Parallel Query Evaluation in Streaming Environments

Murray Steele

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh

2021

Abstract

With the end of Moore's law firmly at hand – and modern microarchitectures tending towards many-core designs – programmers and hardware architects alike must now explore how parallel architectures can be fully exploited for maximum performance. However, even with a general acceptance of the paradigm of parallel programming by mainstream programming languages, producing both correct and bug-free software for parallel microarchitectures is still as much a challenge as ever. Whilst generalised automatic parallelisation is still an open problem, well defined domain specific code generation may be well suited to known methods of automatic parallelisation.

This report continues work started last year in implementing automatic methods of parallel code generated in DBToaster – an SQL-to-C++/Scala compiler – by way of designing, implementing, and evaluating thread-safe MultiMap data structures which are used to represent relations within DBToaster-generated code.

Last year, parallelisation of high-level intermediate language statements was explored in order to produce parallelised programs utilising OpenMP's more recent inclusion of dependent tasks. This year parallelisation at a lower level is explored, in the context of performance-oriented thread-safe data structures where problems universally found shared-memory systems must be overcome.

Acknowledgments

I would like to thank my supervisor Milos Nikolic for his guidance and support over the last two years. It is thanks to them that I have had the opportunity to explore and learn so much about parallel systems and technologies – something that has undeniably influenced me greatly.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Since the early 2000s microarchitectures have become increasingly parallel as single-core performance improvements have become harder to come by. Ultimately, heat dissipation and instruction level parallelism problems brought the demise of the single-core computer and now almost all consumer processors consist of multiple processor cores[1]. With this advancement in microarchitecture design it is increasingly important for software to utilise parallelism to be scalable.

Parallel programming has come a long way since operating systems and microarchitectures first exposed parallel primitives, but with many mainstream languages designed for single-threaded workloads it is still a difficult task to design and implement parallel code that utilises the parallelism of modern systems. With such difficulties, it makes sense to centralise efforts in parallelisation and distributed computing to reduce the errors inherent in parallel programming. Unlike serial programming, parallel programming requires analysing complex interactions between multiple threads of execution which may contain many subtle bugs.

DBToaster[2] – an SQL-to-C++/Scala code generator – is an ideal project for exploring the automatic parallelisation of code as the code it produces can be analysed and used to inform parallelisation efforts.

1.2 Previous Work Carried Out

Last year the problem of automatic parallelisation of DBToaster-generated code was approached by way of adding parallel primitives to DBToaster's intermediate language – M3 – and implementing a parallelisation pass capable which transformed and optimised M3 programs into parallel M3 programs. This approach produced mixed results which demonstrated that this form of parallelisation could be effective, but also that it is not universally applicable.

With this in mind, the decision was taken to adjust the approach for this year to con-

tinue efforts towards automatic parallelisation by focusing on the parallelisation of simple loop constructs generated by DBToaster. Parallel iteration is already a well researched and supported area of parallel computing and is supported well by the project's parallelisation tool of choice – OpenMP[3] – making it an ideal target for parallelisation efforts.

Whilst loops over set ranges or contiguous data structures is a solved problem, DBToaster-generated loop constructs often iterate over the elements of *MultiMaps* – a custom data structure which models relations – and may involve modifying other MultiMaps by either adding or removing elements. For this reason, thread-safe MultiMaps are needed to enable loop-based parallelism in DBToaster-generated code. As such, the focus of this year's work is to design, implement, and evaluate performant thread-safe MultiMaps capable of concurrent modification.

1.3 Goals & Achievements

This year the core goal of the project was initially to enable parallelisation of DBToaster-generated loop constructs by way of designing, implementing, and integrating thread-safe MultiMaps into the existing DBToaster project. As the project progressed, the scope was adjusted to focus on the three MultiMap implementations produced over its duration, and their evaluation over a set of microbenchmarks tailored to their potential future use within DBToaster.

The project has succeeded in implementing three thread-safe MultiMaps inspired by state-of-the-art key-value data structure implementations found in recent publication, and in exploring the implications of modern microarchitectural designs and capabilities on the performance and implementation of thread-safe data structures. Beyond the three thread-safe MultiMap implementations, the project has also produced a wait-free memory pool implementation supporting parallel iteration over its allocated elements, and a suite of benchmarks for evaluating thread-safe MultiMap performance.

Finally, through evaluation of the three thread-safe MultiMap implementations, the project has succeeded in highlighting areas of difficulty which future work can focus on.

1.4 Overview & Organisation

The project is presented in a linear fashion, which first reviews relevant technologies and data structure implementations before moving onto a discussion of design, implementation, and evaluation of the three thread-safe MultiMaps. Afterwards, proposals of future work based on the final evaluation are presented, along with a reflection of the work undertaken.

The formal report structure is detailed below:

1. **Review** – A review of foundational and existing technologies which influence the design and implementation of any thread-safe data structure, including the thread-safe MultiMaps presented in this report.
2. **Methods** – A walk-through of the design of the three MultiMaps produced by the project, the memory pool designed to support them, and a discussion of their theoretical advantages and disadvantages.
3. **Implementations** – A walk-through of significant implementation details of the three thread-safe MultiMaps and the memory pool, as well as noteworthy optimisations and technologies.
4. **Evaluation** – An evaluation of the three thread-safe MultiMap implementations with respect to the current serial MultiMap implementation, over a set of tailored microbenchmarks.
5. **Conclusion** – An reflection of work undertaken during the project, its successes and failures, and work proposed for the future.

Chapter 2

Review

In this chapter, technologies and implementations relevant to thread-safe data structure design and implementation, and to the project are discussed. The chapter covers relevant features of parallel microarchitectures which enable parallelism, the OpenMP API used to implement all three thread-safe MultiMaps, DBToaster's MultiMap implementation, and existing implementations of performant thread-safe key-value data structures representing the state-of-the-art.

2.1 Modern Microarchitectures

Understanding modern microarchitectures is an extremely important part of designing and implementing thread-safe data structures. The capabilities of modern parallel microarchitectures vary greatly – from tiny embedded processors to huge multi-threaded behemoths, there is no universal implementation, and so this section focuses on general paradigms and technologies implemented across many systems.

Modern parallel microarchitectures can be divided into two paradigms – simultaneous multi-threading (SMT) and chip multiprocessing (CMP) – which implement thread-level parallelism in two distinct ways. SMT implements multiple threads by sharing the resources of a single core between them, whilst CMP implements multiple threads by way of having several processing cores which have their own dedicated resources. For this section, the multi-cored CMP microarchitecture is the main focus, as it is the more common of the two today and has greater performance implications. However, an SMT system is considered during the evaluation later on (see chapter 5).

2.1.1 Shared Memory Model

Any multi-core microarchitecture requires a method of allowing multiple threads to collaborate on the same data together – this is the core technology that allows for data structures to be used by multiple threads. This is universally implemented using some form of shared memory which all hardware threads have access to. For SMT this solves the problem of sharing data between threads as cache and memory slots are shared between SMT threads, but for CMP this presents fundamental problem: with

each core having its own cache, how does one ensure that all cores have a consistent view of shared memory when multiple cores may be interacting with the same shared memory? The solution to this problem is cache coherency protocols.

Cache coherency protocols are responsible for ensuring that cache lines in separate caches are kept consistent with one another, such that if one core reads data from a cache line it will always return the most up-to-date version of that data across all cores and main memory, even if that means fetching it from another cache or main memory. Cache coherency protocols can be implemented in a variety of ways, but among the most popular today are the MSI and MESI protocols[4].

Universally, cache coherency protocols are implemented by utilising a shared bus between cores to communicate requests for, or modifications to, cache lines. This information is then used by other cores to share or invalidate their own cache lines, possibly requiring that subsequent reads or writes on that cache line invoke communications with other cores before being executed.

In solving the cache coherency problem, cache coherency protocols add some overhead in their communication with other cores. For every invalidation or share of a cache line, future operations on that cache line may require that some number of cycles is spent communicating with other cores before doing so. For programs utilising many threads to operate on shared memory, this is generally unavoidable, but its effect may be lessened by avoiding *false sharing*[5], where independent data is stored within the same cache line, and thus modifications to either data element invokes a cache invalidation. By manually aligning independent data to separate cache lines, cache invalidations can be avoided, and thus the overheads associated.

2.1.2 Atomic Instructions

Cache coherency succeeds in maintaining a consistent view of memory across multiple cores, but leaves an important problem unsolved. Reads and writes of memory are kept consistent by cache coherency protocols, but instructions that need to read and write to memory may have the corresponding memory change between their read and write, resulting in unexpected behaviour. This problem is solved by atomic instructions – a special set of *atomic* instructions provided by many ISAs[6] which disallow any interleaved reads and writes to their corresponding memory locations for the duration of the instruction.

Whilst atomic instructions can be used to ensure atomicity of single instructions, it is still possible for atomic instructions to be interleaved with one another. For this reason, the compare-and-swap (CAS) instruction sees widespread use to implement more complex atomic operations. It is defined as:

$$CompareAndSwap(a,b,c) = \begin{cases} a \leftarrow c, & \text{if } a = b \\ b \leftarrow a, & \text{otherwise} \end{cases}$$

Where a, b, and c are in shared memory. Compare-and-swap allows us to read b from memory, do some kind of computation to produce c, and then only write it back to

the memory location of a if $a = b$. CAS has seen wide-spread utilisation in many high-performance thread-safe data structure implementations[7, 8] for its ability to implement more complex atomic transactions on data which may be repeated until a CAS instruction succeeds.

Because they are implemented at the level of microarchitectures, atomic instructions represent the lowest-level method of performing atomic transactions on data. Whilst individual atomic instructions are fast because of this, using atomic instructions on the same cache blocks are still prone to cache invalidations. Therefore, even though atomic instructions afford the user a fast method of applying thread-safe modifications to data, care must still be taken in data alignment to ensure they are used efficiently.

2.2 Locking

Though atomic instructions offer atomicity of individual operations, more generic techniques for multi-staged transactions on shared memory are needed. These methods fall under the category of *locking*, wherein some object or data is used to signify a thread's ownership of data. Within this category, there are many implementations of locking objects specialised for specific use-cases. Instead of delving into these, this section covers two types of lock used commonly in parallel code – the mutex and spinlock.

2.2.1 Mutex

A mutual exclusion object, or *mutex*, is an object which allows a thread to acquire it to signify ownership over some associated data, and later release it so that another thread can access that data[9]. Mutexes are generally provided by the operating system, and as a result they see widespread support across a large number of systems.

Behind the scenes, operating systems often make some optimisations around mutexes to maximise use of resources, such as sending threads to sleep that are waiting on mutexes so that another thread can use the processor's resources. Whilst this is a good policy for maximising throughput via executing many programs in parallel, it is a detriment for the thread being put to sleep as it requires a context switch wherein state must be saved for restoration later when the threads' execution is resumed, and allows the next thread that is executed on the core to evict cache lines and affect other caching components such as the TLB[10, 11]. This means that when a thread resumes it may have to re-read cache lines and rebuild its TLB.

In this project, only Linux's implementation of the mutex – the *futex* – is relevant. The futex is an optimised mutex which attempts to avoid context switches by attempting to acquire the underlying mutex several times before giving up and yielding to a context switch. For sufficiently short locking periods this results in far fewer context switches.

2.2.2 Spinlock

Like the mutex, a spinlock can be acquired and released by a thread to indicate ownership of associated data. Unlike the mutex, a core design choice of the spinlock is that context switches are avoided at all costs. As a result, spinlock will never put a thread to sleep, instead preferring that a thread continue looping (or spinning) to eventually acquire it.

Spinlocks are usually implemented with just a single flag which is atomically modified to declare that the spinlock is currently owned or free. This can be implemented using the compare-and-swap instruction by utilising its expected value parameter such that it only succeeds if the actual value of the flag used to implement the spinlock indicates it is not locked. Whilst simple, a well optimised spinlock can be extremely performant, even outperforming pure atomic compare-and-swap instructions[12].

2.2.3 Contention

Unlike atomic instructions, both mutexes and spinlocks are capable of supporting atomicity of arbitrary operations on arbitrary data. However, locking methods in general suffer from performance issues when under *contention*. Contention is when several threads are trying to acquire the lock concurrently, and causes different problems for mutexes and spinlocks. For mutexes, high contention means that threads may need to wait until several threads have completed their use of the mutex before being able to do its own operation, causing context switches even for adaptive designs such as the futex. For the spinlock, contention means that many modifying atomic operations are being attempted on the lock's flag concurrently, causing many cache invalidations, thus increasing the latency of locking and unlocking the spinlock.

The effect of contention is much more detrimental for the spinlock than the mutex because cache invalidation also affects the latency of unlocks, and whilst this can be somewhat offset using special *pause* instructions supported by some ISAs[13, 14], the best spinlock designs are generally far more inefficient in high-contention scenarios than mutexes[15].

2.3 OpenMP

OpenMP is an API implemented by various compilers which allows C, C++, and Fortran programmer to decorate code with special annotations in order to enable parallelisation of that code. Last year, the OpenMP parallelisation API was selected for its performance, ergonomics, and general availability through a number of compilers. This year continues the use of OpenMP for the same reasons, but for different tasks. This section serves as a refresher of what OpenMP is, and how its features support this year's project. Specific applications of OpenMP are discussed inline in future chapters, namely chapters 4 and 5. This section is limited to discussing OpenMP's lock and atomic support.

2.3.1 Atomics & Locks

Being a universal parallelisation API, OpenMP provides a high-level implementation of a lock object, as well as annotations for specifying operations as atomic. The lock object is typically implemented in terms of operating-system supplied mutexes, and thus on Linux platforms it is implemented internally as a futex.

Atomic annotations can take on several forms, allowing the programmer to specify if an operation is an `atomic read`, an `atomic write`, or simply `atomic`, in which case the compiler will infer what instruction can be used to implement it.

Currently supported versions of OpenMP[16] do not implement the compare-and-swap instruction, which was added in the more recent OpenMP v5.0 specification[17]. As compare-and-swap is such an integral operation for many concurrent data structure or spinlock implementations, the the GCC compiler's `__atomic` builtin[18] is utilised the presented implementations when the CAS instruction is needed.

2.4 DBToaster

DBToaster is an SQL-to-C++/Scala compiler capable of transforming input SQL schemas and queries into an intermediate language – M3 – which is optimised before being transformed into a lightweight query engine program which is capable of applying efficient incremental updates to associated queries' results[2].

Whilst the MultiMap data structure used to represent database relations within DBToaster is main focus of this project, understanding how DBToaster beyond what has already been discussed does not play a large role in the understanding the project. Instead, a greater focus is given to the specific implementation of DBToaster's MultiMap and supporting memory pool which are continuously referred back to throughout the report. Furthermore, only the C++ implementation of the MultiMap is considered for this project.

2.4.1 MultiMap

The MultiMap is the data structure used within DBToaster-generated code to represent relations. As previously discussed, DBToaster code is expressed in terms of MultiMap instances and therefore, a performant thread-safe MultiMap is essential for parallelisation of loop constructs.

A MultiMap encodes not only the tuples in a relation, but also a primary hash index for direct single-element lookups. In addition, optional secondary hash indexes for looking up lists of tuples satisfying an associated condition can also be maintained by a MultiMap. As a result, DBToaster's MultiMap implementation is a composite data structure containing a hashmap for the primary hash index, and hashmap-like data structure for the secondary hash indexes.

The primary hash index is a standard open-addressed hashmap consisting of a single bucket array which items are placed into using their hash. Each bucket is either empty, or contains a linked list of nodes containing reference to elements in the associated

relation, along with a count of how many of that element the relation contains so as to avoid duplicates. The first node in any bucket is placed directly inside the bucket array as to avoid a separate allocation for every single element.

The hashmap used to implement the secondary indexes is almost identical to the primary hash index's hashmap with the exception that the nodes forming chains at each bucket are headers for the linked lists of elements conforming to a condition associated with the secondary index. For example, the associated condition could be that all elements with an identical field x are grouped together. These groups are called slices.

Figure 2.1 gives a simple example of the structure of the primary and secondary indexes with a handful of elements.

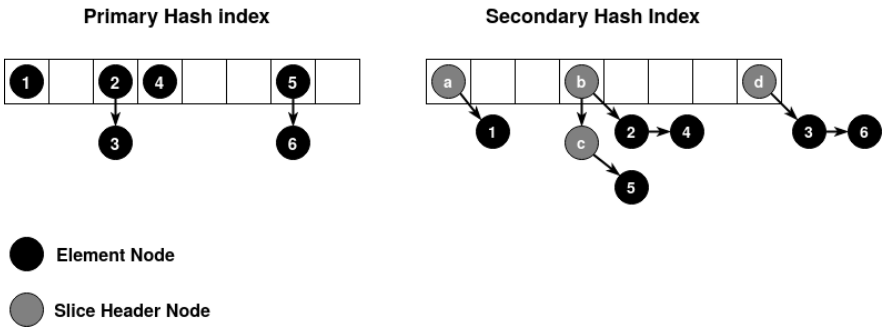


Figure 2.1: Example of MultiMap primary and secondary index structure.

All relation elements are allocated via an associated memory pool and stored as a linked list by the MultiMap. All other type of elements – such as bucket chain nodes – are allocated using their own associated memory pools. as an additional optimisation, memory pool for the same type are shared throughput all MultiMap instances via the singleton pattern.

To manipulate the primary and secondary hash indexes, a few basic hashmap operations are defined on them which are composed to form operations on the entire MultiMap data structure. These operations are: `insert`, `erase`, and `clear`. In addition, the primary hash index defines `get`, and secondary hash indexes define `slice`. These operations are implemented as they would be for a conventional hashmap, and `slice` is implemented almost identically to `get`. These are defined in greater detail below:

- `get(e)` – Gets an element and its count from the associated primary hash index such that e and its hash are equal to the element and its hash.
- `slice(e)` – Gets the slice of elements corresponding to the same condition satisfied by e as defined by the condition associated with the secondary hash index.
- `insert(e)` – Inserts the element e into the primary or secondary hash index.
- `erase(e)` – Removes the element e from the primary or secondary hash index.
- `clear()` – Removes all elements from the primary or secondary hash index.

The MultiMap data structure's core operations are defined with respect to these operations on the primary and secondary hash indexes. For the project, the `foreach` and

`slice` operations were formalised to be encapsulated by the MultiMap. The composite operations are defined in detail below:

- `get(e)` – Gets an element equal to e if it exists in the MultiMap. Otherwise, returns NULL.
- `getOrDefault(e, v)` – Gets an element equal to e if it exists in the MultiMap. Otherwise, returns the given default v.
- `foreach(f)` – Applies the function f to each element in the MultiMap.
- `slice(e, i, f)` – Applies the function f to each element of the slice corresponding to the same condition satisfied by e associated with the ith secondary index of the MultiMap.
- `add(e, n)` – Inserts the given element e into the MultiMap's primary and secondary indexes if it does not exist, and adds n to its count.
- `addOrDelOnZero(e, n)` – Inserts the given element e into the MultiMap's primary and secondary indexes if it does not exist, and adds n to its count. If the final count of the element is zero, then it removes it from the primary and secondary indexes.
- `setOrDelOnZero(e, n)` – Inserts the given element e into the MultiMap's primary and secondary indexes if it does not exist, and sets its count to n. If the final count of the element is zero, then it removes it from the primary and secondary indexes.
- `clear()` – Clears the underlying primary and secondary indexes of the MultiMap.

Because the MultiMap is essentially a collection of hashmaps, its operations inherit their complexities from the conventional hashmap data structure, multiplied by the number of hash indexes accessed by each operation. Appendix A presents the MultiMap's operations as pseudocode in terms of its `primary_index` and `secondary_indexes`. We omit `foreach` and `clear` as their implementations are trivial.

2.4.2 Memory Pool

As previously mentioned, a memory pool is used for allocating MultiMap elements and nodes within the primary and secondary hash indexes. The memory pool utilised for all such allocations is implemented as a linked list whose blocks exponentially increase with each new block allocated. Each block is split into a number of slots which can each hold an instance of the type associated with the memory pool, such as a MultiMap's element or primary hash index bucket node. The memory pool's API consists of three operations – `acquire`, `release`, and `releaseChain`. `acquire` allows the user to acquire a new instance of the type associated with the memory pool, `release` allows data acquired from the memory pool to be released, and `releaseChain` allows the user to release a chain of elements with a single operation. In addition to its list of chunks, a free-list which tracks previously released slots is maintained, and used to

allocate previously released before new ones. Figure 2.2 presents the structure of the memory pool.

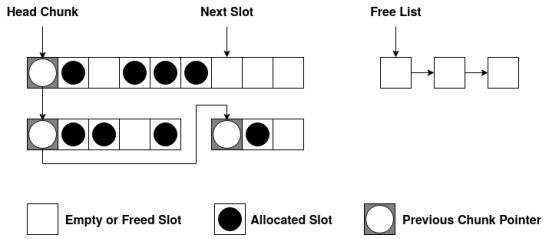


Figure 2.2: DBToaster memory pool structure.

2.5 Thread-Safe Data Structures

Given that the MultiMap's hash indexes are implemented as hashmaps, the most relevant existing implementations of thread-safe data structures to the project are hashmaps and other key-value store data structures. This section reviews state-of-the-art hashmap and trie implementations which have heavily influenced the thread-safe MultiMap implementations presented in this report.

2.5.1 HashMaps

HashMaps are one of the most widely used data structures in modern programming languages, and, as a result, significant research and implementations exist for both serial and thread-safe variants. In the realm of serial hashmaps, most high-performance implementations have settled on closed-hashed, open-addressed designs which utilise SIMD for fast searches across their bucket arrays[19]. For thread-safe hashmaps, there are several state-of-the-art designs with different performance characteristics and trade-offs.

The core problem for any thread-safe hashmap is handling resizes of the underlying bucket array – an operation which takes exponentially longer as the bucket array grows (assuming an exponential resizing scheme). How an implementation handles this tends to one of two options: the bucket array size is never resized and buckets are implemented as chains, or the hashmap is split into segments that are resized independently and can host any particular hashmap design.

Static bucket array based thread-safe hashmaps prioritise performance of operations under the assumption that the user can make a good choice of bucket array size. Under such an assumption, this design is able to leverage atomic instructions – mainly compare-and-swap – to implement fast operations on it's buckets' chains of elements which are implemented as linked lists. If practice, this type of design can result in great throughput thanks to its use of atomics, but there are many workloads where the number of elements that will be stored within a hashmap is not known ahead of time, including our own. Though not in use today, such a design has been a part of the development of rust's crossbeam library[20].

Segment-based hashmaps are a more practical approach to thread-safe hashmap design as they allow bucket arrays to be resized by separating the hashmap into segments which function as independent sub-hashmaps. Concurrent operations are supported by locking segments, but concurrent operations on the same segment are disallowed as a result. Implementation of this design can be seen in the Java 1.7 standard library as its `ConcurrentHashMap` data structure[21, 22], and it has also been applied to Google's own `flat_hash_map`[23] – a state-of-the-art serial hashmap – to produce a performant thread-safe one[24].

Beyond these two paradigms, there exist designs which attempt to combine fast atomic operations with dynamic resizing, but these designs often require methods to solve a fundamental problem in atomic data structures – the ABA problem[25]. The ABA problem is out of scope for this project as we no solely atomics-based MultiMaps are implemented, but there exists a large field of work covering methods of solving it for thread-safe data structures.

2.5.2 Tries

Tries are not as ubiquitous as hashmaps, but within the realm of thread-safe key-value data structures they have some ideal characteristics. Unlike hashmaps, tries do not require lengthily resize operations, instead extending their branches with new arrays to accommodate more elements as collisions occur. This feature alone solves the core problem that any thread-safe hashmap needs to solve to be practical. Unlike hashmaps though, tries inherit their characteristics from trees, and thus aren't usually as cache-friendly and have greater average operation runtime complexities. Most relevant to this project is the hash-array-mapped-trie (HAMT) variant, which utilises its hierarchy of arrays as bucket arrays[26]. Using a portion of an element's hash for each level of the HAMT's hierarchy, an element can be located, inserted, or erased quickly and in a manner similar to a hashmap.

The state-of-the-art in thread-safe HAMT implementation is the *concurrent trie*, or *Ctrie*[27], whose current implementation targets the JVM[28]. The Ctrie utilises atomic instructions – namely compare-and-swap – to insert, modify, and erase elements within its structure without any form of locking. For efficiency, the Ctrie's branches grow and shrink as elements are inserted and removed, and because all bucket arrays used within the structure are of a uniform size, they can be reused when the structure later grows.

Elements within the Ctrie are mapped to buckets using their hash, with each level using a portion offset from the hash's least significant bits, such that more bits used to locate, insert, and erase elements deeper in the trie's bucket array hierarchy. A later version of the data structure added support for atomic snapshots, as well as solving some of the initial design's implementation issues[29].

Though its structure should handicap the trie data structure, its reuse of bucket arrays, along with its atomics-based implementation means that the Ctrie represents one of the fastest concurrent key-value data structure implementations today. However, its implementation requires a garbage collector to solve problems inherent in any atomics-based data structure – the ABA problem being one of them.

Chapter 3

Methods

This chapter presents the project's three thread-safe MultiMap designs: a single-lock based design which forms a baseline for later evaluation, a multi-lock design influenced by segment-based thread-safe hashmaps (see section 2.5.1), and a hash-array-mapped-trie design which departs from the Ctrie's atomics-based lock-free design to be more practical for the MultiMap's constraints. The chapter also presents a lockless thread-safe memory pool design to support the three thread-safe MultiMaps.

3.1 Design Considerations

To be practical, the presented thread-safe MultiMap design were produced under some constraints. Firstly, the interface of all thread-safe MultiMaps are identical and consistent with DBToaster's current MultiMap implementation. Secondly, DBToaster's current MultiMap implementation is used as a basis for comparison and implementation, so designs were chosen as to not stray too far from the current implementation. As such, all presented implementations maintain the composition of several hash indexes – as discussed in section 2.4.1 – as a core design feature.

Though these constraints make implementation more feasible, the use of multiple hash indexes means that an atomics-only MultiMap cannot be designed, as such a design would have to be capable of modifying entries across all hash indexes within a single atomic operation – something which by definition is impossible. This may be made possible by a future design which consolidates all hash indexes into a single cohesive structure, but this was not explored during the project.

Another important consideration is which MultiMap operations need to be thread-safe with respect to one another. For practical purposes only modifying operations need be thread-safe with respect to each other, as read-only operations may already be performed concurrently. There are very few use-cases for read-only operations that can be safely executed concurrently with modifying operations. This can be demonstrated by considering the result of an operation reading a value x while x is being concurrently modified by another operation – the result depends on the order of operations which is non-deterministic and thus essentially random. Such a design referred to as *phase*

concurrent and ensures determinism so long as the two types of operations – read-only and modifying (`add`, `addOrDelOnZero`, and `setOrDelOnZero`) – are not executed concurrently. However, the implementations later presented in chapter 4 are made flexible, and enable a trivial implementation path for allowing concurrency among all types of operations.

Finally, a state-of-the-art memory pool is outside the scope of this project, and a more sophisticated design and implementation than that presented here is left as future work.

3.2 Single-Lock MultiMap

The first and most simple of the three thread-safe MultiMap designs is the Single-Lock MultiMap, which utilises a single lock to allow exclusive access to the entire data structure for a single thread at a time. For any modifying operation on the data structure, the lock is acquired, held whilst the operation is completed, then released afterwards. Such a design takes influence from *Monitor*s – a concurrency abstraction which wraps data with a monitoring object which manages a lock for that data[30].

This design is incredibly simple, not requiring any changes to the original MultiMap structure presented in section 2.4.1, figure 2.1. As such, its operations on the underlying hash index structures are defined identically.

3.2.1 Discussion

The simplicity of the Single-Lock MultiMap design is a boon to implementation, as well as providing a good baseline for more sophisticated designs. However, its simplicity also means that no true concurrency is enabled and so it is expected that such a design would not perform well for locking modifying operations, where individual threads will be forced into operating on the data structure in a sequential order. Moreover, it is expected that the single lock associated with the data structure will be put under contention by any attempted concurrent operations, resulting in poorer performance as more threads attempt to operate on it. Because of this, the Single-Lock MultiMap design is expected to perform poorly for concurrent modifying operations with more than a single thread.

The design is ideal for application of concurrent read-only operations as phase-concurrency means that these operations do not require locking, and thus can be executed with true concurrency. The Single-Lock design does not involve any extra complexity over DBToaster's current MultiMap design for read-only operations, and thus it is expected that this design will be able to utilise the cache-friendliness and fast operations on the single bucket array structures of its hash indexes to demonstrate good performance for multiple threads applying read-only operations on the design.

3.3 Multi-Lock MultiMap

The second of the three thread-safe MultiMap designs is the Multi-Lock MultiMap, which implements a segmented hashmap design as discussed in section 2.5.1 for its

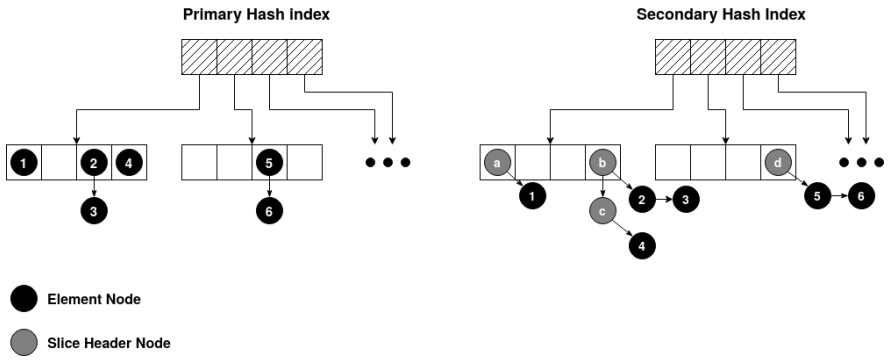


Figure 3.1: Example of Multi-Lock MultiMap primary and secondary index structure.

underlying hash indexes. Each hash index is split into several segments which may be independently locked and concurrently operated on as if they were hashmaps of their own. To map elements to segments a secondary hash function is to produce a secondary hash from the initial hash of the element. Using this secondary hash, the segment associated with the element is found and then locked whilst a modifying operation is completed on it. Because each segment acts as its own hashmap, the core hash index operations – `get`, `slice`, `insert`, and `erase` – are defined identically as for the DBToaster's current Multimap implementation, with the small difference that they are applied to individual segments instead of a single shared bucket array.

Figure 3.1 gives a simple example of the structure of the primary and secondary hash indexes of the Multi-Lock MultiMap for a small number of elements.

Unlike the Single-Lock MultiMap design which requires a single lock for all indexes, the Multi-Lock MultiMap design requires that all indexes have their own set of locks – one for each segment. This is due to separate indexes using different hash functions, and thus the segments that are locked at each hash index for a composite operation modifying multiple indexes – such as `addOrDelOnZero` – do not correlate directly with one another.

3.3.1 Discussion

The Multi-Lock MultiMap design's foremost benefit over the Single-Lock MultiMap is can support true concurrency for all operations. Its multiple hash index segments allow for modifying operations to be executed independently and in parallel with one another, so long as the set of locks that they need to acquire for an operation do not overlap. However, this also means that concurrency is limited by the number segments of each hash index, and the distribution of elements between them. In an ideal scenario it is possible that multiple threads operate on distinct sets of segments and never cause contention on their locks. However, this is not realistic, and it is extremely likely that concurrent modifying operations on the underlying hash indexes will sometimes collide.

Like any resizing bucket array design, the Multi-Lock MultiMap's hash indexes' segments will require occasional lengthily resizes whilst undergoing modifying operations, causing their associated locks to be held for a prolonged period of time, resulting

in a greater likelihood of contention on that lock as other threads perform modifying operations on segments with an assumed uniform distribution. So, despite true concurrency being afforded by the design, it is expected that the Multi-Lock MultiMap will perform better than the Single-Lock MultiMap designs but will not be able to sustain a high throughput of modifying operations whilst resizes are involved.

Like the Single-Lock design, it is expected that the Multi-Lock design will be able to sustain a high throughput of read-only operations which scales with a larger number of threads. However, the extra complexity of a secondary hash function does mean that it is not expected to perform as well as the Single-Lock MultiMap in this case.

A less obvious benefit of the Multi-Lock MultiMap is smaller bucket array growth increments. When a design, such as the Single-Lock MultiMap, utilises a single bucket array, resizing it means allocating twice its size in memory for the new bucket array. For a segmented design, memory usage is split between each segment, and although twice the memory of a single segment needs to be allocated for a resize, segments tend to be smaller and thus the memory overhead during a resize is much smaller than that of single bucket array designs. Thus the Multi-Lock MutliMap is also more suitable for systems with stricter memory constraints, though such a feature does also mean that more resizes are required for the same capacity as a Single-Lock design which may reduce performance.

3.4 HAMT MultiMap

The third and final thread-safe MultiMap design is the Hash-Array-Mapped-Trie (HAMT) MultiMap which departs from previous designs by implementing its hash indexes as tries – a design which is largely influenced by the Ctrie as presented in section 2.5.2. Like the Ctrie, the HAMT MultiMap implements its hash indexes as a hierarchy of bucket arrays which use portions of an elements hash to locate its corresponding bucket at each level. However, because the Ctrie is a design based on atomic instructions – decidedly infeasible for the MultiMap – locks are used at each individual bucket which must be acquired to modify to that bucket or its elements. Whilst not as granular as atomic instructions, a lock-per-bucket scheme guarantees that only modifying operations that explicitly need to access the same bucket will contend a lock.

Figure 3.2 gives a simple example of the structure of the primary and secondary indexes of the HAMT MultiMap for a small number of elements.

Each bucket within the HAMT MultiMap's hash indexes' bucket array hierarchy can either be empty, locked, or contain a pointer to a chain of elements or another bucket array. The lowest-level bucket corresponding to an element within the HAMT's hash indexes is used to store that element. All bucket arrays are of uniform size 2^N, and thus N bits from the hash are used to index into the bucket array at each level, starting from the lowest N bits for the root bucket array, and using the next N bits for each subsequent level. The first bucket encountered when traversing the trie structure using this method that does not reference another bucket array is the bucket corresponding the element whose hash was used to locate it.

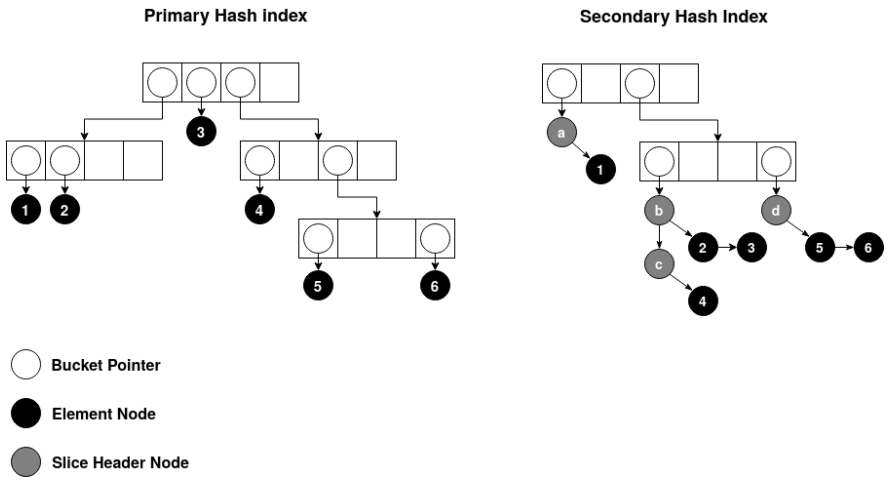


Figure 3.2: Example of HAMT MultiMap primary and secondary index structure.

Unlike the Single-Lock and Multi-Lock designs, operations on the HAMT MultiMap's hash indexes may cause their structure to change. On an `insert` operation, the trie structure is traversed using the elements hash, and then once its associated bucket is located, once of three things happen:

1. **Direct Insert (empty bucket)** – The element is inserted into the bucket.
2. **Displacement (nonempty bucket, maximum depth not reached)** – The element in the bucket is exchanged with a reference to a new bucket array which it is inserted into. The insert operation is then retried on the new bucket array.
3. **Chain Insert (nonempty bucket, maximum depth reached)** – The element is added to the chain of elements in the bucket.

In cases 1 and 3, the structure of the hash index is maintained, but in case 2 it is expanded with a new bucket array which further differentiates elements by more bits of their hash. These three cases on primary and secondary hash indexes are visualised by figures by appendices B and C, and are analogous with the Ctrie's ability to dynamically resize its bucket array branches. As hashes have a finite number of bits, the HAMT MultiMap's hash indexes are of finite depth, and thus as the maximum depth elements are chained rather than being given their own bucket.

Unlike the `insert` operation, the `erase` operation of the HAMT MultiMap's hash indexes does not shrink the underlying trie structure, instead leaving empty bucket arrays in-place. This choice is made to simplify implementation, but could be added in a more sophisticated implementation. An example of mixed `insert` and `erase` operations on a primary index is presented in appendix D.

`get` and `slice` operations can be implemented trivially by first traversing the trie structure using the desired element's hash, then accessing the associated bucket to find the desired element or slice of elements.

3.4.1 Discussion

The core advantage of the HAMT MultiMap's design over the Single-Lock and Multi-Lock designs is its more granular locking scheme along with avoiding lengthily resize operations. With a more granular locking scheme it is less likely that arbitrary operations on the data structure will result in the same lock being contested, and thus it is more likely that operations will not conflict with one another. Dynamic resizing of the underlying trie structure as elements are inserted also has the positive side-effect of differentiating elements with similar hashes, and thus should reduce lock contention as more elements are added to the hash indexes.

Being able to avoid resize operations is a boon to performance as previously discussed in section 2.5, and static bucket array sizes mean that bucket arrays may even be pooled for more efficient allocation, in a similar manner to the original Ctrie design. This feature serves to offset the effect of allocations and indirection in traversing the data structure. Despite operations being more complex than the hashmap-based designs, it is expected that optimisations such as bucket array pooling, dynamic resizing, and the lack of resizes will allow the HAMT MultiMap to be extremely competitive with the Single-Lock and Multi-Lock designs.

Due to the above reasons, it is expected that the HAMT MultiMap will provide consistent improvements in throughput of both modifying and read-only operations. Despite the extra complexity involved, it is expected that it will be competitive with DBToaster's current MutliMap implementation, as well as the Single-Lock and Multi-Lock MultiMaps. However, it is difficult to judge how the extra complexity of the HAMT MultiMap's operations will affect it for small thread counts, even with the above optimisations. Read-only operations are also expected to be a problem for the design as the extra complexity overheads incurred in allowing for fast fine-grained concurrency of modifying operations does not explicitly improve the performance of read-only operations.

3.5 Thread-Safe Memory Pool

The memory pool used to support the three thread-safe MultiMaps is designed to add as little runtime overhead as possible in order to better evaluate the three designs' performance, and to avoid any performance characteristics or limitations imposed by a more restrictive design. The design features a separate sub-memory-pool per thread within the memory pool itself, allowing each thread to manage their own chunks and slots identically to DBToaster's current memory pool implementation (see section 2.4.2) with the exception that a free-list is not used as managing one would introduce a point of contention between threads. Instead, an additional operation – `releaseAll` – is added to reclaim all previously acquired and/or released slots.

In addition, a core goal of the project is allowing for parallel iteration over MultiMap elements, and this is implemented via a `foreach` operation on the memory pools utilised to allocate MultiMap elements. The utilisation of a separate sub-memory-pool per thread gives a natural division of work, wherein threads are given the responsibility of iterating over elements that they have allocated. This does mean, however, that

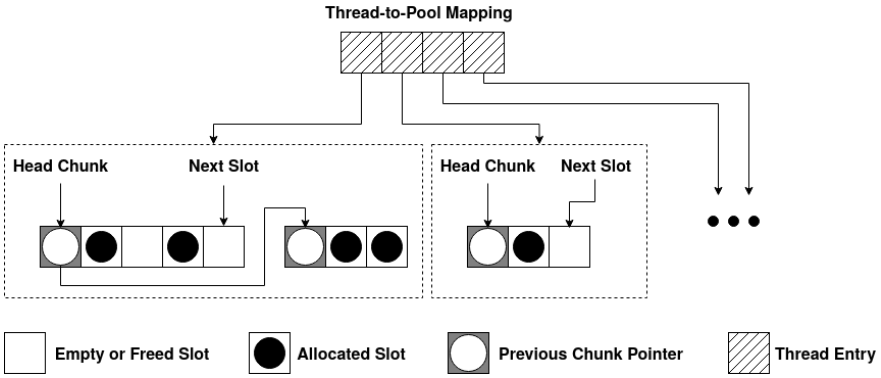


Figure 3.3: Thread safe memory pool structure.

pools cannot be shared among MultiMaps as is currently implemented in DBToaster.

To support `releaseAll` and `foreach` operations on the memory pools, a flag is associated with each slot to indicate if a slot is in use. During the execution of a `releaseAll` or `foreach` operation the flag may be utilised to correctly deconstruct and iterate across only allocated elements respectively.

Finally, because not all memory pools within DBToaster need to deconstruct allocated objects (because they are trivially destructible), nor need to iterate over them, the flag used to indicate used slots is not always needed. For these cases an alternative design which removes these flags is used.

The structure of the thread-safe memory pool is presented by figure 3.3.

3.5.1 Discussion

The implementation of a performant, practical thread-safe memory pool is not a core goal of this project, but despite this the presented memory pool design allows for threads to manage their own memory such that they are never blocked by other threads whilst maintaining the benefits of the exponential block allocation scheme used by DBToaster's current memory pool implementation. The core feature missing from this design is the free-list which allows for slots to be reused after they are released. This may be solved by implementing an efficient thread-safe multi-producer multi-consumer queue to hold references to free slots. Many state-of-the-art thread-safe queue designs exist already[31], and could be explored in a future extension to the project.

Though giving each thread its own memory to manage an allocate will increase the overall number of allocations, it is expected that this effect will be offset by allocations being performed in parallel but a future design may consider centralising allocations for greater efficiency.

Chapter 4

Implementation

The performance of a data structure is not just determined by its design, but also its implementation. This section covers significant details of the implementation which enable core capabilities of the three presented thread-safe MultiMap designs, improve their performance, or demonstrate the use of relevant technologies in the project.

All three thread-safe MultiMaps – and the supporting thread pool – are implemented using C++, OpenMP, and, in some cases, GCC's `__atomic` compiler builtin.

4.1 Transaction Objects

In order to implement operations on the MultiMap modifying both its primary and secondary hash indexes, a method of ensuring exclusive access to its associated data throughout the operation is needed. For each MultiMap designs, chapter 3 presented the basic way in which locks are utilised for operations on the variety of MultiMap designs, but not how they are implemented in practice. For each thread-safe MultiMap implementation, a *transaction object* is implemented, which utilises a design's locks to block conflicting operations. Using the C++ RAII idiom[32], these objects acquire the lock relevant to ensuring a given operation's atomicity then, once out of scope, releases it – allowing another transaction object to acquire the lock for its own operation.

In the implementation of all three thread-safe MultiMaps, transaction objects are associated with the subject element of a MultiMap operation, and takes ownership of the relevant lock in the MultiMap's primary hash index. By requiring that a transaction object is created at the beginning of each locking operation, it is ensured that operations on the same data are performed safely, as they will necessarily need to acquire the same lock beforehand. Instead of a transaction object, secondary hash indexes implement locking within their individual operations, as taking the relevant lock in the primary hash index means that other concurrent operations will never modify the element associated with the transaction object.

As a further optimisation, transaction objects implement `get`, `insert`, and `erase` operations on the primary hash index they correspond to, for the element they are associated with. This allows for an arbitrary number of operations to be performed safely on

the primary index whilst the transaction object is in scope. This API allows for some optimisations to operations as discussed in subsequent sections.

Finally, this abstraction is important as it allows for a uniform implementation of MultiMap operations among all thread-safe MultiMap designs. Modified pseudocode demonstrating this is presented under appendix E, and may be contrasted with that in appendix A.

4.1.1 Single-Lock Transaction Object

The transaction object for the Single-Lock MultiMap implementation acquires the single lock associated with the MultiMap and then releases it again when it goes out of scope. As previously discussed (see section 3.2), this means blocking all other operations on the MultiMap whilst the object remains in scope. As a consequence of this design decision, the single lock of the Single-Lock MultiMaps design is placed within its primary hash index. This does not affect the functionality of the implementation.

4.1.2 Multi-Lock Transaction Object

The transaction object for the Multi-Lock MultiMap implementation determines which segment within the primary hash index its associated element corresponds to using its secondary hash function, and acquires the lock associated with its lock for its lifetime. The transaction object also tracks the segment its associated element correspond to, allowing subsequent operations using the transaction object's API to avoid applying the design's secondary hash function to the element for every operation, thus avoiding some associated overheads in doing so.

4.1.3 HAMT Transaction Object

The transaction object for the HAMT MultiMap implementation traversed the primary hash index's underlying trie structure to find the lowest-level bucket associated with the given element, and then locks that bucket only for the lifetime of the object. The transaction object tracks the bucket it has locked, allowing operations applied through its API to be applied directly rather than traversing the trie every time. As previously mentioned, `insert` operations applied to the HAMT's hash indexes may result in a new bucket array being inserted. If this occurs by using the `insert` operation through the transaction object's API then once the resulting new bucket array is inserted, the bucket corresponding to the transaction's associated object is locked and the previous lock is released, allowing other operations to be applied on the new bucket array concurrently.

4.2 Locks

For the implementation of the three MultiMap designs, two different types of locks were used – OpenMP provided lock objects, and a custom spinlock. Both the Single-Lock and Multi-Lock MultiMap implementations utilise OpenMP locks as periods of

high contention on them is expected due to bucket array resizes, thus making a mutex-based implementation preferable over a spinlock. For the HAMT MultiMap, a custom spinlock implementation is used which utilises the pointer used by each bucket to point to elements or other bucket arrays. A spinlock implementation was selected for this purpose as the more granular locking scheme used by the HAMT MultiMap means that contention should be less common, and thus a lightweight lock implementation which utilises pointers that are already present is optimal. Using bucket pointers as the basis for the spinlock does mean that the pointers need to support atomic operations on them, which is discussed in greater detail in section 4.3.

4.3 HAMT Pointer

A central design feature of the HAMT MultiMap is its use of bucket pointers to selectively point to nothing, a chain of element nodes, or another bucket array. As discussed in section 4.2, another role of the bucket pointer is to support the implementation of a spinlock for its associated bucket. To determine what a pointer is being used for, additional information is required to indicate the type of what is being pointed to, and whether the bucket is locked.

The solution to such a problem is a tagged union – a tag-value pair where the tag determines the type of the associated value. Because bucket pointers need to support a spinlock implementation, the value of the pointer and its tag need to be able to be read or modified in one atomic operation, else either the tag or pointer could be modified in between reads of the two, even if these individual reads or writes are atomic. To solve this problem, the implementation of this pointer tagged-union utilises the lower two bits of bucket pointers to store the tag, allowing for both to be read or written with a single atomic instruction. This solution takes inspiration from LLVM's PointerUnion type, which also utilises lower bits to store type information, but does not support atomic operations[33]. This implementation is made possible by the fact that all data pointer to within the implementation is aligned to at least four bytes by default, leaving the two least-significant bits free to use.

To support storage of type information in the lower bits, a special `PointerSum` class was implemented. The class not only supports extracting the pointer and tag separately, but also atomic load, store, swap, and compare-and-swap operations on the underlying pointer-tag pair. This final addition enables the use of the pointer-tag pair as a spinlock where the tag is used to store a value to indicate the pointer is locked, which may be set atomically, and later replaced atomically with the result of an operation on the bucket.

The full C++ class implementation can be found under appendix F.

4.4 Parallel Iteration

Parallel iteration over MultiMap elements, as per the memory pool design presented in section 3.5, is implemented via the memory pool using slot-associated flags with work divided by way of a thread's own allocations.

This is enabled using OpenMP's `parallel` construct which executes a block of code once for every available thread. In the implementation of the memory pool's `foreach` method, the `omp_get_thread_num()` OpenMP function is used within the parallelised code block to index into each thread's sub-memory-pool to access their allocations. An abbreviated implementation of this can be seen below:

```
1 // Execute the code within the block with each thread independently
2 #pragma omp parallel
3 {
4   // Get a thread's sub-memory-pool entry
5   ThreadEntry& thread_entry = thread_entries[omp_get_thread_num()];
6
7   // Iterate over each allocated slot in the thread's chunk list
8   ...
9 }
```

4.5 Micro Optimisations

Though each individual MultiMap design has its own set of performance benefits and drawbacks as a result of its structure and design-specific overheads, there are some optimisations which were applied universally to improve the performance of all implementations.

Firstly, alignment was applied in places within all implementations to reduce the effect of false sharing, mainly in the form of aligning data to independent cache lines. This technique was applied to the entries for each segment in the Multi-Lock MultiMap to prevent modifications to other segment entries to cause cache invalidations of others, and in the memory pool implementation to do the same for each thread's sub-memory-pool entry.

Secondly, atomic instructions targeting the same data across multiple threads were eliminated where possible. The most important application of this was to remove global counts of elements throughout the MultiMap implementations and replacing them with thread-local counts or, in the case of the Multi-Lock implementations, only keeping segment-local counts. Atomic operations on the same data are, by definition, serialised across a system, and prevent thread-safe data structures from scaling their operation throughput with increasing numbers of threads.

Chapter 5

Evaluation

This chapter presents the results of benchmarking the three thread-safe MultiMaps presented in previous chapters for a range of configurations on a selection of two distinct microarchitectures, and discusses the observed outcomes, their alignment with expectations, and how they influence future work in the area of thread-safe MultiMaps.

5.1 Approach

For the purposes of evaluating the three thread-safe MultiMap implementations, a microbenchmark suite was implemented, which enables repeated runs of benchmarks targeting individual MultiMap operations. For all benchmark results presented, 1000 runs of the chosen benchmark were performed, then the mean runtime was taken and used to compute a number of operations per second. Before each set of 1000 runs, 10 runs are executed to *warm-up* the cores and caches to give more consistent results. OpenMP's `parallel for` construct was used to implement parallelisation for each benchmark, and an example of this can be seen below:

```
1 #pragma omp parallel for schedule(runtime)
2 for (int i = 0; i < N; ++i) {
3   // Benchmark code
4 }
```

Elements used for each benchmark were constructed using the loop index variable, which was also used as the hash for each element. This gives a uniform distribution of hashes, representing the ideal case for the thread-safe MultiMaps. For benchmarks requiring other operations – such as those to `clear` – these operations are performed outside of measurements.

5.1.1 Configurations

For each benchmark, several configurations are considered consisting of two element types of varying complexity, two MultiMap setups, and two microarchitectures.

The two element types considered are a *trivial* type consisting of a single integer, and a *complex* type consisting of both an integer and a C++ standard library string. The trivial type represents elements that take little work to copy, construct, and destruct, whilst the complex type represents elements that require extra work such as allocations to copy, construct, and destruct.

The two MultiMap configurations considered are a simple setup where MultiMaps are composed of only a primary hash index, and a setup where MultiMaps consist of both a primary and secondary hash index. The condition chosen for the secondary hash index ensures all slices are of length 4 so as to induce iteration over slices to locate elements within the secondary hash index. The two setups represent realistic MultiMaps seen in DBToaster-generated code for general-purpose benchmarks such as the TPC-H benchmark suite[34].

The two microarchitectures considered for evaluation are the AMD Ryzen 5-2500U with 4 cores and 8 threads[35], and the Intel i5-6500 with 4 cores and 4 threads[36], of which both run similar Linux distributions.

The Ryzen 5-2500U implements 2-way SMT which, as previously discussed in section 2.1, means that pairs of threads share the same cache and memory slots. The Ryzen 5-2500U also implements a chiplet design, which means that cores are all an approximately uniform distance from memory and thus the performance of cores with respect to memory should be uniform.

The i5-6500 does not implement SMT, and implements a ring design rather than chiplet one. This means that cores are a non-uniform distance from memory, and thus performance with respect to memory varies depending on the core. Ring designs perform well for single-threaded workloads on the core closest to memory, but multi-threaded workloads may not scale as they would otherwise be expected to.

5.1.2 Standardisation

For all benchmarks, environment variables and other settings are chosen to give best performance and stable measurements. For all thread-safe MultiMap benchmarks, OpenMP environment arguments are set to:

- `OMP_DYNAMIC='FALSE'` – Prevents the OpenMP runtime from dynamically choosing the number of threads in use.
- `OMP_SCHEDULE='STATIC'` – Sets the OpenMP schedule to static, meaning that parallel loop iterations are shared evenly among threads.
- `OMP_PROC_BIND='CLOSE'` – Groups threads close together on hardware to reduce communication latency between cores.

For compiling the benchmark suite, the following GCC compiler options were used:

- `-DNDEBUG` – Removes uses of `assert(...)` from the executable.
- `-O3` – Enables aggressive optimisation.
- `-flto` – Enables link-time optimisation.

- `-march=native` – Specifies the native architecture as the target.
- `-fopenmp` – Enables OpenMP.

For each set of benchmarks, the niceness of each benchmark process was set to the lowest value possible to reduce disruption by other processes run on the same system. This makes benchmarks more reproducible and stable, but it should be acknowledged that the systems being used to benchmark are running desktop environments and common Linux daemons which may cause small disruptions to measurements.

The MultiMap designs presented earlier require some parameters for their implementation and evaluation. For the Multi-Lock MultiMap, 16 segments are used per hash index so as to ensure that a restrictive number of segments is not a factor in its performance. For the HAMT MultiMap, bucket arrays consist of 16 buckets, and thus use 4 bits of hash per trie layer, up to a maximum depth of 16 layers. The hash function used by the Multi-Lock MultiMap for producing secondary hashes is the lowest-bias hash function found at time of writing by the *Hash Function Prospector*[37] – a tool used for discovering simple but performant integer hash functions. This hash function is shown in appendix G.

5.1.3 Choice of Benchmarks

Though the three thread-safe MultiMaps presented in this report support a number of operations, evaluating them for all configurations and situations would take far too much time. So instead, only results for a selection of operations, for all configurations and situations, are presented and discussed.

In order to represent the thread-safe MultiMaps' performance with respect to modifying operations, `addOrDelOnZero` alone is used for the evaluation, as its implementation encapsulates the functionality of `add`, and is extremely similar to `setOrDelOnZero`. With `addOrDelOnZero`, the insertion of new elements, removal of elements, and reinserting of elements can be evaluated.

For read-only operations, the `get` operation's throughput is evaluated, as well as the use of `foreach` for iterating over a MultiMap's elements.

5.2 Results

This section presents and discusses the results of the benchmarks performed on the three thread-safe MultiMap implementations for the chosen configurations, as well as for some additional scenarios that are of interest for practical application.

5.2.1 `get` – Element Lookups

The first case for evaluation is the throughput of `get` operations on the three thread-safe MultiMap implementations. As previously discussed in section 3.1, the `get` operation is non-locking, so the complexity of each data structure is the main source of overhead.

For each benchmark, 100,000 distinct elements were inserted into a MultiMap, then `get` was used to lookup each element exactly once.

Figures 5.1 and 5.2 present a comparison of throughputs for the `get` operation for all three thread-safe MultiMap implementations and DBToaster's MultiMap implementation for trivial and complex elements. Figure 5.1 presents results for the Ryzen 5-2500U, and figure 5.2 presents results for the i5-6500. A total of 100,000 `get` operations are executed per benchmark run to target every element within the target MultiMaps exactly once.

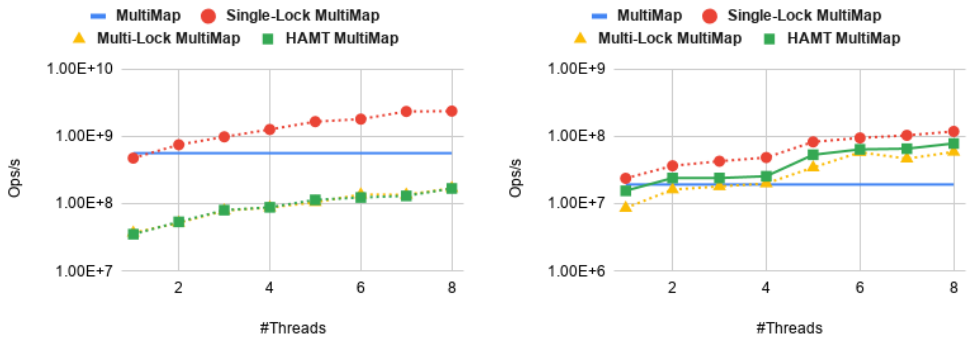


Figure 5.1: `get` operations per second for 1–8 threads on the AMD Ryzen 5-2500U. Blue line is throughput for DBToaster's current MultiMap implementation with a single thread. Left plot: trivial elements. Right plot: complex elements.

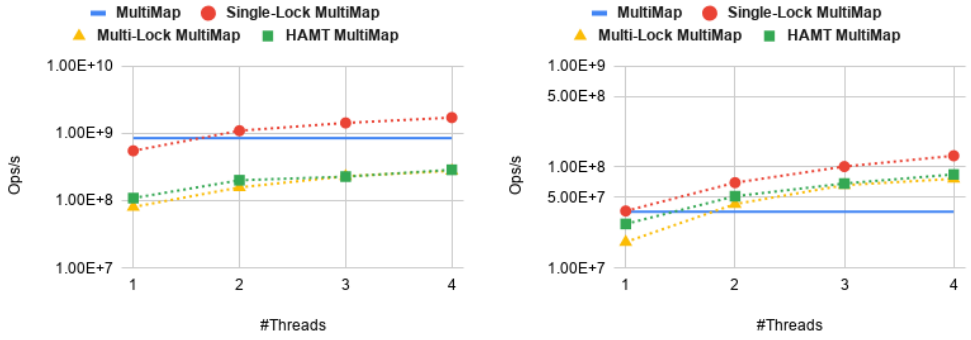


Figure 5.2: `get` operations per second for 1–4 threads on the Intel i5-6500. Blue line is throughput for DBToaster's current MultiMap implementation with a single thread. Left plot: trivial elements. Right plot: complex elements.

For all configurations of elements, MultiMaps, microarchitectures and threads, the Single-Lock MultiMap provides the greatest throughput. Even when compared to the throughput provided by DBToaster's current MultiMap implementation, it performs identically at worst. This was expected as the non-locking nature of `get` means that the Single-Lock MultiMap's implementation of it is identical to that of DBToaster's current MultiMap implementation. Moreover, its simple design allows it to outperform the more complex Multi-Lock and HAMT MultiMaps which incur some overhead in their operations to enable true concurrency of modifying operations.

Whilst the Multi-Lock and HAMT MultiMap implementations appear to perform poorly when compared to the Single-Lock MultiMap, they do still succeed in enabling greater throughput than DBToaster's current MultiMap implementation in the case of complex elements. The increased work involved with complex elements means that performance can be gained by parallelising operations involving them, offsetting the complexities of the MultiMap designs themselves. However, the complexity of the two designs is clearly too much to achieve the same for trivial elements.

5.2.2 `addOrDelOnZero` – Inserting New Elements with Resizes

The second case for evaluation is the use of `addOrDelOnZero` operations to populate an empty MultiMap with 100,000 distinct elements. With each run of the benchmark, a new MultiMap instance is created, thus requiring that the Single-Lock and Multi-Lock MultiMap implementations resize their underlying bucket arrays as items are inserted.

Figures 5.3 and 5.4 presents a comparison of throughputs for the `addOrDelOnZero` operation for all three thread-safe MultiMap implementations where a new element is inserted with each operation, and resizes of underlying bucket arrays are present. Results cover configurations with trivial and complex element types, and MultiMaps with only a primary index and MultiMaps with both a primary and secondary index. Figure 5.3 presents results for the AMD Ryzen 5-2500U, and figure 5.4 presents results for the Intel I5-6500. As with the `get` benchmarks, this set of benchmarks consists of 100,000 inserts per run.

For all but one configuration of elements, MultiMaps, and microarchitectures, the HAMT MultiMap demonstrates the greatest throughput of all MultiMaps when allocated a sufficient number of threads. Over most configurations it demonstrates scaling of throughput with threads – something that both the Single-Lock and Multi-Lock MultiMaps lack. Despite the good performance overall, there are still cases where the design does not scale as well as expected – particularly for complex elements on the Ryzen 5-2500U – and where it does not achieve a better throughput than DBToaster's current MultiMap implementation.

As expected, the Single-Lock design performs extremely poorly in all cases. Its throughput scales negatively as more threads are allocated to it, due to the increased contention around its single lock, and the occasional cache invalidations as more threads access the same structure. Overall, the Single-Lock MultiMap performs best with a single thread, though on the Ryzen 5-2500U it demonstrates peak throughput with two threads for trivial elements. This is due to SMT, which allows two threads to share the same cache and memory slots and thus not experience the performance regressions observed for greater thread counts. As complex elements require more interactions with memory due to their use of C++ standard library strings, SMT does not aid performance.

The Multi-Lock MultiMap's performance is generally consistent for all thread counts, neither improving nor regressing. Like the Single-Lock MultiMap, the Multi-Lock MultiMap's peak performance is observed with two threads on the Ryzen 5-2500U for trivial elements, for similar reasons.

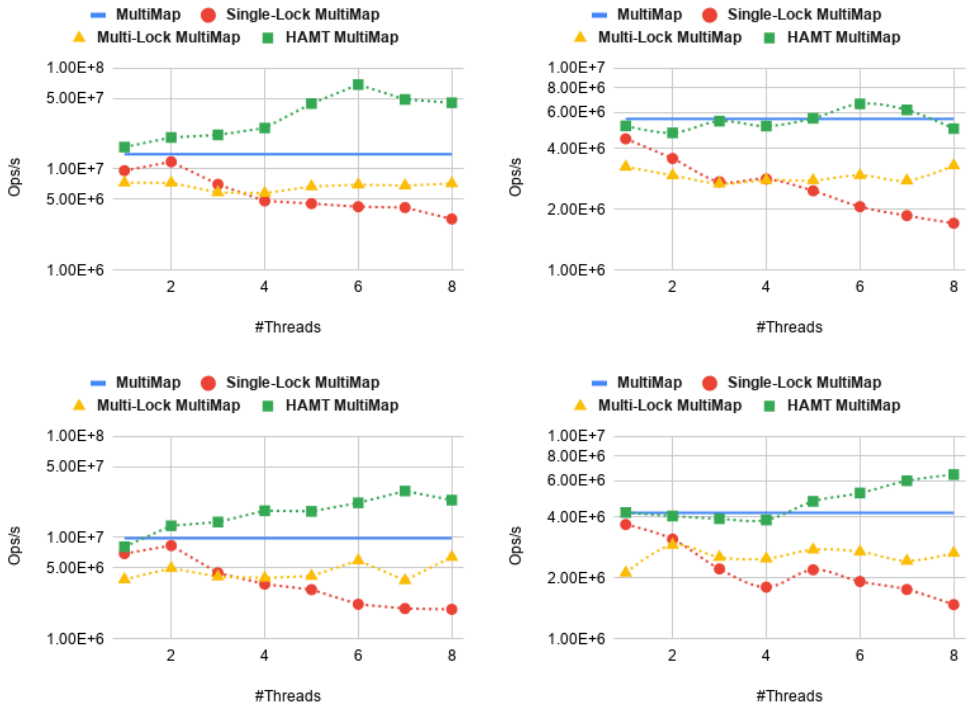


Figure 5.3: Inserting `addOrDelOnZero` operations per second for 1–8 threads on the AMD Ryzen 5-2500U with resizes. Blue line is throughput for DBToaster's current MultiMap implementation with a single thread. Left plots: trivial elements. Right plots: complex elements. Top plots: Primary index only. Bottom plots: Primary and secondary indexes.

5.2.3 `addOrDelOnZero` – Inserting New Elements without Resizes

The third case for evaluation is the use of `addOrDelOnZero` operations to populate an empty MultiMap with 100,000 distinct elements, except that resizes are not performed. The set of benchmarks utilise the microbenchmark suite's warm-up runs to resize underlying bucket arrays, which are cleared between runs and reused. This is a common pattern in DBToaster-generated code for relations which are used to store intermediate results and then cleared afterwards. Note that reusing MultiMaps between runs also means reusing memory allocated by memory pools, thus the HAMT will still benefit from this change.

Figures 5.5 and 5.6 present a comparison of throughputs for the `addOrDelOnZero` operation for all three thread-safe MultiMap implementations where a new element is inserted with each operation, and resizes of underlying bucket arrays are not present. Results cover configurations with trivial and complex element types, MultiMaps with only a primary index, and MultiMaps with both a primary and secondary index. Figure 5.5 presents results for the AMD Ryzen 5-2500U, and 5.6 presents results for the Intel I5-6500.

With resizes now discounted, the Multi-Lock MultiMap now demonstrates the greatest

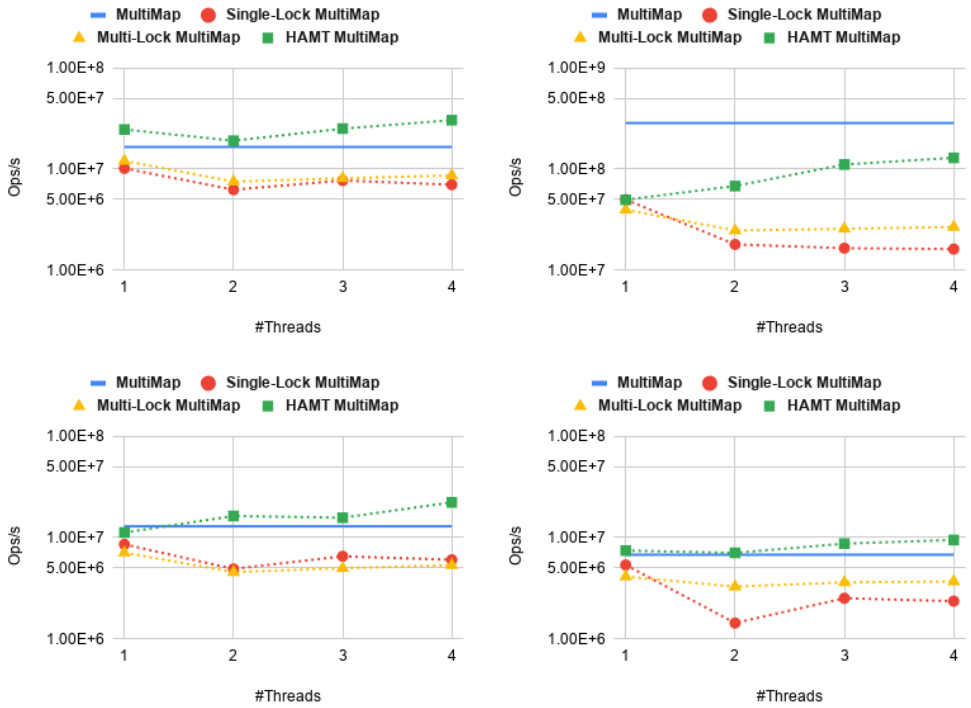


Figure 5.4: Inserting `addOrDelOnZero` operations per second for 1–4 threads on the Intel i5-6500 with resizes. Blue line is throughput for DBToaster's current MultiMap implementation with a single thread. Left plots: trivial elements. Right plots: complex elements. Top plots: Primary index only. Bottom plots: Primary and secondary indexes.

throughput for almost all configurations of elements, MultiMaps, and microarchitectures. In cases involving trivial elements, the HAMT MultiMap demonstrates similar or better performance than the Multi-Lock MultiMap, but overall the Multi-Lock MultiMap clearly produces the best results.

The HAMT MultiMap demonstrates scaling similar to how it did before, but this time the margin between it and DBToaster's current MultiMap implementation is much smaller, and in some cases it fails to produce greater throughput, particularly for cases involving trivial elements. Whilst removing resizes improved the performance of the Multi-Lock MultiMap's throughput greatly, it did not have such an effect on the HAMT MultiMap. Thus, despite enabling throughputs beyond those produced by resizing MultiMaps, the throughput of the design does not remain as competitive once resizes are discounted.

As expected, the Single-Lock MultiMap still performs poorly for more than a single thread, as concurrent operations are not enabled by removing resizes and thus the same problems as in the previous benchmark remain. Similarly, the same effects of SMT can be observed for cases involving trivial elements on the Ryzen 5-2500U.

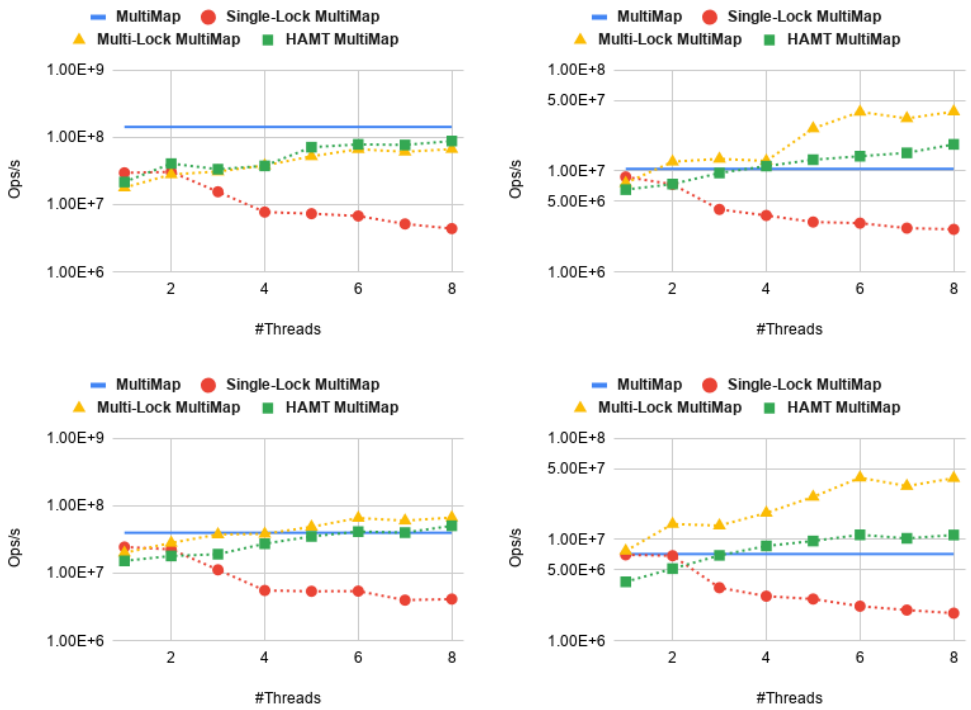


Figure 5.5: Inserting `addOrDelOnZero` operations per second for 1–8 threads on the AMD Ryzen 5-2500U without resizes. Blue line is throughput for DBToaster's current MultiMap implementation with a single thread. Left plots: trivial elements. Right plots: complex elements. Top plots: Primary index only. Bottom plots: Primary and secondary indexes.

5.2.4 `addOrDelOnZero` – Reinserting Elements

The fourth case for evaluation is the reinserting of 100,000 distinct elements already present within a MultiMap. Unlike inserting new elements into a MultiMap, this only requires incrementing the element's associated count, and thus means that overheads incurred by more complex designs are expected to result in particularly poor performance. Before each benchmark is executed, a MultiMap is prepared by inserting 100,000 distinct elements into it, and is then used for all subsequent runs.

Figure 5.7 presents a comparison of throughputs for the `addOrDelOnZero` operation for all three thread-safe MultiMap implementations where every element inserted is already present. Results cover configurations for trivial elements and MultiMaps with only a primary index, as the secondary index is not accessed for such operations. Results for complex elements can be found under appendix H and are omitted here because of their similarity.

As expected, all three thread-safe MultiMap implementations perform poorly when compared to DBToaster's MultiMap implementation. Despite the Multi-Lock and HAMT MultiMaps scaling their throughput as more threads are put to use, they never overtake the DBToaster MultiMap implementation in throughput. In fact, in both presented cases the Single-Lock MultiMap's throughput with a single thread is similar to peak throughput achieved by either the Multi-Lock or HAMT MultiMap.

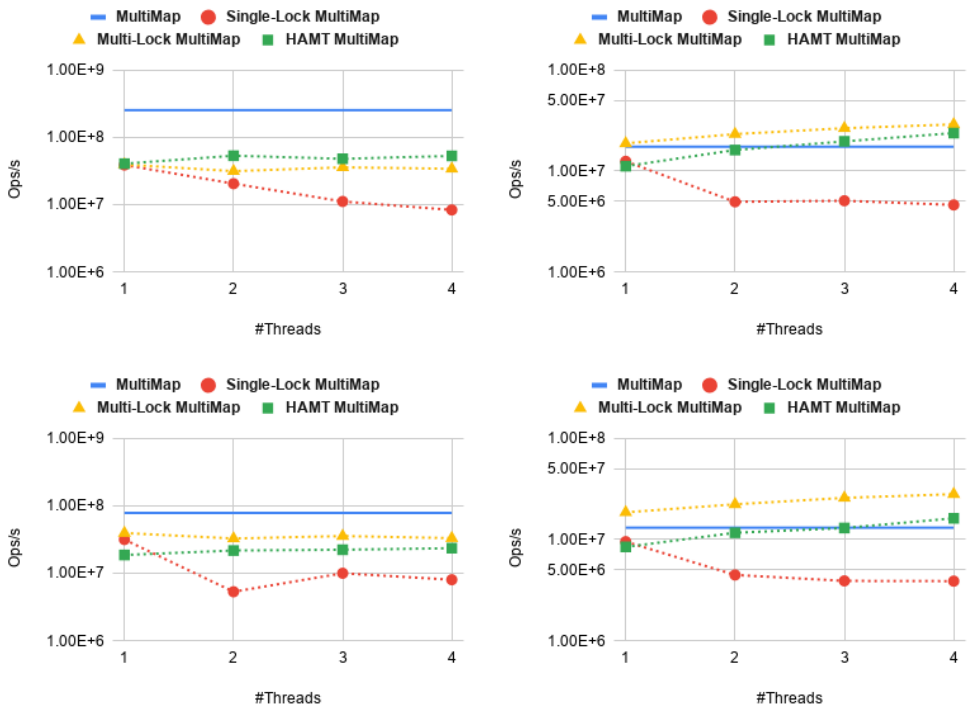


Figure 5.6: Inserting `addOrDelOnZero` operations per second for 1–4 threads on the Intel i5-6500 without resizes. Blue line is throughput for DBToaster's current MultiMap implementation with a single thread. Left plots: trivial elements. Right plots: complex elements. Top plots: Primary index only. Bottom plots: Primary and secondary indexes.

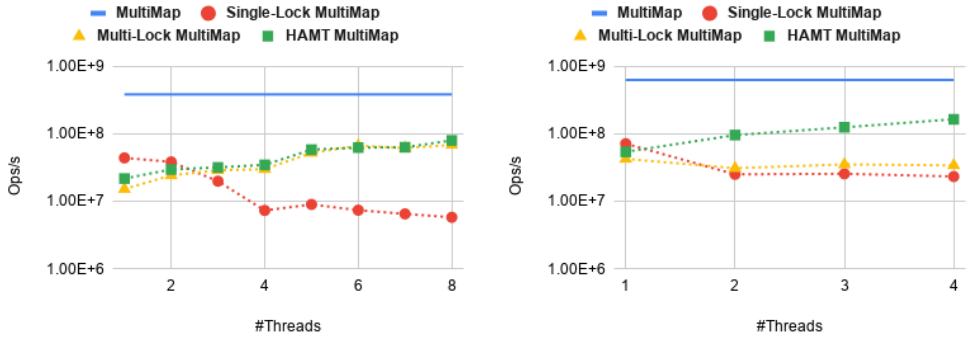


Figure 5.7: Reinserting `addOrDelOnZero` operations per second for 1–8 and 1–4 threads with complex elements. Blue line is throughput for DBToaster's current MultiMap with a single thread. Left plot: AMD Ryzen 5-2500U. Right plot: Intel i5-6500.

For the first time, a significant performance difference is observed between the two microarchitectures. In the case of the Ryzen 5-2500U, the Multi-Lock MultiMap and HAMT perform near-identically for all thread counts. However, for the i5-6500, the throughput of the Multi-Lock MultiMap at all thread counts is significantly lower. This may be due to SMT allowing threads to more efficiently utilise the capabilities of each core, but the exact reason is unclear.

5.2.5 `addOrDelOnZero` – Removing Elements

The fifth case for evaluation is the use of `addOrDelOnZero` operations to remove each element from a MultiMap containing 100,000 elements. Such operations require less work than an inserting `addOrDelOnZero` operation as no memory must be allocated. As a result, it is expected that MultiMaps will exhibit throughputs somewhere between those observed for inserting `addOrDelOnZero` and reinserting `addOrDelOnZero` operations. Before each run of each benchmark, the target MultiMap is prepared by inserting 100,000 distinct elements which are entirely removed by the subsequent run.

Figures 5.8 and 5.9 present a comparison of throughputs for the `addOrDelOnZero` operation for all three thread-safe MultiMap implementations where all elements erased are distinct and present in the MultiMap. Results cover configurations for trivial and complex elements, MultiMaps with only a primary index, and MultiMaps with both a primary and secondary index. Figure 5.8 presents results for the AMD Ryzen 5-2500U, and figure 5.9 presents results for the Intel I5-6500.

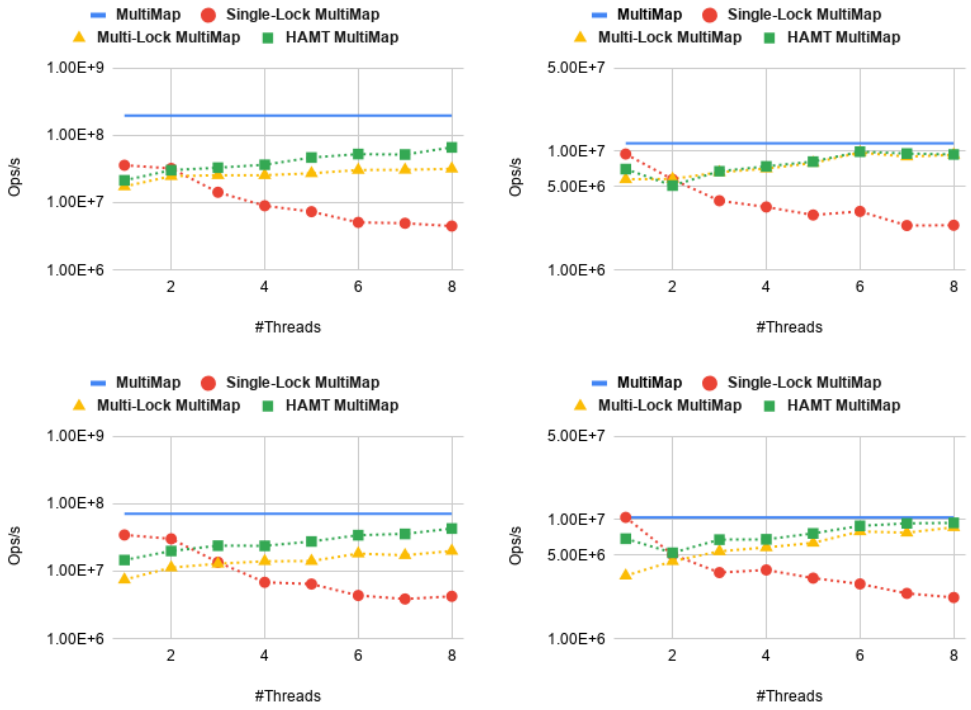


Figure 5.8: Erasing `addOrDelOnZero` operations per second for 1–8 threads on the AMD Ryzen 5-2500U. Blue line is throughput for DBToaster's current MultiMap implementation with a single thread. Left plots: trivial elements. Right plots: complex elements. Top plots: Primary index only. Bottom plots: Primary and secondary indexes.

As expected, all three MultiMap implementations exhibit throughputs similar to that of reinserting `addOrDelOnZero` operations. Like benchmarks prior, both the Multi-Lock and HAMT MultiMaps scale their throughput with increasing numbers of threads, but fail to demonstrate significant improvements beyond DBToaster's MultiMap implementation's throughput. This is true with the exception of the complex-element,

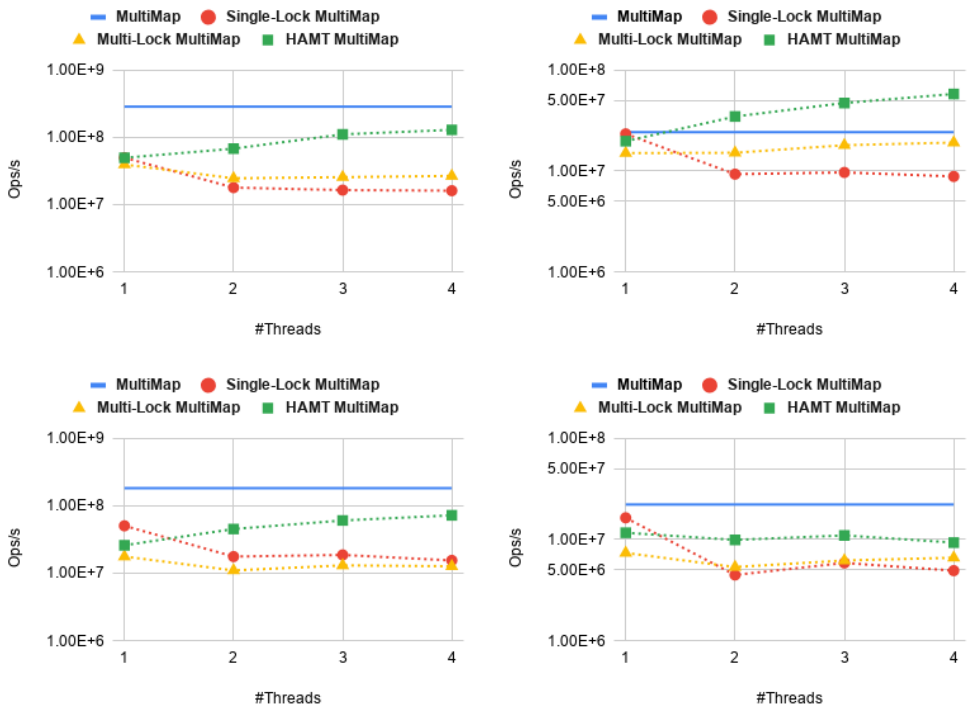


Figure 5.9: Erasing `addOrDelOnZero` operations per second for 1–4 threads on the Intel i5-6500. Blue line is throughput for DBToaster's current MultiMap implementation with a single thread. Left plots: trivial elements. Right plots: complex elements. Top plots: Primary index only. Bottom plots: Primary and secondary indexes.

primary-index-only case on the i5-6500, where the HAMT MultiMap produces significantly more throughput than the DBToaster MultiMap.

Once again, a difference in microarchitectures is observed, as the Multi-Lock MultiMap and HAMT MultiMaps have almost identical performance on the Ryzen 5-2500U, whilst on the i5-6500 their performance is far more distinct. This is likely for the same reasons as the previous benchmark.

As before, the Single-Lock MultiMap performs poorly on this benchmark, achieving best throughput for a single thread, which is sometimes the greatest throughput amongst thread-safe MultiMaps.

5.2.6 `foreach` – Aggregation

The fifth case for evaluation is the calculation of the sum of a MultiMap's trivial elements' integers using the `foreach` operation. because `foreach` is implemented for all thread-safe MultiMaps via the memory pool used to allocate elements, the performance of parallel iteration across a MultiMap's elements is independent of their design. For the serial `foreach` implemented by DBToaster, the benchmark is carried out by accumulating the sum in a single variable, whilst for the parallel `foreach`, separate thread-local variables are used to compute subtotals, which are then summed together to give a total. This benchmark is representative of aggregate computation.

Figure 5.10 presents a comparison of throughputs for addition operations carried out on elements of a MultiMap via `foreach` for serial and parallel iteration. In both cases 100,000 elements are inserted into a MultiMap then the its element's associated integer fields are summed. Elements are uniformly distributed between threads in the case of parallel iteration.

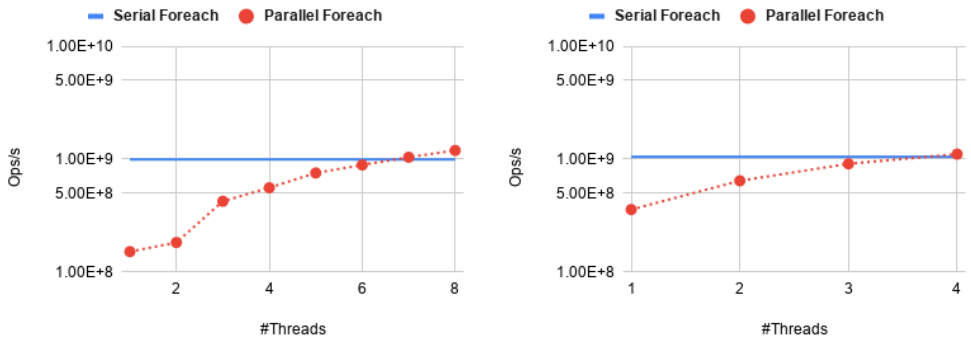


Figure 5.10: Addition operations per second for 1–8 and 1–4 threads. Blue line is throughput for DBToaster's current MultiMap with a single thread. Left plot: AMD Ryzen 5-2500U. Right plot: Intel i5-6500.

For both microarchitectures, parallel iteration shows a slight improvement in aggregate computation over serial iteration. This indicates that parallel iteration can at least be competitive with serial iteration for simple aggregates which extenuate the inherent overheads of parallel computation.

5.2.7 `foreach` – Inserting New Elements without Resizes

The sixth and final case for evaluation is a combination of `foreach` and `addOrDelOnZero` to insert elements from one MultiMap into another. The benchmark is selected to show whether or not the previous benchmarks are indicative of throughput when applied whilst iterating over MultiMap. Instead of evaluating this for all previous benchmarks, the case of inserting new complex elements into a MultiMap with a single primary index was chosen as it demonstrated a significant improvement in throughput for the Multi-Lock and HAMT MultiMaps with respect to DBToaster's MultiMap implementation. Like the agregation benchmark, 100,000 elements are iterated across.

Figure 5.12 presents a comparison of throughputs for the `addOrDelOnZero` operation for all three thread-safe MultiMap implementations where the elements inserted are supplied by iterating across another MultiMap's elements using the `foreach` operation. Results for both the AMD Ryzen 5-2500U and Intel i5-6500 are presented.

The results for this benchmark are almost identical to those for the original benchmark where iteration was performed using a parallelised for-loop instead of by iterating over a MultiMap's elements in parallel. Unlike the aggregation benchmarks, this set of benchmarks – though an ideal case for the thread-safe MultiMaps – demonstrates that combining parallel iteration across MultiMap elements with concurrent MultiMap operations can result in significant throughput improvements for the Multi-Lock and HAMT MultiMap implementations.

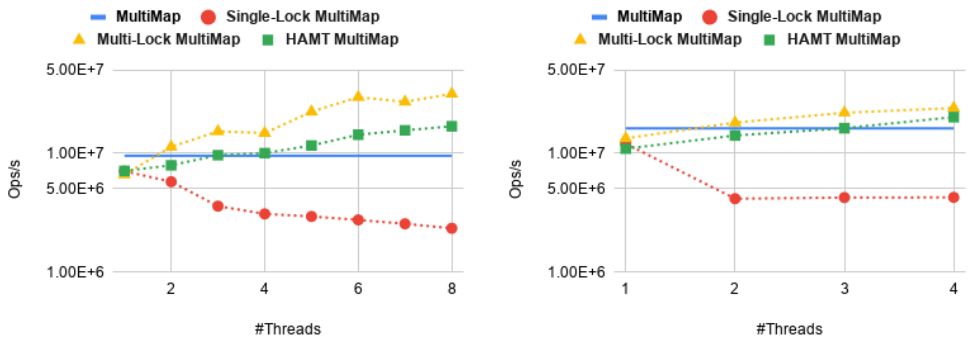


Figure 5.11: Reinserting

Figure 5.12: Inserting `addOrDelOnZero` operations per second for 1–8 and 1–4 threads for complex elements and MultiMaps with only a primary index. Left plot: AMD Ryzen 5-2500U. Right plot: Intel i5-6500.

5.3 Summary

Across all benchmarks, the HAMT MultiMap is the most consistent in demonstrating throughput scaling with increasing thread counts. However, it does not always provide greater throughput than the other thread-safe MultiMaps, or DBToaster's MultiMap implementation. In situations where inserts are common, erases are rare, and complex elements are being stored, the HAMT MultiMap is a good fit. However, the Multi-Lock MultiMap is a close contender, and when resizes are factored out it appears to be the best choice for insert-based workloads. For scenrios where lookups are common, the Single-Lock MultiMap would be a good choice, though its lack of performance in other areas is offputting for practical application.

None of the three thread-safe MultiMap designs demonstrated an improvement upon DBToaster's current MultiMap implementation across all benchmarks, and in come cases, no thread-safe MultiMap was able to beat it. It is clear that in cases where less work was involved for a single operation – such as those involving trivial elements, erasing `addOrDelOnZero` operations, or reinserting `addOrDelOnZero` operations – were less effectively parallelised by the thread-safe MultiMaps. It is difficult to effectively parallelise these operations, as the complexities and overheads involved in ensuring thread-safety for these operations may overshadow the work involved in the operation itself. This may suggest that a hybrid approach of both serial and thread-safe MultiMaps may be able to give the best performance in a practical setting.

The presented results have shown that different architectures designs may also produce different results for some MultiMap designs – namely the Multi-Lock MultiMap – and that this should be taken into account for future designs. Given the complexity of designing around hardware like this, it may be a better choice to implement a design which limits the sharing of data between threads further than the presented designs so as to reduce the effects of different microarchitectural layouts.

Finally, the results presented here are by no means comprehensive, and it is clear that further evaluation is needed to pin down the exact reasons behind some throughput

observations on separate microarchitectures. For example, all workloads presented are well balanced between threads and are not necessarily indicative of the throughputs achievable for other workloads. Future evaluations may consider constructing workloads with more biased hashing, or with mixed operations which may be more instructive of the practical performance of each implementation. Furthermore, the evaluation was limited by a lack of permissions for the target systems, meaning that more in-depth analysis using hardware counters could not be carried out.

Chapter 6

Conclusion

This chapter discusses the various successes of the project, as well as shortcomings in the approaches taken, and possible directions for work in the future.

6.1 Achievements

The project succeeded in producing three thread-safe MultiMaps which demonstrate promising results for a range of cases. It demonstrated how state-of-the-art techniques used in thread-safe hashmaps and tries can be applied to more general data structures such as the MultiMap, and that even more simplistic designs – such as the Multi-Lock MultiMap – can beat out more complex designs – such as the HAMT MultiMap – under the right conditions.

Furthermore, although the project failed to produce a MultiMap with better throughput than DBToaster's current implementation for all cases, it succeeded in highlighting areas of difficulty for future thread-safe implementations to be focused on.

Finally, the project demonstrated that different parallel architectures can have a significant impact on performance of thread-safe data structures, which should be accounted for in their design.

6.2 Future Work

Despite all three thread-safe MultiMap implementations demonstrating that they can be used to achieve greater throughput for some operations, no implementation is universally better than DBToaster's serial MultiMap. Each implementation has its weaknesses, characterised by their design decisions and complexity trade offs. Moreover, there are still components of the implementations which will need to be revisited for practical application, such as the thread-safe memory pool used throughout. For example, the `slice` operation was not parallelised during the project, though for a final implementation this would be desirable.

This project only explored locking data structures, which require that threads block

others while they complete their operations. This is because the MultiMap is defined as a composite of several hash indexes which need to be operated on separately. For now, lock-free data structures that use atomic instructions to complete their operations are infeasible because of this. Its possible that restructuring the MultiMap could make this possible but determining how this could be done is far out of scope of this project. Alternatively, its possible that the project could be continued in two directions: implementing specialised thread-safe MultiMaps for specific cases, or implementing an alternative form of code generation to support alternative designs.

Specialisation is quite common in performant thread-safe data structures, and if possible – such as in the case of hashmaps requiring static bucket array sizes – it can be a powerful tool for improving performance. The MultiMap is described as being composed of multiple hash indexes, but it is not uncommon in DBToaster-generated code for a MultiMap to consist of only a primary hash index. In this case, a specialised design could leverage atomics-based designs as it only have to operate on the single hashmap, potentially allowing for a faster specialised implementation. This kind of data structure specialisation could be leveraged to create an ensemble of performant thread-safe MultiMaps which, together, cover all cases.

Code generation, on the other hand, is another potential route to better performing parallel code in general. Because all MultiMap implementations presented in this report work on shared memory, overheads incurred by cache invalidations will always be a problem, and may result in poorer performance than presented for other workloads. Whilst alignment techniques can help mitigate cache invalidations, cache invalidations they are an inherent problem of working with shared memory. The only way to universally reduce cache invalidations is to reduce the data shared among threads. For this purpose there is an entire class of *wait-free* data structures which achieve this by separating the data each thread has access to. The drawbacks of these designs tend to be that they invoke much greater complexity for their operations and rely on better cache behaviour making up for this. Techniques for constructing such data structures have been presented in the last decade[38], but practical applications are still lacking. Instead, by modifying the current code generation of DBToaster each thread could be given its own MultiMap which it works on independently. Then, when a combined result is needed, each thread-local MultiMap could be efficiently merged with each other to construct a single MultiMap. This would allow for perfect scaling of parallelised loops operating on MultiMaps, but would require that merging is implemented such that it does not nullify the throughput gained by parallel cache-invalidation-free loop execution.

Finally, there may just be more work to do on the implementations of the three thread-safe MultiMaps produced by the project. High performance thread-safe data structures such as Aleksandar Prokopec's Ctrie[27] took more than a single revision to become practical[29], and its possible that the project's implementations simply need additional time for further evaluation and optimisation.

6.3 Reflection

In reflection, this project required a lot of research and experimentation to get to the state is in. For widely used data structures such as queues, lists, hashmaps, and trees, there is a wealth of material covering all kinds of performant thread-safe implementations, but many techniques used in these implementations are simply not applicable to more complex structures such as the MultiMap, and in most cases state-of-the-art techniques researched during the project were discounted for this reason. Despite this, it appears that there are many more possible directions to take this project in future.

Whilst the future work detailed above covers several possible implementations for future investigation, there would may be benefit in spending additional time characterising the short comings of the presented implementations, particularly those uncovered by the evaluation benchmarks. Furthermore, extending benchmarks to include different types of workloads and distributions of elements' hashes would be desirable to really identify whether future implementations are practical.

Throughout the project many other designs were considered, implemented, and discarded. Custom thread pools, alternate lock implementations, memory pools, and supporting data structures were implemented but ultimately not included in this report due to its limited size. Such efforts were not wasted, and helped build a much greater understanding of the project area, without which the results of the projects would likely not be the quality they are. With so many interacting components, one of the only ways to truly know how well parallel code performs is to run it.

Chapter 7

Bibliography

[1] The CPUSHACK Museum. *CPU of the Day: Intel Jayhawk – The Bird that Never Was*. 2018. URL: http://www.cpushack.com/2018/03/15/cpu-of-the-day-intel-jayhawk-the-bird-that-never-was/ (visited on 04/09/2021).

[2] The DBToaster Consortium. *Welcome to dbtoaster.org*. 2017. URL: https://dbtoaster.github.io/ (visited on 04/09/2021).

[3] OpenMP. *OpenMP Application Programming Interface Examples*. 2016. URL: https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf (visited on 04/09/2021).

[4] Mark S Papamarcos and Janak H Patel. "A low-overhead coherence solution for multiprocessors with private cache memories". In: *Proceedings of the 11th annual international symposium on Computer architecture*. 1984, pp. 348–354.

[5] ML Scott and W Bolosky. "False sharing and its effect on shared memory performance". In: *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*. Vol. 57. 1993, p. 41.

[6] Herb Sutter. *atomic Weapons: The C++ Memory Model and Modern Hardware*. 2013. URL: https://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/ (visited on 04/09/2021).

[7] H Sundell and P Tsigas. *Lock-free and practical deques using single-word compare-and-swap," Computing Science, Chalmers University of Technology*. Tech. rep. Tech. Rep. 2004-02, Mar, 2004.

[8] John D Valois. "Lock-free linked lists using compare-and-swap". In: *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. 1995, pp. 214–222.

[9] Lawrence Livermore National Laboratory. *Mutex Variables Overview*. URL: https://hpc-tutorials.llnl.gov/posix/mutex_variables/ (visited on 04/09/2021).

[10] OS Dev. *Context Switching*. 2020. URL: https://wiki.osdev.org/Context_Switching (visited on 04/09/2021).

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

[12] Fedor G Pikus. *C++ atomics: from basic to advanced. What do they really do?* 2017. URL: https://github.com/CppCon/CppCon2017/blob/master/Presentations/C%2B%2B%20Atomics%2C%20From%20Basic%20to%20Advanced/C%2B%2B%20Atomics%2C%20From%20Basic%20to%20Advanced%20-%20Fedor%20Pikus%20-%20CppCon%202017.pdf (visited on 04/09/2021).

[13] Felix Cloutier. *PAUSE – Spin Loop Hint*. URL: https://www.felixcloutier.com/x86/pause (visited on 04/09/2021).

[14] ARM. *ARM Compiler armasm User Guide*. 2021. URL: https://developer.arm.com/documentation/dui0473/j/arm-and-thumb-instructions/yield (visited on 04/09/2021).

[15] matklad. *Mutexes Are Faster Than Spinlocks*. 2020. URL: https://matklad.github.io/2020/01/04/mutexes-are-faster-than-spinlocks.html (visited on 04/08/2021).

[16] OpenMP. *OpenMP Application Programming Interface Version 4.5*. 2015. URL: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf (visited on 04/08/2021).

[17] OpenMP. *OpenMP Application Programming Interface Version 5.0*. 2018. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf (visited on 04/08/2021).

[18] GCC. *Built-in Functions for Memory Model Aware Atomic Operations*. URL: https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html (visited on 04/09/2021).

[19] Martin Ankerl. *Hashmaps Benchmarks - Finding the Fastest, Memory Efficient Hashmap*. 2019. URL: https://martin.ankerl.com/2019/04/01/hashmap-benchmarks-01-overview/ (visited on 04/09/2021).

[20] Maged M. Michael. *High Performance Dynamic Lock-Free Hash Tables and List-Based Sets*. Tech. rep. IBM Thomas J. Watson Research Center, 2002.

[21] OpenJDK. *ConcurrentHashMap.java*. 2011. URL: https://github.com/openjdk-mirror/jdk7u-jdk/blob/master/src/share/classes/java/util/concurrent/ConcurrentHashMap.java (visited on 04/07/2021).

[22] Arun Pandey. *How ConcurrentHashMap Works Internally*. 2016. URL: https://dzone.com/articles/how-concurrenthashmap-works-internally-in-java (visited on 04/07/2021).

[23] Abseil. *Abseil Containers*. 2017. URL: https://abseil.io/docs/cpp/guides/container#hash-tables (visited on 04/09/2021).

[24] Gregory Popovitch. *The Parallel Hashmap*. 2019. URL: https://greg7mdp.github.io/parallel-hashmap (visited on 04/01/2021).

[25] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. "Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs". In: *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE. 2010, pp. 185–192.

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

[27] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. "Cache-aware lock-free concurrent hash tries". In: *arXiv preprint arXiv:1709.06056* (2017).

[28] Aleksandar Prokopec. *Ctries*. 2012. URL: https://github.com/axel22/Ctries (visited on 04/09/2021).

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

[30] Peter A Buhr, Michel Fortier, and Michael H Coffin. “Monitor classification”. In: *ACM Computing Surveys (CSUR)* 27.1 (1995), pp. 63–107.

[31] Maged M Michael and Michael L Scott. “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms”. In: *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. 1996, pp. 267–275.

[32] Microsoft. *Object lifetime and resource management (RAII)*. 2019. URL: https://docs.microsoft.com/en-us/cpp/cpp/object-lifetime-and-resource-management-modern-cpp?view=msvc-160 (visited on 04/10/2021).

[33] LLVM. *LLVM: llvm::PointerUnion¡ PTs ¿ Class Template Reference*. 2021. URL: https://llvm.org/doxygen/classllvm_1_1PointerUnion.html (visited on 04/08/2021).

[34] TPC. *TPC-H Homepage*. 2021. URL: http://www.tpc.org/tpch/ (visited on 04/10/2021).

[35] AMD. *Ryzen 5-2500U Mobile Processor with Radeon Vega 8 Graphics*. 2017. URL: https://www.amd.com/en/products/apu/amd-ryzen-5-2500u (visited on 04/08/2021).

[36] Intel. *Intel Core i5-6500 Processor*. 2015. URL: https://ark.intel.com/content/www/us/en/ark/products/88184/intel-core-i5-6500-processor-6m-cache-up-to-3-60-ghz.html (visited on 04/08/2021).

[37] Christopher Wellons. *Hash Function Prospector*. 2021. URL: https://github.com/skeeto/hash-prospector (visited on 04/10/2021).

[38] Alex Kogan and Erez Petrank. “A methodology for creating fast wait-free data structures”. In: *ACM SIGPLAN Notices* 47.8 (2012), pp. 141–150.

Appendices

Appendix A

Pseudocode for MultiMap Operations

Operation *get(k)*:
```
return primary_index.get(k);
```

Operation *getOrDefault(k)*:
```
elem ← primary_index.get(k);
if elem ≠ NULL then
    return elem.count;
else
    return 0;
end
```

Operation *slice(k, i, f)*:
```
elem ← secondary_indexes[i].slice(k);
while elem ≠ NULL do
    f(elem);
    elem ← elem.next;
end
return;
```

Operation *add(k, v)*:
```
if v ≠ 0 then
    elem ← primary_index.get(k);
    if elem ≠ NULL then
        elem.count ← elem.count + v;
    else
        k.count ← v;
        primary_index.insert(k);
        secondary_indexes.insert(k);
    end
end
return;
```

Operation *addOrDelOnZero(k, v)*:

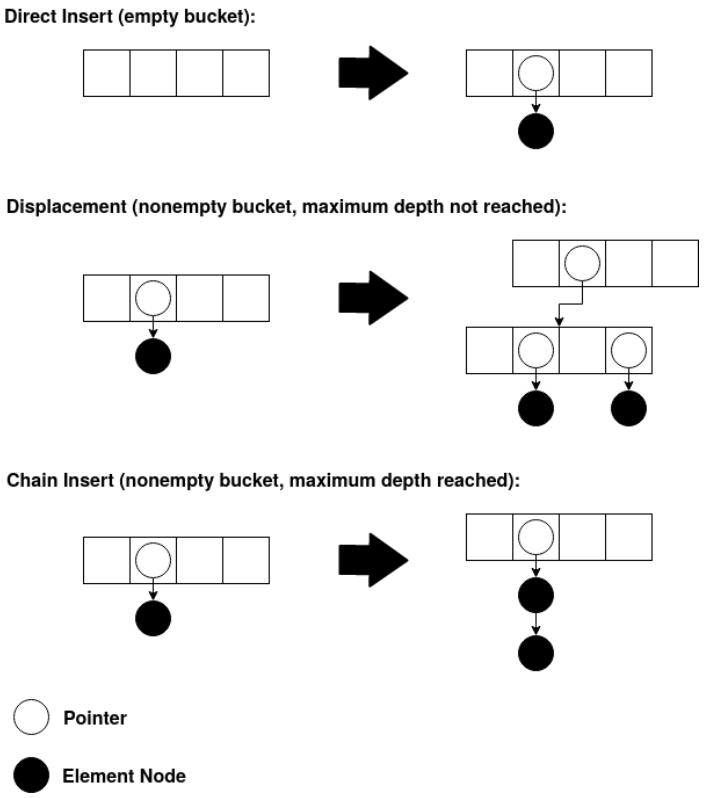
```
if v ≠ 0 then
    elem ← primary_index.get(k);
    if elem ≠ NULL then
        elem.count ← elem.count + v;
        if elem.count = 0 then
            primary_index.erase(elem);
            secondary_indexes.erase(elem);
        end
    else
        k.count ← v;
        primary_index.insert(k);
        secondary_indexes.insert(k);
    end
end
return;
```

Operation *setOrDelOnZero(k, v)*:

```
if v ≠ 0 then
    elem ← primary_index.get(k);
    if elem ≠ NULL then
        if v = 0 then
            primary_index.erase(elem);
            secondary_indexes.erase(elem);
        else
            elem.count ← v;
        end
    else
        k.count ← v;
        primary_index.insert(k);
        secondary_indexes.insert(k);
    end
end
return;
```

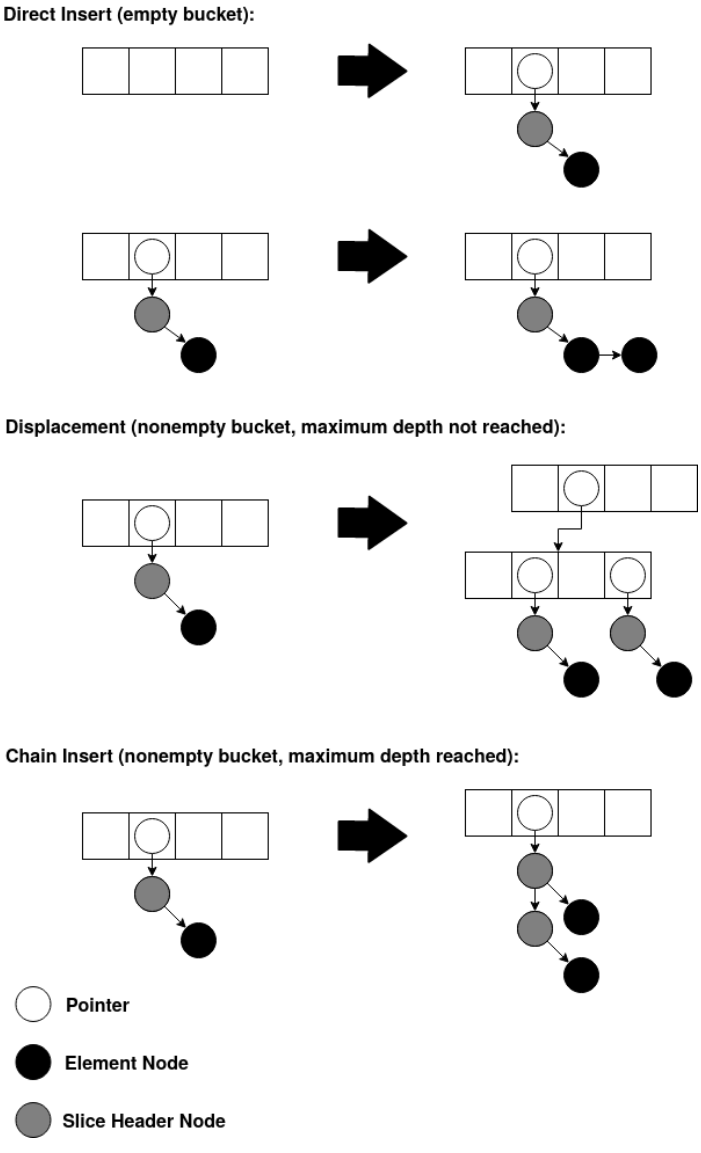
Appendix B

Example Insert Operations On HAMT Primary Hash Index



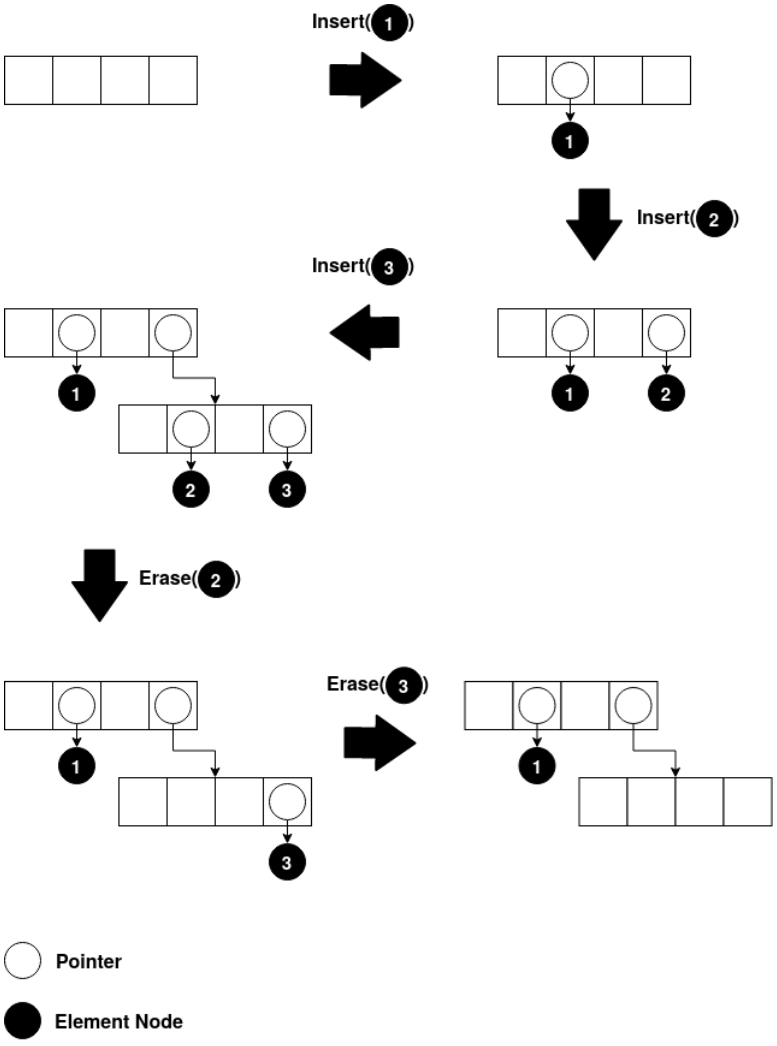
Appendix C

Example Insert Operations On HAMT Secondary Hash Index



Appendix D

Example Operations On HAMT Primary Hash Index



Note: numbers do not correspond to element values

Appendix E

Pseudocode for Thread-Safe MultiMap Operations

Operation *get(k)*:
```
return primary_index.get(k);
```

Operation *getOrDefault(k)*:
```
elem ← primary_index.get(k);
if elem ≠ NULL then
    return elem.count;
else
    return 0;
end
```

Operation *slice(k, i, f)*:
```
elem ← secondary_indexes[i].slice(k);
while elem ≠ NULL do
    f(elem);
    elem ← elem.next;
end
return;
```

Operation *add(k, v)*:
```
if v ≠ 0 then
    transaction ←
      primary_index.transaction(k);
    elem ← transaction.get(k);
    if elem ≠ NULL then
        elem.count ← elem.count + v;
    else
        k.count ← v;
        transaction.insert(k);
        secondary_indexes.insert(k);
    end
end
return;
```

```
Operation addOrDelOnZero(k, v):
    if v ≠ 0 then
        transaction ←
          primary_index.transaction(k);
        elem ← transaction.get(k);
        if elem ≠ NULL then
            elem.count ← elem.count + v;
            if elem.count = 0 then
                transaction.erase(elem);
                secondary_indexes.erase(elem);
            end
        else
            k.count ← v;
            transaction.insert(k);
            secondary_indexes.insert(k);
        end
    end
    return;
```

```
Operation setOrDelOnZero(k, v):
    if v ≠ 0 then
        transaction ←
          primary_index.transaction(k);
        elem ← transaction.get(k);
        if elem ≠ NULL then
            if v = 0 then
                transaction.erase(elem);
                secondary_indexes.erase(elem);
            else
                elem.count ← v;
            end
        else
            k.count ← v;
            transaction.insert(k);
            secondary_indexes.insert(k);
        end
    end
    return;
```

Appendix F

PointerSum Source Code

```
1  #ifndef POINTER_SUM_HPP
2  #define POINTER_SUM_HPP
3
4  /**
5   * This file contains the implementation of the PointerSum class
      used within the
6   * HAMT MultiMap to implement its bucket pointers and spinlocks.
7   */
8
9  #include "macro.hpp" // Supplied with DBToaster
10
11 #include <cassert>
12 #include <cstdint>
13 #include <iostream>
14
15
16 // DBToaster project namespace
17 namespace dbtoaster {
18
19
20 // Anonymous namespace, used for implementation details
21 namespace {
22
23 /**
24  * Template meta programming struct, used to compute the bitmask
      required to extract
25  * a tag with N different values from a pointer.
26  */
27 template <std::size_t N, std::size_t M = 0>
28 struct Mask {
29   constexpr static const std::size_t value = Mask<N / 2, (M * 2) |
      1>::value;
30 };
31
32 template <std::size_t M>
33 struct Mask<0, M> {
34   constexpr static const std::size_t value = M;
35 };
36
```

```
37 /**
38  * Template metaprogramming struct, used to find the index of a type
       in a list of
39  * types.
40  */
41 template <typename S, typename... Ts>
42 struct IndexOf;
43
44 template <typename S, typename T, typename... Ts>
45 struct IndexOf<S, T, Ts...> {
46   constexpr static const std::size_t value = 1 + IndexOf<S, Ts...>::
       value;
47 };
48
49 template <typename S, typename... Ts>
50 struct IndexOf<S, S, Ts...> {
51   constexpr static const std::size_t value = 0;
52 };
53
54 } // anonymous namespace
55
56 /**
57  * PointerSum class. Implements a special pointer type which can
       store both a pointer
58  * and a tag within the size of a single pointer by using lower bits
       of the pointer.
59  *
60  * Ts... defines the types that the pointer supports.
61  */
62 template <typename... Ts>
63 class PointerSum {
64  private:
65
66   // Assert that there is at least one type defined.
67   static_assert(sizeof...(Ts) > 0, "Nonzero␣number␣of␣types␣must␣be␣
       specified");
68
69   // Compute tag and pointer masks
70   constexpr static const std::uintptr_t TAG_MASK = Mask<sizeof...(Ts
       ) - 1>::value;
71   constexpr static const std::uintptr_t PTR_MASK = ~TAG_MASK;
72
73   // Integer used to store pointer and tag
74   std::uintptr_t ptr_;
75
76  public:
77
78   // Disallow the default construction of a PointerSum instance
79   PointerSum() = delete;
80
81   /**
82    * Constructs a PointerSum from the given pointer and type.
       Derives the needed
83    * tag from the given type.
84    *
85    * - T   -- The given type.
```

```
86      * - ptr -- The given pointer.
87      */
88     template <typename T>
89     FORCE_INLINE PointerSum(T* ptr) noexcept : ptr_{(reinterpret_cast<
         std::uintptr_t>(ptr) & PTR_MASK) | IndexOf<T, Ts...>::value} {}
90
91     /**
92      * Atomically assigns the value of another PointerSum to this one.
93      *
94      * - other -- The given PointerSum.
95      */
96     template <typename T>
97     FORCE_INLINE PointerSum& operator=(const PointerSum& other) {
98       #pragma omp atomic write
99       ptr_ = other.ptr_;
100      return *this;
101    }
102
103    /**
104     * Atomically assigns the given pointer of the given type. Derives
          the needed
105     * tag from the given type.
106     *
107     * - T   -- The given type.
108     * - ptr -- The given pointer.
109     */
110    template <typename T>
111    FORCE_INLINE PointerSum& operator=(T* ptr) noexcept {
112      std::uintptr_t next_ptr = (reinterpret_cast<std::uintptr_t>(ptr)
          & PTR_MASK) | IndexOf<T, Ts...>::value;
113      #pragma omp atomic write
114      ptr_ = next_ptr;
115      return *this;
116    }
117
118    /**
119     * Atomically gets the current tag value.
120     */
121    FORCE_INLINE std::uintptr_t tag() const noexcept {
122      std::uintptr_t curr_ptr;
123      #pragma omp atomic read
124      curr_ptr = ptr_;
125      return curr_ptr & TAG_MASK;
126    }
127
128    /**
129     * Gets the tag value associated with the given type.
130     *
131     * - T -- The given type.
132     */
133    template <typename T>
134    FORCE_INLINE constexpr static std::uintptr_t tag_of() noexcept {
135      return IndexOf<T, Ts...>::value;
136    }
137
138    /**
```

```
139     * Atomically swaps with the given PointerSum instance.
140     *
141     * - other -- The given PointerSum instance.
142     */
143    FORCE_INLINE void swap(PointerSum& other) noexcept {
144      std::uintptr_t temp = other.ptr_;
145      #pragma omp atomic capture
146      { other.ptr_ = ptr_; ptr_ = temp; }
147    }
148
149    /**
150     * Performs a compare-and-swap on this pointer with another
         pointer if and only
151     * if the current tag corresponds to the given type. Returns true
         if and only if
152     * the compare-and-swap succeeds.
153     *
154     * - T     -- The given type.
155     * - other -- The other pointer.
156     * - weak  -- Specifies if the compare-and-swap operation should
         be weak or strong,
157     *            weak is true by default.
158     */
159    template <typename T>
160    FORCE_INLINE bool compare_and_swap(PointerSum& other, bool weak =
         true) noexcept {
161
162      // Atomically ready tag-pointer pair.
163      std::uintptr_t curr_ptr;
164      #pragma omp atomic read
165      curr_ptr = ptr_;
166
167      // Return false if the tag is not of the expected type
168      if ((curr_ptr & TAG_MASK) != IndexOf<T, Ts...>::value) return
         false;
169
170      // Attempt a compare-and-swap
171      std::uintptr_t next_ptr = other.ptr_;
172      bool success = __atomic_compare_exchange(&ptr_, &curr_ptr, &
         next_ptr, weak, __ATOMIC_RELEASE, __ATOMIC_ACQUIRE);
173      other.ptr_ = curr_ptr;
174
175      // Return whether the compare-and-swap was successful
176      return success;
177    }
178
179    /**
180     * Returns true if the current tag corresponds to the given type.
181     *
182     * - T -- The given type.
183     */
184    template <typename T>
185    FORCE_INLINE bool contains() const noexcept {
186      std::uintptr_t ptr;
187      #pragma omp atomic read
188      ptr = ptr_;
```

```
189     return (ptr & TAG_MASK) == IndexOf<T, Ts...>::value;
190   }
191
192   /**
193    * Gets the current value of the pointer casted to the given type.
194    *
195    * - T -- The given type.
196    */
197   template <typename T>
198   FORCE_INLINE T* get() noexcept {
199     return reinterpret_cast<T*>(ptr_ & PTR_MASK);
200   }
201
202   /**
203    * Gets the current value of the pointer casted to the given type.
204    *
205    * - T -- The given type.
206    */
207   template <typename T>
208   FORCE_INLINE const T* get() const noexcept {
209     return reinterpret_cast<T*>(ptr_ & PTR_MASK);
210   }
211
212 };
213
214 } // namespace dbtoaster
215
216 #endif // POINTER_SUM_HPP
```

Appendix G

Multi-Lock MultiMap Secondary Hash Function

```
1    std::size_t secondaryHash(HashType h1) const {
2      std::size_t h2 = h1;
3      h2 ^= h2 >> 16;
4      h2 *= 0x7feb352d;
5      h2 ^= h2 >> 15;
6      h2 *= 0x846ca68b;
7      h2 ^= h2 >> 16;
8      return h2 % N;
9    }
```

Appendix H

Full `addOrDelOnZero` Reinsert Microbenchmark Results

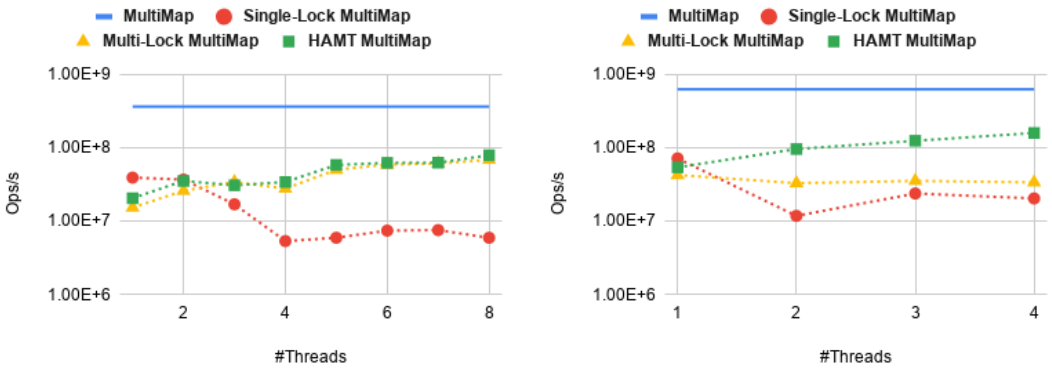


Figure H.1: Reinserting `addOrDelOnZero` operations per second for 1–8 and 1–4 threads with trivial elements. Blue line is throughput for DBToaster's current MultiMap with a single thread. Left plot: AMD Ryzen 5-2500U. Right plot: Intel i5-6500.

Figure H.2: Reinserting `addOrDelOnZero`s per second for an increasing number of threads. Blue line represents serial MultiMap with a single thread. Left: Complex elements on the AMD Ryzen 5-2500U. Right: Complex elements on the Intel i5-6500.