THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

On Design and Applications of Practical Concurrent Data Structures

IVAN WALULYA

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

On Design and Applications of Practical Concurrent Data Structures

IVAN WALULYA

ISBN 978-91-7597-815-4
Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie nr 4496.
ISSN 0346-718X

Technical report 164D
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg, Sweden
Phone: +46 (0)31-772 10 00

Author e-mail: ivanw@chalmers.se

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2018.

On Design and Applications of Practical Concurrent Data Structures

Ivan Walulya
Department of Computer Science and Engineering, Chalmers University of Technology

ABSTRACT

The proliferation of multicore processors is having an enormous impact on software design and development. In order to exploit parallelism available in multicores, there is a need to design and implement abstractions that programmers can use for general purpose applications development. A common abstraction for coordinated access to memory is a concurrent data structure. Concurrent data structures are challenging to design and implement as they are required to be correct, scalable, and practical under various application constraints. In this thesis, we contribute to the design of efficient concurrent data structures, propose new design techniques and improvements to existing implementations. Additionally, we explore the utilization of concurrent data structures in demanding application contexts such as data stream processing.

In the first part of the thesis, we focus on data structures that are difficult to parallelize due to inherent sequential bottlenecks. We present a lock-free vector design that efficiently addresses synchronization bottlenecks by utilizing the combining technique. Typical combining techniques are blocking. Our design introduces combining without sacrificing non-blocking progress guarantees. We extend the vector to present a concurrent lock-free unbounded binary heap that implements a priority queue with mutable priorities.

In the second part of the thesis, we shift our focus to concurrent search data structures. In order to offer strong progress guarantee, typical implementations of non-blocking search data structures employ a “helping” mechanism. However, helping may result in performance degradation. We propose *help-optimality*, which expresses optimization in amortized step complexity of concurrent operations. To describe the concept, we revisit the lock-free designs of a linked-list and a binary search tree and present improved algorithms. We design the algorithms without using any language/platform specific constructs; we do not use bit-stealing or runtime type introspection of objects. Thus, our algorithms are *portable*. We further delve into multi-dimensional data and similarity search. We present the first lock-free multi-dimensional data structure and linearizable nearest neighbor search algorithm. Our algorithm for nearest neighbor search is generic and can be adapted to other data structures.

In the last part of the thesis, we explore the utilization of concurrent data structures for deterministic stream processing. We propose solutions to two challenges prevalent in data stream processing: (1) efficient processing on cloud as well as edge devices and (2) deterministic data-parallel processing at high-throughput and low-latency. As a first step, we present a methodology for customization of streaming aggregation on low-power multicore embedded platforms. Then we introduce Viper, a communication module that can be integrated into stream processing engines for the coordination of threads analyzing data in parallel.

Keywords: atomicity, combining, concurrent data structures, lock-free, locking, multicore, non-blocking, synchronization, stream processing

List of Publications

Appended publications

1. **Ivan Walulya** and Philippas Tsigas, "Scalable lock-free vector with combining," in *the Proceedings of the 31st International Parallel and Distributed Processing Symposium*, pp. 917–926, IEEE 2017.

2. **Ivan Walulya**, Bapi Chatterjee, Ajoy K. Datta, Rashmi Niyoliya, and Philippas Tsigas,"Concurrent lock-free unbounded priority queue with mutable priorities," in *the Proceedings of the 20th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, LNCS, Springer 2018.

3. Bapi Chatterjee, **Ivan Walulya** and Philippas Tsigas, "Help-optimal and languageportable lock-free concurrent data structures," in *the Proceedings of the 45th International Conference on Parallel Processing*, pp. 360–369, IEEE 2016.

4. Bapi Chatterjee, **Ivan Walulya**, and Philippas Tsigas, "Concurrent linearizable nearest neighbour search in lockfree-kd-tree," in *the Proceedings of the 19th International Conference on Distributed Computing and Networking*, pp. 11:1–11:10, ACM 2018.

5. Lazaros Papadopoulos, Dimitrios Soudris, **Ivan Walulya**, and Philippas Tsigas, "Customization methodology for implementation of streaming aggregation in embedded systems," *Journal of Systems Architecture - Embedded Systems Design*, vol. 66-67, pp. 48–60, Elsevier 2016.

6. **Ivan Walulya**, Dimitris Palyvos-Giannas, Yiannis Nikolakopoulos, Vincenzo Gulisano, Marina Papatriantafilou, and Philippas Tsigas, "Viper: A module for communication-layer determinism and scaling in low-latency stream processing," *Future Generation Computer Systems*, vol. 88, pp. 297–308, Elsevier 2018.

Other publications

The following articles were also published during my PhD studies, but not included in this thesis.

A. **Ivan Walulya**, Yiannis Nikolakopoulos, Vincenzo Gulisano, Marina Papatriantafilou, and Philippas Tsigas, "Viper: Communication-layer determinism and scaling in low-latency stream processing," in *Euro-Par 2017: Parallel Processing Workshops*, vol. 10659, pp. 129–140, LNCS, Springer 2018.

B. Lazaros Papadopoulos, **Ivan Walulya**, Philippas Tsigas, and Dimitrios Soudris, "A systematic methodology for optimization of applications utilizing concurrent data structures," *IEEE Transactions on Computers*, vol. 65, no. 7, pp. 2019–2031, IEEE 2016.

C. Lazaros Papadopoulos, **Ivan Walulya**, Paul Renaud-Goud, Philippas Tsigas, Dimitrios Soudris, and Brendan Barry, "Performance and power consumption evaluation of concurrent queue implementations in embedded systems," *Computer Science - Research and Development*, vol. 30, no. 2, pp. 165–175, Springer 2015.

D. Vincenzo Gulisano, Yiannis Nikolakopoulos, **Ivan Walulya**, Marina Papatriantafilou, and Philippas Tsigas, "Deterministic real-time analytics of geospatial data streams through scalegate objects," in *the Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pp. 316–317, ACM 2015.

E. **Ivan Walulya**, Yiannis Nikolakopoulos, Marina, and Philippas Tsigas, "Concurrent data structures in architectures with limited shared memory support," in *Euro-Par 2014: Parallel Processing Workshops*, vol. 8805, pp. 189–200, LNCS, Springer 2014.

F. Lazaros Papadopoulos, **Ivan Walulya**, Philippas Tsigas, Dimitrios Soudris, and Brendan Barry, "Evaluation of message passing synchronization algorithms in embedded systems," in *the Proceedings of the 14th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pp. 282–289, IEEE 2014.

Research Contribution

Paper 1 was authored in collaboration with Philippas Tsigas. I contributed to the design and implementation of the presented algorithms. Additionally, I participated in the writing of the paper. **Paper 2** builds on previous work by Ajoy K. Datta and Rashmi Niyoliya extending the work in Paper 1 to build a concurrent unbounded binary heap. In this paper, I contributed to the design of the presented algorithms, implementations, and authoring of paper.

In **Paper 3**, I contributed to the design of the algorithms, proof sketches, and developed the C/C++ implementations presented in the paper. Additionally, I developed the benchmark suite utilized for all results presented in the paper.

My contributions to **Paper 4** include participation in the design of Nearest Neighbor Search (NNS) algorithm on the concurrent KD-Tree, proof of correctness of the NNS algorithm. Implementation of the designed algorithms and benchmarks used in the evaluation of the algorithm in addition to co-authoring.

Paper 5 is an extension of joint work with Lazaros Papadopoulos in **Papers B, C, F**; together, the works were developed for exploiting parallelism available in low-power embedded systems. In this work, my contributions were on the design of concurrent data structures and algorithms ported onto the embedded systems. Additionally, I participated in the writing of the papers, while Papadopoulos performed the bulk of the experiments presented in the papers.

In **Paper 6**, I integrated ScaleGate (a novel interface for the deterministic merging of multiple data streams) into Apache Storm Stream Processing Engine, and extended ScaleGate to include flow-control thus making it usable in a task-based scheduler as opposed to a thread-based scheduler. Additionally, I implemented and performed benchmarks for throughput, latency, and energy measurements on general purpose processors. Benchmarks on Odroid devices were implemented in collaboration with Dimitris Palyvos-Giannas. Additionally, I was the lead on the writing of the paper in collaboration with all other authors. This work was a continuation of collaboration on **Paper D**.

To my parents, my brothers, and Lynnie.

Acknowledgments

I would like to start by thanking my supervisor Prof. Philippas Tsigas who encouraged me to pursue a research career, has supported and mentored me all these years with great insight and wisdom. I am also grateful to my co-supervisor Prof. Marina Papatriantafilou for great counsel, encouragement, uplifting discussions, and feedback.

A single name appearing on the cover of this thesis is a misrepresentation of the efforts that have gone into the work herein. I owe a debt of gratitude to my co-authors and collaborators that have directly or indirectly contributed to this work. In no particular order, thank you, Bapi Chatterjee, Yiannis Nikolakopoulos, Vincenzo Gulisano, Aras Atalar, Paul Renaud-Goud, Lazaros Papadopoulos, Charalampos Stylianopoulos, Dimitris Palyvos-Giannas, Adones Rukundo, Daniel Cederman, Anders Gidenstam, Dimitrios Soudris, Brendan Barry and, Ajoy K. Datta.

I am honored to have Assoc. Prof. Danny Hendler as the faculty opponent during the thesis defense. I would like to acknowledge members of the grading committee: Dr. Emmanuelle Anceaume, Prof. Håkan Grahn, Prof. Lasse Natvig, and Associate Prof. Pedro Petersen Moura Trancoso. I also wish to thank my examiner Prof. Aarne Ranta, and the follow-up committee for their support during my studies.

I take this opportunity to thank the administration at the Department of Computer Science and Engineering. I received tremendous help on administrative tasks from Eva Axelsson, Rebecca Cyren, Marianne Pleen-Schreiber, Tiina Rankanen, and Peter Helander. I would also like to extend a token of appreciation to my managers Tomas Olovsson and Peter Lundin.

Many thanks to all past and present members of the NS group that have contributed to such a great working environment: Ali, Aljoscha, Amir, Bastian, Elad, Farnaz, Fazeleh, Georgia, Giorgos, Hannaneh, Iosif, Magnus, Nasser, Nhan, Olaf, Oscar, Thomas, Valentin, Valentin, Zhang.

I consider myself particularly lucky to be able to call several current and former members of the department, not just colleagues, but friends. Thank you Alirad, Bhavi, Chloe, Madhavan, Petros, Prajith, Stavros, Stefano, Vagelis, Yiannis, among others.

Special thanks to all the *thirstos*, especially Isaac Muganwa for always pushing me to heights that I might have thought out of reach.

Funding. The work in this thesis was supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2013-2016) / Grant agreement no. 611183, EXCESS Project.

Ivan Walulya
Göteborg, November 2018

Contents

List of Publications v

Personal Contribution vii

Acknowledgements ix

1 Introduction 1

1.1 Shared-Memory Multicore Systems 5

1.1.1 Caches and Memory Consistency 6

1.1.2 Atomic Primitives 8

1.2 Synchronization 9

1.2.1 Blocking Synchronization 10

1.2.2 Non-blocking Synchronization 11

1.2.3 Power of Synchronization Primitives 12

1.3 Concurrent Data Structures 12

1.3.1 Correctness of Concurrent Data Structures 13

1.4 Non-blocking Concurrent Data Structures 15

1.4.1 Design and Implementation Approaches 15

1.4.2 Concurrent Data Structures for Efficient Data Stream Processing 18

1.5 Contributions 20

Bibliography 22

2 Scalable Lock-Free Vector with Combining 31

2.1 Introduction 32

2.1.1 Related Work: 34

2.2 System Model and Definitions 35

2.3 Algorithm 37

2.3.1 Overview of the Algorithm 37

2.3.2 Implementation Details 38

2.3.3 Correctness 46

2.3.4 Memory Management and ABA Problems 49
2.4 Performance Evaluation 50
2.4.1 Experimental Results and Discussion 51
2.5 Conclusion 53
Bibliography 53

3 Concurrent Lock-free Unbounded Priority Queue 59
3.1 Introduction 60
3.2 Preliminaries 62
3.3 Algorithm 63
3.3.1 Lock-free ADT Operations 65
3.3.2 Design Optimizations 67
3.4 Correctness Proof 69
3.5 Evaluation 70
3.6 Conclusion 75
Bibliography 75

4 Help-optimal and Language-portable Lock-free Concurrent Data Structures 81
4.1 Introduction 82
4.1.1 Overview 82
4.1.2 Related Work 85
4.2 Help-optimality: Motivation 86
4.3 Help-optimal Lock-free Linked-list 89
4.3.1 Design 89
4.3.2 Correctness and Lock-freedom 93
4.3.3 Amortized Step Complexity 94
4.4 Help-optimal Lock-free BST 95
4.4.1 Design 95
4.4.2 Correctness and Lock-freedom 101
4.5 Help-optimality: Specification 101
4.6 Experimental Evaluation 102
4.6.1 Overview 102
4.6.2 Experimental Set-up 103
4.6.3 Performance Results and Discussion 104
4.7 Conclusion 108
Bibliography 109

5 Concurrent Linearizable Nearest Neighbour Search in LockFree-kD-tree 113
5.1 Introduction . . . 114
5.1.1 Background . . . 114
5.1.2 A high-level summary of the work . . . 116
5.2 LockFree-kD-tree: Basic Design . . . 118
5.2.1 Design of the LFkD-tree . . . 118
5.2.2 Sequential Behaviour of the ADT Operations . . . 119
5.3 LockFree-kD-tree: Implementation . . . 120
5.3.1 Lock-free Synchronization: Basics . . . 120
5.3.2 Linearizable ADD, REMOVE and CONTAINS operations 124
5.3.3 Linearizable Nearest Neighbour Search . . . 129
5.4 Correctness and Lock-freedom . . . 140
5.5 A real-life application . . . 149
5.6 Experimental Evaluation . . . 150
5.6.1 Experimental Setup . . . 150
5.6.2 Datasets . . . 151
5.6.3 Observations and Discussion . . . 152
5.7 Conclusion and Future Work . . . 156
Bibliography . . . 156

6 Customization Methodology for Implementation of Streaming Aggregation in Embedded Systems 161
6.1 Introduction . . . 162
6.2 Related Work . . . 164
6.3 Streaming Aggregation . . . 165
6.3.1 Streaming Aggregation description . . . 165
6.4 Customization Methodology . . . 168
6.4.1 Design Space . . . 168
6.4.2 Methodology description . . . 170
6.5 Demonstration of the Methodology . . . 171
6.5.1 Platforms description . . . 171
6.5.2 Experimental Setup . . . 174
6.5.3 Time-based aggregation results . . . 175
6.5.4 Count-based aggregation results . . . 181
6.5.5 Performance per watt evaluation . . . 184
6.5.6 Discussion of Experimental Results . . . 185
6.6 Conclusion . . . 189
Bibliography . . . 190

7 Viper: A Module for Communication-Layer Determinism and Scaling in Low-Latency Stream Processing 195

7.1 Introduction 196

7.2 System Model 198

7.2.1 Data Streaming 198

7.2.2 Parallelism, determinism and syntactic transparency . . 199

7.2.3 Streaming operators' performance metrics 201

7.3 Operator- vs communication-layer determinism 201

7.3.1 Limitations of operator-layer determinism 201

7.3.2 Additional potential benefits from determinism provisioning in the SPE-communication-layer 204

7.4 The Viper module 205

7.4.1 Viper as an SPE module: Apache Storm use case 206

7.5 Evaluation 210

7.5.1 Intra-Node Parallel Analysis - Setup 210

7.5.2 Intra-Node Parallel Analysis - Scalability 211

7.5.3 Inter-Node Distributed Parallel Analysis - Setup 219

7.5.4 Inter-Node Distributed Parallel Analysis - Scalability . . 220

7.6 Related work 222

7.7 Conclusions 222

Bibliography 223

8 Conclusions and Future Work 227

1
Introduction

In recent years, multicore systems have become ubiquitous; processors in handheld devices, laptops, desktop computers to super-computers contain multiple cores. This emergence of multicore systems is driven largely by *demand*; demand for more computing power. For more than four decades, two scaling principles guided processor design: Moore's law [1] and Dennard's scaling [2]. Moore's law is an observation that the number of transistors cost-effectively placed on a chip doubles approximately every two years. Dennard scaling is related to Moore's law, in that, the power consumption is proportional to transistor size. When transistors got smaller, voltage and current scaled down; the power density stayed the same from one processor generation to the next.

In the early 2000s, the semiconductor industry started experiencing a break down in Dennard scaling: *threshold voltage*, *current leakage*, and subsequent *heat dissipation* do not scale with size, creating a physical limit to the practical size of a transistor – *Power Wall*. The power wall compelled processor designers to change their approach; rather than to increase the clock rate of a single processor, add multiple processors with lower clock rates on a single chip.

A chip with multiple processing units is commonly referred to as *multicore* and each processing unit as a *core*. Generally, the cores communicate through read and write operations on shared-memory (shared-memory multicores). The primary objective is to achieve more throughput with parallel executions instead of improving the completion time of a single execution.

This shift in processor design has had a significant impact on software design and implementation as programmers seek to utilize the multiple computing cores efficiently. In the ideal case, multiple cores are employed to perform a large task split into sub-tasks; each core independently executes its sub-tasks to completion. Applications that execute in this manner are considered *embarrassingly parallel*. In practice, however, many tasks cannot be easily divided and executed independently in parallel[1]; computing cores need to coordinate their actions.

As an analogy, consider a family attempting to assemble an IKEA dining table. If the whole family is involved in the assembly, they could "probably" complete the task faster. For this to happen, they need to divide up the task in order to assemble different sections of the table at the same time. However, some parts of the table must be assembled in a given order; consequently, some members may have to wait for others to complete a given component – *synchronization*. If one member takes too long on a given task, he risks delaying others that are waiting – *Blocking*. Additionally, if the group spends too much time sharing tools and the instruction leaflet – *communication costs*, then they lose the anticipated *speed-up*. The challenge becomes harder with a larger family.

As with the analogy, multicore program execution typically comprises of tasks that can be executed in parallel without coordination and tasks that require coordination among the processes. Coordination is generally required when processors concurrently access shared resources or objects. The goal is to prevent inconsistencies that may result from the interleaving of concurrent accesses by multiple processes.

Initially SC = 0

P1

```
$old \leftarrow SC$
$new \leftarrow old + 1$
$SC \leftarrow new$
```

P2

```
$old \leftarrow SC$
$new \leftarrow old + 1$
$SC \leftarrow new$
```

Figure 1.1: Shared Counter

Take for example the shared counter in Figure 1.1 incremented by two

[1] "No matter how great the talent or efforts, some things just take time. One cannot produce a baby in one month by getting nine women pregnant." – Warren Buffett

processes. After the two processes complete, we expect that the value of the counter is incremented by 2. However, one of the increments could be lost due to processes overwriting each other's modifications. The program is said to have a "race condition", *i.e.,* the outcome of the execution is dependent on the arbitrary interleaving of events [3].

The need for synchronization in concurrent systems predates multicore architecture [4, 5]; any program that allows concurrent operations must be synchronized, regardless of whether the operations are actually executed in parallel[2]. Synchronization allows one to explicitly dictate which interleavings are acceptable and prohibit those that are unacceptable. Generally, this is achieved by guaranteeing some notion of *atomicity*, in that, a given sequence of instructions executed by a single process appears instantaneous to other processes.

In shared-memory multicore systems, synchronization is ordinarily achieved through mutual exclusion: access to the shared object is guarded by locks. Segments of the execution that are guarded by locks are referred to as *critical sections*. A process that wishes to execute inside a critical section must acquire the respective lock; acquiring the lock guarantees that instructions executed inside the critical section appear atomic to other processes. A process releases the lock on exiting the critical section. The popularity of mutual exclusion locks is attributed to their simplicity in most use cases and efficient implementation in presumably all shared-memory platforms.

However, mutual exclusion is associated with several pitfalls; locks are *blocking* – arbitrary delay or failure of a process holding a lock blocks other processes from making progress. Additionally, care must be taken to avoid deadlocks, priority inversions and convoying.

Non-blocking synchronization has emerged as a solution to many pitfalls associated with locks. It takes a more optimistic approach where a process attempts to access a shared object without blocking other processes. Although non-blocking synchronization was initially desirable for fault-tolerance in asynchronous shared memory systems, recent research has shown that it has the potential to increase parallelism [6–10].

Correct implementation of a synchronization mechanism is required to satisfy *safety* and *liveness* properties [11]. Informally, safety states that "bad" things never happen, such as no two processes execute inside the critical section at the same time. Liveness states that "good" things eventually happen; for example, if a lock is free, then some requesting process eventually acquires the lock.

[2] We consider two operations *concurrent* if their execution interval overlaps in time, *parallel* if the operations can actually be executed at the same time.

Regardless of whether blocking or non-blocking, synchronization of processes is a difficult undertaking for application developers. This complexity is often hidden away from the developer through concurrent programming abstractions. A common abstraction for synchronizing access to shared data is in the form of *concurrent data structures.* In a sequential setting, data structures organize data for efficient access. The abstraction of a data structure is described by an Abstract Data Type (ADT), which is an interface definition of operations that can be executed on the data structure.

In concurrent settings, in addition to implementing the ADT, the data structure hides details on the interaction of processes that simultaneously call operations on the data structure. Application developers can utilize a concurrent data structure without concern for the implementation details as long as it observes the interface defined by the Abstract Data Type. Hence, the implementation of efficient, practical concurrent data structure is of paramount importance for application developers at various levels of expertise to fully exploit parallelism available in multicore platforms.

Abstraction does not eliminate the complexity of correct synchronization, but it nevertheless allows us to reason about concurrent interactions at a high-level of interface operations rather than low-level interleaving of machine instructions. The designer of a concurrent data structure takes on the difficult task of choosing how to implement the ADT efficiently and correctly.

Concurrent data structures are generally classified according to safety and liveness properties that they satisfy. At this level of abstraction, the main safety requirement is that operations on the data structure appear indivisible, i.e., every operation appears to take effect without interruption despite any possible interleaving with other operations. Various formalizations of this requirement exist in the literature *e.g.,* Linearizability [12] and Sequential Consistency [13].

Operations on concurrent data structure are associated with different progress guarantees: *blocking* or *non-blocking*. Delay of a process executing a blocking operation can delay others processes, while, execution of non-blocking operations cannot delay or prevent other processes from making progress. There are several levels of non-blocking progress [14].

Concurrent data structures form basic building blocks for more sophisticated applications. One application that has gained significant interest is Data Stream Processing (DSP). The interest in Data Stream Processing is a result of the unprecedented increase in volumes of data generated at high-rates as we integrate computing in all aspects of our lives from smart watches to self-driving vehicles. Users require useful insights from this data in real-time; the data has to be processed in-motion. Stream processing typically has high-throughput and low-latency constraints and thus an excellent application domain for exploitation of concurrent data structures in order to efficiently utilize multicore platforms.

In this thesis, we contribute to the body of research on efficient, practical concurrent data structures in multicore architectures. We present design mechanisms that efficiently address potential synchronization bottlenecks without sacrificing non-blocking progress guarantees. We design algorithms for portable search data structures that do not rely on the programming language or platform specific constructs. Additionally, we explore the utilizing of concurrent data structures in demanding application contexts such as data stream processing.

The rest of this chapter introduces background on shared-memory multicore architectures, synchronization in shared memory systems and design techniques for concurrent data structures. Section 1.1 briefly describes a basic shared-memory architecture, highlighting the effects of cache coherence and memory consistency on parallel programs. Furthermore, it provides the semantics of various read-modify-write instructions available on many shared-memory platforms that typically underlie implementations of synchronization mechanisms. Section 1.2 presents an overview of synchronization mechanisms in shared-memory multicore architectures. Section 1.3 presents related work, challenges in the design of concurrent data structures, and highlights how concurrent data structures are utilized in solving challenges in deterministic parallel data processing. The contributions of this thesis are summarized in Section 1.5.

1.1 Shared-Memory Multicore Systems

A multicore processor is a processor with multiple independent computational units (referred to as cores) on a single chip. Multicore systems may contain one or more multicore processors – *multiprocessor*. We use the terms *processor* and *core* interchangeably to refer to an independent computational unit. Similarly, we use the terms processes and threads interchangeably to refer to active threads of control that potentially share variables in a shared address space.

Figure 1.2 depicts a classical architecture for shared-memory multiprocessors, including several processors, each having multiple cores with private and shared caches, all connected via a shared memory subsystem. Shared-memory multicores have a single shared address space accessible to all cores. Processing cores execute independently and communicate through read and write operations on shared objects in memory. We consider an asynchronous shared memory model where processes may execute at varying speeds and can experience arbitrary delays due to scheduling interrupts, memory page faults and cache behavior.

Single shared address space does not imply single memory; multiple memory modules may be attached to the system and will appear as a single address space

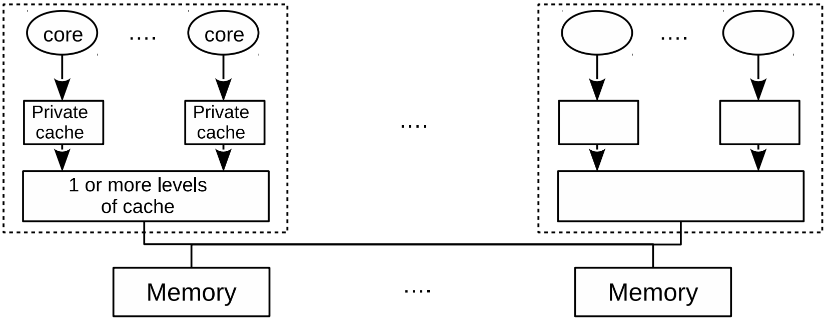


Figure 1.2: Typical architecture for a Shared-Memory Multiprocessor.

to the processors. The shared-memory multiprocessors are classified as either Uniform Memory Access (UMA) – memory access latency is uniform for all processors, or Non-Uniform Memory Access (NUMA) – memory latency is dependent on the memory location and the processor that accesses the location.

1.1.1 Caches and Memory Consistency

The divergence in speeds between processors and memory systems compelled system designers to develop techniques for hiding latency in accessing memory systems. For example, write-back store buffers and cache hierarchies between the cores and memory. Each core typically has a cache hierarchy composed of private and shared caches. The cores store temporary copies of data in cache hierarchy for faster access than reading from memory.

Caches create an illusion of fast high-bandwidth memory by exploiting *locality*. Programs generally access small portions of memory at any small interval in time; either an address is accessed repeatedly, and the accesses are close in time (*temporal locality*) or adjacent addresses are accessed close in time (*spatial locality*). Effectively, bandwidth demands on main memory are reduced, allowing multiple processors to share the same memory.

Unfortunately, when cores store copies of shared data in caches, reasoning about executions by different cores is not straight forward[3]. Replicated copies of data in different caches may not be up-to-date; accordingly, cores may have different views of shared memory locations. The first challenge is ensuring

[3] *"multiprocessor synchronization algorithms assume that each processor accesses the same word in memory, but each processor actually accesses its own copy in its cache. It hardly required a triple-digit IQ to realize that this could cause problems."* – Leslie Lamport

that processors agree on the value returned by a read to a memory location – *coherence*. This is addressed by *cache-coherency* protocols [15] which ensure that all caches hold consistent data. The second challenge is determining when and in what order reads by processors can return values written to different memory locations – *consistency*. We cannot always assume that processors immediately observe changes in memory made by a given processor or that processors observe changes in the order they were issued.

```
                    Initially X = Y = 0
   P1                                         P2

$X \leftarrow 1$                              $Y \leftarrow 1$
if $Y = 0$ then                               if $X = 0$ then
    // enter critical section ;                   // enter critical section ;
```

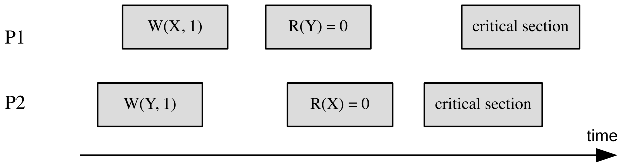


Figure 1.3: Simple algorithm for mutual exclusion

Figure 1.3 illustrates complications that can arise if processors do not have a consistent view of memory access operations. Assume that the processes *P1* or *P2* are running on different processors, the program should ensure that either process *P1* or *P2* executes the critical section, but not *both*. The programmer reasons about the correctness of the program by visualizing its execution, with each line of the program executed line-by-line in *program order* and that writes to a memory address are immediately visible to the other process.

However, if the writes are stalled or buffered and the processes allowed to continue with the next instruction, then memory accesses may appear reordered. P1 and P2 proceed to enter the critical section, before either write is effected in memory. *Memory consistency* models provide a formal specification on whether such executions are allowed. Essentially, consistency models act as a contract between the system hardware and the software on what access orders are legal when multiple processes access common locations.

Sequential Consistency proposed by Lamport [16] extends the programmers reasoning about line-by-line execution: it requires that memory accesses appear to execute one at a time in some global order, and memory accesses by a single process execute in program order. Sequential consistency guarantees that there is no ordering of instructions in Figure 1.3 where both $P1$ and $P2$ enter the critical section. Writes have to be completed before continuing to the **`if`** statement.

Although sequential consistency offers a simple and intuitive programming model, it inhibits many compiler and hardware performance optimizations. Therefore, more relaxed (possibly inconsistent) memory models have been proposed offering substantial performance improvements [17]. Relaxed memory models call for the programmer to explicitly ensure correctness of the program using memory fences (or barriers) and atomic read-modify-write primitives to dictate the appropriate ordering of events.

Modern compiler optimizations may also reorder instructions leading to inconsistent behavior in multicores programs. Therefore, many programming languages provide memory consistency models and synchronization primitives. Explicit synchronization of execution events adds overhead to the execution and therefore should be used prudently to avoid performance degradation.

1.1.2 Atomic Primitives

Modern multicore architectures provide various *read-modify-write* hardware primitives that can atomically read and modify an object in memory. An operation is *atomic* if it appears to complete in a single step or instantaneously to other processes. These atomic instructions are generally used as basic building blocks for implementing synchronization constructs in shared memory multicore systems. Figure 1.4 presents semantics of commonly available atomic primitives.

```
TEST-AND-SET($addr$):
    $temp \leftarrow addr$
    $addr \leftarrow 1$
    return $temp$

FETCH-AND-ADD($addr, val$):
    $temp \leftarrow addr$
    $addr \leftarrow addr + val$
    return $temp$

COMPARE-AND-SWAP($addr, old, new$):
    if $addr = old$ then
        $addr \leftarrow new$
        return $true$
    else
        return $false$
```

```
                                        ▷ LL/SC pair
LOAD-LINKED($addr$):
    $temp \leftarrow addr$
    mark $addr$
    return $temp$

▷ called after LL
STORE-CONDITIONAL($addr, val$):
    if IsMarked $addr$ then
        $addr \leftarrow val$
        return $true$
    else
        return $false$
```

Figure 1.4: Pseudo-code definitions of common atomic primitives.

Architectures support and hardware implementation details may vary for the atomic primitives in Figure 1.4 in different platforms. Compare-and-Swap (`CAS`) and Load-Linked/Store-Conditional (`LL/SC`) are *universal primitives* (details in Section 1.2.3), thus, can be used to construct simulations of arbitrary atomic read-modify-write operations including those listed in Figure 1.4.

In contrast to `CAS`, `LL/SC` pair verifies that the contents of memory address have not been modified, instead of verifying if contents at the address match a given value. This has significant benefits for the programmer using `LL/SC` over `CAS` in avoiding the ABA problem. The ABA problem arises as a consequence of the fact that matching the contents at a target address with a given value does not imply that no changes have occurred at the address. A concurrent process may have changed the contents of the address from A to B and then back to A. In some implementations this behavior is tolerable, however, in others, it may lead to incorrect results.

1.2 Synchronization

The need for synchronization arises in every area where multiple entities have to agree or commit to a given set of steps. In shared-memory multicore processors, synchronization is used to explicitly dictate which process interleavings are acceptable and prohibit those that are unacceptable. Synchronization is generally achieved by guaranteeing some notion of *atomicity*; a given sequence of instructions executed by a single process appears instantaneous to other processes.

Synchronization mechanisms are required to be scalable and correct. Scalability in this context implies that the cost of synchronization should not rise with increasing number of processes or threads. Correctness requires that the synchronization mechanism satisfy both *safety* and *liveness* properties. Informally, a safety property states that "bad" things never happen, while, liveness property (progress condition) states that "good" things eventually happen [11]. Schneider *et al.* [18] more formally define both properties.

Examples of safety properties are: (1) *Mutual Exclusion*: in any execution, at most one process is in the critical section – "bad thing" happening is two or more processes executing in a critical section. (2) *Deadlock Freedom*: if a process attempts to enter a critical section, then eventually some process executes inside the critical section – "bad thing" happening is a deadlock.

Examples of liveness properties include: (1) *Starvation Freedom*: a process makes progress infinitely often – the "good thing" is making progress. (2) *Livelock freedom*: processes do not run forever without progress – the "good thing" is at least one process makes progress.

Note that "good thing" and "bad thing" are not well-defined concepts; therefore, some properties are an intersection of both safety and liveness [18]. Furthermore, without any notion of progress, a synchronization mechanism would be correct by halting all processes making it impractical. Similarly, liveness without safety is trivial but of little practical value.

1.2.1 Blocking Synchronization

Atomicity is most commonly achieved by mutual exclusion using critical sections guarded by locks. At any point in time, at most one process executes instructions inside the critical section. A process that requests for a lock held by another process will block until the lock is released (by busy waiting or yielding), thus guaranteeing that instructions executed in the critical section appear *atomic*. There are relaxations of this requirement, where multiple processes are allowed to hold a lock and execute inside the critical section as long as no modifications are performed (*ReadWrite* locks).

However, locks are *blocking* - arbitrary delay (scheduling preemption, page-faults or cache misses) or failure of a process holding a lock blocks other processes from making progress. Additionally, if a few locks are used to protect large portions of the program (*coarse-grained*), they prohibit many executions that could have correctly run in parallel, thereby reducing parallelism. *Fine-grained* locking is more difficult to implement correctly; thus, care must be taken to avoid well-known pitfalls associated with locks:

1. **Deadlocks**: Circular dependencies might arise where a process P_1 holding a lock L_1 blocks waiting for a lock L_2 held by another process P_2, while P_2 is also blocked waiting on P_1 to release L_1. Mechanisms to avoid deadlocks such as acquiring the locks in a specific order may have a significant impact on the synchronization overhead.

2. **Priority Inversion**: If processes share a processor with preemptive scheduling, a high priority process H_P may have to yield the processor to a low priority process L_P that holds a lock required by H_P. A middle priority process M_P that does not need the lock may preempt the low priority process - leading to a priority inversion between high priority and middle priority processes [19]. The system resets on the Pathfinder mission to Mars popularized priority inversion [20].

3. **Convoying**: If a process holding the lock is arbitrarily delayed inside the critical sections, other processes that wish to enter the critical section queue up waiting for the lock to be released. When the lock is released, the queued threads form a *convoy* as they gain exclusive access to the critical section [14, 21].

As a consequence of their blocking nature, liveness properties associated with lock-based synchronization are dependent on operating system scheduler [22]. Starvation-freedom and livelock-freedom are guaranteed only if no process executes inside the critical section forever, and a process that seeks to enter the critical section is granted access to the critical section unless another process is already in the critical section.

In an asynchronous shared-memory model, synchronization mechanisms are required to be resilient to arbitrary delays and failures of processes. Non-blocking synchronization does not rely on locks; therefore, the arbitrary delay or failure of any process does not cause delay or failure of other processes.

1.2.2 Non-blocking Synchronization

Non-blocking synchronization techniques guarantee atomicity without mutual exclusion; consequently, they are resilient to pitfalls associated with mutual exclusion. Additionally, they do not incur significant performance degradation due to arbitrary process delay.

Generally, non-blocking techniques are optimistic; each process attempts to execute independently or locally for as long as possible and publish their modifications using atomic instructions. An optimistic execution of a process can be invalidated by a concurrent modification, at which point, publication of the modifications will fail, and the process will have to repeat the local execution.

In the literature, there are several levels of non-blocking progress guarantees:

1. *Wait-freedom* [23] – guarantees that every process continues to make progress regardless of delays or failures of other processes. Wait-freedom guarantees individual progress; combines non-blocking progress with starvation freedom.

2. *Lock-freedom* [24] – guarantees that some process makes progress; ensures system-wide progress without starvation freedom.

3. *Obstruction-freedom* [25] – guarantees that a process will make progress if executed in isolation for long enough (*i.e.,* no interruptions from concurrent processes). It does not guarantee progress under contention; starvation and livelocks may happen if processes are executed concurrently.

Strong progress guarantees such as wait-freedom may be required for systems with real-time constraints and resiliency requirements. However, they are non-trivial to achieve efficiently, thus, on modern shared-memory multicore systems, weaker progress guarantees such as lock-freedom and obstruction-freedom generally suffice, and are easier to implement efficiently [26]. In contrast to wait-freedom and lock-free, obstruction-freedom is dependent on the scheduler.

1.2.3 Power of Synchronization Primitives

Herlihy [23] constructed an infinite hierarchy of shared objects based on their ability to solve a classical distributed systems problem in an asynchronous shared memory system with n processes: *wait-free consensus*. Consensus is a coordination problem in which n processes, each with an initial input value unknown to the others agree on a common output value – *decision*. The decision value must be one of the input values. A consensus protocol is wait-free if every process completes in a finite number of its steps.

A consensus number $C(\mathcal{O})$ of an object type $\mathcal{O}$, is the maximum number of processes for which wait-free consensus can be implemented using any number of objects of type $\mathcal{O}$ and read/write registers. Wait-free consensus cannot be implemented in an asynchronous system using read/write registers for more than one process [27, 28]. Thus, the consensus number for read/write registers is 1.

An object is considered universal in a system of n processes if its consensus number is at least n ($C(\mathcal{O}) \geq n$). Some consensus objects are universal in any arbitrary systems, thus, together with read/write registers, can be used to solve wait-free consensus for any number of processes, *e.g.,*`CAS` or `LL/SC`.

In the wait-free hierarchy, an object at level l has consensus number l; and together with atomic read/write registers can implement any wait-free consensus object at level l or lower, but not objects at higher levels. Universal consensus objects are desirable in hardware and fundamental to non-blocking synchronization: any arbitrary shared object can be implemented wait-free if the hardware or programming language supports wait-free universal consensus objects.

1.3 Concurrent Data Structures

Concurrent data structures are high-level abstractions for synchronized access to shared data. They are a fundamental component for building software systems to exploit parallelism available on multicore systems. Concurrent data structures are essentially adaptations of *abstract data types* (ADT) defined for sequential data structures to support concurrent operations.

Each object has a *sequential specification*, typically defined as a set of operations that can be executed on the object and a set of legal operation sequences. Each operation execution is specified by its pre- and post-conditions. A sequential object implementation is correct, if any operation called when pre-conditions are true, terminates with correct post-conditions, and operation sequence is a subset of legal operation sequences.

Reasoning about concurrent operations is challenging. Firstly, we require that operations appear atomic, regardless of the execution interval between the invocation and response of the operation. Secondly, when operations issued by distinct processes overlap in time, it may be unclear in what order the operations take effect.

1.3.1 Correctness of Concurrent Data Structures

Safety properties are used to specify the permissible order in which operations by concurrent processes appear to execute. There are various formalizations of this requirement such as Linearizability [12] and Sequential Consistency [13].

A *history* is a log of executions by concurrent processes. A history H is a finite or infinite sequence of operation invocations and responses. A history is sequential if; the sequence starts with an invocation and a matching response immediately follows each invocation. Not all histories are correct; a history is admissible or legal if it adheres to the object's sequential specification.

A history defines a partial order on the operations it includes. An operation op_1 happens before the operation op_2 in H (denoted: $op_1 \prec_H op_2$) if the response to op_1 precedes the invocation of op_2. Intuitively, the partial order imposes a notion of real-time ordering on the operations in H. Operations are considered *concurrent* if they are unrelated by partial order (neither $op_1 \prec_H op_2$ nor $op_2 \prec_H op_1$). A history H is concurrent if it contains at least one pair of concurrent operations.

Linearizability [12]: A concurrent object is linearizable if for every history H there exists a permutation $\mathcal{S}$ of all operations in H such that

1. $\mathcal{S}$ is sequential and observes the sequential specifications; and

2. For each $op_1 \prec_H op_2$ then $op_1 \prec_{\mathcal{S}} op_2$.

Thus, an execution is linearizable if there exists a sequential history $\mathcal{S}$ of operations in the execution that (1) respects the object's sequential specification, and (2) observes the *real-time* ordering of events at all processes. $\mathcal{S}$ is referred to as a *linearization* of H.

The sequence $\mathcal{S}$ may include a subset of pending operations. An operation may complete all modifications on an object but take arbitrarily long to return a response. Other operations on the object will observe the effects of the operation that is yet to return a response. Therefore, by definition of linearizability, there exists a single instant in time –linearization point, between invocation and response where an operation appears to take effect.

A conventional approach to show that a concurrent execution is linearizable is to define a linearization point for every operation in the execution history. Intuitively, the order induced by a sequence of linearization points preserves the real-time ordering of non-overlapping operations.

Additionally, linearizability is composable; a composition of linearizable histories is linearizable. This property is fundamental; concurrent objects can be designed, verified and implemented independently then combined to make a larger object.

Linearizability strictly requires operations to observe real-time order for all processes. However, in some cases, the real-time order of events at different processes may not be significant. ***Sequential Consistency*** is a correctness condition that exploits this relaxation in precedence ordering of operations [13]. Formally, an execution is sequentially consistent if there exists a permutation $\mathcal{S}$ of all operations in a concurrent history H such that:

1. $\mathcal{S}$ is sequential and observes the sequential specifications; and

2. For each $op_1 \prec_{H|p_i} op_2$ then $op_1 \prec_{\mathcal{S}|p_i} op_2$.

Essentially, instead of preserving the real-time behavior of non-overlapping operations at all processes, it only preserves the program order of operations issued by the same process. Sequential consistency is a weaker condition than linearizability; every linearizable sequence is also sequentially consistent, but the reverse is not true. Additionally, sequential consistency is not composable. In order to enhance parallelism and performance, several relaxations of correctness conditions for concurrent data structures have been proposed [29–34].

The choice of synchronization mechanism to satisfy the safety properties of a concurrent data structure generally dictates the liveness or progress conditions associated with the data structure operations. Correctness can trivially be achieved using a single lock to guard the entire data structure (coarse-grained locking); however, this restricts parallelism and may lead to performance degradation. Multiple distinct locks can protect different portions of the data structure (fine-grained locking) leading to more efficient exploitation of available parallelism.

1.4 Non-blocking Concurrent Data Structures

Herlihy [23] presented general techniques for constructing a wait-free implementation of any sequential object (generally referred to as universal constructions). However, this approach is costly for practical purposes; the method requires copying the entire data structure for some operations and does not allow concurrent updates to the data structure. Consequently, tremendous effort has been made to construct more efficient and practical non-blocking data structures by exploiting specific semantics of the data structures [35–38].

A non-blocking update operation typically involves reading a memory location, taking steps *locally* based on the value returned by the read, and then modify the memory location using *read-modify-write* atomic primitives such as `CAS` (which compares the previously read value to the current value and only executes the modification if these are the same). The `CAS` only fails due to a concurrent modification[4], in which case the steps are restarted. The execution time of a single operation cannot bounded as it may fail and retry arbitrarily.

Many concurrent data structures are represented as *linked* structures. Processes access the data structure through shared pointer variables to specific nodes in the link. The non-blocking concurrent queue by Michael *et al.* [39] is a classic example of this representation; processes access the link structure through either the *head* or *tail* pointers. Local steps start with allocating a new node and then entering a retry loop in which modifications are made to the new node based on the current state of the data structure, and an optimistic attempt is made to add the new node to the data structure if the a priori state as has not changed.

Memory allocation performed with linked structures results in significant memory management overheads. There are several memory management schemes proposed for programming environments without automatic garbage collection [40–47].

1.4.1 Design and Implementation Approaches

(A) Helping

In various implementations of non-blocking data structures [39,48–50], a modify operation may require more than a single atomic step to complete or to update multiple memory locations. Consequently, concurrent operations may observe the shared data structure in an inconsistent transient state. This inconsistency may

[4] Success does not guarantee that there were no concurrent modifications as we explained in ABA problem (Section 1.1.2).

(1) block other operations from making progress, and (2) cost other operations extra steps (*e.g.,* traversing nodes that are logically deleted).

Operations that observe the inconsistency, *help* complete the operation so that either they can ensure progress or that other operations do not pay that cost associated with the inconsistency. Helping was initially conceived for achieving wait-freedom [23]; it is also used in several lock-free implementations to coordinate access to shared data structures [39, 49, 51]. Censor-Hillel *et al.* [52] proposed a formal definition of *help* in wait-free algorithms based on linearization order: a process p helps an operation op_2 by another process q if a step by p determines that op_2 is linearized before some other operation.

In many cases, helping necessitates that the helper has information about the operation that requires the help. *Descriptor* objects introduced in the cooperative technique by Barnes [51] contain sufficient information on a pending operation to allow concurrent threads to complete the operation. The main idea is to detach operations from the executing threads. Thus, if a thread t_1 reads a descriptor allocated by another thread t_2, thread t_1 can complete the pending operation ensuring system-wide progress. This approach is akin to *locking*, except that, instead of a thread owning a lock, the lock belongs to an operation. Thus, the delay of a thread does not block other threads from completing the operation.

Harris [48] introduced the idea of pointer-marking as a pragmatic solution for operations that require more than one atomic primitive to complete correctly. In his approach, a thread declares its intention to delete a node by marking the next pointer of the node to prevent concurrent operations from modifying the pointer. Then physically deleting the node from the list after.

Challenges: Helping may result in performance degradation for the data structure especially when read-only operations such as CONTAINS are compelled to perform writes during helping. These performance concerns have resulted in designs where read-only operations traverse data structure nodes while ignoring concurrent operations that would otherwise have solicited for help [50, 53–58].

Furthermore, pointer marking requires an atomically markable reference, which is expensive or unavailable in some programming languages. A typical mechanism for pointer marking involves utilizing some of the bits in the pointer value for distinguishing the pointer from the marked version of the pointer. Although this is an elegant and practical approach in programming languages such as C/C++, it is not portable to languages where the developer has no access to object references or pointers. Java introduced *AtomicMarkableReference* which maintains an object reference along with a mark bit, however, this is expensive as it utilizes internal objects which create an extra level of dereferencing. As critical components for exploiting multicore processors, designs of concurrent data structures have to be portable; should not rely on architecture or language specific constructs.

(B) Synchronization Bottlenecks

If many threads attempt to modify a shared variable simultaneously, the resulting memory contention may lead to performance degradation, and the shared variable becomes a *hotspot* [59, 60]. The shared variable could be a lock for blocking synchronization or a global variable modified by atomic primitives in non-blocking synchronization [39, 61, 62]. Performance degradation is largely a result of cache invalidation as processes read and modify the shared variable.

Traditionally, *hotspots* in both blocking and non-blocking concurrent data structures are addressed by utilizing fine-grained synchronization. This approach fits naturally to data structures that allow concurrent access to different elements in the structure: bags [63], hash tables [64, 65], linked-lists [49, 57], skip-lists [53, 66] and search trees [50, 54–56]. However, some data structures have inherent synchronization bottlenecks where contention cannot be trivially managed with fine-grained synchronization: queues [39,67], priority-queues [68, 69], and stacks [34, 61, 70]. A well-known approach to reducing contention is *backoff* [9,59,70,71].

Another popular technique to reduce synchronization bottlenecks is for a single process to combine and execute requests on behalf of other processes. Combining has several benefits: 1) eliminate memory contention due to "hotspot" shared variables, and 2) improved cache locality for the combiner thread. Combining was first introduced in software combining trees [72], where requests starting at the leaves of a static tree are combined up the tree. However, static combining trees imply that operations suffer significant synchronization overheads regardless of the level of contention. Combining funnels proposed by Shavit *et al.* [73] reduce the overheads by employing dynamic trees.

Oyama *et al.* [74] proposed a different approach to managing potential synchronization bottlenecks. A shared data structure is protected by a single lock and processes wishing to access the data structure announce their requests in a LIFO list structure. A process that acquires the lock executes in addition to its request, pending requests by other processes and then discards the list.

Hendler *et al.* introduced *flat-combining* [75]; in contrast to Oyama *et al.*, the announcement list typically contains one record per concurrent thread accessing it. If a thread accesses the list for the first time, it adds a record entry to the list which it uses to publish subsequent access requests. After writing each request, the thread attempts to acquire the global lock. A thread that acquires the lock (combiner) scans the list for pending requests, applies them to the underlying data structure, and then writes responses back to the associated records in the list.

Challenges: Combining as well as flat-combining techniques are typically lock-based, and consequently blocking. Fatourou and Kallimanis presented universal constructions that use a `FAA` and an `LL/SC` object for implementing combining technique with wait-free progress guarantees [76]. The idea is to have a process that wants to execute an operation, find out which operations have been announced, apply these operations to a local copy of the object, then finally change the global object pointer to refer to this local copy. However, these constructions do not cope well with large shared objects as they copy the state of the object, then apply changes to the local copy. Fatourou *et al.* [77] presented algorithms for efficiently handling shared objects with large state size, however, these approaches include dedicated threads that apply updates to the data structure. Consequently, compromising the non-blocking progress guarantees in [76]. Therefore, it remains an open challenge to develop mechanisms for reducing synchronization overheads in concurrent data structures with inherent bottlenecks that cannot be trivially managed with fine-grained synchronization.

1.4.2 Concurrent Data Structures for Efficient Data Stream Processing

Interest in Data Stream Processing is a result of the unprecedented increase in volumes of data generated at high-rates that need to be processed in real-time. Stream Processing Engines (SPEs) are generally modeled as directed graphs where vertices are processing operators, and the edges are continuous streams of data between the operators. For example StreamCloud [78], Apache Storm [79], Apache Flink [80] and Saber [81]. *Parallelism* in SPEs is paramount for achieving *high-throughput* and *low latency* processing.

Pipeline and task parallelism are extracted naturally from the directed graphs with independent operators or tasks assigned to different processing units. However, data parallelization or fission [82–86], which involves replicating instances of operators careful orchestration of operators' execution is required to preserve *determinism*. Determinism is required to ensure consistent results independently of the way in which the analysis is parallelized.

Additionally, power consumption has gained significance as a metric for evaluating computing systems. Consequently, there is a growing trend towards efficient utilization of both general-purpose multicore architectures and low-power embedded multicore devices. Benefits of utilizing low-power embedded devices are two-fold: (1) can be deployed as edge and fog devices for close-to-the-source analysis minimizing latency for time-critical applications, (2) they often provide increased performance per watt in comparison with traditional general-purpose multicore servers.

Challenges: Attempts to guarantee determinism in SPEs under execution of parallel instances of an operator rely on dedicated merge-sorting operators. These operators are either added to continuous queries by query compilers [78, 84, 85] or left for developers to place within their streaming applications in SPEs, such as Apache Storm [79]. Minimizing the computational overhead introduced by such dedicated operators is challenging, especially for one-at-a-time, fine-grained low latency tuple processing.

Additionally, deploying dedicated merge-sorting operators in-between the query operators results in a higher number of threads in SPEs such as Storm [79] or Flink [80] or in scheduling overheads for SPEs with schedulers [78, 87, 88] ordering operators' execution, thus degrading throughput and increasing energy consumption. In many cases, the merge-sorting operators become a bottleneck reducing the benefits of parallelizing query operators.

With a growing interest in deploying stream processing applications at the cloud, fog, and edge architectures in order to satisfy different application constraints, we need to be able to port these applications to various platforms efficiently. However, modern embedded systems provide different characteristics and features (such as memory hierarchy, data movement options, OS support, *etc.*) depending on the application domain that they target. The impact of each one of these features on performance and energy consumption of the whole system, when running a specific application, is often hard to predict at design time. This problem becomes even harder when developers attempt to improve more than one metric simultaneously.

1.5 Contributions

The contributions of this thesis are solutions to challenges in developing efficient and practical concurrent data structures. Additionally, we explore the utilization of concurrent data structures for efficient data stream processing in both low-power embedded and general-purpose multicore systems.

Data Structures with Potential Synchronization Bottlenecks (Chapter 2 and 3)

Dynamic vectors are among the most commonly used data structures in applications development. They provide constant time random access and resizable data storage. Additionally, they provide constant time insertion (PUSHBACK) and deletion (POPBACK) at the end of the sequence. However, concurrent PUSHBACK and POPBACK are required to atomically modify the vector storage and the size variable, creating a synchronization bottleneck. Moreover, bounds checking as required for random-access operations further increase the memory contention on the size variable of the vector.

In Chapter 2, we present a lock-free vector that addresses potential synchronization bottlenecks by utilizing the combining technique. The design utilizes the combining technique without compromising the lock-free progress guarantee and correctness of the data structure. Combining in this context implies that a given thread completes operations on behalf of other threads, and, the size variable is only updated after executing a batch of tail PUSHBACK and POPBACK operations.

In Chapter 3, we extend the concurrent lock-free vector to implement an unbounded heap-based priority queue. Priority queues are a fundamental data structure used as a base component in many applications that require priority ordering or scheduling. Priority queues are generally difficult to parallelize as the element with the highest priority creates a bottleneck. Furthermore, many applications, such as Dijkstra's single-source-shortest-path and Adaptive Huffman Trees require changing the priorities of items at runtime. We present the first concurrent lock-free unbounded binary heap that implements a priority queue with mutable priorities. The operations are provably linearizable. We also designed an optimized version of the algorithm by combining concurrent operations that substantially improves the performance.

Lock-free Concurrent Search Data Structures (Chapter 4 and 5)

An optimized helping strategy improves the overall performance of a lock-free algorithm. In Chapter 4, we propose *help-optimality*, which essentially implies that no operation step is accounted for exclusive helping in the lock-free synchronization of concurrent operations. As a rationale behind the term help-optimality, we would like to underline our aim to optimize a lock-free design with respect to the number of (CAS execution) steps incurred in helping under the constraints such as an optimal memory footprint and an optimal amortized step complexity.

To describe the concept, we revisit the designs of a lock-free linked-list and a lock-free binary search tree and present improved algorithms. Our algorithms employ atomic single-word compare-and-swap (CAS) primitives, are linearizable, and do not rely on any language/platform specific mechanism. Concretely, we use neither bit-stealing from a pointer nor runtime type introspection of objects. Therefore, our algorithms are *portable*.

In Chapter 5, we extend concurrent search to multi-dimensional data and similarity search. The popularity of in-memory databases has led to a significant interest in the index structures that can support Nearest Neighbor Search with dynamic concurrent addition and removal of data. We design the first concurrent lock-free multidimensional data structure that supports Nearest Neighbor Search (NNSEARCH) and ensuring the linearizability of NNSEARCH. LockFree-kD-tree (LFkD-tree): a lock-free concurrent kD-tree, which implements an abstract data type (ADT) that provides the operations ADD, REMOVE, CONTAINS, and NNSEARCH. Our algorithm for NNSEARCH is generic and can be adapted to other multidimensional data structures.

Concurrent Data Structures for Efficient Data Stream Processing (Chapter 6 and 7)

In Chapter 6, we propose a step-by-step exploration methodology for the customization of streaming aggregation implemented in embedded systems. The methodology is based on: (1) the identification of the parameters of the streaming aggregation operator that affect the evaluation metrics, and (2) the identification of the embedded platform-specific features that affect the evaluation metrics when executing streaming aggregation. These parameters compose a design space. The methodology provides a set of implementation solutions. For each solution, the application and the platform parameters have different values. In other words, each customized streaming aggregation implementation is tuned differently, so it provides different results for each evaluation metric. Developers can perform trade-offs between metrics, by selecting different customized implementations. Thus, instead of evaluating solutions in an ad-hoc manner, the proposed approach provides a systematic way to explore the design space.

Furthermore, motivated by the observation that the deterministic execution of streaming operators requires expensive synchronization to merge-sort streams delivered by multiple operator instances (or data sources), in Chapter 7, we study the limitations of operator-layer parallelism and how communication-layer determinism can overcome these. We propose Viper, a module that encapsulates and reduces the synchronization costs, enabling deterministic execution to be provided transparently in the communication layer of an SPE. We provide evidence that such a module can be leveraged by SPEs, by integrating it into Apache Storm, a representative SPE of one-at-a-time analysis paradigm, for low latency processing. Our evaluation shows that, with Viper, the throughput of parallel operators increases by up to 70% and at half of the energy consumption.

Bibliography

[1] Gordon E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 86, no. 1, pp. 82–85, 1998.

[2] Robert H. Dennard, Fritz H. Gaensslen, Yu Hwa-Nien, V. Leo Rideovt, E. Bassous, and Andre R. Leblanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[3] Robert H. B. Netzer and Barton P. Miller, "What are race conditions?: Some issues and formalizations," *ACM Letters on Programming Languages and Systems*, vol. 1, no. 1, pp. 74–88, 1992.

[4] Edsger W. Dijkstra, "Solution of a problem in concurrent programming control," *Communications of the ACM*, vol. 8, no. 9, pp. 569, 1965.

[5] Edsger W. Dijkstra, "The structure of the "the"-multiprogramming system," *Communications of the ACM*, vol. 11, no. 5, pp. 341–346, 1968.

[6] Daniel Cederman, Bapi Chatterjee, Nhan Nguyen, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas, "A study of the behavior of synchronization methods in commonly used languages and systems," in *Proceedings of the International Parallel and Distributed Processing Symposium*. 2013, pp. 1309–1320, IEEE.

[7] Vincent Gramoli, "More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2015, pp. 1–10, ACM.

[8] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 2015, pp. 631–644, ACM.

[9] Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas, "Analyzing the performance of lock-free data structures: A conflict-based model," in *Proceedings of the International Symposium on Distributed Computing*. 2015, pp. 341–355, Springer.

[10] Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas, "How Lock-free Data Structures Perform in Dynamic Environments: Models and Analyses," in *Proceedings of the International Conference on Principles of Distributed Systems*. 2017, pp. 23:1–23:17, Schloss Dagstuhl.

[11] Leslie Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, vol. 3, no. 2, pp. 125–143, 1977.

[12] Maurice Herlihy and Jeannette M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[13] Leslie Lamport, "How to make a correct multiprocess program execute correctly on a multiprocessor," *IEEE Transactions on Computers*, vol. 46, no. 7, pp. 779–782, 1997.

[14] Maurice Herlihy and Nir Shavit, *The art of multiprocessor programming*, Morgan Kaufmann, 2011.

[15] John L. Hennessy and David A. Patterson, *Computer architecture: a quantitative approach*, Elsevier, 2011.

[16] Leslie Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, 1979.

[17] Sarita V. Adve and Kourosh Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[18] Bowen Alpern and Fred B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, no. 4, pp. 181–185, 1985.

[19] Butler W. Lampson and David D. Redell, "Experience with processes and monitors in mesa," *Communications of the ACM*, vol. 23, no. 2, pp. 105–117, 1980.

[20] Mike Jones, "What really happened on mars rover pathfinder," *The Risks Digest*, vol. 19, no. 49, pp. 1–2, 1997.

[21] Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price, "The convoy phenomenon," *ACM SIGOPS Operating Systems Review*, vol. 13, no. 2, pp. 20–25, 1979.

[22] Maurice Herlihy and Nir Shavit, "On the nature of progress," in *Proceedings of the International Conference on Principles of Distributed Systems*. 2011, pp. 313–328, Springer.

[23] Maurice Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 1, pp. 124–149, 1991.

[24] Maurice Herlihy, "A methodology for implementing highly concurrent data objects," *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 5, pp. 745–770, 1993.

[25] Maurice Herlihy, Victor Luchangco, and Mark Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *Proceedings of the IEEE International Conference on Distributed Computing Systems*. 2003, pp. 522–529, IEEE.

[26] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit, "Are lock-free concurrent algorithms practically wait-free?," *Journal of the ACM*, vol. 63, no. 4, pp. 31:1–31:20, 2016.

[27] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[28] Michael C. Loui and Hosame H. Abu-Amara, *Memory requirements for agreement among unreliable asynchronous processes*, vol. 4, pp. 163–183, JAI press, 1987.

[29] James Aspnes, Maurice Herlihy, and Nir Shavit, "Counting networks," *Journal of the ACM*, vol. 41, no. 5, pp. 1020–1048, 1994.

[30] Nir Shavit and Asaph Zemach, "Diffracting trees," *ACM Transactions on Computer Systems*, vol. 14, no. 4, pp. 385–428, 1996.

[31] Yehuda Afek, Guy Korland, and Eitan Yanovsky, "Quasi-linearizability: Relaxed consistency for improved concurrency," in *Proceedings of the International Conference on Principles of Distributed Systems*. 2010, pp. 395–410, Springer.

[32] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas, "The lock-free k-lsm relaxed priority queue," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2015, pp. 277–278, ACM.

[33] Nir Shavit and Gadi Taubenfeld, "The computability of relaxed data structures: Queues and stacks as examples," *Distributed Computing*, vol. 29, no. 5, pp. 395–407, 2016.

[34] Adones Rukundo, Aras Atalar, and Philippas Tsigas, "Brief announcement: 2d-stack – a scalable lock-free stack design that continuously relaxes semantics for better performance," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 2018, pp. 407–409, ACM.

[35] Doug Lea, *Concurrent programming in Java: design principles and patterns*, Addison-Wesley Professional, 2000.

[36] Mark Moir and Nir Shavit, "Concurrent data structures.," 2004.

[37] Maurice Herlihy and Nir Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann Publishers Inc., 2008.

[38] Daniel Cederman, Anders Gidenstam, Phuong Ha, Håkan Sundell, Marina Papatriantafilou, and Philippas Tsigas, "Lock-free concurrent data structures," *Programming Multicore and Many-core Computing Systems*, vol. 86, pp. 59, 2017.

[39] Maged M. Michael and Michael L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 1996, pp. 267–275, ACM.

[40] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele, Jr., "Lock-free reference counting," *Distributed Computing*, vol. 15, no. 4, pp. 255–271, 2002.

[41] Maurice Herlihy, Victor Luchangco, and Mark Moir, "The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures," in *Proceedings of the International Conference on Distributed Computing*. 2002, pp. 339–353, Springer.

[42] Maged M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, 2004.

[43] Keir Fraser, "Practical lock-freedom," *PhD thesis, University of Cambridge*, 2004.

[44] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir, "Nonblocking memory management support for dynamic-sized data structures," *ACM Transactions on Computer Systems*, vol. 23, no. 2, pp. 146–196, 2005.

[45] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole, "Performance of memory reclamation for lockless synchronization," *Journal of Parallel and Distributed Computing*, vol. 67, no. 12, pp. 1270–1285, 2007.

[46] Anders Gidenstam, Marina Papatriantafilou, Hakan Sundell, and Philippas Tsigas, "Efficient and reliable lock-free memory reclamation based on reference counting," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 8, pp. 1173–1187, 2009.

[47] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas, "Nbmalloc: Allocating memory in a lock-free manner," *Algorithmica*, vol. 58, no. 2, pp. 304–338, 2010.

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

[49] Timothy L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the International Conference on Distributed Computing*. 2001, pp. 300–314, Springer.

[50] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel, "Non-blocking binary search trees," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 2010, pp. 131–140, ACM.

[51] Greg Barnes, "A method for implementing lock-free shared-data structures," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 1993, pp. 261–270, ACM.

[52] Keren Censor-Hillel, Erez Petrank, and Shahar Timnat, "Help!," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 2015, pp. 241–250, ACM.

[53] Håkan Sundell and Philippas Tsigas, “Fast and lock-free concurrent priority queues for multi-thread systems,” *Journal of Parallel and Distributed Computing*, vol. 65, no. 5, pp. 609–627, 2005.

[54] Shane V. Howley and Jeremy Jones, “A non-blocking internal binary search tree,” in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2012, pp. 161–171, ACM.

[55] Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas, “Efficient lock-free binary search trees,” in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 2014, pp. 322–331, ACM.

[56] Aravind Natarajan and Neeraj Mittal, “Fast concurrent lock-free binary search trees,” in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2014, pp. 317–328, ACM.

[57] Mikhail Fomitchev and Eric Ruppert, “Lock-free linked lists and skip lists,” in *Proceedings of the Symposium on Principles of Distributed Computing*. 2004, pp. 50–59, ACM.

[58] Joel Gibson and Vincent Gramoli, “Why non-blocking operations should be selfish,” in *Distributed Computing*. 2015, pp. 200–214, Springer.

[59] Anant Agarwal and Mathews Cherian, “Adaptive backoff synchronization techniques,” in *Proceedings of the 16th Annual International Symposium on Computer Architecture*. 1989, pp. 396–406, ACM.

[60] Maurice Herlihy, Beng-Hong Lim, and Nir Shavit, “Scalable concurrent counting,” *ACM Transactions on Computer Systems*, vol. 13, no. 4, pp. 343–364, 1995.

[61] R Kent Treiber, *Systems programming: Coping with parallelism*, International Business Machines Incorporated, Thomas J. Watson Research Center New York, 1986.

[62] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup, “Lock-free dynamically resizable arrays,” in *Proceedings of the International Conference on Principles of Distributed Systems*. 2006, pp. 142–156, Springer.

[63] Håkan Sundell, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas, “A lock-free algorithm for concurrent bags,” in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2011, pp. 335–344, ACM.

[64] Nhan Nguyen and Philippas Tsigas, “Lock-free cuckoo hashing,” in *Proceedings of the IEEE International Conference on Distributed Computing Systems*. 2014, pp. 627–636, IEEE.

[65] Maged M. Michael, “High performance dynamic lock-free hash tables and list-based sets,” in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2002, pp. 73–82, ACM.

[66] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit, “A simple optimistic skiplist algorithm,” in *Proceedings of the International Conference on Structural Information and Communication Complexity*. 2007, pp. 124–138, Springer.

[67] Anders Gidenstam, Håkan Sundell, and Philippas Tsigas, “Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency,” in *Proceedings of the International Conference on Principles of Distributed Systems*. 2010, pp. 302–317, Springer.

[68] Nir Shavit and Itay Lotan, “Skiplist-based concurrent priority queues,” in *Proceedings of the International Parallel and Distributed Processing Symposium*. 2000, pp. 263–268, IEEE.

[69] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas, “The lock-free k-lsm relaxed priority queue,” in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2015, pp. 277–278, ACM.

[70] Danny Hendler, Nir Shavit, and Lena Yerushalmi, “A scalable lock-free stack algorithm,” in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2004, pp. 206–215, ACM.

[71] John M. Mellor-Crummey and Michael L. Scott, “Algorithms for scalable synchronization on shared-memory multiprocessors,” *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21–65, 1991.

[72] Pen-Chung Yew, Nian-Feng Tzeng, and Lawrie, “Distributing hot-spot addressing in large-scale multiprocessors,” *IEEE Transactions on Computers*, vol. C-36, no. 4, pp. 388–395, 1987.

[73] Nir Shavit and Asaph Zemach, “Combining funnels: A dynamic approach to software combining,” *Journal of Parallel and Distributed Computing*, vol. 60, no. 11, pp. 1355–1387, 2000.

[74] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa, “Executing parallel programs with synchronization bottlenecks efficiently,” in *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*. 1999, vol. 16, Citeseer.

[75] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir, “Flat combining and the synchronization-parallelism tradeoff,” in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2010, pp. 355–364, ACM.

[76] Panagiota Fatourou and Nikolaos D. Kallimanis, “A highly-efficient wait-free universal construction,” in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2011, pp. 325–334, ACM.

[77] Panagiota Fatourou and Nikolaos D. Kallimanis, “Revisiting the combining synchronization technique,” in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2012, pp. 257–266, ACM.

[78] Vincenzo Gulisano, *StreamCloud: An Elastic Parallel-Distributed Stream Processing Engine*, Ph.D. thesis, Universidad Politécnica de Madrid, 2012.

[79] “Apache Storm,” http://storm.apache.org/, 2017.

[80] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas, "Apache flink: Stream and batch processing in a single engine," *IEEE Data Engineering Bulletin*, vol. 36, no. 4, 2015.

[81] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2016, pp. 555–569, ACM.

[82] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm, "A catalog of stream processing optimizations," *ACM Computing Surveys*, vol. 46, no. 4, pp. 46:1–46:34, 2014.

[83] Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kun-Lung Wu, "Elastic scaling of data parallel operators in stream processing," in *Proceedings of the International Parallel and Distributed Processing Symposium*. 2009, pp. 1–12, IEEE.

[84] Scott Schneider, Martin Hirzel, Bugra Gedik, and Kun-Lung Wu, "Auto-parallelizing stateful distributed streaming applications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 2012, pp. 53–64, ACM.

[85] Scott Schneider, Martin Hirzel, Bugra Gedik, and Kun-Lung Wu, "Safe data parallelism for general streaming," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 504–517, 2015.

[86] Sai Wu, Vibhore Kumar, Kun-Lung Wu, and Beng Chin Ooi, "Parallelizing stateful operators in a distributed stream processing system: How, should you and how much?," in *Proceedings of the International Conference on Distributed Event-Based Systems*. 2012, pp. 278–289, ACM.

[87] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik, "Aurora: a new model and architecture for data stream management," *The International Journal on Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, 2003.

[88] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik, "The design of the borealis stream processing engine.," in *Conference on Innovative Data Systems Research*, 2005, vol. 5, pp. 277–289.

PAPER I

Ivan Walulya and Philippas Tsigas

Scalable Lock-Free Vector with Combining

In the Proceedings of the
31^{st} *IEEE International Parallel and Distributed Processing Symposium*
pp. 917-926, IEEE 2017.

2

Scalable Lock-Free Vector with Combining

Abstract
Dynamic vectors are among the most commonly used data structures in programming. They provide constant time random access and resizable data storage. Additionally, they provide constant time insertion (PUSHBACK) and deletion (POPBACK) at the end of the sequence. However, in a multithreaded system, concurrent PUSHBACK and POPBACK operations attempt to update the same shared object, creating a synchronization bottleneck.

In this paper, we present a lock-free vector design that efficiently addresses the synchronization bottlenecks by utilizing a combining technique on PUSHBACK and POPBACK operations. Typical combining techniques compromise progress guarantees. Our design introduces combining without sacrificing lock-freedom. We evaluate the performance of our design on a dual socket NUMA Intel server. The results show that our design performs comparably at low contention, and outperforms prior concurrent blocking and non-blocking vector implementations at high contention, by as much as 2.7x.

2.1 Introduction

A vector is a fundamental abstract data type that similar to an array, stores a linear sequence of arbitrary objects. Elements in the vector can be randomly accessed using a non-negative integer referred to as an index. In contrast to an array, the vector data structure can grow and shrink in size as elements are added or removed after its initialization. In addition to random access, the vector also provides *tail* operations: PUSHBACK which appends an element to the tail end of the vector, and POPBACK that removes an element from the end the vector.

Implementations of the vector data structure such as the Java Collections vector and the C++ STL vector are used in a myriad of applications. These vector implementations do not support concurrency nor thread-safety. However, multicores/multiprocessors have become ubiquitous; proliferating into all spectra of computing systems ranging from ultra-low power embedded systems to supercomputers. Furthermore, in order to scale to hundreds of cores, architecture designers are opting for Non-Uniform Memory Access (NUMA) [1].

This growth in the prevalence of multicores and shift to NUMA system design poses new challenges for application developers and increases the importance of scalable implementations of concurrent data structures. Concurrent data structures allow for multiple processes to access and modify overlapping regions of the data structure at the same time, resulting in high performance on multicore systems. In addition to improved performance, concurrent data structures are often characterized by their safety and liveness properties (*blocking* or *non-blocking*).

Synchronization of concurrent access to a shared object is commonly achieved using mutual exclusion locks. A lock guarantees that, while a process holds the lock, no other process can access the protected object. Exclusive access can significantly diminish parallelism, and if a process stalls while holding a lock, it may cause other processes to wait for arbitrarily long periods of time (*blocking*). Moreover, process inter-dependence of lock-based implementations introduces vulnerabilities such as deadlocks, livelocks, priority inversions and convoying [2].

Non-blocking techniques have emerged to address both scalability and progress pitfalls associated with the utilizations of locks. Non-blocking data structures provide three widely accepted levels of *progress guarantees*: (1) obstruction-freedom: any process that executes in isolation will complete an arbitrary operation in a finite number of steps [3]. (2) lock-freedom: at least one process will complete its operation in a finite number of steps [4]. (3) wait-freedom: every process will complete its operation in a finite number of steps [4].

Although several universal constructions exist for implementing concurrent data structures from sequential equivalents [2, 5–7], the vector which is ubiquitous in sequential libraries has received little attention. Intel TBB library [8] provides a lock-based vector implementation without POPBACK tail operation. Dechev *et al.* proposed the first lock-free vector implementations in [9]. Feldman *et al.* proposed a wait-free vector design that utilizes the fast-path-slow-path technique [10].

However, non-blocking as well as blocking vector implementations do not scale at high thread[1] count due to the contention from conflicting PUSHBACK and POPBACK tail operations. In these implementations, all concurrent PUSHBACK and POPBACK operations synchronize on a single point which becomes a hot spot, resulting in sequentialized execution.

Combining is an efficient technique for reducing contention on the data structure [6, 11–13]. In this technique, multiple operations are batched together and executed by only one process on the data structure. A process that requires to execute an operation, announces the operation by placing a request on a list, then tries to get ownership of a global lock. The process that manages to acquire the lock handles in addition to its own operation, operations announced by other processes. This technique has been utilized to design common concurrent data structures such as queues, priority queues, stacks [12, 14] . The combining based implementations have been shown experimentally to outperform their fine-grained locking counterparts. However, these implementations are blocking; while a process holds the combiner lock, other processes wait for the process to either release the lock or complete the operation requests.

In this paper, we present, to the best of our knowledge, the first non-blocking concurrent vector design with combining. The design utilizes the combining technique without compromising the lock-free progress guarantee and linearizability provided by the data structure. We evaluated the performance (throughput in operations per second) of our design against prior vector implementations on a dual socket Intel server with 16 logical cores and hyper-threading enabled. The empirical results show that our vector performs comparably with prior implementations at low-levels of thread contention, and outperforms as we increase the number of threads. The results also show that our vector scales well across sockets of the NUMA server.

[1] the terms ”process” and ”thread” are used interchangeably in the paper to refer to a unit of execution in a shared address space.

2.1.1 Related Work:

Dechev *et al.* presented LFvector – the first lock-free vector design and implementation in the literature [9, 15]. The design utilizes a shared descriptor object to synchronize concurrent PUSHBACK and POPBACK operations. As these operations require atomically updating the vector size and data array, a thread must first acquire the shared descriptor object. Concurrent threads can help complete the operation associated with the descriptor. Random access operations are wait-free since they proceed without modifying the shared descriptor object. However, these operations have no bounds checking mechanism, which is left up to the applications developer.

Feldman *et al.* presented WFvector – a wait-free vector implementation that stores elements contiguously in memory [10]. The algorithm also exploits descriptor objects to indicate that an operation is in progress. In contrast to *LFvector*, this approach does not modify a single global descriptor value, instead, places descriptors at the last element and tail positions, then modifies the vector after acquiring both positions. Resizing the vector involves copying over elements from the old to a new object to allow for contiguous storage. Wait-freedom is achieved using the fast-path slow-path technique [16]. Additionally, the implementation includes versions of PUSHBACK and POPBACK (FAAvector) that are efficient when executed without inverse operations, *i.e.,*PUSHBACK operations executed without concurrent POPBACK.

WFvector, as well as LFvector, do not scale with increasing concurrency as tail operations (PUSHBACK, POPBACK) are serialized when threads attempt to either modify the shared descriptor value or write descriptors at the end of the vector. Thereby, create a synchronization bottleneck. Our empirical evaluation shows that; the loss of scalability is exacerbated as we increase concurrency across multiple sockets of a NUMA architecture.

Previous attempts to improve on the efficiency of concurrent algorithms with potential synchronization bottlenecks started with combining trees [17]. The general approach is to have a single thread perform operations announced by other threads. Combining trees exhibited significant synchronization overheads which were improved upon by Shavit [18] with combining funnels that use dynamically built trees of combined operations.

Oyama *et al.* [13] presented a lock-based combining technique with a list of announcements implemented using a stack. Therefore, announcements are processed in LIFO order and then removed from the list. A combiner has to serve all requests that are announced during its execution, which can lead to starvation. This approach also experiences significant overheads as each thread has to repeatedly perform CAS operations until it successfully adds an announcement onto the stack.

Hendler *et al.* introduced a coarse grained locking technique referred to as *flat-combining* [11]. In contrast to Oyama *et al.*, the announcement list typically contains one record per concurrent thread accessing it. If a thread accesses the list for the first time, it adds a record entry to the list which it uses to publish subsequent access requests. After writing each request, the thread attempts to acquire the global lock. A thread that acquires the lock (combiner), scans the list for pending requests, applies them to the underlying data structure, and then writes responses back to the associated records in the list. Threads spin as they wait for the combiner to complete their pending requests or release the lock. Predefined thread records imply that the combiner has to traverse the entire list regardless of whether it contains active requests. Furthermore, similar to Oyama *et al.*, flat-combining is blocking (a thread preempted while holding a lock causes all others to wait) and suffers from pitfalls associated with locks.

Fatourou and Kallimanis presented universal constructions that uses a FAA and an LL/SC object for implementing combining technique with wait-free progress guarantees [19]. The idea is to have a process that wants to execute an operation, find out which operations have been announced, apply these operations to a local copy of the object, then finally change the global object pointer to refer to this local copy. However, these constructions do not cope well with large shared object as they copy the state of the object, then apply changes to the local copy. In [12], Fatourou *et al.* presented algorithms for efficiently handling shared objects with large state size, however this approach includes dedicated unique threads that apply updates to the data structure. Consequently, compromising the wait-free progress guarantee in the prior work.

In this paper, we utilize combining techniques without relaxing the lock-free progress guarantees of a concurrent vector. Contrary to flat-combining where an announcement list contains a request record for each thread, we utilize an unbounded array-based queue for combining. With a queue, a combiner traverses a collection of only active requests.

2.2 System Model and Definitions

We consider an asynchronous shared memory model with a finite set of n processes $p_1, ..., p_n$ where n may exceed the number of physical processors, and each process may exhibit *fail-stop* behavior. Processes communicate by executing atomic operations on predefined *shared objects*. A process may suspend execution at any time for arbitrarily long, and we cannot differentiate between fail-stop and suspended processes.

We assume a sequentially consistent [20] memory model[2] to simplify the presentation of the pseudo-code. However, modern compilers and architectures provide weaker guarantees that allow instruction reordering for performance reasons. This necessitates the use of atomic instructions and explicit memory fences to ensure sequentially consistent memory accesses. An instruction is atomic if the invocation and response occur with a guarantee of isolation from concurrent executions.

In addition to atomic read and write instructions, the system supports *Compare-And-Swap* (CAS) and *Fetch-And-Add* (FAA) atomic read-modify-write instructions. The CAS$(address, old, new)$ instruction; checks if the current value at a memory location $(address)$ is equivalent to the given value old, and only if true, changes the value of $address$ to the new value (new) and returns TRUE; otherwise the memory location remains unchanged and the instruction returns FALSE. The FAA$(address, incr)$ operation adds $incr$ to the value at memory location $address$ and returns its previous value. Both instructions are commonly available in modern processors.

We assume that a *Vector* implements a multiset (or bag) abstract data type which contains elements that allow for random access using a non-negative index. The Vector is resizable and supports PUSHBACK, POPBACK, RESERVE, READ, WRITE and SIZE operations [9].

A PUSHBACK operation appends an element to the end of the vector and increments the vector $size$. A POPBACK operation removes an element from the end of the vector, decrements the vector $size$ and returns the removed item. The RESERVE operation takes an argument n and adjusts the vector capacity to be able to contain at least n elements. This operation has no effect on the size or elements of the vector. A READ operation reads the value at a given index, while as the WRITE operation writes a given value at a specified index. The SIZE operation returns the number of elements in the vector.

To demonstrate the correctness of our concurrent vector design, we verify safety and liveness properties. The safety property that we use is *linearizabilty* [21], which generally implies that an operation on the vector appears to the rest of the system to ”take effect” instantaneously at some point during the interval of its execution. The liveness property that we use is *lock-freedom* [4]; lock-freedom requires that infinitely often some process will complete its operation in a finite number of steps regardless of failed or delayed processes.

[2] Sequentially consistent means that the result of an execution is the same as if the operations were executed in the order specified by the program

2.3 Algorithm

In this section, we present an overview of our vector algorithm with combining technique. We then provide details of the algorithm implementation along with the pseudocode, correctness proofs, ABA avoidance and memory management.

2.3.1 Overview of the Algorithm

Our concurrent vector design is largely based on the lock-free dynamically resizable arrays [9]. The design structure is shown in Figure 2.1. It is comprised of an array of arrays for storing elements, a Descriptor object that describes the vector's size and a reference to the active queue segment of pending operations. Similar to Intel's concurrent vector [8], the vector does not store all elements in contiguous memory and elements are not relocated when the vector is resized. Instead, elements are stored in a two-level array with every new memory block allocated having twice the size of the most recently added block. The size of the first element array is a power of 2, and thus the sizes of the subsequent arrays are always powers of 2.

Descriptor objects introduced in the cooperative technique by Barnes [22] contain sufficient information on a pending operation to allow concurrent threads to complete the operation. Thus if a thread t_1 reads a descriptor allocated by another thread t_2 , thread t_1 can complete the pending operation ensuring system-wide progress. Descriptors have also been exploited to represent *'before'* and *'after'* versions of a memory location allowing concurrent readers access to content of the memory location during progress of a write operation. This technique further allows for rollback of the write operation. To maintain the semantics of a vector ADT, tail operations PUSHBACK and POPBACK need to

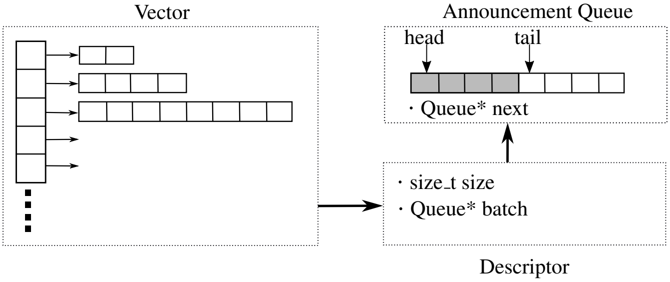


Figure 2.1: Vector Design. Threads add operation descriptors to the announcement queue before attempting to apply them to the vector. Vector size is only updated after using up a queue segment.

modify the data storage array and the size of vector atomically. Thus, a Descriptor object is exploited to atomically modify multiple disjoint memory locations and also allow for cooperative execution of operations.

We utilize the *queue* to store pending operations for concurrent threads. A thread announces its operation by adding it to the queue, then proceed to complete the operation or help other pending operations complete.

Figure 2.2 presents type definitions and declarations for global variables. We extend the Descriptor object to encapsulate information about the *queue*. We simulate an unbounded array-based queue by using a linked-list of bounded queues.

Each queue is represented by a fixed-length array of QSize cells, head, tail, and next pointers. We pack two values into 64-bit head ⟨hindex:32,hcount:32⟩ with the hindex and tail representing the index values in the array and the hcount indicates the number of successfully completed operations. We make the assumption that the value of QSize does not exceed 2^{32}. The cells in the array are initialized with a reserved null value $\perp$.

The operation descriptor – WriteDescr contains the type of operation performed by the thread: PushBack, PopBack. The `Vector` structure holds a pointer to the current descriptor object as well as pointers to the data storage arrays. In the pseudo-code, `th_info` represents the thread-local state.

For the PushBack operation, a thread creates a new write descriptor object containing a new item, then obtains an index i in by performing a FAA on the queue tail. It then attempts to add its write descriptor to the queue at index i. If successful, the operation continues to either complete all operations in the queue before its index or *back-off* and wait for its operation to be completed by others. The PopBack operation proceeds in a similar fashion, except that, it starts off with a null value and terminates with either a return value or `EMPTY` if the vector is empty.

In PushBack and PopBack operations, threads attempt to complete pending operations as indicated by the write Descriptor objects appearing earlier in the queue before continuing with the their own operation. Random access operations (Reserve, Read, Write and Size) execute without modifying current Descriptor or the size of the vector.

2.3.2 Implementation Details

In this section, we present a detailed description of our vector data structure design. The pseudo-code of our implementation is presented in Algorithm 2.1–2.5. In this pseudo-code, we use “&” to obtain the address of an object, “$\curlywedge$” to read the contents of a memory location (dereferencing) and “.” for field access of a

```
▷ Vector consists of data storage array
▷ and a descriptor
struct Vector {            ▷ Array of arrays of node pointers
    Node** vdata;
    Descr* descr;
}
struct Descr {
    size_t size;
    Queue* batch;
}
struct Node {
    value_t value;
}

▷ Global vector initialized
Vector* vector ← ⟨init values⟩

struct WriteDescr {
    bool pending;
    Node* v_new;
    OpType op;
}

struct Queue {
    void* items[QSize];
    size_t tail;
▷ ⟨hindex:32, hcount:32 bit⟩
    size_t head;
    Queue* next;
}

OpType {PUSH, POP};
```

Figure 2.2: Type definitions for the vector structure, operation Descriptor, WriteOp Descriptor along with the Bounded Queue structure.

struct. Additionally, when multiple items are packed into a single word, then we use "$\langle x, y, z \rangle$" notation to represent the packed word and the symbol "$\rightsquigarrow$" to reference a subfield of the word.

We reserve two special "$null$" values, $\bot$, and $\top$ that are never enqueued to the queue of pending operations. Additionally, we assume the existence of a function $At(i)$ which accepts a vector index i and returns cell in a given sub-array of the vector. The function maps a given index into a bucket and returns the cell associated with the index i in that bucket.

The function `newDescr` creates a new Descr object, while as the function `newWriteDescr` creates a new WriteDescr object. The functions `At`, `AllocateBucket`, `GetBucket`, and `Reserve` are left unchanged from the implementations in [9], thus not presented in this paper.

(A) PushBack Operation

The pseudo–code for the PushBack operation is presented in Algorithm 2.1. A thread executing the PushBack operation starts off by creating a new WriteDescr with details about the new element. The thread then attempts to announce the operation by adding it to an active queue segment (`AddToBatch`). After announcing the operation, the thread continues to `Complete` the pending operation if not already completed by other threads. We discuss `Complete` in Section (C).

```
PushBack(thInfo* th, E elem)
    q ← vector.batch ;
    wop ← newWriteDescr(true, elem, PUSH);
    ⟨m_index,m_q⟩ ← AddtoBatch(q, wop);
    while wop.pending do
        Complete(wop, m_q, m_index);
    return ;
```

Algorithm 2.1. The PUSHBACK operation

(B) AddToBatch Phase

In the `AddtoBatch` phase, a thread executing an operation adds the write descriptor wop representing the operation details to the queue so as to enlist help from other threads to complete the operation. The pseudo-code of the `AddToBatch` is presented in Algorithm 2.2.

```
1  AddtoBatch(Queue* q, WriteDesc* wop)
2      while true do
3          if q.tail ≥ QSize then
4              if q.next = null then
5                  new_q ← newQueue(wop);
6                  new_q.tail ← 1;
7                  if CAS(&q.next , null, new_q) then
8                      myindex ← 0 ;
9                      q ← new_q;
10                     break;
11             q ← q.next;
12             continue;
13         else
14             myindex ← FAA( q.tail) ;
15         if myindex ≤ QSize then
16             if CAS(&q.items[myindex] , ⊥, wop) then
17                 break;                                  ▷ late entry
18     vec_q ← vector.batch ;
19     while vec_q ≠ q ∨ vec_q.hindex < m_index do
           Backoff()                                   ▷ Backoff for a threshold.
20         vec_q ← vector.batch ;                      ▷ Advance to next queue.
21         UpdateSize(vec_q);
22         if threshold then
23             return ⟨myindex,q⟩
```

Algorithm 2.2. The ADDTOBATCH operation

To add a write descriptor wop to the queue, a thread attempts to find a queue segment that has free cells. If none exists; the thread creates one, initialized to contain the write descriptor wop as the first element of the array (Algorithm 2.2, line 3–10). We use CAS operation to synchronize concurrent initialization of a *segment*, hence only one thread succeeds and the others fail the operation.

Otherwise, a thread obtains an array index i by performing an FAA on the tail value of the queue segment (Algorithm 2.2, line 14). If the index i is greater than or equal to QSize, the thread will retry to find a new segment and index (Line 15). If the index i is less than the queue size QSize, the thread attempts to atomically replace the value at $Q[i]$ with wop ($Q[i] :\perp\longmapsto$ wop, Line 16). If the CAS operation succeeds, the thread has successfully added its operation descriptor to the queue and can continue to complete the operation. The CAS operation may fail due to an interfering `Complete` operation as discussed in Section (C).

Multiple threads can concurrently add operations to the queue, these operations will be completed in FIFO order. Thus, a thread may be required to wait for operations appearing before its operation in the queue to be completed. If a thread observes that there are pending operations enqueued before its operation, the thread performs a *back-off* routine to allow earlier operations to be completed (Algorithm 2.2, line 19–23).

```
UpdateSize(Queue* q)
  if q.hcount = QSize ∧ q.next ≠ null then
    newD ← newDescr();
    newD.size ← vec_q.size + vec_q.hcount;
    newD.batch ← vec_q.next;
    if ¬CAS(&vector.descr, q, newD) then
                                                    ▷ Free descriptor.
```

Algorithm 2.3. The UPDATESIZE operation

To ensure progress, the *back-off* is bounded so that a thread is not delayed indefinitely in case operations appearing earlier in the queue were not completed. A thread calls the `Backoff` routine for a threshold or until its operation is completed by other threads, then returns. Additionally, during `Backoff`, the thread can help advance the completion of operations from one queue segment to the next by calling the `UpdateSize` routine.

The `UpdateSize` routine updates the size of the vector, which is only update after using up a queue segment. Updating the size in batches allows us to reduce the number of updates on the size variable, thus, minimizing the effects of tail operations on random access operations.

(C) Complete Operation in Detail

The pseudo-code for the `Complete` operation is presented in Algorithm 2.4. The `Complete` involves execution of operations announced in the vector's queue batch: add items in PUSHBACK operation descriptors to the vector storage

```
Complete(mywop, myq, myindex)
  batch ← vector.batch;
  while true do
    head ← batch.head;
    ⟨hindex,hcount⟩ ← head;
                                                        ▷ Gaps found.
    if batch.items[hindex] = [$\bot \lor \top$] $\land \neg$mywop.pending then
      return;                                           ▷ was just helping.
    if hindex = QSize then
      UpdateSize(batch);                                ▷ Advance to next queue.
      batch ← vector.batch;
      continue;
                                                        ▷ linearize with push operation.
    if CAS(&batch.items[hindex] , $\bot$, $\top$) then
      new_head ← ⟨hindex +1 ,hcount⟩;                   ▷ update head
      CAS(&batch.head , head, new_head);
      continue;
    if batch.items[hindex] = $\top$ then
      new_head ← ⟨hindex +1 ,hcount⟩;
      CAS(&batch.head , head, new_head);
      continue;
    wop ← batch.items[hindex];
    $\delta$ ← (wop.op = POP) ? − 1 : 1;
    if $\neg$ wop.pending then
      new_head ← ⟨hindex +1 ,hcount +$\delta$⟩;
      CAS(&batch.head , head, new_head);
      continue;
    index ← hcount + ((wop.op = POP) ? − 1 : 0);       ▷ Operation determines index
    pos ← batch.size + index;
                                                        ▷ Check for empty vector, if operation is POP
    bucket ← GetBucket(pos );
    if vec.vdata[bucket] = null then
      AllocateBucket(vec, bucket);
    addr ← At(vec, cur.offset + hcount);
    val ← addr^;
    if wop.pending $\land$ wop.op = PUSH then
      CAS(addr , val, wop.v_new);
      wop.pending ← false;
    else if wop.pending then
      CAS(&wop.v_new , null, val );
      markNode(pos);
      wop.pending ← false;
    new_head ← ⟨hindex +1 ,hcount +$\delta$⟩;
    CAS(&batch.head , head, new_head);
```

Algorithm 2.4. The COMPLETE operation

arrays, and return items to pending PopBack operation descriptors.

To complete an operation from the queue segment, a thread extracts the hindex field of the head value (Q.head⇝hindex) which represents the current index in the segment. After reading the hindex, the thread proceeds to access the operation at the corresponding array cell.

However, the array cell can be in one of three states:

1. The cell value is $\perp$, which implies that the corresponding `AddToBatch` operation has not completed adding item to the queue.
2. The cell value is a write descriptor (wop) which implies that the `AddToBatch` operation completed successfully.
3. The cell value is $\top$, which implies that the cell value was overwritten by an interfering `Complete` operation before the `AddToBatch` succeeded. This ultimately indicates the failure of the corresponding `AddToBatch` operation.

If a thread helping to complete operations on behalf of other threads observes that the target cell is in state 1 or state 3, the thread returns (Algorithm 2.4, line 7). This gives the slow thread an opportunity to complete its `AddToBatch` operation (state 1) or the thread that invalidated the target cell to continue with the `Complete` phase.

With the three possible values at the queue cell, and a possibly interfering write operation by a concurrent `AddToBatch`, a thread completing an operation from the queue attempts to write a $\top$ at the hindex with a CAS operation, $Q.items[\mathsf{hindex}] : \perp \longmapsto \top$ (Algorithm 2.4, Line 14). If the CAS succeeds, the thread considers the slot empty and that the corresponding `AddToBatch` execution failed. Then, proceeds to update the head value of the queue segment.

However, if the CAS fails, then either an interfering `Complete` operation already rendered the cell invalid in which case the thread attempts to help update the hindex ($Q.head : \langle \mathsf{hindex}, \mathsf{hcount}\rangle \longmapsto \langle \mathsf{hindex} + 1, \mathsf{hcount}\rangle$) or the cell contains a write descriptor. Additionally, if a thread observes an invalid cell, it updates the segment head before continuing (Algorithm 2.4, Line 18).

If the array cell contains a write descriptor wop, the thread proceeds to help complete the operation. We use the δ which depends on the type of operation and the hcount to determine the target index in the vector. We maintain the hcount as the aggregate change in vector size due to operations successfully completed in a given queue segment. Algorithm 2.4(Line 28–30), we compute a target position based on the type of operation; we also check if the vector is empty and return `EMPTY` to a pending PopBack operation.

Depending on the nature of the pending operation, PushBack or PopBack, a thread continues to either write the item to the storage array or read at item from storage array and adding it to the operation descriptor. A thread updates the head value ($Q.head : \langle \mathsf{hindex}, \mathsf{hcount} \rangle \longmapsto \langle \mathsf{hindex} + 1, \mathsf{hcount} + \delta \rangle$ and proceeds with the next item in the queue (Algorithm 2.4, Line 44).

The function `markNode` (Algorithm 2.4, Line 41) is used to indicate that an item has been deleted from the vector. We employ the pointer marking technique described by Harris [23]. Random access operations may attempt to execute at indexes that are higher than the current size of the vector. On reading a marked node, the thread observes that the current operation is executed out of bounds.

A thread continues to complete operations from the queue segments, until: (1) the thread gets to the end of a segment and its operation has already been completed, (2) the thread whose operation has already completed observes contention from concurrent threads. Therefore, a thread only returns from the `Complete` operation after its operation has been completed, and it has also attempted to help other threads complete their operations.

(D) popback Operation

Algorithm 2.5 presents the PopBack operation. Similar to PushBack, a thread executing the PopBack operation starts off by creating a new WriteDescr with details about operation. Then, attempts to announce the operation by adding it to an active queue segment (`AddToBatch`). The thread returns from the `AddToBatch` after having its operation completed by a concurrent thread, or backing-off for the threshold duration. If the operation was not completed, the thread calls `Complete` to help complete any pending operations enqueued before its operation, then complete its own operation. After which the thread returns either a value taken from the back of the vector or `EMPTY` if the vector did not contain any items.

```
PopBack(thInfo* th, E elem)
    q ← vector.batch ;
    wop ← newWriteDescr(true, elem, POP);
    ⟨m_index,m_q⟩ ← AddtoBatch(q, wop);
    while wop.pending do
        Complete(wop, m_q, m_index);
    elem ← wop.v_new.value;
    return elem;
```

Algorithm 2.5. The PopBack operation

(E) Read-Write Operations with bounds checking

In this section, we describe the implementations of random access operations Read and Write. We augment these operations with bounds checking to deal with cases where the *index* is greater than the current size of the vector. Failure to do so implies that READ–WRITE operations cannot be linearized with concurrent POPBACK or PUSHBACK executions.

One solution to identify whether a position is within bounds, is to check the current size of the vector. This involves threads reading the vector descriptor on every random access operation. If every update on the vector size changes the vector descriptor, threads incur significant overhead reading the size value each time before a random operation.

We reduce this overhead by having the vector descriptor changed only when traversing from one operation queue segment to the next. In this way, the size of the vector is only update after approximately every QSize tail operations, instead of after every operation. Thus the random access operations proceed as detailed below:

READ: For a READ operation, a thread compares the index with the vector size value by reading the vector descriptor. If the index is still greater than the size value plus QSize, then it is considered out of bounds and the read operation returns null. If the index is within QSize of the size value, the thread reads the vector batch ($Q.head \rightsquigarrow \mathsf{hcount}$) to ascertain the actual size of the vector.

If the index is within the current size bounds, the thread proceeds to read the node pointer stored at the vector index. Then checks that the value has not been marked by a concurrent POPBACK operation. If true, the thread returns the value associated with the index, otherwise, it considers the position out of bounds. In summary, we use both the vector size and bit-marking to guarantee that read operations are linearized correctly with concurrent tail operations.

WRITE: In addition to bounds checking, the WRITE operation attempts to modify the value at a given index i. Therefore, it starts off with bounds checking as described for the READ operation above, on reading the current node and confirming that it has not been previously marked, proceeds to create a new node with the value V_{new} and swap with the old node $(At(i) : \{\mathsf{V}_{old}\} \longmapsto \{\mathsf{V}_{new}\})$. The operation uses a CAS instruction in an attempt to replace the current value with the new node returning true if successful and false if it fails.

size: The size operation returns the size stored in the current vector size plus the value of $Q.head \rightsquigarrow \mathsf{hcount}$.

(F) Elimination Backoff

Hendlar *et al.* [24] proposed the elimination backoff for the lock-free stack, in which concurrent PUSH and POP operations exchange data and complete without contending at the actual data structure. An array is typically used as the elimination area to allow for opposite pairs of operations to collide and cancel out. Threads set a timeout for the elimination procedure, on timing out without a collision with an inverse operation, the thread attempts to access the underlying data structure directly.

Similar to a concurrent stack, PUSHBACK and POPBACK operations of the vector are converse in nature and concurrent executions can exchange data and cancel out without contending for access to the data structure. For correctness, this is considered equivalent to a POPBACK operation happening immediately after a PUSHBACK. Thus, we can optimize performance by utilizing the elimination backoff strategy.

Naturally, this raises the question as to why we do not utilize Elimination in all cases for opposite PUSHBACK and POPBACK. We observed that the elimination technique is more suited to workloads with approximately equivalent PUSHBACK and POPBACK workloads. For skewed workloads, many operations can not be canceled out at the elimination layer, thereby incurring the overhead without reducing the contention on the underlying data structure.

2.3.3 Correctness

To demonstrate the correctness of our concurrent vector design, we verify safety ($linearizabilty$ [21]) and liveness ($lock\text{–}freedom$ [4]) properties. This section presents proofs sketches that support our arguments for the linearizability and lock-freedom of our concurrent vector design.

(A) Linearizability

Linearizability is a correctness property which requires that each operation should appear to take effect instantaneously at some point (*linearization point*) between its invocation and response [21]. This definition implies that the real-time ordering of operations is preserved, and a concurrent execution history[3] yields an equivalent legal sequential history of the same executions that is consistent with the real-time order. A history is sequential if the first event is an invocation, and each invocation except possibly the last one is followed by a matching response.

[3] A history is a finite sequence of operation invocations and responses. In a history H, an operation is a pair consisting of an invocation and a matching response. A legal history respects the sequential specifications of the object.

Consequently, a $linearizable$ concurrent vector comprises of a concurrent history that can be reordered to yield a sequential history that is correct according to a definition of a sequential vector and is consistent with the operations' real-time ordering.

Identifying a *linearization point* for each operation is one popular approach to show that a concurrent object is a linearizable implementation of a sequentially specified object. Ordering concurrent operations according to the linearization points, every concurrent history is equivalent to some sequential history. In the sequential history, if one operation precedes another, then that operation must have taken effect before the later call.

We denote the linearization point of an operation op as $H_{j(op)}$, and an operation $op1$ precedes $op2$ ($op1 \prec op2$) if the linearization point of $op1$ happens before that of $op2$.

Definition 2.1. *An object is a linearizable vector implementation if it satisfies the following requirements*[4]*. We denote the logical vector as V and a cell at index k of the vector as $V[k]$.*

1. *For each thread T_i adding a sequence of items $k \in \{1, 2, ...\}$ to the vector, $E_k \prec E_{k+1}$.*
2. *Let D_k be the* PopBack *operation that returns the item at index k, and E_k the* PushBack *operation that added the item to the vector, then $E_k \prec D_k$.*
3. *$D_{k+1} \prec D_k$ and there is at most one* PopBack *operation that returns the item at index k.*
4. *The $size$ value is always equivalent to the number of items in the vector.*

Lemma 2.1. *A* PushBack *operation E_k is always linearized before a later* PushBack *E_{k+1} by same thread.*

Proof. During a PushBack operation, a thread either successfully announces its operation to a queue segment using a CAS instruction or fails and retries after acquiring a new index and probably a new queue segment.

Pending operations in the queue are applied to the vector in FIFO order and an operation only returns after the queue head has moved past its index in the queue. Therefore, E_k is linearized before E_{k+1}. □

[4] For clarity we assume that all the items are unique, however the proof can easily be modified to hold without this assumption

Lemma 2.2. *A* PUSHBACK *operation E_k is always linearized before the matching* POPBACK D_k *that removes item from the vector.*

Proof. Similar to the PUSHBACK operation, the POPBACK operation is added to the operations array where operations are completed in FIFO order. The linearization point is either at (Algorithm 2.4, line 30) if the vector is empty or at (Algorithm 2.4, line 41) when the vector executes CAS to mark the vector array cell thus making the operation visible to concurrent random access operations. A POPBACK operation that succeeds as above, only executes after all prior PUSHBACK operations are completed. □

Lemma 2.3. *An element that is successfully added to a queue segment is applied to the vector at most once.*

Proof. `AddToBatch` operation succeeds only if the CAS execution in Line 16 of Algorithm 2.2 succeeds, otherwise, the segment cell is invalidated by a concurrent `Complete` operation (Algorithm 2.4, line 18).

In the `Complete` operation, there are two main challenges; first we must ensure that operations are not added to the queue after we have advanced the queue *head* (`Q.head`) past a given index. This is dealt with as explained in section (C), and empty slots are invalidated with a CAS $(\&q.ids[index], \bot, \top)$.

The other challenge is ensuring that if two threads are concurrently executing an operation taken from the same index of the queue, only one of these threads succeeds and the operation is applied at most once to the vector. A thread reads the value at the target vector slot (Algorithm 2.4, line 35), and only completes the operation if it is the first thread to do so with a CAS in Algorithm 2.4 line 37 or line 40 . Other attempts fail to complete the operation. □

Lemma 2.4. $D_{k+1} \prec D_k$ *and there is at most one* POPBACK *operation that returns the item at index* k

Proof. This follow directly from Lemma 2.3 □

Theorem 1. *Our algorithm is a linearizable implementation of a dynamic vector*

Proof Sketch. Lemma 2.1–2.4 show that given a sequentially specified vector object, we identify specific linearization points mapping an arbitrary concurrent execution history of our concurrent vector design to meet the specifications. For concurrent PUSHBACK and POPBACK, each operation that was successfully added to a cell in the queue of pending operations is essentially linearized before the CAS that moves the hindex past the cell occupied by the operation. □

(B) Lock–freedom

Lock-freedom is a progress guarantee that infinitely often some process will complete its operation in a finite number of steps despite arbitrary delay or failure of any other processes.

Theorem 2. *The presented concurrent vector algorithm is lock-free*

Proof Sketch. In our design, random access operations do not modify the vector descriptor or vector queue; they are wait-free. For tail operations, threads acquire slots in the queue of pending operations using FAA, which always returns an index. Although, the returned index my be larger than QSize, the thread retries and this indicates that other operations have taken indexes less than QSize After a thread has been granted a cell in the queue, this cell is only invalidated by threads that have already completed announcing their operations in the queue. Thus, announcement of an operation only fails, after another thread has guaranteed that its operation will succeed. Announcing an operation guarantees that the operation will definitely complete before other threads can move the queue index past the given cell. Consequently, system wide progress is guaranteed; proposed algorithms are *lock-free*.

□

2.3.4 Memory Management and ABA Problems

The presented algorithms do not rely on any memory management or reclamation mechanism. For our implementation, we utilize SSMEM [25] , a memory allocator with epoch-based garbage collection [26]. While efficient, epoch-based memory reclamation schemes allow for unbounded growth in unreclaimed objects due to thread delay or failure. This is still an open problem, which we are actively investigating. DEBRA by Brown [27], is an attempt at solving the problem using signaling to detect thread failures on Posix based systems.

Additionally, CAS-based implementations are prone to the ABA problem [28]; a thread Th_1reads a value A from a shared memory location, Th_2 changes the value to B and then Th_2 or another thread changes it back A. When Th_1 executes a CAS instruction on the location, it succeeds erroneously as if the location has not been changed since last read by Th_1.

In a concurrent vector, the ABA problem can occur in two cases: (a) an element A is stored at a vector index multiple times, (b) a freed object is reused while some threads still hold references to the object. In our implementation, (a) is solved by storing the elements in garbage collected nodes and only store pointers to these nodes in the data array. The memory reclamation scheme implicitly solves (b).

2.4 Performance Evaluation

In this section, we evaluate the performance of our concurrent lock-free vector compared to several concurrent vector implementations in the literature. In our experiments, we compare the performance of the following implementations:

1. **LFBatchElimTH:** Our implementation of a lock-free vector with combining, elimination and a threshold as described in Section 2.3.2
2. **LFvector:** Lock-Free dynamic resizable array [9] which is the first Descriptor-based lock-free concurrent vector design.
3. **TBBvector:** Lock-based vector implementation from Intel's Thread Building Blocks [8], PUSHBACK is implemented with a fetch-and-add on the vector size and this vector does not support POPBACK operation.
4. **STL+Mutex:** STL vector made thread-safe with a mutex lock. Similar to [9], we experimented with multiple lock synchronization primitives (mutex, ticket-locks, spin-locks), however the mutex lock was the best performing. Thus, we only include results for the mutex lock in the evaluation.
5. **WFvector:** Wait-free vector design that utilizes fast-path slow-path technique to achieve wait-freedom [10]. Additionally, the vector design offers tail operations that are synchronized using FAA (**FAAvector** - no POPBACK operation), and thus does not allow complement operations i.e. if one supports PUSHBACK, POPBACK is not supported.

Hardware Platform We performed our evaluations on a dual-socket server with a 3.4 GHz Intel E5-2687W-v2 having 16 physical cores (32 hardware threads by hyper-threading), 16 GB of RAM, running Ubuntu 13.04 Linux. All the algorithms were implemented in C/C++, compiled with `gcc` version `4.9.2`, `O2` optimization and TCMalloc enabled for the STL implementations [29] to reduce dynamic memory allocation overheads. We manually pin software threads onto hardware threads so as to leverage CPU affinity within sockets.

Benchmark For this evaluation, we generate various workload distributions (PUSHBACK, POPBACK, READ and WRITE) $\in \{(100,0,0,0), (50,50,0,0), (70,10,10,10), (50,30,10,10), (25,25,25,25), (10,10,40,40)\}$ as we vary the number of execution threads $\in \{2,4,8,12,16,20,24,28,32\}$. We measure throughput performance as the number of successful operations per second. Each measurement represents the average over 6 runs with each thread executing 10^6 operations on a shared vector object. Each operation is a: PUSHBACK$(+)$, PopBack$(-)$, random access Read(r) or random access Write(w) with a probability based on the workload distribution. We initialize the vector with a number of elements ($N = 2^{15}$) to reduce effects of empty PopBack operations in our analysis.

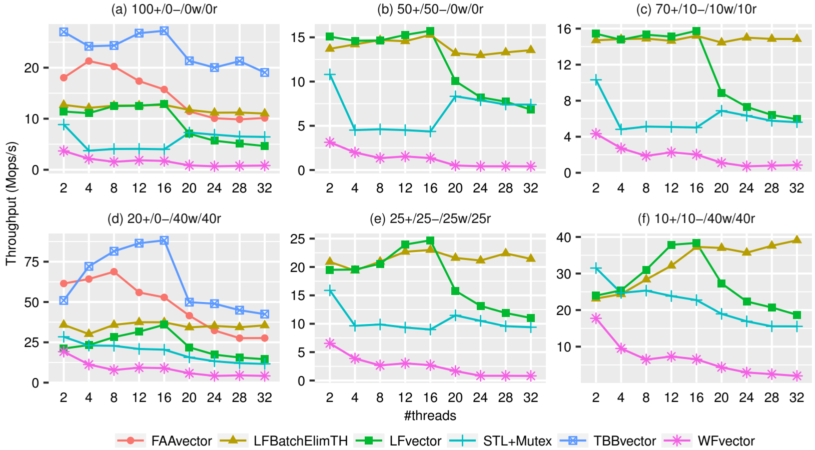


Figure 2.3: **Throughput:** Performance results for different implementations with PushBack$(+)$, PopBack$(-)$, random access Write(w) and random access Read(r) percentage workload distributions as we increase active threads for each distribution.

2.4.1 Experimental Results and Discussion

Figure 2.3 shows the results of our experiments with the vector implementations. In the figure, the throughput in million-operations/per second is plotted on the y-axis of each diagram, while the number of threads is plotted on the x-axis. As FAAvector and TBBvector do not support PopBack operation, we only include them in distributions that exclude the PopBack operation (Figure 2.3 a & d).

We observe that for workloads that exclude the PopBack operation (Figure 2.3 (a) & (d)), FAAvector and TBBvector significantly outperform other designs. Given that they do not support conflicting PopBack and PushBack operations, FAAvector and TBBvector do not incur any performance penalties that are paid by the other implementations that cater for inverse operations. As we increase the number of threads to have executions on the second socket (more than 16 threads), performance degrades for both; with the FAAvector degrading more than the TBBvector.

The LFvector scales poorly for all workloads as we increase the number of threads to execute on both sockets of the server. We attribute the degradation in performance to the increase in contention and the latency resulting from cache misses in the NUMA architecture. With PopBack and PushBack operations having to update the descriptor value of the vector, this results in significant delays due to cache misses as the threads have to update the invalidated value of the descriptor after each successful operation. This is exacerbated on the dual socket architecture. Inter-socket communication cost is much higher than intra-socket costs between cores on the same socket. In contrast, LFBatchElimTH offers

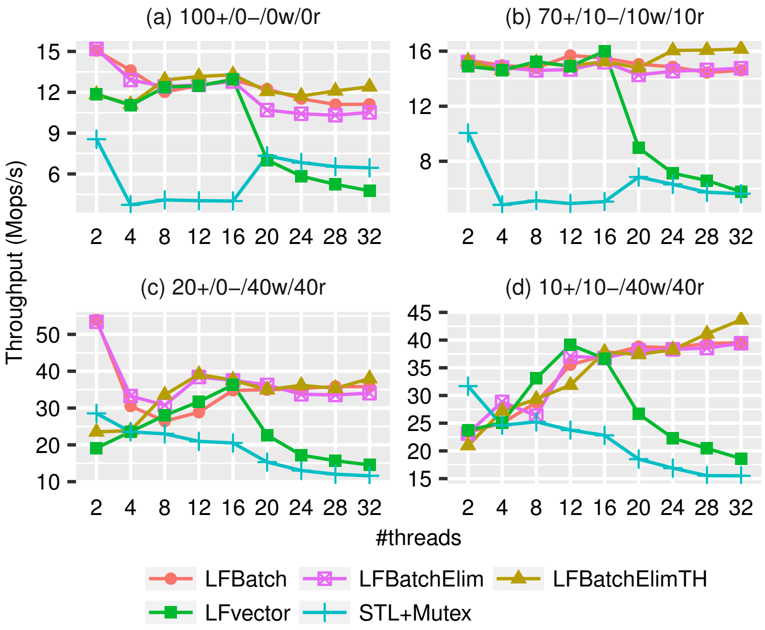


Figure 2.4: Performance results with optimizations proposed in Section (F).

performance similar to the LFvector for threads running on a single socket (≤ 16) and scales well across server sockets (threads > 16) for all workload distributions. It performs on average 2.76 times as many operations as the LFvector at high contention. The major reason for the scalability is that combining reduces contention on the vector descriptor and cache-miss storm that occurs every time the value of the descriptor is changed and invalidates the values held in the cache by each thread. On failure to complete any CAS operation, threads do not attempt to re-read the descriptor value and re-execute the CAS at same memory location. Consequently reducing contention, resulting in better scalability.

Wait-Free implementation does not scale with increasing thread count; performs worse than *STL+Mutex*, and the performance further deteriorates when threads are executed on both sockets (more than 16 threads). The poor performance is attributed to the synchronization costs associated with the tail operations (PUSHBACK, POPBACK), and the cost of maintaining the vector in contiguous memory. Each tail operation involves at least four CAS operations without contention, and these expand with increasing contention.

Degradation in TBBvector, FAAvector, and LFvector for READ/WRITE dominated workloads as we increase threads to occupy both sockets of the server was unexpected due to the random access and independent nature of these operations. The degradation is a result of bounds checking that is added for the benchmarking so as not to access invalid memory. Bounds checking involves reading the current size of the vector, which inherently requires a read of the descriptor [30].

Figure 2.4 compares the performance of our vector design with various optimization techniques implemented as described in Section (F). *LFBatch*: vector with combining, *LFBatchElim*: vector with combining and elimination, *LFBatchElimTH*: vector with combining, elimination and a threshold before combining.

2.5 Conclusion

We presented a scalable and NUMA efficient design and implementation of a lock-free vector. The design augments a prior lock-free vector design with *combining* synchronization technique without compromising the progress guarantees.

We compared our vector design with prior implementations both lock-based and non-blocking using micro-benchmarks. We observed that our implementation performs as well as the most efficient vector design, and it out performs it on NUMA architectures for all workload distributions. Our lock-free combining technique can be utilized to improve the efficiency of other lock-free data structures with potential synchronization bottlenecks (*i.e.,* stacks and queues).

Bibliography

[1] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali, “A case for numa-aware contention management on multicore systems,” in *Proceedings of the international conference on Parallel architectures and compilation techniques*. 2010, pp. 557–558, ACM.

[2] Maurice Herlihy and Nir Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann Publishers Inc., 2008.

[3] Maurice Herlihy, Victor Luchangco, and Mark Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *Proceedings of the IEEE International Conference on Distributed Computing Systems*. 2003, pp. 522–529, IEEE.

[4] Maurice Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 1, pp. 124–149, 1991.

[5] Phong Chuong, Faith Ellen, and Vijaya Ramachandran, "A universal construction for wait-free transaction friendly data structures," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2010, pp. 335–344, ACM.

[6] Panagiota Fatourou and Nikolaos D. Kallimanis, "A highly-efficient wait-free universal construction," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2011, pp. 325–334, ACM.

[7] Maurice Herlihy, "A methodology for implementing highly concurrent data objects," *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 5, pp. 745–770, 1993.

[8] Intel, "Reference for intel threading building blocks," `https://www.threadingbuildingblocks.org/`, 2016.

[9] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup, "Lock-free dynamically resizable arrays," in *Proceedings of the International Conference on Principles of Distributed Systems*. 2006, pp. 142–156, Springer.

[10] Steven Feldman, Carlos Valera-Leon, and Damian Dechev, "An efficient wait-free vector," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 654–667, 2016.

[11] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2010, pp. 355–364, ACM.

[12] Panagiota Fatourou and Nikolaos D. Kallimanis, "Revisiting the combining synchronization technique," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2012, pp. 257–266, ACM.

[13] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa, "Executing parallel programs with synchronization bottlenecks efficiently," in *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*. 1999, pp. 182–204, World Scientific.

[14] Irina Calciu, Hammurabi Mendes, and Maurice Herlihy, "The adaptive priority queue with elimination and combining," in *Proceedings of International Symposium on Distributed Computing*. 2014, pp. 406–420, Springer.

[15] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup, “Understanding and effectively preventing the aba problem in descriptor-based lock-free designs,” in *Proceedings of the IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. 2010, pp. 185–192, IEEE.

[16] Alex Kogan and Erez Petrank, “A methodology for creating fast wait-free data structures,” in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2012, pp. 141–150, ACM.

[17] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie, “Distributing hot-spot addressing in large-scale multiprocessors,” *IEEE Transactions on Computers*, vol. 36, no. 4, pp. 388–395, 1987.

[18] Nir Shavit and Asaph Zemach, “Combining funnels,” *Journal of Parallel and Distributed Computing*, vol. 60, no. 11, pp. 1355–1387, 2000.

[19] Panagiota Fatourou and Nikolaos D. Kallimanis, “A Highly-efficient Wait-free Universal Construction,” in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2011, pp. 325–334, ACM.

[20] Leslie Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs,” *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, 1979.

[21] Maurice Herlihy and Jeannette M. Wing, “Linearizability: A correctness condition for concurrent objects,” *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[22] Greg Barnes, “A method for implementing lock-free shared-data structures,” in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 1993, pp. 261–270, ACM.

[23] Timothy L. Harris, “A pragmatic implementation of non-blocking linked-lists,” in *Proceedings of the International Conference on Distributed Computing*. 2001, pp. 300–314, Springer.

[24] Danny Hendler, Nir Shavit, and Lena Yerushalmi, “A Scalable Lock-free Stack Algorithm,” in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2004, pp. 206–215, ACM.

[25] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis, “Asynchronized concurrency: The secret to scaling concurrent search data structures,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 2015, pp. 631–644, ACM.

[26] Keir Fraser, “Practical lock-freedom,” *PhD thesis, University of Cambridge*, 2004.

[27] Trevor Alexander Brown, “Reclaiming memory for lock-free data structures: There has to be a better way,” in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 2015, pp. 261–270, ACM.

[28] Maged M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, 2004.

[29] Sanjay Ghemawat and Paul Menage, "Tcmalloc : Thread-caching malloc," `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`, 2009.

[30] Bapi Chatterjee, Ivan Walulya, and Philippas Tsigas, "Help-optimal and language-portable lock-free concurrent data structures," in *Proceedings of the International Conference on Parallel Processing*. 2016, pp. 360–369, IEEE.

PAPER II

Ivan Walulya, Bapi Chatterjee, Ajoy K. Datta, Rashmi Niyoliya and Philippas Tsigas

Concurrent Lock-free Unbounded Priority Queue with Mutable Priorities

In Proceedings of the
20^{th} *International Symposium, Stabilization, Safety, and Security of Distributed Systems*
To appear, LNCS, Springer 2018.

3

Concurrent Lock-free Unbounded Priority Queue

Abstract
The priority queue with DELETEMIN and INSERT operations is a classical interface for ordering items associated with priorities. Some important algorithms, such as Dijkstra's single-source-shortest-path, Adaptive Huffman Trees, etc. also require changing the priorities of items in the runtime. Existing lock-free priority queues do not directly support the dynamic mutation of the priorities. This paper presents the first concurrent lock-free unbounded binary heap that implements a priority queue with mutable priorities. The operations are provably linearizable. We also designed an optimized version of the algorithm by combining the concurrent operations that substantially improves the performance. For experimental evaluation, we implemented the algorithm in both C/C++ and Java. A number of micro-benchmarks show that our algorithm performs well in comparison to existing implementations.

3.1 Introduction

A priority queue orders a set of items by a numerical cost – often called *priority* – associated with each item. In its most general form, a priority queue abstract data type (ADT) is defined by two operations – INSERT and DELETEMIN. An INSERT($k, elem$) inserts an item $elem$ with priority k and a DELETEMIN() removes an item with the highest priority from the set of objects. Priority queues are widely used at operating system kernels as well as in user-space. Some well-known applications are discrete event simulations [1], graph search [2], operating systems schedulers [3], SAT solvers [4] and many others. Several of them, such as Dijkstra's single-source-shortest-path (SSSP) algorithm [5], Adaptive Huffman Trees [6], etc. require updating the priorities after inserting the items. In today's application settings, the underlying datasets grow immensely at runtime necessitating the employed data structure to be adaptable to size variations.

At the same time, the proliferation of multi-core systems have essentially mainstreamed the concurrent data structures. Concurrent data structure designs are evaluated on consistency (correctness) and progress guarantees in addition to scalability with increasing number of processing threads. The most common consistency framework used in concurrent settings is linearizability [7], which relates a concurrent execution on an object to its sequential specification. Linearizability requires that an operation appears to take effect instantaneously at a single linearization point between the operation's invocation and its response.

Consistency may be trivially achieved using mutual exclusion locks that serialize the access to the entire data structure, also called coarse-grained locking. However, it severely limits the concurrent operations. Even if the number of locks increase, i.e. fine-grained locking, they are still vulnerable to pitfalls such as deadlock, priority inversion and convoying. An alternative approach is lock-free implementation. In a lock-free concurrent data structure, at least one non-faulty processing thread is guaranteed to complete its operation in a finite number of steps. Effectively, lock-free data structures foster both scalability and progress guarantee. A stronger progress guarantee is wait-freedom, which ensures that all the non-faulty processes finish their operations in a finite number of steps. However, most often wait-freedom results in poor performance. Another approach to implement consistent concurrent data structure is using software transactional memory (STM) [8]. However, the performance of such implementations largely depends on the design of the STM. Unsurprisingly, using STM to design concurrent data structures has often resulted in unacceptable performance [9].

Thus, an efficient and scalable *unbounded* concurrent lock-free data structure implementing a *mutable priority queue*, i.e. one which offers updating priorities of items dynamically, is highly sought-after in a large number of applications.

Based on the employed data structure, a priority queue implementation can be categorized primarily as: (a) heap[*]-based, and (b) skip-list-based.

The previous attempts on heap-based concurrent priority queues have largely been blocking (lock-based) or impractical non-blocking designs. Hunt *et al.* [10] presented a fine-grained lock-based heap, which locks each node separately and operations release and re-acquire locks after each step in bubble-up to prevent deadlocks with concurrent bubble-down operations. Tamir *et al.* [11] extended the work of [10] by including operations, called CHANGEKEY, to update the priority of items. The focus of their work is on the CHANGEKEY operations, which they show that improves the overall performance of Dijkstra's SSSP algorithm.

The first attempt to implement a non-blocking concurrent heap was by Herlihy [12]. However, this wait-free algorithm required copying the entire heap making the implementation inherently sequential and of little practical interest. Barnes [13] proposed a wait-free algorithm to address the drawbacks of Herlihy. His definition of the wait-free property is different from the generally accepted definition. Additionally, no implementation of this algorithm exits. Israeli *et al.* presented a wait-free algorithm for heap-based priority queues [14] which utilizes atomic primitives[†] that are not implemented in existing hardware platforms.

Dragicive *et al.* [9] designed a lock-free heap that uses STM for concurrency control. Their design offered poor performance due to the overhead of the STM. We point out that all the previously available concurrent heaps are bounded to a fixed size allocated at the initialization. There are available works on skip-list-based concurrent priority queues – Shavit *et al.* [15], Tsigas *et al.* [16], etc. Alistarh *et al.* [17] proposed an approximate DELETEMIN operation in skip-lists. However, the skip-list-based implementations face difficulty to implement the algorithms that require mutable priorities at the runtime: observably, the overall performance of the algorithm degrades [11].

We present CoMPiQ - a **Co**ncurrent lock-free unbounded heap-based **M**utable **Pri**ority **Q**ueue. The Table 3.1 summarily contrasts our contributions with the relevant existing works.

In the paper, first we present the system model and the sequential specification of the heap data structure (Section 3.2). Then, we describe the lock-free design of the heap in detail (Section 3.3). We present the proof of linearizability and lock-freedom of the concurrent operations (Section 3.4). We implemented

[*] In this work, by a heap we mean a binary heap.

[†] SC2 which validates and writes to two disjoint memory locations atomically

Table 3.1: Concurrent Priority Queues

Paper	Data Structure	Progress Guarantee	Mutable Priority	Unbounded	Practical Implementation
Herlihy [12]	Heap	Wait-free	No	No	No
Hunt *et al.* [10]	Heap	Lock-based	No	No	Scales poorly
Shavit *et al.* [15]	Skip-list	Lock-free	No	Yes	Yes
Tsigas *et al.* [16]	Skip-list	Lock-free	No	Yes	Yes
Dragicive *et al.* [9]	Heap	Lock-free	No	No	Scales poorly
Tamir *et al.* [11]	Heap	Lock-based	Yes	No	Yes
CoMPiQ	Heap	Lock-free	Yes	Yes	Yes

the algorithm in both C/C++ and Java. We describe the micro-benchmarks that we used to evaluate the algorithm, wherein we also discuss the performance with respect to the design optimizations. Our experiments demonstrate that the presented algorithm performs well in comparison to the existing counterparts (Section 3.5).

3.2 Preliminaries

We consider an asynchronous shared memory system with a finite set of n *processing threads* $p_1, ..., p_n$ where n may exceed the number of physical processors. In addition to the atomic `read` and `write` instructions, the system supports *Compare-And-Swap* (`CAS`) atomic read-modify-write instructions. The `CAS(address, old, new)` instruction checks if the current value at a memory location (`address`) is equivalent to the given value `old`, and only if true, changes the value of `address` to the new value (`new`) and returns `TRUE`; otherwise the memory location remains unchanged and the instruction returns `FALSE`.

The ADT *mutable priority queue* is defined by the following operations:

- INSERT($k, elem$): An INSERT($k, elem$) inserts an item $elem$ with priority k to the heap. We assume that k belongs to a totally ordered set. INSERT is typically a void procedure, however, we return a cross-reference to the insert item instance which can be used in the CHANGEKEY procedure[‡].In case there is an item $elem'$ available in the heap with the same priority k, the item $elem$ gets inserted and the two items $elem$ and $elem'$ can have arbitrary order by their indexes. Thus, the heap allows items with duplicate priority.

- DELETEMIN(): A DELETEMIN removes an item with highest priority

[‡] In our implementation, the INSERT operations never returns a `null` or fails to make any change due to the reason of finding the heap *full*. The heap is never full as long as we have sufficient system memory available.

from the heap and returns that item itself. DELETEMIN returns a special item `EMPTY` making no changes in the heap, if there are no items in the heap.

- CHANGEKEY(it, k_2): A CHANGEKEY(it, k_2) changes the heap so that an item $elem$ referenced by the iterator it, if existing in the heap, is placed at the priority k_2. It returns `EMPTY` if the item referenced by it was deleted from the priority queue.

In our work, a *heap* data structure implements a mutable priority queue. A heap is implemented by way of a *resizable array*. Thus, it contains items that allow for random access using a non-negative index. The array is considered virtually divided in *levels*. In the array, the *root* of the heap occupies the index 1 and is considered to be at the level 0. The *left* and the *right* children of the item at the index i are at the indexes $2i$ and $2i + 1$, respectively. We have considered a *minheap*, which means that the heap maintains the following *heap property*.

Heap property: An item $elem_1$ with priority k_1 has higher priority than the item $elem_2$ with priority k_2, if $k_1 \leq k_2$. Thus, a parent always has a *smaller* priority compared to its children and the root has the highest priority. Moreover, no item exists at level l unless the level $l - 1$ is completely full.

To demonstrate the correctness of our concurrent heap design we verify the safety and liveness properties. The safety property that we use is *linearizability* [7], whereas, the liveness is proved as *lock-freedom* [18].

Lock-free Implementations utilizing `CAS` are prone to the ABA problem [19]: a thread P reads a value A from a shared memory location, a concurrent thread $\hat{P}$ changes the value to B and then $\hat{P}$ or another thread changes it back A; when P executes a `CAS` instruction on the location, it succeeds erroneously as if the location has not been changed since last read by P. Several memory management solutions have been proposed to address the ABA problem [19, 20]. For ease of exposition, we assume the availability of a non-blocking memory management and garbage collection.

3.3 Algorithm

Our heap implementation utilizes the lock-free dynamic resizable arrays [21] as the underlying container, which offers both unbounded storage and lock-free progress guarantees. The ADT operations consist of a series of steps, such as modifying the size and then appending an item to the heap, or swapping the item at the root with the item at the bottom, or for that matter swapping any two items in case of a CHANGEKEY, followed by restoring the heap property. Each step comprises of at least one atomic primitive execution over a shared memory word.

The procedures HEAPIFYUP and HEAPIFYDOWN restore the heap property.

In order to achieve lock-free synchronization on concurrent access, we apply the *cooperative technique* described by Barnes [22]. The main idea is to detach operations from the executing threads. A thread that wishes to execute an operation on a slot of the array, creates a description of the work that it needs to perform, and writes the descriptor on the slot: we call it *marking* the slot. The operation can be completed by any thread that encounters the descriptor, which comes handy to ensure lock-freedom if the thread that initiated the operation is delayed or crashes.

Please note that *marking is not locking* a slot. It can be thought as shutting the door of a slot after putting down the description of all that is to be done inside. Thus any concurrent thread instead of busy waiting at the door actually carries the description with itself and tries to finish the work initiated by another thread in case that thread could not finish in time.

In our design, we maintain a *global descriptor* which encapsulates the current size of the heap and allows atomic modification of the size value and the associated heap slots with a sequence of CAS instructions. Additionally, we use descriptor objects at the slots during HEAPIFYUP and HEAPIFYDOWN calls. The threads calling HEAPIFYUP or HEAPIFYDOWN synchronize by way of executing CAS at these descriptors.

```
1: type Heap {
2:     Slot *vdata[][];
3:     Info *hdescr;
4: }
5: OpType {HPUP, HPDOWN};
                        ▷ Heap initialized
6: Heap* heap ← ⟨vdata, ⟨1, null⟩⟩
```

(a)

```
1: type Info {
2:     bool pending;
3:     size_t size;
4:     size_t pos;
5:     OpType op;
6:     Elem *old, *new;
7:     Info *lup, *rup;
8: }
```

(b)

```
1: type Slot {
2:     Elem *elem;
3:     Info *info;
4: }
5: struct Elem {
6:     value_t key;
7:     T *item
8: }
```

(c)

Figure 3.1: Type definitions for the heap structure, Descriptors and initialization.

Data types and heap initialization are given in Figure 3.1. The Heap structure holds pointers to the data storage arrays and a descriptor object, Figure 3.1a - section 3.3 to 4. A descriptor object, Figure 3.1b, maintains information about the state including the current size of the heap. Therefore, we initialize the heap with a dummy descriptor object with size 1, Figure 3.1a - section 3.3. To store auxiliary data with the priorities, our design maintains the heap as an array of pointers to item nodes. Each slot in the heap has a pointer elem to an Elem and a pointer info to an Info object which records the *state* of the slot: *stable* or *transient* due to an update, Figure 3.1c. An Info descriptor stores enough information, such that a thread encountering a slot in a transient state can help advance the operation.

3.3.1 Lock-free ADT Operations

The mutable priority queue operations in the lock-free heap are shown by flow-charts in Figures 3.2 and 3.4. The main procedures called by these operations are shown in Figures 3.3, 3.5 and 3.6. The pseudo-codes of each of the operations, their subroutines, and detail descriptions thereof are presented in the extended version of the paper [23]. For ease of exposition, the flow-chart based presentation of the algorithm is recursive. However, our implementation is fully non recursive as presented in the pseudo-code in the [23].

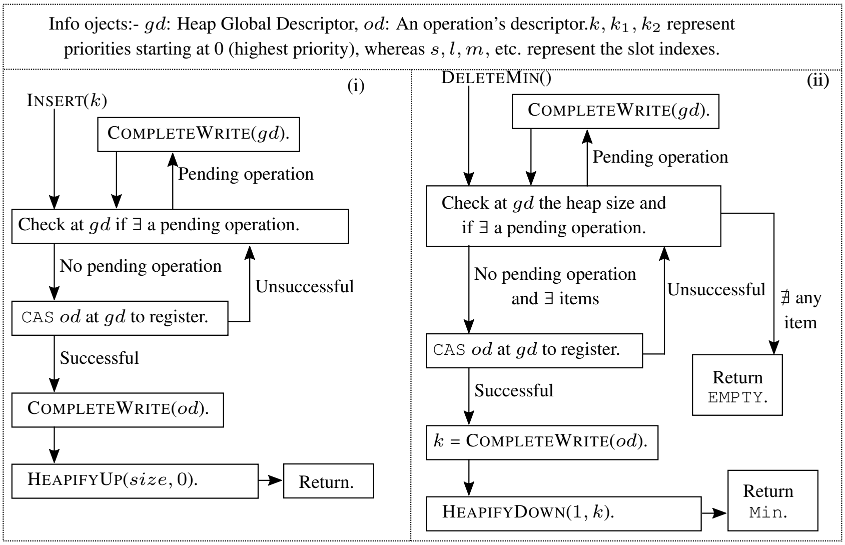

Info ojects:- gd: Heap Global Descriptor, od: An operation's descriptor.k, k_1, k_2 represent priorities starting at 0 (highest priority), whereas s, l, m, etc. represent the slot indexes.
(i)
INSERT(k)
COMPLETEWRITE(gd).
Pending operation
Check at gd if ∃ a pending operation.
No pending operation
Unsuccessful
CAS od at gd to register.
Successful
COMPLETEWRITE(od).
HEAPIFYUP($size$, 0).
Return.
(ii)
DELETEMIN()
COMPLETEWRITE(gd).
Pending operation
Check at gd the heap size and if ∃ a pending operation.
No pending operation and ∃ items
Unsuccessful
∄ any item
CAS od at gd to register.
Return EMPTY.
Successful
k = COMPLETEWRITE(od).
HEAPIFYDOWN(1, k).
Return Min.

Figure 3.2: INSERT and DELETEMIN operations in CoMPiQ.

The INSERT and DELETEMIN operations, Figure 3.2 (i) and (ii), start with an attempt to modify the size of the heap, this is achieved by *registering* the operation by way of executing a CAS to write its descriptor at the heap's global descriptor. That initiates the *preliminary phase* of the operation. The registered operation is considered pending until it is ready to call the procedures for restoring the heap property. The threads that encounter this operation, can help complete the preliminary phase.

The steps to complete the preliminary phase are taken in the procedure COMPLETEWRITE, see Figure 3.3. COMPLETEWRITE first fixes the bottom of the heap and then depending upon the type of restoration required: HPUP or HPDOWN, release the root or bottom. This procedure helps in scaling the method because it releases one end of the heap as soon as the preliminary phase is completed. In case of DELETEMIN operation calling COMPLETEWRITE, it returns the priority of the bottom-most item in the heap.

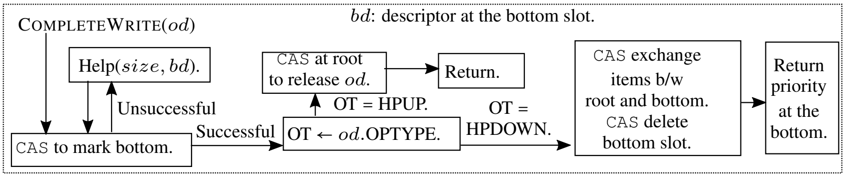

COMPLETEWRITE(od)
bd: descriptor at the bottom slot.
Help($size, bd$).
Unsuccessful
Successful
CAS to mark bottom.
CAS at root to release od.
Return.
OT = HPUP.
OT ← od.OPTYPE.
OT = HPDOWN.
CAS exchange items b/w root and bottom. CAS delete bottom slot.
Return priority at the bottom.

Figure 3.3: COMPLETEWRITE procedure.

A CHANGEKEY operation, Figure 3.4, starts with checking the size of the heap at the global descriptor to verify if the item with the priority that it desires to change exists in the heap. Thereafter, it attempts to register itself by marking the slot of the item, and calls HEAPIFYUP or HEAPIFYDOWN as needed. If the marking fails, it helps the operation that would have marked the slot and thereafter reattempts marking.

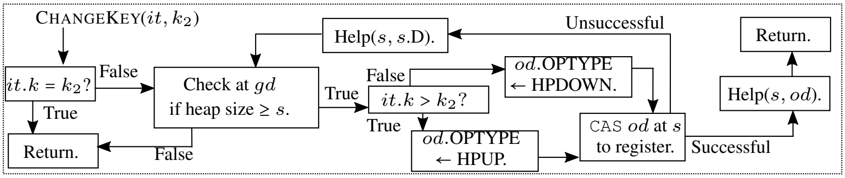


Figure 3.4: CHANGEKEY operation in CoMPiQ.

In the Figure 3.5, the procedures HEAPIFYUP and HEAPIFYDOWN are shown. They take two inputs: the index of the source slot where it starts and the priority of the destination. HEAPIFYUP keeps on exchanging the item with its parent up the heap until the destination priority is set at the slot such that heap property is restored. On the other hand, HEAPIFYDOWN traverses down the heap to do the same. To exchange the item of the current node with that of either the parent or a child, a CAS is used to first put a descriptor over there and thereafter exchange is done atomically. If CAS fails then HELP is called to first help the obstructing operation and then reattempt. The helping procedure ensures lock-freedom.

The HELP call is all about synchronization between concurrent HEAPIFYUP and HEAPIFYDOWN procedures. At a conflict, HEAPIFYDOWN is given priority. HEAPIFYUP allows the HEAPIFYDOWN to gain ownership of a child slot. This is done by marking the slot with a so called flat descriptor that stores the old information as well. This information is carried by the descriptor at the heap

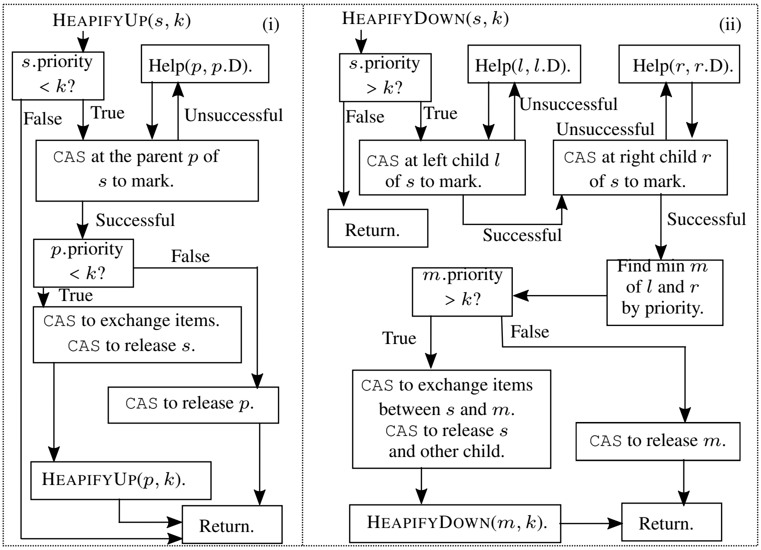


Figure 3.5: HEAPIFYUP and HEAPIFYDOWN procedures.

slots, thereby other concurrent operations help accordingly. A HEAPIFYDOWN after completing its own task, restores the information of HEAPIFYUP if that existed at the slot previously.

Please note that, we compare the items at the slots according to their priorities. Moreover, the higher the value of a priority, the lower is the priority as per the min-heap property.

3.3.2 Design Optimizations

We add two optimizations: (1) “bit-reversal” to ensure that the consecutive INSERT operations traverse different subtrees up the heap to restore heap property [10]. (2) Elimination of INSERT by handing the items off to the concurrent DELETEMIN operations, instead of having the DELETEMIN uproot an item out of position from the end of the heap. An eliminated INSERT operation can return immediately without even attempting to register itself. Below a brief description of the elimination technique is given.

Elimination Optimization: We observe that the DELETEMIN operation lifts an item from the bottom slot in the heap and heapifesDown the heap, while as the INSERT operation appends an item to the end of the heap and heapifies Up the heap. Therefore, we can optimize by allowing the INSERT to hand-off its item to a concurrent DELETEMIN. Thus, the DELETEMIN takes an item from a pending INSERT instead of dislodging one from the end of the heap. Once an INSERT operation successfully hands-off its item, it returns without calling HEAPIFYUP.

We utilize elimination arrays as suggested by Hendlar et. al [24], with each INSERT operation having a dedicated slot in the array. The DELETEMIN operation traverses the array sequentially until it finds a pending INSERT or gets to the end of the array. If the DELETEMIN operation fails to eliminate a pending INSERT, it proceeds with displacing the last item in the heap, otherwise it continues with the item taken from the the pending INSERT as described below.

After eliminating a pending INSERT operation (lifting its item from the elimination array), the DELETEMIN compares the lifted item to the item at the root of the heap. If the lifted item has a higher priority, the DELETEMIN returns the lifted item without having to call HEAPIFYDOWN. Otherwise, it proceeds to place the lifted item and returns the item previously at the root.

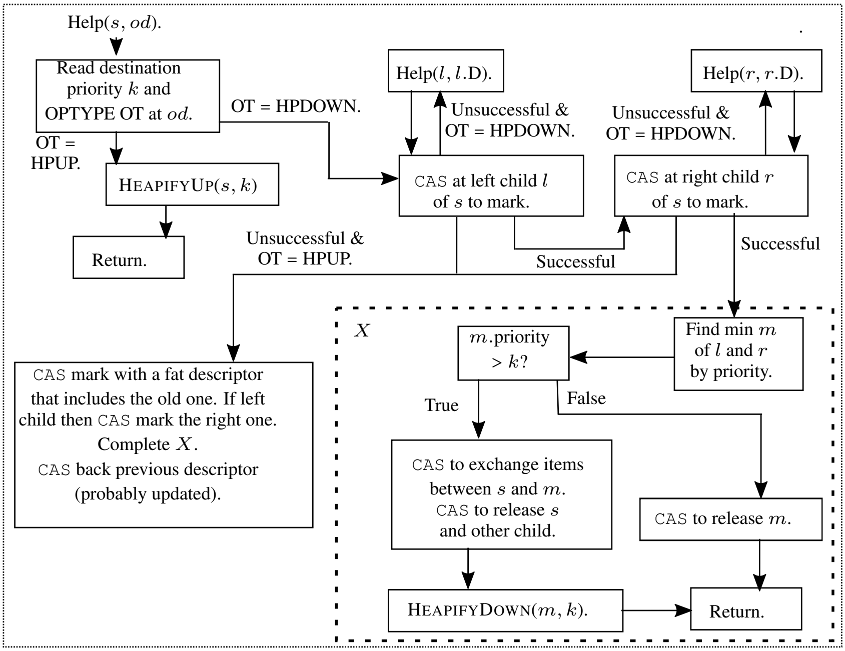

Help(s, od).
Read destination priority k and OPTYPE OT at od.
OT = HPDOWN.
OT = HPUP.
HEAPIFYUP(s, k)
Return.
Help(l, l.D).
Unsuccessful & OT = HPDOWN.
CAS at left child l of s to mark.
Help(r, r.D).
Unsuccessful & OT = HPDOWN.
CAS at right child r of s to mark.
Unsuccessful & OT = HPUP.
Successful
Successful
CAS mark with a fat descriptor that includes the old one. If left child then CAS mark the right one. Complete X. CAS back previous descriptor (probably updated).
X
m.priority $> k$?
Find min m of l and r by priority.
True
False
CAS to exchange items between s and m. CAS to release s and other child.
CAS to release m.
HEAPIFYDOWN(m, k).
Return.

Figure 3.6: HELP procedure in CoMPiQ.

3.4 Correctness Proof

Theorem 1. *The ADT operations implemented by CoMPiQ are linearizable.*

To prove linearizability, we define the *linearization point* of each ADT operation. We order the operations, which have *definitely returned*, according to their linearization points, thus obtaining a *sequential history* of execution. Thereby, it is shown that the *concurrent history* of execution of a finite number of ADT operations is *equivalent* to a sequential history. By induction, any concurrent execution is thus shown to be equivalent to a definite sequential history. Additionally, we need to show that each of the ADT operations necessarily brings the heap in a state that satisfies the heap property before its completion.

Proving lock-freedom requires that infinitely often some non-faulty processing thread will complete its operation in a finite number of steps regardless of the failed or delayed threads. To prove lock-freedom, we shall show that no operation *op busy-waits* (by holding locks, for example) when *obstructed* by a concurrent operation *op′* and goes to help *op′* to finish its operation. It may well be that *op* is repeatedly obstructed by concurrent operations op_i, $i \in \{1, 2, ...\}$ never letting it complete its own operation, however, by virtue of the same protocol it is proved that at least one non-faulty thread completes its operation in finite number of steps. Under the constraints of space, we sketch the two proofs here.

Proof. The linearization points of the ADT operations are the following:

1. INSERT: An INSERT($k, elem$) operation begins with checking the global descriptor gd of the heap. If it finds that there is a pending concurrent operation, it goes to first help that by calling a COMPLETEWRITE(gd). Thus, an INSERT starts taking steps for itself only after the successful CAS that registers it. After that, INSERT calls COMPLETEWRITE to write its descriptor, and on completion, a HEAPIFYUP is called. The HEAPIFYUP finally makes the item $elem$ part of the heap with the successful CAS. Thus for an INSERT operation that successfully performs this CAS step, its linearization point is there. In case it gets helped by a concurrent operation the successful CAS that finally makes the item $elem$ part of the heap is the linearization point. However, in either case the CAS of linearization point is performed before the completion of INSERT. For detail, see [23]. Clearly, the linearization point of an INSERT operation is between its invoke and return.

2. DELETEMIN: Depending on the return, there can be following cases:

   (a) DELETEMIN returns EMPTY: The linearization point is at the atomic read step where the DELETEMIN reads that the heap-size is 1 i.e. it contains a the dummy descriptor object.

(b) DELETEMIN returns an item $elem$: In this case, where it registers itself by a successful CAS at *gd*, it is guaranteed that it will itself complete if not obstructed, or will get helped by a concurrent operation. Also, once the descriptor *od* is written, a concurrent INSERT or DELETEMIN operation treats the root of the binary heap as deleted. Thus, the return of the concurrent operation treats the DELETEMIN that successfully put the descriptor as if it had already returned. Therefore, the linearization point of a DELETEMIN in this case is at the step where it registers itself.

Thus, the linearization point of a DELETEMIN is between invoke and return.

3. CHANGEKEY: Similar to an INSERT, a CHANGEKEY terminates after its item is relocated from one slot to another by way of calling a HEAPIFYUP or a HEAPIFYDOWN. The CAS where the item will be visible to every operation with its modified priority is the linearization point of a CHANGEKEY operation. When a CHANGEKEY returns without making any changes in the heap, its linearization point is at the atomic read step where it reads the size of the heap.

Furthermore, it can be observably determined that no operation returns before the heap property is restored by calling either a HEAPIFYUP or a HEAPIFYDOWN procedure. Any write on a shared memory word in the algorithm happens by way of only a CAS. A dummy descriptor at the root ensures that no null pointer is dereferenced. Clearly, the heap invariant is maintained across the linearization points of the ADT operations. □

Theorem 2. *The ADT operations implemented by CoMPiQ are lock-free.*

Proof. We can observe in the algorithm that a concurrent write at any shared word happens only using a CAS. Further, if op_1 and op_2 are any two concurrent operations, at no point after the failure of a CAS, op_1 or op_2 repeats the same CAS step without helping the other operation. This methodology ensures that at least one of the processes do finish its operation in a finite number of steps. □

3.5 Evaluation

In this section, we present an evaluation of our lock-free heap using microbenchmarks and a parallelized implementation of Dijkstra's SSSP algorithm described in [11]. For the micro-benchmark, we compare the *heap-based* concurrent priority queue implementations described below:

1. **CoMPiQ:** Our implementation of a lock-free heap as described in section 3.3 with elimination optimization.

2. **LB-Heap:** A fine grained locking implementation by Hunt et. al. [10]. Releases locks and re-aquires them on each iteration of the heapifyup operation to prevent deadlocks with concurrent heapifydown operation.

3. **Champ:** . Modification of `LB-Heap` to remove redundant unlock and lock operations. Deadlocks are prevented using `tryLock()` in the heapifyup and only releasing already acquired locks if a subsequent `tryLock()` fails. We received Java code from the authors, reimplemented it in C/C++ and included the exponential back-off and bit-reversal scheme [10] to reduce contention.

4. **STL-Heap:** The C++ STL `std::priority_queue<T>` made thread-safe with a single global lock (coarse-grained locking). We experimented with multiple lock synchronization primitives, however the mutex was the best performing.

Methodology: We performed our evaluations on a dual-socket server with a 3.4 GHz Intel E5-2687W-v2 having 16 physical cores (32 hardware threads by hyper-threading), 16 GB of RAM, running Ubuntu 13.04 Linux. All the algorithms in the micro-benchmark were implemented in C/C++, compiled with gcc version 4.9.2, -O3 and run as part of the ASCYLIB library [25]. Additionally, we pin software threads onto hardware cores so as to leverage CPU affinity within sockets. We utilize SSMEM [25] with epoch-based garbage collection [26].

We measured throughput as Million operations per second (Mops/s), while varying the number of threads, initial heap size and contention (parallel-work: work performed by threads outside accessing the heap). We do not expect the concurrent heap to be repeatedly accessed by threads without work in between so we simulate this work by varying parallel-work (pw), thus giving a more realistic evaluation than just stress testing. The lower the parallel-work, the more contention experienced by threads accessing the heap. We varied the number of items in the heap before starting the measurements with ($k \in \{2^{10}, 2^{17}, 2^{20}\}$). Operations on the heap are randomly chosen with a distribution of 50% Insert and 50% DeleteMin operations. Priorities for inserted items where selected uniformly at random from the range of all 64-bit integers. Each experiment run for 5 seconds, we present the average over 6 runs for each parameter configuration.

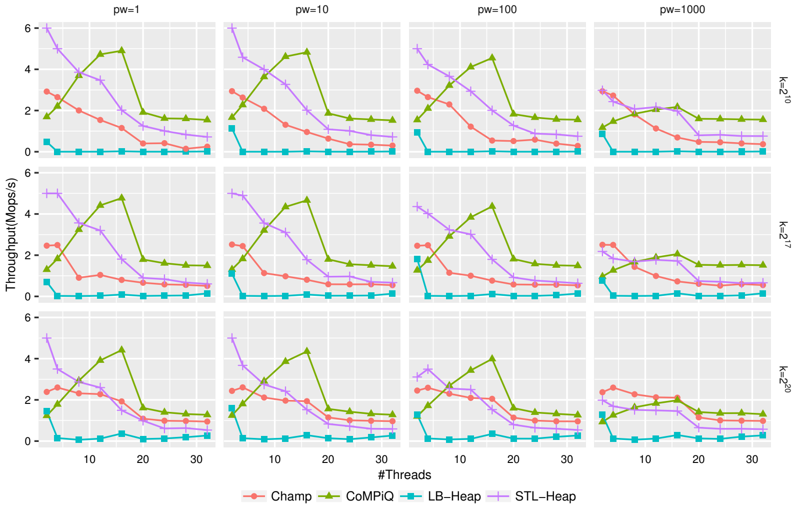


Figure 3.7: Throughput `Insert`/`DeleteMin` operations executed uniformly and randomly independent on the heap implementations as we vary the number of threads and parallel-work (pw) in CPU cycles. `K` represents the initial number of items in the heap.

Throughput: Figure 3.7 presents measured throughput in Million operations per second (Mops/s) as we vary the contention in parallel-work (pw) in CPU cycles and the number of threads. We present three sets of graphs for three initial sizes of the heap ($k \in \{2^{10}, 2^{17}, 2^{20}\}$), this is to show the effect of heap size on the execution time of the operations.

The figure shows that with small initial size 2^{10} (row 1, Figure 3.7), at low thread contention, the single lock implementation STL-Heap outperforms other implementations. This attributed to the low overheads incurred by STL-Heap using mutual exclusion, and high overheads on both the multi-lock LB-Heap, Champ and lock-free CoMPiQ. Similar observation about the single-lock implementation was made in previous works [10, 16].

Champ optimizes on the heapifyup operation of LB-Heap by removing redundant `unlock` and `re-lock` operations in uncontended cases, however, in case of contention, failure to acquire a lock, results in releasing locks held, and an attempt to reacquire them. On modern architectures with private caches, a process that releases a lock has a much higher probability of reacquiring the lock if it attempts to acquire the lock immediately. Thus, an implementation of Champ was showing similar performance figures as LB-Heap. We modified the implementation by adding exponential back-off between releasing a lock, and attempts to re-acquire the same lock. This is the major reason for the performance differences between Champ and LB-Heap.

As we increase the number of threads, contention for the lock increases and performance deteriorates. We observe that the lock-free algorithm with elimination (CoMPiQ), scales up as we increase the thread count. Elimination increases the concurrency exploited by the operations as an INSERT completes without contending for the global descriptor or creating contention within the heap with HEAPIFYUP operations. All implementations degrade in performance as we deploy more than 16 threads due to communication overheads across sockets. We still observe that CoMPiQ offers better throughput on multi-socket executions.

As we increase the initial size of the heap (height of the heap), "bit-reversal" allows for more concurrency, and thus reducing the impact of synchronization overhead on the performance. In this regard, we see that for heap size 2^{20} the performance of the single-lock implementation drops significantly relative to other implementations with increasing thread count. The CoMPiQ performs best with increased opportunities for concurrency and reduced contention on the heap.

Increasing parallel work ($pw \in \{1, 10, 100, 1000\}$) affects the lock-based implementations more than the lock-free implementations because the concurrency overheads no longer dominate performance, but concurrency. Thus, CoMPiQ still outperforms other implementations.

Discussion: Key observations are that – the heap is an inherently sequential data structure and even the most efficient implementation is still outperformed significantly by a single thread executing on a sequential heap for low levels of parallel-work. However, as the parallel work increases, the benefit of increasing concurrency becomes more significant. Additionally, bit-reversal offers more opportunities for disjoint-access allowing better exploitation of concurrency on larger size heaps to offset synchronization overheads. This is less significant in smaller heaps as successive Insert operations conflict on the paths to the root. The root and the size variable create a severe bottleneck in both blocking and non-blocking implementations, as all operations have to modify the size variable, while all DeleteMin operations modify the size and also block the root for exclusive access. CoMPiQ uses elimination to reduce on the contention at the bottleneck, thus resulting in better performance.

Parallel SSSP: One important application of priority queues that utilizes the changeKey operation is the Dijkstra's SSSP algorithm. To evaluate the performance of **CoMPiQ**, we implemented CoMPiQ as part of the benchmark suite received from [11] which included a parallel implementation Dijkstra's

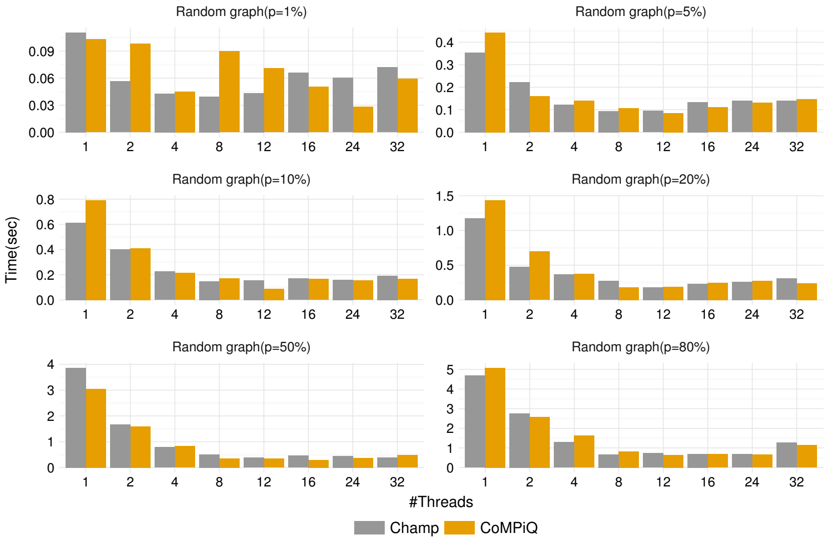


Figure 3.8: Runtimes for parallel Dijkstra's SSSP for different random graphs

algorithm and **Champ** which is the only other implementation that supports `changeKey` operation. The parallel Dijkstra's SSSP algorithm availed in the benchmark relies heavily on locks to ensure correctness, with this in mind, we plugged in our implementation without modifying the parallel SSP algorithm for fair comparison. A more optimistic parallel implementation of Dijkstra's SSSP algorithm is left as future work. In the benchmark, running time is measured over several input graphs and number of execution threads. Each input graph is generated with 10,000 vertices, with edges occurring independently randomly with some probability p and a random weight in the range [1-100]. The parallel Dijkstra's SSSP algorithm and the evaluated priority queues are implemented in Java.

Figure 3.8 shows that the **CoMPiQ** performs comparably with **Champ**. This implies that overheads incurred to ensure lock-freedom do not degrade performance of CoMPiQ when used in parallel applications. Note that node locks are used in this parallelization, thus, as pointed out earlier, we anticipate significant performance improvements with a more optimistic parallelization, that uses atomics to update node weights. We only considered implementations that support the changeKey operation. Please refer to [11] for an evaluation involving skiplist-based priority queues that do not support `changeKey`.

3.6 Conclusion

In this paper, we presented a novel algorithm for an array-based unbounded concurrent lock-free heap. The heap implements a priority queue interface with the additional facility of changing the priority of an item in the runtime. Our work contributes to many important applications, which use the priority queue ADT and need to modify the priority of the items dynamically, in a definitive way. Our micro-benchmark based experiments demonstrated that our algorithm performs well in comparison to similar existing algorithms that use locks.

With array-based implementations, it is trivial to represent a d-ary heap, however, implementation of a concurrent multi-way heap creates new challenges. The multi-way heaps lower the traversal cost by reducing the height of the tree, but increase the synchronization overhead as an operation attempts to determine the priorities of all the d-children. The techniques introduced in this article may be useful in implementing non-blocking versions of the heap-ordered d-ary heaps.

Bibliography

[1] Richard M Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.

[2] Robert Clay Prim, "Shortest connection networks and some generalizations," *Bell Labs Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[3] Gary J Henry, "The unix system: The fair share scheduler," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1845–1857, 1984.

[4] Leonardo De Moura and Nikolaj Bjørner, "Z3: An efficient smt solver," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[5] Edsger W Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[6] Jeffrey Scott Vitter, "Design and analysis of dynamic huffman codes," *Journal of the ACM*, vol. 34, no. 4, pp. 825–845, 1987.

[7] Maurice Herlihy and Jeannette M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[8] Nir Shavit and Dan Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[9] Kristijan Dragicevic and Daniel Bauer, "Optimization techniques for concurrent stm-based implementations: A concurrent binary heap as a case study," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2009, pp. 1–8.

[10] Galen C. Hunt, Maged M. Michael, Srinivasan Parthasarathy, and Michael L. Scott, "An efficient algorithm for concurrent priority queue heaps," *Information Processing Letters*, vol. 60, no. 3, pp. 151–157, 1996.

[11] Orr Tamir, Adam Morrison, and Noam Rinetzky, "A Heap-Based Concurrent Priority Queue with Mutable Priorities for Faster Parallel Algorithms," in *Proceedings of the International Conference on Principles of Distributed Systems*. 2016, vol. 46, pp. 1–16, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[12] Maurice Herlihy, "A methodology for implementing highly concurrent data objects," *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 5, pp. 745–770, 1993.

[13] Greg Barnes, *Wait-free Algoritzms for Heaps*, Department of Computer Science and Engineering, University of Washington, 1994.

[14] Amos Israeli and Lihu Rappoport, "Efficient wait-free implementation of a concurrent priority queue," in *Proceedings of the International Workshop on Distributed Algorithms*. 1993, pp. 1–17, Springer.

[15] Nir Shavit and Itay Lotan, "Skiplist-based concurrent priority queues," in *Proceedings of the International Parallel and Distributed Processing Symposium*. 2000, pp. 263–268, IEEE.

[16] Håkan Sundell and Philippas Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *Journal of Parallel and Distributed Computing*, vol. 65, no. 5, pp. 609–627, 2005.

[17] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit, "The spraylist: A scalable relaxed priority queue," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2015, pp. 11–20, ACM.

[18] Maurice Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 1, pp. 124–149, 1991.

[19] Maged M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, 2004.

[20] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas, "Efficient and reliable lock-free memory reclamation based on reference counting," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 8, pp. 1173–1187, 2009.

[21] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup, "Lock-free dynamically resizable arrays," in *Proceedings of the International Conference on Principles of Distributed Systems*. 2006, pp. 142–156, Springer.

[22] Greg Barnes, "A method for implementing lock-free shared-data structures," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 1993, pp. 261–270, ACM.

[23] Ivan Walulya, Bapi Chatterjee, Ajoy K. Datta, Rashmi Niyoliya, and Philippas Tsigas, “Concurrent lock-free unbounded priority queue with mutable priorities,” Tech. Rep. 2018:06, ISNN 1652-926X, Department of Computer Science and Engineering, Chalmers University of Technology, 2018.

[24] Danny Hendler, Nir Shavit, and Lena Yerushalmi, “A Scalable Lock-free Stack Algorithm,” in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2004, pp. 206–215, ACM.

[25] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis, “Asynchronized concurrency: The secret to scaling concurrent search data structures,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 2015, pp. 631–644, ACM.

[26] Keir Fraser, “Practical lock-freedom,” *PhD thesis, University of Cambridge*, 2004.

PAPER III

Bapi Chatterjee, **Ivan Walulya** and Philippas Tsigas

Help-optimal and Language-portable Lock-free Concurrent Data Structures

In the Proceedings of the
45^{th} *International Conference on Parallel Processing*
pp. 360-369, IEEE 2016.

4

Help-optimal and Language-portable Lock-free Concurrent Data Structures

Abstract
Helping is a widely used technique to guarantee lock-freedom in many concurrent data structures. An optimized helping strategy improves the overall performance of a lock-free algorithm. In this paper, we propose *help-optimality*, which implies that no operation step is accounted for exclusive helping in the synchronization of concurrent operations. To describe the concept, we revisit the designs of a lock-free linked-list and binary search tree and present improved algorithms. Our algorithms employ atomic single-word compare-and-swap (`CAS`) primitives.

We design the algorithms without using any language/platform specific mechanism. Specifically, we use neither bit-stealing from a pointer nor runtime type introspection of objects. Thus, our algorithms are *language-portable*. Further, to optimize the amortized number of steps per operation, if a `CAS` execution to modify a shared pointer fails, we obtain a fresh set of thread-local variables without restarting an operation from scratch.

We use several micro-benchmarks in both C/C++ and Java to validate the efficiency of our algorithms against existing state-of-the-art. The experiments show that the algorithms are scalable. Our implementations perform on a par with highly optimized ones and in many cases yield 10%-50% higher throughput.

4.1 Introduction

4.1.1 Overview

With the wide availability of multi-core processors, efficient concurrent data structures have become ever more important. Lock-free concurrent data structures, which guarantee that some non-faulty threads do finish their operations in a finite number of steps, provide robustness and better performance compared to their blocking counterparts which are vulnerable to pitfalls such as deadlocks, priority inversion and convoying in an asynchronous shared memory system.

The literature on lock-free data structures has grown sufficiently over the last decade [1–9]. Typically, *practical* lock-free designs use single-word atomic compare-and-swap synchronization primitives (henceforth referred to as `CAS`) to modify shared variables. Thus, to implement a lock-free version of a dynamic pointer-based data structure, in which (multiple) mutable links (pointers) are shared among threads in a concurrent set-up, either by design or due to necessity, one or more `CAS` executions are performed to complete a modify (add or remove) operation.

For example, in the lock-free linked-list of [1], two successful `CAS` executions are required to complete a remove operation, whereas in [3] three such executions are required for the same operation. Considering the lock-free external binary search trees (BSTs), three successful `CAS` executions are necessary to remove a node in [6], whereas in [4] and [7], four such executions are required for the same purpose. Furthermore, in [4] and [7], two successful `CAS` executions are required to add a node. Naturally, concurrent operations which modify overlapping sets of links, face each other at a stage where they would have partially completed and would still need to perform one or more `CAS` to complete. Herein, we call this situation *concurrent obstruction*.

For operations on a concurrent data structure, linearizability [10] is the most commonly used consistency framework. Intuitively, a concurrent data structure is linearizable if every execution provides time-points, called *linearization points*, between the invocation and the response of each operation, where it seems to take effect instantaneously. Thus, using a sequence of seemingly instantaneous operations, described by the *real-time order* of the linearization points, we perceive the concurrent operations displaying their sequential behaviour.

In a lock-free algorithm, often a `CAS` execution step is taken as the linearization point of an operation performing multiple `CAS`. Such a step may not necessarily be the last one. Most commonly in a remove operation, on the success of the `CAS` representing the linearization point, the target node is considered *logically removed*, [1–5]. This results in each traversal passing through a logically removed node and hence extra read steps get counted in step complexity of operations.

A well-known mechanism to deal with such situations is *helping*. Helping essentially implies that if multiple operations face concurrent obstruction or need to perform extra read while traversing over a transient deformation in form of a logically removed node, based on a fixed protocol, the pending steps of one of the operations are completed by the concurrent operations, before furthering their own course of steps. This strategy ensures lock-freedom because a non-faulty thread definitely completes its operation in finite number of steps.

In the prevalent research on lock-free data structure design, the helping mechanism now holds a center stage. In the lock-free linked-lists of [1,3], every concurrent operation offers helping to a remove operation which successfully performs the `CAS` to logically remove the target node and is yet to execute one more `CAS`. Barnes [11] proposed a helping mechanism called *cooperative technique*. The cooperative technique applied to a data structure requires a modify operation to atomically write the description of planned steps in the node whose links it targets to modify and thereby a concurrent obstructed operation ensures completion of those steps in case the original operation gets delayed. This method is applied in the BST of [4,7], where even add operations require helping.

In the lock-free BST of Natarajan *et al.* [6], the links are used much in the same way as in the linked-lists of [1,3] to modulate helping, and unlike [4,7], the add operations there do not require help. Broadly, their design provides better progress conditions for concurrent operations, which they showed experimentally. However, we notice that they put the linearization point of a remove operation at the very last `CAS` execution, which necessitates a concurrent remove operation to help a pending remove operation of the same query key, even though it does not change its return that is false. Clearly, the number of helping steps are not necessarily minimized.

A common suggestion found in the papers on lock-free data structures is that one should avoid helping during traversal by an otherwise unobstructed operation, which if the obstructing operation is not delayed, predominantly goes to wastage. The works analysing experimental performance of concurrent data structures [12–14] further emphasize on the same. Gibson *et al.* [15] showed that the amortized number of steps per operation are asymptotically equivalent irrespective of avoiding help by read operations. However, a design optimization to minimize the number of steps incurred by modify operations in helping at a concurrent obstruction is largely un-attempted.

Another noticeable characteristic of existing lock-free algorithms is that their descriptions are very close to the programming language of the sample implementations used by the authors to validate their claim of efficiency. For example, in the linked-lists of [1,3] and the BST of [6], the design is described in terms of using unused bits from a pointer which points to a memory-word aligned at a fixed boundary. This technique is popularly known as *bit-stealing* in programming parlance. The correctness proof thereof is inherently connected to bit-stealing. In Java toolkit [16], `AtomicMarkableReference` and `AtomicStampedReference` classes are used to simulate bit-stealing, but are not too popular from the performance point of view. The lock-free external BST designs of [4,7] use polymorphism, class inheritance and type introspection of objects at runtime (also known as real-time-type-information or RTTI), to describe their algorithm. The correctness proofs in these papers are presented accordingly.

In the lock-free skip-list implementation in Java [17], Doug Lea uses extra *splice nodes* to simulate the pointers masked with stolen bits. Such a node is identified with a specific assignment of one of its fields, for example, the value field of a *marker* node points to itself in [17]. A marker node stores the original pointer in its next field enabling unmasking of the pointer off any stolen bit. Lea remarks that in spite of some temporary extra nodes, this technique could still be faster for a traversal with quick garbage collection of removed nodes and is worth avoiding the overhead of extra type testing.

Usually, the lock-free implementations in C/C++, for example in [12] or [14], use their own memory allocation and garbage collection strategies to improve performance. Obviously, these implementation environments of C/C++ very much resemble one in Java and yet they entail each traversal step to unmask a pointer off a possible stolen bit. This underlines a motivation to present the lock-free algorithms that utilize temporary splice nodes and thereby achieving *language portability*. However, the efficiency of such an implementation in C/C++ remains still unexplored for the research community.

In literature, the efficiency of a lock-free algorithm is also presented in terms of the amortized step complexity per operation [3,7,8]. Often in a lock-free data structure, when a `CAS` execution in a modify operation returns false, the local variables in the thread become unusable for a reattempt. Hence, the thread needs to restart the operation from a *clean location* to get a fresh set of local variables. Usually, the first sentinel node where an operation starts from (head of a linked-list, root of a BST), is always clean. However, there can be as many as c restarts per operation if c concurrent threads access the data structure. To get a pointer to backtrack to a *local* clean location and thus restart the operation from there, improves the amortized number of steps per operation (counting both read and write). It can be interesting to use a splice node to store a pointer to a node in a local clean location and thus *locally restart* a modify operation.

The contributions of this work are the following:

1. We introduce the concept of *help-optimality* which essentially revisits the lock-free algorithms to optimize the number of CAS steps in helping at concurrent obstructions.

2. We describe help-optimal lock-free designs of a linked-list and a BST to implement Set abstract data types (ADT) which export linearizable ADD, REMOVE, and CONTAINS operations. CONTAINS are wait-free in the linked-list for a finite key space.

3. The presented algorithms do not use language specific constructs like bit-stealing or type introspection of objects at runtime and hence are *language-portable* for a programmer.

4. We also show that on a CAS failure at a conflict, the modify operations in our algorithms restart locally to optimize the amortized step complexity per operation.

5. We implement the algorithms in both C++ and Java. Our implementations perform on a par with highly optimized implementations and outperform them in many cases.

The rest of this paper is organized as follows; first, we present a simple lock-free BST algorithm as a motivation for a help-optimal design (Section 4.2). Thereafter, we present efficient lock-free algorithms of a linked-list (Section 4.3) and a BST (Section 4.4), to describe the concept of help-optimality as used in practice. Having described it algorithmically, we specify help-optimality more formally (Section 4.5). Finally, we discuss the experimental performance of the presented algorithms (Section 4.6).

4.1.2 Related Work

The first CAS-based lock-free linked-list was presented by Valois [18]. He suggested to augment every node with an auxiliary node to manage synchronization. Heller *et al.* [19] were perhaps the first to suggest that the CONTAINS operations in a concurrent linked-list must progress in a wait-free manner for a finite key space. They presented lock-based linked-list, called lazy list, to show that it favours performance. They also recommended that the CONTAINS operations in Michael's lock-free linked-list [2] should not be involved in helping. Subsequently, to the best of our knowledge, no concurrent data structure was designed in which CONTAINS operations are obstructed; interestingly, some researchers called it *conservative helping*, for example in [4]. In the lock-free internal BSTs presented by Howley *et al.* [5], Chatterjee *et al.* [8] and Ramachandran *et al.* [9] CONTAINS operations complete without helping any concurrent operation.

4.2 Help-optimality: Motivation

Let us consider a very simple lock-free design of an external BST to implement a Set ADT that exports ADD, REMOVE, and CONTAINS operations as given in Algorithm 4.1.

In this data structure, a node has two pointer fields lt and rt in addition to a key field k, see Line 1. Without ambiguity, we shall use *k* to denote a node with key *k*. The pointer fields lt and rt connect a node to its left and right children respectively, which are null in a leaf (also called external) node. In this BST, the external nodes are *data-nodes* and the internal nodes are *routing-nodes*. There is a *symmetric order* of node-arrangement - the nodes in the *left subtree* of a routing-node *k* have keys less than *k*, whereas in its *right subtree* the nodes have keys at least *k*. We denote the parent of a node *k* by $p(k)$ and there is a unique node called *root* s.t. $p(root) = null$. Each parent is connected to its children via links (we indicate the link emanating from *k* and incoming to *l* by $k \rightsquigarrow l$; we use the terms pointer and link interchangeably). The other child of $p(k)$, *i.e.,* sibling of *k*, is denoted by $s(k)$.

Pseudo-code convention: N.*ref* represents the reference to a variable N. Thus, `f`(N.*ref*) indicates passing N by reference to a method `f`. If x is a member of a class C then pc.x returns field x of an instance of C pointed by pc. dir L and dir R represent the left and right directions. `CAS`(A.*ref*, exp, new) compares A with exp and updates to new in one atomic step if A = exp and returns true; else it returns false without any update at A.

We initialize the BST with a subtree consisting of an internal node root with key ∞_1 and two children with key ∞_0 and ∞_1, where $\infty_1 > \infty_0 > |k|$ $\forall$ key k, as its left- and right- child respectively, see Figure 4.1 and line 2. The method `Search`, Line 12 to 13, is used for traversal by a data structure operation. `Search` takes variables *par*, *cur* and *k* as input which are two node-pointers and a query key, respectively. At the invocation of `Search`, *cur* points to the child of the node pointed by *par* in the direction of the subtree which can contain *k*. At the termination of `Search`, *cur* points to a leaf-node which is identified by the lt field being null.

To perform REMOVE(k), line 20 to 26, we use `Search` to arrive at a leaf-node pointed by ℓ. If *k* matches the key at ℓ, we use a `CAS` to replace ℓ with a special node with the same key, but with its rt field pointing to itself, see line 24.

```
class Node {K k; Node* lt, rt;};

root := Node($\infty_1$, Node($\infty_0$).ref, Node($\infty_1$).ref);

Dir(Node* par, K k) {return k < par.k ? L : R};

ChCAS(Node* par, Node* exp, Node* new, dir cD)
    if (cD == L) and par.lt == exp then
        return CAS(par.lt.ref, exp, new);
    else if (cD == R) and par.rt == exp then
        return CAS(par.rt.ref, exp, new);
    else return false;

GetDead(K k) {n := Node(k); n.rt := n; return n;}

IsDead(Node* leaf) {return leaf.rt == leaf;}

Child(Node* par, dir cD)
    return cD == L ? par.lt : par.rt;

Search(Node* par, Node* cur, K k)
    while cur.lt $\neq$ null do
        par := cur; cur := Child(par, Dir(par, k));

NewNod(Node* a, Node* b, K pKey)
    left := (a.k < b.k ? a : b);
    right := (a.k < b.k ? a : b);
    return Node(pKey, left, right, null);

CONTAINS(K k)
    p := root.ref; $\ell$ := root.lt;
    Search(p.ref, $\ell$.ref, k); cD := Dir(p, k);
    return $\ell$.k == k and !IsDead($\ell$);
```

Algorithm 4.1. A Simple Help-optimal Language-portable Lock-free Binary Search Tree

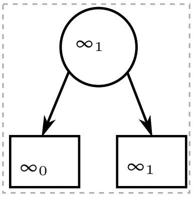


Figure 4.1: Sentinel Nodes: Simple Lock-free BST

```
   REMOVE(K $k$)
20 p := root.$ref$; $\ell$ := root.lt;
21 while true do
22     Search(p.$ref$, $\ell$.$ref$, $k$); cD := Dir(p, $k$);
23     if $\ell$.k $\neq k$ or IsDead($\ell$) then return false;
24     if ChCAS(p, $\ell$, GetDead($k$), cD) then
25         return true;
26     $\ell$ := Child(p, Dir(p, $k$));

   ADD(K $k$)
27 nd := Node($k$).$ref$;
28 p := root.$ref$; $\ell$ := root.lt;
29 while true do
30     Search(p.$ref$, $\ell$.$ref$, $k$); cD := Dir(p, $k$);
31     if !IsDead($\ell$) then
32         if $\ell$.k == $k$ then return false;
33         n := NewNod(nd, $\ell$, $\max\{k, \ell.\mathrm{k}\}$).$ref$;
34         if ChCAS(p, $\ell$, n, cD) then return true;
35     else if ChCAS(p, $\ell$, nd, cD) then return true;
36     $\ell$ := Child(p, Dir(p, $k$));
```

Algorithm 4.1. A Simple Help-optimal Language-portable Lock-free Binary Search Tree

We call such special nodes Dead. See method IsDead at line 10 which is used to identify a Dead node. If the CAS succeeds, REMOVE returns true; if k was not found or ℓ was already Dead, REMOVE returns false. For ADD(k), line 27 to 36, arriving at ℓ using Search, we use a CAS to replace ℓ with (i) a new leaf-node with key k, if ℓ was Dead and (ii) a new internal node created using NewNod, line 14 to 16, if ℓ was not Dead and k does not match at ℓ. If the CAS succeeds at line 34 or at line 35, ADD returns true; if ℓ was not Dead and contained k, it returns false. A CONTAINS(k), line 17 to 19, returns true if k is found at a leaf-node which is not Dead, else it returns false.

The main idea of this algorithm is to discard the requirement of helping by *not* cleaning out a node in a REMOVE operation, which otherwise uses multiple CAS executions. Thus, a single successful CAS is required by both ADD and REMOVE operations, much like a lock-free stack. We skip the proofs of correctness and lock-freedom of this algorithm, which are straightforward. An interested reader may take them as a simple exercise. Please note that we have not used any language specific construct to describe this algorithm.

We implemented Algorithm 4.1 in Java and compared it against the (author provided) implementation of lock-free BST of [4] and the lock-free skip-list of Java library [17]. The set-up and methodology of the experiments are described in Section 4.6. The throughput and memory usage by the algorithms to implement

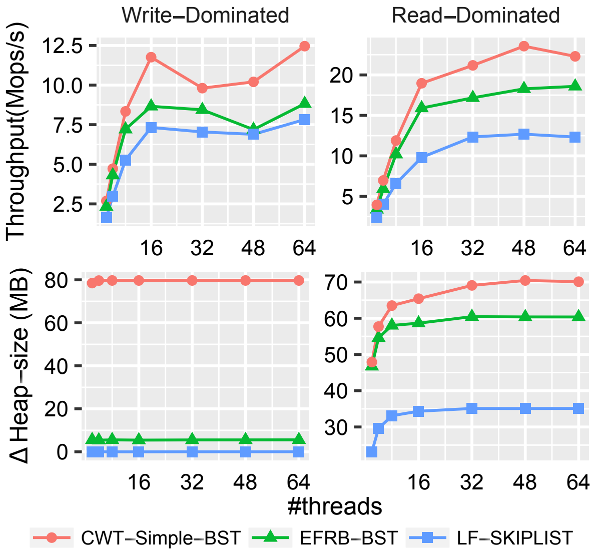


Figure 4.2: Performance graph: Lock-Free Basic BST

a Set formed by at most 2^{20} distinct keys are plotted in the Figure 4.2.

We see that this simple lock-free BST significantly outperforms state-of-the-art implementations of a skip-list and a BST. However, on account of memory footprint, it performs poorly. We get enough motivation to design lock-free data structures which optimally reduces the number of CAS executions by each ADT operation aided with optimal memory footprints.

4.3 Help-optimal Lock-free Linked-list

4.3.1 Design

We implement an ordered linked-list based Set ADT which exports ADD, REMOVE and CONTAINS operations. The pseudo-code is given in the Algorithm 4.2. A node has two pointer fields nxt and bck in addition to the key field k, see line 1. As before, we use k to denote a node with key k. The field nxt points to the successor of k, denoted by $s(k)$. We describe the use of bck later; it is null in a regular node. The predecessor of k is denoted by $p(k)$. Initially, the linked-list consists of four sentinel nodes tailNxt, tail, headNxt and head with keys ∞_1, ∞_0, $-\infty_0$ and $-\infty_1$, respectively, where $\infty_1 > \infty_0 > |k|$ $\forall$ key k. See line 2 to 5 and Figure 4.3.

We aim to reduce the number of CAS steps incurred in helping not only

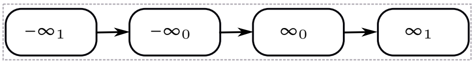


Figure 4.3: Sentinel Nodes: Lock-free linked-list

```
class Node {K k; Node* nxt, bck;};

tailNxt := Node($\infty_1$, null, null);
tail := Node($\infty_0$, tailNxt.ref, null);
headNxt := Node($-\infty_0$, tail.ref, null);
head := Node($-\infty_1$, headNext.ref, null);

Search(Node* pre, Node* nex, Node* cur, Node* suc, K k)
    while cur.k < k do
        if IsSp(suc) then cur := suc.nxt;
        else pre := cur; nex := suc; cur := suc;
        suc := cur.nxt;

CONTAINS(K k)
    c := headNext.nxt;
    while c.k < k do c := cur.nxt;
    return c.k == k and !IsSp(c.nxt);

ADD(K k)
    p := head.ref; n := headNxt.ref;
    c := headNxt.ref; s := headNxt.nxt;
    while true do
        Search(p.ref, n.ref, c.ref, s.ref, k);
        if IsSp(s) then
            while IsSp(s) do {c := s.nxt; s := c.nxt};
        else if c.k == k then return false;
        if CAS(p.nxt.ref, n, Node(k, c, null)) then
            return true;
        BckTrck(p.ref, n.ref); c := p; s := n;

BckTrck(Node* pre, Node* nex)
    nex := pre.nxt;
    while IsSp(nex) do
        pre := nex.bck; nex := pre.nxt;

IsSp(Node* c)  {return c.k == $-\infty_2$;}
```

Algorithm 4.2. Help-optimal lock-free linked-list

during traversal, which is simple, but also in the concurrent obstruction. In Algorithm 4.1 we observed that obstruction can be fully avoided in an external BST if REMOVE operations do not try to clean out the removed nodes. However, that strategy led to undesirably large memory footprint. So, the question we ask - can we overcome the drawbacks? Observing carefully, in a linked-list we can leverage the linear structure to connect the predecessor of the leftmost node to the successor of the rightmost node of a contiguous chunk of removed nodes by a single CAS and thus solve the issue. We describe it below.

At the basic level, ADD(k) in a lock-free linked list comprises - finding $p(k)$ and $s(k)$ s.t. $p(k).\mathsf{k} < k < s(k).\mathsf{k}$, allocating the node k s.t. $k.\mathsf{nxt} = s(k)$ and using a single CAS execution to swing the $p(k).\mathsf{nxt}$ from $s(k)$ to k. We have seen it in [1,3]. Similarly, REMOVE(k) comprises - finding nodes $p(k)$ and k, logically removing k using a single CAS and then swing the $p(k).\mathsf{nxt}$ from k to $s(k)$ using a CAS.

However, our target implementation as described before will require additional tricks over this basic idea. First off, in order to make the algorithm language-portable, we find a way of using *splice nodes* as Lea [17], instead of bit-stealing like [1,3]. For that, when logically removing k, we add a splice node between k and $s(k)$. We fix the key of a splice node as $-\infty_2$, where $\infty_2 > \infty_1$, by which it can be identified, see line 35 and the method IsSp at line 27.

```
REMOVE(K k)
28 p := head.ref; n := headNxt.ref;
29 c := headNxt.ref; s := headNxt.nxt;
30 r := null; spNd := null; mode := INIT;
31 while true do
32     Search(p.ref, n.ref, c.ref, s.ref, k);
33     if mode == INIT then
34         if c.k ≠ k or IsSp(s) then return false;
35         spNd := Node($-\infty_2$, s, p).ref;
36         while true do
37             if CAS(c.nxt.ref, s, spNd) then
38                 if CAS(p.nxt.ref, n, s) then return true;
39                 r := s; mode := CLEAN; break;
40             s := c.nxt; if IsSp(s) then return false;
41             spNd.nxt := s;
42     else if s ≠ spNd or CAS(p.nxt.ref, n, r) then
43         return true;
44     BckTrck(p.ref, n.ref); c := p; s := n;
```

Algorithm 4.2. Help-optimal lock-free linked-list

Secondly, to avoid eager helping during traversal by a modify operation and yet be able to clean out the logically removed nodes (along with the splice nodes succeeding them), we use two trailing node-pointers during traversal. This trick is similar to [6] used in BST. We use the trailing node-pointers to store the address of the last node, which was *not* logically removed, and its successor. Thus, at the termination of a traversal, we have reference to the predecessor, say $p(k)$, of the leftmost node of a possible contiguous chunk of logically removed nodes. Hence, when we swing the pointer $p(k)$.nxt, using a CAS, to connect either to a new node *k* for ADD or to $s(k)$ for REMOVE, zero or more logically removed nodes are cleaned out.

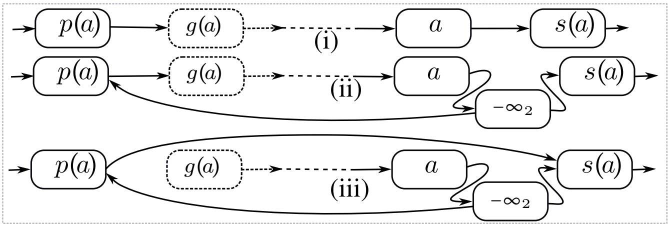


Figure 4.4: Steps of REMOVE in the linked-list.

And thirdly, to backtrack to a clean zone on CAS failures, we use the idea of back-pointers as applied in [3]. However, our approach differs from them. When allocating a splice node, we save the address of $p(k)$ in its bck field, which is always null for a regular node. Essentially, our approach is novel in the following ways - (a) we do not use an extra CAS to fix (*flagging*) the pointer $p(k)$.nxt. Given the use of trailing pointers, we do not often travel a long chain of back pointers. And (b) we do not set back-pointer of a regular node and thus, save an extra atomic write of a shared pointer. Indeed, a splice node in our algorithm splices two node paths.

The basic steps of REMOVE(k), are shown in the Figure 4.4. The node $n(k)$ denotes the first node of a possible contiguous chunk of logically removed nodes before and adjacent to *k*. In case there is no such chunk before *k*, $n(k)$ coincides with it.

We perform traversal for a modify operation using the method Search, line 6 to 9. We advance the variables *pre* and *nex* only if *suc* is not a splice node, that is when *cur* is not a logically removed node. Otherwise, we advance *cur* to the node saved at the *nxt* of the splice node *suc*. In REMOVE and ADD, at the first call of Search, the variable *pre* points to head, *nex*, *cur* point to headNxt and *suc* points to the successor of headNext, see lines 13 and 14. Thus, at the termination of a traversal, when *cur* points to a node with key *not greater than k*, *pre* points to the predecessor of the first node of a possible contiguous chunk of logically removed nodes and *nex* points to the first node of such a chunk.

To perform REMOVE(k), line 30 to 44, at the termination of a traversal, we check the key at the node pointed by c, and if *k* does not match at it or the node pointed by s is found splice (indicating node pointed by c is already logically removed), we return false, line 34. Otherwise, we perform a CAS to add a splice node between c and s to logically remove c, line 37. The steps taken up to this point are identified by a variable mode with value INIT. After this step, mode changes to CLEAN and we attempt to swing the p.nxt from n to s using a CAS at line 38. If the CAS fails, we save s as r, and perform a BckTrck at line 44 to find a fresh pair of p and n.

In the method BckTrck, line 24 to 26, we keep on traversing back, following the bck of splice nodes, until we find the first node which is not logically removed. If the call of BckTrck was due to a CAS failure caused by an ADD of a new node, added between *pre* and *nex*, it is guaranteed that the chunk of contiguous logically removed nodes must have been cleaned out. We explain it in the next paragraph.

The operation ADD(k), line 13 to 22, performs a similar traversal. At the termination of the traversal, we check if the node pointed by c is logically removed by checking whether s points to a splice node, line 17. If the node at c is not logically removed and contains the query key k, we return false; else, we find the first node succeeding it which is still not logically removed, line 18, and attempt a CAS to add the new node between p and c to return true. On a CAS failure, we perform BckTrck as explained before and reattempt the previous steps. Thus, on a successful ADD it cleans out a complete chunk of contiguous logically removed nodes.

Note that, a modify operation in Algorithm 4.2 differs from one in [1,3], in the sense that on a CAS failure at $p(k)$.nxt, we do not perform any help before reattempting. Instead of that, we selfishly attempt the CAS from a clean location. Thus, the operations are essentially *selfish* in our algorithm.

A CONTAINS operation, line 10 to 12, traverses in a wait-free manner and returns true only if the node at which it terminates, the one pointed by c, is not logically removed and contains the query key, else it returns false.

4.3.2 Correctness and Lock-freedom

It is easy to observe that the field k of a node is never modified after initialization. Scanning through the pseudo-code, we can observe that once a splice node is added at the nxt of a node, no CAS is performed at it. Further, unless the nxt of a node k is splice, it is not removed from the list. Thus, we can show that a node $p(k)$ is present in the list, when we connect a new node k or successor $s(k)$ of a removed node k to it. Additionally, we can observe that a traversal terminates with c pointing to a node which has a key greater than or equal to k in all the operations, which in turn shows that we maintain the order of node arrangement in an ADD or a REMOVE operation. At the initialization, the sentinel nodes form a valid ordered list. Hence, using induction we can prove that the ADT operations maintain a valid ordered list.

The linearization point for an unsuccessful ADD operation is at line 9 during a call of `Search`. Similarly for a successful CONTAINS operation it is at line 11, when we read c.nxt for the first time. For a successful ADD or a REMOVE operation, the linearization point lies at the first successful `CAS` execution to add a new or a splice node. For an unsuccessful CONTAINS, the linearization point is (a) just after that of the concurrent REMOVE operation which (logically) removed k, if k existed in the list at the invocation point of CONTAINS and (b) at the invocation point itself, if *k* was not present in the list at that point. The linearization point of an unsuccessful REMOVE is determined similar to an unsuccessful CONTAINS operation.

We can observe that the `CAS` to add a splice node is reattempted only if a new node is added at the nxt of *k*. Before every reattempt of a `CAS` to swing the nxt pointer of $p(k)$, in both ADD and REMOVE, we perform a `BckTrck` and a `Search` which guarantees that we have a fresh set of variables for references of $p(k)$ and $n(k)$. Hence, it is guaranteed that a modify operation can not take an infinite number of steps without a modification in the data structure. It proves the lock-freedom of the ADD and REMOVE operation. It is easy to observe that a non-faulty CONTAINS always finishes in a finite number of steps if the key space is finite and thus is wait-free.

4.3.3 Amortized Step Complexity

We can observe that the splice nodes are never adjacent. Similar to [13], we do not perform help in a CONTAINS operation. Additionally, in ADD and REMOVE as well, no step is taken for helping during traversal. On a `CAS` failure to add a splice node, we do not perform any traversal. On a `CAS` failure to add a new node or to clean out a chunk of logically removed nodes, we perform backtrack and do not start from the head. Following the same method as [3], we can show that the amortized number of steps per operation is $O(n{+}c_I)$, where c_I is the total number of concurrent operations between invocation and response of o, called *interval contention* [20] and n is the size of the list at the invocation of o. In the light of theorem 1 of [15], it is asymptotically equivalent to $O(n{+}c_P)$, where c_P is the maximum number of concurrent operations at any point in the lifetime of o, called *point contention* [21].

4.4 Help-optimal Lock-free BST

Having described a simple lock-free BST and an improved lock-free linked-list, where we do not spend any CAS execution for helping, we are ready to describe an efficient lock-free BST, in which we introduce the notion of *help-awareness*.

4.4.1 Design

The pseudo-code of the design is given in Algorithm 4.3. The symmetric order of the BST is same as that in Section 4.2. We borrow the notations from Algorithm 4.1 along with the methods Dir, Child, ChCAS, GetDead, IsDead and NewNod as they are described there. We denote the parent of $p(k)$ by $g(k)$.

The main drawback of the lock-free BST of Algorithm 4.1 was removing a node *k* by replacing it with a Dead node and not cleaning the Dead node out that caused memory-wastage. Therefore, in Algorithm 4.3 we make a REMOVE operation clean out the added Dead node. Consequently, the ADD operations will have to synchronize with the REMOVE operations which now make structural changes in the BST.

In a sequential set-up, removing a node *k* from an external BST is a one step process of modifying the link $g(k){\leadsto}p(k)$ to connect $s(k)$ to $g(k)$. This process also cleans out the removed node. It essentially removes the node $p(k)$ from the (unordered) linked-list described by the nodes on the path from the root to $s(k)$. Thus, to perform REMOVE(k) with cleaning out *k* in a lock-free BST can be visualized as a two stage process - (a) single CAS to logically remove *k* by replacing it with a Dead node as in Algorithm 4.1 and (b) two CAS steps to remove $p(k)$ - adding a splice node between $p(k)$ and $s(k)$ to logically remove $p(k)$ and then swinging the pointer $g(k){\leadsto}p(k)$ to connect $s(k)$ to $g(k)$, as in Algorithm 4.2. Let us call these stages LREMOVE and PREMOVE, respectively. This understanding gives us the fundamental idea of Algorithm 4.3.

LREMOVE is quite straightforward. Now, to perform PREMOVE efficiently, along the lines of Algorithm 4.2, we carry two trailing node-pointers during the traversal for a modify operation. Thus, the method Search in Algorithm 4.3, line 4 to 7, becomes a blend of the same method in the previous two algorithms. At the termination of Search, the variable *gPar* points to the parent of the root of the sub-tree in which all the nodes are logically removed. To avoid special cases arising in placing the trailing node-pointers in an empty BST, we use a set of sentinel nodes as shown in line 1 to 3 and Figure 4.5.

We assign key $-\infty_3$ for a splice node, such that $\infty_3 > \infty_2 > \infty_1 > \infty_0 > |k|$ $\forall$ key k. It ensures that at a splice node a traversal always *goes right*. Hence,

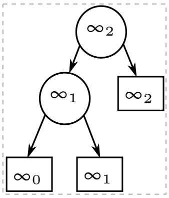


Figure 4.5: Sentinel Nodes: Lock-free BST

```
class Node {K k; Node* lt, rt, bck;};
----------------------------------------
root := Node($\infty_1$); grRoot := Node($\infty_0$);
root.lt := Node($\infty_2$).ref; root.rt := Node($\infty_1$).ref;
grRoot.lt := root.ref; grRoot.rt := Node($\infty_0$).ref;
----------------------------------------
Search(Node* gPar, Node* nex, Node* par, Node* leaf, K k)
    while leaf.lt $\neq$ null do
        if IsSp(leaf) then par := leaf.rt;
        else gPar := par; nex := leaf; par := leaf;
        leaf := Child(par, Dir(par, k));
----------------------------------------
GetNxt(Node* leaf)
    return IsSp(leaf) ? leaf.rt : leaf;
----------------------------------------
GetKey(Node* leaf) {return GetNxt(leaf).k;}
----------------------------------------
GetDeadBl(Node* gPar, K k)
    n := GetDead(k); n.bck := gPar; return n;
IsBl(Node* leaf) {return leaf.bck $\neq$ null;}
----------------------------------------
GetSp(Node* gPar, Node* leaf)
    if IsDead(leaf) then
        return GetDeadBl(gPar, leaf);
    else return Node($-\infty_3$, leaf.lt, leaf, gPar);
----------------------------------------
AddSp(Node* par, Node* gPar, dir sD)
    while true do
        sib := Child(par, sD);
        if IsBl(sib) then return sib;
        else if ChCAS(par, sib, GetSp(gPar, sib), sD) then return sib;
----------------------------------------
IsSp(Node* leaf) {return leaf.k == $-\infty_3$;}
BckTrck(Node* gPar, Node* nex, K k)
    nex := Child(gPar, k);
    while IsSp(nex) do
        gPar := nex.bck; nex := Child(gPar, k);
```

Algorithm 4.3. Help-optimal lock-free binary search tree

```
   ADD(K k)
24 g := grRoot.ref; n := root.ref; p := root.ref;
25 ℓ := root.lt; nd := Node(k).ref;
26 while true do
27     Search(g.ref, n.ref, p.ref, ℓ.ref, k);
28     cD := Dir(p, k); pD := Dir(g, k);
29     if !IsDead(ℓ) then
30         if GetKey(ℓ) == k then return false;
31         nl := NewNod(nd, GetNxt(ℓ), $\frac{k+\text{GetKey}(\ell)}{2}$).ref;
32         if IsSp(ℓ) then
33             if ChCAS(g, n, nl, pD) then return true;
34         else if ChCAS(p, ℓ, nl, cD) then return true;
35     else
36         if IsBl(ℓ) then
37             sib := AddSp(p, g, !cD);
38             if !IsDead(sib) then
39                 nl := NewNod(nd, GetNxt(sib), $\frac{k+\text{p.k}}{2}$).ref;
40                 if ChCAS(g, n, nl, pD) then return true;
41             else if ChCAS(g, n, nd, pD) then return true;
42         else if ChCAS(p, ℓ, nd, cD) then return true;
43     BckTrck(g.ref, n.ref, k); p := g; ℓ := n;
```

Algorithm 4.3. Help-optimal lock-free binary search tree: ADD

we connect $s(k)$ to rt of a splice node. We copy the lt field of $s(k)$ to the splice node that it connects to, which if null, indicates that $s(k)$ is a leaf node. Thus, a traversal may terminate at a splice node. Considering that, we always use the method GetNxt to access the actual leaf node, see line 8; and following that the method GetKey gives the key at that leaf node, see line 9. Further, to achieve local restart as in Algorithm 4.2, we include a bck pointer in the node structure to implement splice nodes that can provide reference to a node in a local clean zone. However, the local restart here is more complex, as discussed below. Consider these cases:

(A) An ADD operation o just before performing its CAS step gets pre-empted by the operating system scheduler. Let g be the trailing node-pointer pointing to the last internal node of the traversal path which is not logically removed. Suppose that, by the time o wakes up, the BST changes in a way that both the children of the node pointed by g are replaced by Dead nodes and the node itself cleaned out of the BST. Consequently, o will have *no* link to reach a clean zone except restarting from the root of the BST, which we want to avoid. To tackle this issue, we use the bck pointer of a Dead node, which replaces a node k in a REMOVE operation, to store g. We call a Dead node with a non-null bck field a DeadBl node.

(B) Two concurrent REMOVE operations o_1 and o_2, at the end of their

```
44 REMOVE(K $k$)
45   g := grRoot.ref; n := root.ref; p := root.ref; $\ell$ := root.lt;
46   dNdBl := null; sib := null; mode := INIT;
47   while true do
48     Search(g.ref, n.ref, p.ref, $\ell$.ref, $k$);
49     cD := Dir(p, $k$); pD := Dir(g, $k$);
50     if mode == INIT then
51       if GetKey($\ell$) $\neq k$ or IsDead($\ell$) then return false ;
52       dNd := GetDead($k$);
53       if !IsSp($\ell$) and p $\neq$ g then
54         dNdBl := GetDeadBl(g, $k$);
55         if ChCAS(p, $\ell$, dNdBl, cD) then
56           sib := AddSp(p, g, !cD); mode := CLEAN;
57           if IsSp(sib) then return true;
58           else if IsDead(sib) then {ChCAS(g, n, dNd, pD); return true;};
59           else if ChCAS(g, n, sib, pD) then return true;
60       else if ChCAS(g, n, dNd, pD) then return true;
61     else
62       if $\ell$ == dNdBl and p $\neq$ g then
63         if ChCAS(g, n, sib, pD) then return true;
64       else return true;
65     BckTrck(g.ref, n.ref, $k$); p := g; $\ell$ := n;
```

Algorithm 4.3. Help-optimal lock-free binary search tree: REMOVE

traversal, target to remove two leaf nodes k_1 and k_2, which are children of the same internal node, say p. Also suppose that o_1 and o_2 have same pair of trailing node-pointers - g and n - in their thread-local memory and thus for both o_1 and o_2 there is access to no link to backtrack above g in the BST. Suppose that LREMOVE stage of both o_1 and o_2 finished without contention, and thus after that both the children of p are DeadBl. Therefore, after its PREMOVE, if o_1 successfully connects the DeadBl node k_2 to g, o_2 will not get a node-pointer to reach a local clean zone to get a fresh g. It becomes untenable to restart o_2 in such a situation without accessing root, which we want to avoid (it may well be with o_1 symmetrically). To tackle this issue, we let o_2 fall back to the approach of Algorithm 4.1 and instead of cleaning the DeadBl node out it adds a Dead node containing key k_2 at g and gets out of the system to ensure progress. Therefore, similar to Algorithm 4.1, we make an ADD operation replace a Dead node with a new leaf node, knowing that no REMOVE operation takes step to clean out such a node.

With basics in place, we are ready to describe the pseudo-code of REMOVE and ADD operations of Algorithm 4.3; a CONTAINS operation works absolutely same as that in Algorithm 4.1.

The steps of a REMOVE(k) operation, line 45 to 65, are shown in Figure 4.6. Let $n(k)$ be the last logically removed internal node in the traversal path obtained

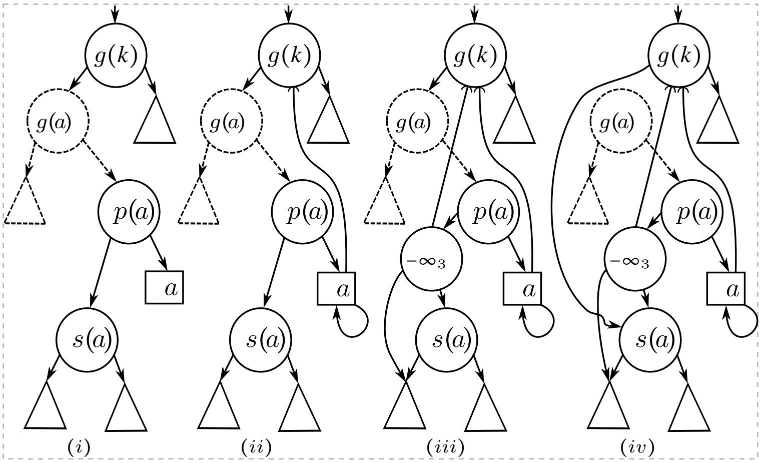


Figure 4.6: Steps of REMOVE in the BST.

by a call of `Search` at line 49, and $g(k)$ be its parent as shown in Figure 4.6 (i). $n(k)$ coincides with $p(k)$ in case there is no chunk of logically removed nodes above $p(k)$ in the traversal path. Replacing the target node *k* with a `DeadBl` node containing same key, Figure 4.6 (ii), logically removes *k*, line 55. After that, we add a splice node between $p(k)$ and $s(k)$ to logically remove $p(k)$ as shown in Figure 4.6 (iii). Finally, update the link $g(k)\rightsquigarrow n(k)$ to connect $s(k)$ to $g(k)$ as shown in Figure 4.6 (iv). If all these `CAS` executions are successful, we complete the REMOVE operation with cleaning out the `DeadBl` node.

In the stage LREMOVE itself, if the leaf-node at which traversal terminates, say ℓ, does not contain *k* or is found `Dead` (note that a `DeadBl` node is also `Dead`), REMOVE(k) returns false, line 51. If ℓ is a splice node, it shows that the actual node to remove is pointed by `GetNxt`(ℓ) which is ℓ.rt. To make a REMOVE operation *selfish*, we do not perform any `CAS` to help the pending REMOVE. However, we do not have a possibility for a local restart to get a fresh $g(k)$ after the completion of LREMOVE, similar to the case (B) above. Therefore, we replace the link $g(k)\rightsquigarrow n(k)$ by a `Dead` node containing *k*, and return true if the `CAS` succeeds, line 60.

In the stage PREMOVE, the first step is to add a splice node between $p(k)$ and $s(k)$, line 56. We use the method `AddSp`, line 15 to 18, to do that. In `AddSp`, if $s(k)$, denoted by sib, is found splice or `DeadBl`, indicated by non-null bck link, we return it as it is, line 17, because both indicate that the child-link of *par*, referred by sib, is never updated ever after. If sib is `Dead`, we perform a `CAS`, line 18, to replace it with a `DeadBl` node connecting its bck field to *gPar*, created at line 13, so that a concurrent ADD does not replace it directly.

If the method `AddSp` returns a splice node at line 56, it indicates that a concurrent ADD operation is working selfishly to progress (we describe it later) and we can safely allow the remaining steps of REMOVE(k) to be assimilated in the steps of ADD. Considering that, REMOVE(k) returns true, line 57. We call this behaviour of REMOVE(k) its *help-awareness* which is a main component of a *help-optimal implementation*.

On the other hand, if `AddSp` returns a `Dead` or `DeadBl` node, it indicates a scenario of case (B) and we handle it accordingly, see line 58. Finally, if a regular leaf node is returned as sib, we attempt a `CAS` to connect it to $g(k)$ to return true at line 59. If this `CAS` fails, it indicates that $g(k) \rightsquigarrow n(k)$ has changed and therefore we perform a `BckTrck` at line 65 similar to Algorithm 4.2 and reattempt the `CAS` if required, see line 63. Along the lines of Algorithm 4.2, the steps taken to add splice node between $p(k)$ and $s(k)$ are identified by the value of a variable mode set as `INIT` and after that mode is changed to `CLEAN`, line 56.

To add a new node in an external BST, we add a new sub-tree. We use the following *midpoint rule* to determine the key at the root of the new sub-tree.

Theorem 4.1 (**Midpoint rule**). *Let k be a query key and A be the (partially ordered) set of keys stored in a sub-tree. Let* $a_l \leq a\ \forall\ a \in A$ *and* $a_u \geq a\ \forall\ a \in A$*. To add a new node at the root of the sub-tree, assign a key* k_p *at the root of the new sub-tree such that* $k_p = \frac{k+a_u}{2}$ if $k > a_u$ *and* $k_p = \frac{k+a_l}{2}$ if $k < a_l$*.*

The mid-point rule maintains the symmetric order of the BST. Intuitively, rule 4.1 optimizes the average hight of the BST. We do not delve into an analytical discussion of this rule in the present work. In experiments, we observed that this approach improves the average throughput.

An ADD(k), line 25 to 43, performs a traversal using `Search` to reach a leaf node ℓ. If ℓ is neither `Dead` nor `DeadBl`, we find the regular leaf node using `GetNxt`(ℓ). It calls the `NewNod` method to create a new internal node pointed by nl. We apply rule 4.1 at line 31. If ℓ is a regular node, it perform as in Algorithm 4.1. However, if ℓ is a splice node, it does not take steps to help the pending REMOVE operation and behaves in a selfish manner to directly update $g(k) \rightsquigarrow n(k)$ to nl using a `CAS`. If `CAS` succeeds, it not only ensures success of ADD(k), but also guarantees the completion of some pending REMOVE operations. If `CAS` to connect nl at line 33 or 34 succeeds, we return true.

On the other hand, if ℓ is found `Dead`, ADD(k) behaves along the lines of Algorithm 4.1, see line 42. And finally, if ℓ is `DeadBl`, to ensure progress, we first fix the sibling of ℓ using the method `AddSp`, line 37, and then add either a new node, line 41, or a new internal node, line 40, at $g(k)$ in a selfish fashion. The call of `AddSp` at line 37 may assimilate the steps of a concurrent pending REMOVE operation, which being help-aware, terminates immediately, as discussed before. Note that, to apply rule 4.1 here, we use p.k instead of GetKey(sib) because the latter may not provide the required bound of the set of keys stored in the sub-tree rooted at sib. On a CAS failure, we perform a BckTrck at line 43 to get a fresh set of thread-local variables and reattempt.

4.4.2 Correctness and Lock-freedom

Proving that the modify operations maintain a valid external BST requires similar approach as that in Algorithm 4.2. Therefore, without repeating them, we mention that we derive an induction based proof building on the arguments that the sentinel nodes form a valid BST at the initialization and no modify operation invalidates the symmetric order of the BST.

In this algorithm, the linearization points of a successful ADD, REMOVE and CONTAINS operations and an unsuccessful ADD operation are similar to their counterparts in Algorithm 4.2. A CONTAINS or a REMOVE operation returns false also in case the node containing query key is found Dead, in addition to the cases already discussed in Algorithm 4.2. The linearization point of such a CONTAINS or REMOVE operation is taken at own invocation point.

Finally, we can prove the lock-freedom of Algorithm 4.3 using arguments which are parallel to those used in Algorithm 4.2. Very evidently, a CONTAINS is wait-free for a finite key universe.

4.5 Help-optimality: Specification

We consider an *asynchronous shared memory system* U comprising of a finite set of *threads* P and a finite set of *shared variables* V. At time t, the *states* of P and V are denoted by P_t and V_t, respectively. Let Υ be a lock-free data structure formed by variables $v{\in}V$. Let $\mathcal{O}_p$ be the set of *operations* performed by a $p{\in}P$ on Υ. A *step* s of an operation $o{\in}\mathcal{O}_p$ comprises local computations in p and at most a single execution of an atomic-primitive $a{\in}\{\texttt{read}, \texttt{write}, \texttt{CAS}\}$ on a shared variable $v{\in}V$. A state Υ_t of Υ is formed by variables $v{\in}V_t$. On execution of a step s, Υ can change from a state Υ_t to another state $\Upsilon_{t'}$. We denote such a state change by $\Delta\Upsilon_{t,t'}$. Let S_o denote the set of steps to complete an operation $o{\in}\mathcal{O}_p$.

We call $s{\in}S_o$ an *altruistic step* of o, if (a) it is executed to apply a state change $\Delta\Upsilon_{t,t'}$ (b) $\Delta\Upsilon_{t,t'}$ is necessary for completion of a concurrent operation $o'{\in}\mathcal{O}_{p'}$ and (c) $\Delta\Upsilon_{t,t'}$ is *not* necessary for completion of o. We call an operation o *selfish* if *no* step $s{\in}S_o$ is altruistic.

We call $s \in S_o$ a *wasted step* of o, if (a) it is executed to apply a state change $\Delta\Upsilon_{t,t'}$, (b) $\Delta\Upsilon_{t,t'}$ is necessary for completion of o (c) $\Delta\Upsilon_{t,t'} \subseteq \Delta\Upsilon_{t,t''}$ and (d) $\Delta\Upsilon_{t,t''}$ has already been applied by a set of steps $\{s'_1, \ldots, s'_n\}$, where $s'_i \in S_{o'_i}$ is a step of a concurrent operation $o'_i \in \mathcal{O}_{p'_i}$. We call $o \in \mathcal{O}_p$ *help-aware* if it performs *no more than one* wasted step.

A lock-free data structure Υ is called *help-optimal* if every operation $o \in \mathcal{O}_p$ for each $p \in P$ is both selfish and help-aware. In algorithms 4.1 to 4.3, we can observe that every operation satisfies the requirements of both selfishness and help-awareness. We skip a rigorous definitional discussion on selfishness, help-awareness, and help-optimality to a future work.

Censor-Hillel *et al.* [22] defined *help-freedom*, which intuitively implies that an operation does not *altruistically* help a concurrent (slow) operation to guarantee wait-freedom. In contrast to that, in lock-free algorithms, help-optimality not only implies an absence of altruistic helping but also indicates that an operation is aware of intended modification getting applied as a part of a modification by a concurrent operation. Thereby, on account of helping, the aggregate number of steps is minimized. In this work, we do not delve into a formal comparison between help-optimality and help-freedom.

As a rationale behind the term help-optimality, we would like to underline our aim to optimize a lock-free design with respect to the number of (CAS execution) steps incurred in helping under the constraints such as an optimal memory footprint and an optimal amortized step complexity.

4.6 Experimental Evaluation

4.6.1 Overview

In this section, we present a detailed performance analysis of our implementations in both C/C++ and Java. The following concurrent linked-lists and lock-free BSTs are compared:

1. **HO-LL:** An optimized variant of lock-free linked-list of [1], where a CONTAINS does not perform help.

2. **Lazy-LL:** Lock-based linked-list of [19], in which logically removed nodes are ignored during traversal.

3. **CWT-LL:** Lock-free linked-list described in Section 4.3.

4. **EFRB-BST:** Lock-free external BST of [4], in which both ADD and REMOVE require help to complete at a conflict.

5. **LF-SKIPLIST:** Concurrent skip-list implementation that is part of the java.util. concurrent package [17].

6. **NM-BST:** Lock-free external BST of [6], in which multiple nodes under REMOVE by different threads are cleaned out together similar to Algorithm 4.3.

7. **CWT-BST:** Lock-free BST described in Section 4.4.

8. **CWT-Simple-BST:** Lock-free BST of Section 4.2.

4.6.2 Experimental Set-up

We performed our evaluations on a dual-socket server with a 3.4 GHz Intel (R) Xeon (R) E5-2687W-v2 having 16 physical cores (32 hardware threads by hyperthreading), 16 GB of RAM, running Ubuntu 13.04 Linux (3.8.0-35-generic x86_64) with Java HotSpot (TM) 64-Bit Server VM (build 25.60-b23, mixed mode). We compiled all Java implementations with `javac` version `1.8.0_60` and used the runtime flags `-d64 -server`. C/C++ implementations were compiled with `g++` version `4.9.2`, `O3` optimization and TCMalloc [23] to reduce dynamic memory allocation overheads.

We compared the performance in terms of the throughput as Million operations per second (Mops/s). We measured the memory consumption as the change in heap-size of the JVM on loading the set with initial elements and on execution of the workload. Each experiment was run for 5 seconds, then the average over 6 trials was taken under the following parameters:

1. **Workload Distribution:** Similar to [6], we considered three workload distributions: (a) *write-dominated*: 0% CONTAINS, 50% ADD and 50% REMOVE. (b) *mixed*: 70% CONTAINS, 20% ADD and 10% REMOVE, and (c) *read-dominated*: 90% CONTAINS, 9% ADD and 1% REMOVE.

2. **Set Size:**. The maximum set size depends on the range of the keys. We consider the following key ranges for the linked-lists: 2^7, 2^9, 2^{10} and 2^{12}. For the BSTs we consider the ranges: 2^{10}, 2^{14}, 2^{17} and 2^{20}. In each experiment, the set is pre-loaded with roughly half the keys in the key range.

C/C++ and Java Implementations: As we pointed out in the Section 4.1, many concurrent data structures are designed with language specific dependencies and as such offer varying performance in different languages. Additionally, original implementations of the algorithms are available either in Java or C/C++. With this in mind, we implemented our new language-portable lock-free algorithms in both C/C++ and Java. The code is available at `https://github.com/bapi/ConcurrentSet`.

To ensure a fair comparison, we implemented our C/C++ versions of the algorithms as part of the ASCYLIB library [14], with SSMEM - a memory allocator with epoch-based garbage collection. We used the same benchmarks which are part thereof. HO-LL in Java employs RTTI. For locking, Lazy-LL uses ReentrantLock in Java and a ticket lock in C/C++.

4.6.3 Performance Results and Discussion

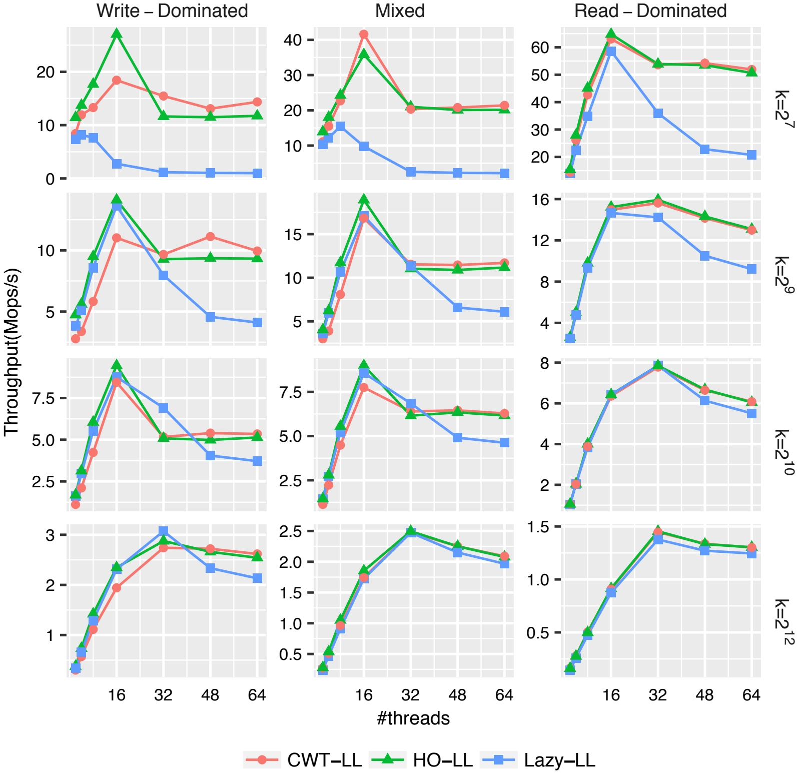


Figure 4.7: Concurrent linked-list algorithms: Java Implementation

Figures 4.7 and 4.8 depict the comparative performance of linked-list-based Set algorithms in Java and in C/C++, respectively. At low contention *i.e.*, with read-dominated workloads and large key space sizes, the lists scale with increasing thread count. CWT-LL performs on a par with HO-LL in both Java and C/C++. In the high contention cases, mainly write-dominated and small key space sizes, Lazy-LL degrades significantly with increasing thread count. This is mainly due to the increased contention on the locks and cache misses resulting from the lock migrations. Contention increases as the list gets shorter in size with a smaller key space size. At high contention CWT-LL outperforms HO-LL by 5% for Write-Dominated and 3%-6% for Mixed workloads. This can be attributed to the local restart and the ability to clean out multiple nodes in a single step.

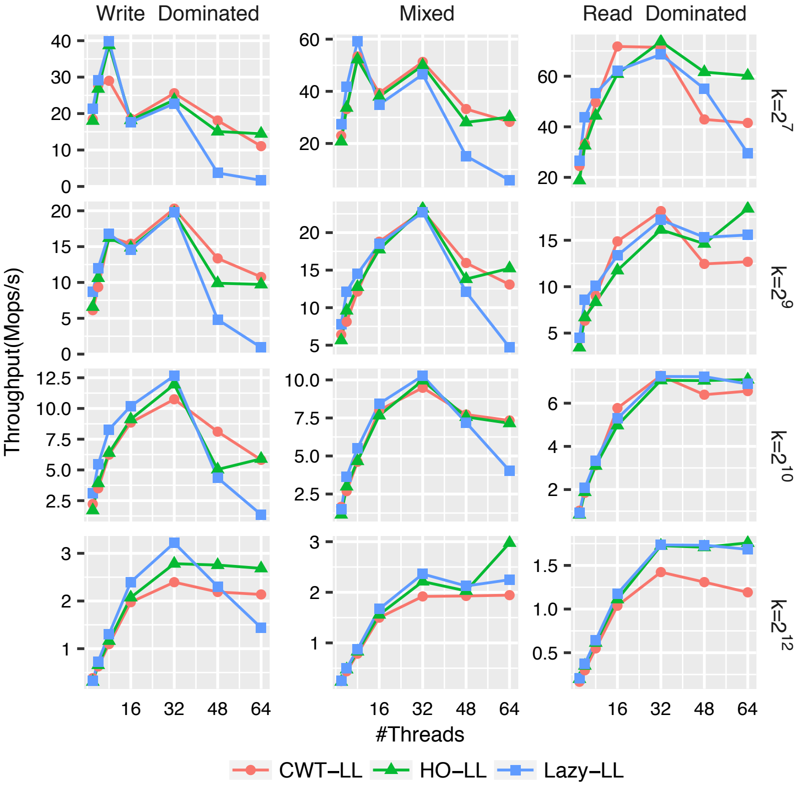


Figure 4.8: Concurrent linked-list algorithms: C/C++ Implementation

In C/C++ we observe similar relative performance, however, the list performance degrades significantly when the cores are saturated with threads (most especially in the write-dominated workload). The effect of oversubscribing the cores with more threads is bigger in Lazy-LL than that in other algorithms as a result of increased lock-contention.

Figures 4.9 and 4.10 shows the comparative performance of considered lock-free BST and skip-list algorithms in Java and C/C++, respectively. We have not included CWT-Simple-BST here considering its incomparable memory footprint. It is clear that among the Java implementations, CWT-BST offers the best throughput for all key space sizes and workloads. CWT-BST outperforms EFRB-BST by 10%- 50% and LF-SKIPLIST 20%-100% over Write-Dominated and Mixed workloads.

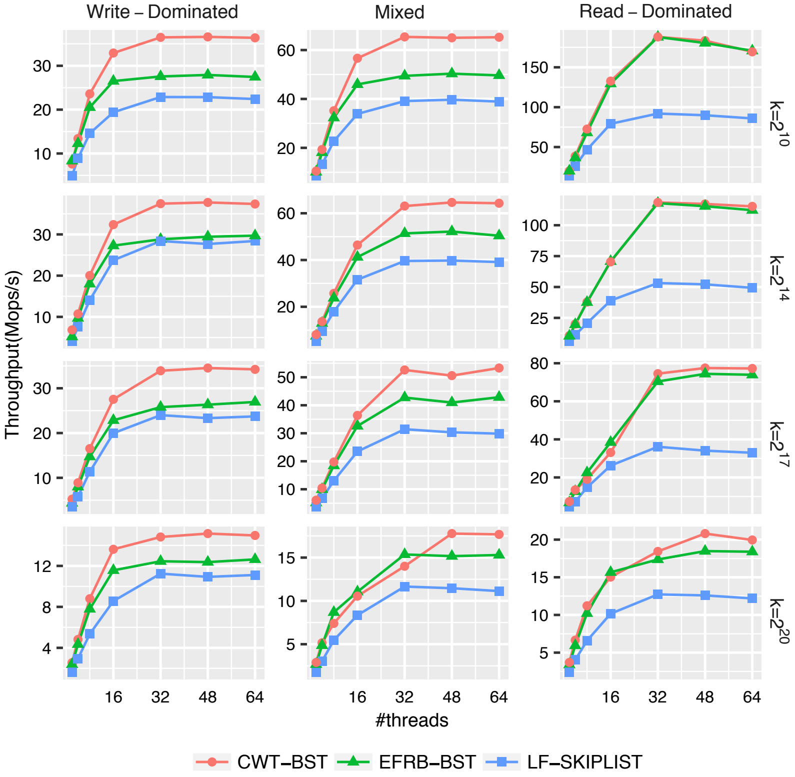


Figure 4.9: Lock-Free BST algorithms: Java Implementation

In C/C++, NM-BST outperforms others at high contention. This can be attributed to the advantage of bit-stealing over explicit object allocations. Bit masking, unmasking and other bitwise operations in C/C++ are simple and faster than object creation, however not portable to other high-level languages. As we increase the key space size, CWT-BST offers performance similar to NM-BST,

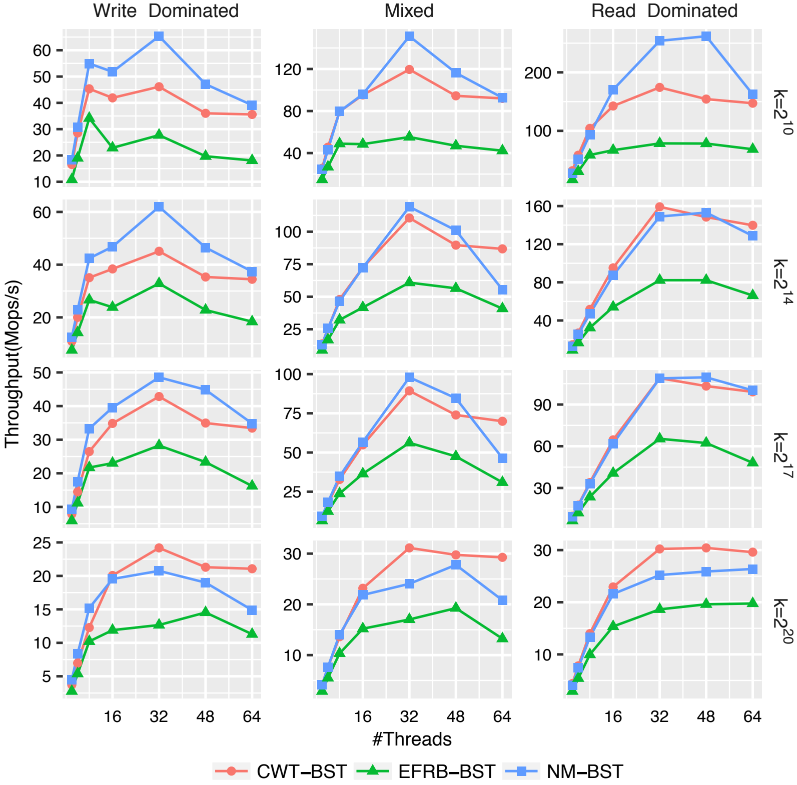


Figure 4.10: Lock-Free BST algorithms: C/C++ Implementation

especially in Mixed and Read-Dominated workloads, even dominating in the low contention case with key space (2^{20}) by 3%-15%. This can be attributed to a comparative cost of object allocation but lowered cost of reading a pointer without bit unmasking. It can be noted that although EFRB-BST implementation is based on bit-stealing, CWT-BST outperforms it in every case scenario by 10%-50%.

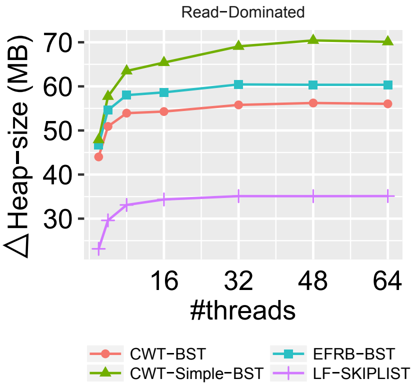


Figure 4.11: Heap size change

Memory Reclamation: As a REMOVE operation allocates a splice node, the load on garbage collector certainly increases. However, as illustrated by fig. 4.11, in a garbage collected environment, CWT-BST experiences no unexpected growth in heap-memory usage. In fact, on this account, it outperforms EFRB-BST. Though the figure presents a case for one workload setting, we observed similar relative memory usage with every workload settings. Nevertheless, we do advise that these techniques should not be used without memory reclamation.

4.7 Conclusion

In this chapter, we introduced the notion of help-optimality in a lock-free algorithm. Intuitively, in a lock-free data structure, which satisfies help-optimality, at a conflict over modification of a shared variable, we avoid both offer and acceptance of help in form of a step comprising a CAS execution. Help-optimality consists of the notions of selfishness and help-awareness. Selfishness implies optimization of the count of steps of CAS executions by an obstructed operation, whereas help-awareness implies the same for an obstructing operation.

The present work is mostly experimental in nature to demonstrate the utility of the concept of help-optimality in a lock-free linked-list and a BST. In future, we plan to develop formal specifications of the introduced notions.

Following a state-of-the-art implementation of the lock-free skip-list in Java library, in this chapter, we designed the lock-free data structures to provide a language-portable implementation. We experimentally showed that such an implementation performs on a par with highly optimized implementations in C++ which use the technique of bit-stealing. The Go programming language, which reasonably focuses on concurrency, provides pointers without pointer-arithmetic and does not provide type-inheritance. We believe that with growing popularity of such languages, designing language-portable lock-free data structures will be increasingly significant.

Bibliography

[1] Timothy L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the International Conference on Distributed Computing*. 2001, pp. 300–314, Springer.

[2] Maged M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2002, pp. 73–82, ACM.

[3] Mikhail Fomitchev and Eric Ruppert, "Lock-free linked lists and skip lists," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 2004, pp. 50–59, ACM.

[4] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel, "Non-blocking binary search trees," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 2010, pp. 131–140, ACM.

[5] Shane V. Howley and Jeremy Jones, "A non-blocking internal binary search tree," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2012, pp. 161–171, ACM.

[6] Aravind Natarajan and Neeraj Mittal, "Fast concurrent lock-free binary search trees," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2014, pp. 317–328, ACM.

[7] Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert, "The amortized complexity of non-blocking binary search trees," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 2014, pp. 332–340, ACM.

[8] Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas, "Efficient lock-free binary search trees," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 2014, pp. 322–331, ACM.

[9] Arunmoezhi Ramachandran and Neeraj Mittal, "A fast lock-free internal binary search tree," in *Proceedings of the International Conference on Distributed Computing and Networking*. 2015, pp. 37:1–37:10, ACM.

[10] Maurice Herlihy and Jeannette M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[11] Greg Barnes, "A method for implementing lock-free shared-data structures," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 1993, pp. 261–270, ACM.

[12] Daniel Cederman, Bapi Chatterjee, Nhan Nguyen, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas, "A study of the behavior of synchronization methods in commonly used languages and systems," in *Proceedings of the International Parallel and Distributed Processing Symposium*. 2013, pp. 1309–1320, IEEE.

[13] Vincent Gramoli, "More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2015, pp. 1–10, ACM.

[14] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 2015, pp. 631–644, ACM.

[15] Joel Gibson and Vincent Gramoli, "Why non-blocking operations should be selfish," in *Proceedings of the International Symposium on Distributed Computing*. 2015, pp. 200–214, Springer.

[16] "java.util.concurrent," `https://docs.oracle.com/javase/8/docs/api/`.

[17] Doug Lea, "`ConcurrentSkipListMap`," in *`java.util.concurrent`*.

[18] John D. Valois, "Lock-free linked lists using compare-and-swap," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 1995, pp. 214–222, ACM.

[19] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit, "A lazy concurrent list-based set algorithm," in *Proceedings of the International Conference on Principles of Distributed Systems*. 2006, pp. 3–16, Springer.

[20] Yehuda Afek, Gideon Stupp, and Dan Touitou, "Long lived adaptive splitter and applications," *Distributed Computing*, vol. 15, no. 2, pp. 67–86, 2002.

[21] Hagit Attiya and Arie Fouren, "Algorithms adapting to point contention," *Journal of the ACM*, vol. 50, no. 4, pp. 444–468, 2003.

[22] Keren Censor-Hillel, Erez Petrank, and Shahar Timnat, "Help!," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 2015, pp. 241–250, ACM.

[23] Sanjay Ghemawat and Paul Menage, "Tcmalloc : Thread-caching malloc," `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`, 2009.

PAPER IV

Bapi Chatterjee, **Ivan Walulya** and Philippas Tsigas

Concurrent Linearizable Nearest Neighbour Search in LockFree-kD-tree

In the Proceedings of the
19^{th} *International Conference on Distributed Computing and Networking*
pp. 11:1–11:10, ACM 2018.

5

Concurrent Linearizable Nearest Neighbour Search in LockFree-kD-tree

Abstract
The Nearest neighbour search (NNS) is a fundamental problem in many application domains dealing with multidimensional data. In a concurrent setting, where dynamic modifications are allowed, a linearizable implementation of NNS is highly desirable.

This paper introduces the LockFree-kD-tree (LFkD-tree): a lock-free concurrent kD-tree, which implements an abstract data type (ADT) that provides the operations ADD, REMOVE, CONTAINS, and NNSEARCH. Our implementation is linearizable. The operations in the LFkD-tree use single-word read and compare-and-swap (CAS) atomic primitives, which are readily supported on available multi-core processors.

We experimentally evaluate the LFkD-tree using several benchmarks comprising real-world and synthetic datasets. The experiments show that the presented design is scalable and achieves significant speed-up compared to the implementations of an existing sequential kD-tree and a recently proposed multidimensional indexing structure, PATRICIA-hypercube-tree or PH-tree.

5.1 Introduction

5.1.1 Background

Given a dataset of multidimensional points, finding the point in the dataset at the smallest distance from a given *target point* is typically known as the nearest neighbour search (NNS) problem. This fundamental problem arises in numerous application domains such as data mining, information retrieval, machine learning, robotics, *etc.*

A variety of data structures available in the literature, which store multidimensional points, solve the NNS in a sequential setting. Samet's book [1] provides an excellent collection of data structures for storing multidimensional data. Several of these have been adapted to perform parallel NNS over a static data structure. However, both sequential and parallel designs primarily consider NNS queries without accommodating dynamic addition or removal (modifications) operations on the data structure. Allowing concurrent dynamic modifications exacerbates the challenge substantially.

The wide availability of multi-core machines, large system memory, and a surge in the popularity of in-memory databases, have led to a significant interest in the index structures that can support NNS with dynamic concurrent addition and removal of data. However, to our knowledge no complete work exists in the literature on concurrent data structures that support NNS.

Typically, a hierarchical tree-based multidimensional data structure stores the points following a space partitioning scheme. Such data structures provide an excellent tool to *prune* the subsets of a dataset that do not contain the target nearest neighbour. Thus, an NNS query *iteratively scans* the dataset using such a data structure. The iterative scan procedure starts with an initial guess, at every iteration visits a subset of the data structure (*e.g.,* a subtree of a tree) that can potentially contain a better guess, and is unvisited until the last iteration, updates the current guess if required, and thereby finally returns the nearest neighbour.

In a concurrent setting, performing an iterative scan along with concurrent modifications, faces an inescapable challenge. Consider the case of an operation op performing an NNS query in a hierarchical multidimensional data structure that stores points from $\mathbb{R}^d$ and where Euclidean distance is used. Let $a=\{a_i\}_{i=1}^{d} \in \mathbb{R}^d$ be the target point of the NNS. Let us assume that $k^*=\{k_i^*\}_{i=1}^{d} \in \{k : k \text{ is key of a node}\}$ is the nearest neighbour of a at the invocation of $nns(a)$. In a sequential setting, where no addition or removal of data-points occurs during the lifetime of $nns(a)$, k^* remains the nearest neighbour of a at the return of $nns(a)$. However, if a concurrent addition is allowed, a new node with key k^{**} may be added to the data structure in a sub-structure that may already have been visited or pruned by the completion of the latest iteration step. Clearly, $nns(a)$ would not revisit that sub-structure. Suppose that k^{**} was closer to a compared to k^{*}, if $nns(a)$ returns k^{*}, it is not consistent to an operation which observes that the addition of k^{**} completes before $nns(a)$.

Considering concurrent operations on data structure, *linearizability* [2] is the most popular framework for *consistency*. A concurrent data structure is linearizable if every execution has a *linearization points*, between the invocation and response of each operation, where it seems to take effect instantaneously. In a concurrent setting, we desire linearizability of an NNS query.

Non-blocking progress guarantees are preferred for concurrent operations. In an asynchronous shared-memory system, where an infinite delay or crash failure of a thread is possible, a lock-based concurrent data structure is vulnerable to pitfalls such as deadlock, priority inversion and convoying. On the other hand, in a *non-blocking* data structure, threads do not hold locks, and at least one non-faulty thread is guaranteed to finish its operation in a finite number of steps(lock-freedom). Wait-freedom is a stronger progress condition that all threads will complete an operation in a finite number of steps.

In recent years, a number of practical lock-free search data structures have been designed: skip-lists [3], binary search trees (BSTs) [4–7], *etc.* Despite the growing literature on lock-free data structures, the research community has largely focused on one-dimensional search problems. To our knowledge, no complete design of any lock-free multidimensional data structure exists in the literature.The challenge appears in two ways: designing a concurrent lock-free multidimensional data structure that supports NNS and ensuring the linearizability of NNS.

One of the most commonly used multidimensional data structures for NNS is the kD-tree, introduced by Bentley [8]. In principle, a kD-tree is a generalization of the BST to store multidimensional data. Friedmann *et al.* [9] proved that a kD-tree can process an NNS in expected logarithmic time assuming uniformly distributed data points. Various efforts, including approximate solutions, have contributed to improving the performance of NNS in kD-trees [10, 11]. Furthermore, several parallel kD-tree implementations have been presented, specifically in the computer graphics community, where the focus is on accelerating the applications, such as the ray tracing, in single-instruction-multiple-data (SIMD) programming model [12].

Unfortunately, these designs do not fit concurrent setting where we desire linearizable NNS with concurrent modifications. For robotic motion planning, Ichnowski *et al.* [13] used a kD-tree of 3-dimensional data in which they add nodes concurrently. However, this design does not support REMOVE and the canonical implementation of NNSEARCH, using recursive tree-traversal, is not linearizable.

The contributions of this work are the following:

1. We describe a linearizable implementation of an abstract data type (ADT) that provides ADD, REMOVE, CONTAINS and NNSEARCH operations for a multidimensional dataset.
2. To illustrate the implementation, we present LockFree-kD-tree (LFkD-tree) - an efficient concurrent lock-free kD-tree. LFkD-tree requires atomic single-word read and compare-and-swap primitives.
3. For experimental validation of the LFkD-tree, we use a 2-dimensional real-world dataset and several synthetic datasets representing extreme cases. We evaluate our implementation against an existing sequential kD-tree implementation and a recently proposed multidimensional index structure - PATRICIA-hypercube-tree implementation [14].

The rest of this paper is organized as follows; first, we present the basic design of the LockFree-kD-tree (LFkD-tree) (Section 5.2). Thereafter, we detail the lock-free implementation (Section 5.3). On describing the algorithm, we present the proof of its correctness (Section 5.4). We describe an interesting real-life application of this work (Section 5.5). Finally, we describe experimental evaluation of our algorithm against an existing sequential kD-tree and the PATRICIA-hypercube-tree [14]* (Section 5.6). For experimental evaluation of the implementations, we use a 2-dimensional real-world dataset and several synthetic datasets representing extreme cases.

5.1.2 A high-level summary of the work

The main challenge in implementing a linearizable NNSEARCH is to ensure that it is not oblivious to the concurrent modifications in the data structure. NNSEARCH requires an iterative scan, which collects, along with pruning, an atomic *snapshot*.

In general, concurrent data structures do not trivially support atomic snapshots. Some exceptions are - the lock-based BST by Bronson *et al.* [15], the lock-free Trie by Prokopec *et al.* [16] and lock-free k-ary search tree by Brown *et al.* [17]. Petrank *et al.* presented a method to support atomic snapshots in one dimensional lock-free ordered data structures that implement sets [18]. They illustrated their method in lock-free linked-lists and skip-lists.

The main idea in [18, 19] is augmenting the data structure with a pointer to a special object, which provides a platform for an ADD/ REMOVE/ CONTAINS

*In this work, we are not interested in an existing parallel or sequential implementation that does not provide a REMOVE operation, in which case lock-free design poses little challenge. We could find only these two existing implementations that provide REMOVE along with NNSEARCH.

operation to *report* modifications to a concurrent operation performing a full or partial snapshot. Nevertheless, collecting an atomic snapshot of a multidimensional data structure to perform an NNSEARCH would be naive. We need to adapt the procedure of iterative scan, which benefits from an efficient hierarchical space partitioning structure, to a concurrent setting.

Our work proposes a solution based on augmenting a concurrent data structure with a pointer to a special object called *neighbour-collector*. A neighbour-collector provides a platform for *reporting* concurrent modifications that can otherwise *invalidate* the output of a linearizable NNSEARCH.

Essentially, an operation NNSEARCH(α) first searches for an exact match of α in the data structure, and if it succeeds, returns α itself as its nearest neighbour. If an exact match is not found, before starting the iterative scan, NNSEARCH(α) *announces* itself. The announcement uses a new *active* neighbour-collector that contains the target point α and the current best guess for the nearest neighbour of α. On completing the iterative scan, it *deactivates* the neighbour-collector. A concurrent operation, after completing its steps, checks for any active neighbour-collector, and if found, reports its output if it is a better guess than the current best guess available. Finally, NNSEARCH(α) outputs the best guess among the collected and the reported neighbours as the nearest neighbour of α.

Naturally, there can be multiple concurrent NNSEARCH operations with different target points, and we must allow each of them to continue its iterative scan, after announcing it as soon as it begins. To handle multiple concurrent announcements, we use a lock-free linked-list of neighbour-collector objects. The data structure stores a pointer to one end of this list, say the *head*. A new neighbour-collector is allowed to be added only at the other end, say the *tail*.

Consequently, before announcing a new iterative scan, an NNSEARCH operation goes through the list and checks whether there is an active neighbour-collector with the same target point. If an active neighbour-collector is found, it is used for a concurrent *coordinated* iterative scan(explained in the next paragraph). A neighbour-collector is removed from the lock-free linked-list as soon as the associated iterative scan is completed. Hence, at any point in time, the length of the list is at most the number of active NNSEARCH operations.

During an iterative scan, a subset of the dataset is pruned depending on whether the distance of the target point from a *bounding box* covering the subset is greater than that from the current best guess. Now, if the current best guess at a neighbour-collector is the outcome of already pruned many subsets, an NNSEARCH that starts its iterative scan at a later time-point, or is slow (or even delayed), will be able to complete much faster. Thus, the coordination among the concurrent NNS, via their iterative scans at the same neighbour-collector, speeds them up in aggregation.

The basic design of the LFkD-tree is based on the lock-free BST of Natarajan *et al.* [6]. To perform an iterative scan, we implement an efficient fully *non-recursive traversal* using *parent* links, which is not available in [6]. Thus, to manage an extra link in each node, our design requires extra effort for the lock-free synchronization. The modify operations use single-word-sized atomic CAS primitives. The *helping mechanism* is based on the *operation descriptors* at the child-links. Consequently, extra object allocations for synchronization is avoided. The linearizable implementation of NNSEARCH is not confined to the LFkD-tree, and it can be used in a similar concurrent implementation of any other multidimensional data structure available in [1].

5.2 LockFree-kD-tree: Basic Design

5.2.1 Design of the LFkD-tree

The LFkD-tree is a *point kD-tree* in which each node, that stores data, is assigned at most one data-point. Typically, to partition $\mathbb{R}^d$, we use *axis-orthogonal hyperplanes* that are given by $x_i{=}c$, $1{\le}i{\le}d$. The structure and consequently the NNS performance of a kD-tree heavily depends on the *splitting rule* - the procedure to select the partitioning hyperplanes. Traditionally, in a sequential setting, to construct a kD-tree from static data, the partitioning hyperplanes are chosen to coincide with points that belong to the given dataset. In this approach, similar to an internal BST representation [7], each node is used for storing data. However, removing a node from an internal BST is costly, more so in a concurrent setting [5, 7]. With this in mind, we opt for an external BST representation [4, 6] to design the LFkD-tree. In this design, only *leaf-nodes* contain the data-points and *internal-nodes* route a traversal, see fig. 5.1 (b). More importantly, it gives us the flexibility to compute c and $i : 1{\le}i{\le}d$ for a hyperplane $x_i{=}c$, which may not coincide with a data-point.

To compute the values of c and i, in the scenarios where the entire dataset is available beforehand, a number of splitting rules exist in the literature [9, 10]. These rules focus on the hierarchical partition of a *closed hyperrectangle* that covers the entire dataset and not only tries to balance a kD-tree but also optimize its depth. In a concurrent setting, where we do not have knowledge of the entire dataset in advance, the partitioning hyperplane needs to be computed dynamically and in a very localized fashion. For the LFkD-tree, we formulate a simple and practical splitting rule, the ***local-midpoint rule***, as given in Section 5.2.2. In this work, we do not delve in to an analytical discussion of the splitting rule.

A leaf-node of a LFkD-tree Υ, contains a unique data-point as its *key*, whereas, an internal-node corresponds to a partitioning hyperplane. Without

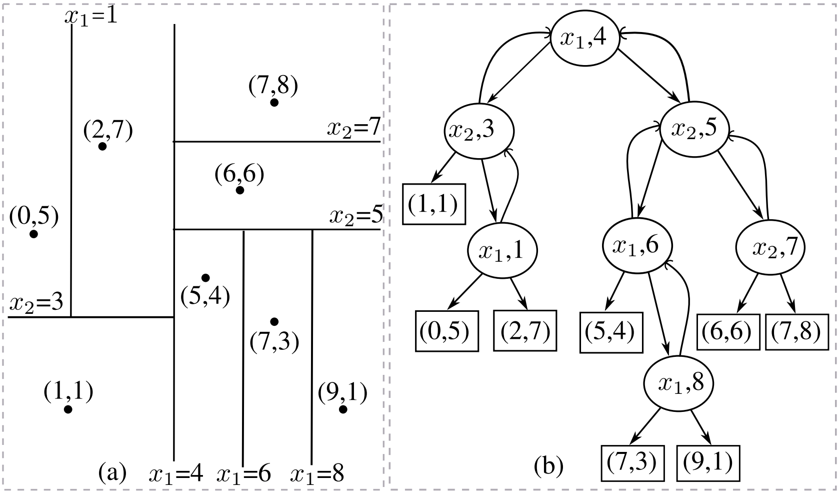


Figure 5.1: LFkD-tree Structure

ambiguity, we denote a leaf-node containing key k=$\{k_i\}_{i=1}^{d}$∈$\mathbb{R}^d$ by $\texttt{Nd}(k)$ (or $\texttt{Nd}(\{k_i\}_{i=1}^{d})$), and an internal node associated with a hyperplane x_i=c, by $\texttt{Nd}(i, c)$. An internal-node has three *links* connected to its *left-child*, *right-child* and *parent*. We indicate the link emanating from a node N and incoming to a node M by N⇝M. Access to Υ is given by the *address* of (pointer to) a unique node *root*. A node N is said to be *present* in Υ, denoted by N∈Υ_t, if it can be *reached* following the links starting from the root. For each internal-node $\texttt{Nd}(i, c)$, Υ maintains the following invariants: (i) a node $\texttt{Nd}(\{k_i\}_{i=1}^{d})$ belongs to the *left subtree*, if k_i<c, (ii) a node $\texttt{Nd}(\{k_i\}_{i=1}^{d})$ belongs to the *right subtree*, if k_i≥c and (iii) both subtrees are themselves LFkD-tree. (i) and (ii) together are called the *symmetric order* of the LFkD-tree. Figure 5.1 illustrates the structure of a subtree of a LFkD-tree corresponding to a sample 2-dimensional dataset.

5.2.2 Sequential Behaviour of the ADT Operations

LFkD-tree implements an abstract data type that provides operations ADD, REMOVE, CONTAINS and NNSEARCH. For each of the operations, we start with a *query*: start from the root, traverse down Υ, at each internal node decide left / right child direction using the symmetric order until arrive at a leaf-node.

To perform ADD(a), a∈$\mathbb{R}^d$, if the query terminates at a leaf-node $\texttt{Nd}(b)$, b∈$\mathbb{R}^d$, and b = a (an element-wise comparison of keys), ADD(a) returns false. However, if b ≠ a, we allocate a new internal-node $\texttt{Nd}(i, c)$ with its child links connected to two leaf-nodes $\texttt{Nd}(a)$ and $\texttt{Nd}(b)$. If $p(\texttt{Nd}(b))$ was the parent of $\texttt{Nd}(b)$ at the termination of query, we connect the parent link of $\texttt{Nd}(i, c)$ to $p(\texttt{Nd}(b))$. We update the link $p(\texttt{Nd(b)})\rightsquigarrow\texttt{Nd(b)}$ to point to $\texttt{Nd}(i, c)$ and return true. To compute i and c, we employ the local-midpoint rule as given below.

Local-midpoint rule: *$1{\le}i{\le}d$ is the index of the coordinate axis along which a and b have the maximum coordinate difference; if there are more than one such axis then select the one with the lowest index. Take the hyperplane as $x_i = \frac{a[i]+b[i]}{2}$.*

To perform $\textsc{Remove}(a)$, if the leaf-node where the query terminates, has the key a, *i.e.,* $\texttt{Nd}(a){\in}\Upsilon$, we modify the link from the *grandparent* of $\texttt{Nd}(a)$, denoted by $g(\texttt{Nd}(a))$, to its parent, to connect the *sibling* of $\texttt{Nd}(a)$, $s(\texttt{Nd}(a))$, to $g(\texttt{Nd}(a))$; and return true. If $\texttt{Nd}(a){\notin}\Upsilon$, $\textsc{Remove}(a)$ returns false. To perform $\textsc{Contains}(a)$, using a similar query we check whether $\texttt{Nd}(a){\in}\Upsilon$ and return true or false accordingly.

The operation $\textsc{NNSearch}(a)$ is non-trivial. On termination of the initial query, if we reach at $\texttt{Nd}(b)$ and $b = a$, clearly the nearest neighbour of a, available in the dataset stored in Υ, is a itself. However, if $b \neq a$, we take b as our *current best guess* and check whether the *other subtree* of $p(\texttt{Nd}(b))$ (the current subtree consists the single node $\texttt{Nd}(b)$) stores a *better guess*.

Suppose that $p(\texttt{Nd}(b)){=}\texttt{Nd}(i, c)$. Now, any point on the *other side* of the hyperplane $x_i{=}c$ will be at least at a distance $|a_i{-}c|$ from the target point $\{a_i\}_{i=1}^{d}$. Therefore, if $|a_i{-}c|{>}||a, b||_2$ (the Euclidean distance between a and b), we must prune the other subtree *i.e.,* one rooted at $s(\texttt{Nd}(b))$, otherwise we visit it in the *next iteration*.

A subtree once visited is not visited again and thus we traverse back to the root of Υ. At the termination of the iterative scan of Υ, the current best guess is returned as the nearest neighbour of a.

5.3 LockFree-kD-tree: Implementation

5.3.1 Lock-free Synchronization: Basics

In a sequential setting, when $\textsc{Remove}(a)$ modifies $g(\texttt{Nd}(a))\rightsquigarrow p(\texttt{Nd}(a))$, no operation is executed concurrently with a possibility to modify either of the links - $p(\texttt{Nd}(a))\rightsquigarrow\texttt{Nd}(a)$ or $p(\texttt{Nd}(a))\rightsquigarrow s(\texttt{Nd}(a))$. However, in a concurrent setting, where these pointers are shared by multiple operations, an $\textsc{Add}$ operation can concurrently modify any of these pointers. It may result into the newly added node not being a part of the LFkD-tree. Similarly, if $s(\texttt{Nd}(a))$ is an internal-node, a concurrent $\textsc{Remove}$ operation trying to remove a child of $s(\texttt{Nd}(a))$ may end up connecting $p(\texttt{Nd}(a))$ to the sibling of the removed child which results into a wrong outcome. Essentially, for a correct concurrent implementation of modify operations in a LFkD-tree, we need to keep the pointers $p(\texttt{Nd}(a)) \rightsquigarrow \texttt{Nd}(a)$ and $p(\texttt{Nd}(a)) \rightsquigarrow s(\texttt{Nd}(a))$ *fixed* when $g(\texttt{Nd}(a)) \rightsquigarrow p(\texttt{Nd}(a))$ is updated to $g(\texttt{Nd}(a)) \rightsquigarrow s(\texttt{Nd}(a))$. Additionally, because we maintain parent pointers, we also need to keep the pointer $g(\texttt{Nd}(a)) \rightsquigarrow p(\texttt{Nd}(a))$ fixed when $s(\texttt{Nd}(a)) \rightsquigarrow p(\texttt{Nd}(a))$ is updated to $s(\texttt{Nd}(a)) \rightsquigarrow g(\texttt{Nd}(a))$, in case $s(\texttt{Nd}(a))$ is an internal node.

For a lock-free synchronization we can not use locks to keep these shared pointers fixed. Instead of locks, we design the *helping* protocol for operations. Basically, the idea is: whenever an operation encounters a shared pointer fixed (although not by a lock) by a concurrent modify operation, *i.e.,* obstructed, it takes necessary steps to complete the pending operation and thereby avoids the obstruction in its own progress. This ensures that no non-faulty thread is blocked due to a delayed or crashed thread and thereby provides progress guarantee.

Ellen *et al.* [4] suggested to put *operation descriptors*, using CAS, at the nodes $g(\texttt{Nd}(a))$ and $p(\texttt{Nd}(a))$ by a Remove operation and at $p(\texttt{Nd}(a))$ by an Add operation, before updating the necessary pointers. An operation descriptor stores information about the changes that a modify operation needs to make. If CAS fails, appropriate helping is performed, using the information from the descriptor, before a reattempt.

Natarajan *et al.* [6] suggested that instead of putting the descriptors at the nodes $g(\texttt{Nd}(a))$ and $p(\texttt{Nd}(a))$, putting them at the links $p(\texttt{Nd}(a)) \rightsquigarrow \texttt{Nd}(a)$ and $p(\texttt{Nd}(a)) \rightsquigarrow s(\texttt{Nd}(a))$ improves performance. Both these designs use single-word-sized CAS to put descriptors and update the pointers.

As mentioned before, the basic structure of our LFkD-tree is based on an external BST. Therefore, for the lock-free synchronization in the LFkD-tree, we build upon the lock-free BST algorithm of [6]. The fundamental idea of the design is a *lazy remove* procedure that is essentially based on a protocol of atomically injecting *operation descriptors* on the links connected to the node to be removed, and then modifying those links to disconnect the node from the LFkD-tree. If multiple concurrent operations try to modify a link simultaneously, they synchronize by *helping* one of the pending operations that would have successfully injected its descriptor.

More specifically, to Remove the node $\texttt{Nd}(a)$, as shown in the Figure 5.2(b), we use a CAS to inject operation descriptors at the links $p(\texttt{Nd}(a)) \rightsquigarrow \texttt{Nd}(a)$, $g(\texttt{Nd}(a)) \rightsquigarrow p(\texttt{Nd}(a))$ and $p(\texttt{Nd}(a)) \rightsquigarrow s(\texttt{Nd}(a))$, in this order. We call these descriptors `mark`, `tag` and `flag` respectively. An operation descriptor works as an *information source* about the steps already performed in Remove(a) and thus a concurrent operation, if obstructed at a link with descriptor, *helps*

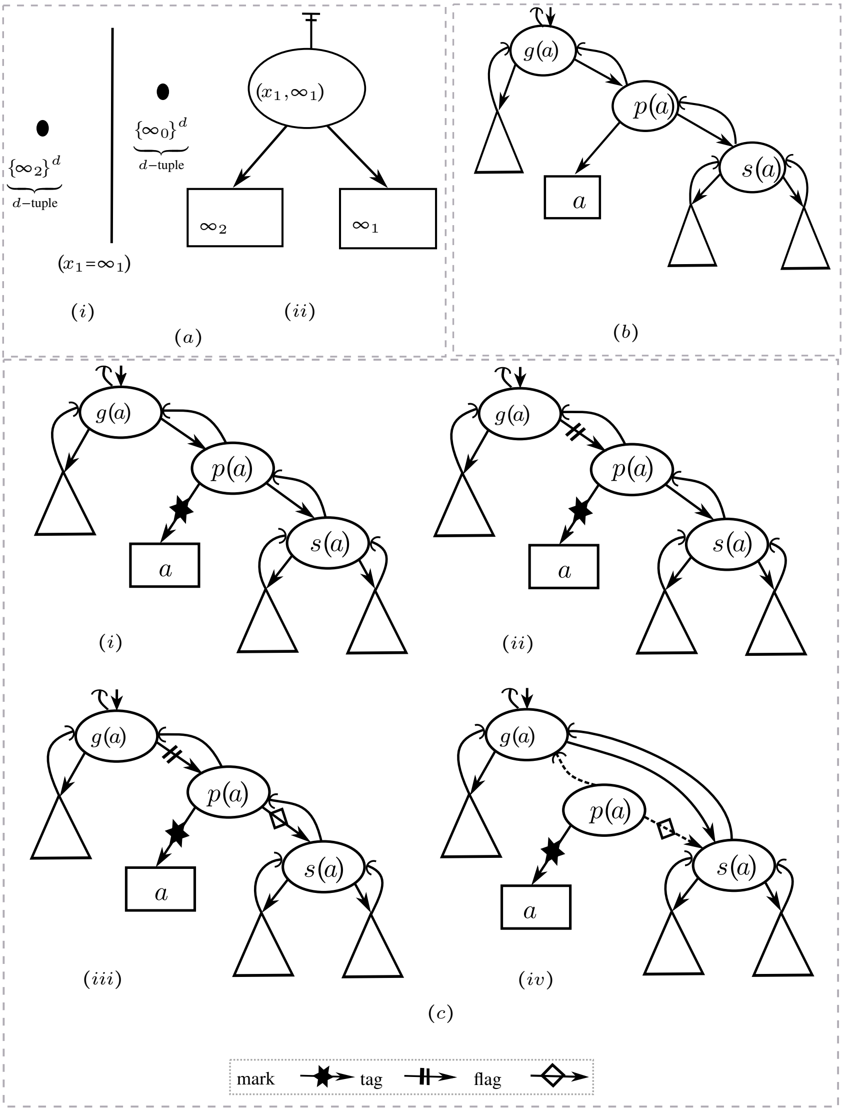


Figure 5.2: ADD and REMOVE operations in LFkD-tree

by performing the remaining steps. In particular, a `mark` at a link indicates that the next step would be to inject a `tag` at the link $g(\texttt{Nd}(a)) \rightsquigarrow p(\texttt{Nd}(a))$, whereas, a `tag` indicates that the next step is to inject the descriptor `flag` at the link $p(\texttt{Nd}(a)) \rightsquigarrow s(\texttt{Nd}(a))$. Finally, a `flag` indicates the completion of steps of injecting operation descriptors and thereafter the required link updates are done. The *helping mechanism* ensures that the concurrent ADD and REMOVE operations do not violate any invariant maintained by the LFkD-tree. The steps of a REMOVE operation are shown in the fig. 5.2(c). An ADD operation uses a single CAS to update the target link only if it is free from any operation descriptor, otherwise it helps the concurrent pending REMOVE operation. A CONTAINS or NNSEARCH operation does not perform help.

We call the CAS step, which injects a `mark` at $p(\texttt{Nd}(a)) \rightsquigarrow \texttt{Nd}(a)$, the *logical remove* of a. After this step, a CONTAINS(a) that reads $p(\texttt{Nd}(a)) \rightsquigarrow \texttt{Nd}(a)$ returns false. Accordingly, ADD(a) helps to complete the pending REMOVE(a), if it reads $p(\texttt{Nd}(a)) \rightsquigarrow \texttt{Nd}(a)$ with a `mark` descriptor, and then reattempts its own steps. The helping mechanism guarantees that a logically removed node will be eventually detached from the LFkD-tree.

To realize the atomic step to inject an operation descriptor, we replace a link using a CAS with a single-word-sized packet of a link and a descriptor. Given a pointer delegates a link, a well-known method in C/C++ to pack extra information with a pointer in a single memory-word is *bit-stealing*. In a x86/64 machine, where memory allocation is aligned on a 64-bit boundary, three least significant bits in a pointer are unused. The three operation descriptors used in our algorithm fit over these bits.

For ease of exposition, we assume that a memory allocator always allocates a variable at a new address and thus an ABA problem does not occur. For lock-free memory reclamation in a C/C++ implementation of the algorithm, a method such as one based on reference counting [20] can be used. Whereas, traditionally a Java implementation uses the JVM garbage collector. Furthermore, to avoid null pointers at the beginning of an application, we use a subtree containing an internal-node and two leaf-nodes which work as *sentinel nodes*. See fig. 5.2(a). The keys in the sentinel nodes maintain $\infty_0 > \infty_1 > \infty_2 > k_i$, $1 \leq i \leq d$, for any data point $\{k_i\}_{i=1}^{d}$ stored in the LFkD-tree. The sentinel internal-node $\texttt{Nd}(1, \infty_1)$ works as the root of the LFkD-tree and the entire dataset is stored in its left subtree.

```
1 class INode { ▷ A subclass of Node.
      long i; double c;
      Node∗ lt, rt, pr;
2 }
3 class LNode { ▷ A subclass of Node.
      K k;
4 }
5 root := INode∗(1, $\infty_1$, LNode∗($\{\infty_2\}^d$), LNode∗($\{\infty_0\}^d$), null);
```

Algorithm 5.1. The node structure in the LFkD-tree

5.3.2 Linearizable ADD, REMOVE and CONTAINS operations

(A) Overview

First, we present the node-structures in the LFkD-tree, which will help in the subsequent discussion. The classes **INode** and **LNode**, which represent an internal- and a leaf- node respectively, are shown in lines 1 and 3 in Algorithm 5.1. Every **INode**, in addition to the fields i and c that represent the associated hyperplane, has three pointers lt, rt and pr that delegate the *left-child*, *right-child* and *parent* links, respectively. An **LNode** contains only an array k to represent a data-point $k=\{k_i\}_{i=1}^{d} \in \mathbb{R}^d$. The node-pointer root, line 5, delegates address of the sentinel node $\mathtt{Nd}(1, \infty_1)$. As a convention, if x is a *field* of a class C, we use pc·x to indicate the field x of an instance of C pointed by pc; and, the type of a pointer to an instance of C is indicated by C∗. Note that, **INode** and **LNode** inherit **Node**.

(B) The algorithm

```
Dir(Node∗ Nd(i,c).ref, K k)                    ▷Return a child-direction.
    return k[i] ¡ c ? L : R;                    ▷Directions - L: left, R: right.

Child(Node∗ pa, dir cD)                         ▷Return a child-pointer.
    return cD = L ? pa.lt : pa.rt;

Search(Node∗ pa, Node∗ a, K k, , )
    while Ptr(a)·class ≠ LNode do
        pa := Ptr(a); a := Child(pa, Dir(pa, k));
    return ⟨pa, a⟩;
```

Algorithm 5.2. LFkD-tree: `Search` method

We have already described in Section 5.3.1 the operation descriptors and their denotation about the different steps of a REMOVE operation. In the following algorithms, we use the methods `IsMark`, `IsFlag` and `IsTag` to check whether a pointer has descriptor `mark`, `flag` and `tag` (actually `ltag` and `rtag`, see below), respectively. Further, to pack these descriptors, we use the methods `Mark`, `Flag` and `Tag`, respectively. To get the value of a pointer free from all descriptors, which gives a node-address, we use the method `Ptr`(). The ADD, REMOVE and CONTAINS operations, along with the methods called by them, are described in a modular fashion in the Algorithm 5.2.

The basic methods `Dir` and `Child` are used in traversal. The method `Search`, line 3 to 5, which performs a query, returns the pointers to the leaf-node and its parent, where the query terminates.

```
   CONTAINS(K k)
6      pa := root; a := pa.lt;
7      ⟨pa, a⟩ := Search(pa, a, k, , );
8      if !IsMark(a) then
9          Sync(Ptr(pa), Ptr(a));
10         return k = Ptr(a).k ? true : false;
11     else return false;
```

Algorithm 5.2. LFkD-tree: The CONTAINS operation

A CONTAINS, line 6 to 11, starts with calling `Search`, returns true only if the pointer a does not have `mark` and the query key matches at the leaf-node pointed by a at line 10; else it returns false, line 11. A CONTAINS calls `Sync`, line 9, before return to synchronize with concurrent NNSEARCH operations. We describe `Sync` in the Section 5.3.3.

The method `AddNode`(), line 12 to 25, attempts to add a new node in the LFkD-tree. It starts with calling `Search`, line 14. If the returned leaf-node-pointer a is found containing `mark`, it indicates that the node containing the query key is logically removed, and therefore, the method `Help`() is called to help the concurrent pending REMOVE operation, line 24. Otherwise, the node pointed by a is checked whether it contains the query key, line 16, and if found, false is returned, line 17. `AddNode`() also outputs the descriptor-free pointers to the leaf-node and its parent where the query terminated. However, if the leaf-node did not contain the query-key, it is checked whether a has the descriptor `flag`, which indicates a pending REMOVE of the *sibling* of the node pointed by a; and if `flag` is found, `Help`() is called, line 18. We describe `Help`() in the next subsection. Only in the case a is descriptor-free, the method `NewNode`() (see lines 26 to 31) is called to allocate a new node, and a `CAS` executed in the method `ChCAS`() (see lines 32 to 36), called at line 22, modifies a to add the new node. On that, return includes true.

The operation ADD, line 37 to 38, calls `AddNode`() to get the pointer to the node and its parent, either added by itself or already present there, containing its query key, and the result of addition accordingly. Thereafter, ADD calls the method `Sync`, line 38, and outputs the result.

```
AddNode(K k)
  pa := root; a := pa.lt;
  while true do
    ⟨pa, a⟩ := Search(pa, a, k, , );
    if !IsMark(a) then
      if k = Ptr(a).k then
        return ⟨Ptr(pa), Ptr(a), false⟩;
      if IsFlag(a) then pa := Help(pa, a);
      else
        n := LNode(k); cD := Dir(pa, k);
        newNd := NewNod(a, n.ref, pa);
        if ChCAS(pa, a, newNd.ref, cD) then
          return ⟨newNd.ref, n.ref, true⟩;
    else pa := Help(pa, a);
    a := Child(pa, Dir(pa, k));

NewNod(Node∗ a, Node∗ b, Node∗ p)          ▷Crates a new internal-node.
  ka := a.k; kb := b.k;
  i := {i : 1≤i≤d and |ka[i]−kb[i]|≥{|ka[j]−kb[j]|}$_{j=1}^{d}$};
  c := $\frac{ka[i]+kb[i]}{2}$;                       ▷Local-midpoint rule is applied.
  left := (ka[m] < kb[m] ? a : b);
  right := (ka[m] > kb[m] ? a : b);
  return INode(m, c, left, right, p);

ChCAS(Node∗ pa, Node∗ exp, Node∗ new, dir cD)
  if (cD = L) and pa.lt = exp then
    return CAS(pa.lt.ref, exp, new);
  else if (cD = R) and pa.rt = exp then
    return CAS(pa.rt.ref, exp, new);
  else return false;

ADD(K k)
  ⟨pa, a, result⟩ := AddNode(k);
  Sync(pa, a); return result;
```

Algorithm 5.2. LFkD-tree: The ADD operation

The REMOVE operation, line 39 to 49, performs query in a similar way calling `Search`, line 41. At the return of `Search`, if a is found to have `mark`, it indicates that even if the query key *k* was present in the LFkD-tree, has already been logically removed and therefore REMOVE returns false, line 48. If a is free of `mark`, we check if the node pointed by a contains the query key, and if not, REMOVE returns false, line 43. However, if the pointer a is found to have the descriptor `flag`, it indicates a pending REMOVE of the sibling of the node pointed by a, and therefore we call the method `Help`() to perform helping steps. After return of `Help`(), the steps are reattempted. Finally, if a was descriptor-free, `mark` is injected on it via the method `ChCAS`(), line 46, and if it succeeds, the `Help`() is called to take further steps and true is returned, line 47.

```
REMOVE(K k)
39  pa := root; a := pa.lt;
40  while true do
41      ⟨pa, a⟩ := Search(pa, a, k, , );
42      if !IsMark(a) then
43          if k ≠ Ptr(a).k then return false;
44          if IsFlag(a) then pa := Help(pa, a);
45          marker := Mark(a); cD := Dir(pa, k);
46          else if ChCAS(pa, a, marker, cD) then
47              Help(pa, a); return true;
48      else return false;
49      a := Child(pa, Dir(pa, k));
```

Algorithm 5.2. LFkD-tree: The REMOVE operation

(C) The Helping steps

The method `Help`() is described In the Algorithm 5.3, line 1 to 6. We call `Help`() at a pointer to a leaf-node which has been injected with either the descriptor `mark` or `flag`. Therefore, it first decides the type of descriptor, and then accordingly calls either `HelpMrk`, line 5, or `HelpFlg`, line 6.

The method `HelpMrk`, line 7 to 10, first calls `ApndTag` to fix the $g(\texttt{Nd}(a))$, pointed by ga. And then calls `HelpTag` to complete the remaining steps of REMOVE. To distinguish between the `tag` put by the REMOVE of left and right child of $p(\texttt{Nd}(a))$, we use two types of `tag`: `ltag` and `rtag`. In the method `ApndTag`, line 13 to 26, if the link was found already tagged, the type of `tag` (`ltag` or `rtag`) is read using the method `TagDir`. And, if the link was found to be tagged by a REMOVE of the other child of $p(\texttt{Nd}(a))$, first that REMOVE is helped and then we reattempt, line 19, otherwise we return ga, line 18. However, if the link $g(\texttt{Nd}(a)) \rightsquigarrow p(\texttt{Nd}(a))$ is found `flagged`, line 22, it indicates a pending REMOVE of $s(p(a))$ and therefore we help it before

```
Help(Node* pa, Node* a)
    cD := (a.k[pa·i] ¡ pa·c) ? L : R;
    if IsFlag(a) then
        ga := Pr(pa); sa := Child(pa, !cD);
        pD := (a.k[ga·i] ¡ ga·c) ? L : R;
        return HelpFlg(ga, pa, sa, pD);
    else return HelpMrk(pa, a, cD);

HelpMrk(Node* pa, Node* a, dir cD)
    ga := ApndTag(pa, a, cD);
    pD := Dir(ga, a.k); pl := Child(ga, pD);
    if Ptr(pl) = pa then HelpTag(ga, pl, pD);
    return ga;

HelpTag(Node* ga, Node* pl, bool pD)
    pa := Ptr(pl); sD := (TagDir(pl) = L ? R : L);
    HelpFlg(ga, pa, AddSp(pa, sD, ,) sD);

ApndTag(Node* pa, Node* a, dir cD)
    while true do
        ga := Pr(pa); pD := Dir(ga, a.k);
        pl := Child(ga, pD);
        if Ptr(pl) = pa then
            if IsTag(pl) then
                if TagDir(pl) = cD then return ga;
                else HelpTag(ga, pl, pD);
            else if IsFlag(pl) then
                grGa := Pr(ga);
                HelpFlg(grGa, ga, pa, Dir(grGa, a.k));
            else if ChCAS(ga, pl, Tag(pl, cD), pD) then
                return ga;
        else if pl = a then pa := ga;
        else return ga;

AddSp(Node* pa, dir sD, )
    while true do
        sa := Child(pa, sD);
        if IsMark(sa) then return sa;
        else if IsFlag(sa) then return Ptr(sa);
        else if IsTag(sa) then HelpTag(pa, sa, sD);
        else if ChCAS(pa, sa, Flag(sa), sD) then
            return sa;
```

Algorithm 5.3. LFkD-tree: Help() method

```
    HelpFlg(Node∗ ga, Node∗ pa, Node∗ sa, dir pD)
34  if Ptr(pl := Child(ga, pD)) = pa then
35      if Pr(Ptr(sa)) = pa then
36          CAS(Pr(Ptr(sa)).ref, pa, ga);
37      ChCAS(ga, pl, sa, pD);
38  return ga;
```

Algorithm 5.3. LFkD-tree: `Help()` method

reattempt. On successfully `tagging` the link $g(\mathtt{Nd}(a)) \rightsquigarrow p(\mathtt{Nd}(a))$, we return the pointer ga, line 24. Also, if $g(\mathtt{Nd}(a))$ is found not connected with $p(\mathtt{Nd}(a))$, we return ga, line 26, and REMOVE operation terminates because it indicates the completion.

The method `HelpTag`, line 11 to 12, reads the direction of the child whose REMOVE had tagged the link $g(\mathtt{Nd}(a)) \rightsquigarrow p(\mathtt{Nd}(a))$ (represented by *pl*), line 11, `flags` the (sibling) link calling `AddSp` and finally calls `HelpFlg` to perform the remaining steps, see line 12.

In `AddSp`, line 27 to 33, if the link $p(\mathtt{Nd}(a)) \rightsquigarrow s(\mathtt{Nd}(a))$ (represented by sa) was found `marked`, line 29, we return this link as it is, because it is guaranteed that the REMOVE operation that `marked` this link, will perform helping before reattempting its `CAS` to put a `tag` in the method `ApndTag`. In that case, the `marked` link is further carried to the method `HelpFlg` and connected to $p(\mathtt{Nd}(a))$. If $p(\mathtt{Nd}(a)) \rightsquigarrow s(\mathtt{Nd}(a))$ is found flagged, we return $s(\mathtt{Nd}(a))$, represented by the value of sa without any descriptor i.e. `Ptr`(sa), line 30. On a successful `CAS` to `flag` the link, we return address of $s(\mathtt{Nd}(a))$ represented by sa, line 33.

Finally, the method `HelpFlg`, line 34 to 37, if required, connects the pr pointer of $s(\mathtt{Nd}(a))$ to $g(\mathtt{Nd}(a))$, see line 36. And lastly, node a is detached from the LFkD-tree by connecting $s(\mathtt{Nd}(a))$, represented by *sa*, to $g(\mathtt{Nd}(a))$ using a `CAS` at line 37.

5.3.3 Linearizable Nearest Neighbour Search

In this section, we begin with the algorithm that addresses the case where concurrent NNSEARCH operations have coinciding target points. We build on it to present the algorithm for general cases without any restriction. However, before describing the NNSEARCH algorithms, we discuss the linearizability of the operations as its motivation.

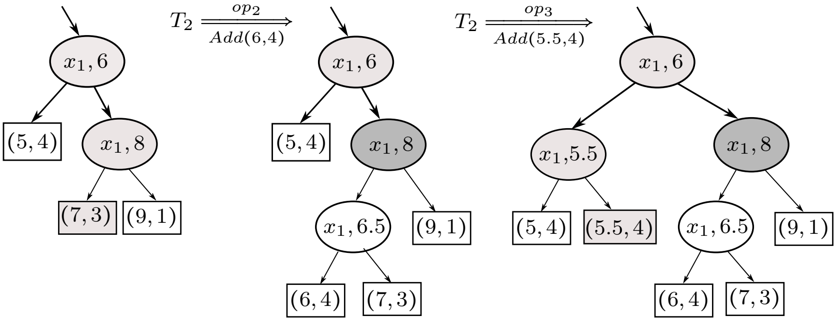


Figure 5.3: Illustration of modification operations concurrent with an NNSEARCH$(6, 4)$ operation. Light shaded nodes denote nodes currently on the traversal path of NNSEARCH and the dark shaded nodes denote roots of sub-trees that have been pruned.

(A) Linearization argument

As an illustration, consider Figure 5.3. In this example, an NNSEARCH(6.4) operation op_1 by thread T_1 is concurrent with modification operations by thread T_2 in the LFkD-tree. T_2 completes operation op_2, ADD$(6, 4)$, to a sub-tree that has already been traversed by T_1, then proceeds to complete operation op_3, $Add(5.5, 4)$ to a sub-tree that is yet to be traversed by T_1. Thus, T_1 observes the operation op_3 but not op_2, even though, to T_1, $op_2 \rightarrow op_3$. In case op_1 returns $(5.5, 4)$ as the nearest neighbour, then the operations op_1, op_2 and op_3 can not be linearized as explained in the Section 5.1.1. Thus, op_2 essentially needs to *report* its modification to op_1, after completing its own steps.

Suppose that op_2 got delayed after adding a new node $\texttt{Nd}(6, 4)$ to the LFkD-tree and could not report it to op_1. If in the a concurrent CONTAINS operation, say op_4 by thread T_3, reads node $\texttt{Nd}(6, 4)$ and later make modifications to the tree that are observable to op_1 and thus linearizable before op_1. Similarly, operations op_4 and op_1 can not be ordered sequentially without violating linearizability.

Therefore, op_4 also needs to report its output to op_1. Now, given that op_2, op_3 amd op_4 are made to report their modifications to op_1, we need to change the linearization point of op_1. To maintain the order, we put the linearization point of op_1 just after reading reports made by concurrent operations before returning the result of the iterative scan..

Note that, we need to be careful about unnecessary reporting, which may possibly be degrade performance. Suppose that op_2 and op_3 both got delayed after their linearization. Now, if invocation of *op* happened after that, *op* is guaranteed to read `Nd`, if `Nd` contained the nearest neighbour of the target point. But, if in between the linearization of op_3 and invocation of *op*, a concurrent REMOVE removed `Nd`, *op* will certainly not read it, and a reporting may render the linearization point of *op* to be shifted to even before its invocation, which is undesired. To avoid this situation, before every reporting, we first ascertain whether the node to be reported is logically removed by calling the method `IsMark()`.

```
1  class Nebr {Node∗ a; double d;}                       ▷ Neighbour
2  class NNCNode {                                       ▷ Neighbour-collector
     K tgt; bool isAct
     Nebr∗ col, rep;
     NbrClctr∗ next;
3  }
4  ncp := NbrClctr∗(null, false, null, null, null);
5  tail := NbrClctr∗(null, null, null, null, false);
6  head := NbrClctr∗(null, null, null, tail.ref, false);
```

Algorithm 5.4. LFkD-tree: Structure of Neighbour-collector

Before describing the algorithm for NNSEARCH, we describe the classes to implement the *neighboour-collector*. See the algorithm 5.4. The class **Nebr**, line 1, represents a packet of a data-point, as contained in a leaf-node pointed by the node-pointer a, and its distance, given as d, from the target point of an NNSEARCH. The class **NNCNode**, line 2, represents a *neighbour-collector*: the platform for collecting and reporting the nearest neighbour. **NNCNode** contains pointers to two **Nebr** instances: col points to one that contains collected data-point during iterative scan by an NNSEARCH operation and rep points to one that contains a data-point reported by a concurrent operation, in addition to the target point tgt. It also contains a boolean isAct, which if set true, implies an *active* neighbour-collector; and a neighbour-collector-pointer nxt to implement an augmented lock-free linked-list of neighbour-collectors. The LFkD-tree is augmented with a pointer ncp, line 4, initialized to point to an *inactive* neighbour-collector.

(B) Concurrent NNSEARCH with coinciding target points

When concurrent NNSEARCH operations have coinciding target points, they can output same result by adopting a single atomic step, which is performed during the lifetime of one of them, as the linearization point for each of them; the real-time order amongst them can be taken as the order of any fixed step for example their invocation step. Thus, essentially they require a single *iterative* scan. Principally, it is similar to the linearizable snapshot algorithm of [18]. The pseudo-code of the algorithm is given in the Algorithm 5.5.

The methods `Seek` and `NextGuess`, see lines 2 and 17, are used to perform a non-recursive traversal of the LFkD-tree. We describe these methods in the subsection (D). Here, we describe how the non-recursive traversal is used to perform co-ordinated iterative scan by concurrent NNSEARCH operations.

```
NNSearch(K k)
1   pa := root; a := pa.lt; hi := $\{\infty_0\}^d$; lo := $\{-\infty_0\}^d$;
2   ⟨pa, a⟩ := Seek(pa, a, k, hi, lo);
3   dst := IsMark(a) ? $\infty$ : $||k$, a.k$||_2$;
4   if dst $\neq 0$ then return NNSync(pa, a, dst, k, hi, lo);
5   else {Sync(pa, a); return k;}

NNSync(Node∗ pa, Node∗ a, double dst, K k, K hi, K lo)
6   while true do
7       on := ncp;
8       if on·isAct = false then
9           cN := Nebr∗(a, dst); nn := NbrClctr∗(k, true, cN, cN, null);
10          if CAS(ncp.ref, on, nn) then break;
11      else
12          if ChkValid(pa, a) then dst := AdNebr(a, on, col);
13          nn := on; break;
14  nn := Collect(pa, a, dst, k, hi, lo, nn, );
15  Deactivate(nn); return Process(nn);

Collect(Node∗ pa, Node∗ a, K k, K hi, K lo, double dst, NbrClctr∗ nn, )
16  while pa $\neq$ Ptr(root) and dst $\neq 0$ do
17      ⟨pa, a⟩ := NextGuess(pa, a, dst, k, hi, lo, );
18      if ChkValid(pa, a) then dst := AdNebr(a, nn, col);
19  return nn;

AdNebr(Node∗ a, NbrClctr∗ nn, bool nt) ▷ nt (Neighbor-type):col or rep.
20  while true do
21      nbr := (nt == col) ? nn·col : report(nn);
22      if nn·isAct and !IsFinish(nbr) then
23          ⟨dst, nb⟩ := NearNbr(a, nn);
24          if nb = null then return dst;
25          if nt = col then res := CAS(nn·col.ref, nbr, nb);
26          else res := CAS(report(nn).ref, nbr, nb);
27          if res then return dst;
28      else return 0;
```

Algorithm 5.5. LFkD-tree: NNSEARCH with coinciding target points

The operation NNSEARCH, line 1 to 5, starts with calling the method `Seek`, line 2, to perform the initial query to arrive at a leaf-node. If the pointer to leaf-node a is free of descriptor mark, which indicates that the node pointed by a is not logically removed, and if the query key *k* matches at the leaf-node, which is checked by the distance between *k* and the key at the leaf-node, *k* itself is the nearest neighbour available in the dataset and NNSEARCH returns, line 5. Otherwise, NNSEARCH calls the method NNSync, which performs further steps and returns the nearest neighbour, line 4. The arrays hi and lo are used to support non-recursive traversal, described in the subsection (D). NNSync and methods called subsequently are described here.

The method NNSync, line 6 to 15, starts with checking whether ncp points to an active neighbour-collector, and if it does not, it allocates a new active neighbour-collector and attempts a CAS to modify ncp to point to the new one, line 10. In case ncp was pointing to an active neighbour-collector, we attempt to update the current best guess of nearest-neighbour as the key in the leaf-node. On an active neighbour-collector, the method Collect is called to perform a *coordinated iterative scan*, line 14.

Collect, line 16 to 18, calls the method NextGuess, line 17, to perform next iteration that can better the current best guess of the nearest neighbour. Before attempting to add the new guess, contained in a leaf-node, to the neighbour-collector using the method AdNebr, it is always checked whether the leaf-node is logically removed by calling the method ChkValid. Please note that, given a (possibly stale) pointer to a leaf-node, we can not directly check whether it was logically removed. Therefore, we also supply the pointer to the parent and thus the method ChkValid, line 41 to line 45, gets the latest pointer to the leaf-node considering the fact that a new internal-node may get added between the parent of the leaf-node and the leaf-node to be reported.

AdNebr, line 20 to 28, is called to add a collected or reported neighbour to an active neighbour-collector. It calls the method NearerNbr, shown in line 29 to 31, which returns a new neighbour only if the distance of the new guess is less than the distance of the already collected or reported neighbours to the neighbour-collector.

After completion of the iterative scan, the method Deactivate is called by NNSync at line 15. Deactivate, line 46, other than setting the IsAct to false, also injects a descriptor finish at both the neighbour-pointers of the neighbour-collector using the method BlockNebr. BlockNebr, line 34 to line 40, performs a CAS to replace a neighbour-pointer with one that has the descriptor finish over it, see lines 37 and 39. It ensures that each of the concurrent NNSEARCH operations using same neighbour-collector have same view of it after linearization. The method IsFinish returns true when called on a neighbour-pointer with descriptor finish. Thus, AdNebr can not add a new neighbour in a neighbour-collector if the corresponding pointer is injected

```
NearNbr(Node* a, NbrClctr* nn)
    distTgt := $||a.k, nn \cdot tgt||_2$; col := nn·col; rep := report(nn);
    if distTgt < col·d and distTgt < rep·d then
        return ⟨distTgt, Nebr*(a, distTgt)⟩
    else
        return ⟨distTgt, null ⟩

BlockNebr(NbrClctr* nn, bool nt)         ▷ nt (Neighbour-type):col or rep.
    nbr := (nt == col) ? nn·col : report(nn);
    while !IsFinish(nbr) do
        if nt = col then
            CAS(nn·col.ref, nbr, Finish(nbr))
        else
            CAS(report(nn).ref, nbr, Finish(nbr))
        nbr := nt == col ? nn·col : report(nn);

ChkValid(Node* pa, Node* a)
    k := a.k; ch := Child(pa, Dir(pa, k));
    while Ptr(ch)·class ≠ LNode do
        ch := Ptr(Child(ch, Dir(ch, k)));
    if IsMark(ch) then return false;
    return ch == a ? true: false;

Deactivate(NbrClctr* nn)
    BlockNebr(nn, col); nn·isAct := false; BlockNebr(nn, rep);

Process(NbrClctr* nn)
    if report(nn)·d < nn·col·d then return A(report(nn));
    else return A(nn·col);

Sync(Node* pa, Node* a)
    if ncp·isAct then
        ⟨d, nb⟩ := NearNbr(a, ncp);
        if nb ≠ null and ChkValid(pa, a) then Report(a, ncp);

Report(Node* a, NbrClctr* nn) {AdNebr(a, nn, rep);}
```

Algorithm 5.5. LFkD-tree: NNSEARCH with coinciding target points

with finish, see line 22.

Finally, the method Process, line 47 to 48, is called by NNSync to select the better candidate between the reported and the collected neighbours of the target point, which is returned to the caller NNSEARCH to output. Note that, once a neighbour-collector is *deactivated* by an NNSEARCH, the method AdNebr returns 0, line 28. This in turn, immediately terminates the **While** loop in Collect at the line 16. Thus, as mentioned in Section 5.1.2, we can observe that the *coordination* among the concurrent iterative scans at the same neighbour-collector helps a delayed NNSEARCH operation to complete faster.

The method Sync, line 49 to 51, is used by an ADD or a CONTAINS after their completion, see Algorithm 5.2 at lines 9 and 38. Sync is also used by NNSEARCH in the case a matching key is found, see line 5. It first checks the active status of the neighbour-collector and then calls the method NearerNbr to create a neighbour. If the point to be reported is not better than the current best guess available, NearerNbr returns null and in that case Sync returns without any change. Otherwise, it checks whether the leaf node with the point to be reported is logically removed by calling the method ChkValid, and then calls the method Report, which in turn calls AdNebr to add the reported neighbour, line 52.

(C) A general case of Concurrent NNSEARCH with multiple distinct target points

To allow multiple concurrent NNSEARCH with non-coinciding target points to progress together, we need to have as many active neighbour-collectors as the number of different target points. Essentially, we need to have a dynamic list of neighbour-collectors. In this list, before adding a new neighbour-collector, an NNSEARCH must scan through it so that if there was already an active neighbour-collector with a matching target point, coordination among the concurrent iterative scans with coinciding target points can be achieved. For each of the operations in the LFkD-tree to be lock-free, we ensure the lock-freedom of this list as well. Hence, we augment the LFkD-tree with a single-word CAS based lock-free list of neighbour-collectors.

The linearization points remain as before: the concurrent NNSEARCH with coinciding target points share an atomic step during the lifetime of one of them as their linearization point with some order among themselves.

The pseudo-code of the algorithm is given in the Algorithm 5.6, in which every method is absolutely same as those in the Algorithm 5.5, except NNSync and Sync. The list is initialized with two sentinel nodes pointed by tail and head, with head.nxt set as tail, as given in lines 1 and 2. A new neighbour-

```
tail := NbrClctr*(null, false, null, null, null);

head := NbrClctr*(null, false, null, null, tail);

NNSync(Node* pa, Node* a, double dst, K k, K hi, K lo)
  nn := null; mode := INIT;
  retry:
  while true do
    p := null; c := head; n := c.nxt;
    while Ptr(n) ≠ tail do
      if n = nn and mode = CLEAN then
        val := Clean(c, nn);
        if val ≠ null then return val;
        else goto retry;
      else if k = n·tgt and n·isAct then
        nn := n; mode := COLLECT; break;
      else {p := c; c := n; n := n.nxt;}
    if mode = INIT and IsMark(n) then
      CAS(p.nxt.ref, c, Ptr(n)); goto retry;
    if mode ≠ CLEAN then
      ⟨val, mode⟩ := Finalize(pa, a, dst, k, hi, lo, p, c, mode);
      if val ≠ null then return val;
    else return Process(nn);

Sync(Node* pa, Node* a)
  n := head.nxt;
  while n ≠ tail do
    if n·isAct then
      nb := NearNbr(a, n);
      if nb ≠ null and ChkValid(pa, a) then  Report(a, n) ;
      else break;
    else n := Ptr(n.nxt);
```

Algorithm 5.6. LFkD-tree: NNSearch with multiple target points

collector is added to this list at one of the ends only, which is just before the node pointed by tail. The method of maintaining this list is similar to the lock-free linked-list of Harris *et al.* [21], except the fact that no addition happens anywhere in the middle of the list. Removal of a neighbour-collector, say one pointed by c, takes two successful CAS steps: first we inject a mark descriptor at the c.nxt using a CAS and then modify the pointer p.nxt to n with a CAS, if p and n happened to be the pointers to the predecessor and successor, respectively, of the neighbour-collector pointed by c.

We use the method Mark to get a word-sized packet of a neighbour-collector-pointer and the descriptor mark, whereas, the method Ptr masks the descriptor off such a packet and does not change a neighbour-collector-pointer. Please note that, earlier we used the same notation mark for an operation descriptor over a pointer to a LFkD-tree node. However, without any ambiguity, they indicate the descriptor for the type of pointer in the context. Similarly, the methods Mark and IsMark are used depending on the context. Adding a neighbour-collector takes a single successful CAS similar to [21].

The method NNSync, line 3 to 20, as called by NNSEARCH after the initial query in Algorithm 5.5, starts with traversing the list. We maintain an **enum** variable mode that indicates the stages of NNSync. Initially, the mode is INIT. During the traversal, if an active neighbour-collector with matching target point is found, the mode is changed to COLLECT and traversal terminates, line 13. Otherwise, the traversal terminates in the mode INIT itself. On the termination of the traversal in the mode INIT, it is checked whether the neighbour-collector, where traversal terminated (in this case c), is already logically removed, line 15, and if it is, a CAS is attempted to detach it from the list and the traversal is restarted, line 16.

After that, if the mode is INIT or COLLECT, the method Finalize is called. Finalize, line 28 to 36, if called in the mode INIT, allocates a new neighbour-collector by calling the method Allocate, line 37 to 40, otherwise uses the input neighbour-collector. If Allocate could not add a new neighbour-collector, it returns null and the entire process restarts from scratch with a fresh traversal. After successfully adding a new neighbour-collector to the list or asserting that it needs to use an existing one, Finalize calls the methods Collect and Deactivate similar to those in Algorithm 5.5. On deactivating the neighbour-collector, the method Clean is called to remove it from the list and return the value of the nearest neighbour.

Clean, line 41 to 45, performs the two CAS steps to remove the neighbour-collector and calls the method Process, line 44, to compute the nearest neighbour. However, if after injecting mark, it could not modify the nxt pointer of the predecessor, it returns null, which again causes a fresh traversal in the mode `CLEAN` in `Finalize`. A traversal in mode `CLEAN`, if finds the deactivated neighbour-collector, calls the method `Clean`, line 10, to redo the remaining steps and return the nearest neighbour. If the traversal terminates in the mode `CLEAN`, that implies that a concurrent NNSEARCH would have detached the deactivated neighbour-collector and therefore `Process` is called to finish, line 20.

```
Finalize(Node* pa, Node* a, double dst, K k, K hi, K lo, NbrClctr* p, NbrClctr* c, enum md)
28  if md = COLLECT then nn := c; pre := p;
29  else if md = INIT then
30      nn := Allocate(a, dst, k, c); pre := c;
31      if nn ≠ null then mode := COLLECT;
32  if md = COLLECT then
33      nn := Collect(pa, a, dst, k, hi, lo, nn, );
34      Deactivate(nn); md := CLEAN;
35      if (val := Clean(pre, nn)) ≠ null then
36          return ⟨val, md⟩;

Allocate(Node* a, double dst, K k, NbrClctr* c)
37  cNb := Nebr*(a, dst);
38  nn := NbrClctr*(k, true, cNb, cNb, tail);
39  if CAS(c.ref, on, nn) then return nn;
40  else return null;

Clean(NbrClctr* pre, NbrClctr* nn)
41  nxt := nn.nxt;
42  while !IsMark(nxt) do
43      CAS(nn.nxt.ref, nxt, Mark(nxt)); nxt := nn.nxt;
44  if CAS(pre.nxt.ref, nn, Ptr(nxt)) then return Process(nn);
45  else return null;
```

Algorithm 5.6. LFkD-tree: NNSEARCH with multiple distinct target points

(D) The Non-recursive Traversal

The main tool of the non-recursive traversal for the iterative scan is to keep track of an (orthogonal) axis aligned bounding box (AABB) of the points in the subtrees, both visited and pruned. An AABB is described by its two corner points. We use the variables hi and lo throughout the algorithms to represent the two corner points. Initially, in order to begin the query in the operation NNSEARCH, the corner points are taken as $\{\infty_0\}^d$ and $\{-\infty_0\}^d$, see line 1 in the Algorithm 5.5, which cover the entire dataset.

The method `Seek`, line 1 to 7, which is called by NNSEARCH for the initial

```
   Seek(Node∗ $pa$, Node∗ $a$, K $k$, K $hi$, K $lo$)
1  cD := ($a$.k[$pa$·i] ¡ $pa$·c) ? L : R;
2  while Ptr($a$).lt ≠ null do
3      $pa$ := Ptr($a$); $cD$ := Dir($pa$, $k$);
4      $a$ := Child($pa$, $cD$);
5      if $cD$ = L then $hi$[$pa$·i] := $pa$·c;
6      else $lo$[$pa$·i] := $pa$·c;
7  return ⟨$pa$, $a$⟩;

   NextGuess(Node∗ $pa$, Node∗ $a$, double $dst$, K $k$, K $hi$, K $lo$, )
8  cD := ($a$.k[$pa$·i] ¡ $pa$·c) ? L : R;
9  leafKey := $a$.k;
10 while $pa$ ≠ root do
11     if $cD$ = L then ntVsted := ($pa$·c≥$hi$[$pa$·i]);
12     else ntVsted := ($pa$·c≤$lo$[$pa$·i]);
13     if $|pa·c − k[pa·i]| < dst$ and ntVsted then
14         $cD$ := ($cD$ = L ? R : L); $a$ := Child($pa$, $cD$);
15         Seek($pa.ref$, $a.ref$, $cD.ref$, $k$, $hi$)$lo$;
16         leafKey := $a$.k;
17         if (leafdst := ||$k$, leafKey||$_2$) ¡ $dst$ then
18             if !IsMark($a$) then
19                 $dst$ := leafdst; break;
20     else
21         $a$ := $pa$; $pa$ := Pr($pa$); $cD$ := Dir($pa$, leafKey);
22         if $cD$ = L then
23             if $pa$·c > $hi$[$pa$·i] then $hi$[$pa$·i] := $pa$·c;
24         else
25             if $pa$·c < $lo$[$pa$·i] then $lo$[$pa$·i] := $pa$·c;
26 return ⟨$pa$, $a$⟩;
```

Algorithm 5.7. Non-recursive traversal

query at line 2 in the Algorithm 5.5, starts with the initial AABB as described by the two arrays hi and lo with their initial values, and performs a query absolutely similar to the method Search to arrive at a leaf-node. At the termination of Seek, the arrays AABB represent the bounding box that covers every data-point that can be in the sub-tree of the parent of the leaf-node, where it terminates, which has the same direction as the leaf-node with respect to its parent. We follow the convention that an array is always passed by reference and therefore any modification at any element in a method call persists even after the return of the method call. Thus, at the return of Seek, if the query point did not match at the key of the leaf-node, we go to perform further iterations using the method NextGuess with the current bounding box which represents the rectangular region of the Euclidean space that we have covered.

The method NextGuess, line 8 to 26, performs an iteration for a better guess of the nearest neighbour given the distance of the current guess from the target point. We input the pointers to the current leaf-node and its parent along with the AABB described by its two corners. The first step is to find the direction of the current sub-tree and then decide whether the other sub-tree of the parent is visited or not, see lines 8, 11 and 12.

In essence, we check whether the axis-orthogonal hyperplane associated with the parent node is beyond the AABB. Having done that, we check whether the unvisited AABB on the other side of the hyperplane should be visited by checking its distance from the target point and comparing it with the current distance as input, see line 13. Now, if we need to visit the other sub-tree, the method `Seek` is called to perform the query and update AABB, line 15, else we traverse back to root. When we traverse back to root, the AABB is widened to cover both sub-tree rooted at an internal node, see lines 23 and 25.

Thus, the method `Collect` repeatedly calls `NextGuess` to perform an iterative scan of the LFkD-tree, see line 17 in algorithm 5.5.

(E) Approximate Nearest Neighbour Search

Practitioners prefer better query latency at the cost of exact solution in various applications that require a nearest neighbour search, which is commonly known as approximate-Nearest Neighbour (ANN) [11,22–24]. Consider a target point $q=\{q_i\}_{i=1}^{d} \in \mathbb{R}^d$ of the NNS, given $\epsilon > 0$, we say that a point k^* is the $(1+\epsilon)$-ANN of q if

$$dist(k^*, q) \leq (1 + \epsilon) dist(k, q),$$

where k is the *true* nearest neighbour to q.

Generally, in a hierarchical multidimensional data structure like kD-tree, ANN algorithms relax the pruning criterion so that an NNSEARCH operation visits lesser number of subsets and thereby it speeds up the performance. Implementing ANN in a concurrent hierarchical multidimensional data structure does not impact the design-complexity as long as we follow the same consistency framework.

5.4 Correctness and Lock-freedom

In section (A), we discussed the arguments that determine linearization steps of NNSEARCH operations when target points are coincident. We also stated in section (C) that the linearization point of an NNSEARCH operation remains unchanged even if the target points of the concurrent NNSEARCH operations do not coincide. Here we list out the linearization points of the operations as the following:

Definition 5.1 (***Linearization points***)**.**

1. *For a successful* ADD *operation, it is at line 33 or line 35 in the method* `ChCAS`*, which is called at line 22 in the method* `AddNode` *and which in turn was called by* ADD.

2. *For a successful* REMOVE *operation, it is at line 33 or line 35 in the method* `ChCAS`*, which is called at line 46 in* REMOVE.

3. *For an unsuccessful* ADD *and a successful* CONTAINS *operation it is at line 4 in the method* `Search` *called from these operations.*

4. *For an unsuccessful* CONTAINS *and* REMOVE *operation, it can be either just after the linearization point of a concurrent* REMOVE *operation or at the invocation point of these operations.*

5. *For a* NNSEARCH *operation, if it returns a data-point which was contained in a collected-neighbour, the linearization point is at line 3 in algorithm 5.7 in the method* `Seek` *called from the* NNSEARCH.

6. *For a* NNSEARCH *operation, if it returns a data-point which was contained in a reported-neighbour, the linearization point is just after the linearization point of either* CONTAINS *or* ADD *that reported the neighbour.*

It is easy to observe in the pseudo-codes presented in the chapter that these linearization points are in between $t^i(op)$ and $t^r(op)$ for an operation $op{\in}\mathcal{O}$, where $\mathcal{O}{=}\{\text{ADD}, \text{REMOVE}, \text{CONTAINS}, \text{NNSEARCH}\}$.

Now with that, given any concurrent execution history $\mathcal{H}$ of an implementation $\mathcal{I}_{\mathcal{O}}$, where $\mathcal{O}{\subseteq}\{\text{ADD}, \text{REMOVE}, \text{CONTAINS}, \text{NNSEARCH}\}$, we form an equivalent sequential history $\mathcal{S}$ by following the steps as described above. And thus it remains to be shown that such a sequential history will be consistent.

To do that, we essentially show that the invariants of the LFkD-tree, as stated in the Section 5.2.1 are maintained, and the sequential specifications as described in the Section 5.2.2, are satisfied by the consistent operations. Because the implementation of the lock-free list of neighbour-collectors is orthogonal to the implementation of the LFkD-tree, we also need to show that the invariants of the list, as stated in the Section 5.3.3(C), are maintained by the NNSEARCH operations. Therefore, first we state the invariants and present some observations and lemmas which help us in that process.

Given a LFkD-tree Υ, let $\texttt{Nd}(i, c)$ be an internal-node and $\texttt{Nd}(\{k_i\}_{i=1}^{d})$ be a leaf node. Υ maintains the following invariants:

Invariant 5.1. *A node* Nd($\{k_i\}_{i=1}^d$) *belongs to the left subtree, if* k_i*<c.*

Invariant 5.2. *A node* Nd($\{k_i\}_{i=1}^d$) *belongs to the right subtree, if* k_i*≥c.*

Invariant 5.3. *A node* Nd($\{k_i\}_{i=1}^d$) *belongs to the right subtree, if* k_i*≥c.*

A LFkD-tree state Υ_t that satisfies the invariants 5.1 to 5.3 is called a *valid state*. Now, for the list of the neighbour-collectors, we denote a neighbour-collector by NC($\{k_i\}_{i=1}^d$) if the target point that it contains is $\{k_i\}_{i=1}^d$. A neighbour-collector list maintains following invariant:

Invariant 5.4. *In the list there can not be two neighbour-collectors* NC($\{k_i\}_{i=1}^d$) *and* NC($\{j_i\}_{i=1}^d$) *such that* $k_i = j_i\ \forall\ i\ :\ 1 \leq i \leq d$*.*

To prove that the above invariants are maintained throughout the algorithms, we present following observations and lemmas.

Observation 5.1. *The fields* k *and* i *are never changed in a* **Node***.*

Observation 5.2. *Any link in a LFkD-tree is updated only using a* CAS*.*

Observation 5.3. *The sentinel nodes are never removed.*

Observation 5.4. *The* pr *pointer of the node* root *is never dereferenced.*

Going through the pseudo-code we can observe that once we allocate a node, we never call any store step on the fields k and i and any pointer update is done using a CAS. The choice of keys in the sentinel nodes verifies the third observation. The pr pointer of an internal node is dereferenced only if a REMOVE operation on any of its children is called. Thus the observation 5.3, implies the observation 5.4.

Lemma 5.1. *In each call of* Dir*, line 1, variable* Nd(*i,c*)*.ref represents a pointer which is* clean *and points to an internal-node and thus is not* null*.*

Lemma 5.2. *In each call of* Child*, line 2, pa is* clean *and points to an internal-node and thus is not* null*.*

Lemma 5.3. *In each call of* ChCAS*, line 32 to 36, pa is* clean *and points to an internal-node, whereas new is* clean *and points to a leaf-node; thus pa and a are both not* null*.*

Lemma 5.4. *In each call of* Search*, line 3, pa is* clean *and points to an internal-node, whereas a is* clean *and points to a node (internal or leaf); thus both are not* null*.*

Lemma 5.5. *In each call of* `Search`*, line 3, pa and a satisfy $a = pa.\mathsf{lt} \mid pa.\mathsf{rt}$.*

Lemma 5.6. *In each call of* `HelpMrk`*, line 7, pa is* `clean` *and points to an internal-node, whereas a is* `clean` *and points to a leaf-node; thus both are not* null.

Lemma 5.7. *In each call of* `HelpFlg`*, line 34, ga and pa are* `clean` *and point to two different internal-nodes, whereas sa is eitherpoints to a leaf-node and thus are not* null.

Lemma 5.8. *In each call of* `HelpTag`*, line 11, ga is* `clean` *and points to an internal-nodes, whereas pl is either* `ltag` *or* `rtag` *and points to an internal-node and thus are not* null.

Lemma 5.9. *In each call of* `ApndTag`*, line 13, pa and a are* `clean`*. pa points to an internal-nodes, whereas a points to a leaf-node and thus both are not* null.

Lemma 5.10. *In each call of* `ApndFlg`*, line 27, pa is* `clean` *and points to an internal-node and thus is not* null.

Lemma 5.11. *A pointer once injected with a descriptor* `mark`*,* `flag`*,* `ltag` *or* `rtag` *is not injected with any descriptor ever after.*

The lemma 5.1 to 5.10 provide a base to prove that at no point an implementation of the presented algorithm faces a segmentation fault due to the dereferencing of a null pointer during the operations ADD, REMOVE and CONTAINS. To prove these lemmas we inspect the pseudo-code in the algorithms 5.2 and 5.3. At each call of the utility methods we find that the inputs to the utility methods follow the requirements of these lemmas. A listing of the lines of the pseudo-code containing call of these methods verifies this claim. The statements of this set of lemmas is what we need to prove the next set of lemmas which provides the verified base for postconditions of the LFkD-tree operations.

Lemma 5.12. *At the termination of* `Search` *at line 5,*

(a) *pa points to an internal-node and is* `clean`.

(b) *a points to a leaf-node and can be either* `clean` *or* `mark` *or* `flag`.

(c) *pa and a satisfy $a = pa.\mathsf{lt} \mid pa.\mathsf{rt}$.*

(d) $a.\mathsf{k}[pa{\cdot}\mathsf{i}] \geq pa{\cdot}\mathsf{c} \implies a = pa.\mathsf{rt}$.

(e) $a.\mathsf{k}[pa{\cdot}\mathsf{i}] < pa{\cdot}\mathsf{c} \implies a = pa.\mathsf{lt}$.

Following from the lemmas 5.4 and 5.5, the **while** loop ensures that the variable a always points to one of the child-pointers of the node pointed by pa; this ensures the validity of the lemma 5.12 (a), (b) and (c).

Now, Following the lemma 5.11 shows that the CAS steps are performed orderly in a REMOVE operation. It is easy to verify that if the CAS steps are orderly in a REMOVE operation, it does not result into the malformation of the LFkD-tree. Also, for an ADD operation, because the single CAS that it requires can not happen over a link with descriptor.

Now, the keys in the sentinel nodes vacuously prove the following lemma 5.13, which provides base condition for an induction to prove the theorem 5.1.

Lemma 5.13. *Initially, the LFkD-tree consisting of the sentinel nodes satisfies the invariants as stated in Section 5.2.1.*

Now we are prepared to prove theorem 5.1. We use induction to prove it. Using lemma 5.13, when no update has happened, the nodes in the LFkD-tree satisfy the invariants. It is straightforward to observe that no CONTAINS or NNSEARCH operation involves a write (CAS) step and therefore they do not change the state of the LFkD-tree. From lemma 5.12, at the end of every call to Search, which satisfies the symmetric order of the LFkD-tree, a CAS to ADD does not violate the invariant 5.1 to 5.3. For a REMOVE operation, after the CAS to logically removing the node *i.e.,* mark CAS, the order of CAS do not let any update operation let the node reappear in the LFkD-tree following the lemma 5.11.

Thus if the state of the LFkD-tree was consistent before the application of an update operation, it remains so after its linearization. Using induction the theorem 5.1 follows.

Theorem 5.1. *At any time $t \geq 0$ the LFkD-tree state Υ_t is a valid state.*

Now considering the neighbour-collector-list, its semantics are absolutely same as those of Harris's lock-free linked list [21] and which was further improved my Micheal [25]. A very sophisticated proof of the state change and thus validity of the list algorithm was provided by Micheal [25]. The invariant maintained our list, invariant 5.4, can be proved along the same lines and we skip the detail here. Now, we prove the linearizability of the implementation $\mathcal{I}_{\mathcal{M}}$ as given below.

Theorem 5.2. *(Correctness) The operations* ADD, REMOVE, CONTAINS *and* NNSEARCH *are linearizable.*

Proof. We show that a sequential history $\mathcal{S}$ obtained by following the steps: (a) in an arbitrary history $\mathcal{H}$ append appropriate response (in any arbitrary order) of all the operations which have performed their linearization steps as defined in definition 5.1 to obtain $ext(\mathcal{H})$, (b) drop the invocation steps without a matching response to obtain $complete(ext(\mathcal{H}))$, and (c) construct $\mathcal{S}$ by arranging the invocation-response pair of operations according to their linearization points, is consistent.

Let $\mathcal{S}_n$ be a sub-history of $\mathcal{S}$ that contains the *first* n complete operations. Let $\mathbb{A}_n$ be the dataset which was added to the LFkD-tree by the successful ADD operations in $\mathcal{S}_n$. Let $\mathbb{B}_n$ be the dataset which was removed from the LFkD-tree by the successful REMOVE operations in $\mathcal{S}_n$. Let $\mathbb{C}_n = \mathbb{A}_n/\mathbb{B}_n$. We use (strong) induction on n to show that $\mathcal{S}_n$ is consistent $\forall\ n{\geq}1$.

Suppose that $\mathcal{S}_n$ is consistent $\forall\ n\ :\ 1{\leq}n{\leq}i$. Let the $(i+1)^{th}$ operation in $\mathcal{S}_n$ be $op(k)$, where $k{\in}\mathbb{R}^d$. Then for $\mathcal{S}_{i+1}$ we prove the following:

1. Let $op(k)$ be an ADD operation.

   (a) Let $op(k)$ returns true. We show that if $op_1(k)$ is an ADD operation such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ and $op_1(k)$ returns true then $\exists$ a REMOVE operation $op_2(k)$ such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ and $op_2(k)$ returns true.

   Suppose there does not exist such a REMOVE operation. Now, following lemma 5.12, at the termination of `Search`, line 4 in the Algorithm 5.2, *pa*⇝*a* is a leaf-node pointer. Now using the construction of $\mathcal{S}_i$ and definition 5.1-(1), at the linearization of *op*, it performed a successful `CAS` at the link *pa*⇝*a* which must have been `clean`. Using the same argument op_1 also performed a successful `CAS` at the link *pa*⇝*a* which must have been `clean`.

   Now because op_1 linearized before *op*, the set of nodes that the `Search` called from *op*, terminates at, by the consistency of $\mathcal{S}_i$ *op* must find k being the key at that leaf-node. Now unless the link *pa*⇝*a* was already injected with the descriptor `mark`, *op* would not have continued beyond the termination of `Search` and reading the descriptor at it and thereby returning false. Therefore, there must have been a REMOVE operation which marked the link *pa*⇝*a* before *op* read and thus it had the linearization point before that of *op*. This is a contradiction.

   (b) Let $op(k)$ returns false. We show that $\exists$ an ADD operation $op_1(k)$, which returns true, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ and $\nexists$ a REMOVE operation $op_2(k)$, which returns true, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.

Suppose the contrary. Then at the termination of `Search`, line 4 in the algorithm 5.2, by definition 5.1-(3) the link $pa \rightsquigarrow a$ is `clean` and $a.\mathsf{k} = k$. But, following (a) as above and the consistency of $\mathcal{S}_i$, there must exist an $op_1(k)$ in $\mathcal{S}_i$ which returns true and that does not precede an $op_2(k)$ which returns true- which contradicts our assumption.

Now, it is easy to see that after the linearization of an ADD operation that returns true, the node added by it is reachable from root following the links and thus that node belongs to the LFkD-tree which in turn implies that $k \in \mathbb{C}_{i+1}$. Thus, combining this fact with (a) and (b) together, the mapping definition of ADD is satisfied. Thus, ADD is consistent in $\mathcal{S}_{i+1}$.

2. Let $op(k)$ be a REMOVE operation.

   (a) Let $op(k)$ returns true. We show that if $op_1(k)$ is a REMOVE operation, which returns true, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ then $\exists$ an ADD operation $op_2(k)$, which returns true, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.
   We use similar argument as given in (1) to prove it.

   (b) Let $op(k)$ returns false. We show that one of the following is true:

      i. If $op_1(k)$ is a REMOVE operation, which returns true, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ then $\nexists$ an ADD operation $op_2(k)$, which returns true, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.
      Suppose the contrary is true. Then, because $op_1(k)$ return true, by the construction of $\mathcal{S}_{i+1}$ and the definition of the linearization point definition 5.1-(2), either a leaf-node does not exist with key k or the link to it is injected with `mark`. Now if that is the case and op also returns true, then there must have been a link to a leaf-node with key k which was `clean`. But that was possible only if an ADD existed before op, which added a leaf-node with key k. This contradicts our claim.

      ii. There $\nexists$ an ADD operation $op_1(k)$, which returns true, and $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.

We can observe that at the linearization of $op(k)$, the link to the leaf-node with key k gets injected with `mark` and thus after that $k \notin \mathbb{C}_n$. Combining this fact with (a) and (b) satisfies the sequential specification of REMOVE. Thus, REMOVE is consistent in $\mathcal{S}_{i+1}$.

3. Let $op(k)$ be a CONTAINS operation.
   (a) Let $op(k)$ returns true. We show that $\exists$ an ADD operation $op_1(k)$ such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ and $\nexists$ a REMOVE operation $op_2(k)$ such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.
   The arguments are similar to (1)(b) above.
   (b) Let $op(k)$ returns false. We show that one of the following is true:
      i. If $op_1(k)$ is a REMOVE operation, which returns true, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ then $\nexists$ an ADD operation $op_2(k)$, which returns true, such that $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.
      ii. There $\nexists$ an ADD operation $op_1(k)$, which returns true, and $op_1(k) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.

   The arguments are similar to (2)(b) above. Combining (3)(a) and (3)(b), CONTAINS is consistent in $\mathcal{S}_{i+1}$.

4. Let $op(k)$ be a NNSEARCH operation that returns k^*. We show that (a) there $\exists$ $op_1(k^*)$ such that $op_1(k^*) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ and (b) if there $\exists$ $op_1(k^{**})$, which returns true, where op_1 is either ADD or CONTAINS and $||k^{**},\ k||_2 < ||k^*,\ k||_2$ such that $op_1(k^{**}) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$ then there $\exists$ a REMOVE operation $op_2(k^{**})$, which returns true, such that $op_1(k^{**}) \xrightarrow[\mathcal{S}_{i+1}]{} op_2(k^{**}) \xrightarrow[\mathcal{S}_{i+1}]{} op(k)$.

   To prove (a), it is easy to see that if such an ADD did not exist preceding *op* then at the linearization of *op* it can not read a leaf-node containing k^*. Therefore, (a) is true.

   Now, for (b), suppose the contrary is true. Thus, if there did not exist a REMOVE operation op_2 then at the linearization of *op*, which is either at the termination of the method `Seek` called by itself or at the termination of the method `Search` called by reporting CONTAINS or at the `CAS` step performed by a reporting ADD operation, the leaf-node containing k^{**} must have been connected by a `clean` link. But then either *op* would have read the `clean` link to the leaf-node with k^{**} or the operation reporting to it would have done the same. Thus the method `Process` that is called by NNSEARCH before its return, by virtue of $||k^{**},\ k||_2 < ||k^*,\ k||_2$, would have returned k^{**} which in turn would have been returned as the nearest neighbour of k by *op*. Which is a contradiction. Thus, NNSEARCH is consistent in $\mathcal{S}_{i+1}$.

By (1) to (4), $\mathcal{S}_{i+1}$ is consistent whenever $\mathcal{S}_n$ is consistent $\forall n : 1 \leq n \leq i$. Therefore, using (strong) induction, $\mathcal{S}_n$ is consistent for every positive integer n. □

Theorem 5.3. *(Lock-freedom) The LFkD-tree operations* ADD*,* REMOVE*,* CONTAINS *and* NNSEARCH *are lock-free and thus the presented algorithm implements a lock-free LFkD-tree.*

Proof. We take the NNSEARCH operation separately because it also involves the steps related to the lock-free list. By the description of the algorithm, a non-faulty thread performing a CONTAINS will always return unless its search path keeps on getting longer forever. If that happens, an infinite number of ADD operations would have successfully completed adding new nodes making the implementation lock-free. So, in the context of ADD, REMOVE and CONTAINS, it will suffice to prove that the modify operations are lock-free.

Suppose that a process $p \in \mathcal{P}$ performs a modify operation *op* on a valid state of LFkD-tree Υ_t and takes infinite steps and no other modify operation completes after that. Now, if no modify operation completes then Υ_t remains unchanged forcing p to retract every time it wants to execute its own modification step on Υ_t. This is possible only if every time p finds the injection point of *op* with descriptor `mark`, `flag`, `ltag` or `rtag`. This implies that a REMOVE operation is pending. It is trivial to observe in the method ADD that if it gets obstructed by a concurrent REMOVE, then before retrying after recovery from failure, it helps the pending REMOVE by executing all the remaining steps of that. We can also observe that whenever two REMOVE operations obstruct each other, one finishes before the other. It implies that whenever two modify operations obstruct each other one finishes before the other and so Υ_t changes. It is contrary to our assumption. Hence, by contradiction we show that no non-faulty process shall remain taking infinite steps if no other non-faulty process makes progress where the executed operation is either ADD or REMOVE.

Now we consider a NNSEARCH with concurrent ADD, REMOVE or CONTAINS operations. We consider the case where concurrent NNSEARCH operations do not necessarily have coinciding target points; this case obviously covers the case when they do have coinciding target points. We can see that a REMOVE operation does not have to report to a concurrent NNSEARCH operation. Moreover, an ADD or a CONTAINS operation to perform a reporting, needs to first traverse through the unordered list and then possibly perform a `CAS` if required to report. Now, unless the number of NNSEARCH operations keep on increasing infinitely, the total length of the unordered list will be finite and thus the traversal path for an ADD or a CONTAINS operation to report will be finite. Now, at each neighbour-collector, where the reporting is required, if a `CAS` to report fails, that implies that a concurrent CONTAINS or ADD operation succeeds.

Similarly, when a CAS by a NNSEARCH operation fails, it indicates that a CAS by a concurrent NNSEARCH operation succeeded. Finally, a CAS to add a new neighbour-collector only indicates that either a new neighbour-collector by a concurrent NNSEARCH has been successfully added or a NNSEARCH operation has terminated. In case of a CAS failure to add a new neighbour-collector, a NNSEARCH operation always helps a concurrent pending NNSEARCH operation before reattempting, in case it finds the link with descriptor `mark`. It shows that in all cases at least one non-faulty thread succeeds with respect to execute a NNSEARCH operation concurrent to any other LFkD-tree operation. Thus we arrive at the theorem 5.3. □

This concludes the proof of the presented algorithm.

5.5 A real-life application

Let us consider a web application that provides support for a real-time dynamic speed dating. The requirements of this application are as the following:

(a) Users join and leave dynamically.

(b) Users respond to a set of 5 multiple choice questions and based on the response their profile is created as a 5-tuple. A user is indexed by his/her profile.

(c) Users query for the most similar matching profile concurrently with profiles getting adding and removed.

(d) The application aims to utilize the multiple cores of a commonly available shared memory machine to get speed-up.

(e) In the fully asynchronous setting of the application, the concurrent operations must return consistent result. Additionally, progress guarantee is desired, that is, if multiple concurrent threads are assigned to the tasks of add, remove and similarity match queries by users, the application should tolerate any number of individual threads getting faulty.

We face many similar instances in our day-to-day experience with web based software. Given a 5-tuple $a{=}\{a_i\}_{i=1}^{5}$ representing the profile of a user querying similarity match, the problem here is to find the profile of a user, represented by $b{=}\{b_i\}_{i=1}^{5}$, such that $d(a,b){\le}d(a,k)\ \forall\ k{=}\{k_i\}_{i=1}^{5}$, where $d()$ is a real-valued metric and k represents a 5-tuple corresponding to an *active* user. The problem becomes challenging for the dynamic nature of the application. Furthermore, desiring speed-up along with consistency and progress guarantee broadens the challenge.

Although the above problem statement is hypothetical but to our surprise we found that the sequential kD-tree used for throughput comparison in this work is perhaps being used in a similar web application as mentioned here `http://home.wlu.edu/~levys/software/kd/`. This clearly motivates our work which can most certainly speed up such an application with a provable progress guarantee.

5.6 Experimental Evaluation

5.6.1 Experimental Setup

We implemented the LFkD-tree algorithm in Java using RTTI. We used the library objects `AtomicReferenceFieldUpdater` to perform `CAS`. The test environment comprised a dual-socket server with a 2.0GHz Intel (R) Xeon (R) E5-2650 with 8 physical cores each (32 hardware threads in total with hyper-threading enabled). The server has 64 GB of RAM, runs Ubuntu 13.04 Linux (Kernel version: 3.8.0-35-generic x86_64) with Java HotSpot (TM) 64-Bit Server VM (build 25.60-b23), and we compiled all the implementations with `javac` version 1.8.0_60.

For experimental evaluation, in addition to our designs, we considered two other implementations that support NNSEARCH. The implementations in the evaluation are:

1. Levy-Kd: An implementation of kD-tree of [26] by Levy [27] that supports REMOVE operations (we could not find any other Java implementation of a kD-tree with REMOVE). To allow for concurrent access, we augmented the implementation with coarse-grained `ReadWrite`[†] lock.

2. LFKD: Our implementation of the LFkD-tree with NNSEARCH.

3. A-LFKD: Our implementation of the LFkD-tree with *approximate*-NNSEARCH ($\epsilon = 2$).

4. PH-tree: A multi-dimensional storage and indexing data structure by Zäschke *et al.* [14] that supports REMOVE operations. Similar to *Levy-Kd*, we add coarse-grained `ReadWrite` lock to allow for concurrency.

[†] A *ReadWriteLock* consists of a pair of locks: *read* lock may be held by multiple readers as long as the write lock is free, *write* lock is exclusive)

We run each test for 5 seconds and measured throughput as the total number of operations per microsecond executed by all threads in this time duration. We run each experiment in a separate instance of the JVM, starting off with a 2-second "warm-up" period to allow the Java HotSpot compiler to initialize and optimize the running code. During this warm-up phase, we performed random Add, Remove and Contains operations, and then flushed the tree.At the start of each execution, the data structure is pre-filled with keys in the selected key-range.

To simulate the variation in contention and tree structure, we chose following combination of workload configurations: i) dataset space dimension $\in \{2, 3, 4, 5\}$, ii) distribution of (ADD-REMOVE-NNSEARCH) $\in \{(05, 05, 90), (25, 25, 50), (40, 40, 20)\}$, and iii) number of threads $\in \{1, 2, 4, 8, 16, 32\}$.

We did not include CONTAINS operations in experiment because essentially it would increase the proportion of exact-match NNSEARCH. All executions use the same set of randomly generated points for the selected workload characteristics. The graphs present average of throughput over 6 runs of each experiment.

5.6.2 Datasets

We performed evaluation using a 2D real-world dataset and a set of synthetic benchmarks. For the real-world dataset, we used the United States Census Bureau 2010 TIGER/Line KML [28] dataset that consists of polylines describing map features of the United States of America. TIGER/Line is a standard dataset used for benchmarking spatial databases. For this evaluation, we extracted points representing the mainland, resulting in $18.4 * 10^6$ unique 2-d points, with x-y coordinates that lie between -124.85 $\leqslant x \leqslant$ -66.89 and 24.40 $\leqslant y \leqslant$ 49.38.

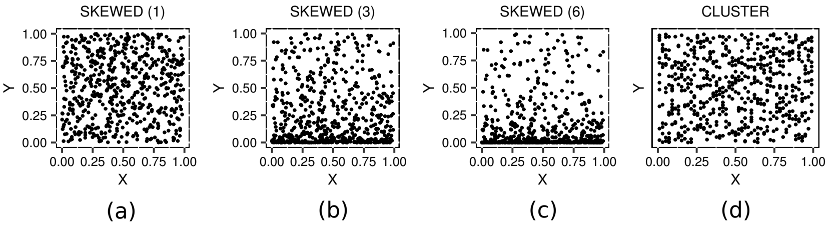


Figure 5.4: Synthetic dataset.

To investigate more variable workloads, two synthetic datasets were utilized. The SKEWED data simulates datasets in which different dimensions may have varying distributions. The SKEWED(α) dataset contains uniformly distributed points which fall within 0.0 and 1.0 in every dimension that have been skewed in the y-dimension. For each point in the dataset, the y value is replaced with the value y^{α}, for example in the 2-dimension case, each point (x, y) is replaced with (x, y^{α}). In the fig. 5.4(a), we show examples for SKEWED(1) which is intuitively uniform distribution in all dimensions. SKEWED(3) and SKEWED(6) are shown in the fig. 5.4(b) and fig. 5.4(c), respectively .

The CLUSTER dataset [14] is an extension of a synthetic dataset previously described by Arge *et al.* [29]. In this evaluation we used clusters of 1000 points evenly spaced on a horizontal line. Each of the clusters is filled with evenly distributed points and stretches 0.00001 in every dimension. Figure 5.4(d) depicts an example of the CLUSTER dataset with 49 points per cluster. The line of clusters falls within (0.0, 1.0) along the x-axis and is parallel to every other dimensional axis with a 0.5 offset.

5.6.3 Observations and Discussion

The Figures 5.5, 5.6 and 5.7 show the performance of the implementations for TIGER/Line, SKEWED and CLUSTER datasets respectively. In Figure 5.6 and 5.7, each row represents a combination of the range of the number of unique keys (k=N, N being the maximum) and the associated workload distribution while each column the dimensionality of key (d=dimension).

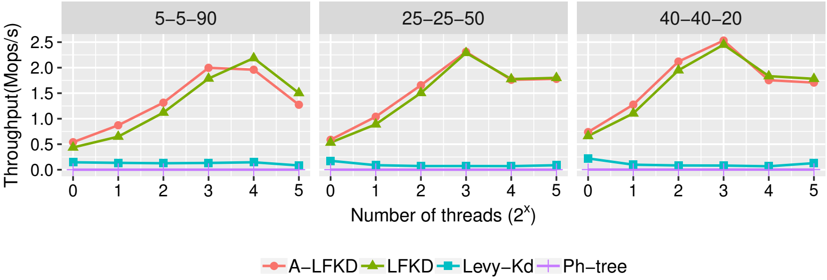


Figure 5.5: Performance on the 2-D TIGER/Line dataset.

In all of experiments, LFKD and A-LFKD have better throughput performance(in million operations per second) compared to both the PH-tree and the Levy-Kd, even in single thread cases, for all workload distributions. The performance significantly scales up with increasing thread count. This shows that our implementation is both lightweight and scalable. As we increase the key dimension, the performance degrades for workloads dominated by the NNSEARCH. This degradation with increasing key dimensions is expected in kD-trees due to the *curse of dimensionality* [1]. This performance pattern is identical for different key ranges. However, the LFKD still achieve speed-up over the single threaded implementations.

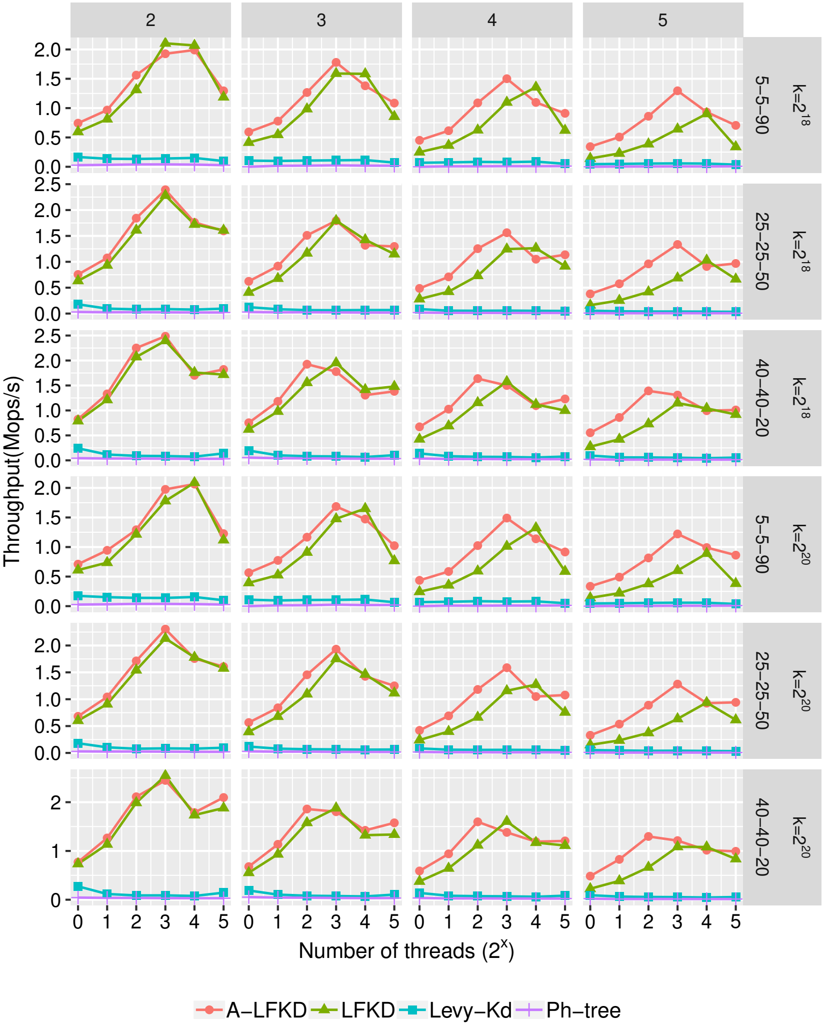


Figure 5.6: Performance on the SKEWED(6) dataset.

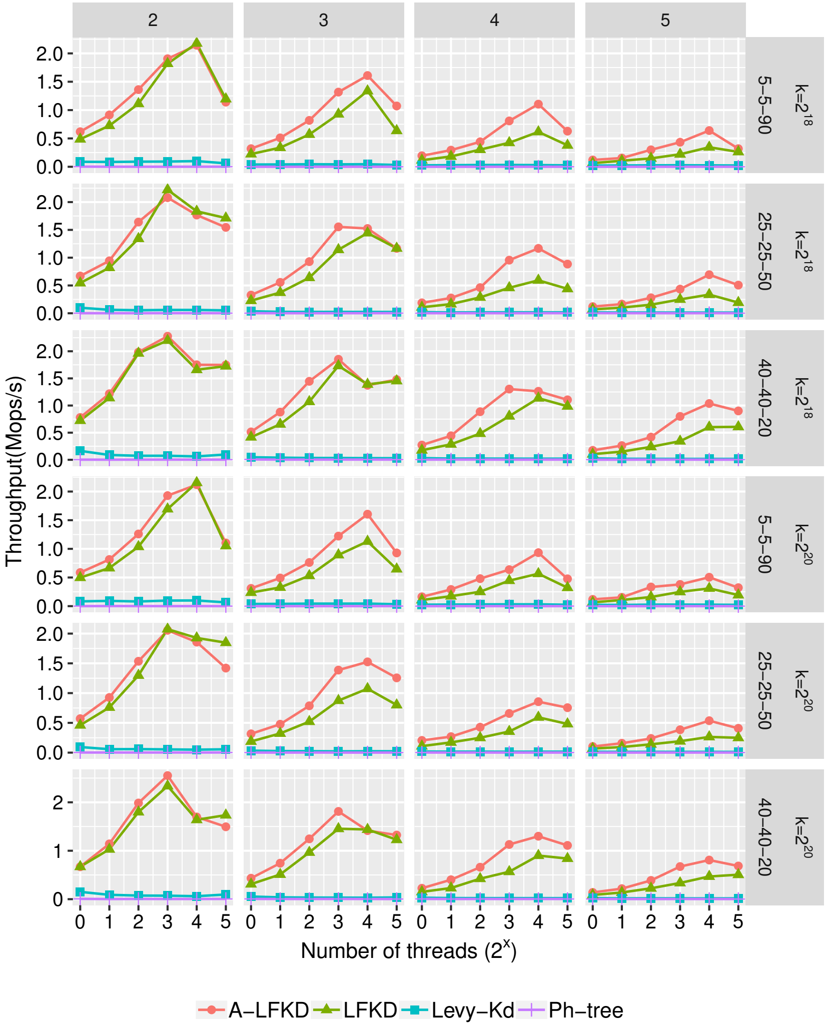


Figure 5.7: Performance on the CLUSTER dataset.

We further observe that, as expected, A-LFKD outperforms LFKD in NNSEARCH dominated workload, the performance benefit increases with increasing dimensionality of the data set that brings the increased load of iterative scan. This can be explained by early termination of the iterative scan in the A-NNSEARCH in A-LFKD which prunes parts of the tree with are otherwise traversed by the NNSEARCH in LFKD.

For the TIGER/Line dataset, in a single thread case, both LFKD and LFKD(SC) perform at least 2.5× better than Levy-Kd, and, it goes up to 19× in the NNSEARCH dominated workload. Additionally, the PH-tree outperforms the Levy-Kd only for workloads that do not involve NNSEARCH (00% NNSEARCH, 50% ADD and 50% REMOVE).

We observe that for NNSEARCH dominated workload (90% NNSEARCH, 5% ADD and 5% REMOVE), the A-LFKD achieves speed-ups up to 66× for SKEWED and up to 150× for CLUSTER datasets over the sequential implementations. These observations can be partially attributed to the local-midpoint rule, which carries the essence of the sliding-midpoint-splitting rule of [10] that targets the extreme cases such as a CLUSTER dataset, to a concurrent setting.

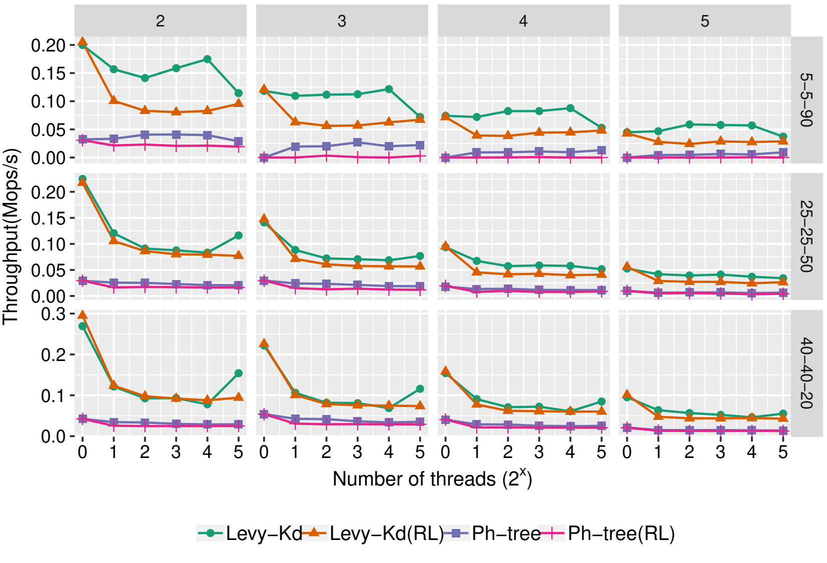


Figure 5.8: System throughput for different lock implementations.

For a mixed workload (50% NNSEARCH, 25% ADD and 25% REMOVE), the performance of LFkD-tree degrades by increasing key dimension. The absolute throughput figures are higher for the NNSEARCH dominated workload in lower dimensions than in mixed workloads. This is because the modify operations incur higher synchronization (conflicts, expensive atomic operations, and helping) overhead. However in higher dimensions, the throughput of the NNSEARCH is lower as the number of visited nodes increases tremendously with dimension.

Figure 5.8 depicts the performance of implementations augmented with the course-grained locks. In the figure, implementations with (RL) are augmented with *Reentrant* Locks while the others are implemented with *ReentrantReadWrite* locks. As expected, *ReentrantReadWrite* locks perform significantly better for read dominated workloads, and comparably for write dominated workloads. This result further highlights that even for lock-based implementations, choice of lock has a significant impact on the performance of the implementation.

5.7 Conclusion and Future Work

For a large number of applications, which require a multidimensional data structure supporting dynamic modifications along with nearest neighbour search, research community has largely focused on improving the design of sequential data structures. Parallel implementations of the sequential designs speed up loading of and NNS on a fully loaded data structure. Thus, they do not address the issue of dynamic modifications in the datasets. On the other hand, the concurrent data structure research is primarily confined to one-dimensional problems.

Our work is the first to extend the concurrent data structures to problems covering multidimensional datasets. We introduced LFkD-tree, a lock-free design of kD-tree, which supports linearizable nearest neighbour search operations with concurrent dynamic addition and removal of data. We provided a sample implementation which shows that the LFkD-tree algorithm is highly scalable.

Our method to implement linearizable nearest neighbour search is generic and can be adapted to other multidimensional data structures. We plan to design lock-free data structures which are suitable for nearest neighbour search in high dimensions, for example, the ball-tree [30]. We also plan to extend our work to k-nearest neighbour (kNN) search.

Bibliography

[1] Hanan Samet, *Foundations of multidimensional and metric data structures*, Morgan Kaufmann, 2006.

[2] Maurice Herlihy and Jeannette M. Wing, “Linearizability: A correctness condition for concurrent objects,” *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[3] Håkan Sundell and Philippas Tsigas, “Fast and lock-free concurrent priority queues for multi-thread systems,” *Journal of Parallel and Distributed Computing*, vol. 65, no. 5, pp. 609–627, 2005.

[4] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel, “Non-blocking binary search trees,” in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 2010, pp. 131–140, ACM.

[5] Shane V. Howley and Jeremy Jones, “A non-blocking internal binary search tree,” in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2012, pp. 161–171, ACM.

[6] Aravind Natarajan and Neeraj Mittal, “Fast concurrent lock-free binary search trees,” in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2014, pp. 317–328, ACM.

[7] Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas, “Efficient lock-free binary search trees,” in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 2014, pp. 322–331, ACM.

[8] Jon Louis Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[9] Jerome H. Friedman, Jon Louis Bentley, and Raphael A. Finkel, “An algorithm for finding best matches in logarithmic expected time,” *Transactions on Mathematical Software*, vol. 3, no. 3, pp. 209–226, 1977.

[10] David M Mount and Sunil Arya, “Ann: a library for approximate nearest neighbor searching,” `http://www.cs.umd.edu/~mount/ANN/`, 1998.

[11] Sunil Arya and Ho-Yam Addy Fu, “Expected-case complexity of approximate nearest neighbor searching,” *SIAM Journal on Computing*, vol. 32, no. 3, pp. 793–815, 2003.

[12] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo, “Real-time kd-tree construction on graphics hardware,” *ACM Transactions on Graphics*, vol. 27, no. 5, pp. 126, 2008.

[13] Jeffrey Ichnowski and Ron Alterovitz, “Scalable multicore motion planning using lock-free concurrency,” *IEEE Transactions on Robotics*, vol. 30, no. 5, pp. 1123–1136, 2014.

[14] Tilmann Zäschke, Christoph Zimmerli, and Moira C. Norrie, “The ph-tree: A space-efficient storage structure and multi-dimensional index,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2014, pp. 397–408, ACM.

[15] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun, “A practical concurrent binary search tree,” in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2010, pp. 257–268, ACM.

[16] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky, "Concurrent tries with efficient non-blocking snapshots," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2012, pp. 151–160, ACM.

[17] Trevor Brown and Hillel Avni, "Range queries in non-blocking k-ary search trees," in *Proceedings of the International Conference on Principle of Distributed Systems*, pp. 31–45. Springer, 2012.

[18] Erez Petrank and Shahar Timnat, "Lock-free data-structure iterators," in *Proceedings of the International Symposium on Distributed Computing*. 2013, pp. 224–238, Springer.

[19] Bapi Chatterjee, "Lock-free linearizable 1-dimensional range queries," in *Proceedings of the International Conference on Distributed Computing and Networking*. 2017, pp. 9:1–9:10, ACM.

[20] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas, "Efficient and reliable lock-free memory reclamation based on reference counting," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 8, pp. 1173–1187, 2009.

[21] Timothy L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the International Conference on Distributed Computing*. 2001, pp. 300–314, Springer.

[22] Marshall Bern, "Approximate closest-point queries in high dimensions," *Information Processing Letters*, vol. 45, no. 2, pp. 95–99, 1993.

[23] Sunil Arya and David M. Mount, "Approximate nearest neighbor queries in fixed dimensions," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*. 1993, pp. 271–280, Society for Industrial and Applied Mathematics.

[24] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *Journal of the ACM*, vol. 45, no. 6, pp. 891–923, 1998.

[25] Maged M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2002, pp. 73–82, ACM.

[26] Andrew W. Moore, "Efficient memory-based learning for robot control," Tech. Rep. 209, University of Cambridge, 1991.

[27] Simon D. Levy, "`KDTree`," in *`edu.wlu.cs.levy.CG.KDTree`*.

[28] "`https://www.census.gov/geo/maps-data/data/tiger.html`," .

[29] Lars Arge, Mark De Berg, Herman Haverkort, and Ke Yi, "The priority r-tree: A practically efficient and worst-case optimal r-tree," *ACM Transactions on Algorithms*, vol. 4, no. 1, pp. 9:1–9:30, 2008.

[30] Ting Liu, Andrew W. Moore, and Alexander G. Gray, "New algorithms for efficient high-dimensional nonparametric classification," *Journal of Machine Learning Research*, vol. 7, no. Jun, pp. 1135–1158, 2006.

PAPER V

Lazaros Papadopoulos, Dimitrios Soudris, **Ivan Walulya** and Philippas Tsigas

Customization Methodology for Implementation of Streaming Aggregation in Embedded Systems

Journal of Systems Architecture - Embedded Systems Design
Vol.: 66–67, pp. 48–60, Elsevier 2016.

6

Customization Methodology for Implementation of Streaming Aggregation in Embedded Systems

Abstract

Streaming aggregation is a fundamental operation in the area of stream processing and its implementation provides various challenges. Data flow management is traditionally performed by high performance computing systems. However, nowadays there is a trend of implementing streaming operators in low power embedded devices, due to the fact that they often provide increased performance per watt in comparison with traditional high performance systems. In this work, we present a methodology for the customization of streaming aggregation implemented in modern low power embedded devices. The methodology is based on design space exploration and provides a set of customized implementations that can be used by developers to perform trade-offs between throughput, latency, memory and energy consumption. We compare the proposed embedded system implementations of the streaming aggregation operator with the corresponding HPC and GPGPU implementations in terms of performance per watt. Our results show that the implementations based on low power embedded systems provide up to 54 and 14 times higher performance per watt than the corresponding Intel

Xeon and Radeon HD 6450 implementations, respectively.

6.1 Introduction

Efficient real-time processing of data streams produced by modern interconnected systems is a critical challenge. In the past, low-latency streaming was mostly associated with network operators and financial institutions. Processing of millions of events such as phone calls, text messages, data traffic over a network and extracting useful information is important for guaranteeing high Quality of Service. Stream processing applications that handle traditional streams of data were mostly implemented by using Stream Processing Engines (SPEs) running on high performance computing systems.

However, nowadays digital data come from various sources, such as sensors from interconnected city infrastructures, mobile cameras and wearable devices. In the device-driven world of Internet of Things, there is a need in many cases for processing data on-the-fly, in order to detect events while they are occurring. This data-in-motion comes in the form of live streams and should be gathered, processed and analyzed as quickly as possible, as it is produced continuously. Low-power embedded devices or embedded micro-servers [1] are expected not only to monitor continuous streams of data, but also to detect patterns through advanced analytics and enable proactive actions. Applying analytics to these streams of data before the data is stored for post-event analysis (data-at-rest) enables new service capabilities and opportunities.

Streaming aggregation is a fundamental operator in the area of stream processing. It is used to extract information from data streams through data summarization. Aggregation is the task of summarizing attribute values of subsets of tuples from one or more streams. A number of tuples are grouped and aggregations are computed on their attributes in real-time fashion. High frequency trading in stock markets (*e.g.,* continuously calculating the average number of each stock over a certain time window), real time network monitoring (*e.g.,* computing the average network traffic over a time window) are examples of data stream processing, where streaming aggregation along with other operators is used to extract information from streams of tuples.

Streaming aggregation performance is affected a lot by the cost of data transfer. So far, streaming aggregation scenarios have been implemented and evaluated in various architectures, such as GPUs, Nehalem and Cell processors [2]. Indeed, there is a trend to utilize low power embedded platforms on running computational demanding applications in order to achieve high performance per watt [3] [4] [5] [6].

Modern embedded systems provide different characteristics and features (such as memory hierarchy, data movement options, OS support, *etc.*) depending on the application domain that they target. The impact of each one of these features on performance and energy consumption of the whole system, when running a specific application, is often hard to predict at design time. Even if it is safe to assume in some cases that the utilization of a specific feature will improve or deteriorate the value of a specific metric in a particular context, it is hard to quantify the impact without testing. This problem becomes even harder when developers attempt to improve more than one metric simultaneously. A similar problem is the porting of an application running on a specific system to another with different specifications. The application usually need to be customized in the new platform differently, in order to provide improved performance and energy efficiency. The typical solution followed by developers is to try to optimize the implementation of the application on the embedded platform in an ad-hoc manner, which is a time consuming process that may yield suboptimal results. Therefore, there is a need for a systematic customization approach: Exploration can assist the effective tuning of the application and platform design options, in order to satisfy the design constraints and achieve the optimization goals.

Towards this end, in this work, we propose a semi-automatic step-by-step exploration methodology for the customization of streaming aggregation implemented in embedded systems. The methodology is based i) on the identification of the parameters of the streaming aggregation operator that affect the evaluation metrics and ii) on the identification of the embedded platform specification features that affect the evaluation metrics when executing streaming aggregation. These parameters compose a design space. The methodology provides a set of implementation solutions. For each solution, the application and the platform parameters have different values. In other words, each customized streaming aggregation implementation is tuned differently, so it provides different results for each evaluation metric. Developers can perform trade-offs between metrics, by selecting different customized implementations. Thus, instead of evaluating solutions in ad-hoc manner, the proposed approach provides a systematic way to explore the design space.

The main contributions of this work are summarized as follows:

i. We present a methodology for efficient customization of streaming aggregation implementation in embedded systems.

ii. We show that streaming aggregation implemented on embedded devices yields significantly higher performance per watt in comparison with corresponding HPC and general purpose GPU (GPGPU) implementations.

Finally, based on the experimental results of the demonstration of the methodology, we draw insightful conclusions on how each one of the application and platform parameters (*i.e.,* design options) affects each one of the evaluation metrics. The methodology is demonstrated in two streaming aggregation scenarios implemented in four embedded platforms with different specifications: Myriad1, Myriad2, Freescale I.MX.6 Quad and Exynos 5 octa. The evaluation metrics are throughput, memory footprint, latency, energy consumption, and scalability.

The rest of the paper is organized as follows. Related work on streaming aggregation and stream processing on embedded systems is presented in Section 2. Section 3 describes the streaming aggregation operator and the design challenges. The design space and the exploration methodology are presented in Section 4. Section 5 presents the demonstration of the methodology and in Section 6 we draw conclusions.

6.2 Related Work

Stream processing on various high-performance architectures has been studied in the past extensively. Many works focus on the parallelization of stream processing [7], [8], [9]. They describe how the stream processing operators should be assigned to partitions to increase parallelism. The authors in [10] describe another way of improving the performance of streaming aggregation; they propose lock-free data structures for the implementation of streaming aggregation on multicore architectures. The evaluation has been conducted on a 6-core Xeon processor, and the results show improved scalability.

With respect to stream processing engines (SPEs), Aurora and Borealis [11] are among the most well known ones. Several works that focus on the evaluation of stream processing operators on specific parallel architectures can be found in the literature. For example, an evaluation on heterogeneous architectures composed of CPU and a GPU accelerator is presented in [9]. The authors of [2] evaluate streaming aggregation implementations on Core 2 Quad, Nvidia GTX GPU and on Cell Broadband Engine architectures. The aggregation model used in this work is more complex, since it focuses on timestamp-based tuple processing.

There exists several works that describe the usage of low power embedded processors to run server workloads. More specifically, many works propose the integration of low-power ARM processors in servers [3] [4], or present energy-efficient clusters built with mobile processors [5].

In the area of embedded systems stream processing, several works focus on compilers that orchestrate parallelism, while they handle resource and timing constraints efficiently [12]. A programming language for stream processing in

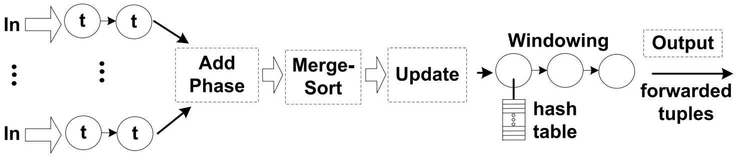


Figure 6.1: Time-based streaming aggregation scenario phases.

embedded systems has been proposed in [13]. These works are complementary to ours. The conclusions we drive from this work could assist the implementation of efficient compilers and development frameworks for stream programming.

Design space exploration in embedded systems is another area related to the present work. Exploration methodologies have been proposed for tuning at system architecture level [14], for customization of dynamic data structures [15], and of dynamic memory management optimization [16]. These customization approaches are complementary to the one proposed in the present work. Performance and energy consumption of streaming aggregation implementation could improve with effective customization of data structures or the dynamic memory management of the system.

6.3 Streaming Aggregation

In this Section we provide a description of the streaming aggregation operator and we analyze the design challenges of implementing a streaming aggregation scenario on an embedded platform.

6.3.1 Streaming Aggregation description

Streaming aggregation is a very common operator in the area of stream processing. It is used to group a set of inbound tuples and compute aggregations on their attributes, similarly to the *group-by* SQL statement. In the context of this work, we discuss two aggregation scenarios: *multiway time-based with sliding windows* and *count-based with tumbling windows*.

(A) Multiway time-based streaming aggregation

In multiway aggregation, multiple streams of incoming tuples, which are stored in queues, are combined into one stream, through a merge operator and their tuples are sorted given their timestamp attribute. It consists of 4 phases, as presented in Figure 6.1:

1. *Add*: Incoming tuples are fetched from each input stream.

2. *Merge-Sort*: The tuples are merged and sorted, by the *merge* operator.

3. *Update*: Each tuple is assigned to the windows that it contributes to.

4. *Output*: Tuples with the computed aggregated value are forwarded.

During the *Add* phase tuples from each input stream are fetched and forwarded to the Merge-Sort phase. Since the incoming tuples are stored in a queue, they are forwarded in a FIFO manner.

Merge-Sort operation is used to combine streams that were sorted on a given attribute into a single stream, whose tuples are also ordered on the same attribute. In the context of this work, the tuples are sorted in timestamp order.

Merge and *Sort* are tightly coupled operations in streaming aggregation scenarios since they share the same resource (*i.e.,* the incoming dequeued tuples) and they can be considered a single primitive operation. Merge-Sort phase ensures deterministic processing of the incoming tuples. A tuple is ready to be processed and forwarded to the next phase, if at least one tuple with an equal or higher timestamp has been received at each input stream.

In the *Update* phase the windowing operation is taking place and each single tuple is assigned to the window that it contributes to. In the context of this work, the aggregated values are computed over sliding windows, which have two attributes: *size* and *advance*. As an example, a window with *size* 5 time units and *advance* 2 time units, covers periods: [0, 5), [2, 7), [4, 9), *etc.* A tuple with timestamp 3, would contribute to windows [0, 5) and [2, 7).

In the *Output* phase, the aggregated value is calculated for all windows in which no more incoming tuples are expected to contribute (*i.e.,* completed windows). The deterministic processing of tuples that took place in the earlier phases (more specifically during the *Add* and *Merge-Sort* phases), ensures that the aggregated value will be calculated only for completed windows. A new tuple is created for each aggregated value and it is forwarded, as a result of the aggregation operator.

Multiway time-based streaming aggregation provides pipeline parallelism, which can be exploited by assigning each phase on a different processing element (PE). However, performance relies not only on the exploitation of parallelism or on the computational power that the system provides, but also on the efficient data transfer between the phases. The sorted tuples of the Merge-Sort phase are used by the Update phase to be assigned to the windows that each one contributes to. The Update phase provides to the Output phase information on the windows in which the last tuples contributed to. Thus, the Output phase identifies the completed windows and calculates the aggregated value for each one. The utilization of efficient means of forwarding the information from one

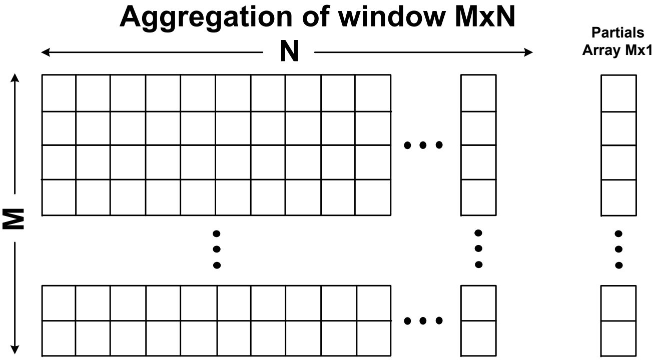


Figure 6.2: Window and partials array data structures used in the count-based streaming aggregation scenario.

phase to another, affects both performance and energy consumption. The same applies to the way by which memory accesses on shared data are synchronized. Other important implementation issues that should be taken into account are the size of the queues in which the inbound tuples are stored (input queues) and the memory allocation of both the queues and the data structure in which the windows are stored.

(B) Count-based streaming aggregation

In count-based aggregation, the window size is determined by the number of tuples buffered, instead of the time passed. Our case study considers fixed size windows and aggregation takes place periodically, *i.e.,* when a specific number of tuples is received. Every time an aggregation is completed, all currently stored tuples are evicted and the next window is initially empty (tumbling window).

To implement the count-based aggregation scenario, we followed an approach based on [2]. The time intervals between aggregations are based on the number of tuples stored in the window and results of a specific window may depend on results of the previous one. Thus, an extra data structure is needed to store the partially aggregated results of the last window, which may be used in the following aggregation.

Figure 6.2, shows the data structures used in the count-based scenario: A *M*x*N* window and the partials array, with 1x*M* entries. *M* is the maximum number of input streams and *N* is the window width. When it is not possible to compute the aggregated value of *N* tuples for a specific input stream before the current window is forwarded, the partially aggregated result is stored in partials array. This result is used by the following window to compute the aggregated

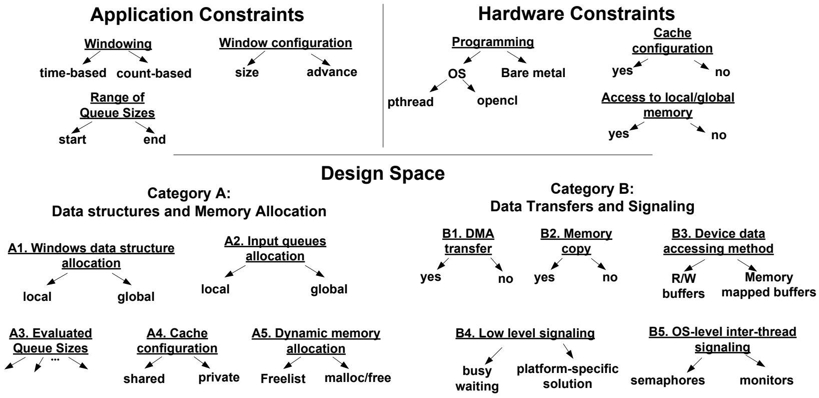


Figure 6.3: Constraints and Design space for streaming aggregation.

value of N tuples for the specific input stream. The output is a single tuple that it is produced by a query executed in the M aggregated values.

Apparently, count-based streaming aggregation provides data parallelism. Each window row can be assigned to a different processing element (PE) to compute the aggregated value of each input stream in parallel. Similarly to the time-based scenario, data transfer overhead, memory allocation issues and the window size affect the performance and the energy consumption of the operator. The embedded systems provide various solutions and each one has different impact on each evaluation metric. The design options for all the aforementioned implementation issues compose a design space that it is described in the following Section.

6.4 Customization Methodology

In this Section, we first present the design space for the streaming aggregation customization and then we describe the proposed methodology.

6.4.1 Design Space

The design space of the streaming aggregation implementation is presented as a set of decision trees, grouped into two categories (Figure 6.3):

- *Category A* consists of decision trees that refer to memory configuration and allocation. Cache configuration options (private cache for each core or shared cache for all cores) are depicted in decision tree *A4*. *A5* is related

Table 6.1: Decision trees or leaves disabled for each application and hardware constraint.

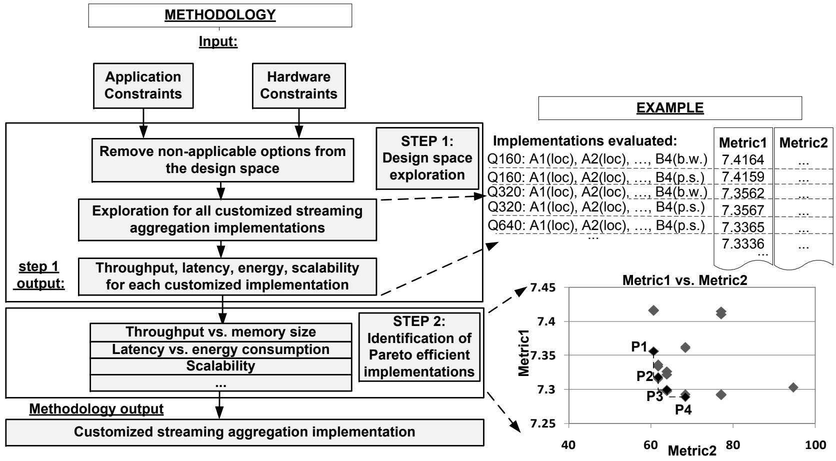
App./Hw constraint	Decision tree/ leaf disabled
Windowing(tuple-based)	A2, A3, A5, B4
Window configuration	may disable A1(local)
Programming(bare metal)	B3 and B5
Programming(pthread)	B1, B3, B4
Programming(OpenCL)	B1, B2, B4, B5
Cache config.(no)	A4
Access to local/global(no)	A1, A2

with the dynamic memory allocation that can be based on freelists or in *malloc*/*free* system calls.

- In *category B* are assigned decision trees related to data movement and means by which accesses to shared resources are synchronized. The first three decision trees refer to different ways that data can be copied from global to local memories, or from one local memory to another (depending on the embedded system's memory hierarchy). Decision trees *B4* and *B5* are about synchronization between PEs, when accessing shared buffers. At low level, synchronization can be accomplished by spinning on shared variables (*i.e.,* busy waiting) or by using other platform specific solutions. In platforms that run OS and support POSIX threads developers can utilized semaphores or monitors.

Apparently, not all design options are applicable in any context. Figure 6.3 shows the application and the hardware constraints that affect which decision trees or leaves are applicable in each specific context. The constraints are used to prune the decision trees and leaves that yield implementations which do not adhere to developer's requirements or they are not supported by the embedded platform.

Table 6.1 summarizes the design options that are disabled, due to application and hardware constraints. As an example, if the embedded platform runs an OS, access to DMA and to low-level signaling mechanisms are most likely handled by the OS directly, so these design options are not exposed to developers. *Window configuration* constraint may force the allocation of the data structures in a global memory. All constraints are provided manually. Constraints that prune non-compatible design space options "convert" the platform-independent design space into platform-dependent. Thus, they make the customization approach applicable in different contexts and in various embedded platforms.


METHODOLOGY
Input:
Application Constraints
Hardware Constraints
Remove non-applicable options from the design space
STEP 1: Design space exploration
Exploration for all customized streaming aggregation implementations
step 1 output:
Throughput, latency, energy, scalability for each customized implementation
Throughput vs. memory size
Latency vs. energy consumption
Scalability
...
STEP 2: Identification of Pareto efficient implementations
Methodology output
Customized streaming aggregation implementation
EXAMPLE
Metric1 vs. Metric2
7.45
7.4
7.35
7.3
7.25
Metric1
40
60
80
100
Metric2
P1
P2
P3
P4

Implementations evaluated:	Metric1	Metric2
Q160: A1(loc), A2(loc), ..., B4(b.w.)	7.4164	...
Q160: A1(loc), A2(loc), ..., B4(p.s.)	7.4159	...
Q320: A1(loc), A2(loc), ..., B4(b.w.)	7.3562	...
Q320: A1(loc), A2(loc), ..., B4(p.s.)	7.3567	...
Q640: A1(loc), A2(loc), ..., B4(p.s.)	7.3365	...
...	7.3336	...
	...	

Figure 6.4: Customization methodology.

After the pruning, valid customized streaming aggregation implementations are instantiated from the remaining decision tree leaves of the design space. In other words, the implementations that will finally be explored are the ones that are produced by combining the remaining leaves to create consistent implementations. Each one of these combinations is a valid customized solution that should be evaluated. All combinations of the remaining tree leaves are evaluated by brute-force exploration.

6.4.2 Methodology description

The exploration methodology consists of two steps and it is presented in Figure 6.4. The inputs of the methodology are the application and hardware constraints. The output is a streaming aggregation implementation with customized software and hardware parameters.

The first step of the methodology aims at the pruning of the design space and the implementation of the design space exploration. First, the non-applicable options are removed from the design space due to the application and hardware constraints. Then, the streaming aggregation is executed once for each different combination of the decision tree leaves of the design space. For each customization, throughput, latency, memory size, and energy consumption results are gathered. Scalability is another metric that can be evaluated, in case there is a relatively large number of PEs available. In the second step, the Pareto efficient implementations are identified. The trade-offs that can be performed by customization of the streaming aggregation on an embedded platform are presented

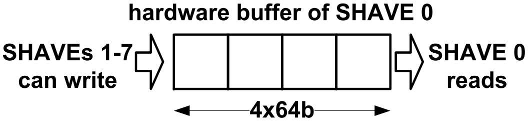


Figure 6.5: Myriad1 hardware buffers.

in the form of Pareto curves. Developers can select the implementation that is most efficient according to the optimization target.

The tool flow that supports the methodology consists of a set of bash shell scripts that handle the first step of the methodology. For the second phase, the design space pruning and the exploration are performed automatically, provided that the hardware constraints are set manually. All performance results are collected automatically. However, power (which is used to calculate energy consumption) is measured manually, since it is usually based on platform-specific hardware instrumentation. Also, the tool flow integrates a script that calculates the Pareto curve for each requested pair of metrics.

Finally, it is important to state that most design options are normally provided as functions, macros, or compiler directives from either the platform SDK, or from the POSIX/OpenCL libraries. Therefore, it should not require significant programming effort by developers to switch between the design options presented in Figure 6.3. Although the number of available implementations in some cases is increased, the systematic methodology we propose guarantees that all Pareto efficient implementations can be identified.

6.5 Demonstration of the Methodology

In this Section we first provide a short description of the embedded architectures that we used for demonstration of the methodology. Then, we present the experimental setup and the evaluation results, which are discussed in the last subsection.

6.5.1 Platforms description

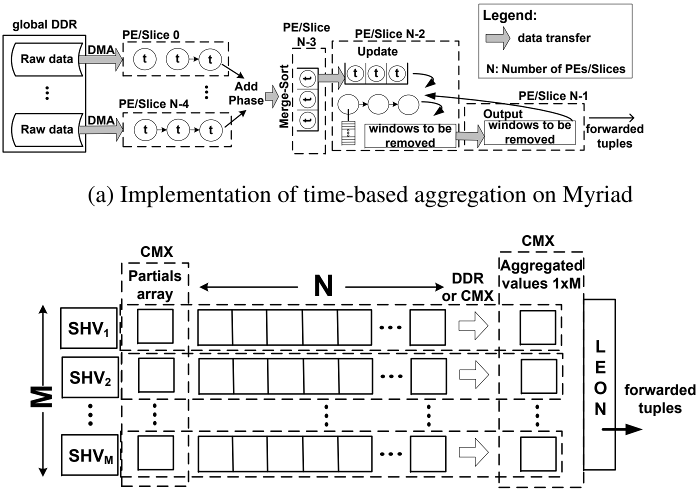
Myriad embedded processors are designed by Movidius Ltd. [17]. They target computer vision and data streaming applications. Myriad architectures are utilized in the context of Project Tango, which aims at the design of mobile devices capable of creating a 3D model of the environment around them [18]. They belong to the family of low power mobile processors and provide increased performance per watt [6].

Myriad1 architecture is designed at 65nm. It integrates 8 VLIW processing cores named Streaming Hybrid Architecture Vector Engine (SHAVEs) operating at 180MHz and a LEON3 processor that controls the data flow, handles interrupts, *etc.* More technical information about Myriad1 can be found in [19]. A local DMA engine is available for each SHAVE. Additionally, Myriad1 provides a set of hardware buffers for direct communication between the SHAVE cores. Each SHAVE has its own hardware buffer and they are accessed in FIFO manner. The size of each one is 4x64 bit words. As shown in Figure 6.5, each SHAVE can push data into the buffer of any other SHAVE and it can read data only from its own buffer. A SHAVE writes to the tail of another buffer and the owner of the buffer can read from the head. An interesting feature of the Myriad1 hardware buffers is the fact that when a SHAVE tries to write to a full FIFO or read from its own FIFO that happens to be empty, it stalls and enters a low energy mode. We take advantage of this, in order to propose energy efficient streaming aggregation implementations on Myriad1 platform.

Myriad2 is designed at 28nm [20]. In contrast to Myriad1, Myriad2 integrates 12 SHAVE cores operating at 504MHz, along with two independent LEON4 processors: LEON-RT targeting job management and LEON-OS suitable for running RTEMS/Linux, *etc.* Myriad2 provides a single top-level DMA engine and the hardware buffers size is 16x64 words.

Regarding the memory specifications, Myriad1 provides 1MB local memory with unified address space that it is named Connection Matrix (CMX). Each 128KB are directly linked to each SHAVE processor providing local storage for data and instruction code. Therefore, the CMX memory can be seen as a group of 8 memory ”slices”, with each slice being connected to each one of the 8 SHAVEs. Each SHAVE accesses its own CMX slice more efficiently in comparison with the rest CMX slices. Myriad2 CMX memory is 2MB and each slice is 128KB. Also, Myriad2 provides 1KB L1 and 256KB L2 cache. Finally, both platforms provide a global DDR memory of 64MB.

Concerning the memory allocation of the time-based streaming aggregation data structures, the incoming streams of raw data (produced by sensors, cameras, *etc.*) are placed in DDR memory. Each input queue is handled by a different SHAVE and it is placed in its local slice. Each SHAVE that handles an input queue fetches chunks of raw data in its own memory slice, by using DMA transfers. Then, it converts the raw data into tuples and stores them in its own input queue. The windows are stored in a linked list data structure, which is allocated in the CMX slice of the SHAVE core that handles the Update phase. Memory allocation and other implementation details are displayed in Figure 6.6a. Regarding the count-based aggregation scenario that uses a *M*x*N* window, each one of the *M* SHAVEs continuously fetches raw data that correspond to *N* tuples



(a) Implementation of time-based aggregation on Myriad

(b) Implementation of count-based aggregation on Myriad.

Figure 6.6: Implementation of time-based and count-based streaming aggregation on Myriad.

from DDR to CMX. However, if *N* is very large and tuples cannot be stored and processed in CMX, they are placed and aggregated in DDR. Each SHAVE computes the aggregated value of *N* tuples and forwards the result to LEON, which produces the output tuple that corresponds to the specific window. The implementation diagram in Figure 6.6b.

Freescale I.MX 6 Quad integrates four ARM Cortex A9 cores that operate at 1GHz [21]. It belongs to a family of multicore ARM-based platforms that target single board computers and run Linux-based OS. It provides 1GB RAM and two cache memory levels. On I.MX.6 the raw data are placed in data files. Chunks of raw data are fetched in RAM using *fread()* function. Then, tuples are created and placed in the input queues to be forwarded to the subsequent streaming aggregation phases.

Exynos 5 octa is an ARM-based platform that targets mobile computers. It is designed at 28nm by SAMSUNG and it is based on big.LITTLE architecture [22]. It integrates two ARM clusters: 4 Cortex-A15 and 4 Cortex-A7 cores. Exynos 5 integrates a PowerVR SGX544 GPU that supports OpenCL1.1. It includes 3 processing cores running at 533MHz. The evaluation board integrating Exynos is the Odroid-XU that provides 2GB DDR3 RAM [23]. In the context of this work, we used PowerVR GPU to perform aggregation in the count-based streaming scenario, implemented in OpenCL.

6.5.2 Experimental Setup

The dataset we used to demonstrate the proposed methodology has been collected from the online audio distribution platform SoundCloud [24]. It consists of a subset of approximately 40,000 users that exchanged about 250,000 comments between 2007 and 2013. The incoming tuples contain the following attributes: *timestamp*, *user_id*, *song_id* and *comment*. The aggregation function forwards the id of the user with the largest number of comments in each window. In the time-based aggregation scenario the window is sliding, while in the count-based, the window is tumbling, so the aggregated value is calculated over the last MxN tuples.

Table 6.2: Hardware constraints for Myriad1, Myriad2, I.MX.6 Quad and Exynos for both scenarios.

	Time-based aggregation			Count-based aggregation		
	Myriad1	Myriad2	I.MX.6	Myriad1	Myriad2	Exynos
windowing	time	time	time	count	count	count
programming	bare metal	bare metal	pthread	bare metal	bare metal	OpenCL
cache config.	no	yes	no	no	yes	no
memory local/global	yes	yes	no	yes	yes	yes

The aggregation operator is implemented entirely in C. Throughput is measured as tuples processed per second, while latency as the timestamp difference between an output tuple with the aggregated value and the latest input tuple that produced it. The energy consumption results on I.MX.6 were obtained based on hardware instrumentation using a Watts Up PRO meter device and following a setup similar to methods proposed in the literature [25, 26]. In Myriad2 power was measured though the MV198 power measurement board integrated on Myriad2 evaluation board. In Myriad1 power was estimated, based on moviSim simulator provided by Movidius MDK. In Exynos it is measured based on power sensors that are provided by Odroid-XU-e evaluation board [23]. All the values presented are the average of 10 executions, by elimination of the outliers. Each single experiment is executed from 30 seconds up to one minute.

The time-based aggregation scenario, which is actually a pipeline, is demonstrated in Myriad and I.MX.6 Quad platforms. The count-based scenario, that provides increased data parallelism, is demonstrated in Myriad and in Exynos embedded GPU. As stated earlier, Myriad1 provides 8 PEs. In time-based aggregation, each one of the merge-sort, update and output phases is assigned to a single PE. Each one of the remaining 5 PEs handles a single input queue. In Myriad2, which integrates 12 PEs, the input queues are 9. In I.MX.6 Quad that provides 4 PEs, we assigned each phase on single PE and the remaining PE handles 5 input queues.

The hardware constraints of the evaluation boards are presented in Table 6.2. The experiments we performed are the following: In the time-based aggregation scenario, in I.MX.6 we implemented the methodology using a single window configuration. However, for Myriad1 and Myriad2, we present results for two different scenarios: in the first one the window configuration (*i.e.,* the window *size* and *advance* values) are set, so that the maximum memory size of the windows data structure is small enough to fit in the local memory. In the second experiment, the windows data structure can only fit in the global memory. Thus, we study how the memory allocation of the windows data structure affects the evaluation metrics. In the count-based scenario, the aggregation is performed in parallel by the accelerator of each platform: The SHAVEs in Myriad and the GPU in Exynos.

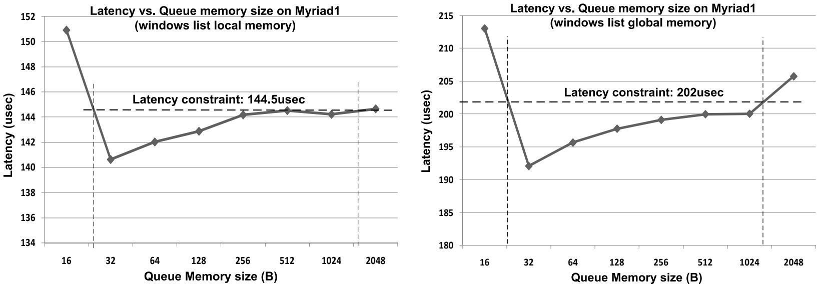
The output of the methodology is a set of Pareto points for throughput vs. memory size and latency vs. energy consumption. In time-based scenario, we present results for scalability for Myriad1 and Myriad2. The implementations that are evaluated for scalability are the ones that were found to be Pareto efficient in latency vs. energy consumption evaluation.

6.5.3 Time-based aggregation results

In the time-based scenario, we evaluate each implementation for a number of queue sizes. The queue sizes we select are the ones that provide latency below a fixed threshold. Therefore, we first measure latency for a range queue sizes and select the size values which provide latency below the threshold. Then, we proceed to the implementation of the methodology. 48 implementations are evaluated in Myriad and 4 in I.MX.6 Quad. The number of implementations that are evaluated can be reduced by selecting a smaller number of queue size values. (However, in this case fewer Pareto points may be identified).

(A) Demonstration on Myriad1

In the first experiment in Myriad1 the window size and advance values are configured so that the windows data structure can fit in the local memory. Assuming latency constraint of 144.5usec, the range of queue sizes that we evaluate are from 32B to 1024B (Figure 6.7a).



(a) Windows list in local mem.

(b) Windows list in global mem.

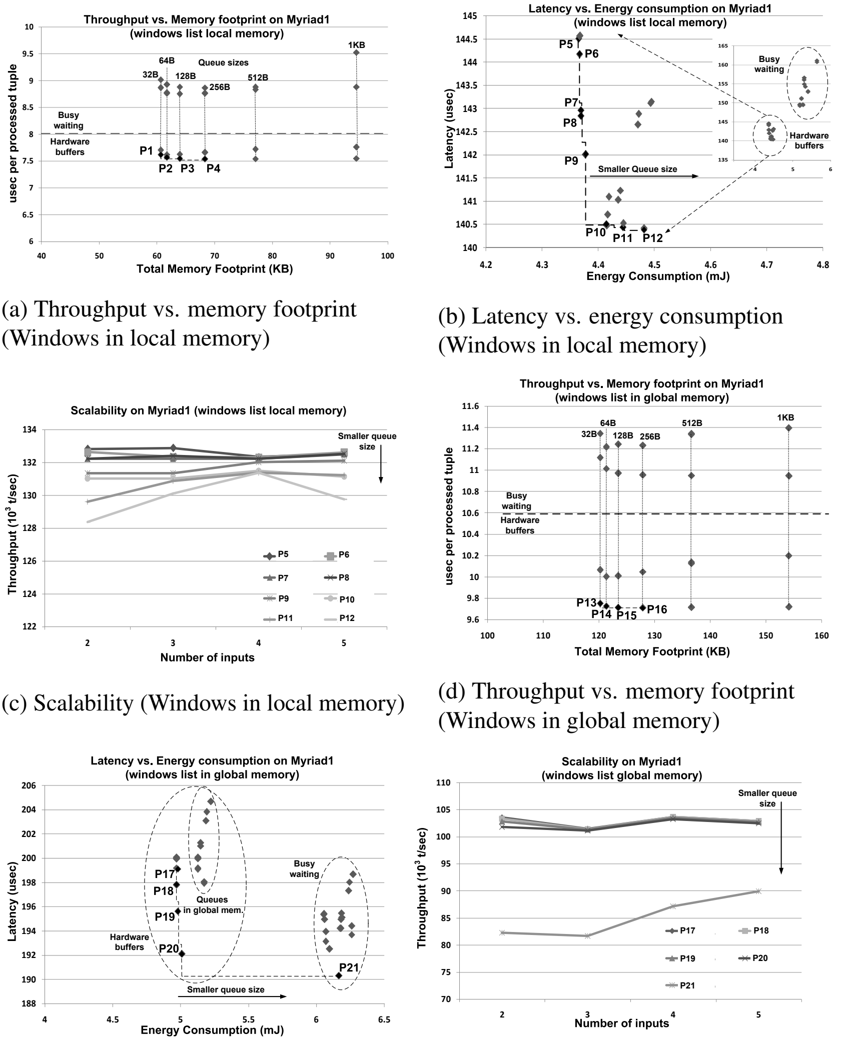
Figure 6.7: Latency vs. Queue size on Myriad1.

Table 6.3: Myriad1 Pareto efficient points description. B4(p.s.) (*i.e.,* platform specific) refers to Myriad hardware buffers.

Pareto	Description	Pareto	Description	Pareto	Description
P1	A1(l), A2(l), A3(32B), A5(fl), B2(yes), B4(p.s.)	P8	A1(l), A2(l), A3(128B), A5(fl), B2(yes), B4(b.w.)	P15	A1(l), A2(l), A3(128B), A5(fl), B2(yes), B4(b.w.)
P2	A1(l), A2(l), A3(64B), A5(fl), B2(yes), B4(p.s.)	P9	A1(l), A2(l), A3(64B), A4(fl), B1(yes), B4(p.s.)	P16	A1(on), A2(on), A3(256B), A5(fl), B2(yes), B4(p.s.)
P3	A1(l), A2(l), A3(128B), A5(fl), B2(yes), B4(p.s.)	P10	A1(l), A2(l), A3(64B), A5(fl), B2(yes), B4(p.s.)	P17	A1(l), A2(l), A3(256B), A5(fl), B1(yes), B4(p.s.)
P4	A1(l), A2(l), A3(256B), A5(fl), B2(yes), B4(p.s.)	P11	A1(l), A2(l), A3(64B), A5(fl), B1(yes), B4(p.s.)	P18	A1(l), A2(l), A3(128B), A5(fl), B2(yes), B4(p.s.)
P5	A1(l), A2(l), A3(512B), A5(fl), B2(yes), B4(p.s.)	P12	A1(l), A2(l), A3(32B), A5(fl), B2(yes), B4(p.s.)	P19	A1(l), A2(l), A3(64B), A5(fl), B2(yes), B4(p.s.)
P6	A1(l), A2(l), A3(256B), A5(fl), B2(yes), B4(p.s.)	P13	A1(l), A2(l), A3(32B), A5(fl), B2(yes), B4(p.s.)	P20	A1(l), A2(l), A3(32B), A5(fl), B2(yes), B4(p.s.)
P7	A1(l), A2(l), A3(128B), A5(fl), B1(yes), B4(p.s.)	P14	A1(l), A2(l), A3(64B), A5(fl), B2(yes), B4(p.s.)	P21	A1(l), A2(l), A3(32B), A5(fl), B2(yes), B4(b.w.)

The results for throughput vs. memory evaluation are displayed in Figure 6.8a. We notice that the Pareto points can be divided in two categories: The ones with performance lower than 8.0usec/tuple that correspond to implementations that utilize busy waiting and the rest ones that utilize the Myriad hardware buffers. (In both axes, the lower the values, the higher the efficiency). 4 Pareto efficient points are identified, which are described in Table 6.3. All Pareto efficient customized implementations can be used to perform trade-offs between throughput and memory: throughput can increase up to 1.02% and maximum memory size can drop up to 11.2% by selecting P4 and P1 solutions respectively.

Pareto points of latency vs. energy can be grouped into the same categories: The ones that exploit busy waiting and the rest that utilize hardware buffers. The later are more efficient both in terms of latency and energy consumption. 8



(a) Throughput vs. memory footprint (Windows in local memory)

(b) Latency vs. energy consumption (Windows in local memory)

(c) Scalability (Windows in local memory)

(d) Throughput vs. memory footprint (Windows in global memory)

(e) Latency vs. energy consumption (Windows in global memory)

(f) Scalability (Windows in global memory)

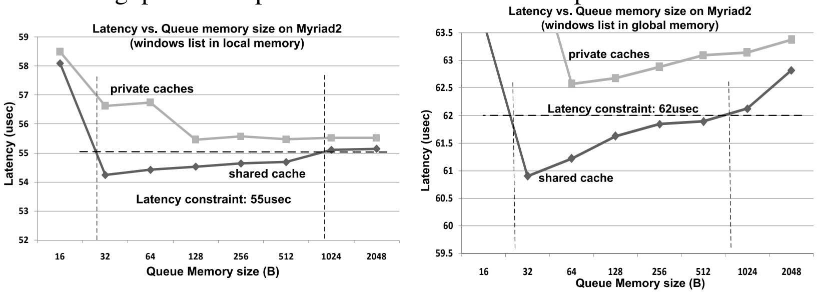
Figure 6.8: Evaluation of time-based streaming aggregation implementations on Myriad1.

Pareto points can be identified that can be used to perform trade-offs between the aforementioned metrics: up to 2.85% lower latency (P12) and up to 2.6%

lower energy consumption (P5).

Finally, scalability evaluation of the Pareto points of latency vs. energy is shown in Figure 6.8c. Throughput remains almost constant for all implementations or increases with the number of inputs. The only exception is P12, in which the queues have very small size (32B).

In the second experiment, we assume latency threshold to be 202usec (Figure 6.7b). We notice in both Figure 6.8d and Figure 6.8e that throughput is lower and latency higher in comparison with the previous experiment, since in this one the windows are placed in the global memory. The Pareto efficient points demonstrated in Figure 6.8d can be used to perform trade-offs between throughput and memory size (up to 0.5% for throughput by selecting P16 and up to 5.9% in memory size by selecting P13). In Figure 6.8e, we notice that Pareto point P21 is the most efficient in terms of latency (4.45% lower in comparison with P17), while P17 implementation is the most energy efficient (19.3% lower consumption than P21). In the scalability evaluation of Figure 6.8f, it is shown that all implementations provide high throughput that it is affected by the number of inputs only slightly, apart from P21 that utilizes busy-waiting and yields much lower throughput in comparison with the rest of the implementations.



(a) Windows list in local mem..

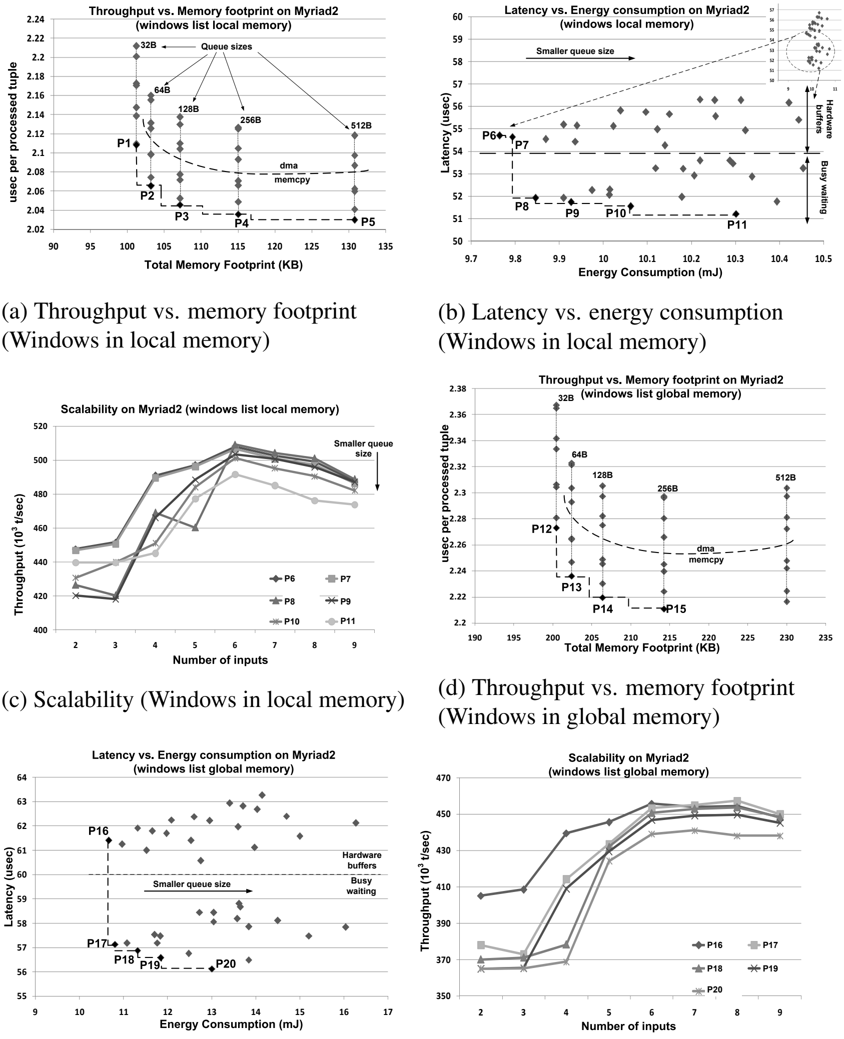
(b) Windows list in global mem..

Figure 6.9: Latency vs. Queue size on Myriad2.

(B) Demonstration on Myriad2

Figure 6.9a and Figure 6.9b show latency vs. queue sizes on Myriad2 for two different cache configurations, shared and private (decision tree *A4* in Figure 6.3). We notice that shared cache provides lower latency than private in both cases, up to 4.2%. Therefore, all implementations that utilize private cache are pruned and they are not evaluated in step 1 of the methodology.

In the first experiment in Myriad2, the windows data structure is placed in the local memory. Latency constraint is assumed to be at 55usec and therefore



(a) Throughput vs. memory footprint (Windows in local memory)

(b) Latency vs. energy consumption (Windows in local memory)

(c) Scalability (Windows in local memory)

(d) Throughput vs. memory footprint (Windows in global memory)

(e) Latency vs. energy consumption (Windows in global memory)

(f) Scalability (Windows in global memory)

Figure 6.10: Evaluation of time-based streaming aggregation implementations on Myriad2.

queue sizes from 32B to 512B will be evaluated (Figure 6.9a).

Throughput vs. memory footprint results of the methodology are shown in

Table 6.4: Myriad2 Pareto efficient points description. B4(p.s.) (*i.e.,* platform specific) refers to Myriad hardware buffers.

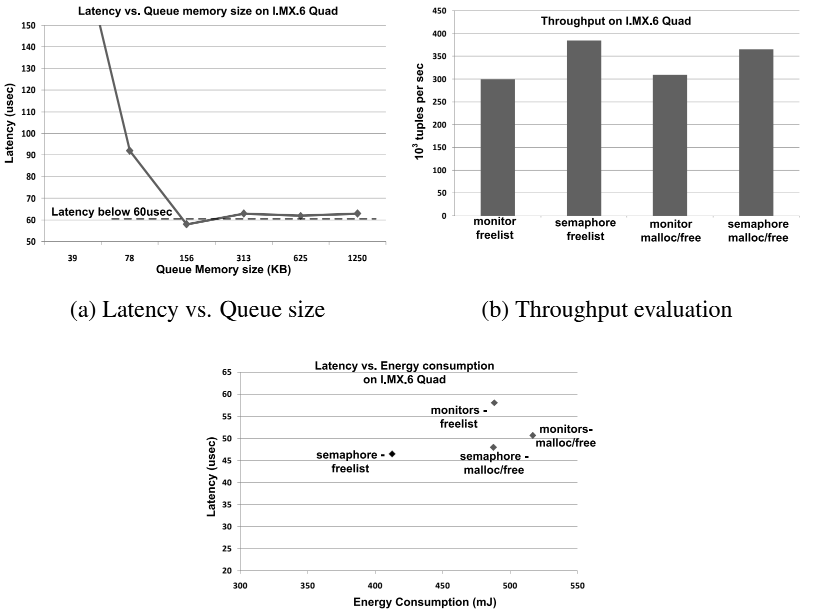
Par.	Description	Par.	Description	Par.	Description
P1	A1(l), A2(l), A3(32B), A4(s), A5(fl), B2(y), B4(b.w.)	P8	A1(l), A2(l), A3(512B), A4(s), A5(fl), B2(y), B4(b.w.)	P15	A1(l), A2(l), A3(256B), A4(s), A5(fl), B2(y), B4(p.s.)
P2	A1(l), A2(l), A3(64B), A4(s), A5(fl), B2(yes), B4(p.s.)	P9	A1(l), A2(l), A3(128B), A4(s), A5(fl), B1(y), B4(b.w.)	P16	A1(l), A2(l), A3(256B), A4(s), A5(fl), B2(y), B4(p.s.)
P3	A1(l), A2(l), A3(128B), A4(s), A5(fl), B2(y), B4(p.s.)	P10	A1(l), A2(l), A3(64B), A4(s), A5(fl), B2(y), B4(b.w.)	P17	A1(l), A2(l), A3(512B), A4(s), A5(fl), B1(y), B4(b.w.)
P4	A1(l), A2(l), A3(256B), A4(s), A5(fl), B2(y), B4(p.s.)	P11	A1(l), A2(l), A3(32B), A4(s), A5(fl), B1(y), B4(b.w.)	P18	A1(l), A2(l), A3(128B), A4(s), A5(fl), B2(y), B4(b.w.)
P5	A1(l), A2(l), A3(512B), A4(s), A5(fl), B2(y), B4(p.s.)	P12	A1(l), A2(l), A3(32B), A4(s), A5(fl), B2(y), B4(b.w.)	P19	A1(l), A2(l), A3(64B), A4(s), A5(fl), B2(y), B4(b.w.)
P6	A1(l), A2(l), A3(512B), A4(s), A5(fl), B2(y), B4(p.s.)	P13	A1(l), A2(l), A3(64B), A4(s), A5(fl), B2(y), B4(p.s.)	P20	A1(l), A2(l), A3(32B), A4(s), A5(fl), B2(y), B4(b.w.)
P7	A1(l), A2(l), A3(256B), A4(s), A5(fl), B1(y), B4(p.s.)	P14	A1(l), A2(l), A3(128B), A4(s), A5(fl), B2(y), B4(p.s.)		

Figure 6.10a. Implementations based on *memcpy* provide higher performance than the ones based on dma transfers between the CMX slices. The 5 Pareto efficient points that are identified provide trade-offs up to 3.7% for throughput (P5) and up to 22.5% for memory footprint (P1).

Latency vs. energy results are displayed in Figure 6.10b. The Pareto points can be grouped into 2 categories: the ones that utilize busy waiting synchronization scheme and the rest ones that are based on hardware buffers. The 6 Pareto efficient points can be used to perform trade-offs between latency and energy (up to 6.37% for latency by selecting implementation P11 and 5.2% for energy consumption, by selecting P6).

With respect to scalability in Figure 6.10c, we notice that throughput for all implementations increases up to 6 inputs and then it drops slightly. As in Myriad1 experiments, implementations with lower queue size tend to provide lower throughput.

In the second experiment, in which the windows data structure is placed in global memory due to its increased memory size, latency constraint is set to 62usec (Figure 6.9b) and throughput vs. memory footprint results are presented in Figure 6.10d. 4 Pareto efficient points have been identified that provide throughput vs. memory size trade-offs (up to 6.4% for throughput and up to 3.07% for latency). Correspondingly, the 5 Pareto efficient points in latency vs. energy consumption evaluation displayed in Figure 6.10e can be used for performing trade-offs, up to 8.59% for latency (P20) and 18% for energy (P16). Scalability results in Figure 6.10f are slightly different from the ones in the previous experiment. Implementations scale up to 8 inputs and most of them tend to provide slightly lower throughput when 9 inputs are used.



(a) Latency vs. Queue size

(b) Throughput evaluation

(c) Latency vs. energy consumption

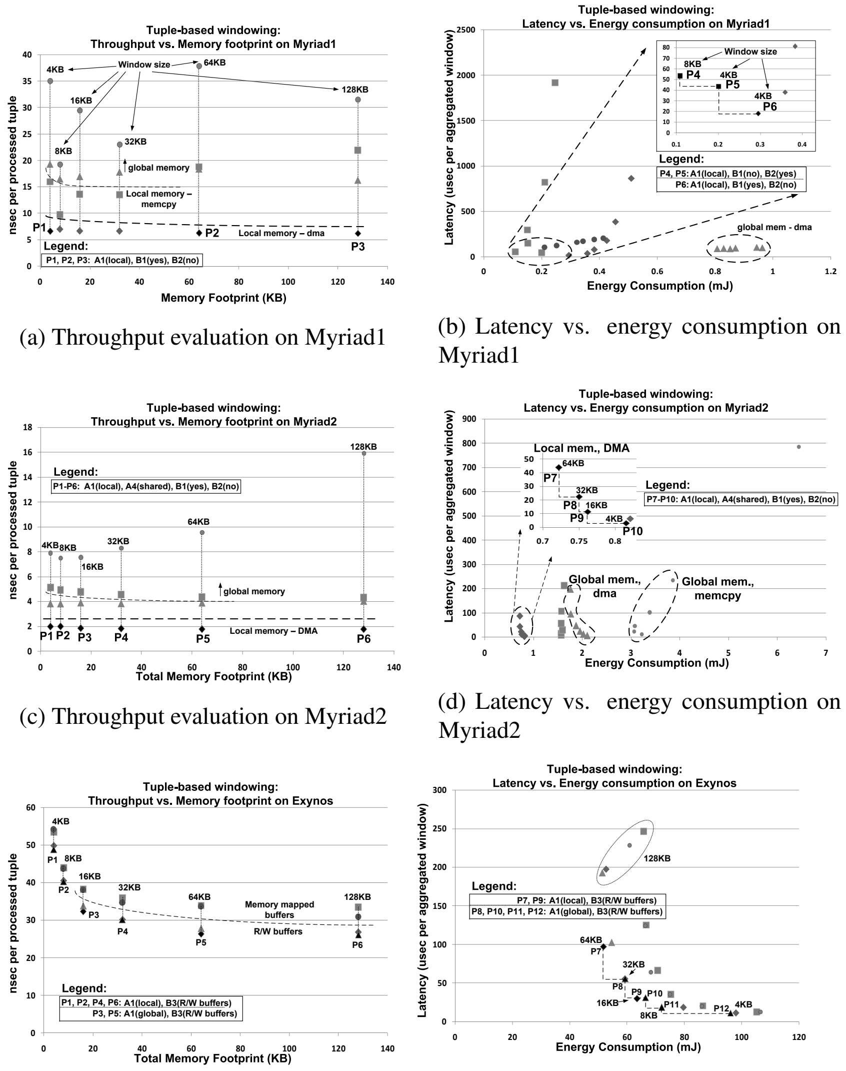
Figure 6.11: Evaluation results of time-based streaming aggregation implementations on I.MX.6 Quad.

(C) Demonstration on I.MX.6 Quad

Few customized implementations exist for I.MX.6, since the operating system handles many design options. In the I.MX.6 Quad experiment latency threshold has been set to 60usec and a single effective queue size has been found: 156KB (Figure 6.11a). 4 customized implementations have been evaluated and throughput results are shown in Figure 6.11b, while latency vs. energy results are displayed in Figure 6.11c. We notice that the most efficient implementation in terms of both throughput, latency and energy is the one that utilizes semaphores for synchronization, along with freelist-based memory management.

6.5.4 Count-based aggregation results

In the count-based scenario, we evaluate each implementation for different window sizes. The selected values are provided to the first step of the methodology. 24 different implementations are evaluated in each platform.



(a) Throughput evaluation on Myriad1

(b) Latency vs. energy consumption on Myriad1

(c) Throughput evaluation on Myriad2

(d) Latency vs. energy consumption on Myriad2

(e) Throughput evaluation on Exynos

(f) Latency vs. energy consumption on Exynos

Figure 6.12: Evaluation results of count-based streaming aggregation implementations.

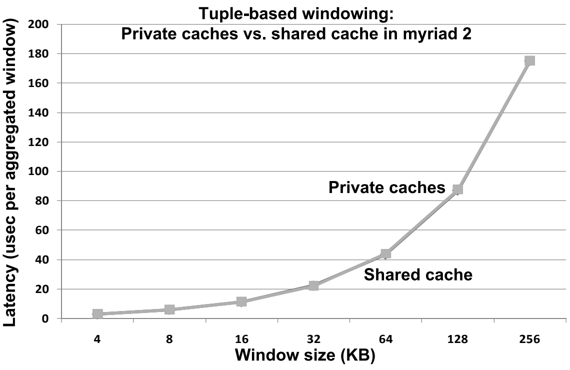


Figure 6.13: Latency vs. window size on Myriad2 for count-based streaming aggregation.

(A) Demonstration on Myriad1

Figure 6.12a shows throughput vs. memory footprint on Myriad1. Implementations that process tuples in local memory and transfer data from global to local memory through DMA provide higher throughput. For instance, at 4KB window size, P1 provides 58% higher throughput than the implementation that uses *memcpy* for data transfer.

Latency vs. energy consumption results are presented in Figure 6.12b. We notice that smaller windows provide lower latency. Also, transferring tuples in local memories provides lower latency than processing windows in Myriad1 global memory. 3 Pareto points are identified that provide trade-offs between latency and energy consumption.

(B) Demonstration on Myriad2

In Myriad2, we first evaluate latency vs. window size for two different cache configurations. As shown in Figure 6.13, utilization of shared cache provides slightly lower latency than private caches (less than 1%). Therefore, implementations that utilize private caches are pruned and the design space is reduced.

As in Myriad1, implementations that provide higher throughput are the ones in which tuples are transferred through DMA and processed in local memory. Figure 6.12c shows that throughput increases up to 59% using the aforementioned implementation, in comparison with the implementation in which tuples are processed in global memory, with window size 64KB. Also, we notice that larger windows provide slightly higher throughput. For instance, increasing window size from 4KB to 128KB, yields throughput increase about 10% (P1 to

P6).

Implementations that utilize local memory and DMA transfers provide both low latency and energy efficiency, as shown in Figure 6.12d. Processing in global or in local memory affects both latency and energy consumption results. For instance, tuples in local memory and utilization of DMA with 4KB window size provides 31.4% lower energy consumption than the corresponding implementation with tuple processing in global memory.

(C) Demonstration on Exynos 5

Throughput vs. memory footprint results are displayed in Figure 6.12e. Larger window sizes provide higher throughput. Implementations that utilize R/W buffers yield higher performance than corresponding implementations with memory mapped data buffers: up to 21% for 64KB window size.

Regarding latency vs. energy consumption, displayed in Figure 6.12f, 6 Pareto points are identified. Smaller window sizes provide lower latency, but higher energy consumption, due to the increased rate of data transfers. Utilization of R/W buffers is more efficient than memory mapped ones, both in terms of latency and energy consumption. Due to the relatively small buffer size, the overhead of utilizing R/W buffers is also small.

6.5.5 Performance per watt evaluation

One of the goals of this work is to compare performance per watt of streaming aggregation mapped on low power embedded platforms with the corresponding results on an HPC CPU and a GPGPU. In this subsection, we first provide details on the implementation of the operator on the aforementioned platforms and then we present the evaluation results.

We implemented the time-based streaming aggregation scenario on an Intel Xeon E5 CPU with 8 cores operating at 3.4GHz, with 16GB RAM, running Ubuntu Linux 12.04. Compiler is gcc v.4.9.2 and optimization flag is *-O3*. Power consumption was measured through hardware instrumentation and refers to dynamic CPU Power. Throughput and latency were measured similarly to the embedded implementations. Data transfer was based on *memcpy()* operations and synchronization based on semaphores.

The results are presented on Table 6.5. The values for Myriad1, Myriad2 and I.MX.6 correspond to the implementation that provides the best results for each specific metric. To ensure fair comparison, all values for all platforms utilize 5 input queues. Performance per watt is calculated as number of tuples forwarded per second, per watt.

Table 6.5: Time-based streaming aggregation: Comparison between latency, throughput and performance per watt on embedded and Intel Xeon architectures.

	Latency (usec)	Throughput (t/sec)	(t/sec)/watt
Myriad1	140.38	132,622	379,041
Myriad2	39.8	497,154	1,004,766
I.MX.6	58	384,952	446,787
Xeon	15	1,105,221	18,412

In Table 6.5, we notice that in terms of performance, latency on Intel Xeon is 62.3% lower than in Myriad2, while it is 3.8 and 9.3 times lower than in I.MX.6 and Myriad1, respectively. In terms of throughput, Xeon provides more than two times higher throughput than Myriad2, 2.8 than I.MX.6 and 8.3 times higher than Myriad1. The high performance of Intel Xeon is related with the higher computational power it provides and the fact that it operates in much higher frequency than the embedded architectures. However, in terms of performance per watt, embedded platforms outperform Intel Xeon. Since they are very low power, they achieve higher performance watt: 54 times higher in Myriad2, while in Myriad1 it is 20 times higher. Finally, I.MX.6 provides 24 times higher performance per watt in comparison with Intel Xeon.

Count-based aggregation scenario was implemented in OpenCL 1.1 and evaluated in AMD Radeon HD 6450 general purpose GPU [27]. Host run Ubuntu Linux 12.04 with gcc v.4.9.2. Throughput and latency were measured similarly to the corresponding embedded implementations, while power consumption is estimated based on GPU's specifications. Device data accessing is based on R/W buffers.

The results are presented in Table 6.6. Embedded platforms provide lower throughput and higher latency than Radeon GPGPU. However, both Myriad boards yield higher performance per watt than GPGPU, due to the very low power that they require. More specifically, Myriad2 provides about 14 higher performance per watt, while Myriad1 7 times.

6.5.6 Discussion of Experimental Results

In this subsection we summarize the conclusions we draw from the demonstration of the methodology that is presented in the previous subsections. The trade-offs we demonstrated in the experimental results can be used to draw conclusions about the relation between the customization options and the evaluation metrics.

Table 6.6: Count-based streaming aggregation: Comparison between latency, throughput and performance per watt on embedded and Radeon HD 6450.

	Latency (usec)	Throughput (Mt/sec)	(Mt/sec)/watt
Myriad1	17.98	151.8	593
Myriad2	3.04	505.4	1286
Exynos	7.5	47.4	7,93
GPGPU	1.94	2576.3	85.87

(A) Time-based streaming aggregation conclusions

Observation 1: Streaming aggregation should be customized differently, not only between I.MX.6 Quad and Myriad architectures, but also between Myriad1 and Myriad2.

For example, in Myriad1, in the first experiment, in the implementation that provides the lowest latency, data transfer is based on hardware buffers. On the contrary, in Myriad2 it is based on busy waiting mechanism. In the implementation that provides the highest throughput, the queue is 256B in Myriad1, while it is 512B in Myriad2.

Observation 2: There is a threshold in the queue size, below which latency is very high. Very large queue sizes may also negatively affect latency.

We notice that in both Myriad and I.MX.6, latency is very high for small queue sizes, which is due to the high overhead of constantly fetching data for refiling the queues with new tuples. In these cases, the thread that executes the merge-sort phase, often finds the queues to be empty. As the queue size increases latency drops drastically. However, in Myriad1 and Myriad2 experiments, we notice that as the queue size increases, latency tends to increase, as well (Figure 6.7 and Figure 6.9). The reason is the fact that the larger the queue, the more cycles it takes to complete a DMA transfer of data from the DDR to the local memory and start refilling the queue with new tuples. Thus, the tuples that entered the update phase before a new DMA transfer and exit the output phase after it, they have higher latency than the rest ones. In contrast with Myriad, on I.MX.6 we can use much bigger queues, since the available memory is much larger. However, after a specific queue size, throughput and latency on I.MX.6 do not seem to be significantly affected any more (Figure 6.11a).

Observation 3: Throughput is mainly affected by either the data transfer mechanism (in Myriad2) or by the signaling mechanism (in Myriad1).

In general, in Myriad1 and Myriad2, throughput drops when the queue size becomes smaller, due to the overhead of the DMA transfers, which is added more frequently when the queues are small (*e.g.,* Figure 6.8a and Figure 6.10a). However, latency becomes lower in that case, as stated earlier. In Myriad2, throughput is mainly determined by whether *memcpy* or DMA data transfer mechanism is used. Indeed, data transfer options seem to have major impact on throughput (Figure 6.10a and Figure 6.10d). On the other hand, in Myriad1 the utilization of hardware buffer or of busy waiting scheme is the dominant factor that affects throughput (Figure 6.8a and Figure 6.8d). In Myriad2 signaling design options have much lower impact in comparison with data transfer options. On the contrary, in Myriad1, data transfer mechanism has relatively small effect on throughput in comparison with the signaling mechanism (*memcpy* however is slightly more efficient). In I.MX.6, the utilization of freelists to avoid the frequent system calls improves throughput and latency results. However, the main factor that improves performance is the utilization of semaphores instead of monitors (Figure 6.11b).

Observation 4: Latency is affected by the synchronization mechanism. Different mechanism should be used in Myriad1 than in Myriad2.

The synchronization mechanism is the main design option that affects latency and energy in both Myriad architectures. Busy waiting mechanism provides lower latency in Myriad2 and slightly lower energy consumption. On the contrary, the utilization of hardware buffers in Myriad1 is more efficient it terms of latency. The data transfer mechanism has much lower impact in both architectures in terms of latency and energy.

Observation 5: The frequency by which data movements are performed from global to local memory affects energy consumption in Myriad. We notice that larger queue sizes are more energy efficient in both Myriad1 and Myriad2, due to the lower rate by which data are fetched in the local memory (*e.g.,* Figure 6.8b and Figure 6.10b). On I.MX.6 Quad, energy consumption is determined mainly the by synchronization scheme that it is used.

Finally, an interesting observation is the fact that the memory allocation of the input queues affects neither the performance nor the energy consumption in Myriad significantly. The reason is the fact that both Myriad architectures provide cache memory and the rate of cache misses for accessing the queues by the PE that performs the merge-sort operation is relatively small. On the other hand, the allocation of the windows data structure in global memory has major impact in both performance and energy consumption. For instance, in Myriad2, by allocating the windows data structure in global memory, latency increases about 9%, throughput drops by 7% and energy consumption increases by 20% in comparison with the allocation in local memory.

The above observations can be used to draw more general conclusions on how the streaming aggregation should be customized on embedded platforms. When the optimization target is performance, the following considerations should be taken into account:

- The queue size should be large enough to decrease the rate by which data transfers are instructed. Frequent small data transfers lower performance. However, for implementations that are very sensitive to latency, it should be noted that too large queue sizes may increase latency.
- Window *size* and *advance* values affect a lot the maximum size of the windows data structure and therefore the memory allocation design options and the performance. Platforms with very small local memory may be not suitable for implementing streaming aggregation, since they would limit the window configuration values that can be used, if allocation of the data structure in global memory is not a option, due to very strict performance requirements.
- Platform-specific options for efficient communication between cores (such as the hardware buffers on Myriad) should be evaluated, when the streaming aggregation is implemented at low level. In some cases (such as in Myriad1) they can provide increased performance.

On the other hand, if the main goal is energy efficiency, the following issues should be considered:

- The queues should be as large as possible to avoid the energy consumption overhead of frequent small data transfers.
- For window *size* and *advance* values apply the same that are stated earlier: Window configuration that forces the allocation of the windows data structure in global memory has negative impact in energy consumption.
- Finally, developers should try to evaluate features that set the PEs in a low-energy mode when they are forced to wait (such as the hardware buffers in Myriad1).

(B) Count-based streaming aggregation conclusions

Observation 1: Both throughput and latency in Myriad implementations are affected by the memory allocation of the processed tuples. In Exynos implementations, they are mainly affected by the data accessing method by the device.

In general, throughput is apparently affected by the window size. Apart from that, design choices such as the allocation of the window in local memory and R/W buffers in OpenCL implementations, yield increased throughput.

In contrast with throughput, smaller window sizes provide lower latency. Implementations in which tuples are processed in local memories in Myriad and utilize R/W buffers in mobile GPU provide the lowest latency.

Observation 2: Energy consumption is mainly affected by the memory allocation and the window size.

Energy consumption in Myriad is affected by both the type of memory in which tuples are processed and the size of the window (Figure 6.12d). In Exynos, window size has the highest impact in energy (Figure 6.12f). Since the rate of data transfers is increased when smaller windows are used, energy consumption is also increased.

To summarize, when the optimization target is performance, DMA transfers and R/W OpenCL buffers provide higher throughput than the rest of the design choices. Large windows yield increased throughput, while smaller ones provide low latency. Finally, window sizes that allow processing in local memory benefit both performance and energy.

The methodology we propose in this work provides a systematic approach to the efficient customization of the streaming aggregation on embedded platforms. Instead of trying to tune the application and hardware parameters arbitrary to achieve the desired results, the proposed methodology provides a set of customization solutions from which developers can select the one that is more suitable according the design constraints.

Finally, it is important to state that the methodology is not fundamentally limited to streaming aggregation. The design space could be adapted to be applicable to other streaming operators, as well (such as join, filter, *etc.*) and to embedded platforms with various other features. New attributes can be integrated in the design space for exploration as new decision trees, leaves or categories. The application and hardware constraints should be updated accordingly to retain the coherency of the customized implementations.

6.6 Conclusion

We proposed a customization methodology for the implementation of streaming aggregation in modern embedded devices. The methodology was demonstrated in 4 different embedded architectures, 2 aggregation scenarios and a real-world data set. The customized implementations provided by the methodology can be utilized by developers to perform trade-offs between several parameters, taking into consideration the design constraints that are imposed by both the application requirements and the embedded architecture. In the future, we intend to extend the design space by integrating more streaming aggregation operators and evaluate the approach in embedded platforms with various features.

Acknowledgments

This work was partially supported by EXCESS Project (www.excessproject.eu) under grant agreement 611183. The authors would like to thank Movidius Ltd. for providing us with Myriad evaluation boards in material transfer to conduct this research.

Bibliography

[1] "Appliedmicro x-gene2," in *IEEE Hot Chips 26 Symposium (HCS)*, 2014, pp. 1–24.

[2] Scott Schneidert, Henrique Andrade, Buğra Gedik, Kun-Lung Wu, and Dimitrios S Nikolopoulos, "Evaluation of streaming aggregation on parallel hardware architectures," in *Proceedings of the International Conference on Distributed Event-Based Systems*. 2010, pp. 248–257, ACM.

[3] Phillip Stanley-Marbell and Victoria Caparrós Cabezas, "Performance, power, and thermal analysis of low-power processors for scale-out systems," in *Proceedings of the International Parallel and Distributed Processing Symposium Workshops*. 2011, pp. 863–870, IEEE.

[4] Karl Fürlinger, Christof Klausecker, and Dieter Kranzlmüller, "Towards energy efficient parallel computing on consumer electronic devices," in *Information and Communication on Technology for the Fight against Global Warming*, pp. 1–9. Springer, 2011.

[5] Nikola Rajovic, Alejandro Rico, Nikola Puzovic, Chris Adeniyi-Jones, and Alex Ramirez, "Tibidabo: Making the case for an arm-based hpc system," *Future Generation Computer Systems*, vol. 36, pp. 322–334, 2014.

[6] Mircea Horea Ionica and David Gregg, "The movidius myriad architecture's potential for scientific computing," *IEEE Micro*, vol. 35, no. 1, pp. 6–14, 2015.

[7] Cagri Balkesen, Nesime Tatbul, and M. Tamer Özsu, "Adaptive input admission and management for parallel stream processing," in *Proceedings of the International Conference on Distributed Event-based Systems*. 2013, pp. 15–26, ACM.

[8] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez, "Streamcloud: An elastic and scalable data streaming

system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2351–2365, 2012.

[9] Uri Verner, Assaf Schuster, and Mark Silberstein, "Processing data streams with hard real-time constraints on heterogeneous systems," in *Proceedings of the International Conference on Supercomputing*. 2011, pp. 120–129, ACM.

[10] Daniel Cederman, Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas, "Brief announcement: Concurrent data structures for efficient streaming aggregation," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2014, pp. 76–78, ACM.

[11] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik, "The design of the borealis stream processing engine.," in *Conference on Innovative Data Systems Research*, 2005, vol. 5, pp. 277–289.

[12] Yoonseo Choi, Yuan Lin, Nathan Chong, Scott Mahlke, and Trevor Mudge, "Stream compilation for real-time embedded multicore systems," in *Proceedings of the IEEE International Symposium on Code Generation and Optimization*. 2009, pp. 210–220, IEEE Computer Society.

[13] Ryan R. Newton, Lewis D. Girod, Michael B. Craig, Samuel R. Madden, and John Gregory Morrisett, "Design and evaluation of a compiler for embedded stream programs," in *Proceedings of the ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. 2008, pp. 131–140, ACM.

[14] Tony Givargis, Frank Vahid, and Jörg Henkel, "System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip," in *Proceedings of the IEEE International Conference on Computer-aided Design*. 2001, pp. 25–30, IEEE Press.

[15] Christos Baloukas, Jose L. Risco-Martin, David Atienza, Christophe Poucet, Lazaros Papadopoulos, Stylianos Mamagkakis, Dimitrios Soudris, J. Ignacio Hidalgo, Francky Catthoor, and Juan Lanchares, "Optimization methodology of dynamic data structures based on genetic algorithms for multimedia embedded systems," *Journal of Systems and Software*, vol. 82, no. 4, pp. 590–602, 2009.

[16] Sotirios Xydis, Alexandros Bartzas, Iraklis Anagnostopoulos, Dimitrios Soudris, and Kiamal Z. Pekmestzi, "Custom multi-threaded dynamic memory management for multiprocessor system-on-chip platforms," in *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, 2010, pp. 102–109.

[17] "Movidius Ltd.:," `http://www.movidius.com`.

[18] "Project Tango:," `https://www.google.com/atap/project-tango/`.

[19] David Moloney, "1tops/w software programmable media processor," in *IEEE Hot Chips 23 Symposium (HCS)*, 2011, pp. 1–24.

[20] Brendan Barry, Cormac Brick, Fergal Connor, David Donohoe, David Moloney, Richard Richmond, Martin O'Riordan, and Vasile Toma, "Always-on vision processing unit for mobile applications," *IEEE Micro*, vol. 35, no. 2, pp. 56–66, 2015.

[21] "Freescale i.mx 6 quad application processors for industrial products data manual," Tech. Rep., Freescale Semiconductor Inc., 2014.

[22] Hongsuk Chung, Munsik Kang, and Hyun-Duk Cho, "Heterogeneous multi-processing solution of exynos 5 octa with arm® big. little™ technology," .

[23] "Hardkernel. Odroid-xu:," `http://www.hardkernel.com/`.

[24] "SoundCloud:," `https://www.soundcloud.com`.

[25] Nicholas Hunt, Paramjit Singh Sandhu, and Luis Ceze, "Characterizing the performance and energy efficiency of lock-free data structures," in *Proceedings of the Workshop on Interaction Between Compilers and Computer Architectures*. 2011, pp. 63–70, IEEE Computer Society.

[26] Karan Singh, Major Bhadauria, and Sally A McKee, "Real time power estimation and thread scheduling via performance counters," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 2, pp. 46–55, 2009.

[27] "AMD Radeon HD 6450 GPU:," `http://www.amd.com/en-us/products/graphics/desktop/6000/6450`.

PAPER VI

Ivan Walulya, Dimitris Palyvos-Giannas, Yiannis Nikolakopoulos, Vincenzo Gulisano, Marina Papatriantalou and Philippas Tsigas

Viper: A Module for Communication-Layer Determinism and Scaling in Low-Latency Stream Processing

Future Generation Computer Systems
Vol. 88, pp. 297–30, Elsevier 2018.

7

Viper: A Module for Communication-Layer Determinism and Scaling in Low-Latency Stream Processing

Abstract

Stream Processing Engines (SPEs) process continuous streams of data and produce results in a real-time fashion, typically through one-at-a-time tuple analysis. In Fog architectures, the limited resources of the edge devices, enabling close-to-the-source scalable analysis, demand for computationally- and energy-efficient SPEs. When looking into the vital SPE processing properties required from applications, determinism, which ensures consistent results independently of the way the analysis is parallelized, has a strong position besides scalability in throughput and low processing latency. SPEs scale in throughput and latency by relying on shared-nothing parallelism, deploying multiple copies of each operator to which tuples are distributed based on its semantics. The coordination of the asynchronous analysis of parallel operators required to enforce determinism is then carried out by additional dedicated sorting operators. To prevent this costly coordination from becoming a bottleneck, we introduce the Viper communication

module, which can be integrated in the SPE communication layer and boost the coordination of the parallel threads analyzing the data. Using Apache Storm and data extracted from the Linear Road benchmark and a real-world smart grid system, we show benefits in the throughput, latency and energy efficiency coming from the utilization of the Viper module.

7.1 Introduction

Data streaming emerged to meet the stringent demands of massive on-line data analysis in various contexts, such as cloud and edge-computing architectures. Stream Processing Engines (SPEs) allow programmers to formulate continuous queries, defined as Directed Acyclic Graphs of interconnected operators, to process incoming data producing results in a continuous fashion; e.g., StreamCloud [1], Apache Storm and Flink [2,3] and Saber [4].

In upcoming IoT-based cyber-physical systems, edge and fog devices can enable close-to-the-source analysis minimizing latency for time-critical applications and adding high cumulative computational power to the resources available in existing data centers. To do so, nevertheless, the limited resources of individual edge and fog devices demand for computationally and energy efficient SPEs.

Parallelism in SPEs is key for achieving *high-throughput* and *low latency* processing for large data volumes in evolving cyber-physical infrastructures [5]. The importance of scaling in throughput while keeping low-latency processing in SPEs is clearly understood, it has also been manifested by work in [1,6–9]. Pipeline and task parallelism are easily extracted from Directed Acyclic Graphs with operators or tasks assigned to different processing units. However, with data parallelization or fission [10–14], careful orchestration of operators' execution is required to preserve *determinism*, which is required to ensure consistent results independently of the way in which the analysis is parallelized. Data parallelism involves replicating instances of operators, that work concurrently on data streams. An operator's parallel implementation is *deterministic* if, given the same sequences of input tuples, the same sequence of output tuples is produced independently of the tuples' inter-arrival times or the parallelism degree of the operator [12,13,15,16].

Previous attempts to guarantee determinism in SPEs under execution of parallel instances of an operator rely on dedicated merge-sorting operators. These operators are either added to continuous queries by query compilers [1,12,13] or left for developers to place within their streaming applications in SPEs, such as Apache Storm [2]. This type of approach is henceforth referred to as *operator-layer* determinism in the paper. Minimizing the computational overhead introduced by such dedicated operators is challenging, especially for one-at-a-time, fine-grained low latency tuple processing.

We address the issue of guaranteeing determinism in a modular, automated and efficient way. We start by observing that, commonly in SPEs, each physical stream is piped from a producer (e.g., an incoming link from a sensor, or an outgoing link of an operator instance) to its consumer (another operator instance), without coordination or sharing state. Efficient synchronization over shared memory achieved transparently, is challenging but integral to providing determinism to application developers without requiring the latter to develop custom solutions.

Gulisano *et al.* [15] proposed *ScaleGate*, a data structure which is customized to guarantee determinism, and which has been used for aggregate and join operators in shared memory systems. In this paper we build upon ScaleGate and provide the following contributions: (i) We modularly shift a procedure of guaranteeing determinism, from the operator-layer to the *communication layer* of an SPE, thus relieving application developers from the burden of devising application-dependent methods. (ii) We design and implement a module, called *Viper*, which can be transparently integrated in an SPE communication layer. Building on ScaleGate, we lift the data-structure's context into the communication layer of an SPE architecture. From ScaleGate to Viper, the novelty is on the transparency provided to the application developer in efficiently guaranteeing determinism. (iii) We integrate Viper in Apache Storm (as a representative example of an SPE) and demonstrate the idea of modularly providing determinism, while caring for efficiency in parallelism, through an experimental evaluation of the proposed methodology. For the evaluation, we chose streaming operators of the Linear Road benchmark [17] and a use-case from a real-world smart grid system as representative examples of where stream processing in fog and edge architectures can be far better than processing in the cloud, as also discussed in [18]. The study clearly shows the throughput, latency and energy-efficiency benefits induced.

In the paper, we present preliminary concepts in Section 7.2; we describe our proposal for enforcing determinism at the communication layer (rather than the operator layer) of an SPE and discuss the advantages of the former as we introduce the Viper module, in Section 7.3. We show the use of Viper, as an SPE module, by using Apache Storm as a use case in Section 7.4.1 and we evaluate the benefits of Viper in Section 7.5. We discuss related work and conclude in Sections 7.6 and 7.7, respectively.

7.2 System Model

This section introduces data streaming, parallel and deterministic execution of *continuous queries* and the performance metrics to assess the results.

7.2.1 Data Streaming

A stream is defined as an unbounded sequence of tuples $t_0, t_1, \ldots$ sharing the same schema $\langle ts,\ A_1, \ldots, A_n \rangle$. Given a tuple t, $t.ts$ represents its creation timestamp while $A_1, \ldots, A_n$ are application-related attributes.

Continuous queries (henceforth simply queries) are defined as DAGs of *operators* that consume and produce tuples. Operators are distinguished into *stateless* or *stateful*, depending on whether they keep any state that evolves with the tuples being processed. Stateless operators include Map (to alter the schema of tuples) and Filter (to discard or route tuples). Stateful operators include Aggregate (to compute aggregation functions such as sum or average over tuples) and Join (to match tuples coming from multiple streams). Due to the unbounded nature of streams, stateful operations are computed over *sliding windows*, which can be time-based or tuple-based and are defined by parameters *size* and *advance*. Following the data streaming literature (e.g., [1, 19, 20]), we assume that streams fed by each data source contain timestamp-sorted tuples. If this is not the case, sorting mechanisms such as [21] can be leveraged.

The performance of an operator depends on its *cost* and *selectivity*. The cost represents the average time needed to process an input tuple and (optionally) produce any resulting output tuple. It is thus coupled with the selectivity, which represents the average number of output tuples produced upon the processing of one input tuple (e.g., an operator with selectivity 0.5 will produce, on average, one output tuple each time it processes two input tuples).

To illustrate the aforementioned terms and notions, Figure 7.1A presents a sample streaming application from the Linear Road benchmark [17]*, where position reports are forwarded by vehicles traveling on a highway. The application performs three updates for each incoming report. First, it checks if the report refers to a vehicle entering, leaving or changing segment. It then updates the number of vehicles and the tolls of the involved segments. Finally, it notifies the interested vehicles.

Figure 7.1B presents an example centralized query that implements the application's semantics through basic streaming operators. The schema of each stream is presented on top of the operators. A first Aggregate A1 enriches each position report with the previous segment observed for the same vehicle.

*Section 7.5 contains a detailed description of the benchmark.

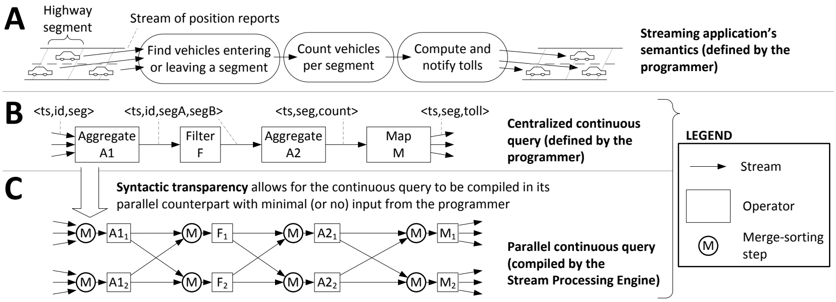

A
Highway segment
Stream of position reports
Find vehicles entering or leaving a segment
Count vehicles per segment
Compute and notify tolls
Streaming application's semantics (defined by the programmer)
B
<ts,id,seg>
<ts,id,segA,segB>
<ts,seg,count>
<ts,seg,toll>
Aggregate A1
Filter F
Aggregate A2
Map M
Centralized continuous query (defined by the programmer)
C
Syntactic transparency allows for the continuous query to be compiled in its parallel counterpart with minimal (or no) input from the programmer
M
A1₁
F₁
A2₁
M₁
A1₂
F₂
A2₂
M₂
Parallel continuous query (compiled by the Stream Processing Engine)
LEGEND
Stream
Operator
Merge-sorting step

Figure 7.1: Streaming application part of the Linear Road benchmark [17], presented together with a sample centralized continuous query implementing its semantics and its parallel counterpart.

Subsequently, a Filter F discards reports referring to vehicles that have not changed segment. Aggregate A2 updates the count for each segment and Map M computes the toll for a segment, based on the number of vehicles in it, and notifies vehicles.

7.2.2 Parallelism, determinism and syntactic transparency

A parallel version of a centralized query (such as the sequential one in the example of Figure 7.1B) is desirable because of two reasons: (i) to cope with the large and fluctuating volume of data (in this example the position reports observed in a highway); (ii) to possibly deploy the analysis over a distributed network of nodes, each responsible for e.g. a subset of segments. The latter would avoid centralized data gathering and processing, which indeed might not be feasible because of too high data transmission latency, infrastructure limitations or privacy legislation, among other reasons.

For a parallel query, deterministic execution should ensure that the results given by it are exactly the same that would be given by its centralized counterpart. This is equivalent to the notion of *external determinism* (or *determinacy*), as described in e.g., [22, 23]. As shown in [23], processing systems that are described as directed graphs of operations guarantee this property on the global level, if they satisfy determinacy (i) on the operation level and (ii) in the data flows between communicating operations. This implies that if we have a parallel implementation of the query using deterministic processing components and deterministic flow of the results to downstream operators, then the issue is addressed. As argued in [1, 24], in the context of query processing in SPEs, for having global determinism it is sufficient to enforce that (i) splitting streams to downstream operator is done according to the semantics of those operators (e.g. for aggregates, tuples with the same group-by key and same timestamp are routed along the same outgoing link); and (ii) when merging streams, attention is paid to order the tuples, so that they provide a single timestamp sorted stream. For this reason, special *m*erge-sorting (M) operators are defined before each operator *instance* fed by a parallel upstream operator, while splitting steps[†] are defined after each operator instance feeding a parallel downstream operator; in Figure 7.1C, we can see an example presenting a parallel version of the sample centralized query with two instances for each of the example query's operators. M merge-sorts deterministically the incoming timestamp-sorted input streams of an operator instance into a single timestamp-sorted stream of tuples. By doing this, it allows for the operator instance's execution to be deterministic independently of the arrival interleaving of its input streams [1, 24] and degree of parallelism. A sufficient condition to guarantee deterministic merge-sorting of multiple incoming streams of tuples is for the M step to ensure that tuples are forwarded when they are *ready*. More specifically:

Definition 7.1 (ready tuple [15, 16])**.** *Let t_i^j be the i-th tuple from timestamp-sorted stream S_j. t_i^j is* ***ready*** *to be processed if $t_i^j.ts \le merge_{ts}$, where $merge_{ts} = min_k\{t_l^k.ts\}$ is the minimum timestamp among the timestamps in the set of tuples comprising the latest received tuples t_l^k from each timestamp-sorted stream S_k.*

Given Definition 7.1, as soon as a tuple t is ready, so are all other tuples sharing t's timestamp. Furthermore, if the latest received tuples t_l^k from each timestamp-sorted stream S_k share the same timestamp, all of them are ready. In general, a tuple becomes ready as soon as another, at least equally timestamped tuple is available at the other timestamp-sorted streams. Notice that the latency incurred in becoming ready does not depend on the incoming tuples' timestamps. If the order of tuples sharing the same timestamp can potentially affect the sequence of tuples produced by an operator, this ordering can become unique by making each timestamp unique. In practice, this can be achieved by having each data source and operator augmenting each input tuple's timestamp with the unique source-id or operator-id and an increasing counter.

Splitting involves routing tuples between an operator instance and its *downstream* operator instances. For example, it guarantees that tuples referring to the same vehicle are always routed to the same (and unique) operator instance in the example of Figure 7.1C.

[†] We use here the term steps rather than operators because, as shown in the following sections, merge-sorting and splitting can be both assigned to dedicated operators or integrated in the communication layer of an SPE.

Syntactic transparency allows for the centralized query to be converted into its parallel counterpart with minimal [2] or possibly no [1] configuration requested from the programmer. Elastic systems such as [1, 13] can initially deploy one instance of each operator and later provision and decommission instances depending on their load. Static systems, on the other hand, rely on the programmer to define the number of instances of each operator.

7.2.3 Streaming operators' performance metrics

We refer to streaming operators' metrics that are commonly used to assess the performance of a streaming framework (from individual operators to queries or SPEs). In particular, we focus on *throughput*, *latency* [1, 15] and *energy consumption* [25]. Throughput, commonly measured in tuples per sec (t/s), represents the maximum rate at which tuples can be fed to the operators composing a given query. Latency, commonly measured in msec, is the time between the forwarding of an output tuple and the timestamp carried by the latest input tuple contributing to it. Finally, energy consumption measures the average power consumption (in Watts) incurred by the SPE.

7.3 Operator- vs communication-layer determinism

As we explained in Section 7.1, determinism is typically enforced at the SPE operator layer. That is, the merge-sorting required to enforce determinism (cf. Section 7.2) is run by dedicated operators, deployed together with the operators defined by the application programmer. Alternatively, as we propose, determinism can be taken care of in the communication layer of an SPE, the one in charge of buffering operators' input and output tuples. To introduce *layering* for the SPE functionality provisioning, wlog, we consider in the following the node in Figure 7.2. This node shows the operators F, $A2$ and M of Figure 7.1B. Our discussion holds independently of whether other operators are deployed within the SPE running the query or if more than two instances are defined for each operator.

7.3.1 Limitations of operator-layer determinism

Here we explain limitations and potential bottlenecks from implementing determinism in the SPE operator-layer, that can be alleviated through provisioning of

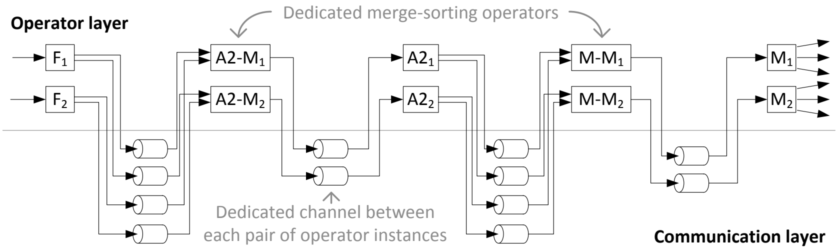


Figure 7.2: Parallel query run by an SPE with operator-layer determinism.

determinism in a communication-layer module, as we propose.

Limitation 1. *Determinism implemented in the SPE-operator layer implies more operators to deploy and run at each node.*

As discussed in Section 7.2, determinism is typically achieved by deploying dedicated operators merge-sorting the multiple timestamp-sorted input streams of each operator instance fed by two or more upstream operator instances (e.g., input mergers [1] or SUnions [19]). In our example in Figure 7.2, two instances for such dedicated operator $A2$-M are defined for operator $A2$.

The deployment of dedicated merge-sorting operators in-between the query's operators results in a higher number of threads in SPEs such as Storm [2] or Flink [3] or in scheduling overheads for SPEs with schedulers [1, 26, 27] ordering operators' execution, thus degrading throughput and increasing energy consumption.

Limitation 2. *Determinism implemented in the SPE-operator layer implies the inter-operator overhead per tuple increases.*

Dedicated merge-sorting operators rely on dedicated queues that allow for the pipelining of their data processing with the processing defined by the operator instance to which they are connected to, as shown in Figure 7.2. The overhead grows with the increase in operator instances and queues each tuple traverses. The additional operator instances (and the subsequent traversing of queues) decrease the overall throughput, increasing the processing latency and the energy consumed by the node [25]. In our example, each tuple goes through 3 queues and 3 operator instances to traverse operator A2 from operator F to operator M.

Limitation 3. *Determinism implemented in the SPE-operator layer implies that merge-sorting steps can become a bottleneck.*

The maximum throughput of an operator instance can be observed only if its preceding operator(s) (that is, its upstream operators and the latter's upstream peers) are not under-provisioned. If dedicated merge-sorting operators are used, this implies that the maximum throughput of an operator instance can be achieved as long as the cost of the merge-sorting itself does not constitute a bottleneck. Unfortunately, the latter might have a cost comparable to or higher than both stateful and stateless operators, depending on the latter's selectivity, as we show in Section 7.5.

Limitation 4. *Determinism implemented in the SPE-operator layer implies that the parallelism degree of an operator depends on its upstream and downstream peers.*

If we have an overloaded operator `Op`, parallel instances of operator `Op`, may mitigate the processing overhead at A, however resulting in a bottleneck for upstream operators feeding `Op`, or for the downstream operators that take resulting tuples from the parallel instances of `Op`,. Therefore, a high degree of parallelism is not always the solution, as it will not only increase compute resource requirements, but it might also cause degradation on the throughput and latency. In the example deployment of Figure 7.2, for instance, assume that the cost of merge-sorting for operator instances $A2$-$M_1, A2$-M_2 is higher than the processing cost of operator instances $A2_1, A2_2$. In such a case, the degree of parallelism for operator $A2$ could be increased to e.g., four instances. By doing this, each of the four instances of operator $A2$-M would then be responsible for the merge-sorting of half of the input tuples. Nevertheless, each instance of the merge-sorting operator preceding operator M (not shown in the figure) would now observe a higher cost for the merge-sorting of its input tuples (coming from four rather than two input streams). Hence, increasing the degree of parallelism for $A2$ could overload the merge-sorting of tuples feed to M, thus decreasing, rather than increasing, the overall throughput of the query to which the two operators belong to. The critical drawback to this synchronization overhead is not given by the cost of merge-sorting since this must be paid to enforce determinism. It is rather to be found in the fact that such cost is assigned to a specific operator with dedicated input and output queues rather than shared among multiple operators [15].

Limitation 5. *Determinism implemented in the SPE-operator layer implies that the pipelining of the query's operators might be limited by the merge-sorting steps.*

A second synchronization overhead depends on how operator instances are assigned to threads by a given SPE. On one hand, a single thread can be in charge of the processing of multiple operators instances. SPEs such Aurora [26], Borealis [27] and StreamCloud [1] define a scheduler thread within each node that runs (one at a time) the operator instances associated with the node. This design decision simplifies the synchronization for the operator instances' communication since it avoids concurrent access to the shared queues (i.e., it does not require locking mechanisms that can affect operator's throughput and latency performance). On the downside, it also limits the throughput and latency performance by having, for each scheduler, exactly one operator instance running at the time. A different option is to have dedicated threads for each operator instance, as done in Storm. This might have a higher synchronization cost because of the concurrent accesses to the shared queues and might also result in unbalanced work, depending on the cost and selectivity of each operator. When assigning one thread per operator instance, threads assigned to the query's operators might be underutilized when compared with threads assigned to merge-sorting operators. The optimal mapping of threads to operators is itself complex, given that it depends on many factors (some of which might change at runtime in a deployed query) such as the number of input streams to be merge-sorted by a given operator instance or the cost and selectivity of the latter, among others.

The synchronization overhead (especially when using naive locking mechanisms as in [26, 27]) can result in throughput and latency degradation while unbalanced workload among the threads deployed within one node can potentially result in unnecessary energy consumption, especially if a smaller number of threads than the ones deployed in a node are sufficient to carry out the latter's analysis.

7.3.2 Additional potential benefits from determinism provisioning in the SPE-communication-layer

A basic aim of communication-layer determinism is to avoid the deployment of merge-sorting operators in between each operator and its upstream peer instances. As shown in Figure 7.3, this allows for the parallel instances of an operator F to be directly connected to those of operator $A2$. Since the merge-sorting would still need to run to attain determinism, a requirement of communication-layer determinism is to leverage threads that are already deployed by the SPE and share the merge-sorting cost rather than assigning them to a dedicated sort operator, as this would, in turn, result in the previously discussed overheads. As discussed in [15], communication-layer merge-sorting can be carried out by multiple threads in a scalable fashion where the cost and the degradation that merge-sorting itself introduces, in terms of throughput and latency, is minimized by avoiding coarse-grained locking mechanisms.

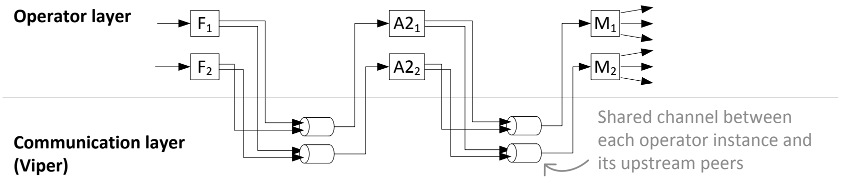


Figure 7.3: Parallel query run by an SPE with communication-layer determinism.

7.4 The Viper module

The Viper module enables determinism at communication-layer, thereby coping with the potential bottlenecks and limitations described above. In a nutshell, it enables determinism by merge-sorting the tuples delivered to an operator instance when such tuples are delivered by two or more physical streams. This is achieved transparently (without requiring the application programmer to explicitly place any merge-sorting operator) at the communication-layer of the SPE, allowing for the merge-sorting cost to be distributed among the threads delivering the tuples rather than assigning them to a dedicated thread, as we further explain in the remainder. Viper provides an API defined by three main methods, as presented in Table 7.1. A channel is maintained at the communication-layer for any set of source operator instances $S_1, \ldots, S_m$ feeding a reader operator instance R (we use the term channel to refer to the data object used by a set of operator instances to share information, be it a queue or other buffer object). The channel is either a thread-safe concurrent queue (when exactly one source S_1 and the reader R are connected) or a ScaleGate [15] object (when at least two source operators S_1, S_2 and the reader R are connected). ScaleGate allows tuples from different input streams to be merge-sorted into a single list, assuming that each source delivers tuples in non-decreasing timestamp order (ScaleGate's API provides for this the `addTuple(channel, sourceID, tuple)` method). Furthermore, ScaleGate allows the list to be read in timestamp order by an arbitrary number of readers (method `getNextReadyTuple(channel, readerID)`), guaranteeing that only *ready* tuples (cf. Definition 7.1) will be delivered. The original ScaleGate is a dynamic data structure with a poll-based API where sources work independently from readers. Thus, the data structure can grow arbitrarily in size, e.g., in cases where the readers are slower than the sources. In this work, we propose and integrate a flow-control approach using special watermark tuples [28] internally in the data structure. Such tuples are added periodically by the sources and allow the readers to acknowledge the consumption rate to the sources, through a handshake mechanism, so that the latter can limit their injection rate if the readers are slower.

Viper allows for the cost of merge-sorting to be shared by the threads assigned to the parallel instances of an operator which feed the same downstream operator instance. It also results in a scalable, probabilistically logarithmic merge-sorting due to its algorithmic implementation (presented in [15]), which is also lock-free, thus reducing the necessary synchronization overheads [29].

Table 7.1: API of the Viper module

Method	Description
`void register(channel, sources, reader)`	Register a new *channel*, specifying which *sources* will add tuples to it and which *reader* will get such tuples as a timestamp-sorted stream of ready tuples.
`void addTuple(channel, sourceID, tuple)`	Add a *tuple* from a given *sourceID* to the specified *channel*.
`tuple getNextReadyTuple(channel, readerID)`	Retrieve next ready *tuple* (if any) for the given *readerID* from the specified *channel*.

7.4.1 Viper as an SPE module: Apache Storm use case

To provide an evaluation on how communication-layer determinism boosts the intra-node performance of an SPE, we integrated the Viper module in Apache Storm [2] (henceforth referred to as Storm), one of the most widely used open source SPEs. In this section, we provide an overview of Storm's architecture (focusing on the operator instances deployed within a given node) and discuss its communication and synchronization overheads, in relation to those introduced in Section 7.3.1. Subsequently, we present how the Viper module is leveraged within Storm, discussing why it improves the performance of operators deployed in the same node, complementing the empirical evaluation presented in Section 7.5.

Storm refers to queries as *topologies* while it distinguishes operators into *spouts* and *bolts*. The former represent data sources and thus define only one (or multiple) output streams. The latter are generic operators and define one (or multiple) input and output streams. When deployed within the same Worker (an instance of the JVM), the instances of an operator share the same virtual memory. In this sense, a Worker represents Storm's counterpart of our node, to which we refer to in the previous sections.

Storm partially provides deterministic execution and syntactic transparency. With respect to determinism, Storm allows the programmer to specify a routing policy (referred to as *grouping*) for the tuples exchanged between an operator instance and its downstream peer's instances. It leaves the merge-sorting task to the programmer, who must then deploy it using regular operators placed before

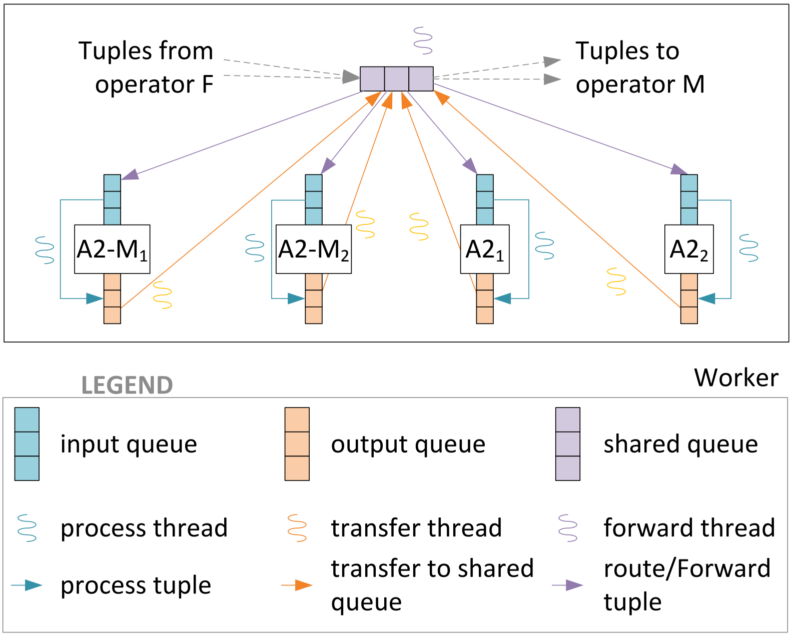


Figure 7.4: Storm Worker with two instances of the $A2$ operator (and the instances of its merge-sorting operator $A2$-M) deployed in it. To ensure determinism, a dedicated thread is required for merge-sorting the tuples fed to each operator instance.

each operator instance fed by two or more upstream operator instances. With respect to syntactic transparency, Storm relies on the programmer to specify the number of instances for each parallel operator composing a certain query and the aforementioned routing policy. It offers transparency, nevertheless, on the compilation and deployment of the parallel instances at the set of available Workers.

(A) Overheads of operator-layer determinism in Apache Storm

Figure 7.4 overviews the architecture of a Worker, also presenting the different threads operating in it. The figure shows a Worker with the same operators presented in Figure 7.2. As shown, dedicated merge-sorting operator instances $A2$-M_1, $A2$-M_2 are deployed for the operator instances $A2_1$, $A2_2$.

Each operator instance is deployed with a local *input* and *output* queue and assigned two threads. A *process* thread is responsible for processing tuples from the input queue and for storing resulting tuples into its output queue (also specifying the downstream instance to which an output tuple should be routed to). A second *transfer* thread is then responsible for copying tuples to a *shared* queue (shared among all the operators deployed within the same Worker). A global *forward* thread is responsible for copying tuples from the shared queue to the input queue of a locally deployed operator instance (if the former are to be fed to such instance) or sending them (through the network) to a different Worker. A distinct dedicated thread (not shown in the figure) is responsible for the delivery of input tuples taken from the network to the input queue of an operator instance. The queues used by Storm are the ones defined by the LMAX Disruptor library [30]. It should be noted that one individual input and one individual output queue are defined for each operator instance. Tuples coming from multiple operator instances are thus maintained in the same queue.

Based on its architecture, Storm incurs all the limitations discussed in Section 7.3.1. As shown in Figure 7.4, a tuple traveling from operator F to operator M is traversing seven queues (the shared queue, the input and output queues of a merge-sorting operator instance, the shared queue, the input and output queues of an instance of operator $A2$ and, finally, the shared queue). At the same time, by having one dedicated thread in charge of performing the merge-sorting of the tuples fed to each operator instance, the merge-sorting operation itself can constitute a bottleneck to the scaling up of the instances deployed within the same Worker, especially when the cost of the operator is lower than the cost of merge-sorting itself.

(B) Additional overheads - sharing tuples

As presented, a dedicated transfer queue accommodates all the tuples produced by the operator(s) instances within a Worker before the latter are either forwarded to another Worker or copied to the input queue of a different operator within the same Worker. This extra transfer step further affects performance and energy consumption. Moreover, it also introduces a second bottleneck (non-disjoint parallelism) since the operator(s) instances running at a Worker can be only fed as fast as this thread can deliver tuples. This architectural choice can also limit the lockstep processing of tuples. If two tuples to be processed by two different operators exist in this shared queue, the respective operators will get them in sequential order while in principle both tuples could be processed in parallel.

(C) Integration of the Viper module

As shown in Figure 7.5, the integration of the Viper module in Storm allows for each operator to (1) *bypass* its input and output queues, and rather rely on

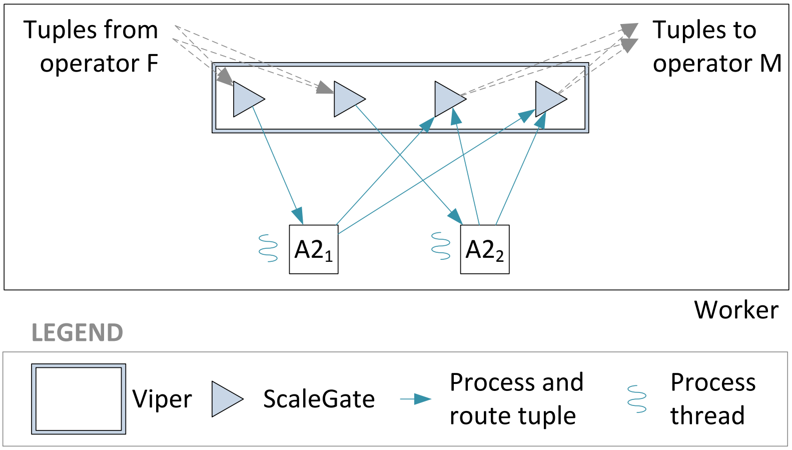


Figure 7.5: Storm Worker with two instances of the $A2$ operator connected by the Viper module.

two channels maintained at the Communication-layer and (2) rely on a single thread to fetch input tuples and route and store output tuples. Bypassing input and output queues removes the need for a transfer thread. Since channels ensure deterministic processing, the Viper module does not require the deployment of dedicated merge-sorting operators (as discussed in Section 7.4). This further reduces the number of threads running in a Worker by a factor of two, for each operator instance fed by two or more upstream operator instances. This joint reduction results in higher-throughput because of the reduced number of copies and queues each tuple traverses in this setup. At the same time, it results in lower energy consumption given that it requires for fewer active threads deployed in a Worker instance.

It is important to note that Viper enables disjoint-parallel execution of operator(s) instances as all operators can access input and output channels concurrently, without compromising consistency of the channel. When used with SPEs that, such as Storm, provide routing but not merge-sorting, an additional benefit of the Viper module is that, while enforcing determinism, it also augments the syntactic transparency from the programmer's perspective, since the latter does not need to deploy dedicated merge-sorting operators at all.

Table 7.2: IDs, description, cost and selectivity of the Linear Road operators included in the performance evaluation.

ID	Description	Selectivity
`pos_rep`	Forward an incoming tuple if it is a position report (stateless operator).	0.99
`new_seg`	Check whether a vehicle is entering a new segment. That is, if the previous and the current tuples' positions refer to two different segments (stateful operator).	0.34
`zero_speed`	Forward an incoming tuple if it indicates that the vehicle's speed is zero (stateless operator).	0.0001

7.5 Evaluation

In this section, we evaluate the performance achieved by a streaming application when relying on the Operator-layer determinism as provided by Storm and compare it with the Communication-layer determinism provided by the Viper module (cf. Section 7.4) integrated in Storm. We study this both from an intra-node perspective in relation to parallel analysis defined by the Linear Road benchmark and from an inter-node parallel and distributed stream analysis perspective with an application from a real-world smart grid. For both cases, we first introduce the hardware, the software, and the experimental setup. We then proceed studying the performance that, as mentioned in Section 7.2, is measured in terms of throughput (t/s), latency (ms) and, for intra-node parallelism case, the power consumption (Watts) for both Communication-layer and Operator-layer.

7.5.1 Intra-Node Parallel Analysis - Setup

We conducted our experiments on a dual-socket Intel Xeon E5-2687W 3.4GHz server, with 8 cores per socket (yielding a total of 16 cores, 32 threads) and 64 GB of RAM. The server runs Scientific Linux 6.5 (5) based on the Red Hat Enterprise Linux operating systems. We used *likwid* [31] to read out RAPL Energy counters for the power metrics presented in our evaluation. All experiments were run using Storm version 0.9.7 and OpenJDK Java version 1.8.0_91. The ScaleGate implementation utilized in the Viper module is the one available at [32]. For channels accessed by a single source and reader (cf. Section 7.3), the Viper module relies on Java's `ConcurrentLinkedQueue`.

As discussed in Section 7.2, dedicated merge-sorting operators are deployed for operator instances fed by two or more upstream operator instances to enforce deterministic processing in Operator-layer. State of the art merge-sorting operators such as Input Mergers [1] rely on individual queues where tuples forwarded by the upstream operator instances are buffered and later merge-sorted.

In our evaluation, we make use of data generated from the Linear Road benchmark [17]. Linear Road is an established benchmark to study SPEs' performance that simulates vehicular traffic on a number of linear expressways, each composed of predefined *segments*. *Position reports* are forwarded every 30 seconds and carry the vehicle's *position* and *speed*. Vehicles are charged with a variable toll based on the traffic congestion level and the presence of *accidents*. The generated data is continuously processed to (i) detect possible accidents and (ii) compute tolls and notify vehicles[‡]. The Linear Road benchmark [17] is a representative example where stream processing in fog/edge architectures can imply extra benefits compared to processing in the cloud, as discussed in [18]. The generated data simulated the traffic over 10 highways.

We provide the evaluation results for both a stateful and a stateless operator on data extracted from the benchmark. Table 7.2 contains a summary of the operators, their ids, and selectivity. We evaluate individual operators rather than pipelines since the increased number of Storm's communication channels, and synchronization bottlenecks in the latter case would unfairly degrade its Operator-layer based parallelization approach compared to Communication-layer.

7.5.2 Intra-Node Parallel Analysis - Scalability

To study the performance of parallel instances of an operator, we start by deploying one instance of such operator together with one data injector and one sink (pipeline parallel). The injector generates input tuples as fast as downstream operators can process them while maintaining the throughput statistics. During execution, we measure throughput as the number of tuples generated over each 5-second period and report the average throughput per second. The sink maintains average latency statistics in seconds. The experiments are repeated 6 times; the reported values are averages over the runs of the same experiment. This initial deployment allows us to measure the performance of an operator's centralized execution.

The performance of the data parallel counterpart depends on the parallelism degree (i.e., its number of parallel instances) and the parallelism degree of its upstream operator (i.e., the overhead introduced by deterministic merge-sorting of the streams of the parallel upstream operator), as discussed in Section 7.3. For this reason, we increase the number of instances both for the injector and

[‡] The benchmark also defines other historical queries, which do not relate to the topic of this paper and are thus not discussed here.

the operator to 2, 4 and 6 (i.e., we deploy 1 injector and 1, 2, 4 and 6 parallel operator instances, 2 parallel injectors and 1, 2, 4 and 6 parallel instances, ...) for a total of 16 configurations for each operator. The number of parallel sink instances deployed in each experiment is equal to the number of parallel operator instances for the sink not to constitute a bottleneck.

The purpose of the benchmarks is to highlight the overheads of achieving determinism. Therefore, with Operator-layer determinism, a merge-sorting operator is deployed for each instance of the operator if two or more injectors are deployed. Similarly, a merge-sorting operator is deployed before each instance of the sink if two or more parallel operator instances are deployed (no extra merge-sorting operators are needed for Communication-layer determinism using the Viper module). The highest degree of parallelism for the injector and operator is chosen so that the overall number of threads for both Operator-layer and Communication-layer that process and forward tuples is in the same order as the number of logical threads provided by the server.

(A) Operator `pos_rep`

Figure 7.6 presents the performance results for the `pos_rep` operator for Communication-layer (left) and Operator-layer (right) determinism. Each sub-graph contains 4 lines, for 1, 2, 4 and 6 injectors, respectively. The upper sub-graphs present the throughput for the increasing number of instances of the parallel pos_rep operator. The middle sub-graphs present the latency while the lower sub-graphs present energy consumption.

The stateless operator `pos_rep` has a very high selectivity, almost each input tuple results in an output tuple. Thus, although it is a light filtering operator, its communication overhead (i.e., receiving and forwarding tuples) dominates the cycles spent processing tuples.

For Operator-layer, the best throughput is achieved when one injector is deployed, increasing the number of injectors degrades the performance. There is no significant improvement in performance as we increase the number of processing operators. This is mainly a result of communication overheads being the dominant factor and these coupled with parallelization overheads (managing contention when processing in parallel while maintaining determinism) outweigh the benefits of data-parallel processing. Latency figures follow a similar trend, with latency increasing with injectors and operators, indicating that with communication bottlenecks, data parallel execution does not offer performance improvements and may result in degradation.

In contrast, Communication-layer determinism, which optimizes the communication and parallelization costs, offers better performance with both increasing

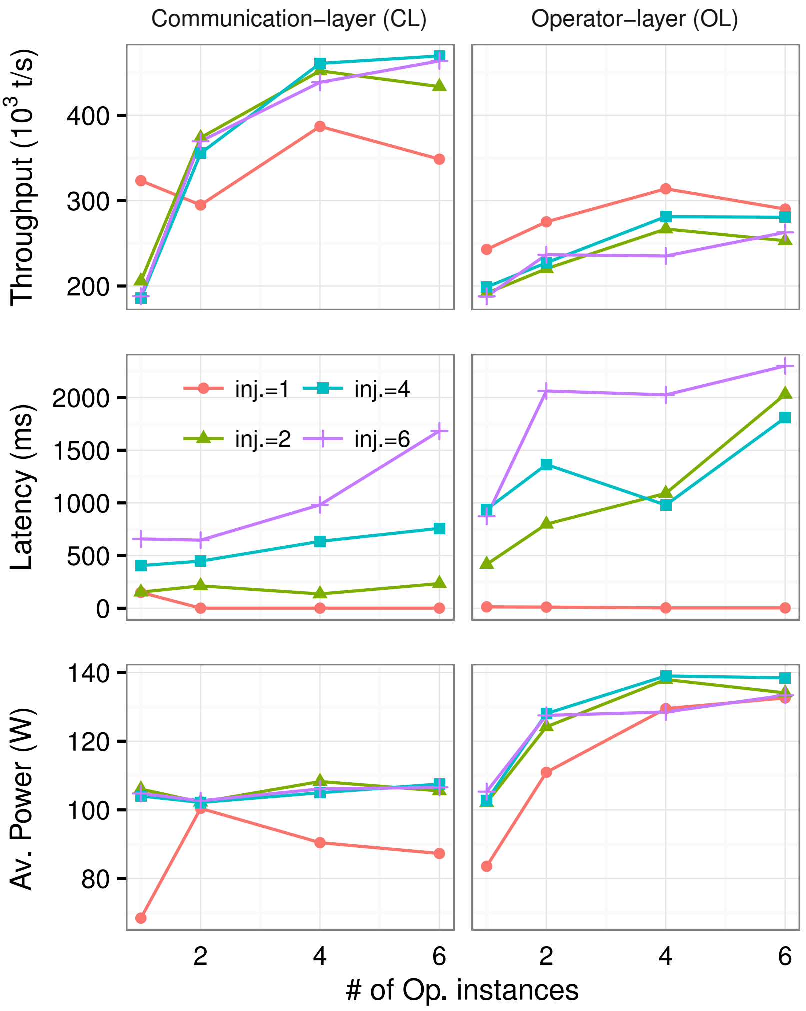


Figure 7.6: Operator `pos_rep` performance evaluation.

injectors and processing operator instances. The performance with a single injector does not improve with increasing operators, which leads us to conjecture that the injector is a bottleneck in this deployment. We examine this conjecture later in this section. The figure further shows that Communication-layer determinism manages to achieve a significant improvement in throughput without compromising latency. The latency values increase with mores injectors or operators, however, the values are less than the ones observed for Operator-layer determinism.

While still incurring similar latency (lower in this case for Communication-layer), Operator-layer's consumption grows up to 165W; at the same time, Communication-layer's consumption does not grow with increasing number of operators, due to the shared sorting work performed by the threads already deployed in Storm. We defer a discussion on power consumption to Section (D).

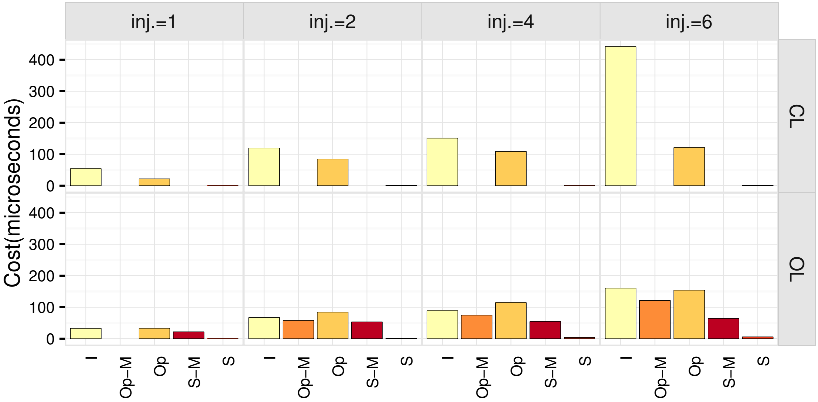


Figure 7.7: Costs of the operators deployed for Communication-layer (CL) and Operator-layer (OL) when 6 instances of operator `pos_rep` are deployed in a Worker.

For better insight into performance results for both Communication-layer and Operator-layerdeterminism, we show in Figure 7.7 the costs for the different operators deployed in the Storm Worker for Communication-layer (top row) and Operator-layer (bottom row). The operator names are I, Op-M, Op, S-M, and S; for the injector, the operator upstream merge-sorting peer, the operator itself, the sink upstream merge-sorting peer and the sink respectively. As shown, the cost of merge-sorting for Operator-layer (`Op-M` and `S-M`) is comparable to the cost of the operator `pos_rep` itself. Comparing the two, we observed that, due to the reduced number of operators and the fine-grained synchronization enabled by ScaleGate, the costs for both the processing operator (`OP`) and the sink (`S`) are significantly reduced. The lower cost for operator `pos_rep` in Communication-layer and reduced cost of achieving determinism (no need for `Op-M`, `S-M`) allows for higher throughput which scales with data parallelism. Additionally, the figure further highlights that the injectors are the bottlenecks as conjectured earlier. Observe that the cost of the Injector (`I`), includes that of sorting tuples and synchronization in the shared channels.

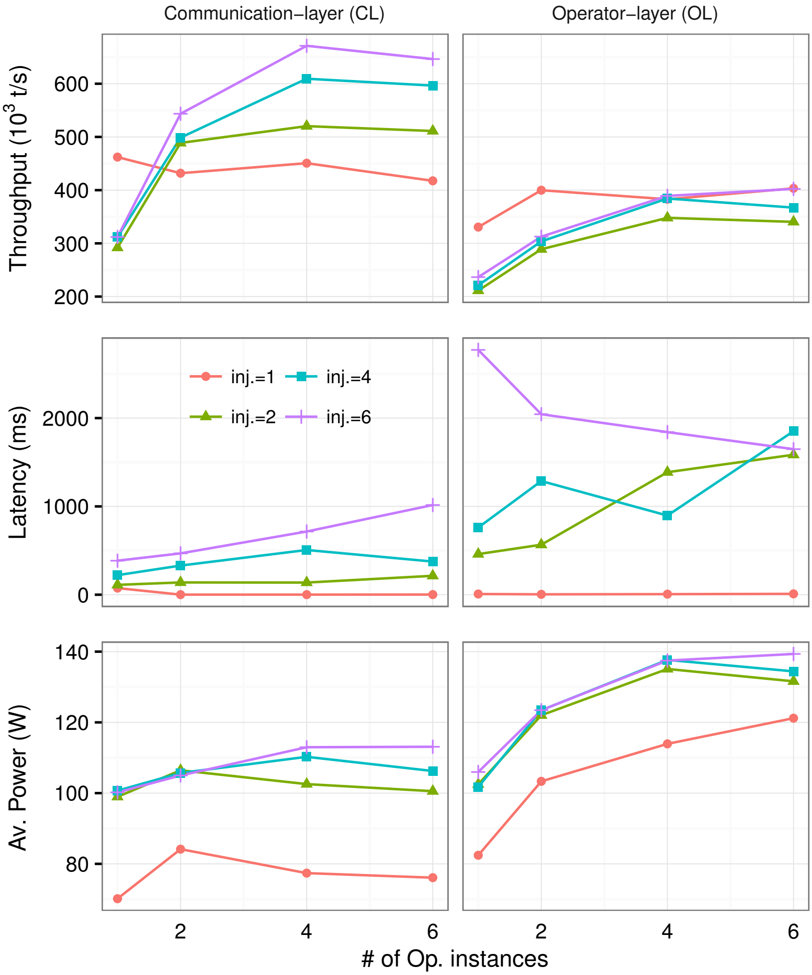


Figure 7.8: Operator `new_seg` performance evaluation.

(B) Operator `new_seg`

Figure 7.8 presents the performance results for Communication-layer and Operator-layer determinism for the execution of `new_seg` operator. In contrast to the stateless operator `pos_rep`, the stateful operator `new_seg` is characterized by lower selectivity. This implies that, for the same input rate, the latter results in a lower output stream rate. Given its stateful nature, a higher number of cycles is spent executing the operator compared to communication overheads and thus expected to benefit from data-parallel execution. Figure 7.8, top row, shows that both Communication-layer and Operator-layer benefit from data-parallelism, thus as we increase number of operators, throughput increases significantly.

With a single injector in Communication-layer, the injector becomes the bottleneck, and thus the overheads of parallelization dominate causing a degradation in performance as we increase to more than two operators. Increasing the number of injectors enables increased utilization of parallel operator instances, thus resulting in improved throughput performances. Generally, Communication-layer achieves higher throughput than Operator-layer, which increases with more injectors and saturates at 4 injectors. Similar to `pos_rep`, increasing injectors while maintaining one processing operator offers less throughput compared to a single injector case. This is expected as a result of sorting overheads incurred to achieve deterministic processing in Communication-layer as well as Operator-layer.

Additionally, we note that even for the stateful operator, increasing the number of operators while maintaining determinism should increase the latency. This is the trend observed in Figure 7.8 (middle row), latency increases with both increase in injectors and with operators, with worst latency record for 6 injectors and 6 injectors. However, we still observe that Communication-layerdeterminism offers better performance on latency than observed with Operator-layerdeterminism.

The power consumption trend is also similar to that observed for `pos_rep`. Communication-layer determinism consumes less power than Operator-layer determinism by not deploying dedicated merge-sorting operators.

Figure 7.9 presents the costs of processing tuples for the different operators deployed in the Storm Worker for Communication-layer and Operator-layer determinism (the layout of the figure is the same as the one discussed for Figure 7.7). The figure shows that the cost of the `new_seg` operator (`I`) is lower than that of the merge-sorting operators for Operator-layer determinism. This is because of the lower selectivity of operator `new_seg`. In this case, a lower probability of producing an output tuple results in lower amounts of time (on average) spent creating the objects, encapsulating the output tuples and copying them in their respective output queue.

Additionally, the higher throughput observed for Communication-layer determinism is due to the lower cost of operator (`Op`) and the fact that no merge-sorting operators are deployed. However, as highlighted previously, the injector cost (`I`) is significantly higher since it includes synchronization and sorting overheads using shared channels.

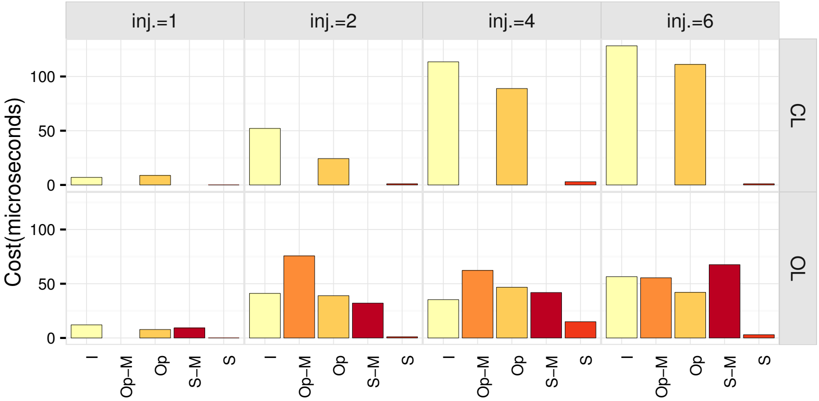


Figure 7.9: Costs of the operators deployed for Communication-layer (CL) and Operator-layer (OL) when 6 instances of operator `new_seg` are deployed in a Worker.

(C) Operator `zero_speed`

The last operator we take into account is the stateless `zero_speed` operator (Figure 7.10 and Figure 7.11). In contrast to previous operators, `zero_speed` is characterized by a considerably low selectivity (0.0001). As a result, the operator incurs negligible cost, since it seldom creates and forwards an output tuple. We expect that low selectivity and very light processing should benefit from operator parallelism. Figure 7.10 shows that indeed, the throughput in both Communication-layer and Operator-layer scales with increasing number of operators. Communication-layer determinism achieves very good scalability up to 850k tuples per second with 6 operators without significantly compromising latency. The exceptions are cases where the injector becomes a bottleneck. As with previous operators, Operator-layer determinism achieves best performance with a single injector instance.

Figure 7.11 presents details on the inexpensive nature of the processing operator (`Op`) and that most of the overheads are due to injectors (`I`) for both Operator-layer and Communication-layer, and merge-sorting operators (`Op-M` and `S-M`) for the Operator-layer based topology.

The performance results for `zero_speed` (Figure 7.10) show that inexpensive operators also benefit from data parallel executions when communication overheads and parallelization costs are optimized. Communication layer determinism allows us to optimize both costs and achieves better performance than Operator-layer determinism.

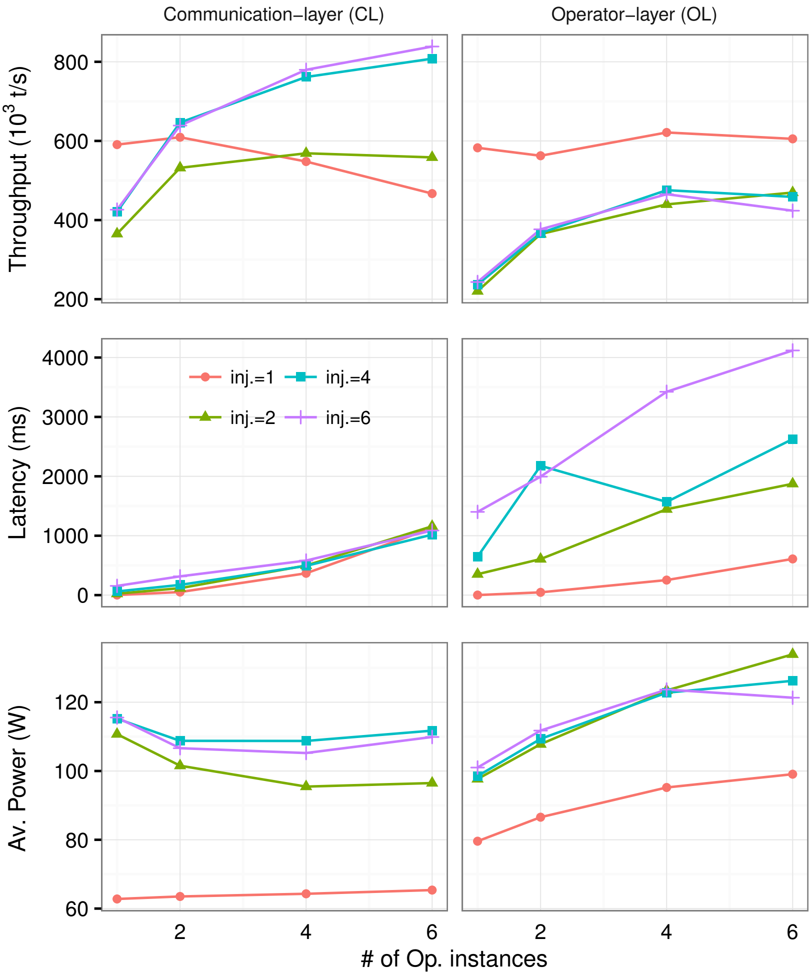


Figure 7.10: Operator `zero_speed` performance evaluation.

(D) Discussion on Power Consumption

Modern architectures deploy dynamic frequency scaling or CPU throttling where processors in idle state run at low frequency to conserve power and scale up the frequency on-demand. We observe in Figures 7.6, 7.8 and 7.10, that Operator-layer determinism dissipates on average more power than Communication-layer determinism. This is a result of differences in the number of threads

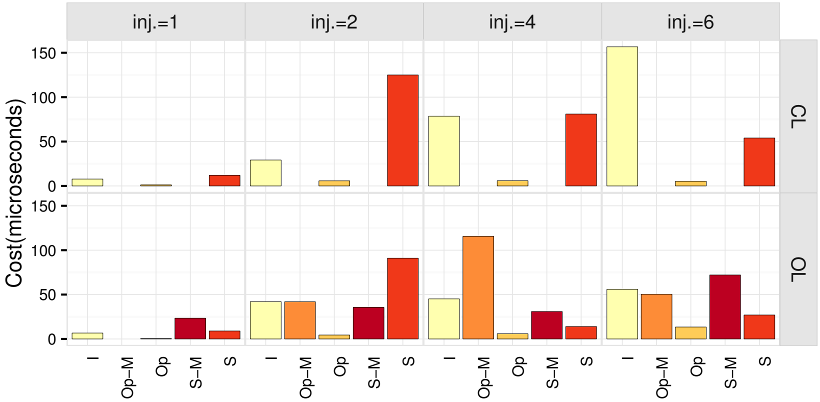


Figure 7.11: Costs of the operators deployed for Communication-layer (CL) and Operator-layer (OL) when 6 instances of operator `zero_speed` are deployed in a Worker.

utilized during a computation. Operator-layer determinism employs active merge-sorting threads and inter-operator transfer threads which are not required with Communication-layer determinism, with increasing number of execution threads, more cores are activated and run at high frequency which ultimately increases the power consumption. In this work we present power in Watts, instead of energy per tuple, as one can easily argue that with higher throughput we expect less energy per tuple. The values presented as average power during execution allow us to highlight the benefits of utilizing Communication-layer determinism in terms of energy regardless of the throughput values achieved.

7.5.3 Inter-Node Distributed Parallel Analysis - Setup

The experiments about distributed and parallel analysis share the same software setup described in Section 7.5.1 but are conducted using with a network of 6 Odroid-XU4 [33] (or simply Odroid in the remainder), equipped with a Samsung Exynos5422 Cortex-A15 2Ghz and Cortex-A7 Octa core CPUs and with 2 GB of memory.

The use-case is an application that validates and aggregates continuously the data retrieved in a smart grid setup, in which communication-enabled "smart" meters (SMs) sense energy consumption and report it continuously to the energy utility through a network of Meter Concentrator Units (MCUs). Each MCU validates the data incoming from SMs (or other MCUs) and, when receiving multiple input streams, aggregates them before forwarding the aggregated readings

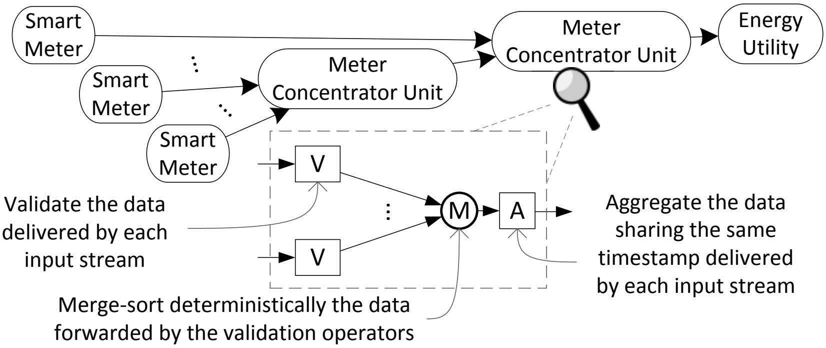

Smart Meter
Smart Meter
Smart Meter
Meter Concentrator Unit
Meter Concentrator Unit
Energy Utility
V
V
M
A
Validate the data delivered by each input stream
Merge-sort deterministically the data forwarded by the validation operators
Aggregate the data sharing the same timestamp delivered by each input stream

Figure 7.12: Distributed and parallel analysis performed by each Meter Concentrator Unit to validate and aggregate deterministically the data gathered from the Smart Meters up to the Energy Utility.

to the energy provider.

Figure 7.12 shows how the data flows from the SMs to the Energy Utility while also presenting the query that validates and aggregates the data. As shown in the figure, multiple validation operators (V) are deployed at each MCU, one for each incoming stream. The validation operators check the correctness of the readings (which cannot be negative and should not exceed the capacity of the fuse installed at each SM) and feed an aggregate operator (A) that aggregates readings sharing the same timestamp. As presented in the figure, merge-sorting steps are defined in order to aggregate data deterministically.

In our setup, each Odroid device represents one MCU. Five Odroids represent MCUs that gather data from SMs while one Odroid aggregates together the data forwarded through the network by the previous Odroids. The used data consists of anonymized energy consumption readings collected from households in 2010. In the experiment, we increase the parallelism of the validation operators (and the number of sources, accordingly) from 1 to 2, 3 and 4. We choose 4 as the maximum since, for the latter value, the number of threads deployed within the SPE instance running at each Odroid exceeds the number of available physical threads. Each experiment (for a fixed parallelism degree of the validation operator) lasts five minutes.

7.5.4 Inter-Node Distributed Parallel Analysis - Scalability

Figure 7.13 presents the throughput results (upper row) and latency results (bottom row) for for Communication-layer determinism (left column) and Operator-layer determinism (right column) for the increasing number of validation opera-

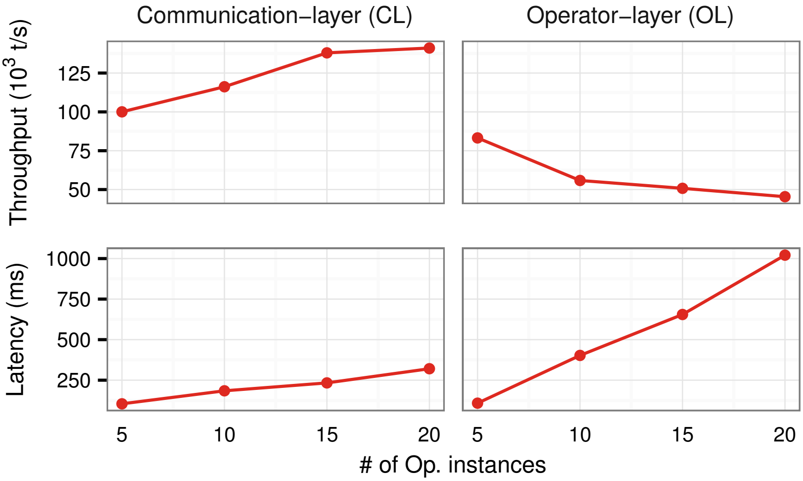


Figure 7.13: Throughput and latency results for the distributed and parallel validation and aggregation query for Communication-layer and Operator-layer determinism.

tors deployed at the five Odroid gathering data from the SMs (5 when each Odroid gathers data from 1 SM, 10 when each Odroids gathers data from 2 SMs, and so on). As it can be observed, then determinism is enforced at the Communication-layer, we observe an increasing throughput that starts at approximately 100,000 t/s and grows to approximately 140,000 t/s. As shown, the throughput increases with a milder slope when more than 3 validation operators are deployed at each Odroid. This is expected since the number of threads deployed within each SPE instance starts exceeding the available physical threads. An opposite trend can be observed for Operator-layer determinism, with a throughput that degrades from 80,000 t/s to less than 50,000 t/s accordingly to the increasing number of validation operators. At the same time, we can observe a latency that increases accordingly with the increasing parallelism of the validation operator for both Communication-layer and Operator-layer determinism. While in both cases the latency observed when each Odroid deploys gathers data from a single SM is less than 250 ms, nonetheless, it does not exceed 400 ms for Communication-layer determinism while it grows to more than one second for Operator-layer determinism.

7.6 Related work

Parallel execution of streaming operators has been first discussed by Flux [34] and StreamCloud [1, 24]. The latter provided dedicated merge-sorting operators (added to queries by a compiler) to enforce deterministic execution at the operator layer, incurring the limitations discussed in Section 7.3. The techniques in [1, 24, 34] are now found in widely-adopted SPEs. It should be noted that parallel execution of streaming operators is a first step towards elastic protocols [1, 1, 6–9, 13] that can adjust the parallelism degree of streaming operators according to varying computational loads. However, these works either rely on dedicated mergers [1, 6, 7, 13] or assume operators are stateless [9] and ignore determinism.

Schneider et. al. [13], present a compiler and run-time system that transparently extracts data parallelism, with consideration for determinism (termed as safe data parallelism). In their work, the compiler generates parallel regions and ordering is maintained on tuples exiting the parallel regions regardless of the degree of parallelism. However, as with previous efforts to achieve determinism, ordering is achieved through dedicated mergers. This is thus prone to the limitations highlighted in Section 7.3.1 and discussed further in the evaluation section (Section 7.5.2)).

The communication-layer determinism we introduce in this paper is motivated by the increasing research interest in shared-memory parallelism. The most relevant advances, nonetheless, have so far been only tailored to Aggregates [16, 35] and Joins [15, 36, 37]. The principles of the ScaleGate data object [32] have been proposed in [38] and leveraged in parallel streaming aggregation [16] and joining [15]. In relation with our work, papers such as [25, 39] discuss and provide evidence of the importance of careful design decisions for the internal communication mechanisms of SPEs. Differently from this work, nonetheless, optimizations focus on the reduction of unnecessary copies of tuples for the Borealis SPE in [25] (not considering determinism) and in a batching mechanism (complementary to the mechanism we propose) for Apache Storm.

7.7 Conclusions

Motivated by the observation that deterministic execution of streaming operators requires expensive synchronization to merge-sort the streams delivered by multiple operator instances (or data sources), we studied the limitations of operator-layer parallelism and how these can be overcome by communication-layer determinism. Reducing the communication and synchronization costs among operator instances running within an SPE is key in boosting the latter's scale-up potential, as needed in emerging cloud, fog and edge architectures in cyber-physical systems.

We propose Viper, a module that encapsulates and reduces the aforementioned costs, enabling deterministic execution to be provided transparently in the communication layer of an SPE. We provide evidence that such a module can be leveraged by SPEs, by integrating it into Apache Storm, a representative SPE of one-at-a-time analysis paradigm, for low latency processing. Our evaluation shows that, with Viper, the throughput of parallel operators increases by up to 70% and results in half of the energy consumption.

We separate discussions about general contributions from those specific to Apache Storm. The work shows that the common approach to scalability achieved by having dedicated per-thread input and output queues for each operator does not necessarily perform as good as approaches in which shared data objects (as proposed in Viper) can distribute among threads the work that could otherwise saturate an individual thread when assigned exclusively to the latter (as in the case of merge-sorting for deterministic execution of parallel and distributed streaming applications). Our contribution is a first step towards handling the communication and synchronization for deterministic streaming analysis outside the scope of the operators defining the latter. Future work can focus on the integration of Viper in Software Defined Networks, reducing the synchronization cost bypassing the OS stack.

Bibliography

[1] Vincenzo Gulisano, *StreamCloud: An Elastic Parallel-Distributed Stream Processing Engine*, Ph.D. thesis, Universidad Politécnica de Madrid, 2012.

[2] "Apache Storm," http://storm.apache.org/, 2017.

[3] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas, "Apache flink: Stream and batch processing in a single engine," *IEEE Data Engineering Bulletin*, vol. 36, no. 4, 2015.

[4] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2016, pp. 555–569, ACM.

[5] Vincenzo Gulisano, Yiannis Nikolakopoulos, Ivan Walulya, Marina Papatriantafilou, and Philippas Tsigas, "Deterministic real-time analytics of geospatial data streams through scalegate objects," in *Proceedings of the International Conference on Distributed Event-Based Systems*. 2015, pp. 316–317, ACM.

[6] Tiziano De Matteis and Gabriele Mencagli, "Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing," in

Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2016, pp. 13:1–13:12, ACM.

[7] Tiziano De Matteis and Gabriele Mencagli, "Proactive elasticity and energy awareness in data stream processing," *Journal of Systems and Software*, vol. 127, no. C, pp. 302–319, 2017.

[8] Nicolas Hidalgo, Daniel Wladdimiro, and Erika Rosas, "Self-adaptive processing graph with operator fission for elastic stream processing," *Journal of Systems and Software*, vol. 127, no. C, pp. 205–216, 2017.

[9] Le Xu, Boyang Peng, and Indranil Gupta, "Stela: Enabling stream processing systems to scale-in and scale-out on-demand," in *Proceedings of the IEEE International Conference on Cloud Engineering*, 2016, pp. 22–31.

[10] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm, "A catalog of stream processing optimizations," *ACM Computing Surveys*, vol. 46, no. 4, pp. 46:1–46:34, 2014.

[11] Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kun-Lung Wu, "Elastic scaling of data parallel operators in stream processing," in *Proceedings of the International Parallel and Distributed Processing Symposium*. 2009, pp. 1–12, IEEE.

[12] Scott Schneider, Martin Hirzel, Bugra Gedik, and Kun-Lung Wu, "Auto-parallelizing stateful distributed streaming applications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 2012, pp. 53–64, ACM.

[13] Scott Schneider, Martin Hirzel, Bugra Gedik, and Kun-Lung Wu, "Safe data parallelism for general streaming," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 504–517, 2015.

[14] Sai Wu, Vibhore Kumar, Kun-Lung Wu, and Beng Chin Ooi, "Parallelizing stateful operators in a distributed stream processing system: How, should you and how much?," in *Proceedings of the International Conference on Distributed Event-Based Systems*. 2012, pp. 278–289, ACM.

[15] Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas, "Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join," in *Proceedings of the IEEE International Conference on Big Data*. 2015, pp. 144–153, IEEE.

[16] Vincenzo Gulisano, Yiannis Nikolakopoulos, Daniel Cederman, Marina Papatriantafilou, and Philippas Tsigas, "Efficient data streaming multiway aggregation through concurrent algorithmic designs and new abstract data types," *ACM Transactions on Parallel Computing*, vol. 4, no. 2, pp. 11:1–11:28, 2017.

[17] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts, "Linear road: a stream data management benchmark," in *Proceedings of the International Conference on Very Large Data Bases*. 2004, pp. 480–491, VLDB Endowment.

[18] Stefania Costache, Vincenzo Gulisano, and Marina Papatriantafilou, “Understanding the data-processing challenges in intelligent vehicular systems,” in *Proceedings of IEEE Intelligent Vehicles Symposium (IV)*. 2016, pp. 611–618, IEEE.

[19] Magdalena Balazinska, Hari Balakrishnan, Samuel R Madden, and Michael Stonebraker, “Fault-tolerance in the Borealis distributed stream processing system,” *ACM Transactions on Database Systems*, vol. 33, no. 1, 2008.

[20] Evangelia Kalyvianaki, Marco Fiscato, Theodoros Salonidis, and Peter Pietzuch, “Themis: Fairness in federated stream processing under overload,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2016, pp. 541–553, ACM.

[21] Yuanzhen Ji, Hongjin Zhou, Zbigniew Jerzak, Anisoara Nica, Gregor Hackenbroich, and Christof Fetzer, “Quality-driven continuous query execution over out-of-order data streams,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2015, pp. 889–894, ACM.

[22] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun, “Internally deterministic parallel algorithms can be fast,” in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2012, pp. 181–192, ACM.

[23] Richard M Karp and Rayamond E Miller, “Properties of a model for parallel computations: Determinacy, termination, queueing,” *SIAM Journal on Applied Mathematics*, vol. 14, no. 6, pp. 1390–1411, 1966.

[24] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, and Patrick Valduriez, “Streamcloud: A large scale data streaming system,” in *Proceedings of the IEEE International Conference on Distributed Computing Systems*. 2010, pp. 126–137, IEEE.

[25] Shoaib Akram, Manolis Marazakis, and Angelos Bilas, “Understanding and improving the cost of scaling distributed event processing,” in *Proceedings of the International Conference on Distributed Event-Based Systems*. 2012, pp. 290–301, ACM.

[26] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik, “Aurora: a new model and architecture for data stream management,” *The International Journal on Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, 2003.

[27] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik, “The design of the borealis stream processing engine.,” in *Conference on Innovative Data Systems Research*, 2005, vol. 5, pp. 277–289.

[28] Theodore Johnson, S. Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck, “A heartbeat mechanism and its application in gigascope,” in *Proceedings of the International Conference on Very Large Data Bases*. 2005, pp. 1079–1088, VLDB Endowment.

[29] Daniel Cederman, Bapi Chatterjee, Nhan Nguyen, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas, "A study of the behavior of synchronization methods in commonly used languages and systems," in *Proceedings of the International Parallel and Distributed Processing Symposium*. 2013, pp. 1309–1320, IEEE.

[30] LMAX-Exchange, "Lmax disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads," http://lmax-exchange.github.io/disruptor/, 2011.

[31] Jan Treibig, Georg Hager, and Gerhard Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proceedings of the International Conference on Parallel Processing Workshops*. 2010, pp. 207–216, IEEE.

[32] "ScaleGate," https://github.com/dcs-chalmers/ScaleGate_Java, 2017.

[33] "Odroid-XU4," `http://www.hardkernel.com`, 2016.

[34] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin, "Flux: An adaptive partitioning operator for continuous query systems," in *Proceedings of the IEEE International Conference on Data Engineering*. 2003, pp. 25–36, IEEE.

[35] Scott Schneider, Henrique Andrade, Buğra Gedik, Kun-Lung Wu, and Dimitrios S. Nikolopoulos, "Evaluation of streaming aggregation on parallel hardware architectures," in *Proceedings of the International Conference on Distributed Event-Based Systems*. 2010, pp. 248–257, ACM.

[36] Buğra Gedik, Rajesh R. Bordawekar, and Philip S. Yu, "CellJoin: a parallel stream join operator for the cell processor," *The International Journal on Very Large Data Bases*, vol. 18, no. 2, pp. 501–519, 2009.

[37] Jens Teubner and Rene Mueller, "How soccer players would do stream joins," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2011, pp. 625–636, ACM.

[38] Daniel Cederman, Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas, "Brief Announcement: Concurrent Data Structures for Efficient Streaming Aggregation," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. 2014, pp. 76–78, ACM.

[39] Meichun Hsu, Matthias J. Sax, Qiming Chen, and Malu Castellanos, "Aeolus: An optimizer for distributed intra-node-parallel streaming systems," in *Proceedings of the IEEE International Conference on Data Engineering*. 2013, pp. 1280–1283, IEEE.

8 Conclusions and Future Work

As the number of cores available in multicore systems continues to grow, high-performance concurrent applications remain a challenge. In this thesis, we presented designs and implementations of efficient, practical concurrent data structures for both inherently sequential data structures and more scalable concurrent search data structures. We proposed mechanisms to minimize the synchronization bottlenecks by employing the combining technique without compromising the progress guarantees of lock-free vector implementations. Additionally, we extend the lock-free vector to implement a non-bounded heap-based priority queue with mutable priorities.

As future work for concurrent heap-based priority queues, one exciting continuation is the implementation of a concurrent multi-way heap to reduce bottlenecks on inserting items into the heap. The multi-way heaps lower the traversal cost by reducing the height of the tree but increase the synchronization overhead as an operation attempts to determine the priorities of all the d-children. The techniques introduced in this may be useful in implementing non-blocking versions of the heap-ordered d-ary heaps. Furthermore, relaxation of heap semantics may increase the scalability of the heap, making it easier to parallelize.

We presented efficient designs of concurrent lock-free linked-lists and binary search trees. The presented designs can easily be implemented in any programming language as they do not rely on any language specific constructs such as reference marking or runtime type introspection. We believe that such language independent designs contribute to the uptake of non-blocking data structures by application developers. We extend our research into concurrent search data structures to multi-dimensional data and similarity search.

Although tremendous effort has gone towards search data structures, not much attention has been paid to multi-dimensional data. However, many advanced applications require the manipulation of multidimensional data. We present the first lock-free multi-dimensional data structure and a lock-free linearizable algorithm for nearest neighbor search. Our method to implement the linearizable nearest neighbor search is generic and can be adapted to other multidimensional data structures.

The popularity of in-memory databases has led to a significant interest in the index structures that can support nearest neighbor search with dynamic concurrent addition and removal of data. As a continuation of our work on search data structures, we intend to explore k-nearest neighbor search, range-search queries, and iterators on multi-dimensional data. kD-trees suffer from the *curse of dimensionality* (performance degrades as the number of dimensions increases), so we plan to design lock-free data structures which are suitable for nearest neighbor search in high dimensions, for example, the ball-tree.

Finally, we considered higher-level abstractions of concurrent data structures as communication modules in data stream processing applications. Data stream applications process possibly infinite streams of data with high-throughput and low-latency demands; meeting these demands requires parallelism. There is also a growing interest in combining Cloud-server based analytics with processing closer to the Edge in order to improve performance. Data processing closer to the Edge allows for low-latency, real-time response to events, and utilization of energy efficient embedded devices. However, this requires the design of stream processing applications to consider constraints present in embedded devices. Another challenge is how to build hybrid systems that seamlessly integrate both processing at the Edge and in the Cloud.

Our experience in design and development of concurrent data structures has shown that memory management has a profound influence on the performance of dynamic data structures. Although, some memory reclamation schemes have been proposed in the literature, add significant overhead to the implementation, while others are not trivial for programmers to use efficiently. Additionally, many a time, programmers are compelled to utilize blocking memory reclamation schemes for non-blocking data structures. Therefore, there is a need for research into efficient memory management mechanisms that are easy to integrate correctly without compromising progress guarantees associated with the data structure.