset in order

```
1  % compute the sequences
2  compute(I, I) :- !.
3  compute(I, IL) :-
4     atom_seq(I, AS),
5     (combine_atoms(AS, _), fail; true),
6     I1 is I + 1,
7     compute(I1, IL).
8
9  % generation of atoms
10 combine_atoms(AS, R) :-
11    atom_concat(A1, A2, AS),
12    atom_concat(A2, A1, R).
```

Listing 3.4: Generation of the atoms to be inserted in the atom table

to pre-insert part of the atoms in the atom table and thus increase the ratio of searches when running the benchmark. We considered three different scenarios: no pre-insertion of atoms; pre-insertion of 50% of the atoms; and pre-insertion of all atoms (i.e., only searches are done during benchmark execution).

Each benchmark run results in a total of 7,680,000 atom table operations, of which 3,737,741 are inserts in the scenario with no pre-insertion, thus corresponding to a ratio around 51% searches and 49% insertions in the atom table, and to a total of 562,723 expansions in the LFHT data structure. The scenario with 50% pre-insertion results in 1,924,444 insert operations, which corresponds to around 75% searches and 25% insertions. With 100% pre-insertion, all operations are searches. In the original YAP implementation, the atom table grows in size a total of 4 times, tripling in size in each time it grows, which happens when the program is being loaded and the internal atoms are inserted, and so, before the benchmark begins.

Figure 3.10 shows the speedup obtained by YAP with the atom table replaced by the LFHT data structure against YAP's original implementation for every combination of 1 to 24 threads with no pre-insertion. The results show that, on average, we can achieve a minimum speedup of 1.8 with a single thread and a maximum speedup around 3.4 with 23 threads. The speedup for 24 threads is slightly worse than for 23 threads because, as the LFHT version has better CPU utilization, it is more affected by background/operating system processes when all cores are in use.

These results show that we can achieve not only better overall performance, but also much better scalability. In particular, the readers-writer locks present in the original atom table can be a significant bottleneck that the LFHT data structure is able to avoid.

To put the results in perspective, we also compared the YAP results with SWI-Prolog. Figures 3.11, 3.12 and 3.13 show the throughput of sequences that are computed per second in both the YAP (original and LFHT-based atom tables) and SWI-Prolog implementations for the three pre-insertion scenarios.

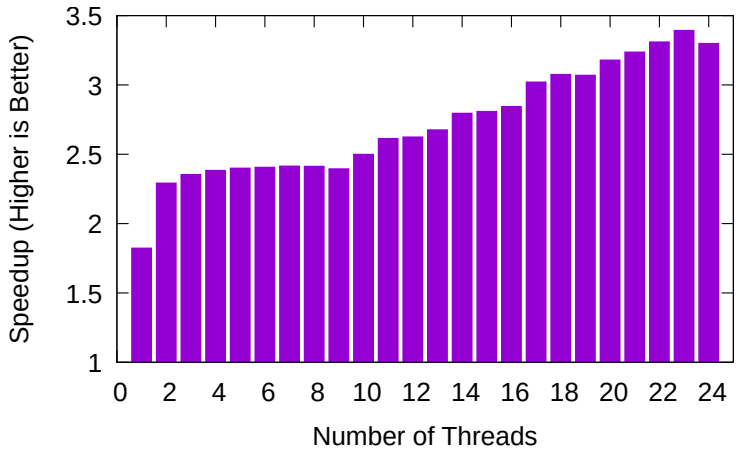


Figure 3.10: Speedup of YAP's LFHT version against YAP's original implementation (no pre-insertion scenario)

Figure 3.11 shows the throughput without any pre-insertion. In the YAP implementation with LFHT, it shows values from 28549.4 sequences per second with 1 thread to 402064.4 sequences per second for 24 threads, resulting in a speedup of 14.08. In the original YAP implementation, it shows values from 15658.8 sequences per second with 1 thread to 121872.6 sequences per second for 24 threads, resulting in a speedup of 7.78. In the SWI-Prolog implementation, it shows values from 19106.2 sequences per second with 1 thread to 16365.9 sequences per second for 24 threads resulting in a slowdown of 0.86.

Figure 3.12 shows the throughput with 50% pre-insertion. In the YAP implementation with LFHT, it shows values from 28187.6 sequences per second with 1 thread to 598618.9 sequences per second for 24 threads resulting in a speedup of 21.24. In the original YAP implementation, it shows values from 15476.4 sequences per second with 1 thread to 181063.9 sequences per second for 24 threads resulting in a speedup of 11.70. In the SWI-Prolog implementation, it shows values from 19010.8 sequences per second with 1 thread to 24247.0 sequences per second for 24 threads resulting in a speedup of 1.28.

Figure 3.13 shows the throughput with 100% pre-insertion. In the YAP implementation with LFHT, it shows values from 30363.8 sequences per second with 1 thread to 696013.3 sequences per second for 24 threads resulting in a speedup of 22.92. In the original YAP implementation, it shows values from 14096.4 sequences per second with 1 thread to 255306.8 sequences per second for 24 threads resulting in a speedup of 18.11. In the SWI-Prolog implementation, it shows values from 19544.7 sequences per second with 1 thread to 39331.4 sequences per second for 24 threads resulting in a speedup of 2.01.

As one can observe, the original YAP implementation already provides much better performance and scalability than SWI-Prolog, and the LFHT-based atom table is able to provide a considerable improvement on top of it. For example, with 24 threads, our LFHT-based implementation is able to achieve 24.6 times the throughput of SWI-Prolog in the no pre-insertion scenario. In the

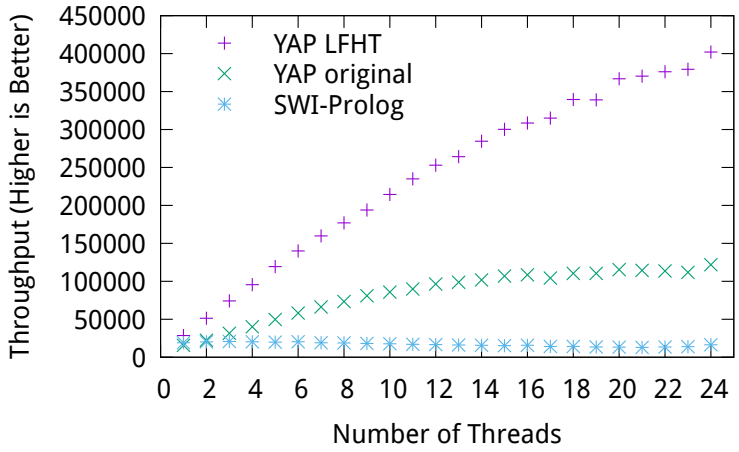


Figure 3.11: Throughput for YAP and SWI-Prolog (no pre-insertion scenario)

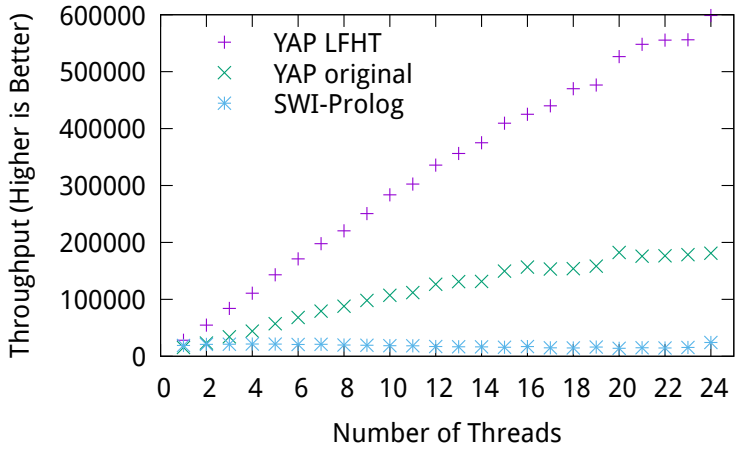


Figure 3.12: Throughput for YAP and SWI-Prolog (50% pre-insertion scenario)

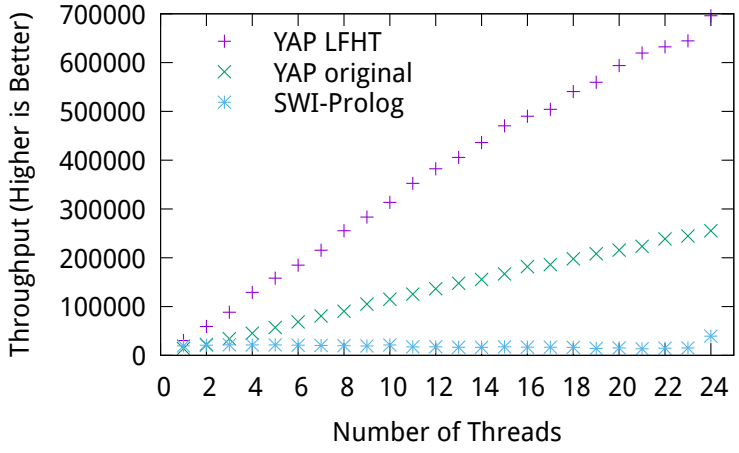


Figure 3.13: Throughput for YAP and SWI-Prolog (100% pre-insertion scenario)

scenarios with pre-insertion, we see better throughput overall as the amount of insert operations are replaced by the cheaper search operations. The original YAP implementation sees the most relative benefit due too its use of readers-writer locks, specially in the 100% pre-insertion scenario where it achieves almost linear scalability, but even so is unable to come close to the lock-free implementation's throughput.

3.4 Compression

The main disadvantage of trie based hash maps is the higher depth compared to table-based hash maps. The higher depth requires more memory accesses in order to reach the desired location on the data structures. And, as memory access is the main bottleneck in most data structures, specially if they contain large amounts of data that does not fit in the CPU cache, minimizing the depth and consequently the number of memory accesses can significantly increase performance. By using a compression mechanism we can reduce the depth of the LFHT data structure while keeping most of its advantage compared to the table-based counterparts.

In this Section, we will start by introducing our compression design on the LFHT data structure by example. Then we will show the details behind the implementation followed by the supporting algorithms. Next, we will show the experimental results obtained by such changes to the LFHT data structure.

3.4.1 Design by Example

The key idea of our design is to apply *lock-free compression* to clusters of hash nodes in order to reduce the average depth of hash levels needed to be traversed within the hash map. In a nutshell, to activate compression on a cluster, the condition is to have a hash node (called the *head node* of the cluster of hash nodes) with all its bucket entries referring other hash nodes. A second condition is that the head node does not belong to a second cluster where another compression is undergoing. If two or more compressions intersect in at least one hash node, then priority is given to the compression whose head node has the lowest depth (i.e., nearest to the root of the hash map). Non-priority compressions will be postponed (or aborted) until the most priority one completes. At the end of a compression, the cluster of hash nodes is replaced by a single hash node representing the cluster and the depth of any path traversing the cluster is reduced in one level.

Figure 3.14 shows an example of applying *lock-free compression* to a cluster of hash levels. For the sake of simplicity of illustration, we consider that hash nodes are initially allocated with two bucket entries and that R_1 to R_6 represent references to arbitrary hash or leaf nodes. Figure 3.14(a) shows the initial configuration where one can observe the existence of two clusters of hash nodes: cluster C_1 with head node H_i and including H_k and H_l; and cluster C_2 with head node H_k and including H_m and H_n. Since H_k, the head node of C_2, also belongs to C_1,

priority is given to the compression of cluster C_1. Figure 3.14(b) shows the configuration after the compression of cluster C_1 where one can observe that H_i, H_k and H_l were replaced by a single new hash node H_x that has twice the size of bucket entries (four bucket entries in this case).

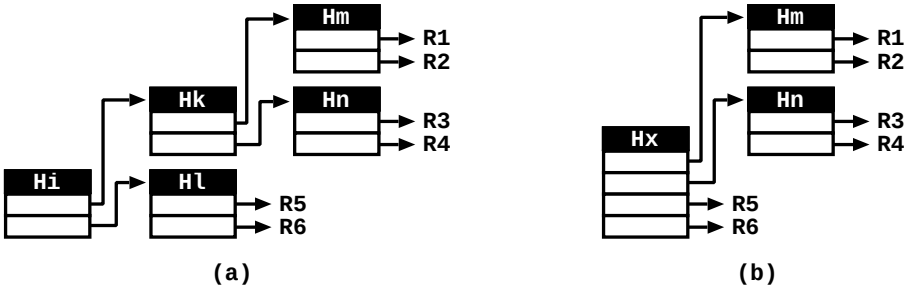


Figure 3.14: Compression of a cluster of hash levels

Consider a thread traversing the configuration in Fig. 3.14 looking for reference R_3. Without compression (Fig. 3.14(a)), the thread begins by visiting H_i, then follows the reference in the first bucket to access H_k, next the reference in the second bucket to access H_n, and finally the reference in the first bucket to reach R_3. In the worst case, if the header and the corresponding bucket entry for each hash node do not fit inside the same cache line, reaching R_3 will require six memory accesses (two times the number of hash levels). After compression (Fig. 3.14(b)), the thread begins by visiting H_x and reaching R_3 requires one less hash level, corresponding to four memory accesses, in the worst case.

Let us consider now that lock-free compression is first triggered and successfully applied to C_2 and only then H_l is concurrently added to the hash map data structure to form cluster C_1. Figure 3.15(a) shows the resulting configuration, where one can observe that H_k, H_m and H_n were replaced by a single new hash node H_z with four bucket entries. As before, the access to references R_1, R_2, R_3 and R_4 were all reduced by one level, but the access to R_5 and R_6 remains unchanged and still requires traversing two hash levels. This illustrates one of the advantages of prioritizing the compressions near the root of the hash map. A second advantage is that the application of compressions following the priority order of being near the root of the hash map converges to a *canonical structure*, while any other order of application can lead to different configurations at the end. A key motivation of lock-free compression is that regardless of which cluster is compressed first, the hash hierarchy will converge to a canonical structure. We next discuss how this is done in our design.

Starting from the configuration in Fig. 3.15(a), we now have a cluster C_1 formed by the head node H_i and including H_z and H_l. The problem is that, due to the fact that H_z already represents two hash levels as a result of a previous compression, H_z and H_l have a different number of bucket entries (4 and 2 entries, respectively) and, therefore, we cannot replace cluster C_1 by a single new hash node, as done previously. Figure 3.15(b) and Fig. 3.15(c) show two alternative approaches for compressing C_1 in this case.

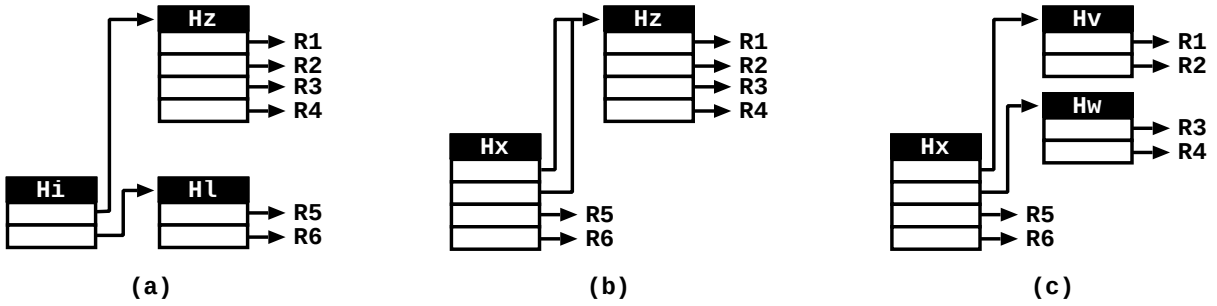


Figure 3.15: Splitting of previously compressed hash levels

The approach illustrated in Fig. 3.15(b) tries to preserve previous compressions. A new hash node H_x is introduced to represent C_1 (thus replacing H_i and H_l) but H_z is maintained. As intended, this approach succeeds in reducing the access to R_5 and R_6 in one level. However, since H_z represents two hash nodes, the first two bucket entries of H_x are made to refer to H_z. This violates an invariant of the LFHT design, which requires not having more than one bucket entry referencing the same hash node, and makes it impossible to swap references to hash nodes with just a single word CAS operation, limiting further compressions.

The approach illustrated in Fig. 3.15(c) tries to preserve the canonical structure. Since H_z represents a lower priority compression, it proceeds by undoing the previous compression and, for that, it splits H_z in two hash levels (H_v and H_w in Fig. 3.15(c)), each with half the bucket entries. Then, a new hash node H_x is still introduced to represent C_1, thus replacing H_i, H_l and part of H_z. As before, this approach succeeds in reducing the access to all references in one level, but now there are no duplicate references to hash nodes in H_x. One can observe that this configuration is identical to the one presented in Fig. 3.14(b), which represents the canonical form. This example shows that, regardless of the order of cluster compression, the hash hierarchy will converge to a canonical structure although, as in this situation, the compress operation would require extra steps.

3.4.2 Implementation Details

Starting from the high-level description of the previous section, we now discuss in more detail how lock-free compression is implemented on top of the LFHT data structure. Such detail is important since we want to show that lock-free compression is implemented by following a well-defined sequence of CAS operations. To implement lock-free compression, the following extensions were made to the LFHT data structure: (i) bucket entries now include a *freeze flag* that, when set, indicates that further updates cannot be made to the corresponding bucket entry; and (ii) the header of the hash nodes now includes a *compression representative* field, which refers to the new hash node representing the cluster being compressed, and a *compression count* field, which counts the number of bucket entries referring to hash nodes (and is used to trigger compression). Note that we also keep the modified hash reading order from Section 3.3.2, as it simplifies the order between entries during compression.

Figure 3.16 details the sequence of steps involved in the compression of a *standard* cluster of hash nodes, i.e., without splitting. For that, it considers a bucket entry B_k referring to a cluster with head node H_i and including H_k and H_l. As before, R_1, R_2, R_3 and R_4 represent references to arbitrary hash or leaf nodes.

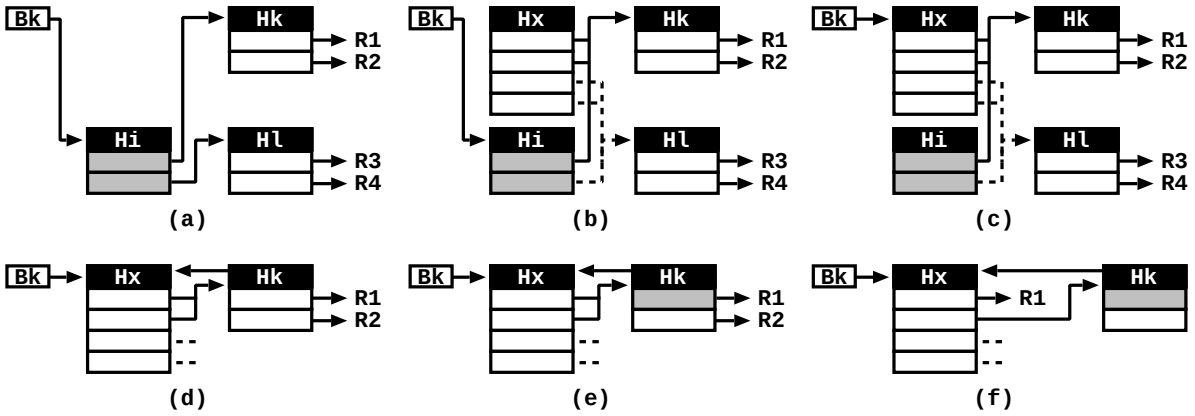


Figure 3.16: A step by step compression operation without splitting

Figure 3.16(a) shows the first step of the compression procedure, where CAS operations are used to set the freeze flag of each bucket entry in the head node H_i (in what follows, frozen entries are marked gray). Remember that a frozen entry remains unchanged for the remaining lifetime. This freezing process is important because it implements the strategy where priority is given to the compression whose head node has the lowest depth. For example, if a lower priority compression is being done on cluster with head node H_k, it will be aborted because it cannot update the corresponding first (frozen) bucket entry of H_i.

Next, Fig. 3.16(b) shows the second step of the compression procedure, where a new hash node H_x is first allocated and then initialized by copying the references from the bucket entries in H_i. In this case, since H_k and H_l are default sized (non-compressed) hash nodes, the size of H_x corresponds to doubling the size of H_i, and each pair of bucket entries in H_x is initialized to match the corresponding H_i's entry. The size of H_x should match $sizeof(H_i) \times min(sizeof(H_k), sizeof(H_l))$ For example, the first two bucket entries of H_x are set to H_k, which is the reference in H_i's first entry, whereas the second two bucket entries of H_x are set to H_l, which is the reference in H_i's second entry. After this initialization, H_x is ready to be inserted in the LFHT data structure and, for that, a CAS operation is applied to B_k trying to replace H_i with H_x. Figure 3.16(c) shows the resulting configuration. It is important to notice that lock-freedom requires that, at any moment of the compression procedure, no thread can be blocked from traversing and accessing the available hash and leaf nodes. Figure 3.16(c) show us that, even in a scenario where a thread T is preempted in H_i, T is still able to traverse forward to the deeper levels H_k and H_l.

At this point, it is also important to notice that the configuration in Fig. 3.16(c) violates the invariant of not having more than one bucket entry referencing the same hash node. However, here, this is not a problem because the bucket entries in H_x are not yet the synchronization

points for further updates on the cluster, since they are still referring to H_k and H_l. Thus, future steps involve copying R_1 to R_4 from the bucket entries of H_k and H_l to the bucket entries of H_x. Figure 3.16(d) to Fig. 3.16(f) show how this is done for reference R_1. The same process applies to the remaining references (not shown here to simplify the illustration).

The next step is to set the new compression representative (header) fields of H_k and H_l to refer to H_x. Figure 3.16(d) shows the configuration after setting the compression representative field of H_k. The same process applies to H_l (not shown here to simplify the illustration). Note that copying the references R_1 and R_2 to H_x, will turn H_k invalid. The compression representative field implements a kind of reconnection path for invalid hash nodes. For example, in a scenario where a thread T is preempted in H_k and H_k turns invalid, the compression representative field allows T to recover to H_x.

The final steps involve freezing the first bucket entry of H_k, meaning that no further updates can be done there, and applying a CAS to the corresponding bucket entry in H_x in order to update it to R_1. Figure 3.16(e) shows the configuration after the freezing and Fig. 3.16(f) shows the configuration after the updating of R_1 in H_x. The same process is applied afterward to the remaining bucket entries in H_x, adjusting R_2, R_3 and R_4, to finish the compression procedure. Note that these final steps do not violate the lock-freedom property of a search, insert, remove or expand operation being done concurrently, since the synchronization point in H_k is being moved to the corresponding bucket entry in H_x. In other words, an operation that would require updating the frozen bucket entry in H_k, will now follow the compression representative field to reach H_x and change the corresponding bucket entry there.

We conclude this section by describing a second compression situation, but now for a scenario leading to the splitting of previously compressed hash levels, as illustrated in Fig. 3.15. Figure 3.17 details the sequence of steps involved in the compression of a cluster with head node H_i and including H_z and H_l, where H_z is already the result of a previous compression.

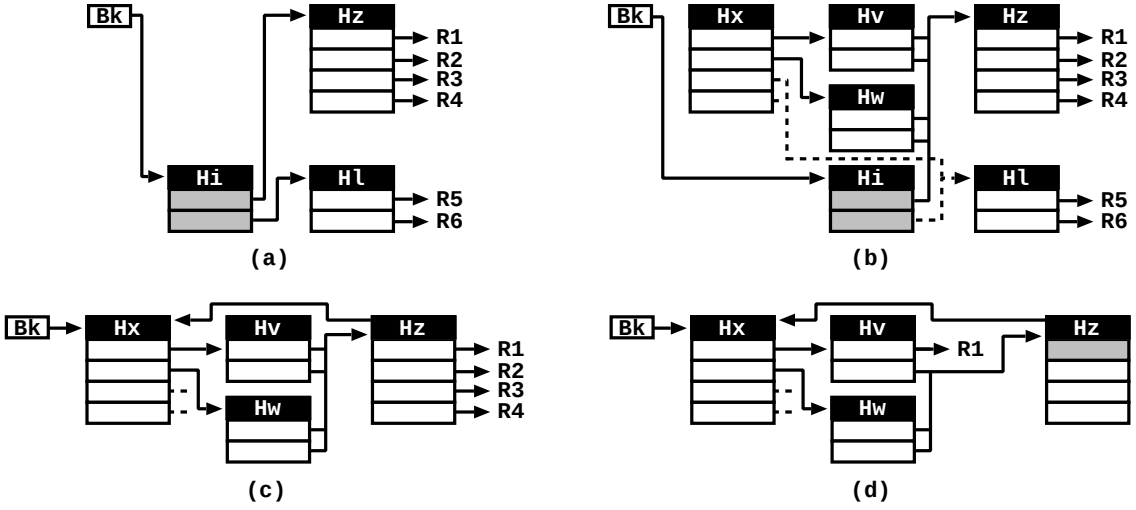


Figure 3.17: A step by step compression operation with splitting

As before, Fig. 3.17(a) shows the first step of the compression procedure, where CAS operations are used to set the freeze flag of each bucket entry in the head node H_i. Then, Fig. 3.17(b) shows the second step of the compression procedure, where new hash nodes H_x, H_v and H_w are first allocated (H_v and H_w representing the splitting of H_z in two hash levels, each with half the bucket entries) and then initialized by copying the references from the bucket entries in H_i. Next, Fig. 3.17(c) shows the configuration after updating the compression representative field of H_z to refer to H_x. The same process applies to H_l (not shown here to simplify the illustration). Note that H_v and H_w are not set as representative as, in general, this would require not a single representative field but an array of representatives (equal to the number of bucket entries per hash node). Finally, Fig. 3.17(d) shows the configuration after freezing the first bucket entry of H_z and after applying a CAS to the corresponding bucket entry in H_v in order to update it to R_1. The same process is then applied to the remaining bucket entries in H_v and H_w, adjusting references R_2 to R_6, to finish the compression procedure.

3.4.3 Algorithms

This section presents the key algorithms required to easily reproduce our implementation. We begin with Alg. 3 showing the pseudocode for the lock-free compression procedure for a given head node H_i.

Algorithm 3 *Compression*(*hash node* H_i)

```
1: FreezeBucketEntries(H_i)
2: H_x ← CompressionInit(H_i)
3: if CompressionCommit(H_i, H_x) then
4:     CompressionReps(H_x)
5:     CompressionRefs(H_x)
```

FreezeBucketEntries() starts by implementing the first step of the compression procedure, as shown in Fig. 3.16(a) and Fig. 3.17(a). Then, *CompressionInit*() implements the second step, as shown in Fig. 3.16(b) and Fig. 3.17(b). Next, the conditional call to *CompressionCommit*() implements the step where H_i is replaced by H_x, as shown in Fig. 3.16(c). If it fails, meaning that there is an overlapping high priority compression being done, this compression is aborted. Otherwise, *CompressionReps*() sets the compression representative fields, as shown in Fig. 3.16(d) and Fig. 3.17(c), and *CompressionRefs*() updates the references in the bucket entries of the new hash nodes, as shown in Fig. 3.16(e–f) and Fig. 3.17(d). Pseudocode for *CompressionInit*(), *CompressionReps*() and *CompressionRefs*() is presented in more detail in Alg. 4, 5 and 6, respectively.

In these algorithms, *HS* is the default number of bucket entries for a standard hash node. In Alg. 6, the *comprCount* field counts the number of bucket entries in a hash node referring to deeper hash nodes and is used to trigger lock-free compression when all bucket entries are referring to deeper hash nodes (lines 18–21 in Alg. 6).

Algorithm 4 *CompressionInit*(*hash node* H_i)

```
1:  H_x ← AllocHashNode(H_i.size × HS)
2:  for i ← 0 to H_i.size do
3:      H_j ← H_i.array[i]
4:      if H_j.size = HS then
5:          for j ← 0 to HS do
6:              H_x.array[i × HS + j] ← H_j
7:      else {splitting case}
8:          for j ← 0 to HS do
9:              H_v ← AllocHashNode(H_j.size ÷ HS)
10:             H_x.array[i × HS + j] ← H_v
11:             for v ← 0 to H_v.size do
12:                 H_v.array[v] ← H_j
13: return H_x
```

Algorithm 5 *CompressionReps*(*hash node* H_x)

```
1:  i ← 0
2:  while i < H_x.size do
3:      H_k ← H_x.array[i]
4:      if H_k.level ≠ H_x.level then {splitting case}
5:          H_k ← H_k.array[0]
6:      H_k.comprRepresentative ← H_x
7:      i ← i + HS
```

3.4.4 Performance Analysis

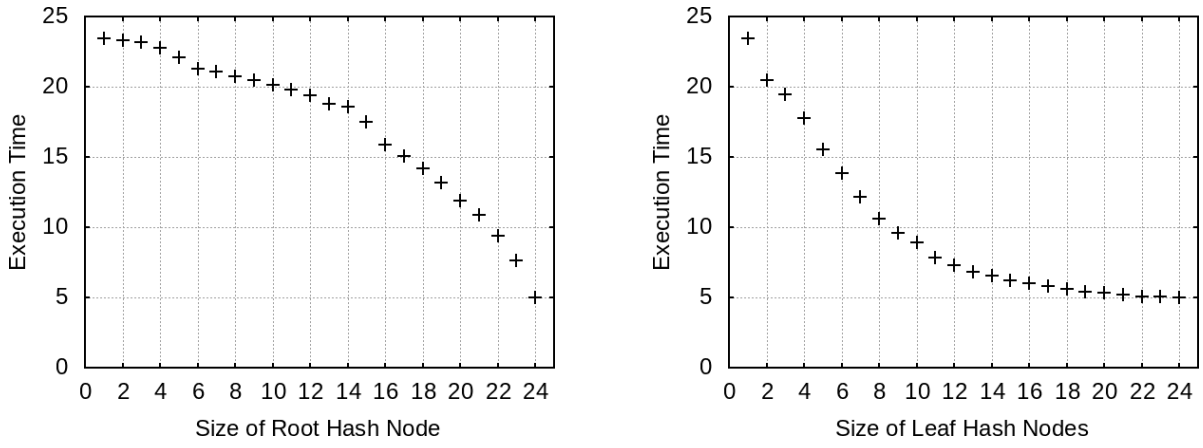
The environment for our experiments was a SMP system based in a NUMA architecture with two Intel Xeon X5650, each having 6 cores (12 threads) at 2.66 GHz, 12 MB Intel Smart Cache, 96 GB of main memory, and running the Linux kernel 4.15.0-72. To measure execution time, all programs were compiled with GCC 9.2.0 with -O3 and using the jemalloc memory allocator 5.0. We ran each benchmark 5 times and took the average of those runs.

Compression Benefits

Compression benefits heavily rely on the memory environment where we are running our benchmarks. Factors like cache sizes, placement policies and prefetching optimizations can have a significant impact on the overall performance of the LFHT design. To put our results in perspective, first we ran a specific benchmark designed to address the potential gains that one would expect to have when using compression. For that, we used a static version of the LFHT design that implements fixed predefined configurations of hash levels, with a different number of bucket entries on each hash node, and we measured the execution time for one thread performing

Algorithm 6 *CompressionRefs*(*hash node* H_x)

```
1:  xCount ← 0
2:  for x ← 0 to H_x.size do
3:      H_k ← H_x.array[x]
4:      if H_k.level = H_x.level then
5:          R ← FreezeBucketEntry(H_k.array[x mod HS])
6:          CAS(H_x.array[x], H_k, R)
7:          if R.type = HASHNODE then
8:              xCount ← xCount + 1
9:      else
10:         xCount ← xCount + 1
11:         kCount ← 0
12:         for k ← 0 to H_k.size do
13:             H_z ← H_k.array[k]
14:             R ← FreezeBucketEntry(H_z.array[(x mod HS) × H_k.size + k])
15:             CAS(H_k.array[k], H_z, R)
16:             if R.type = HASHNODE then
17:                 kCount ← kCount + 1
18:         if AtomicAdd(H_k.comprCount, kCount) = H_k.size then
19:             Compression(H_k)
20: if AtomicAdd(H_x.comprCount, xCount) = H_x.size then
21:     Compression(H_x)
```



(a) Compression from root to leaves

(b) Compression from leaves to root

Figure 3.18: LFHT's compression effects for 2^{24} search operations with one thread

only search operations on those configurations.

Starting from a maximal configuration of 24 uncompressed hash levels, all with the same minimal size of 2^1 bucket entries, we studied the effect of applying two different types of compression operations: (i) by reducing the number of hash levels from the root hash node to the leaf hash nodes; and (ii) by reducing the number of hash levels from the leaves to the root. Figure 3.18 shows the execution time, in seconds, for executing 2^{24} search operations with one thread when reducing the number of hash levels in both directions (Fig. 3.18(a) for the root to leaves compression and Fig 3.18(b) for the leaves to root compression) until reaching the configuration with just a single hash node with 2^{24} bucket entries. The x-axis represents the number of hash levels compressed in a configuration. In both figures, the x-axis value of 1 represents the maximal configuration of 24 uncompressed hash levels and the x-axis value of 24 represents the single fully compressed hash node with 2^{24} bucket entries. The other x-axis values represent intermediate configurations. For example, the x-axis value of 10, in Fig. 3.18(a) represents the configuration whose first hash node includes 2^{10} bucket entries followed by 14 uncompressed hash levels, and in Fig. 3.18(b) represents the configuration with 14 initial uncompressed hash levels followed by a final hash level with hash nodes with 2^{10} bucket entries. In Fig. 3.18(a), one can observe that, for root hash nodes with less than 2^{14} bucket entries, the benefits are small, but then, for higher compression ratios, the results show a significant impact on reducing the execution time. This happens because most of the execution time is spent on waiting for memory accesses and because the hash nodes closest to the root tend to remain in cache. Consequently, compressing the first 14 levels only reduces the amount of cache accesses, which results in a poor impact on the total executing time. On the other hand, further compression is able to effectively reduce the number of memory accesses.

In Fig. 3.18(b), one can observe that compressions up to a size of about 2^{10} are quite effective in reducing the execution time, whereas after that size they are not as much. This can be explained by the fact that, after a certain size, the benefits of compression are absorbed by the caching effects.

As a result of this study, in what follows, we have chosen to set the root hash node of the LFHT design with 2^{16} bucket entries, thus ensuring that compressions would have an impact in the execution time. This will create a memory overhead, which can be considered negligible, since it amounts to just 512 KiB. All the other hash nodes, allocated during execution, begin with 2^4 bucket entries, which is the minimum size allowed by the original LFHT design.

Performance Results

In this subsection, we analyze the performance of our compression design in three different scenarios: (i) *Search Only*, where threads search for N keys in a hash map with the N keys inserted; (ii) *Insert Only*, where threads insert N keys in an empty hash map; and (iii) *Remove Only*, where threads remove N keys in a hash map with N random keys. On each scenario, we used two sets of N random keys, namely 10^8 and 10^9 keys. To support concurrent randomicity on

each thread, we used glibc PRNG (Pseudo Random Number Generator), such that, for insertions we just insert random keys by giving each thread a different seed, and for search and remove, we reuse the seeds used for insertion, ensuring that we search or remove each key only once.

Figure 3.19 shows throughput results (higher is better) comparing our compressed design (LFHT-Compress) against the original design (LFHT-Original), and the Concurrent Hash Map design (CHM) of Intel-TBB library [75], when running a number of threads from 1 to 24 with 10^8 and 10^9 keys in the three previously mentioned scenarios.

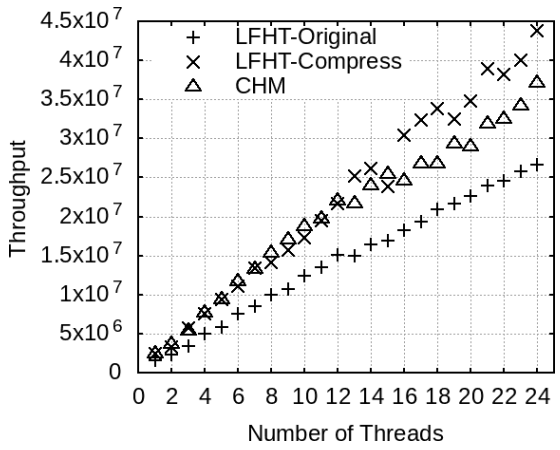
Figure 3.19(a) and Fig. 3.19(b) show throughput results for the *Search Only* scenario. Comparing the two LFHT designs, one can observe that LFHT-Compress obtains improvements against LFHT-Original of around 50% with 10^8 keys and around 100% with 10^9 keys. When comparing against CHM, LFHT-Compress has almost always the best results, with CHM very close. This can be explained by the fact that the final configuration of both designs is quite similar, since CHM also uses only a root hash level to do the initial scatter of keys.

Figure 3.19(c) and Fig. 3.19(d) show throughput results for the *Insert Only* scenario. Comparing the two LFHT designs, one can observe that both achieve similar results for 10^8 keys but LFHT-Compress is clearly better for 10^9 keys. Even though LFHT-Compress is doing more work by compressing hash levels, it is able to improve the overall throughput. This happens because the cost of doing extra work on compression is compensated by the shorter paths leading to the insertion points. When comparing with CHM, LFHT-Compress has almost always the best results, however in this scenario the difference is more significant as we increase the number of threads. One reason that can explain this difference is the fact that, since CHM is lock-based, it seems unable to scatter the concurrency spots as we increase the number of threads, since each lock is being used to block a large portion of paths within the hash map. On the other hand, since LFHT-Compress is lock-free, it is able to control the concurrency spots with the fine grain given by the CAS operation.

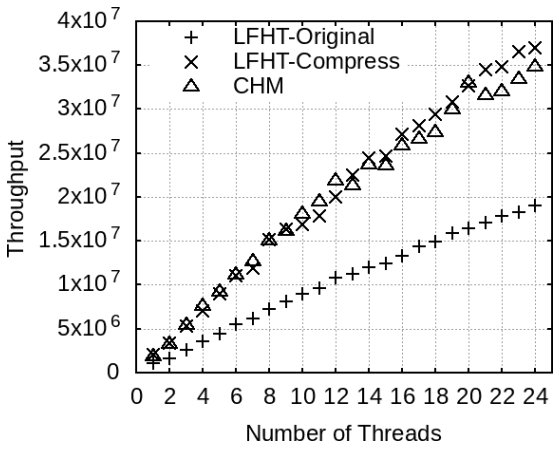
Finally, Fig. 3.19(e) and Fig. 3.19(f) show throughput results for the *Remove Only* scenario. Comparing the two LFHT designs, one can observe that LFHT-Compress is again better than LFHT-Original and that the difference increases as we increase the number of threads. This can be explained by the gains observed for LFHT-Compress on the search operation. When comparing with CHM, LFHT-Compress is again better by far than CHM in the 10^8 scenario, with the difference increasing as the number of threads increases, whereas in the 10^9 scenario the difference is almost constant. This can be explained by the same reasons mentioned before for the *Insert Only* scenario (lock-based vs lock-free).

3.5 Generally Solving the Delegation Problem

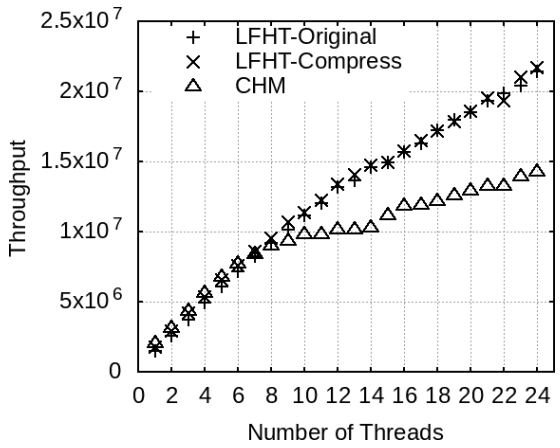
In this Section, we describe two general solutions to the delegation problem introduced in order to make the LFHT data structure compatible with most SMR methods.



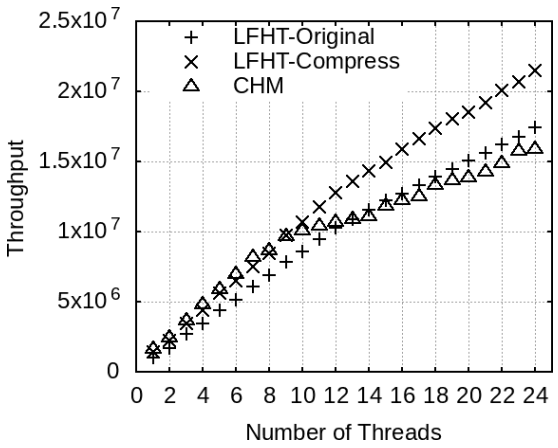
(a) *Search Only* with $N = 10^8$ keys



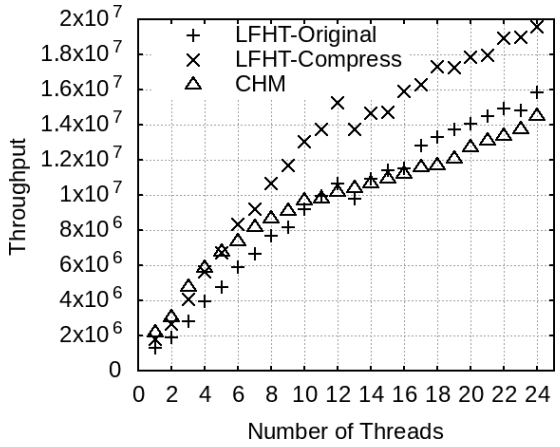
(b) *Search Only* with $N = 10^9$ keys



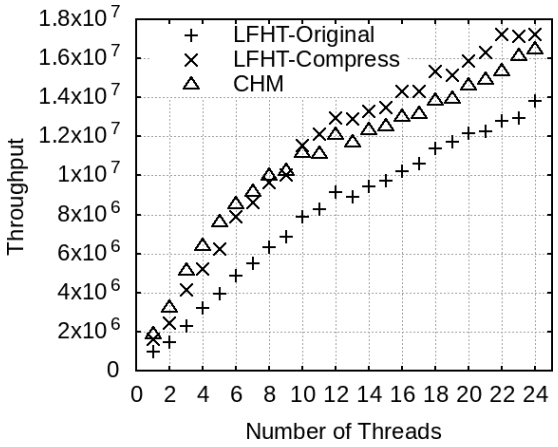
(c) *Insert Only* with $N = 10^8$ keys



(d) *Insert Only* with $N = 10^9$ keys



(e) *Remove Only* with $N = 10^8$ keys



(f) *Remove Only* with $N = 10^9$ keys

Figure 3.19: Throughput for the *Search Only*, *Insert Only* and *Remove Only* scenarios

3.5.1 Solutions

The first solution found to the delegation problem was achieved by relying on the change done in the HHL method to the LFHT data structure, where expansions are given priority over insertions. In this solution threads collaborate to finish an ongoing expansion if they are trying to insert a node in a path that is being expanded. However, as seen in Section 3.2.2, this change alone does not prevent the delegation problem because if the last node in a chain being expanded is invalidated, the threads performing the expansion operation might not see the invalidation and expand the node, effectively reinserting it in the data structure, even if it was already unlinked from the chain by another thread (either the removing thread or another expanding thread). Note that this only happens when the last node on the chain (the one that connects to the new hash node) is invalidated, as during the expansion nodes are always updated to point to the new hash node before being inserted in it, which means that any other invalidation on the chain except for the last node will be visible to an expanding thread before it attempts to reinsert it in the new hash node. Our solution thus relies on the fact that the problem only occurs in this case, when the last node on a chain that is being expanded is invalidated. We prevent the invalidation of such nodes by forcing threads that want to remove such nodes to first collaborate to finish the expansion and only then invalidate the node, when it no longer references a new hash node.

Areias and Rocha [8] developed a method to remove empty hash nodes from the LFHT data structure. Although this hash removal method is incompatible with the HHL method for reclaiming memory, by using the delegation prevention mechanism allowed us to apply the standard hazard pointers method to this new version of the LFHT data structure, which resulted in a LFHT design that is able to remove both leaf and hash nodes and perform SMR on all such nodes [71].

Even though we have a general solution to the delegation problem, it adds further complexity to the LFHT design, specially, if we try to combine it with both the compression mechanism presented in Section 3.4 and the hash node removal mechanism developed by Areias and Rocha [8]. As an alternative, we propose a new version of the LFHT design that simplifies the expansion mechanism, which is the most complex part of the original LFHT design, and at the same time improves performance by making the design more cache friendly. We will call this new version of the LFHT data structure *Simplified Lock-Free Hash Tries* (SLFHT)

In what follows, we will start by presenting the new SLFHT design, followed by the relevant algorithms that show in detail how this new version can be implemented, and then we show some experimental results comparing it to the original design.

3.5.2 New Design

To deal with node collisions on a bucket entry, the original design chains these nodes in a linked list up to a certain threshold, at which point a new insertion triggers an expansion that inserts a new hash node and moves the nodes one by one to the new level. The key idea behind the new

design is to replace these collision chains with a simple array of nodes. This change transforms what would be random memory accesses while traversing the list into sequential accesses, and takes advantage of the fact that in modern CPUs a memory access causes an entire cache line to be loaded (and often more than one cache line due to prefetching optimizations at the CPU level). So, leaf nodes become an array with a header that specifies the number of nodes in the array, followed by all the nodes that collide in that hash node entry sequentially in memory. In what follows, we will call these arrays of nodes *leaf arrays*.

The insertion procedure, instead of adding a node to the chain, now replaces the entire leaf array with a new one containing all the nodes that were present in addition to the new node. Figure 3.20 shows an example of nodes K_1 and K_2 being inserted in a hash node H_i. We start with an empty hash node H_i in Fig. 3.20(a). Then, in Fig. 3.20(b) we insert node K_1 by adding a leaf array with size 1 and node K_1. Finally, as node K_2 collides on bucket B_k, in Fig. 3.20(c) we replace the leaf array with K_1 with a new leaf array with size 2 that contains both K_1 and K_2.

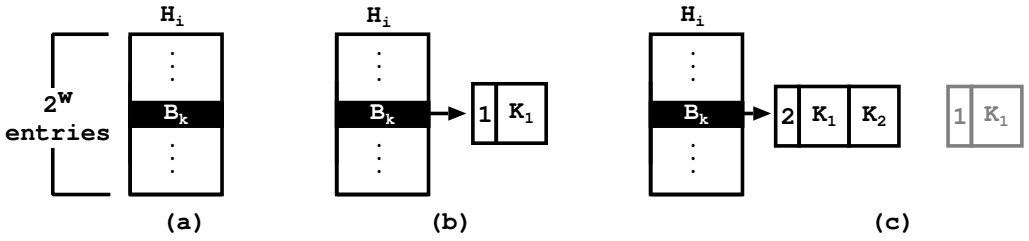


Figure 3.20: Insertion of nodes in a hash level

Similarly, the removal procedure replaces the entire leaf array with a new one that contains all the previous nodes except the one being removed. Figure 3.21 shows an example of the removal of the nodes K_1 and K_2 from H_i. We show the initial state in Fig. 3.21(a). Then, in Fig. 3.21(b) we remove K_2 by replacing the leaf array containing K_1 and K_2 with a new leaf array that only contains K_1. Finally, in Fig. 3.21(c) we remove K_1 by removing the entire leaf array as there are no nodes left to keep in B_k.

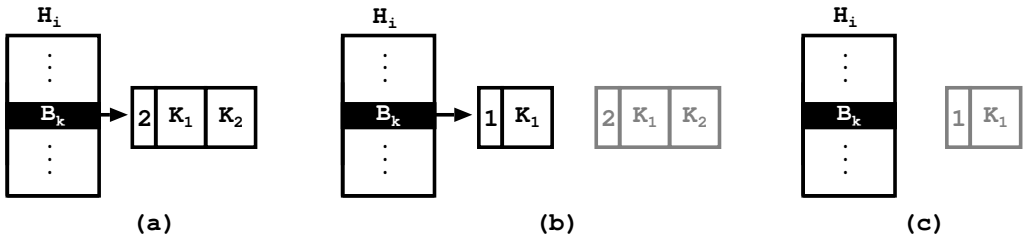


Figure 3.21: Removal of nodes in a hash level

These replacements are performed with a CAS that ensures that the procedure succeeds only if no other thread replaced the leaf array in the meantime, and in the case of failure we simply need to retry the insertion/removal procedure.

As in the case of the original design, when the number of collisions reaches a certain threshold,

we trigger an expansion. The expansion now also becomes much simpler, as we can locally create the new hash node with all the nodes present in the leaf array previously inserted into it, and then replace the leaf array with the new hash node with a single CAS. In the case of a CAS failure, we retry the operation from the beginning. Note that it might not trigger an expansion anymore if the cause for the CAS failure was either the removal of a node from the leaf array, which allows us to insert the node without expanding, or the expansion of the entry by another thread that is also performing an insertion. Figure 3.22 shows an example of an expansion triggered by the insertion of a node K_4 that falls in the bucket B_k (we assume an expansion threshold of 3). We start the expansion in Fig. 3.22(a) by creating a new hash node H_{i+1}. Then, in Fig 3.22(b) we locally insert the nodes present in the leaf array of B_k into H_{i+1}. Next, in Fig 3.22(c) we replace the leaf array in B_k by the hash node H_{i+1} with all the nodes pre-inserted. Finally, in Fig 3.22(d) we insert node K_4 by replacing the leaf array in bucket B_n by a new leaf array containing both K_2 and K_4.

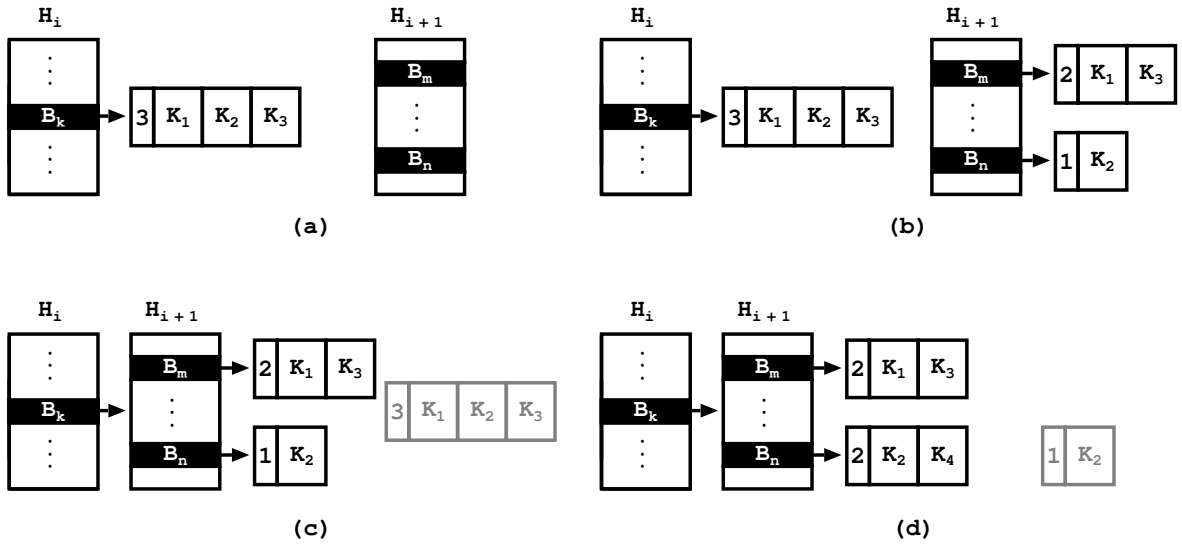


Figure 3.22: Expansion of nodes in a hash level

As we no longer need a reference to the previous hash node or the level to be present in the header of the hash node, we chose to remove the header altogether from the hash node and store the node type in the least significant bit of the pointer to the hash node. So, hash nodes become just an array of bucket entries. Note that the level can always be inferred from context as we can no longer end up in a different hash node by following a chain of leaf nodes, as leaf arrays have no references. This change not only saves memory in the hash node, but also allows for a faster traversal, as we no longer need to read both the header and the entry in every hash node, and only need to read the desired bucket entry.

The main disadvantage of this design is that, even though it ends up consuming less memory (as we no longer need a reference field to the next node for every leaf node, but only a header field per group of nodes in a leaf array), we cycle through a lot more memory. Now even insertions produce memory to be reclaimed, when a leaf array is replaced the old one needs to be reclaimed,

and even removes produce memory allocations, when a leaf array contains more than one node, a new one needs to be allocated to contain the remaining nodes and replace the old one. As we will show in Chapter 4, we argue that with modern memory allocators and SMR methods these extra costs can be mostly mitigated.

3.5.3 Algorithms

In this Section, we present the algorithms supporting the new SLFHT design. We start with Alg. 7 that shows how the data structure is traversed and is then used by Alg. 8, 9 and 10 to implement the relevant operations.

The *Find*() procedure in Alg. 7 takes four arguments: (i) the hash node where it starts the traversal; (ii) the level in which the hash node is at (note that the level needs to be passed as an argument as it is no longer present in the header of the hash node); (iii) the hash of the node or insertion point to be found; and (iv) the key of the node or insertion point to be found. It returns the following values: (i) the corresponding value if a node with the given hash and key was found and *Null* otherwise; (ii) the hash node where the node was found or where it should be inserted; (iii) the corresponding leaf array where the node is or the leaf array where it should be inserted, or *Null* if there is no leaf array in the bucket; (iv) the level of the hash node being returned; and (v) the index in the leaf array where the node was found or *Null* if it was not found. The *Index*() function extracts the relevant bits from the hash based on the level, and the *NodeType*() function extracts the type of the node from the least significant bit of the given reference.

Algorithm 7 *Find*(*node Hn*, *level l*, *hash h*, *key k*)

```
1: C ← Hn[Index(h, l)]
2: if NodeType(C) = HASHNODE then
3:    return Find(C, l + 1, h, k)
4: if C ≠ Null then
5:    for i ← 0 to C.size do
6:       if C.array[i].hash = h ∧ KeyEquals(C.array[i].key, k) then
7:          return ⟨C.array[i].value, Hn, C, l, i⟩
8: return ⟨Null, Hn, C, l, Null⟩
```

In Alg. 8, the *Search*() procedure starts by calculating the hash for the given key and then uses the *Find*() algorithm to find the node. Then, it returns the value content of the node in case of success, and *Null* in case of failure.

Algorithm 8 *Search*(*node Hn*, *key k*)

```
1: h ← Hash(k)
2: ⟨r, Hn, C, l, i⟩ ← Find(Hn, 0, h, k)
3: return r
```

In Alg. 9, the $Insert()$ procedure retries until it is either able to insert the node or find it already present in the data structure. If at any point, the leaf array it is trying to insert on is full, it first performs an expansion and then retries the insertion. Also note that the $CreateExpandHash()$ function creates a new hash node with the nodes present in the given leaf array already inserted in the respective buckets, and the $CreateAddNode()$ function creates a new leaf array with the new node in addition to the contents of the previous leaf array.

Algorithm 9 $Insert(node\ Hn,\ key\ k,\ value\ v)$

```
1:  h ← Hash(k)
2:  l ← 0
3:  loop
4:    ⟨r, Hn, C, l, i⟩ ← Find(Hn, l, h, k)
5:    if r ≠ Null then
6:      return ⟨False, r⟩
7:    if C ≠ Null ∧ C.size = THRESHOLD then
8:      N ← CreateExpandHash(l + 1, C)
9:      if CAS(Hn[Index(h, l)], C, N) then
10:       Hn ← N
11:   else
12:     N ← CreateAddNode(C, h, k, v)
13:     if CAS(Hn[Index(h, l)], C, N) then
14:       return ⟨True, v⟩
```

In Alg. 10, the $Remove()$ procedure retries until it is either able to remove the node or not able to find it. Note that both $CreateAddNode()$ and $CreateRemoveNode()$ consider that an empty leaf array is represented by a $Null$ reference.

Algorithm 10 $Remove(node\ Hn,\ key\ k)$

```
1:  h ← Hash(k)
2:  l ← 0
3:  loop
4:    ⟨r, Hn, C, l, i⟩ ← Find(Hn, l, h, k)
5:    if r = Null then
6:      return ⟨False, Null⟩
7:    N ← CreateRemoveNode(C, i)
8:    if CAS(Hn[Index(h, l)], C, N) then
9:      return ⟨True, r⟩
```

3.5.4 Experimental Results

The hardware used was a machine with 2 AMD Opteron™ Processor 6274 with 16 cores each, 16 KiB of L1 cache per core, 2048 KiB of L2 cache per two cores and 14 MiB of usable shared

L3 cache per CPU. It had a total of 32 GiB of DDR3 memory. The machine was running the Ubuntu 22.04 operating system with Linux kernel version 5.15.0-91. All programs were compiled with GCC 13.2.1 with -O3 and using the jemalloc memory allocator version 5.2.1. The results shown in the following figures were obtained by taking the mean of 5 benchmark runs.

In what follows, we analyze the performance of our new SLFHT design compared to the original LFHT design. In this analysis we perform no memory reclamation as we intend to show the results caused just by the design differences. Memory reclamation for this new design will be discussed in Section 4.3.

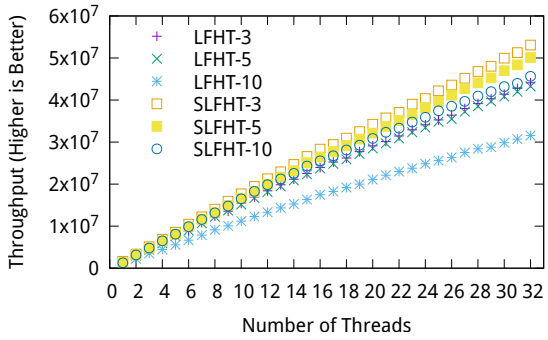
We show results for multiple different scenarios with varying ratios of search, insert and remove operations. On each scenario, we used two sets of hash node sizes, namely 2^4 and 2^8 and for each hash node size we used three values for the maximum number of collisions on a hash node entry before triggering an expansion, namely 3, 5 and 10. In all scenarios we always perform a total of 10^7 operations with random keys. The benchmark has a preparation stage in which all keys that are going to be searched or removed during the benchmark are pre-inserted. To support concurrent randomicity on each thread, we used glibc PRNG, such that, for insertions we just insert random keys by giving each thread a different seed, and for search and remove, we reuse the seeds used for insertion, ensuring that we search or remove each key only once. The results are shown in throughput, which is obtained by dividing the number of operations performed by the average of the time taken to perform them.

Figures 3.23, 3.24 and 3.25 show the results for one kind of operation each, and Figs. 3.26, 3.27 and 3.28 show varying ratios of searches with the rest of the operations being inserts and removes in a one to one ratio, so the number of nodes in the data structure remain somewhat constant throughout the benchmark. In all graphs the legend specifies the LFHT design using *LFHT* for the original design and *SLFHT* for the new design, that, is followed by the threshold of node collisions required to cause an expansion.

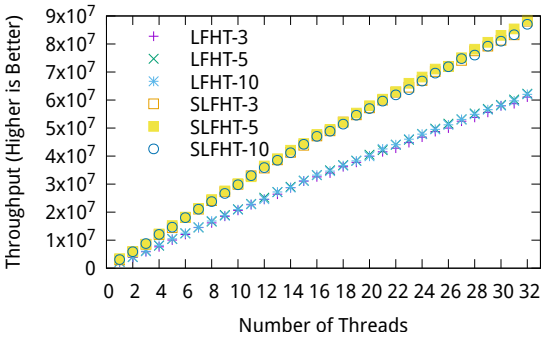
As we can see, for hash nodes with 2^4 entries (figs. 3.23(a), 3.24(a), 3.25(a), 3.26(a), 3.27(a) and 3.28(a)) entries the ideal threshold for expansion is 3 nodes in both designs. For hash nodes with 2^8 entries (figs. 3.23(b), 3.24(b), 3.25(b), 3.26(b), 3.27(b) and 3.28(b)) the ideal threshold for expansion is 5 nodes. However, and unlikely the 2^4 entries configuration, there is not as much of a penalty for using higher expansion thresholds, which might be a worthy trade-off in order to save on memory usage.

Across the board we can see an increase in throughput for the new SLFHT design, when comparing similar expansion thresholds. The improvement is specially noticeable for larger hash nodes and in benchmarks with more search operations, the best case scenario being shown in Fig. 3.23(b) where we can see an improvement of about 42% with 32 threads and an expansion threshold of 5. Figures 3.24(a), 3.26(a) and 3.27(a) show the new design losing some scalability with an expansion threshold of 10, this happens mainly due to the extra pressure on the memory allocator, which is exacerbated by the fact that we are not reclaiming and thus, not reusing memory. Even so, the new design still manages to outperform the original with the number of

threads benchmarked.

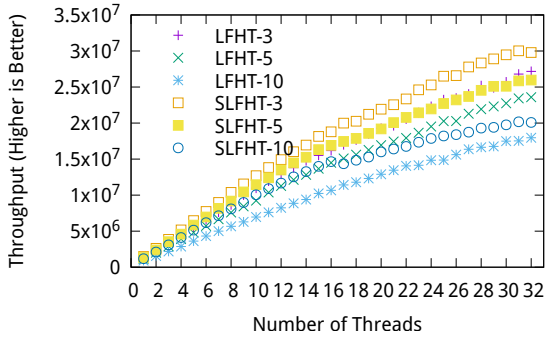


(a) Hash nodes with 2^4 entries

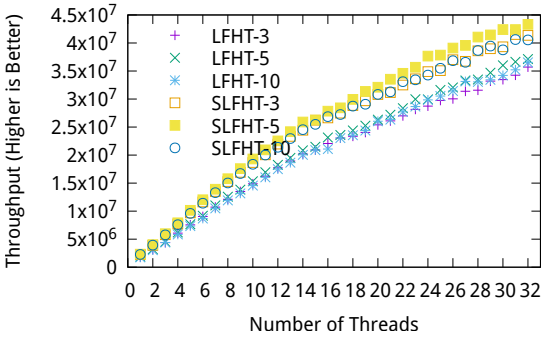


(b) Hash nodes with 2^8 entries

Figure 3.23: Throughput for the *Search Only* scenario

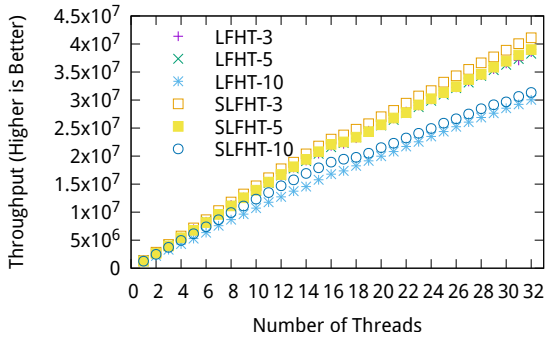


(a) Hash nodes with 2^4 entries

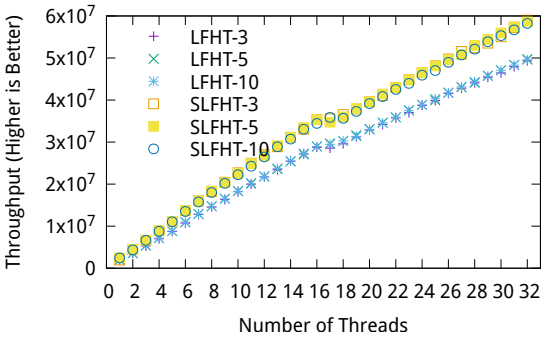


(b) Hash nodes with 2^8 entries

Figure 3.24: Throughput for the *Insert Only* scenario

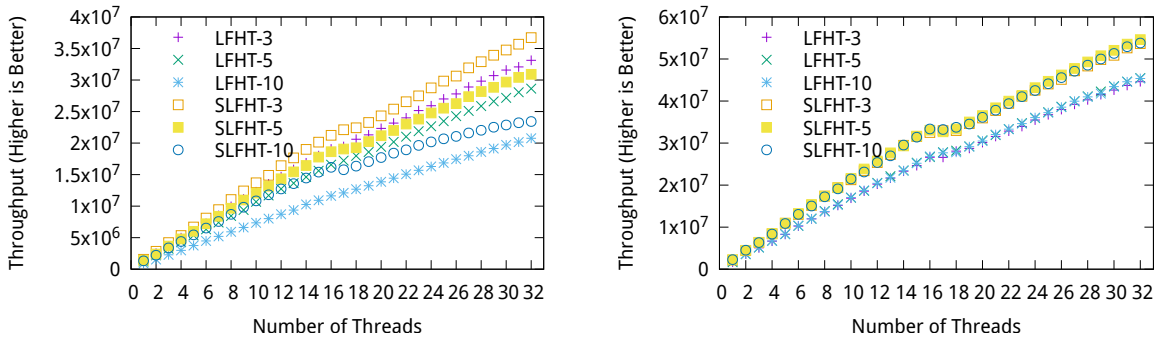


(a) Hash nodes with 2^4 entries



(b) Hash nodes with 2^8 entries

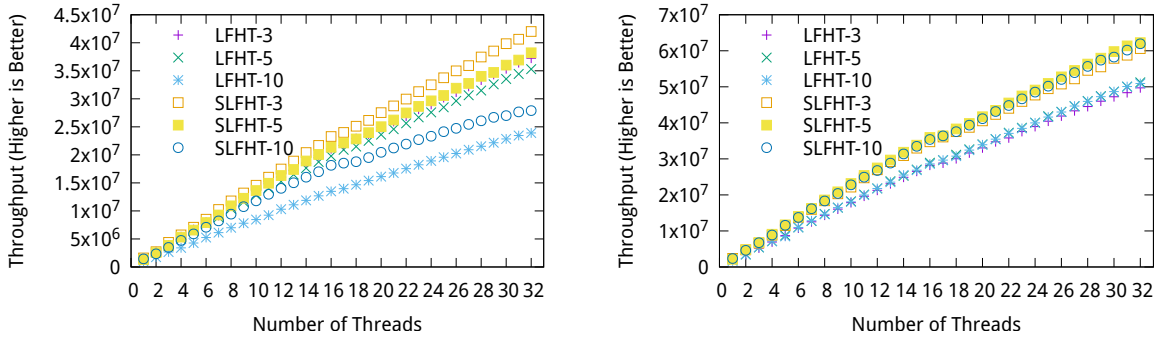
Figure 3.25: Throughput for the *Remove Only* scenario



(a) Hash nodes with 2^4 entries

(b) Hash nodes with 2^8 entries

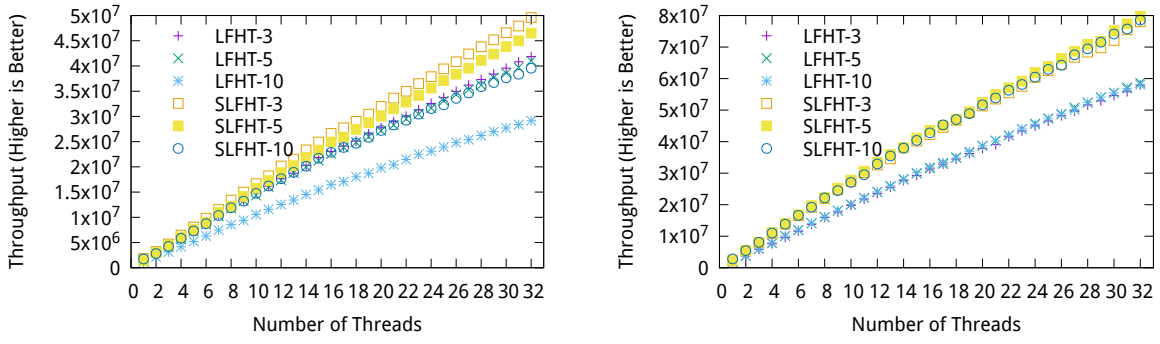
Figure 3.26: Throughput for the 50% *Inserts* and 50% *Removes* scenario



(a) Hash nodes with 2^4 entries

(b) Hash nodes with 2^8 entries

Figure 3.27: Throughput for the 50% *Searches*, 25% *Inserts* and 25% *Removes* scenario



(a) Hash nodes with 2^4 entries

(b) Hash nodes with 2^8 entries

Figure 3.28: Throughput for the 90% *Searches*, 5% *Inserts* and 5% *Removes* scenario

Chapter 4

Memory Management and SMR

As seen in Section 2.5, not all SMR methods can rely on a standard memory allocator for memory management. This happens for one of two reasons: (i) the method requires a *type-preserving allocator*, which means that at least some fields in the allocated memory need to remain valid from the point the memory is freed until it is reused by an allocation, and such memory can only be reused for allocations of the same type. An example is the LFRC based methods [84, 86] that need the reference counting field in the nodes to remain valid throughout the lifetime of the data structure, as they might be incremented and decremented even after the node has been freed. And, (ii) the method requires the memory that has been freed to remain accessible, but has no constraints on the content, as all reads to such memory will be ignored. An example being the optimistic access based methods [17–19] that perform optimistic reads to reclaimed memory, but ignore the contents if they later detect that such memory could have already been reclaimed.

To work around the issue of having freed memory accessible, optimistic access based methods implement a recycling mechanism to manage the memory being used. However, this prevents the memory used in this manner from being reused in other parts of the same process and from being released to the operating system. In this Chapter, we propose a solution to this problem without having to make the whole application aware of the memory reclamation method. Our proposal is to extend *LRMalloc* [48], a lock-free general purpose memory allocator, in such a way that we can guarantee memory allocations to be readable even after we free such allocations. No guarantees are given about the content of freed memory, or how it is reused by the rest of the application. This is a good match for OA because it already ensures that the contents of reads on possibly reclaimed memory are to be ignored, and that memory to be written is protected from reclamation by the use of hazard pointers.

We start by solving the problem at the memory allocator level, by adapting LRMalloc such that it does not release memory used by the OA method back to the operating system. This allows us to simplify the implementation of the OA memory reclamation method as we no longer need a recycling mechanism in order to manage the distribution of memory between threads. This task is now covered by the memory allocator as it was designed for this task in a general sense. We also gain the ability to reuse memory reclaimed by the OA method across the whole

process. As we will see, all this is possible with minimal changes to the LRMalloc memory allocator.

Then, to complete our solution, and have the ability to release the memory used by the OA method to the operating system, we exploit how current operating systems/hardware use virtual memory. As we need the virtual addresses (pages) to remain accessible after they have been used by the OA method, but we do not care about the contents on the physical memory (frames) they are mapped to, we map all these multiple pages to the same frame. This allows us to free all the frames our pages were previously mapped to while keeping the pages still valid for access.

Modern operating systems apply similar strategies, e.g., when a memory request is made to the operating system, no frame is immediately reserved, only the pages are made valid by being all pointed to a single *copy on write* zero filled frame. Only when a memory write is attempted on these pages, is that the operating system copies the zero filled frame to a new free frame and maps the page to it. This all happens transparently to the application, which never notices that the memory given to it at the start was not actually backed by physical memory. One of the strategies we propose to implement the remapping of pages exploits this operating system behavior, while the other strategy will do the remapping in a more explicit fashion using the shared memory mechanisms of current operating systems.

The remainder of this Chapter is organized as follows. First, we introduce virtual memory, memory allocation in general with a focus on the LRMalloc memory allocator, and go into more detail over the optimistic access method focusing on how it recycles memory. Then, we present in detail the main ideas supporting our approach and discuss its current limitations and its broader applicability to other use cases. Finally, we apply this new approach to the new design of the LFHT data structure and compare it to the HHL method described in Section 3.2.2.

4.1 Memory Management Background

This section briefly introduces relevant background about virtual memory and memory allocation systems and describes in more detail the LRMalloc memory allocator and the recycling mechanism used by the Optimistic Access SMR method.

4.1.1 Virtual Memory

Virtual memory is a memory management system that works as an abstraction layer that allows for a multitude of optimizations in modern operating systems. The main idea is to have a translation layer between the memory addresses viewed by a user process and the actual physical addresses in main memory. The translation is done in hardware by the *memory management unit* (MMU) and relies on a cache named *Translation Lookaside Buffer* (TLB). This introduces an overhead, as with virtual memory, when trying to access a memory location, one first needs to consult where the virtual address resides in physical memory. This requires extra memory

accesses in order to obtain the physical memory location, however by the use of an efficient TLB this disadvantage is mostly mitigated. Modern systems define the granularity of a page/frame to be a power of 2, usually between 4 KiB and 1 GiB total size.

The main benefits provided by virtual memory are the ability for processes to oversubscribe memory allowing them to use more memory than what is physically available, the ability of multiple processes having the same address space, the ability to move unused pages from memory to persistent storage when under memory pressure, and the ability to block a process from accessing or modifying any memory that does not belong to it. Virtual memory also allows memory to be shared between processes, the most common case being shared libraries, so multiple processes can use the same copy of a library in physical memory but each have it in a different memory address. Another important use case is efficient inter-process communication, made possible by having two or more processes mapping a single region of physical memory into their own address spaces.

Further optimizations include the ability to only load frames when they are needed, meaning that when a process is loaded into memory, it does not need to be entirely loaded, only the necessary frames are loaded as the corresponding pages are accessed. For example, an error routine that is never called would never actually be loaded into physical memory. When a process requests memory from the operating system, a similar optimization can be done, every page the process requests can be initially mapped to a single zero filled frame and only mapped to free memory frames when they are actually written to. As we will see later, this is one of the features that we will take advantage of for our proposal.

4.1.2 Memory Allocation

Memory allocators [91] serve as an interface between processes and the operating system, satisfying memory requests of any size in such a way that processes waste as little additional memory and time as possible. To do so, a memory allocator starts by acquiring pages from the operating system that are then subsequently divided to satisfy smaller allocation requests, and later combined in order to give complete pages back to the operating system. Classic memory allocators [91] tended to use strategies like *best-fit*, in which they find the smallest block of contiguous memory that can satisfy the request and, if such a block is still too big, it is split to the right size, so they can keep what remains to a future allocation. Another strategy is *first-fit*, in which instead of finding the smaller continuous block that satisfies a request, they simply use the first block found. This strategy has a speed advantage, but can increase memory waste.

A more modern strategy is to use *size classes* [91], where any request is met by rounding up to the nearest size class. Blocks of a size class are created by splitting a bigger block into many blocks of the same size. The size classes need to be carefully selected, therefore avoiding too many different classes and possibly allocations of large blocks that result in a limited amount of allocations from it, or too few classes and possibly wasting memory by having to provide a much larger allocation than needed due to the nonexistence of a large enough smaller size. Size classes

are very time efficient and tend to improve memory locality, therefore also improve the global performance of applications beyond memory allocation.

With the advent of multicore processors, in order to further improve performance and scalability, different proposals were adopted to minimize the amount of synchronization between threads. These gave origin to mechanisms such as *private heaps* [11], in which each thread has a private allocator implementing specific strategies to deal with frees that occur in threads different from the one where the memory was allocated. These strategies can be used to keep the free memory in the thread in which it was freed until it is allocated again; to immediately give back the free memory to the thread it was allocated on; or to give back only after a threshold is met. An alternative mechanism is to use a per thread cache on top of a shared heap [47].

Keeping freed memory that was allocated in another thread in the private heap until it is allocated again can lead to a phenomenon known as *blowup* [11]. That is, in a scenario of producer consumer workload in which some threads only free memory, if they never return such memory to the private heap it originated from, then that memory can accumulate indefinitely without being able to be reused or returned to the operating system. On the other hand, immediately returning freed memory to its originating heap can lead to significant performance degradation, as in the previous example of a producer consumer workload, would result in almost as much synchronization as not having a private heap per thread in the first place. The ideal compromise is either the usage of a threshold in private heaps, or a shared heap with thread caches, as either can prevent blowup by limiting the amount of freed memory they can accumulate, and at the same time amortize the cost of synchronization over many allocations.

4.1.3 LRMalloc

LRMalloc [48] is a modern lock-free memory allocator that uses size classes and thread caches as described above. It offers the best performance in lock-free memory allocation, while being competitive with state-of-the-art blocking allocators, such as jemalloc and TCMalloc. Due to relying on size classes, it achieves no external fragmentation and 25% internal fragmentation, in the worst case. It also does not suffer from *blowup* due to the use of thread caches.

LRMalloc has three main components: (i) the *thread caches*, one per thread; (ii) the *heap*; and (iii) the *pagemap*. Figure 4.1 shows the relationship between these three components, the user's application and the operating system,

The thread caches use a stack for every size class, so that a memory request becomes simply a pop on the corresponding size class stack, and a memory free becomes a stack push. When a memory request is made and the corresponding stack is empty, then the stack is filled from the heap, and when a memory free happens and the stack is full, it is flushed back to the heap. The size of the stack is limited in order to prevent *blowup* [11]. The caches are local to a thread, so they only synchronize with other threads when a fill or flush from/to the heap occurs.

The heap is responsible for managing *superblocks*, which are large blocks of memory obtained

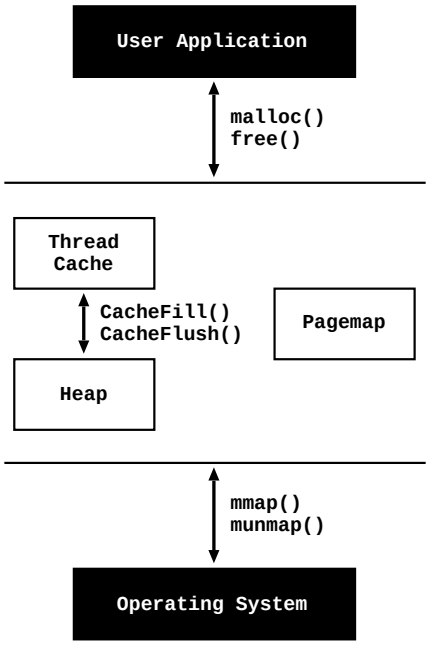


Figure 4.1: LRMalloc's overview

from the operating system that are then divided into blocks of a size class to be given to the thread caches. Superblocks are managed through *descriptors*, an object that contains the superblock metadata and that is never reclaimed. When a superblock is released to the operating system, the associated descriptor is added to a recycling pool in order to be reused for a future superblock. The descriptor contains information, such as, where the superblock begins, its associated size class, the number of blocks it possesses, the index of the first free block and the number of free blocks.

Superblocks can be in one of three states: (i) *full*, if all its blocks are in use; (ii) *empty*, if all its blocks are available for allocation; or (iii) *partial*, if it has available and allocated blocks. The initial state of a superblock is always full, as all its blocks are immediately used to fill a cache. Then it becomes partial as some blocks are returned to it by cache flushes, at which point it can either become full again, if a thread uses it to fill its cache, or it can become empty, if all blocks are returned to it. When a superblock becomes empty, its memory is released back to the operating system. When threads try to fill their caches they give priority to partial superblocks and, if none is available, a new superblock is created by requesting memory from the operating system.

The pagemap is a simple lock-free data structure that stores metadata for each page in use. Taking into account that superblocks are always aligned with pages and have a size that is a multiple of the page size, blocks in the same page always belong to the same superblock. So this metadata includes the superblock that a page belongs to and its associated descriptor. The main usage of the pagemap is to allow finding the corresponding superblock for a block that is flushed from the cache, or to allow finding the appropriate cache (with the correct size class) when memory is received from the application through a call to the *free()* procedure.

4.1.4 Optimistic Access

In general, an SMR method for a lock-free data structure is a mechanism that detects when a node removed from the data structure can no longer be referenced by any running thread, and thus uses such information to free the corresponding memory to the memory allocator/operating system. Usually, such methods require some sort of validation to avoid accessing memory that has been already reclaimed.

An alternative approach is the one followed by the OA method [19], which, as the name implies, allows memory accesses before making sure the memory has not been reclaimed, and only then checks the validity of the access by reading a specific *warning-bit*. If the access corresponds to reclaimed memory, the result is ignored and the procedure is restarted from a memory location known to be valid. However, modifying operations cannot be performed optimistically as an optimistic CAS could incorrectly succeed due to an ABA problem [21]. For that, OA uses a hazard pointer strategy, so before performing any atomic CAS update operation, it first protects all memory addresses involved by assigning hazard pointers to them and then performs a single additional validity check by reading the *warning-bit*, therefore ensuring that the memory was valid when it was protected by the hazard pointers. These hazard pointers are then used to prevent the recycling of the memory they are assigned to.

The OA memory recycling mechanism is composed by three pools: (i) the *ready pool* that contains all the nodes ready to be allocated, (ii) the *retire pool* to where nodes are added when they are retired from the data structure, and (iii) the *processing pool* that holds the nodes that are in the process of being recycled. The recycling mechanism works in phases, and a new phase is triggered when the ready pool is exhausted. At the start of a new phase, the nodes present in the retire pool before the phase starts are moved to the processing pool. Next, all threads are informed of the current recycling by their *warning-bit* being set. Finally, the nodes in the processing pool that are protected by hazard pointers are moved back to the retire pool, the ones not protected are moved to the ready pool. Threads that try to retire a node during the process of moving nodes from the retire pool to the processing pool need to help finish the move before retiring the node. Threads that try to start a new recycling phase while one is already in progress need to help finish the current phase before starting a new one.

While the recycling mechanism is complex and time-consuming, it is rarely executed, which mitigates its cost. For the more frequent operations, such as the traversal of the data structure, this method only needs to perform an extra read per node traversed instead of a write, as it is the case for the hazard pointers memory reclamation method, and it also requires a much less expensive memory barrier, which in *total store ordering* (TSO) architectures like x86-64, translates to a simple compiler barrier and no additional hardware instructions are emitted. Also note that modifying operations can set multiple hazard pointers and only after perform a single validity check that requires an expensive memory barrier, unlike the hazard pointers method that needs to validate after every hazard pointer, and as such one expensive barrier per hazard pointer. This effectively reduces the number of expensive memory barriers in read

only operations, from one per node traversed to zero, and in modifying operations, from one per node traversed to one per operation. These characteristics make the OA memory reclamation method extremely efficient and performant compared to the state-of-the-art, while also having low memory bounds and not requiring any specific support from the operating system.

A consequence of allowing optimistic accesses to possibly reclaimed nodes is that nodes need to remain accessible after being reclaimed. However, there is no need for the contents of the node to be maintained, as the result of the access will be ignored in the case it was invalid. To ensure the nodes are accessible after being reclaimed, the recycling mechanism is used, which allows nodes to be reused, but never released to the memory allocator or to the operating system.

4.2 Memory Allocation and SMR

In this section, we start by introducing how we made LRMalloc compatible with the OA memory model and how we can use it to simplify the OA method. Next, we present how we can exploit virtual memory in order to allow memory to be released to the operating system.

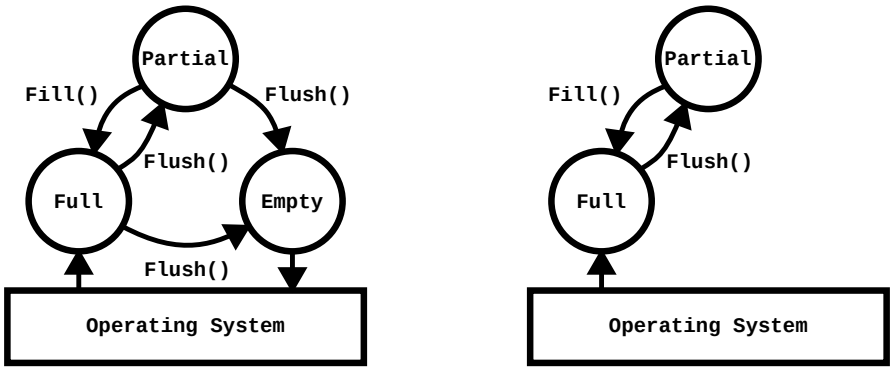
4.2.1 Memory Recycling at the Allocator Level

Remember that the OA method allows memory to be read even after being reclaimed because reads are validated. Writes to reclaimed memory are prevented because memory is protected by hazard pointers. In a program using a lock-free data structure in combination with the OA method, the memory reclaimed can be reused by the same data structure, but it cannot be reused by other parts of the program, at least without extensive modifications both to the memory reclamation method and to the rest of the program. Our solution to avoid this restriction is to adapt the memory recycling mechanism at the allocator level. To achieve this we extended LRMalloc with a new persistent allocation function that we named *palloc()*.

To implement *palloc()* we follow the same process as in a regular allocation, but the *superblock* that contains the memory block being allocated is marked as *persistent*. This mark is then used to guarantee that persistent superblocks never reach the *empty* state, even if all its blocks are available. This change ensures that memory allocated with *palloc()* is never released to the operating system, but can still be reused by future allocations anywhere on the same process. Figure 4.2 shows the state diagram for superblocks before and after being marked as persistent.

By having an allocator that satisfies these properties, we can now extensively simplify the memory reclamation method. As we no longer need the memory recycling mechanism employed originally in the OA method, we can use a much simpler mechanism, similar to the one used by the hazard pointers memory reclamation method, as shown in Alg. 11.

The idea is as follows. When a node is retired, we add it to the reclaiming thread's *limbo list*, and when the list's size reaches a certain threshold, we perform the reclamation procedure.



(a) Non-persistent (b) Persistent

Figure 4.2: State diagram for superblocks

Algorithm 11 *Retire*(*Node N*)

```
1: LimboList.add(N)
2: if LimboList.full() then
3:    for T in Threads do
4:       T.warning_bit.set()
5:    Reclaim(LimboList)
```

During such procedure, we only need to set all the other threads' *warning-bit* and then free all nodes that are not protected by a hazard pointer using the *Reclaim()* procedure, as shown in Alg. 13.

This mechanism however is not ideal for data structures with long chains, such as linked lists, since as we trigger more warnings, more restarts are needed. These restarts are inexpensive on data structures with short chains, such as hash tables, but not so much in linked lists, not only because the amount of work lost by a restart is high, but also because the beginning of the chain is most likely out of the L1 cache by the time of the restart.

To mitigate this issue, we implemented another warning mechanism that is based on the one used in the *Version Based Reclamation* (VBR) method [81]. In this mechanism, instead of having a warning bit per thread, in which a thread has to set all other threads warning bit in order to send a warning, we have a monotonic global variable that we increment when we want to send a warning to all threads. Threads then check for the warning by comparing the last value they saw in the global variable with the current value, i.e., when a thread detects an increment in the global variable, it knows a warning has been sent to it and every other thread. With this mechanism we can allow threads to piggyback of each other warnings[1], as we can forego sending a warning if one has happened in the period between the time the node was retired and the time we try to reclaim it. Note that we not only take advantage of other threads warnings when we see the increment in the global variable, but also when we try to increment it with a CAS and it fails, which means that a warning was successfully fired by another thread and we can take

[1]Note that this is not possible on Alg. 11 as the warnings are not atomic with one *warning-bit* per thread.

advantage of it. The *Retire()* procedure based on this alternative mechanism is shown in Alg. 12.

Algorithm 12 $Retire(Node\ N)$

```
1: if LimboList.full() then
2:   if LastRetireTime = LocalClock then
3:     CAS(GlobalClock, LocalClock, LocalClock + 1)
4:     LocalClock ⇐ GlobalClock
5: if LastRetireTime < LocalClock and LimboList.size() > X then
6:   Reclaim(LimboList)
7: LastRetireTime ⇐ LocalClock
8: LimboList.add(N)
```

GlobalClock represents the monotonic global variable that can be updated by any thread sending a warning. *LocalClock* is a thread local variable used to store the last seen value of the global variable and thus can be updated in other functions by the same thread, e.g., when a search is restarted due to a warning. *LastRetireTime* is a local variable accessed only by this procedure and used to take advantage of the other threads warnings. We also reuse the *Reclaim()* procedure introduced in Alg. 11 and described next in Alg. 13.

Algorithm 13 $Reclaim(List\ LimboList)$

```
1: MemoryBarrier()
2: for T in Threads do
3:   HPSet.add(T.hazard_pointers)
4: for M in LimboList do
5:   if not HPSet.contains(M) then
6:     LimboList.remove(M)
7:     Free(M)
8: HPSet.reset()
```

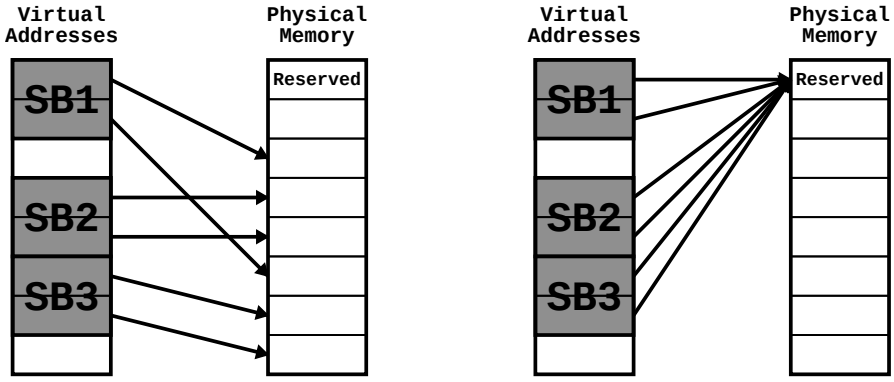
Algorithm 13 shows how nodes are freed, which is identical to how nodes are freed in the hazard pointers method. We start by issuing a memory barrier and by reading all the hazard pointers, and then we free all the nodes that are not referenced by any hazard pointer.

As mentioned earlier, with this method we end up with memory that we can never release to the operating system throughout the lifetime of the process. In the case that a large amount of memory is allocated with *palloc()*, that memory will continue in the process even if the amount of memory it requires for the remainder of its lifetime is much lower. The main advantage of this mechanism is that it requires no additional features from the operating system or hardware compared to any other lock-free memory allocator.

4.2.2 Using Virtual Memory

Now that we have made the memory allocator compatible with the OA model, we next focus on the interaction with the operating system. Remember that the memory allocator cannot release superblocks marked as persistent to the operating system because they need to remain accessible.

If we take a closer look at this problem, taking into account the virtual memory system, we can observe that what actually needs to remain accessible is the address range of the superblocks marked as persistent and not the backing physical memory, as there is no requirement regarding the contents of the reclaimed memory. Thus, the problem can be solved if we can release the physical memory associated with such superblocks but maintain the addresses range accessible[2]. To do so, we can remap the address range of a persistent superblock becoming empty into a default pre-reserved frame. Thus, independently of how many empty superblocks we have, they will just consume a single frame of physical memory. This single frame could even be a frame already in use by the process, as long as we can ensure it will remain accessible throughout the lifetime of the process. Figure 4.3 illustrates this remapping process. In Fig. 4.3(a) we show multiple persistent superblocks using 2 pages each, with each page mapped to a different frame, and in Fig. 4.3(b) we show how the superblocks can be remapped in order to release all their frames while keeping the access to them valid.



(a) Before (b) After

Figure 4.3: Memory mappings before and after the remapping process

However, we need to be careful as the virtual address space is an abundant but limited resource. So, some mechanism to recycle the virtual addresses of the remapped superblocks still needs to be used. But this is almost already done by LRMalloc when it needs to recycle the *descriptors* that contain the metadata of a superblock. Remember that when a non-persistent superblock becomes empty, the superblock is unmapped and the descriptor is added to the recycling pool. Later, when a new superblock is requested, first, a descriptor is obtained from the recycling pool, then a superblock is mapped from the OS, and finally the metadata in the descriptor is rewritten with the metadata of the new superblock. So, if we use instead the address

[2]Note that now we are considering again that all superblocks can become empty, i.e., ready to be released to the operating system.

range stored in the descriptor obtained from the recycling pool to map the new superblock, we are effectively recycling the virtual address space by piggybacking on the descriptor. In the actual implementation, we added another recycling pool with this mechanism, which we give priority to obtain blocks from, and keep the original for descriptors originated from non-persistent superblocks. The reason for the second pool will become clearer in Section 4.2.3.

For the actual remapping process, we propose two methods. The first method is to advise the operating system that the memory will not be needed. On Linux, this is accomplished by the use of the *madvise()* system call with the *MADV_DONTNEED* flag, which reverts the memory mapping to a state similar to when the superblock was first allocated, i.e., all pages are mapped to a single copy on write zero filled frame. This frees all physical memory previously associated with the map until it is written again. Note that reads to these ranges of memory do not cause a page fault, but only an actual read from the zero filled frame. With this method, when we get a descriptor from the recycling pool, we do not need to do any extra work for remapping as the original address range is already valid and ready to use.

This first method has the advantage of being simpler and more efficient, but has two main disadvantages. One disadvantage is that even though this system call and flag are defined in the POSIX standard, the standard itself does not impose the behavior observed on Linux, which makes this method not portable. Another disadvantage is that some OA derived methods, like VBR [81], use DWCAS on reclaimed memory, even though the DWCAS is certain to fail[3] as otherwise it would lead to corruption, the operating system is unable to ascertain that and faults a frame in through the copy on write mechanism. This does not cause a correctness issue but could lead to some memory leaking, as some pages would be reserved for unallocated superblocks.

The second method is to use the shared memory mechanism. We start by defining a shared memory region and then, when we want to deallocate a superblock, we map its address range to the shared memory region. We can choose a size for the shared memory region that varies from the size of a page to the size of a superblock, which can lead to different performance trade-offs as we need one system call to do the remap if we choose the size of a superblock, two system calls if we choose half the size of a superblock, and so on. Note that the physical memory associated with the shared memory region could be used to store something useful in the meantime. For example, it could be used to store the *descriptors*. Later, to reuse the virtual range of the superblock we need to remap it again to new memory. Note that this remap only requires one system call, independently of the size of the shared memory region. On Linux, this method is accomplished with the use of the *mmap()* system call with the flags *MAP_FIXED* and *MAP_SHARED* to release the physical memory, and *MAP_FIXED*, *MAP_PRIVATE* and *MAP_ANON* to reuse the superblock.

Although this method might look a bit abusive, it is supported by the POSIX standard. However, this support is not explicit and, on Linux, the memory statistics report wrong results, as it counts all the ranges mapped to the shared mapping into the *resident set size* (RSS) of the

[3]It uses tagged pointers as an ABA prevention mechanism.

process, even though it only uses one shared mapping of physical memory. This method can also be used in other operating systems outside the POSIX world, and does not lead to memory leakage when CAS instructions are used on reclaimed memory. It requires extra system calls, but we were not able to measure any performance degradation caused by them.

4.2.3 Limitations

The LRMalloc memory allocator uses a size class allocation strategy, which means that allocations up to a reasonable size (16 KiB) are handled through this mechanism. For all size class allocations, LRMalloc uses superblocks of the same size (16 MiB), which simplifies our remapping logic as we can reuse retired superblock addresses to different size classes. Meaning that memory obtained through *palloc()* can be reused in any kind of allocation of any size class after being freed. This is ideal in most scenarios, as most allocations fall into the size class range. However, for allocations larger than the biggest size class, it requires a different mechanism. For such allocations, LRMalloc relies directly on the operating system, as other lock-free memory allocators do [33, 49, 59, 79]. Relying on the operating system for large allocations does not meaningfully impact performance as this kind of allocations are uncommon. Large allocations work similarly to size class allocations, but the thread caches are skipped and a superblock with the exact size needed is mapped to satisfy the allocation.

This way of dealing with large allocations is not ideal, as it requires a different mechanism in order to recycle the range of virtual addresses of such allocations. In this regard, we have chosen to restrict the persistent memory allocation to sizes that are compatible with the size classes. This is not a problem in most situations as lock-free data structures tend to either use small allocations for their internal structure, or the large allocations last the lifetime of the data structure and as such need no reclamation, one example being Michael's lock-free hash tables [57]. The exceptions are lock-free hash maps that use large arrays that are resizable, as during the resizing process they need to allocate a new array and reclaim the old one, and the LFHT data structure with the compression mechanism described in Section 3.4, as hash nodes can become large enough to not fit in the size class allocations and then be replaced by even larger ones. Data structures with these mechanisms are rather uncommon as the resizing processes tend to be complex and synchronization heavy, which leads to performance loss. As such, we leave the resolution of this limitation to future work.

This limitation is also the reason why we need another recycling pool for descriptors when a superblock becomes *empty*. If the superblock is not marked as persistent[4], the superblock is unmapped and the descriptor is added to the pool with the original behavior. If the superblock is marked as persistent, we remap the superblock as shown in the previous section and add the descriptor to the new pool. When we need a new descriptor we try to obtain one using the following priority: (i) the new pool that already has the virtual range of the superblock associated with it and as such is only compatible with superblocks intended for size class allocations; (ii)

[4]Note that only superblocks used for size class allocations can become persistent.

the original pool that has generic descriptors; and finally, (iii) we allocate a new descriptor. We only go down the priority list if either the pool is incompatible or is exhausted.

4.2.4 Applicability

In this section, we start by discussing the applicability of our ideas to other memory allocators, and then we discuss other possible use cases for the *palloc()* functionality in systems outside the OA memory reclamation method.

Other Allocators

Most modern memory allocators, if not all, divide allocations in two major classes: huge allocations, in which they rely directly on the operating system to satisfy the allocations; and regular allocations, in which they request blocks of memory from the operating system that they then divide in order to satisfy the allocations.

For regular allocations, as the blocks of memory requested from the operating system tend to be all of a unique fixed size, the same remapping strategies could be applied to such blocks in order to make memory persistent. However, a mechanism to mark such blocks as persistent is still required when allocating memory through *palloc()*. This can trivially be implemented by having an additional marking bit in the data structure that manages such blocks. Another issue is the necessity to recycle the virtual address space. In the worst case, this can be solved by having an additional data structure to store the virtual addresses of blocks *freed* through remapping, or by using the data structure that already manages the blocks. In order to be able to do persistent allocations of any size, the data structure that manages virtual addresses also needs to be able to coalesce and split the virtual memory ranges.

Some memory allocators already rely (or can be configured to rely) on *MADV_DONTNEED*, *MADV_FREE* or equivalent modes of releasing memory to the operating system. For such allocators, it would be easy to implement *palloc()* based on advising the operating system as they already have all the required structures.

Other Use Cases

Our allocator extension was developed with the OA memory model in mind, but it can also be applied to other use cases.

One such case is the VBR method. As mentioned before, VBR is another memory reclamation method based on optimistic accesses. Instead of relying on hazard pointers, VBR extends the optimistic access to write operations and, to do so without suffering from ABA problems, it replaces all atomic fields in the data structure with a tuple of the field and a monotonic tag and uses DWCAS instead of CAS to update both simultaneously. In order to replace VBR's

recycling mechanism and be able to return memory to the memory allocator/operating system, we could use our allocator extension with a simplified method similar to the ones we presented for the OA method although, as discussed previously in section 4.2.2, the *madvise()* approach can lead to memory leaks, while the other approaches would be fully compatible.

Another example is the case of Software Transactional Memory (STM) systems. As STM systems need to validate every transaction before committing, it is possible to achieve safe memory management just by relying on the *palloc()* and *free()* procedures without the full OA method. However, for this to be possible, the STM system has to satisfy the following properties: (i) it has to rely on lazy version management (deferred update) as memory cannot be written to before the transaction being validated/committed; and (ii) the memory from the application and STM system needs to be segregated (not obtained in the same allocation request), as the lifetime and update semantics of the STM system memory can be different from the application memory.

Word based STM and Object based STM by Fraser and Harris [30], and the per stripe commit-time variants of Transactional Locking STM by Dice and Shavit [25] and Transactional Locking 2 by Dice et al. [26][5] are examples of STM systems that could rely on *palloc()* to achieve safe application memory management. We believe that any STM system that satisfies these properties is a good candidate to use our *palloc()* implementation in order to achieve safe memory reclamation.

4.2.5 Experimental Results

In order to evaluate the impact of our changes to the OA method, we compare the results of our two implementations of the OA method, the one with warning-bits and the one with the monotonic global variable, against the original OA method, and against no reclamation, in which memory is never reclaimed, reused or freed. From this point onwards, we will refer to the original OA method as just *OA*, our simplified OA method with the warning-bit per thread as *OA-BIT*, the alternative with the monotonic global variable as *OA-VER*, and the no reclamation alternative as *NR*.

Methodology

The hardware used was a machine with 2 AMD Opteron™ Processor 6274 with 16 cores each, 16 KiB of L1 cache per core, 2 MiB of L2 cache per pair of cores and 12 MiB of usable shared L3 cache per CPU. It has a total of 32 GiB of DDR3 memory. The machine used was running the Ubuntu 22.04.1 LTS GNU/Linux operating system with the Linux kernel 5.15.0-60-generic.

We benchmarked the four methods with the commonly used Michael's lock-free hash tables [57] and Harris-Michael's lock-free linked lists [58]. For all benchmarks, we use LRMalloc as the

[5]Note that on both Transactional Locking STM systems memory can only be freed after all locks to such memory are released as our system dos not allow writes to freed memory

memory allocator, and although for our simplified versions it uses the new *palloc()* procedure for allocation, for both the original OA and no reclamation it uses the regular *malloc()* procedure. Note that the OA method only uses the allocator to create its memory pool before the benchmark begins and performs no allocations during the benchmark itself.

The benchmarks were run with varying ratios of searches, inserts and removes, but we kept the ratio between inserts and removes at 1:1 in order to keep the size of the data structure constant throughout the benchmark. For linked lists, we ran the benchmarks with 5K nodes pre-inserted. For hash tables, we used both 10K and 1M nodes and a load factor of 0.75. The results are the mean of 10 runs of 1 second each, and we show the results in the form of throughput (number of operations per second) for every combination of threads from 1 to 32.

For all these experiments, we are not showing comparisons between the different approaches to memory remapping because we were unable to measure any difference in performance (outside a margin of error) between keeping the memory in the allocator, advising the operating system with *MADV_DONTNEED* and remapping with a shared memory region.

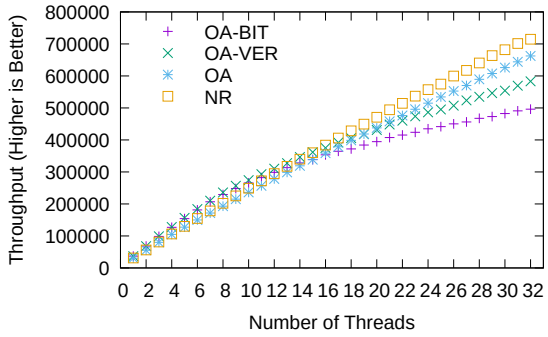
Results

Figure 4.4 shows the results for the benchmark using linked lists with 5K nodes pre-inserted. Figure 4.4(a) shows the case with only modifying operations (50% inserts and 50% removes) and Fig. 4.4(b) shows a more balanced set of operations (50% searches, 25% inserts and 25% removes). Figures 4.5 and 4.6 then show the results for the benchmarks using hash tables with 10K nodes and 1M nodes, respectively. For both benchmarks, we also have the case with only modifying operations (50% inserts and 50% removes) and with a more balanced set of operations (50% searches, 25% inserts and 25% removes).

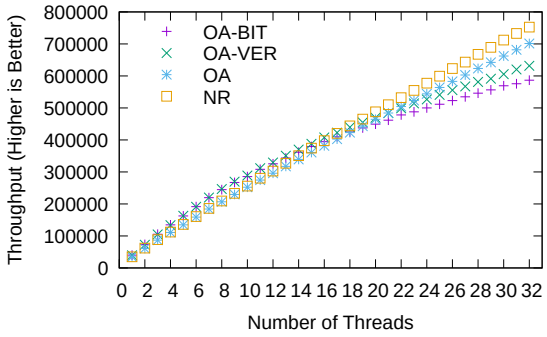
For linked lists with only modifying operations, the OA-VER method shows significant improvements to the OA-BIT method due to its ability to fire less warnings. This effect is somewhat reduced for linked lists with 50% searches, as there are fewer removes, and becomes negligible in both benchmarks using hash tables (Figs. 4.5 and 4.6) due to the much shorter chains.

For low amounts of threads, we can see that both OA-BIT and OA-VER outperform the OA and even the NR method for linked lists. This happens because with low amounts of threads our methods use less memory, keeping most of the memory used in lower level caches. With increasing number of threads, our two methods start using more memory due to the per thread caches of LRMalloc and thus loose this advantage to the OA method that has a memory pool of a fixed size and to the NR method that suffers from less overhead caused by synchronization between the many threads. A memory allocator with different characteristics could show a different behavior here. Linked lists are an unresting example to study the behavior of the system, but they are not the ideal tool when performance matters due to their asymptotic complexity characteristics.

The benchmarks using hash tables show a kind of inversion of the results. In general, the

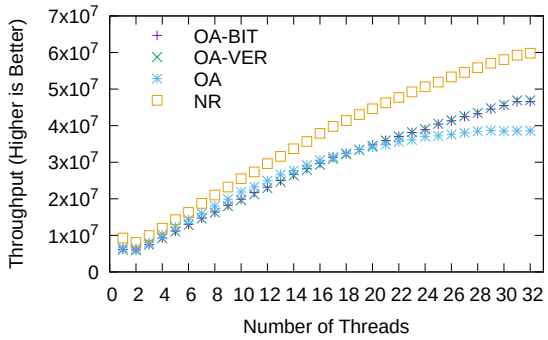


(a) 50% inserts and 50% removes

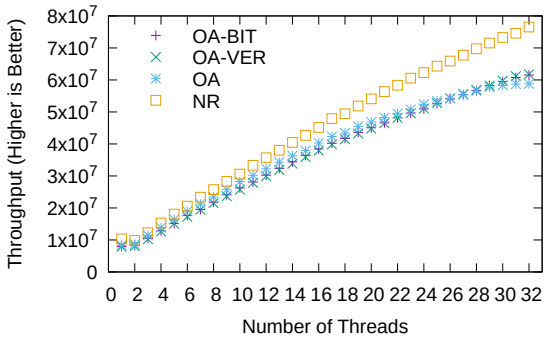


(b) 50% searches, 25% inserts and 25% removes

Figure 4.4: Linked List with 5K nodes

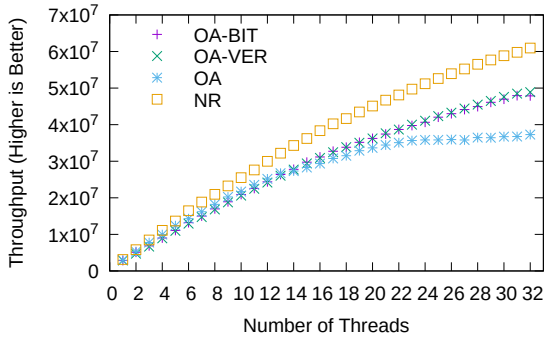


(a) 50% Inserts and 50% Removes

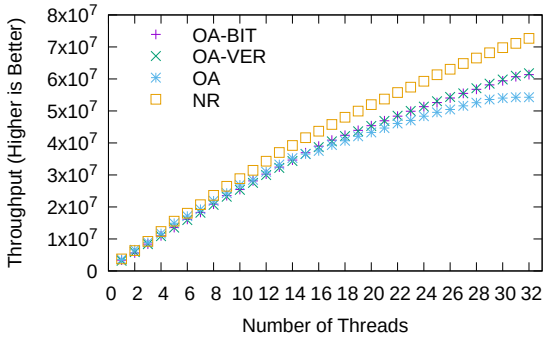


(b) 50% Searches, 25% Inserts and 25% Removes

Figure 4.5: Hash Table with 10K nodes



(a) 50% Inserts and 50% Removes



(b) 50% Searches, 25% Inserts and 25% Removes

Figure 4.6: Hash Table with 1M nodes

OA method shows slightly better performance than our methods for low amounts of threads, but a clear lack of scalability for higher thread counts. Here, since we are working with much higher throughput and larger amounts of memory, the weight of synchronization becomes much more relevant compared to memory usage and thus cache locality. The fixed size of the memory pool in the OA method proves detrimental as it requires much more recycling phases as the throughput and thread counts increase, causing synchronization to increase as well. In both our methods, we do not suffer from these drawbacks as the thread caches in the allocator and private

limbo lists allow for less synchronization and thus better scalability.

Please remember the main contribution is the added ability of releasing memory to the memory allocator/operating system and the simplification of the memory reclamation method, not the performance and scalability gains, even thought they are welcome.

4.3 Applying OA to the SLFHT Data Structure

The new SLFHT design handles memory in a somewhat different manner to most lock-free data structures. For example, the insertion of a node causes the replacement of the memory used by nodes in the same leaf array, as they are copied to a new leaf array, that then, replaces the previous one. This feature does not significantly impact the difficulty of the application of the SMR method to the data structure, as it only requires an extra retire call for the replaced node. However, it increases the load on the SMR method by requiring more memory than usual to be reclaimed, and thus, it becomes increasingly important that the SMR method is as efficient as possible. As presented in the previous section, the OA method achieves this high efficiency requirement, and with the use of a memory allocator with the palloc extension, we remove its main disadvantage of not being able to release memory to the operating system.

In this Section, we explain the key aspects of applying the OA method with palloc to the new SLFHT data structure. Then, we present some performance issues found during this implementation with the LRMalloc memory allocator, and the corresponding extensions and optimizations implemented to solve them. Finally, we will present the results we obtained and compare them to the original LFHT with the HHL SMR method.

4.3.1 Memory Reclamation on the SLFHT Data Structure

The OA method works by optimistically reading memory locations and only then verifying if such reads were valid by checking for a warning from a reclaiming thread. In the case of receiving a warning, the method forces the current operation to restart from a safe location. As hash nodes are never removed in the LFHT data structure (both in the original LFHT and the new SLFHT) we can regard any hash node as a safe location for restart when applying the OA method. We can also forego the verification of the warning when accessing hash nodes in the new SLFHT design. Since the type of the node is embedded into the reference to it, it means that we know the type of the node before accessing it, and thus do not need to read the node type from the node itself, which would require a validation of such read.

On the other hand, when accessing leaf arrays, we cannot read the whole leaf array and only then verify the warning. This is because we do not know the size of the leaf array before accessing it. Consider a scenario where we start accessing the leaf array by reading its size field, but such leaf array was already reclaimed and the memory reused. In such a case, the size field could have any value, and as such, could cause us to read beyond the end of the leaf array, which could

be memory that was never allocated and thus trigger a segmentation violation[6]. The general solution is to treat accesses to leaf arrays as two separate accesses, the first access reads the size and verifies if it is valid, and after ensuring we have a valid size for the node we can perform the second access that reads the array of nodes (that also needs to be verified after). Note that even if the node is reclaimed and reused after the first access succeeds its verification, we can be sure that a leaf array with that size at that memory address has existed in the past and as such the memory must be accessible, even if their contents are invalid. Whenever we fail to validate an access in a leaf array, we can always restart the operation from the hash node where the leaf array was.

For inserts and removes, where we need to use hazard pointer protection, only one hazard pointer is required as we only change one reference on a hash node per operation and hash nodes require no protection. As usual, to protect a node with a hazard pointer, we first set the hazard pointer to the node, and then verify the protection was successful by checking for a warning.

Algorithm 14 shows the new $Find()$ procedure extended with the OA method. The main changes are the new call to $Warning()$ after reading the size, and then a second call to $Warning()$ after finding the node or reaching the end of the leaf array without finding it, both of which restart the procedure from the current hash node if a warning is detected.

Algorithm 14 $Find(node\ Hn,\ level\ l,\ hash\ h,\ key\ k)$

```
1:  C ← Hn[Index(h, l)]
2:  if NodeType(C) = HASHNODE then
3:    return Find(C, l + 1, h, k)
4:  if C ≠ Null then
5:    Size ← C.size
6:    if Warning() then
7:      return Find(Hn, l, h, k)
8:    for i ← 0 to Size do
9:      if C.array[i].hash = h ∧ KeyEquals(C.array[i].key, k) then
10:       r ← C.array[i].value
11:       if Warning() then
12:         return Find(Hn, l, h, k)
13:       else
14:         return ⟨r, Hn, C, l, i⟩
15: if Warning() then
16:   return Find(Hn, l, h, k)
17: return ⟨Null, Hn, C, l, Null⟩
```

The changes to the $Insert()$ procedure are shown in Alg. 15. They include the hazard pointer protection followed by a warning verification, and the added $Retire()$ calls used to retire

[6]Note that this would not be a problem in the original OA method because the recycling mechanism acts similarly to a type-preserving allocator.

replaced nodes. The *Remove*() procedure, shown in Alg. 16, is changed similarly to the *Insert*() procedure.

Algorithm 15 *Insert*(*node Hn, key k, value v*)

```
1: h ← Hash(k)
2: l ← 0
3: loop
4:   ⟨r, Hn, C, l, i⟩ ← Find(Hn, l, h, k)
5:   if r ≠ Null then
6:     return ⟨False, r⟩
7:   HP ← C
8:   if ¬Warning() then
9:     if C ≠ Null ∧ C.size = THRESHOLD then
10:      N ← CreateExpandHash(l + 1, C)
11:      if CAS(Hn[Index(h, l)], C, N) then
12:        Retire(C)
13:        Hn ← N
14:    else
15:      N ← CreateAddNode(C, h, k, v)
16:      if CAS(Hn[Index(h, l)], C, N) then
17:        Retire(C)
18:        return ⟨True, v⟩
```

Algorithm 16 *Remove*(*node Hn, key k*)

```
1: h ← Hash(k)
2: l ← 0
3: loop
4:   ⟨r, Hn, C, l, i⟩ ← Find(Hn, l, h, k)
5:   if r = Null then
6:     return ⟨False, Null⟩
7:   HP ← C
8:   if ¬Warning() then
9:     N ← CreateRemoveNode(C, i)
10:    if CAS(Hn[Index(h, l)], C, N) then
11:      Retire(C)
12:      return ⟨True, r⟩
```

4.3.2 LRMalloc Performance

During the application of the OA method to the new SLFHT data structure, we found no performance problems with the OA method, however it revealed some inefficiencies in the LRMalloc memory allocator.

The first problem was a large amount of concurrent page faults that the operating system found difficult to deal with. We believe this issue was revealed by the increased amount of allocations in general and in different sizes of allocations (that correspond to more size classes being used). At the allocator level, this happens because when a new superblock is created, it is immediately initialized into a stack implemented as a linked list, which means that every block is initialized (written) with a reference to the next contiguous block. As discussed in Section 4.1.1, the operating system only actually assigns physical pages to virtual addresses when they are first written to, which in this case happens to the whole superblock at once during initialization of the linked list. When multiple threads concurrently initialize superblocks this triggers a bottleneck in the operating system.

The second problem was a large amount of synchronization when the allocator was flushing its thread caches back to the superblocks. This issue was hidden by the way we performed the benchmarks. As we try to ensure that when we remove a node, it is actually present in the data structure, we end up causing that every memory block is always allocated and freed by the same thread. This ends up leading to almost no synchronization between the different threads in the allocator with most data structures. But, with our new SLFHT data structure, when we replace a leaf array to insert or remove a node, that thread ends up freeing the replaced node that could now have been allocated by any other thread.

In what follows, we will start by introducing the solution to the first problem that involves changing the addressing mode used for the stack implemented as a linked list, and then show the solution to the second problem that involves optimizing the cache sizes and how memory is obtained from the operating system.

New Addressing Mode

One desired property for memory allocators is to avoid writing to the memory they are managing for the user. This property allows the virtual memory optimizations to propagate into the user, as it allows the physical frames to be allocated as late as possible. We were not able to fully achieve this property, but we were able to make sure that the memory allocator does not write to pages at least until the memory is freed, at which point it was very likely that it was already written to by the user. This allows the large amount of page faults we were observing to be reduced and more homogeneously distributed in time.

To make this possible, we changed the addressing mode used in the stack implementation that is used both in the superblock and the in thread caches. Now, instead of each block having the absolute address to the next block, it will have an address relative to its own address. More specifically, it would have how many addresses we need to move forwards or backwards in memory to find the next block. In the case where we have multiple blocks sequentially in memory, previously we would have the block in position 1 with a value 2 in order to reference the block in position 2, the block in position 2 with a value of 3 in order to reference the block in position 3, and so on. Now, all blocks would have a value of 1 to reference the next block, as that is the

number of positions we need to move forwards to find the next block, i.e., $1 + 1 = 2$, $2 + 1 = 3$, etc. With this change, all blocks would start with the same value in their next reference (please note that instead of the constant value 1, as in the previous example, the constant value to be stored would be the block size of the superblock), but we would still need to write such value to all blocks. But, if we introduce an offset to this addressing mode we can make the initial value whichever value we want, it would simply add an extra constant when calculating addresses. And, as we know that all memory that comes from the operating system comes all filled with zeros, we can choose an offset that corresponds to the initial value being zero. So, by choosing an offset equal to the block size for each superblock/thread cache, we have the initialized state of the superblock/thread cache being the whole superblock fully zeroed. This means that when we request memory from the operating system for a superblock it comes with our stack already initialized, and as such we no longer need to write anything to such memory. This allows us not only to never touch memory until it is freed, but also to forego part of the superblock initialization procedure.

Cache sizes

LRMalloc follows a very simple strategy in order to flush the blocks of memory from the thread caches to the superblocks. It moves partial lists of consecutive blocks in the cache that belong to the same superblock (i.e., that are already consecutive in the cache). It starts by checking to which superblock the first block in the cache belongs, and then goes through the list until a block belonging to a different superblock is found, at which point it moves such partial list to the superblock. The process is then repeated until the cache is empty. This strategy is very efficient in most scenarios, but when blocks are shuffled between threads, it can lead to a major synchronization bottleneck, which is what happens with the new SLFHT data structure. The bottleneck happens specially if the caches are large and there aren't too many superblocks, as in such a case we will end up with multiple threads trying to do a large amount of flush operation in a few superblock stacks.

Our first attempt to solve this problem was to try to sort the blocks in the cache per superblock before flushing them in order to minimize the number of synchronizations with the superblocks. However, we were not able to find a sorting strategy that is less costly (performance wise) than the cost of extra synchronization, and that can be accomplished without a significant memory cost. Note that inside the memory allocator we cannot rely on it to allocate memory, and as such we need to rely either on pre-reserved blocks or the process stack, both of which are severely limited.

As an alternative, we reduced the superblock/thread cache size from 16 MiB to 256 KiB, which mostly solved the bottleneck as it increases the total number of superblocks and reduces the number of blocks each thread has to flush. However, this change introduces a different problem. By reducing the size of the superblocks, the number of *mmap*() calls to obtain memory from the operating system grows at the same rate, which generates a different bottleneck. To solve this

new bottleneck, we decided to add a cache when requesting memory from the operating system. We create one cache per thread that is used every time a thread needs a new superblock. The first time a thread requests a new superblock, it starts by filling the cache by requesting a block of 16 MiB from the operating system, and then takes 256 KiB for the superblock. Subsequent requests will take other 256 KiB until the cache is empty, at which point it is again refilled from the operating system in the same way. Note that this cache does not change the way memory is returned to the operating system, which continues to happen in chunks of the size of a superblock. This change significantly increases the amount of calls that need to be done to release memory to the operating system, but we were unable to observe any performance degradation caused by that.

4.3.3 Experimental Results

In order to evaluate how the new SLFHT design performs with memory reclamation, we compare the new design with the OA method, against the old design with the HHL method and both designs without memory reclamation, in which memory is never reclaimed or reused.

The hardware used was a machine with 2 AMD Opteron™ Processor 6274 with 16 cores each, 16 KiB of L1 cache per core, 2 MiB of L2 cache per pair of cores and 12 MiB of usable shared L3 cache per CPU. It has a total of 32 GiB of DDR3 memory. The machine used was running the Ubuntu 22.04.1 LTS GNU/Linux operating system with the Linux kernel 5.15.0-91-generic.

We show results for multiple different scenarios with varying ratios of search, insert and remove operations. On each scenario, we used two sets of hash node sizes, namely 2^4 and 2^8 but, now we only show the ideal threshold value for expansion for each hash node size, as this comparison was already done in Section 3.5.4 and memory reclamation does not change their relative performance significantly. In all scenarios, we always perform a total of 10^7 operations with random keys. The benchmark has a preparation stage in which all keys that are going to be searched or removed during the benchmark are pre-inserted. To support concurrent randomicity on each thread, we used glibc PRNG, such that, for insertions we just insert random keys by giving each thread a different seed, and for search and remove, we reuse the seeds used for insertion, ensuring that we search or remove each key only once. The results are shown in throughput, which is obtained by dividing the number of operations performed by the average of the time taken to perform them. All benchmarks use the LRMalloc memory allocator, and the OA method takes advantage of the *palloc* extension introduced in Section 4.2.

Figures 4.7, 4.8 and 4.9 show the results for one kind of operation each, and Figs. 4.10, 4.11 and 4.12 show varying ratios of searches with the rest of the operations being inserts and removes in a one to one ratio, so the number of nodes in the data structure remain somewhat constant throughout the benchmark. In all graphs, the legend specifies the LFHT design using *LFHT* for the original design and *SLFHT* for the new design, that is followed by the memory reclamation method or *NR* for no reclamation.

Figure 4.7 shows that for the search only scenario, the OA method has a small performance impact, but it is still able to outperform the original design. We can also see in the original design, the version with memory reclamation outperforming the no reclamation version, this is likely due to the fact that the HHL method adds the hash flag to the hash node bucket entries, which allow the traversal to skip the hash node header and read the next bucket entry directly.

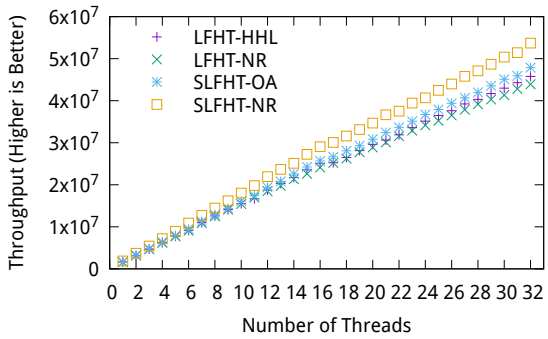
Figure 4.8 shows the worst case scenario for the new SLFHT design with the OA method as it is the only one that is actually performing memory reclamation, as in the original LFHT design no memory is being retired. Even so, the original LFHT design with the HHL method has a slight edge in Fig. 4.8(a) and is closely tied in Fig. 4.8(b)

In Fig. 4.9, we can see the largest impact caused by memory reclamation, which is expected as only remove operations are being performed. However, the new SLFHT design with the OA method still manages a good performance advantage compared to the original LFHT design with the HHL method.

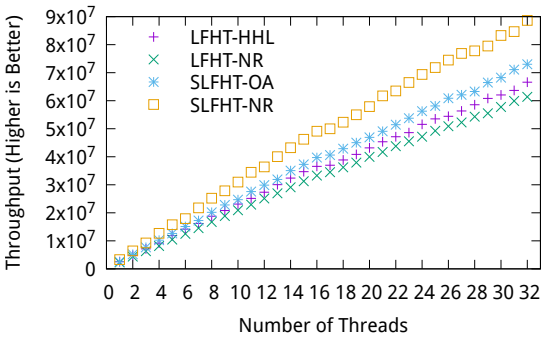
Figures 4.10 and 4.11 show the OA method outperforming or closely matching the no reclamation version, this is likely due to the effective use of the allocator thread caches achieved with reclamation, as the number of allocations and frees are closely matched with memory reclamation, but only allocations happen in the no reclamation scenario.

Figure 4.12 shows what is probably the closest to a real world scenario, and we are still able to see some performance advantage by the new SLFHT design.

Overall, we can see that for the memory reclamation versions, there is a gap in performance between the new SLFHT design and the original LFHT design and that the new SLFHT design is able to maintain an advantage in almost all scenarios.

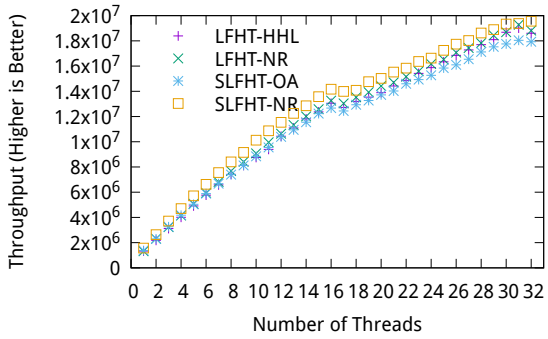


(a) Hash nodes with 2^4 entries and a threshold of 3

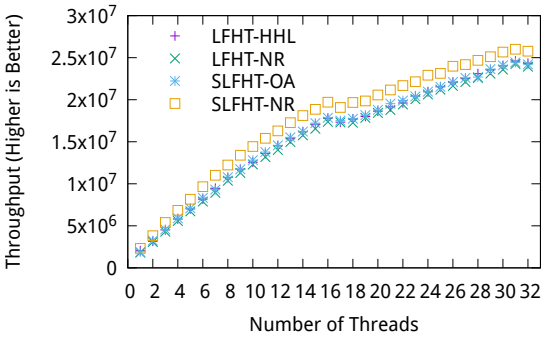


(b) Hash nodes with 2^8 entries and a threshold of 5

Figure 4.7: Throughput for the *Search Only* scenario

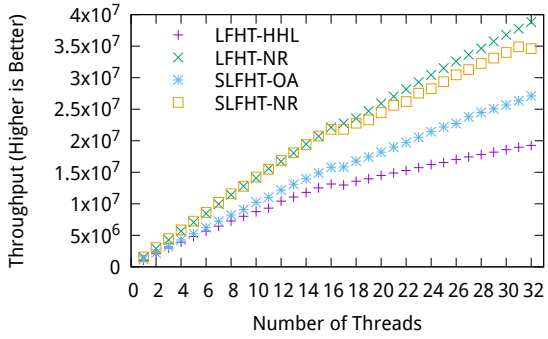


(a) Hash nodes with 2^4 entries and a threshold of 3

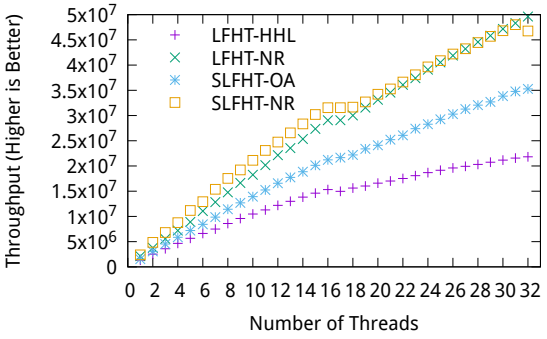


(b) Hash nodes with 2^8 entries and a threshold of 5

Figure 4.8: Throughput for the *Insert Only* scenario

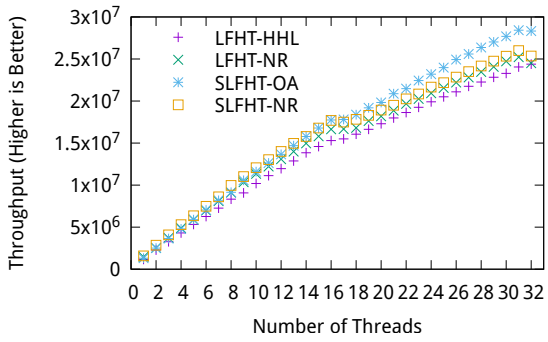


(a) Hash nodes with 2^4 entries and a threshold of 3

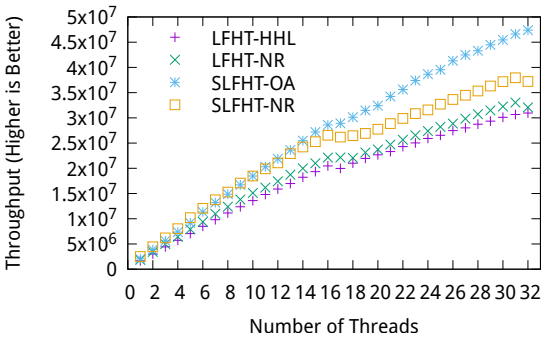


(b) Hash nodes with 2^8 entries and a threshold of 5

Figure 4.9: Throughput for the *Remove Only* scenario

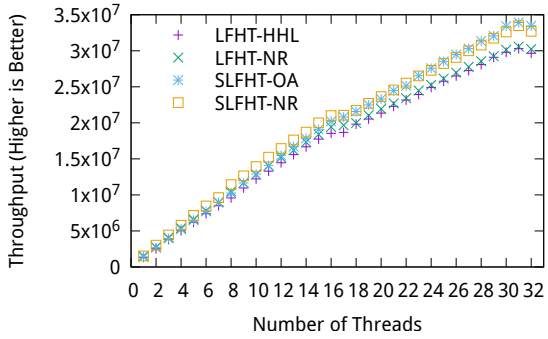


(a) Hash nodes with 2^4 entries and a threshold of 3

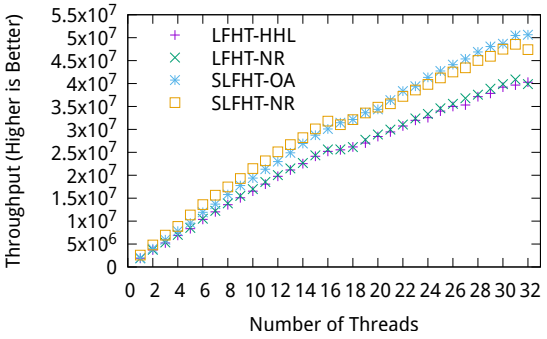


(b) Hash nodes with 2^8 entries and a threshold of 5

Figure 4.10: Throughput for the 50% *Inserts* and 50% *Removes* scenario

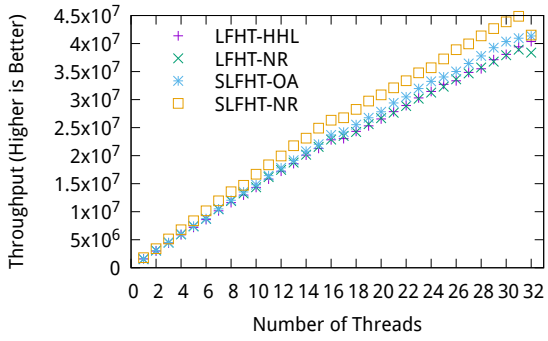


(a) Hash nodes with 2^4 entries and a threshold of 3

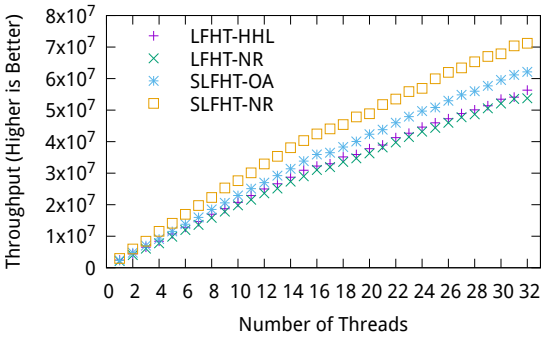


(b) Hash nodes with 2^8 entries and a threshold of 5

Figure 4.11: Throughput for the 50% *Searches*, 25% *Inserts* and 25% *Removes* scenario



(a) Hash nodes with 2^4 entries and a threshold of 3



(b) Hash nodes with 2^8 entries and a threshold of 5

Figure 4.12: Throughput for the 90% *Searches*, 5% *Inserts* and 5% *Removes* scenario

Chapter 5

Conclusion

This thesis focuses on lock-free memory management in general and how it can be practically used in the current hardware and software. The main focus is lock-free data structures and safe memory reclamation, but we explore how they can take advantage of current hardware instructions, operating system designs and modern memory allocators, and we show how they can be used in real world applications. We believe that the complexity of most lock-free designs can be a deterrent to adoption in the real world, as it exponentially increases the time required to implement them, the difficulty in debugging and the maintenance burden, so, making lock-free designs as simple as possible is a key aspect to take into consideration when designing them. As such, this thesis focus not only in achieving the best performance, but doing so while keeping the designs as simple and adaptable as possible. In all our work presented, we were able to at least maintain performance, and in most cases improve it, while making the designs practical.

5.1 Main Contributions

In Chapter 2, we start by introducing the relevant background behind lock-free systems that goes from theoretical progress guarantees to CPU design considerations. We also discuss important topics like the ABA problem, and extensively survey and describe the state-of-the-art in lock-free memory reclamation.

Chapter 3 focus on the LFHT data structure, and starts by describing it along with the HHL SMR method that is part of previous work. Then, we introduce a real world application of the LFHT data structure to a programming language system, namely the YAP Prolog system, and show how it can severely boost its performance and even outperform other state-of-the-art Prolog systems. It shows that lock-free data structures can be applied to real world systems and provide large benefits to such systems.

Next, we present a compression mechanism for the LFHT data structure. LFHT can be configured with varying hash node sizes resulting in a memory usage/performance trade-off. Using large hash node sizes to store low amounts of data can result in a relatively large memory overhead

without much benefit to performance, and small hash node sizes can result in a significant loss of performance for large amounts of data, with relatively minimal memory usage gains. The proposed compression mechanism allows the data structure to dynamically adapt to the amount of data by joining fully populated hash nodes, thus reducing the depth of the data structure, while also allowing the usage of small hash node sizes, and as such, have less memory overhead for small amounts of data. This allows the LFHT data structure to be used without having to take into consideration how much data will be stored during runtime, as it will dynamically adapt. We show the performance benefits this mechanism provides for large amounts of data, and how it outperforms a state-of-the-art lock-based hash table implementation.

In the next step, we have redesigned the LFHT data structure, in order to simplify it as much as possible, make it compatible with most SMR methods by eliminating the delegation problem, and improve its performance at the same time by making it more cache friendly. Even though we achieve a significant performance gain, we consider that the main advantage of the new design is its simplicity, as it makes it more desirable for adoption in real world applications, more reliable as it leaves less chance for mistakes during implementation, and allows for features like the compression mechanism to be more easily added.

In Chapter 4, we start with a brief introduction to memory management in modern systems, including how it is performed at the operating system and memory allocator level. We also introduce the LRMalloc memory allocator [48] and go further in detail on the optimistic access SMR method [19] as they are both used as a basis for our work.

Then, we describe how the optimistic access SMR method can be simplified and at the same time allow memory to be released to the memory allocator and the operating system, eliminating its main disadvantages. This is accomplished by extending the memory allocator API with palloc, a new primitive that ensures that allocated memory obtained through it will remain accessible even after being freed. We show that this extension allows for a major simplification of the SMR method without any significant performance impact, and with minor modifications to the memory allocator. We also show how this memory allocator extension can be applied to other memory allocators and how it can be useful in other systems, e.g., software transactional memory.

Finally, we combine the simplified optimistic access SMR method with SLFHT, the new simplified design of the LFHT data structure, and show how it can still outperform the original design with the HHL SMR method. This gain in performance is achieved with much less complexity in both the data structure and SMR method, making the whole implementation much easier to do and work with, and thus more desirable for real world usage.

In conclusion, this thesis discusses, proposes and describes how lock-free data structures can be applied to real world scenarios, how they can be made more dynamically adaptable to more use cases and how they can be simplified. We also show that by understanding the complete stack from hardware to software, we can be more efficient in managing memory in simpler ways. Most data structure and SMR designs tend to be complex, and thus interesting from an academic

point of view, but, often such complexity makes them undesirable for real world applications. In this thesis, we show that it is viable to use lock-free systems in real world scenarios, and that such systems can be made simpler without a performance cost, and thus more desirable for real world use.

5.2 Further Work

As further work, and starting from the new SLFHT design, we plan to add all the previously developed features of the old design to this new design, which include the compression mechanism and iterator described in this thesis and the hash node removal method developed by Areias and Rocha [8].

Another research topic is that the OA SMR method relies on hazard pointers for modification operations, which is not ideal for real world use, as we either need to know beforehand how many threads will be used or be able to dynamically allocate hazard pointers that then can never be freed. The VBR SMR method [81] solves this problem by using DWCAS with tags, but such a solution could result in a ABA problem without a type-preserving allocator. To prevent the ABA problem, we plan on researching further alternatives in order to be able to use the VBR SMR method with our palloc extension.

The compression mechanism with our palloc extension also poses a compatibility problem since allocations larger than the size class range are currently done directly through the operating system in the LRMalloc memory allocator, and as such, we have no means of properly recycling the virtual addresses after the memory as been freed. We thus plan to develop a lock-free coalescing and splitting mechanism, so virtual addresses can be recycled for allocations of any size inside the memory allocator.

By taking advantage of our experience with the YAP Prolog system, we also plan to explore other real world applications to which lock-free data structures and the new simplified OA method can be applied to and find how effective they can be.

Bibliography

[1] Anant Agarwal and M. Cherian. Adaptive backoff synchronization techniques. *SIGARCH Comput. Archit. News*, 17(3), 1989.

[2] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. StackTrack: An Automated Transactional Approach to Concurrent Memory Reclamation. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14. ACM, 2014.

[3] Dan Alistarh, William M Leiserson, Alexander Matveev, and Nir Shavit. ThreadScan: Automatic and Scalable Memory Reclamation. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, SPAA '15. ACM, 2015.

[4] Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. Concurrent deferred reference counting with constant-time overhead. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '21. ACM, 2021.

[5] Andrea Arcangeli, Mingming Cao, Paul E McKenney, and Dipankar Sarma. Using Read-Copy-Update Techniques for System V IPC in the Linux 2.5 Kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference*, USENIX ATC '03. USENIX Association, 2003.

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

[7] 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*, SBAC-PAD '17. IEEE, 2017.

[8] Miguel Areias and Ricardo Rocha. Towards an Elastic Lock-Free Hash Trie Design. In *Proceedings of the 20th International Symposium on Parallel and Distributed Computing*, ISPDC '21. IEEE, 2021.

[9] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. Fast and robust memory reclamation for concurrent data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16. ACM, 2016.

[10] William C. Benton and Charles N. Fischer. Interactive, scalable, declarative program analysis: from prototype to implementation. In *Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*. ACM, 2007.

[11] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11), 2000.

[12] Guy E. Blelloch and Yuanhao Wei. LL/SC and Atomic Copy: Constant Time, Space Efficient Implementations Using Only Pointer-Width CAS. In *34th International Symposium on Distributed Computing*, DISC '20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

[13] Guy E. Blelloch and Yuanhao Wei. Concurrent reference counting and resource management in wait-free constant time. *CoRR*, abs/2002.07053, 2020.

[14] Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: Lightweight memory management for non-blocking data structures. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13. ACM, 2013.

[15] Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13. ACM, 2013.

[16] Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15. ACM, 2015.

[17] Nachshon Cohen. Every data structure deserves lock-free memory reclamation. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018.

[18] Nachshon Cohen and Erez Petrank. Automatic memory reclamation for lock-free data structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '15. ACM, 2015.

[19] Nachshon Cohen and Erez Petrank. Efficient memory management for lock-free data structures with optimistic access. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15. ACM, 2015.

[20] Andreia Correia, Pedro Ramalhete, and Pascal Felber. OrcGC: Automatic Lock-Free Memory Reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21. ACM, 2021.

[21] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs. In *Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '10. IEEE, 2010.

[22] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2), 2012.

[23] Detlefs, David L. and Martin, Paul A. and Moir, Mark and Steele,Jr., Guy L. Lock-free reference counting. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01. ACM, 2001.

[24] Stan Devitt, Jos De Roo, and Helen Chen. Desirable features of rule based systems for medical knowledge. In *W3C Workshop on Rule Languages for Interoperability*. W3C, 2005.

[25] Dave Dice and Nir Shavit. What really makes transactions faster? In *ACM SIGPLAN Workshop on Transactional Computing*, TRANSACT '06. ACM, 2006.

[26] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC '06. Springer, 2006.

[27] Dave Dice, Maurice Herlihy, and Alex Kogan. Fast non-intrusive memory reclamation for highly-concurrent data structures. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM '16. ACM, 2016.

[28] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. Mixed-Size Concurrency: ARM, POWER, C/C++11, and SC. *SIGPLAN Not.*, 52(1), 2017.

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

[30] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007.

[31] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99. USENIX Association, 1999.

[32] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems*, 20(8), 2009.

[33] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. NBmalloc: Allocating memory in a lock-free manner. *Algorithmica*, 58(2), 2010.

[34] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with linux. *IBM Systems Journal*, 47(2), 2008.

[35] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01. Springer, 2001.

[36] 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), 2007.

[37] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier, 2011. ISBN: 9780080569581.

[38] Maurice Herlihy and Nir Shavit. On the nature of progress. In *Proceedings of the 15th International Conference on Principles of Distributed Systems*, OPODIS '11. Springer, 2011.

[39] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Dynamic-sized lock-free data structures. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing*, PODC '02. ACM, 2002.

[40] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02. Springer, 2002.

[41] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2), 2005.

[42] ISO/IEC 9899:2011. Information technology — programming languages — c. Standard, International Organization for Standardization, Geneva, CH, 2011.

[43] Prasad Jayanti. A Complete and Constant Time Wait-Free Implementation of CAS from LL/SC and Vice Versa. In *Proceedings of the 12th International Symposium on Distributed Computing*, DISC '98. Springer, 1998.

[44] Jaehwang Jung, Janggun Lee, Jeonghyeon Kim, and Jeehoon Kang. Applying hazard pointers to more concurrent data structures. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '23. ACM, 2023.

[45] Jeehoon Kang and Jaehwang Jung. A marriage of pointer- and epoch-based reclamation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '20. ACM, 2020.

[46] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3), 1980.

[47] Sangho Lee, Teresa Johnson, and Easwaran Raman. Feedback Directed Optimization of TCMalloc. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14. ACM, 2014.

[48] Ricardo Leite and Ricardo Rocha. LRMalloc: A Modern and Competitive Lock-Free Dynamic Memory Allocator. In *High Performance Computing for Computational Science*, VECPAR '18. Springer, 2019.

[49] Tianlin Li, Yiping Yao, Wenjie Tang, Feng Zhu, and Zhongwei Lin. An efficient multi-threaded memory allocator for PDES applications. *Simulation Modelling Practice and Theory*, 100, 2020.

[50] Udi Manbar and Richard E. Ladner. Concurrency control in a dynamic search structure. *ACM Trans. Database Syst.*, 9(3), 1984.

[51] Udi Manber. Concurrent maintenance of binary search trees. *IEEE Transactions on Software Engineering*, SE-10(6), 1984.

[52] Paul E. McKenney. Is parallel programming hard, and, if so, what can you do about it? *CoRR*, abs/1701.00854, 2017.

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

[54] Paul E McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read copy update. In *AUUG Conference Proceedings*, AUUG '01, 2001.

[55] Paul E McKenney, Dipankar Sarma, Ingo Molnar, and Suparna Bhattacharya. Extending RCU for realtime and embedded workloads. In *Ottawa Linux Symposium*. Citeseer, 2006.

[56] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*, PODC '02. ACM, 2002.

[57] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02. ACM, 2002.

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

[59] Maged M. Michael. Scalable lock-free dynamic memory allocation. *SIGPLAN Not.*, 39(6), 2004.

[60] 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.

[61] Pedro Moreno and Ricardo Rocha. Releasing memory with optimistic access: A hybrid approach to memory reclamation and allocation in lock-free programs. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '23. ACM, 2023.

[62] Pedro Moreno, Miguel Areias, and Ricardo Rocha. Memory Reclamation Methods for Lock-Free Hash Tries. In *2019 31st International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '19. IEEE, 2019.

[63] Pedro Moreno, Miguel Areias, and Ricardo Rocha. A compression-based design for higher throughput in a lock-free hash map. In *Euro-Par 2020: Parallel Processing: 26th International Conference on Parallel and Distributed Computing*, Euro-Par '20. Springer, 2020.

[64] Pedro Moreno, Miguel Areias, and Ricardo Rocha. On the implementation of memory reclamation methods in a lock-free hash trie design. *Journal of Parallel and Distributed Computing*, 155, 2021.

[65] Pedro Moreno, Miguel Areias, Ricardo Rocha, and Vítor Santos Costa. On the implementation of a lock-free atom table in a prolog system. In *Proceedings of the 16th International Symposium on High-level Parallel Programming and Applications*, HLPP '23, 2023.

[66] P. Moura. ISO/IEC DTR 13211–5:2007 Prolog Multi-threading Predicates, 2008.

[67] Chris Mungall. Experiences using logic programming in bioinformatics. In *Proceedings of the 25th International Conference on Logic Programming*, ICLP '09. Springer, 2009.

[68] Ruslan Nikolaev and Binoy Ravindran. Snapshot-free, transparent, and robust memory reclamation for lock-free data structures. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '21. ACM, 2021.

[69] Pierre M. Nugues. *An Introduction to Language Processing with Perl and Prolog: An Outline of Theories, Implementation, and Application with Special Consideration of English, French, and German (Cognitive Technologies)*. Springer, 2006.

[70] David Page and Ashwin Srinivasan. ILP: A Short Look Back and a Longer Look Forward. *Journal of Machine Learning Research*, 4, 2003.

[71] J. C. Pereira, P. Moreno, and R. Rocha. Memory Reclamation for an Elastic Lock-Free Hash Trie Map Design Using Hazard Pointers. In *13th INForum - Simpósio de Informática*, INForum '22, 2022.

[72] Erez Petrank. Can parallel data structures rely on automatic memory managers? In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '12. ACM, 2012.

[73] William Pugh. Concurrent maintenance of skip lists. Technical Report UMIACS-TR-90-80, 1990.

[74] Pedro Ramalhete and Andreia Correia. Brief announcement: Hazard eras - non-blocking memory reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17. ACM, 2017.

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

[76] V. Santos Costa. On Supporting Parallelism in a Logic Programming System. In *Workshop on Declarative Aspects of Multicore Programming*, DAMP '08, 2008.

[77] Vítor Santos Costa, Kostis Sagonas, and Ricardo Lopes. Demand-Driven Indexing of Prolog Clauses. In *Proceedings of the 23rd International Conference on Logic Programming*, ICLP '07. Springer, 2007.

[78] Dipankar Sarma and Paul E McKenney. Making RCU safe for deep sub-millisecond response realtime applications. In *Proceedings of the 2004 USENIX Annual Technical Conference*, USENIX ATC '04. USENIX Association, 2004.

[79] Sangmin Seo, Junghyun Kim, and Jaejin Lee. SFMalloc: A Lock-Free and Mostly Synchronization-Free Dynamic Memory Allocator for Manycores. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11. IEEE, 2011.

[80] Gali Sheffi and Erez Petrank. The ERA Theorem for Safe Memory Reclamation. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*, PODC '23. ACM, 2023.

[81] Gali Sheffi, Maurice Herlihy, and Erez Petrank. VBR: Version Based Reclamation. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21. ACM, 2021.

[82] Ajay Singh, Trevor Brown, and Ali Mashtizadeh. NBR: Neutralization Based Reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21. ACM, 2021.

[83] Daniel Solomon and Adam Morrison. Efficiently reclaiming memory in concurrent search data structures while bounding wasted memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21. ACM, 2021.

[84] Håkan Sundell. Wait-free reference counting and memory management. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '05. IEEE, 2005.

[85] Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14. ACM, 2014.

[86] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95. ACM, 1995.

[87] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[88] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. Interval-based memory reclamation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18. ACM, 2018.

[89] J. Wielemaker. Native Preemptive Threads in SWI-Prolog. In *International Conference on Logic Programming*, number 2916 in ICLP '03. Springer, 2003.

[90] J. Wielemaker and K. Harris. Lock-free atom garbage collection for multithreaded prolog. *Theory and Practice of Logic Programming*, 16(5-6), 2016.

[91] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, IWMM '95. Springer, 1995.