THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

On Composability, Efficient Design and Memory Reclamation of Lock-free Data Structures

NHAN D. NGUYEN

Division of Networks and Systems
Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2014

On Composability, Efficient Design and Memory Reclamation of Lock-free Data Structures
Nhan D. Nguyen
ISBN: 978-91-7597-014-1

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie 3695
ISSN 0346-718X

Technical report 107D
Department of Computer Science and Engineering
Distributed Computing and Systems

Division of Networks and Systems
Chalmers University of Technology
SE-412 96 Gothenburg, Sweden
Phone: +46 (0)31-772 10 00

Author e-mail: `nhann@chalmers.se`

Printed by Chalmers Reproservice
Gothenburg, Sweden 2014

On Composability, Efficient Design and Memory Reclamation of Lock-free Data Structures

Nhan D. Nguyen

Division of Networks and Systems, Chalmers University of Technology

ABSTRACT

The transition to multicore processors has brought synchronization, a fundamental challenge in computer science, into focus. In looking for solutions to the problem, interest has developed in the lock-free approach, which has been proven to achieve several advantages over the traditional mutual exclusion approach. This thesis studies challenges in interprocess synchronization in shared memory multicore systems using the lock-free approach. Our contributions focus on efficient design and implementation, composition, and dynamic memory reclamation of lock-free data structures, a key component in lock-free solutions to synchronization problems.

First, we show that lock-free synchronization offers several advantages. Lock-free implementations of data structures can achieve decent throughput performance while managing to provide competitive fairness among the sharing participants in accessing the shared data. We also show that although lock-freedom does not guarantee starvation-freedom, it is composable in terms of the progress guarantee. Multiple lock-free data objects can concurrently use another lock-free object without compromising their lock-free progress guarantees because operations they invoke at that object get starved.

Having shown that lock-free synchronization possesses several advantages, we then propose lock-free implementations of data structures, as they play a vital role in solving synchronization problems. We present a lock-free hash table based on cuckoo hashing scheme and a lock-free skip-list with extended functionality. Cuckoo hashing uses two hash tables to offer two positions for any key, so hashing conflicts are solved efficiently and simply by placing conflicted keys in different positions. We develop a lock-free implementation by addressing challenges in manipulating elements in their two possible positions. The evaluation results show that our lock-free cuckoo hash table outperforms other state-of-the-art hash tables in the literature. The extended functionality for the skip-list is motivated by the parallelization of mark-split, an algorithm in the literature designed to reclaim unused memory.

Programming lock-free data structures raises a challenge in reclamation of dynamic memory; this is the subject that we study in the last part of the thesis. Reclaiming dynamically allocated memory blocks of data structures has always been a big issue, because they can be accessed, removed, or freed by any parallel processes. In lock-free programming the problem becomes even more complicated; because no process is allowed to wait for others. Automatic memory reclamation, or garbage collection, can free programmers from such a challenging task by safely reclaiming memory blocks that are no longer used. Based on the introduced skip-list, we propose a parallel design and implementation of the mark-split, a garbage collection algorithm that collects garbage using two steps: *mark* live objects and *split* free memory chunks to exclude occupied spaces from free memory. Furthermore, we address performance bottlenecks in the garbage collection when working on Non-uniform Memory Access (NUMA) multicore systems and introduce a NUMA-aware mark-compact garbage collector which is implemented in the OpenJDK's HotSpot virtual machine.

Keywords: Multicore Programming, Concurrent Data Structure, Synchronization, Non-blocking, Lock-free, Composability, Garbage Collection, Parallel Garbage Collection, Mark-Split, Mark-Compact, NUMA

Preface

This thesis is based on the work contained in the following publications:

I 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**", Proceedings of *the* 27^{th} *International Parallel and Distributed Symposium (IPDPS 2013)*, pages 1309-1320, IEEE Press 2013.

II **Nhan Nguyen**, Philippas Tsigas: "**Progress Guarantees when Composing Lock-free Objects**", Proceedings of *the* 17^{th} *International European Conference on Parallel and Distributed Computing (Euro-Par)*, Lecture Notes in Computer Science Vol.: 6853, pages 148 - 159, Springer-Verlag 2011.

III **Nhan Nguyen** and Philippas Tsigas: "**Lock-free Cuckoo Hashing**", Proceedings of *the 34th International Conference on Distributed Computing Systems (ICDCS - to appear)*, 2014.

IV **Nhan Nguyen**, Philippas Tsigas and Håkan Sundell: "**ParMarkSplit: A Parallel Mark-Split Garbage Collector Based on a Lock-Free Skip-List**", Proceedings of *the* 27^{th} *International Symposium on DIStributed Computing (DISC, Brief Announcement)*, Lecture Notes in Computer Science Vol.: 8205, pages 557 - 558, Springer-Verlag 2013.

V **Nhan Nguyen**, Lokesh Gidra, Gäel Thomas, Julien Sopena, Marc Shapiro: "**A NUMA-Aware Parallel Mark-Compact Garbage Collector**", Technical Report 2014:04, ISNN 1652-926X, *Department of Computer Science and Engineering, Chalmers University of Technology*, 2014.

This thesis is dedicated to the loving memory of my beloved Mom
and my dear sister Thuy.

Acknowledgments

First of all, I would like to express my gratitude to my supervisor Philippas Tsigas for his guidance, experiences and encouragement throughout my PhD journey. This thesis could not have been possible without his support.

I am honored to have Michael Spear from Lehigh University as my opponent. I would like to thank Marina Papatriantafilou for her support and many helpful insights for my research. I thank Thierry Coquand - my examiner, Patrik Jansson and Gerardo Schneider for their constructive discussions during my follow-up meetings. I also would like to thank Håkan Sundell from Borås University, and Marc Shapiro, Lokesh Gidra, Gäel Thomas, and Julien Sopena from LIP6, Université Pierre et Marie Curie for their collaboration.

I thank my two cool office mates, Zhang and Bapi, for fun times and for a lot of interesting discussions. I want to give my appreciation to Daniel and Yiannis from Distributed Computing and Systems group for their support and collaboration. I thank the rest of the colleagues in the group: Giorgos, Andreas, Elad, Farnaz, Olaf, Magnus, Valentin, Iosif, Thomas, Oscar, Vincenzo, Ivan, Aras, and Paul. You all have made the group such a great place to work.

I also take this opportunity to thank all my colleagues in the Department of Computer Science and Engineering, for their effort to make the department such an excellent environment to work. I would like to specially thank Eva, Tiina and Peter for always being helpful and responsive.

I have not only worked during these years and a bunch of time I have spent off-campus with my friends. I sincerely thank to my friends Duy, Quynh, co Lan, Cuong, Q.Anh, Dzung, Huyen, and other Vietnamese gangs in Gothenburg: Phu-Anh, Tung-Thuy, Khoa-Thanh, Cong, Nam-Dung, Nam-Thi, Tang-Linh for sharing a lot of joys during the years and recharging my mental batteries whenever they were getting low. To all friends I have not thanked yet, both near and far away, I am appreciated for all the fun and sad things we have shared.

Most importantly, I wish to thank my parents and my family for their endless and unconditional love, for their sustained support and encouragement. I am so grateful that they have always been there for me.

Nhan D. Nguyen
Göteborg, 2014

Contents

Abstract **i**

Preface **iii**

Dedication **v**

Acknowledgments **vii**

I INTRODUCTION 1

1 Introduction **3**

1.1 Shared-memory Multicore Systems 6

1.2 Synchronization 7

1.2.1 Atomic Primitives 8

1.2.2 Blocking Methods 9

1.2.3 Non-blocking Methods 10

1.2.4 Concurrent Data Structures 12

1.2.5 Composition of Lock-free Objects 14

1.2.6 Safe Memory Reclamation 15

1.3 Garbage Collection 17

1.3.1 Garbage Collection Algorithms 18

1.3.2 Generational Heap Layout 20

1.3.3 Garbage Collection in OpenJDK's HotSpot 21

1.4 Contributions 23
1.4.1 Statement of Personal Contributions 26
1.5 Conclusion 27
Bibliography 29

II PAPERS 37

2 PAPER I - A Study of the Behavior of Synchronization Methods in Commonly Used Languages and Systems 41
2.1 Introduction 42
2.2 Synchronization Methods 45
2.3 Behavior: Throughput and Fairness 48
2.4 Case Studies 49
2.4.1 Data Structures 49
2.4.2 Programming Environments 50
2.5 Experimental Setup 52
2.6 Analysis 54
2.6.1 Queue: General Discussion 54
2.6.2 Queue: Environment Specific Discussion 63
2.6.3 Hash table: General Discussion 64
2.6.4 Hash table: Environment Specific Discussion 72
2.7 Conclusions 74
Bibliography 75

3 PAPER II - Progress Guarantees when Composing Lock-free Objects 81
3.1 Introduction 82
3.2 Progress Guarantees when Composing 84
3.2.1 Lock-free Implementations of Data Objects 84
3.2.2 Examining Lock-free Progress Guarantees in Object-Oriented Programs 85
3.3 A Synchronization Mechanism for Composing 86

3.3.1 Overview of Our Proposed Approach 86
3.3.2 The Operation Descriptor 87
3.3.3 The Synchronization Mechanism 87
3.3.4 Addressing the ABA Problem 90
3.3.5 Linearizability 91
3.3.6 Progress Guarantees 92
3.4 Experimental Evaluation 94
3.5 Conclusions 98
Bibliography 99

4 PAPER III - Lock-free Cuckoo Hashing 103
4.1 Overview 104
4.2 Lock-free cuckoo hashing algorithm 106
4.3 Detailed Algorithmic Description 110
4.3.1 Search operation 110
4.3.2 Find operation 113
4.3.3 Insert operation 116
4.3.4 Remove operation 116
4.3.5 Relocation operation 117
4.4 Proof of correctness 120
4.5 Experimental Evaluation 129
4.5.1 The Experimental Setup 129
4.5.2 Results 130
4.6 Conclusions 133
Bibliography 133

5 PAPER IV - ParMarkSplit: A Parallel Mark-Split Garbage Collector Based on a Lock-Free Skip-List 137
5.1 Introduction 138
5.2 Related Work 141
5.2.1 The Mark-Split Algorithm 141
5.2.2 Garbage Collection in Java Virtual Machine 142
5.2.3 Parallelizing Mark-Split 143

5.3 Concurrent Skip-List with Extended Functionality 144
5.3.1 Implementation 147
5.3.2 Correctness 153
5.4 Parallel Mark-Split 154
5.4.1 Lazy Splitting 155
5.4.2 Implementation 156
5.5 Evaluation 158
5.5.1 Evaluation Methodology 158
5.5.2 Stop-the-world Scenario 160
5.5.3 Concurrent Scenario 163
5.5.4 Memory Usage 166
5.5.5 Characterization of Applications that Benefit from ParMarkSplit 167
5.6 Conclusion 169
Bibliography 169

6 PAPER V - A NUMA-Aware Parallel Mark-Compact Garbage Collector 173
6.1 Introduction 174
6.2 Parallel Throughput Collector 176
6.2.1 Heap layout 176
6.2.2 Allocation 177
6.2.3 Young Generation Collection 177
6.2.4 Old Generation Collection 178
6.3 The Old Generation NUMA Space 180
6.3.1 Current Old Generation Space 180
6.3.2 Performance Issues 181
6.3.3 Fragmented Space 182
6.3.4 NUMA-aware Old Space 182
6.4 NUMA-aware Parallel Mark-Compact 183
6.4.1 Marking Phase 183
6.4.2 Summary Phase 184

6.4.3 Compaction Phase 185
6.5 Discussions 187
6.6 Related Works 188
6.7 Conclusions 189
Bibliography 189

List of Figures

1.1 Semantics of synchronization primitives. 8

1.2 Heap Layout of the HotSpot's Parallel Collector. 21

2.1 Throughput of the lock-free and fine-grained queues on the Intel system under high contention 55

2.2 Fairness of the lock-free and fine-grained queues on the Intel system 56

2.3 Fine-grained and lock-free queues which show major differences in fairness across platforms at 24 threads 57

2.4 Throughput of all hash tables on the Intel system under high contention 65

2.5 Fairness of all hash tables on the Intel system 66

2.6 Hash tables which have major differences in fairness across platforms at 24 threads 67

3.1 Measurement results in high contention level 95

3.2 Measurement results in medium contention level 96

3.3 Measurement results in low contention level 97

3.4 Performance of DCAS and CAS 98

4.1 State transition of two possible positions of a key in primary (upper) and secondary sub-tables 107

4.2 Concurrent inserts can create the existence of two instances of a key in the table 123
4.3 Throughput as a function of concurrency at load factor 40% . . 131
4.4 Throughput as a function load factor at 16 and 32 threads . . . 132
4.5 The number of cache-misses per operation. 132

5.1 Multiple-step process for marking and deleting blocks simultaneously with inserting new nodes, thus fulfilling the corresponding (to the right) abstract operations on the free-list. . . . 145
5.2 Garbage collection time in the stop-the-world scenario 161
5.3 Pause time when the old generation's garbage collectors work concurrently with the mutator 163
5.4 Benchmark time for the HotSpot with different concurrent GCs 165

6.1 Heap Layout of the HotSpot for Parallel Collector 176
6.2 An example of regions of different types 179

List of Acronyms

BST	Binary Search Tree
CAS	Compare-And-Swap
CMS	Concurrent Mark-Sweep
DCAS	Double-Compare-And-Swap
FAϕ	Fetch-And-ϕ
G1	Garbage-First
GC	Garbage Collector, or Garbage Collection
HTM	Hardware Transactional Memory
JVM	Java Virtual Machine
LL/SC	Load-Link/Store-Conditional
LSB	Least Significant Bit
NUMA	Non-Uniform Memory Access
PC	Parallel Throughput Collector
PMC	Parallel Mark-Compact
PLAB	Promotion-Local Allocation Buffer
PMS	Parallel Mark-Split
PMS_Lock	Parallel Mark-Split based on Lock
PMS_O	Parallel Mark-Split Optimized with lazy splitting
PS	Parallel Scavenge
TAS	Test-And-Set
TTAS	Test and Test-And-Set
TLAB	Thread Local Allocation Buffer
UMA	Uniform Memory Access

Part I

INTRODUCTION

1 Introduction

One of the guiding principles of computer architecture is known as Moore's Law [63], named after Intel's co-founder Gordon E. Moore. In 1965, he stated that the number of transistors on a chip would roughly double every 2 years. Greater transistor density packed in smaller chips has made it possible to achieve speedup by increasing clock frequency. Until the mid-2000s, microprocessor frequency was synonymous with performance; higher frequency meant a faster, more capable computer. However, it became harder and harder to achieve higher clock speeds due to several issues, notably heat, power consumption, and current leakage problems. Instead of driving clock speed higher, major processor manufacturers turned to packing multiple processing cores into a single chip [50], [64], starting the *multicore* era.

In multicore systems, multiple processors can perform computation simultaneously, thus achieving concurrency. They can execute independent computations or cooperate to complete a single task. This presents to the processors the possibility of contending to get access to shared resources where they must be synchronized. In fact, synchronization appears as a generic term everywhere in the real world. A train approaching a platform at a station must synchronize with the station control center to ensure that the platform is available. Two trains that share a single track segment and are traveling from opposite directions must synchronize so that only one train can occupy the segment during a certain period. Two computers that share a printer synchronize so that they are not printing at the same time.

An example of synchronization in computer systems is a shared First-In-First-Out job queue. The queue can be used in operating systems to manage jobs to be completed. Jobs are executed by processes running on hardware processors. A spare process extracts a job from the head of the queue and executes it, while a new job can be added to the tail of the queue. The queue maintains two variables, a head pointing to the first element and a tail pointing to the last element. In a multicore system where processes can concurrently access the queue, they probably fetch the job from the head at the same time. That both of them succeed in fetching the same job results in waste of computational resources, because multiple processes execute the same job. There are also scenarios where concurrency, if not handled properly, results in failure. To ensure that the head always points to the first job, any process, after successfully fetching a job, updates head to point to the next job in queue. An interleaving of such two-step operations executed by different processes can happen as in the following scenario. A process A fetches a job. Before it updates head, another process B manages to fetch two jobs and update head to point to the third job in queue. Process A now continues by trying to update the head to the job next to the first one, i.e. the second one. However, that job has been already extracted and completed by process B, and might have been removed from the system. If process A proceeds to update head without such awareness, head can be updated to an invalid element. Such a scenario, if not handled properly by a synchronization algorithm, can bring the queue to invalid state and/or the involving processes to memory access violation or wrong results. Similar scenarios in which processes or processors contend to access shared resources are ubiquitous in multicore systems. So are synchronization problems.

The transition to multicore computing has brought synchronization, a fundamental challenge in computer science, into focus. Solutions to the synchronization problems are, in general, difficult to achieve because of the nature of concurrency and interleaving among processes, as well as asynchrony. Multicore computers, similar to other modern computer systems, are asynchronous: each core can run at a different speed, activities can be halted or delayed, without warning, by interrupts, cache misses, preemption, failures, and so on. One way to synchronize accesses to a shared resource is to use mutual exclusion, which basically allows one process to access the resource during a period. However, synchronization algorithms based on such an approach are not able to tolerate even a single process failure, eliminate concurrency, and suffer from performance degradation. In the last two decades, interests in solutions to synchronization problems based on non-blocking approaches have developed. An implementation is non-blocking if no process can be blocked by the inaction of other processes. Non-blocking implementations have been proved to provide several advantages over their blocking counter-parts, such as high performance, progress guarantees, and fault tolerance. However, they are usually hard to design, implement, and prove their correctness.

This thesis studies challenges in interprocess synchronization in shared memory multicore systems. We focus on the composability, algorithmic design and implementation, and memory reclamation issues in order to achieve efficient lock-free solutions to the synchronization problems. We are motivated by the lock-free, a non-blocking, approach because our study of the behavior of different synchronization methods finds that lock-free implementations manage to balance the throughput performance and the fairness among contending participants in accessing shared data [10]. Our next study [18] also suggests that lock-free data objects can become composable in the sense that multiple lock-free objects can concurrently use another lock-free object while their lock-free progress guarantees are not compromised because of the starvation of the operations that they invoked at the latter object. As it is known that data structures play a key role in solving the synchronization problem [58], we then propose an efficient design and implementation of a lock-free cuckoo hash table [67] and an extended design of a lock-free skip-list [68]. Along with algorithmic design and implementation challenges to achieve lock-free data structures, it has always been a difficult task to safely reclaim dynamically allocated memory blocks of the structures because those blocks can be accessed, removed, or freed by any concurrent processes. It is then natural for lock-free implementations to employ some sorts of safe memory reclamation schemes or an automatic garbage collector. We develop a parallel version of the mark-split garbage collection algorithm based on the extended skip-list in OpenJDK's HotSpot virtual machine [68]. Furthermore, we address performance bottlenecks introduced by new NUMA multicore architecture to the current parallel garbage collection implementations and introduce a NUMA-aware parallel collector for the HotSpot [66].

The thesis is organized to two parts: Introduction and Papers. This *Introduction* continues with background knowledge, including synchronization and garbage collection in shared memory multicores, followed by a summary of our contributions, and discussions of the future research directions. The *Papers* part presents our contributions in the form of published papers.

1.1 Shared-memory Multicore Systems

A multicore processor is a multiprocessor in which all processors, or *cores*, are on the same chip and share the same memory. Each core can function as an independent computational unit. A basic multicore processor usually has very fast on-chip Level 1 cache and off-chip Level 2 cache, privately to a core. Some architectures have a shared Level 3 cache. Caches provide fast accesses to cached data but are usually small. Meanwhile, main memory is slower in access speed but is much larger in size than caches. Communications among cores and to main memory are accomplished either using a single communication bus or an interconnection network.

All the processors in a multicore system can share the physical memory uniformly as in the uniform memory access (UMA) architecture [3] or non-uniformly, as in the non-uniform memory access (NUMA) architecture [46]. In a NUMA system, access time to a memory location is dependent on the physical distance between the processor making the request and the memory chip containing the data. A processor can access its local memory faster than non-local memory, i.e., memory local to other processors or memory commonly shared among processors. In a UMA system, however, access time to a memory location is independent of which processor makes the request or which memory chip contains the data.

1.2 Synchronization

In multicore systems, a process or a thread corresponds to a given computation and runs on a hardware processor. A thread is also called a lightweight process; however, modern computing distinguishes between a process and a thread. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space. A process creates multiple threads to perform computation which share the process's resources, such as memory. In some cases, it is all right not to distinguish between a process and a thread, which is the case of this thesis. We refer to a process or a thread as an execution computation on a processor and they can share resources, especially memory. Concurrency is achieved when multiple processes are running simultaneously on different processors.

There is a natural demand for exchanging information among processes as they often cooperate to complete a common task or compete with each other to access a shared resource. Interactions among processes are of two types [78]: either data communications or synchronization. Data communications involve exchanges of data by sending/receiving messages, or by reading/writing of shared memory. Synchronization involves exchanges of control information among processes or processors. Synchronization is required when operations of processes need to obey certain order restrictions. Such a restriction can be

```
TAS(ref X)
  ⟨ old ← x; x ← 1; return
      old;⟩

FAϕ(ref X; in v)
  ⟨ old ← X; X ← op(X,v);
      return old;⟩

CAS(ref X; in old, new)
  ⟨ if (X=old) {X←new; return
      true;}
    else return false;⟩
```

```
LL(ref X)
  ⟨ return the value of X;⟩

SC(ref X; in new)
  ⟨
    if (no process has written
        to X since the last LL
        (X))
      {X ← new; return true;}
    else return false; ⟩
```

Figure 1.1: Semantics of synchronization primitives.

that no two processes can modify a shared variable at them same time, or a read must happen after a write has been completed.

Synchronization among processes in a shared memory multicore is made possible through reading from and writing to the shared memory. Multicore processors provide atomic primitives as a tool to synchronize concurrent accesses and to make consistent updates to one or a few memory words. Synchronization of concurrent accesses to a larger data set often requires the use of concurrent data structures and synchronization algorithms.

1.2.1 Atomic Primitives

A primitive is atomic meaning that no other process is able to interfere with or interrupt the execution of. There are different kinds of atomic primitives available on different platforms. Some commonly used primitives for synchronization, whose semantics are presented in Figure 1.1, are:

- Test-And-Set (TAS) - TAS sets the value of a variable to 1 if it has not been set

- Fetch-And-ϕ (FAϕ) - FAϕ atomically reads and performs a numeric or binary operation ϕ (e.g., add or binary AND) on a variable.
- Compare-And-Swap (CAS) - CAS is a more powerful primitive. It can atomically change a variable to a given new value only if the current value is equal to a given value.
- Load-Link/Store-Conditional (LL/SC) - LL and SC are a pair of instructions and together implement an atomic Read/Write. LL first reads the current value of a variable. If no other processor changes the content of the variable in-between, the subsequent SC operation of the same process succeeds and modifies the value stored; otherwise it fails.

One of the key differences among atomic primitives is their power in solving the synchronization problem. Herlihy [36] introduced a universality hierarchy that ranks atomic primitives according to their relative computational power. The power is represented by a primitive's consensus number, which indicates the number of processes that can agree on a value, using the primitive after each process executes a finite number of its own steps. Read/write to a register or a memory word has consensus number 1, TAS or FAϕ has consensus number 2. Meanwhile, CAS and LL/SC are universal operations with consensus number infinity, which means it is possible for any n processes using those primitives to agree on a value within a finite number of any process's own steps. Using atomic primitives, it is possible to implement several synchronization algorithms, which can be categorized into two types: blocking methods and non-blocking methods.

1.2.2 Blocking Methods

The traditional way of synchronizing accesses to shared resources is to use mutual exclusion. Mutual exclusion defines certain blocks of code, usually those that access a shared resource, as a critical section and ensures that only one process can be in the critical section at a time. The standard way to approach mutual exclusion is through a lock object. Other ways are, for example, semaphores, and monitors [78], [39]. As a process accessing the shared resource blocks other processes from accessing it, synchronization methods based on mutual exclusion are also referred to as blocking methods.

Blocking synchronization methods are widely used in several systems nowadays thanks to the simplicity, ease of implementation, as well as longer history compared to other methods. Mutual exclusion is often provided by the operating systems as a primitive. Modern programming languages also provide rich supports for the blocking synchronization through the language built-in constructs or libraries. Synchronization algorithms based on blocking approaches are often easier to design and to prove for their correctness than those based on non-blocking methods introduced in the next subsection.

Blocking synchronization methods suffer from many drawbacks. The major one is that they do not tolerate even a single process failure: if a process never releases a lock it has taken, all other concurrent processes that are waiting for the lock are halted. Second, the extensive use of blocking methods can make an algorithm vulnerable to dead-locks or live-locks, a situation that causes the involved processes to not make progress. Third, algorithms based on blocking methods also suffer several performance issues. The blocking behavior also means that processes that are eligible to run have to wait for some other process to complete its access to the shared resource, which results in serializing accesses to shared resources and kills the parallelization. Performance can also be degraded because of lock-convoying or priority inversion. In lock convoying, when processes fail to acquire the lock, they force context switches, which in general are computationally intensive. Priority inversion happens when a high priority task that wants to access a critical section is preempted by a lower priority process that is holding the lock.

1.2.3 Non-blocking Methods

Non-blocking property guarantees that a stalled process cannot cause all other processes to be stalled indefinitely [40]. Non-blocking synchronization can be implemented in the forms of obstruction-freedom [38], lock-freedom, and wait-freedom [36], in the order from weak to strong progress guarantees. Stronger progress guarantees are usually provided at the cost of reduced overall performance compared to the weaker ones.

Wait-freedom [36], the strongest progress guarantee, ensures per-process progress. An algorithm is wait-free if it guarantees that a process finishes the execution of an operation after a finite number of its own steps. In practice, it means that a wait-free algorithm can tolerate any number of process failures, is starvation-free, deadlock-free, and livelock-free.

Lock-freedom [36] ensures system-wide progress but allows some processes to starve. An algorithm is lock-free if and only if at any given point in time a process completes the execution of an operation after a finite number of steps. In practice, it means that a lock-free algorithm can tolerate any number of process failures, is deadlock-free and livelock-free, but is not starvation-free.

Obstruction-freedom [38], the weakest non-blocking progress guarantee, ensures progress only in the absence of contention. An algorithm is obstruction-free if and only if a process completes the execution of an operation after it has executed in isolation a finite number of steps. This means that in the case of contention among concurrent processes, obstruction-freedom does not guarantee progress for any contended thread.

Herlihy [37] introduced a universal construction proving that any sequential algorithm can be implemented wait-free using universal objects[1] and read-write registers. The implementation serves as a proof of concept rather than for any practical use. Herlihy's universal construction has been improved in many ways [4], [14], [20]; however, they are still far from being efficient enough for practical uses.

The strong progress guarantee that wait-freedom provides is usually needed in real-time systems. Weaker progress guarantees, such as lock-freedom or obstruction-freedom, are usually enough in most use cases and have been implemented in several libraries and software systems, such as Intel Threading Building Block [43], Java Concurrency Package [47], the NOBLE library [75],

[1] n-consensus object is a fundamental abstraction that allows any number of processes to agree on one of their input values

and the ConcurrencyKit library [2]. These algorithms are capable of achieving several advantages over the blocking counter-parts but do not require extreme effort to design compared to the wait-free ones. In this thesis, we have evaluated different synchronization methods, ranging from locking to non-blocking ones and the results reassure the advantages of lock-free synchronization over the blocking one. Our evaluation results suggest that lock-free implementations manage to balance between throughput performance and fairness among processes involving in accessing the shared data.

1.2.4 Concurrent Data Structures

Data structures have been one of the key components in software design. The choice of data structures is a very important decision in designing a non-blocking environment [58]. Research interest has developed in creating efficient and scalable concurrent data structures that satisfy the lock-freedom property. A large number of such lock-free implementations have been introduced in the literature. We revisit some representatives in this section.

A linked list is a general data structure that can be used to implement other abstract data types. Valois [82] and Harris [35] have constructed lock-free implementations of singly-linked lists using the CAS atomic primitive. One remark from Harris's linked list is the mark-bit technique, which uses the unused least significant bits of a pointer representation to indicate a deletion intention before actually removing the deleted node from the linked list using the CAS primitive. The technique has been widely used in later works. Lock-free algorithms for doubly linked list were introduced by Valois [83] and Greenwald [33] based on the double-word CAS primitive, i.e. a primitive which can perform a CAS operation on two distinct words. Later, lock-free double-linked lists was improved by Michael [56] based on the double-width CAS primitive, i.e. CAS on a two-word block, and by Sundell and Tsigas [77] using the single-word CAS.

A stack is a last-in first-out type of buffer and a queue is a first-in first-out type of buffer. Lock-free implementations of stacks and queues based on linked-list implementations were proposed by Valois [82] and by Michael [55]. Gong and Wing [32], Shann et al. [73], and then Tsigas and Zhang [80] presented lock-free queues based on a cyclic array and the CAS primitive. Recently, Petrank and Krogan [45] introduced a practical wait-free queue based on the lock-free queue implementation of Michael and Scott [59]. It achieves wait-freedom by employing a priority-based helping scheme in which faster processes help the slower peers to complete their pending operations.

The dictionary abstract data type allows association of values with keys to be stored and to be searched for. Hash tables, skip-lists, and binary search trees (BSTs) can implement this abstract data type. Non-blocking BSTs have been extensively studied recently. Ellen et al. [21] introduced a design, without implementation, of an internal non-blocking BST based on single-word CAS. The algorithm achieves non-blocking by using a helping mechanism to help any possibly on-going update operation that is encapsulated in an operation descriptor. Howley et al. presented a non-blocking internal BST [41] based on a similar technique. Natarajan et al. [65] presented a wait-free red-black tree using Tsay and Li's window-based framework [79] for designing lock-free tree algorithms. The framework allows designing a non-blocking update operation by making a copy of a part of the tree, i.e., a window, performing updates on that part, and then transforming the updates to the tree. However, such an approach introduces high overhead to the update operations.

A skip-list stores elements in several ordered linked lists with different densities. Skip-lists offer probabilistic logarithmic search complexity similar to tree but does not require a complicated balance operation like balanced trees. Håkan and Tsigas [76] introduced a lock-free algorithm for a concurrent skip-list data structure based on the CAS atomic primitive. The algorithm makes use of the basic idea of Harris singly linked list, which uses CAS to give hints for the concurrent operations to help an in-progress deletion. Similar constructions were presented later by Fraser [27] and Fomitchev and Rupert [25].

A hash table is a data structure that associates keys to values. A hash table uses a hash function to compute the location of a key in an array. A hash table offers amortized constant search time, thus is more efficient than a binary search tree in several cases. For this reason, they are widely used in computer software, particularly for database indexing, caches, and sets. Michael [54] presented a lock-free hash table that uses a lock-free ordered linked list presented in the same paper to store the chain of keys that are hashed to the same bucket, i.e. conflicted keys. Shalev and Shavit [72] introduced a solution that combines a hash table with an ordered linked list to allow the hash table to dynamically increase (but not shrink) in size. Gao et al. [29] presented a lock-free hash table that allows fully resizing. However, due to its open addressing and that it marks deleted elements as tombstones rather than removing them, the algorithm requires an intensive migration process (to a new table) after many insertions and deletions have been performed. Click [15] presented a similar design and implementation in Java with more architectural and language-based tunings.

Efficient and scalable data structures are a key component in solutions for the synchronization challenges. This thesis introduces an efficient lock-free implementation of cuckoo hashing, a simple but effective hashing scheme that makes use of multiple hash tables to resolve hash conflicts. We also design extended functionality for a lock-free skip-list. The extension allows it to perform composite operations composed of multiple basic ones.

1.2.5 Composition of Lock-free Objects

Software, on an abstract level, consists of (a set of) data structures interacting with each other. In choosing data structures, a designer needs to consider minimum requirements as well as desirable characteristics. When lock-freedom is a desirable property, it is important that the software composed of selected data structures also achieves lock-freedom.

Lock-free composition from different perspectives has previously been studied in the literature. Composition of data structures is usually referred to as how to compose distinct atomic operations of two data objects into one atomic operation. Gidenstam et al. [31] and Cederman et al. [11] study the problem of combining two operations from two different lock-free objects into one compound atomic operation. These results make it possible to perform complex atomic operations such as *moves* that can move an item from one lock-free data object to another in a lock-free way. Petrank and Steensgaard [69], meanwhile, studied the problem of composing lock-free programs and services. The authors formally prove a composition theorem which states that lock-free progress is guaranteed for a lock-free program when composing with a service supporting lock-freedom. This contribution is a step towards formally studying lock-freedom. However, the paper does not consider the case when multiple programs share a service and compete with each other to use it. This way of composing programs and services can affect their progress guarantees.

We investigate progress guarantees in a composition of multiple lock-free objects. Let us take an example of a composition of three lock-free objects: a *Queue*, a *Stack*, and a *Memory Manager*. The *Memory Manager* is lock-free, so it provides lock-free progress guarantee to the *Stack* and the *Queue*. Both the *Stack* and the *Queue* are designed lock-free, if their memory allocation requests are satisfied by the *Memory Manager*. They can also be used by other objects in the program and therefore, are expected to provide lock-free progress guarantees. Though as we figure out in this thesis, with this way of composing objects, the lock-freedom of the *Stack* and the *Queue* is compromised, i.e., can no longer guarantee progress; because the operations they invoke to the *Memory Manager* get starved. A solution to regain lock-free progress guarantee when composing objects is proposed in this thesis.

1.2.6 Safe Memory Reclamation

A memory block no longer used should be reclaimed and returned to the memory allocator or the operating system. Answering the question of when to reclaim a memory block can be difficult in non-blocking synchronization due to the concurrent accesses to the shared data by different threads and no waiting among them. A non-blocking operation, by definition, does not either wait for actions by other operations or prevent other operations from taking actions. Pointers to dynamic structures can be referenced in some operations while they can also be removed and freed by other operations. Without precautions, a non-blocking operation is vulnerable to accessing a memory block that has been removed from the structure and freed by another operation. This leads to problematic outcomes such as access violation to free memory, corruption of another structure that happens to allocate the freed memory blocks, or other unexpected outcomes. It is natural for non-blocking implementations to employ certain safe memory reclamation schemes, i.e., mechanisms that allow memory blocks that are no longer used to be reclaimed without harming the implementation's correctness.

In addition to that, a well-known issue in non-blocking data structures based on the widely deployed CAS primitive is the ABA problem. The CAS primitive is usually used to assign a new value to a variable if the variable contains the same value as a previously read value. However, the CAS could not distinguish if the variable holds the read value A, or it has changed the value from A to B and then back to A. In the latter case, the CAS operation still succeeds as if the value has never changed. This is known as the ABA problem. The problem usually appears when the CAS is performed on a pointer variable where the pointed memory object is freed and reclaimed for reuse (possibly for another structure) when a concurrent thread is still referring to the original object.

One way to avoid the ABA problem is by attaching a timestamp to the variable, which is increased every time the variable changes. The CAS, which operates on the variable and compares it with the variable's original value, will not succeed if the variable has been changed because of the difference of the time stamps. As the time stamp occupies some bits, the amount of information that the variable can store is decreased. There are methods to solve the ABA problem without using time stamps through a memory reclamation scheme based on reference counting or hazard pointers. A brief list of memory reclamation solutions that are also ABA-safe, along with their implications, has been summarized by McKenney [53] and Michael [58]:

- Automatic garbage collection (GC) - On systems with GC, such as Java or C# applications, memory safety is implicitly guaranteed. The GC does not reclaim the memory as long as some thread is holding a reference to it. This makes sure that the ABA problem caused by freeing and reusing an in-use memory object no longer happens.

- RCU(Read-Copy-Update) - RCU-like solutions provide a mechanism to establish quiescence points where no thread is holding references to a block that is set to be freed. When reaching that point, the block is safely freed [53].

- Reference counting - Each memory block is associated with a counter that is incremented and decremented when a reference to that block is established. Typically, a block is freed only when its reference count goes to zero [82], [30].

- Hazard pointers - Each thread maintains a list of hazardous pointers that point to shared memory blocks that the thread is referencing but can be freed by other threads. When a thread dereferences a block, the pointer is removed from its list. Other threads that may remove the block guarantee that the block is not freed until no hazard pointer is poiting to it [57].

Automatic garbage collection reclaims memory blocks when they are no longer used, thus freeing programmers from dealing with difficult and error-prone dynamic memory reclamation. It is also a solution for safe memory reclamation and ABA prevention for non-blocking algorithms (though it raises the question of whether the GC itself is lock-free [58]). Vice versa, the advantages of lock-free data structures can be applied to improve the performance of garbage collectors.

1.3 Garbage Collection

Automatic memory reclamation has been studied since the 1950s [52]. Although garbage collectors benefit the program's simplicity and robustness, they had not been widely used compared to the traditional explicit memory management because of performance concerns. Only since the wide acceptance of Java programming language has garbage collection (GC) entered the main stream [70] and been used in large systems. Nowadays, garbage collectors have been deployed as a part of many modern programming languages, for example C# [60], Haskell [51], Python [26].

As mentioned, memory reclamation is one of the important features in designing a non-blocking synchronization solution. Garbage collection offers a safe memory reclamation; however performance has always been a concern for its wide acceptance. The development of multicore processors and systems has brought opportunities to improve the performance of GC. As a safe memory reclamation solution for non-blocking data structures, GC also can take advantage of the lock-free data structures, which is part of this thesis's contributions. We also address the new challenges created by the contemporary multicore hardware to the current garbage collectors. Our studies of GC are based on the implementations of garbage collectors in the OpenJDK's Hotspot Java virtual machine. In this section, we briefly review different garbage collection algorithms, both sequential and parallel. Then, we introduce some GC features that the HotSpot offers.

1.3.1 Garbage Collection Algorithms

Garbage collection algorithms have been extensively studied in the literature. Common garbage collectors can be categorized into either reference counting, mark-sweep, or copying, or derivations of them. *Reference counting* methods [17] record the number of references to each object and identify an object as *live* as long as its reference count is greater than zero. An object whose reference count is zero is considered *dead* and its memory can be reclaimed. Reference counting is simple to implement but has some disadvantages such as overhead space and memory fragmentation. Its major weakness is the inability to reclaim objects that contain cyclic references. The issue has also been addressed [28], [48], [7].

Copying collectors [23], [12], [8] divide memory into two semispaces, active and inactive. Objects are allocated from the active region only. When the active region is full of allocated objects, program execution is stopped and the heap is traversed. The garbage collector copies all live objects from the active to inactive space. Then, the roles of the two spaces are swapped. Copying collectors need to traverse the heap only once with the complexity proportional to the size of live spaces and they can avoid heap fragmentation. However, copying collectors usually waste half of the heap, and require moving objects. Moreover, copying collections must be performed when the program is stopped. A variation of copying collectors that is used to collect recently allocated, i.e., young, objects has three spaces: an *eden* space and two survivor spaces, called *From* and *To*. Objects are allocated in the eden space, and objects surviving from the eden space in the previous collection has been placed in the From-Space. During a collection, live objects in the eden space are copied to To-Space, which has been inactive after the previous collection, and all live objects from the From-Space are promoted to another space which store rather long live objects. After a collection, the From-Space and the To-Space flip their roles.

Mark-sweep algorithms [52] collect garbage in a two-phase procedure. In the *mark* phase, they perform a transitive closure from the root set to find and mark all reachable objects (the root set is the set of global and local (stack/register) variables visible to the active program). Objects that are not reachable are identified as garbage and can be reclaimed in the *sweep* phase. The advantages of mark-sweep are little overhead space, and that it does not move objects. However, mark-sweep has complexity proportional to the size of the collected heap and it suffers from heap fragmentation. The latter issue can be taken care of using an additional compaction phase or a mark-compact algorithm [16], a derivation of the mark-sweep algorithm. After the mark phase, mark-compact algorithms perform a *compact* phase by traversing the heap in three passes to (i) calculate the new addresses, (ii) update all references, and (iii) then move all the live objects to one end of the collected space. The result is that free memory is a contiguous chunk at the other end of the space. The problem with this method is that the compacting phase is really expensive.

Great effort has been made to design parallel GC. Halstead [34] developed a parallel version of Baker's semi-space copying GC for Multilisp on shared memory multiprocessors. During collection, the heap is logically partitioned into per-thread From-Space and To-Space, and a thread traces objects from its set of roots and copies them to its To-Space. Since then, parallel copying GCs have been improved in many ways, especially on the work-stealing mechanism among GC threads, by Imai and Tick [42], Siegwart and Hirzel [74], Flood et al. [24], Attanassio et al. [6], and Cheng and Blelloch [13]. Endo et al. [22] developed a parallel mark-sweep collector for shared memory multiprocessors that balances the loads among GC threads in the mark phase by per-object work-stealing, and in the sweep phase by fine-grained partitioning of the heap into several small blocks that are processed in parallel. Ben-Yitzhak et al [9] augmented a parallel mark-sweep collector with periodically selecting and clearing a certain area, which reduces heap fragmentation. The mark-compact algorithm that performs compaction in three heap passes was parallelized by Flood et al. [24], which divides the heap into several areas to be compacted in parallel by GC threads. The number of heap passes in compaction is improved to two, then one pass by Aboaiadh et al. [1] and by Kermany and Petrank [44], respectively.

In 2006, Sagonas and Wilhelmsson [71] introduced the mark-split algorithm, which combines advantages of the copying algorithm and the mark-sweep algorithm. It marks all reachable objects as the mark-sweep does and creates the list of free memory on-the-fly while marking, rather than leaving that task to the sweeping phase. The evaluation shows promising results but the algorithm has not yet been parallelized since then.

1.3.2 Generational Heap Layout

Studies in the literature across applications and languages show that most objects die young: most objects have a higher probability to become garbage soon after being allocated, while those surviving GCs tend to live long [49], [62], [5], [81]. The young objects should be garbage collected more frequently. A generational garbage collector exploits this by dividing the heap into several generations and collecting the younger generations more frequently.

When generational collection is used, memory is divided into separate pools holding objects of different ages, aka generations. The heap in the OpenJDK's

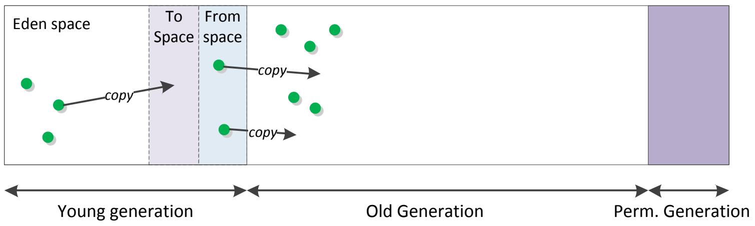


Figure 1.2: Heap Layout of the HotSpot's Parallel Collector.

HotSpot virtual machine contains three generations: young, old (also called tenured), and permanent generations. An example of the heap layout of the Parallel Throughput Collector in the HotSpot is presented in Figure 1.2. The young generation collection occurs relatively frequently and fast because the generation is usually small and contains mostly garbage. Objects that survive some collections of the young generation are promoted to the old generations. As it contains older objects, which tend to live long, the old generation is usually bigger and its occupancy grows slowly. Therefore, the old generation collection is less frequent but usually takes longer to complete. The permanent generation mostly contains objects that usually live throughout the lifetime of programs, and is not usually of interest in the study of garbage collection. It is not uncommon to use different GC algorithms to collect different generations in order to achieve efficiency.

1.3.3 Garbage Collection in OpenJDK's HotSpot

The OpenJDK's HotSpot offers a wide range of garbage collector choices for both sequential and parallel environments. Both throughput collectors and low pause collectors are available. The objective of the throughput-oriented collectors is, by decreasing the overhead of the GC, to increase the application throughput. Meanwhile, the low pause collectors prioritize the application response time. Their objective is to minimize the pause time of the application during the garbage collection. For parallel environments, the HotSpot has Parallel Throughput Collector, Concurrent Mark-Sweep, and recently introduced Garbage-First. All the collectors are generational.

Parallel Throughput Collector (PC)
The PC is a throughput collector that is designed to minimize the amount of time that the application spends to collect garbage. It is a stop-the-world parallel collector, which means it works in parallel using many processors when the application is suspended. The memory heap layout of PC is presented in figure 1.2. The collector uses Parallel Scavenge which implements a parallel copying algorithm [61] to garbage collect the young generation, which consists of one *Eden* space and two survivor spaces called *To* and *From*.

The old generation is collected by the Parallel Mark-Compact collector, which implements a two-phase parallel mark-compact algorithm. In the mark phase, the GC threads trace the object graph from the root set and mark all the live objects. The new addresses are then calculated from the mark bit map. The compact phase performs a sliding compaction in one heap pass to move objects to one end of the space and updates all the references to the new addresses, similar the Compressor collector [44].

Concurrent Mark-Sweep Collector (CMS)
CMS is an almost concurrent garbage collector that operates most of its tasks concurrently with the execution of the applications and only pauses the application for some short periods of time. CMS is suitable for applications that have a strict latency requirement.

CMS has a similar generational heap layout as Parallel Scavenge, except for its old space, which is not compacted and therefore consists of several free chunks spreading over the space, rather than a single contiguous free chunk. It uses a parallel copying collector, similar to the Parallel Scavenge, for the young generation. The old generation collection uses the concurrent mark-sweep algorithm [70] with only initial mark and remark phase being stop-the-world while other phases are executed concurrently with the execution of the application. To deal with fragmentation, it either joins free blocks if they are contiguous or performs a stop-the-world compaction.

Garbage-First Collector (G1)
Garbage-First is a concurrent and parallel collector that can achieve real-time goal with high probability [19]. It is a low pause collector that is aimed to replace CMS collector in the future. It partitions the heap into equal-size heap regions, each with a remembered set of recording pointers from all other regions. Any set of regions can be chosen for collection to ensure a short pause time. The G1 heap retains the similar types of Eden, Survivor, and Old memory pools as the above collectors; but instead of these being contiguous blocks of memory, each region is logically categorized into one of these pools. A collection of G1 starts with a global marking phase that marks all the live objects. The marking phase runs concurrently with the application. Concurrent marking can give hints to identify a collection set of regions that contain mostly garbage for reclamation. The compacting evacuation then copies all live objects in those regions to other locations in the heap, thus freeing the collection set of regions. This evacuation is performed in parallel on multi-processor systems.

The work of this thesis on garbage collection relates to the Parallel Throughput collector and the CMS collector in the HotSpot.

1.4 Contributions

The contribution of this thesis is solutions to challenges in order to achieve composability, efficient design and implementation of concurrent data structures, as well as parallel safe memory reclamation in multicore hardware. The results in this thesis have been published in the following technical papers:

Paper I - A Study of the Behavior of Synchronization Methods in Commonly Used Languages and Systems: We investigate the effects of different software and hardware factors on the behavior of synchronization methods. The variation in the behavior is measured using two metrics: *throughput* that a method achieves in an application and *fairness* among the cooperating threads. The synchronization methods that we studied range from atomic-primitive-based locks, language built-in synchronization constructs, to lock-free methods. Several parameters and factors affect the behavior of the synchronization methods through complex interactions among (i) the language and the language constructs that it supports, (ii) the system architecture, (iii) possible run-time environments, virtual machine options, and memory management supports, and (iv) applications. The study provides a comprehensive and systematic view of the considered factors to the throughput and the level of fairness that the synchronization methods can offer in different applications. The results can be used as reference guidelines for selection of programming environments and synchronization methods in connection to the application and the system characteristics.

Paper II - Progress Guarantees when Composing Lock-free Objects: The level of progress guarantee is an important characteristic that a designer needs to consider when designing software involving synchronization. By choosing data structures with the lock-freedom property, it is natural for a designer to expect that the data structures keep providing such property in any composition of the data structures. In this paper, we investigate the consequences of the fact that lock-freedom suffers from starvation to the composition of multiple lock-free objects. Since lock-freedom does not guarantee starvation-free, starvation of operations of a lock-free object can prevent the invokers, which are also lock-free objects, from making progress. This would mean that the lock-freedom is not composable in such a scenario. We propose a synchronization mechanism, based on a software implemented Double-Compare-And-Swap operation, which can atomically perform compare-and-swap on two distinct memory words, to help with fixing this issue. Using our mechanism, the lock-freedom of every object participating in the sharing combination is regained, making the lock-freedom a composable property.

Paper III - Lock-free Cuckoo Hashing: Data structures are a key component in solutions for synchronization challenges. A hash table is an associative data structure that associates keys to values, by using a hash function to calculate the index of a key in an array of slots. Hash tables offer constant search time, thus are widely used in computer software. This paper presents an efficient lock-free implementation of the cuckoo hashing algorithm. Cuckoo hashing uses two hash tables with two distinct hash functions to store elements, and elements can be moved between its two possible positions on the tables. Efficiently managing two hash tables in a single data structure while respecting its correctness is challenging in lock-free programming. One challenge comes from the different existing instances of the same key on different tables, which can violate the hash table's semantics. Another challenge comes from the interleaving between query and movement of a key, which causes the key to be invisible to the query. We have addressed those challenges using a combination of several concurrent programming techniques, such as a two-round query protocol, lazy deletion a duplicated key, fine-grained design for the relocation process, etc. The result is, to the best of our knowledge, the first efficient lock-free implementation of cuckoo hashing in the literature, which experimentally outperforms other state-of-the-art concurrent hash tables.

Paper IV - ParMarkSplit: A Parallel Mark-Split Garbage Collector Based on a Lock-Free Skip-List: Automatic garbage collection offers safe memory reclamation for the non-blocking data structures. In a broader context, it is an important component in several programming languages and runtime systems. In the multicore era, garbage collection can take advantage of computational power brought by multicore architectures. One way to do that is to exploit the advantages of high performance lock-free data structures in the parallelization of GC. In this paper, our introduction of an extended design of a lock-free skip-list makes it natural to parallelize the mark-split garbage collection algorithm. Mark-split collects garbage in two steps: *mark* live objects and *split* memory chunks to exclude spaces occupied by the live objects. The extension means that, besides basic operations, the skip-list supports composite operations of multiple basic ones, like the *split* operation. The parallel mark-split garbage collector is implemented in OpenJDK's HotSpot virtual machine. The evaluation results show that our new design outperforms a parallel mark-split using coarse-grained locking binary search tree. Our ParMarkSplit also performs better than the HotSpot's Concurrent Mark-Sweep collector in applications satisfying certain characteristics.

Paper V - A NUMA-aware Parallel Mark-Compact Garbage Collector: The paper presents a NUMA-aware garbage collector for managed runtime environment. Current garbage collectors, in particular those in OpenJDK's HotSpot, are lacking of NUMA-awareness. This leads to performance issues such as overload of a single node, memory imbalance, and poor locality, which prevent the GCs from scaling well in NUMA machines. We have redesigned the current throughput-oriented garbage collector, including the heap layout and the parallel mark-compact collection algorithm so that it can work efficiently and scale in the NUMA multicore architecture.

1.4.1 Statement of Personal Contributions

I hereby state my personal contribution on joint-work publication.

A Study of the Behavior of Synchronization Methods in Commonly Used Languages and Systems - This paper was written under the supervision of Philippas Tsigas and Marina Papatriantafilou, and support from Daniel Cederman in technical writing. The development and evaluation of the data structures, and the writing of the technical material regarding C#, Java, and C++ programming languages, were performed by Bapi Chattejee, Yiannis Nikolakopulous, and me, respectively. We equally share in writing the remaining technical material.

Progress Guarantees when Composing Lock-free Objects - This paper was written under the supervision of Philippas Tsigas. I have designed, implemented and evaluated the synchronization mechanism. I wrote and developed most of the technical material.

Lock-free Cuckoo Hashing - This paper was written under the supervision of Philippas Tsigas. I designed, implemented, and evaluated the algorithm. I wrote and developed most of the technical material.

ParMarkSplit: A Parallel Mark-Split Garbage Collector Based on a Lock-Free Skip-List - I designed, implemented, and evaluated the ParMarkSplit garbage collector in the HotSpot environment, while Håkan Sundell contributed the lock-free skip-list implementation. I wrote most of the technical material.

A NUMA-aware Parallel Mark-Compact Garbage Collector - I am the main contributor to the paper. I studied the old generation collector, i.e Parallel Mark-Compact, and introduced NUMA-awareness to it. I have developed and implemented the collector under the supervision of Gäel Thomas, Julien Sopena, and Marc Shapiro. The NUMA-aware fragmented space policy was contributed by Lokesh Gidra. The technical material was mainly written by myself.

1.5 Conclusion

Designing efficient parallel algorithms for multicore platforms is still a challenging task for most software developers. In this thesis, we have studied different aspects in algorithmic design and implementation, composability of lock-free data structures, as well as memory reclamation in multicore programming. We have studied the influences of multiple design factors and implementation environments of synchronization methods to their behaviors and suggested guidelines to programmers in a selection of synchronization methods in the constraints of different software and hardware factors. We have found that the lock-free methods balance throughput and fairness among the involved threads. We have also discovered that progress guarantees can be compromised when composing multiple lock-free objects, and we proposed a method to regain them. Regarding the design and implementation of lock-free data structures, we introduced a new lock-free algorithm for cuckoo hashing and extended the functionality of a lock-free skip-list. The lock-free cuckoo hash table supports total concurrency among operations in a lock-free manner. It outperforms state-of-the-art concurrent hashing designs such as hopscotch and lock-based chain hashing. The skip-list design, besides providing basic operations such as search, insert, and remove, is extended to support composite operations of multiple basic operations.

As memory reclamation is an important component in lock-free programming, as well as in multicore programming in general, we have also explored this domain, especially the application of concurrent data structures to improve garbage collection. Our parallel version of the mark-split garbage collection algorithm was made possible through the use of the extended lock-free skip-list.

We explored further the new challenges to performance and scalability of the garbage collectors created by the contemporary multicore hardware with high core counts. In order to exploit the parallelism of the new hardware, the design of garbage collection algorithms must realize new design factors such as NUMA topology, memory locality, and memory balance. We have improved the current garbage collectors with such realization and introduced a NUMA-aware parallel mark-compact algorithm.

Synchronization is still a hot research topic in multicore programming. Studying high performance and scalable algorithms for more complex data structures, such as trees or graphs, is still an open research direction. Recently, processors with the hardware transactional memory (HTM) feature, e.g. Intel's Haswell, which can perform an update to a set of memory locations in one atomic step, have been introduced. HTM has increased the amount of data that can be processed in one atomic operation. Non-blocking data structures can take advantage of HTM to achieve higher performance within a simpler implementation. However, current HTM implementations like the one in the Haswell processor have limitations, such as spurious transactional abortion and no progress guarantees. Further study of using HTM for non-blocking data structures is required. As different synchronization methods are used, composability among data structures based on those methods is still an issue. The scenario in which a developer must compose data structures implemented using different blocking and non-blocking synchronization methods in one application is a reality. However, the interaction among methods has not been well studied. What kind of progress guarantees does such a composition provide? What should be expect in terms of performance?

Studying synchronization algorithms in their application contexts exposes them to new design and implementation challenges in order to meet the functionality and performance requirements. Such study can also introduce opportunities for the algorithms to evolve. Recent concerns regarding energy consumption create a new design dimension.

The NUMA multicore architecture has introduced new challenges to algorithmic design and implementation of concurrent data structures. In order to exploit the parallelism, the algorithmic design of concurrent data structures must realize new design factors. Many NUMA-aware synchronization algorithms have recently been introduced. Studying concurrent data structures for NUMA architectures can pose more challenges in both algorithmic design and implementation, which is an interesting future study. The study of garbage collection in NUMA architectures also opens more research questions. One of the issues is the influence of memory allocation patterns to the garbage collector. Another is the possibility to apply our method to bring NUMA-awareness to other garbage collectors such as the low pause Garbage-First.

Bibliography

[1] D. Abuaiadh, Y. Ossia, E. Petrank, and U. Silbershtein. An efficient parallel heap compaction algorithm. *SIGPLAN Not.*, 39(10):224–236, Oct. 2004.

[2] S. Al Bahra. Nonblocking algorithms and scalable multicore programming. *Commun. ACM*, 56(7):50–61, July 2013.

[3] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3):72–109, 1994.

[4] J. H. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 184–193, New York, NY, USA, 1995. ACM.

[5] A. W. Appel. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper.*, 19(2):171–183, Feb. 1989.

[6] C. R. Attanasio, D. F. Bacon, A. Cocchi, and S. Smith. A comparative evaluation of parallel garbage collector implementations. In *Proceedings of the 14th international conference on Languages and compilers for parallel computing*, LCPC'01, pages 177–192, Berlin, Heidelberg, 2003. Springer-Verlag.

[7] D. F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In J. L. Knudsen, editor, *Proceedings of the Fifteenth European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 207–235, Budapest, Hungary, June 2001. Springer-Verlag.

[8] H. G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, Apr. 1978.

[9] O. Ben-Yitzhak, I. Goft, E. K. Kolodner, K. Kuiper, and V. Leikehman. An algorithm for parallel incremental compaction. *SIGPLAN Not.*, 38(2 supplement):100–105, June 2002.

[10] D. Cederman, B. Chatterjee, N. Nguyen, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. A study of the behavior of synchronization methods in commonly used languages and systems. In *Proceedings of the IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 1309–1320, Los Alamitos, CA, USA, 2013. IEEE Computer Society.

[11] D. Cederman and P. Tsigas. Supporting lock-free composition of concurrent data objects. In *Proceedings of the 7th Conference on Computing Frontiers*, pages 53–62. ACM, 2010.

[12] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13:677–678, November 1970.

[13] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. *SIGPLAN Not.*, 36(5):125–136, May 2001.

[14] P. Chuong, F. Ellen, and V. Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 335–344, New York, NY, USA, 2010. ACM.

[15] C. Click. A lock-free wait-free hash table. `http://www.stanford.edu/class/ee380/Abstracts/070221_LockFreeHash.pdf`, 2007. Lecture notes in Course EE380 (2006-2007), Stanford University.

[16] J. Cohen and A. Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Trans. Program. Lang. Syst.*, 5(4):532–553, Oct. 1983.

[17] G. E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3:655–657, December 1960.

[18] N. Dang and P. Tsigas. Progress guarantees when composing lock-free objects. In E. Jeannot, R. Namyst, and J. Roman, editors, *Proceedings of the 17th International Conference on Parallel Processing (Euro-Par 2011)*, volume 6853 of *Lecture Notes in Computer Science*, pages 148–159. Springer Berlin Heidelberg, 2011.

[19] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM.

[20] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *Proceedings of the 31st ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 115–124, New York, NY, USA, 2012. ACM.

[21] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM.

[22] T. Endo, K. Taura, and A. Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, Supercomputing '97, pages 1–14, New York, NY, USA, 1997. ACM.

[23] R. R. Fenichel and J. C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12:611–612, November 1969.

[24] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 21–21, Berkeley, CA, USA, 2001. USENIX Association.

[25] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 50–59, New York, NY, USA, 2004. ACM.

[26] T. P. S. Foundation. The python standard library: Garbage collector interface. 2010.

[27] K. Fraser and T. L. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, 25(2), 2007.

[28] D. Friedman, Daniel P.; Wise. Reference counting can manage the circular environments of mutual recursion. *Information Processing Letters*, 8(1), January 1979.

[29] H. Gao, J. Groote, and W. Hesselink. Almost wait-free resizable hashtables. In *Proceedings. 18th International Parallel and Distributed Processing Symposium, 2004*, page 50a, 2004.

[30] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Trans. Parallel Distrib. Syst.*, 20(8):1173–1187, Aug. 2009.

[31] A. Gidenstam, M. Papatriantafilou, and P. Tsigas. Allocating memory in a lock-free manner. *Algorithmica*, 58:304–338, 2005.

[32] C. Gong and J. M. Wing. A library of concurrent objects and their proofs of correctness. Technical report, Computer Science Department, Carnegie Mellon University, 1990.

[33] M. Greenwald. Two-handed emulation: How to build non-blocking implementations of complex data-structures using dcas. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, PODC '02, pages 260–269, New York, NY, USA, 2002. ACM.

[34] R. H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 9–17. ACM, 1984.

[35] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Lecture Notes in Computer Science*, pages 300–314. Springer-Verlag, 2001.

[36] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, pages 124–149, 1991.

[37] M. Herlihy. A methodology for implementing highly concurrent objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.

[38] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–, Washington, DC, USA, 2003. IEEE Computer Society.

[39] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[40] M. Herlihy and N. Shavit. On the nature of progress. In *Principles of Distributed Systems*, volume 7109 of *Lecture Notes in Computer Science*, pages 313–328. Springer Berlin Heidelberg, 2011.

[41] S. V. Howley and J. Jones. A non-blocking internal binary search tree. In *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 161–171, New York, NY, USA, 2012. ACM.

[42] A. Imai and E. Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *IEEE Trans. Parallel Distrib. Syst.*, 4(9):1030–1040, Sept. 1993.

[43] Intel. Threading building blocks. 2009.

[44] H. Kermany and E. Petrank. The compressor: Concurrent, incremental, and parallel compaction. *SIGPLAN Not.*, 41(6):354–363, June 2006.

[45] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. *SIGPLAN Not.*, 46(8):223–234, Feb. 2011.

[46] R. P. LaRowe. *Page Placement For Non-Uniform Memory Access Time (NUMA) Shared Memory Multiprocessors*. PhD thesis, Duke University, Durham, North Carolina, USA, 1991.

[47] D. Lea. The java concurrency package (jsr-166). 2009.

[48] Y. Levanoni and E. Petrank. An on-the-fly reference-counting garbage collector for java. *ACM Transactions on Programming Languages and Systems*, 28(1):1–69, 2006.

[49] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983.

[50] G. Lowney. Why intel is designing multi-core processors. In *Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, pages 113–113, New York, NY, USA, 2006. ACM.

[51] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 11–20, New York, NY, USA, 2008. ACM.

[52] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3:184–195, April 1960.

[53] P. E. McKenney. Structured deferral: Synchronization via procrastination. *Commun. ACM*, 56(7):40–49, July 2013.

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

[55] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, PODC '02, pages 21–30, 2002.

[56] M. M. Michael. Cas-based lock-free algorithm for shared deques. In *Proceedings of the International Conference of Parallel Processing (EuroPar)*, volume 2790 of *Lecture Notes in Computer Science*, pages 651–660. Springer Berlin Heidelberg, 2003.

[57] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[58] M. M. Michael. The balancing act of choosing nonblocking features. *Commun. ACM*, 56(9):46–53, 2013.

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

[60] Microsoft. Garbage collection. 2010.

[61] S. Microsystems. Memory management in the java hotspot virtual machine. 2006.

[62] D. A. Moon. Garbage collection in a large lisp system. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 235–246, New York, NY, USA, 1984. ACM.

[63] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *Solid-State Circuits Society Newsletter, IEEE*, 11(5):33–35, Sept 2006.

[64] J. H. Moreno. Chip-level integration: the new frontier for microprocessor architecture. In *Proceedings of the 18th Annual ACM Symposium on Parallelism in*

Algorithms and Architectures, SPAA '06, pages 328–328, New York, NY, USA, 2006. ACM.

[65] A. Natarajan, L. Savoie, and N. Mittal. Concurrent wait-free red black trees. In *Proceedings of the 15th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 8255 of *Lecture Notes in Computer Science*, pages 45–60. Springer International Publishing, 2013.

[66] N. Nguyen, L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A numa-aware parallel mark-compact garbage collector. Technical Report 2014:04, Chalmers University of Technology, Department of Computer Science and Engineering, January 2014.

[67] N. Nguyen and P. Tsigas. Lock-free cuckoo hashing. In *Proceedings of the 34th International Conference on Distributed Computing Systems (to appear)*, ICDCS 2014, 2014.

[68] N. Nguyen, P. Tsigas, and H. Sundell. Brief announcement: Parmarksplit: A parallel mark-split garbage collector based on a lock-free skip-list. In Y. Afek, editor, *The 27th International Symposium on Distributed Computing (DISC)*, volume 8205 of *Lecture Notes in Computer Science*, pages 557–558. Springer Berlin Heidelberg, 2013.

[69] E. Petrank, M. Musuvathi, and B. Steesngaard. Progress guarantee for parallel programs via bounded lock-freedom. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 144–154, 2009.

[70] T. Printezis and D. Detlefs. A generational mostly-concurrent garbage collector. *SIGPLAN Not.*, 36:143–154, October 2000.

[71] K. Sagonas and J. Wilhelmsson. Mark and split. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 29–39. ACM, 2006.

[72] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM*, 53(3):379–405, May 2006.

[73] C.-H. Shann, T.-L. Huang, and C. Chen. A practical nonblocking queue algorithm using compare-and-swap. In *Parallel and Distributed Systems, 2000. Proceedings. Seventh International Conference on*, pages 470–475, 2000.

[74] D. Siegwart and M. Hirzel. Improving locality with parallel hierarchical copying gc. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 52–63, New York, NY, USA, 2006. ACM.

[75] H. Sundell and P. Tsigas. Noble: A non-blocking interprocess communication library. In *Proceedings of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*, LNCS. Springer Verlag, 2002.

[76] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.*, 65(5):609–627, 2005.

[77] H. Sundell and P. Tsigas. Lock-free deques and doubly linked lists. *J. Parallel Distrib. Comput.*, 68(7):1008–1020, July 2008.

[78] G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[79] J.-J. Tsay and H.-C. Li. Lock-free concurrent tree structures for multiprocessor systems. In *Parallel and Distributed Systems, 1994. International Conference on*, pages 544–549, 1994.

[80] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '01, pages 134–143, 2001.

[81] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 157–167, New York, NY, USA, 1984. ACM.

[82] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 214–222. ACM, 1995.

[83] J. D. Valois. *Lock-free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, USA, 1996. UMI Order No. GAX95-44082.

Part II

PAPERS

PAPER I

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 the proceedings of
the 27th *IEEE International Parallel and Distributed Symposium (IPDPS)*
Boston, Massachusetts USA. May 20-24, 2013
pp. 1309-1320, IEEE Press 2013.

2

PAPER I - A Study of the Behavior of Synchronization Methods in Commonly Used Languages and Systems

Abstract

Synchronization is a central issue in concurrency and plays an important role in the behavior and performance of modern programmes. Programming languages and hardware designers are trying to provide synchronization constructs and primitives that can handle concurrency and synchronization issues efficiently. Programmers have to find a way to select the most appropriate constructs and

primitives in order to gain the desired behavior and performance under concurrency. Several parameters and factors affect the choice, through complex interactions among (i) the language and the language constructs that it supports, (ii) the system architecture, (iii) possible run-time environments, virtual machine options and memory management support and (iv) applications.

We present a systematic study of synchronization strategies, focusing on concurrent data structures. We have chosen concurrent data structures with different number of contention spots. We consider both coarse-grain and fine-grain locking strategies, as well as lock-free methods. We have investigated synchronization-aware implementations in C++, C# (.NET and Mono) and Java. Considering the machine architectures, we have studied the behavior of the implementations on both Intel's Nehalem and AMD's Bulldozer. The properties that we study are throughput and fairness under different workloads and multiprogramming execution environments. For NUMA architectures fairness is becoming as important as the typically considered throughput property. To the best of our knowledge this is the first systematic and comprehensive study of synchronization-aware implementations.

This paper takes steps towards capturing a number of guiding principles and concerns for the selection of the programming environment and synchronization methods in connection to the application and the system characteristics.

2.1 Introduction

Synchronization has always been a core research problem in parallel and concurrent programming. Synchronization is required to assure the correctness of multi-threaded applications, but it can also become a bottleneck for performance. It becomes even more crucial in the multi-core and many-core era when multiprocessor computers are widely used.

Modern processors are provided with machine instructions for synchronization primitives such as *test-and-set*, *compare-and-swap* and many more. Using them, several synchronization methods have been proposed in the literature, ranging from traditional lock-based methods, such as locks, semaphores and monitors, to non-blocking approaches, such as lock-free/wait-free synchronization and software transactional memory [4, 9, 13, 18]. Building on them, programming languages can now provide built-in support for synchronization constructs as either an API in the language (Java and C#) or as user-friendly libraries (e.g. Intel TBB, NOBLE [24] or PEPPHER [3]). This means that when selecting a language to write an application in, a programmer has implicitly chosen the synchronization constructs offered by the language API or language-specific third-party libraries. In addition, selecting a programming language to use also involves several other options, which in turn have their own importance to the performance of the concurrent applications. For example, C++ offers basic memory management functionality, but also allows the programmer to access low level memory. Java or C#, on the other hand, offer automatic garbage collection, but they limit direct access to the memory. Still, even after selecting a language, the programmer has a wide range of synchronization methods to choose from. We argue that selecting the best synchronization constructs to achieve the desired behavior is a non-trivial job, which requires thorough consideration of different aspects. Besides languages and their features, the selection is also governed by several other parameters and factors, and the interplay among them, e.g. the system architecture of the implementation platforms; possible run-time environments, virtual machine options and memory management support; and the characteristics of the applications.

The implementation hardware platforms have their own role to play in this context. Although the widely available multi-core processors are mainly based on a cache-coherent NUMA design, they differ in the way they have implemented multi-threading to exploit instruction-level and thread-level parallelism. These differences are not only in the size and speed of the cache, but also in the number of threads that can share resources simultaneously [14], the memory-controller mechanism and the inter-processor connector designs that are employed on and off the chip [2]. After selecting the language, the subsequent selection of a virtual machine and/or operating system, from a wide range of options, increases the complexity of the problem even further.

There are several available synchronization methods to select from. None of them is even close to be the silver bullet which can solve all the synchronization issues that the application developers have to address in all possible hardware and software environments in the domain of concurrent programming. In the literature a number of efforts have been made to evaluate such methods through micro benchmarks [4, 16, 22] as well as macro benchmarks [21, 26, 27]. These benchmarks try to rank synchronization constructs by measuring their potential for high throughput and also examine a subspace of the parameters that we examine in this paper. Evaluating synchronization mechanisms exclusively for high throughput [6] could give misleading results. For example, consider evaluating the throughput of a simple concurrent data structure, with little or no inherent potential for concurrency, using different synchronization methods. Among the methods that give the best throughput, methods that consistently favor the same set of threads to get access to the data structure, while leaving others to starve, have the potential to rank among the best. This underpins the importance to measure fairness of the synchronization methods for a particular application.

In this paper we evaluate different types of lock-based (from fine-grained to coarse-grained), as well as lock-free, synchronization methods with regard to their potential for high throughput as well as fairness. We will focus the discussion on how these two measurements relate to each other. The studied synchronization mechanisms are applied to two different types of data structures, that have different potential for concurrency. Considering the variation in contemporary multi-core architectures, the experiments are performed on two multiprocessor machines, one with two Intel (Nehalem) processors and another with four AMD (Bulldozer) processors. Further, to explore the variation due to the choice of language and runtime, as well as memory management, we have implemented the algorithms in C++, Java and C#. To the best of our knowledge this is the first head-to-head, systematic evaluation that considers the interactions among (i) the programming language and the language constructs that it supports, (ii) the system architecture where the application is running on, (iii) possible run-time environments, virtual machine options and memory management support, and (iv) characteristics of the applications.

Our experiments put forward an interesting observation that the change in the multi-threading model at the level of architecture brings a big difference in the behavior of synchronization primitives, even though the processors have comparable speed and inter-processor connection design. Furthermore, our experiments show that high performing synchronization methods may have very poor fairness, and a wise selection is very important to make a good trade-off between the two. We also show that the choice of memory management, runtime and operating system may significantly change the performance and behavior of a concurrent application. This paper takes a step towards improving methodologies for choosing the programming environment and synchronization methods in connection to the application and the system characteristics.

The structure of the paper is the following. In Section 2.2 we go through the different synchronization mechanisms that we have examined. In Section 2.3 we discuss the concepts of fairness and throughput and how they relate to each other. Here, we also give a new quantitative measure of fairness for synchronization-aware implementations and give arguments as to why the measurement we have selected is useful in this context. In Section 2.4 we present the algorithmic designs of the data structures that were used in the experiments. Further in Section 2.5, we present the design and the setup of the experiments, as well as the detailed architectures that we have chosen for implementation. Analysis of the results is presented in Section 2.6. Section 2.7 concludes the paper.

2.2 Synchronization Methods

There exists a multitude of common methods for synchronization. These can be divided into different categories depending on what kind of progress guarantees they provide. If no progress guarantee can be provided, which is the most common case and holds true for most locks, the synchronization construct is said to be *blocking*. If a synchronization construct can guarantee that at least one thread, out of the contending set, can finish its operation in a finite number of its own steps, the construct is said to be *lock-free*. What lock-free synchronization means in practice is that a thread does not need to wait for another thread to finish.

Of this great variety of synchronization methods, some are quite popular and well established in the literature. Many of them are available through the API specification of some of the tested languages (Java, C#). Others can be easily implemented by a programmer in many languages, while some more complex ones are usually implemented in standard or third party libraries. To allow for comparison, the following synchronization methods have been implemented in a similar manner for all the programming platforms that we have examined:

- Test-And-Set-based lock (TAS) – Mutual exclusion is achieved by repeatedly trying to set a flag using an atomic exchange primitive. The thread that manages to set the flag is given access to the critical section.

- Test-Test-And-Set-based lock (TTAS) – To lower the number of expensive atomic operations, the value of the flag is read before attempting to change it. If it is already set, no atomic operation is needed.

- Array lock – The lock consists of an array of flags and an index to the first flag. Initially only the first flag is set. A thread trying to acquire the lock atomically increments the index and spins on the flag in the array that the old index was pointing to. When the flag is set, the thread can enter the critical section. Upon exiting, it sets its own flag to false and raises the flag for the thread waiting at the next index [1, 12].

- Lock-free – The lock-free implementations used depend on the specific data structures and for the cases of our study they are described in Section 2.4.

Moreover, in today's great need of concurrency, every programming environment provides their own toolset of internal libraries or implicit language constructs. They are usually well integrated and easy to use, while they can also be optimized by the underlying virtual machine or just-in-time compiler. Below them the host operating system can also provide valuable tools for synchronization. In detail, the following are the common platform specific methods for synchronization that we also consider in our study:

- Reentrant lock – The Reentrant lock, provided by Java's `concurrent.locks` package, comes in two variations; a simple and a fair one. The Reentrant lock is based on an internal waiting queue which is a variant of the CLH lock [7, 17]. Specifically, the nodes of the queue are used to block the competing threads, while every node that releases the lock that it owned signals its successor. However, an important design difference is that being the first node in the queue does not guarantee the lock acquisition, but only the right to contend for the lock itself. A thread that tries to acquire the lock first contends for it using a compare and swap (CAS) operation. If it fails, it gets enqueued. This first step is not performed when the fair version of the Reentrant lock is used. An interesting observation here is that this internal queue acts as a backoff mechanism for the lock's contention.

- Synchronized/Lock – In Java and C# every object is associated with an intrinsic monitor. The use of a `synchronized` or `lock` statement respectively, with a specified object as an argument before a block of code, assures that the execution of that critical section will not take place unless the object's monitor is locked. The actual monitor implementation is platform and virtual machine dependent [23].

- Mutex in C# – Compared to the `lock` keyword, the Mutex construct in C# is a heavyweight implementation with a high overhead, as it is designed to work across multiple processes. Mutex can be used to synchronize threads across processes and requires inter-process communications.

- Pthread Mutex (PMutex) in C++ – The Pthread mutex construct is available in the Linux kernel from version 2.6.x and above. It is implemented using Fast Userlevel Locking (Futex), created by Franke H. et al. [8]. A futex consists of a shared variable in user space indicating the status of the lock and an associated waiting queue in kernel space. In the uncontended case, acquiring or releasing a futex involves only atomic operations on its lock status word in user space. In the contended case, a system call into the kernel is required to add the calling thread to the waiting queue or to wake up any waiting processes.

2.3 Behavior: Throughput and Fairness

One of the most desired properties of a synchronization method is having high throughput. The more successful operations that can be achieved in a unit of time, the more efficient the method is. Throughput is one of the two main properties that we consider in our study.

As NUMA architectures are becoming the standard in industry, and different ways of Simultaneous Multi-Threading are being presented, fairness of synchronization constructs is becoming important. Possible differences in the access latencies of competing threads for a memory location may even lead some of them to starvation. In preliminary experiments we observed that between different architectures, under identical conditions, different levels of fairness were provided to threads that were competing for atomically swapping a memory location.

A relevant definition of fairness was introduced into this context by Ha et al. [10] comparing the minimum number of operations a thread had with the average number of operations of all threads. This helps distinguishing cases of starving or less served threads. For identifying the opposite cases we can compare the average number of operations with the maximum ones among the threads. Since our goal is to detect any unfair behavior, we use as a fairness measure the minimum of the above values, formally:

$$fairness_{\Delta t} = \min\left\{\frac{N \cdot \min(n_{i_{\Delta t}})}{\sum_i n_{i_{\Delta t}}}, \frac{\sum_i n_{i_{\Delta t}}}{N \cdot \max(n_{i_{\Delta t}})}\right\}$$

where $n_{i_{\Delta t}}$ is the number of successfully performed operations by the thread i, in the time interval Δt. Fairness index values close to 1 indicate fair behavior, while lower values imply the existence of a set of threads being treated differently from the rest. The fairness index achieves value 1 when all the threads perform equal number of operations, i.e. perfect fairness. The fairness index is 0 when at least one thread completely starves. For a critical analysis of quantitative measures of fairness, one may refer to the paper by Jain et al. [15].

2.4 Case Studies

2.4.1 Data Structures

We study the synchronization behavior of two types of data structures: FIFO queues and hash tables. They are both widely used and represent data structures with different number of contention points. The queues we are using in our case study are the lock-based and the lock-free linked list based queues introduced by Michael and Scott [20]. The lock-based queue uses locks to grant the enqueuer/dequeuer mutually exclusive access to either the head or the tail of the queue. Two locking strategies are applied to the lock-based queue: *coarse-grain* and *fine-grain* locking. The coarse-grained lock-based queue uses only one lock for both the head and the tail, while the fine-grained one uses two different locks, one for each of them. Hereafter, we refer to them as *coarse-grained queue* and *fine-grained queue*, respectively. The lock-free queue uses the CAS synchronization primitive to atomically modify the head or the tail without any locking mechanism.

The second case study is on the hash table data structure. The hash table we used is implemented as an array of buckets, each one pointing to a linked list that contains the key-value pairs which are hashed to the same bucket. The hash tables provide search, insert and remove operations. Insertion, removal or search for a key operate only on the linked list associated with the bucket to which the key is hashed to. This is where the synchronization is required. Both a lock-based and a lock-free hash table are implemented. The lock-based version has one lock for each bucket which, once locked, provide mutually exclusive access to the associated linked list. The lock-free version uses the implementation introduced by Maged Michael [19]. In this implementation, insertion of an item, i.e a node between two nodes in a linked list, is done with the help of a CAS to atomically swap the next pointer of the previous node to the new node. A thread which wants to remove a node first marks the last bit of the pointer to that node, so that other concurrent operations know its intention. Then the node is removed by using CAS, to make the previous node point to the next node. The design is proved to be correct and lock-free [19]. The reader can refer to that paper for more technical details.

2.4.2 Programming Environments

In our study of the behavior of synchronization methods, we have examined three different programming environments, C++, Java and C#.

C++ with POSIX threads

C++, prior to the C++11 standard, does not contain built-in support for multithreaded applications. Instead it relies on libraries and the operating system to provide such functionality. On Unix-like operating systems, *POSIX threads*, a.k.a Pthreads, is widely used to provide multithreaded programming support. The Pthreads library provides mutex constructs as means of implementing thread synchronization. In the C++ environment, it is possible for a programmer to *pin* a thread to a specific core, This prevents the scheduler from moving the thread from one core to another, thus avoiding unnecessary overhead. As we observed that pinning threads to cores benefited the throughput of the concurrent data structures, we applied it to all experiments in C++. We pin the threads to fill up one processor before assigning threads to the next one.

C++ provides very basic memory management functionality. Memory allocation/deallocation are done with the help of `new` and `delete`. In concurrent programming, especially lock-free programming, allocating and de-allocating memory is performed by multiple concurrent threads, which might need to be synchronized very often at runtime. Many implementations of lock-free data structures try to avoid that by using their own lock-free memory manager on top of C++ `new`/`delete`. In our context, we want to examine if user level memory management plays a significant role as a synchronization component.

Lock-free Memory Manager We have implemented a lock-free memory manager (MM) for allocating and de-allocating memory for lock-free implementations in C++. The scheme contains two parts: one main memory allocator shared by all threads and per-thread allocators. The main allocator contains a number of blocks of pre-allocated memory that it gets from the system memory. It provides blocks of memory to the per-thread allocators. Every thread has one per-thread allocator. Whenever a thread wants to allocate memory for the data structure, it gets one from the per-thread allocator. When this allocator runs out of memory, it can request new blocks of memory from the main allocator. When a block of memory is no longer used by the data structure, it will be returned to the memory block where it is allocated from, to be reused later.

This memory manager can provide fast allocation for each thread since allocating new memory usually only involves operation on its local block, which does not require synchronization. Synchronization is only needed when the thread uses up the block assigned to it and needs to allocate a new block from the main allocator.

Java

Java offers an extensive API for concurrent programming via its `concurrent` package. In addition to several standard data structures, it also includes most of the low level synchronization primitives needed, such as TAS, CAS or Fetch-And-Add. However, whether these methods actually implement the respective machine instructions or include some implicit locks, depends entirely on the implementation of Java's Virtual Machine for each architecture and operating system [23]. Also, a well specified memory model accompanies the implicit synchronization constructs of the language. Memory management has been left to Java's implicit garbage collector.

C#

The native runtime environment for C# is the .NET framework provided by Microsoft which runs exclusively on Windows. To be able to perform our ex-

Languages	C++	C#	Java
Memory management	malloc, customized	implicit memory management	
Synch. constructs & language features	PMutex	Mutex, `lock`	Reentrant, `Synchronized`
	TAS, TTAS, Lock-free, Array lock		
Contention	Low, High		
Number of Threads	2, 4, 6, 8, 12, 24, 48		
Measurement intervals (sec)	0.2, 0.4, 0.6, 0.8, 1, 2, 3, 4, 5, 10		

Table 2.1: Experimental Setup

periments in the same Linux environment used for the other languages, we have also used Mono. This is an open source runtime for the .NET framework which allows programs written in C# to be executed both on Windows and on Linux. The `System.Threading` namespace provides classes – Mutex, Monitor and Interlocked – for synchronizing thread activities. Mutex and Monitor classes represent locking synchronization whereas the Interlocked class comes with atomic primitives which can be used to create various non-blocking synchronization methods.

2.5 Experimental Setup

Our purpose is to evaluate the throughput and fairness values of the test cases in all the languages and for different contention levels. For every data structure – as discussed in Section 2.4 – we ran a set of experiments consisting each time of a different number of threads, that were concurrently competing to access the data structure. Every such experiment ran for a fixed amount of time and multiple different time intervals were used. All the different parameters of our experiments along with their values can be seen in Table 2.1. Every experiment was replicated 10 times resulting in a sample satisfying normality with $\alpha = 0.05$ level of significance (Shapiro-Wilk test). The means of these values are presented in our results. Furthermore, limited according to time and resources, samples from cases where the means were different but close were compared with ANOVA tests in order to confirm their difference, with the same level of significance.

In the queue case the operations were an enqueue or a dequeue with equal probability. Each thread was assigned the same probability distribution in all the experiment sets, across the different parameters respectively. In order to calculate the throughput value we used the 10 seconds long tests. There we counted the total number of successful operations for all the threads and divided by the exact duration on each experiment. The shorter time intervals were used for calculating the fairness index according to our definition in Section 2.3. The reason for this variety of shorter intervals, is that fairness results can be deceiving the longer an execution runs. In order to vary the contention level in the queue experiments, dummy work was introduced in every thread between the operations on the data structure.

The operations on the hash table were insert and delete with 10% probability each and search with 80% probability. Again the same probability distributions were assigned per thread in all the experiments. The fairness index this time was furthermore calculated per operation basis. The contention level was varied by changing the number of buckets, 8 for the high contention and 32 for the low.

For the implementations in Java, the IcedTea6 version 1.11.3 of the OpenJDK6 Runtime Environment was used. We ran the C# implementations using version 2.10.5 of Mono. For the C++ case GCC 4.4.1 was used. The host operating system for all of the above was based in version 3.0.0 of the Linux kernel. The C# implementation was also tested in the .NET Framework version 4.0 on Windows 7.

We performed our experiments on an Intel based workstation with 2 sockets of 6-core Xeon E5645 (Nehalem) processors with Hyper Threading (24 logical cores in total). In order to investigate how a different hardware architecture can influence the fairness values of our case studies we also performed the experiments on a second contemporary workstation. That consists of 4 sockets with AMD Opteron 6238 (Bulldozer) 12-core processors (48 logical cores in total). The processors had comparable CPU clock speeds (2.4 and 2.6 GHz respectively) and both the machines had DDR3 at 1366 MHz main memory. The Intel machine is provided with Quick-Path Interconnect for connectivity between chips and I/O subsystem, whereas, the AMD machine had Hyper-Transport for the same [11]. However, the implementation of Simultaneous Multi-Threading [28] on the two architectures differ. In an Intel (Nehalem) processor two threads can share the resources on each physical core [25], making it appear as two logical cores to the operating system. The AMD (Bulldozer) processor follows a modular architecture [5]. Here inside each module, two threads share resources other than their individual integer cores.

2.6 Analysis

In order to present, comprehend and describe the observations of the wide extent of experiments performed, a summary of the main observations regarding each of the test cases are available in Table 2.2 and 2.3. There, they are divided in common observations that stand for all the programming environments tested and then per language basis. In every type of measurement the observations are also grouped according to the most influential parameters (contention regarding throughput, architecture regarding fairness). A third column in every case exists for observations regarding the relation between throughput and fairness.

The discussion in the following subsections also follows a similar structure, namely key comments on common behavior for all the environments appear before comments regarding specific environments.

2.6.1 Queue: General Discussion

The throughput of all queue implementations in different programming languages is presented in Figure 2.1. A summary of the main observations for throughput, fairness and their relation can be found in Table 2.2.

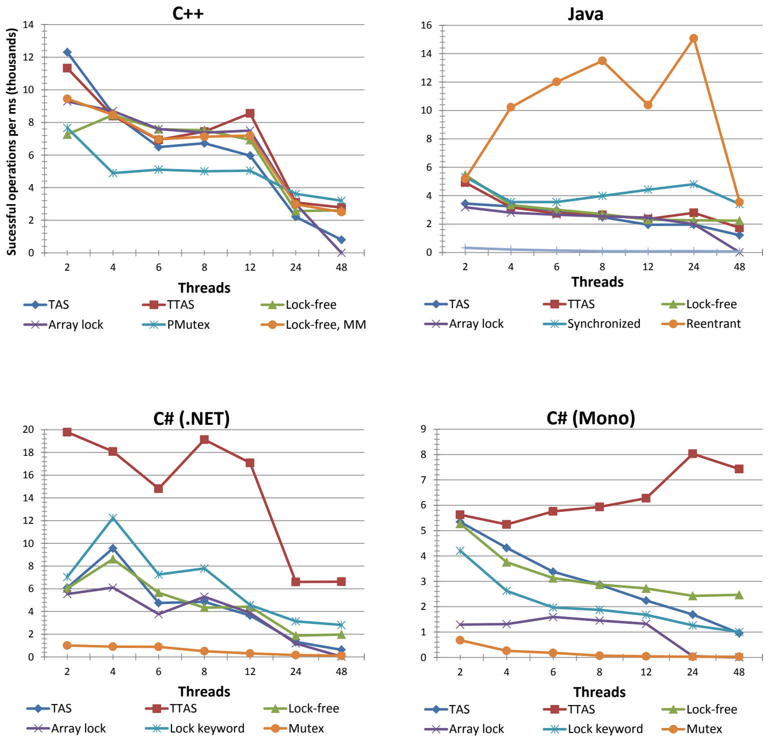


Figure 2.1: Throughput of the lock-free and fine-grained queues on the Intel system under high contention

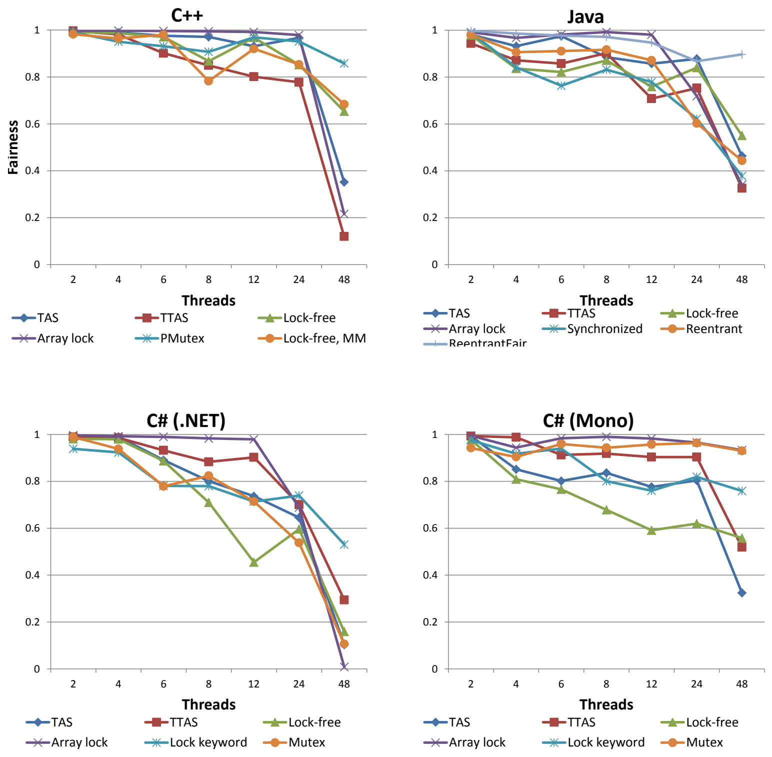


Figure 2.2: Fairness of the lock-free and fine-grained queues on the Intel system (600 ms time interval)

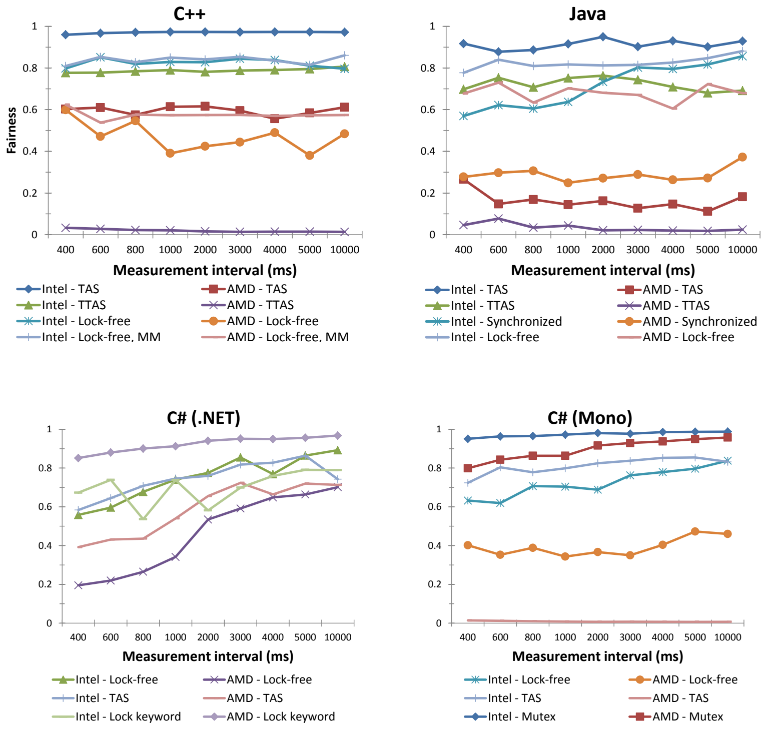


Figure 2.3: Fine-grained and lock-free queues which show major differences in fairness across platforms at 24 threads

Table 2.2: A summary of the main observations regarding the *queue* case study

Throughput	**Fairness**	**Throughput versus Fairness**
	All languages	
- Fine-grained queues perform better than the coarse-grained ones in most of the cases. - The array lock based constructions consistently achieve the worst throughput values in the multiprogramming case of 48 threads.	- Fairness deteriorates as the number of threads increases. - The fine-grained queues are almost always fairer than their coarse-grained counterparts. - When the threads are more than the hardware limit, the results of the array lock are so low that the solution can be considered as inapplicable. - In lower contention scenarios everything is fair until the contention is practically increased by the number of competing threads. The trends there are similar to the high contention cases, but with better absolute values.	- A trade-off must be made between throughput and fairness in most of the synchronization methods. - The lock-free queues in general provide a fair balance between throughput and fairness.

continued on next page

Table 2.2 – continued from previous page

Throughput	**Fairness**	**Throughput versus Fairness**
	C++	
High Contention - Steep drop of throughput values when the number of competing threads increases from 12 to 24. - TAS and TTAS based queues are among the queues which achieve the highest throughput in the cases of up to 4 competing threads. TTAS performs better as the number of threads increases. - Pmutex performs lower than the other queues between 4 and 12 threads, but scales better from 24 to 48 threads where it achieves the higest throughput value. *Low Contention* - The lock-free queue with lock-free memory management outperforms the others.	*Intel* - Below the hardware limit, most of the implementations achieve very high fairness values. - Lock-free and PMutex based queues maintain high fairness values at and above the hardware limit. - For most methods the fairness values at 8 threads are lower than those at 6 or 12. *AMD* - Fairness values deteriorate sooner than the Intel case (12 vs 24 threads). In general, the structures and locks on the AMD machine are less fair than on the Intel machine. - The array lock based queue is the most fair, with the PMutex based queue usually performing fairer than the remaining methods.	- Up to 24 threads, the TTAS lock has its throughput among the highest and its fairness among the lowest. TAS based and PMutex have the exact opposite behavior. - The inverse relation does not cover all the methods. The array lock achieves high throughput and fairness up to 24 threads and the PMutex lock gives the highest throughput and fairness at 24 and 48 threads. - The lock-free queues achieve throughput among the highest while maintaining a good, though not top, fairness. Thus they manage to provide a balance between throughput and fairness.

continued on next page

Table 2.2 – continued from previous page

Throughput	**Fairness**	**Throughput versus Fairness**
	C#	
- Throughput is consistently higher with the .NET framework compared to Mono. - The Mutex lock constructs has distinctively lower throughput than the other synchronization methods. *High Contention* - The TTAS locks has significantly higher throughput. *Low Contention* - The lock-free implementation performs better than all other methods. - the TTAS locks display lower throughput than the language provided `lock` keyword.	- The language provided lock constructs have a very high fairness measure overall. - For a low number of threads, all methods show a high degree of fairness. *Intel* In the 48 thread case the fairness drops drastically. *AMD* - The variation of fairness values along different numbers of threads is higher. - For more than 8 threads, the fairness drops by 50% for the TAS and TTAS locks. The lock-free version shows a similar trend. - For more than 12 threads, the fairness of the TAS lock drops close to zero.	- The TTAS locks provide high throughput, but low fairness. - Array locks and the language-provided lock constructs are very fair, but with low throughput. - The lock-free implementation provides a trade-off between throughput and fairness.

continued on next page

Table 2.2 – continued from previous page

Throughput	**Fairness**	**Throughput versus Fairness**
	Java	
High Contention - The constructions based on the simple Reentrant locks outperform all the rest in most of the cases. - The Synchronized based locks followed by the lock-free implementation have a scalable behavior and a relatively good throughput. *Low Contention* - The lock-free and the fine-grained queues based on TAS, TTAS and Synchronized blocks present the highest throughput. - The fair version of the Reentrant lock is the slowest except for the array lock that severely drops in 48 threads.	*Intel* - The absolute winners in most cases are the fair Reentrant lock and the array lock. - Also the TAS based queues follow closely, especially in the case of 24 threads, while in 48 the differences are widened. - The lock-free queue is the next with a similar behavior except for the 48 thread cases where it is slightly better than the TAS based queues. *AMD* - Worse fairness values for lower thread cases. - The Synchronized block, TAS and TTAS based locks are always worse than in the Intel. - For up to 12 threads the lock-free queue is fairer. After that it achieves lower values but it is still the third in order after the fair Reentrant and the array lock constructs.	- If fairness is a critical objective, then locking methods which inherently have a queue waiting structure (fair Reentrant or array lock), are definitely the choice, sacrificing throughput though. - The unfair Reentrant lock and the synchronized block give absolute throughput but fairness is not guaranteed at all. On the same side of the balance is TTAS. - The lock-free queue manages to balance this tradeoff with relatively good results in both sides.

The fine-grained queues achieve in most of the cases higher throughput than their coarse-grained counterparts. This is expected, as doubling the locks allows up to two threads to operate in parallel, one enqueueing and one dequeueing. The trends among the different lock types in the coarse-grained queues are similar comparing to the respective in the fine-grained ones. Therefore, unless explicitly mentioned, from now on all references to lock based queues will be based on the ones of the fine-grained kind.

The throughput results of lock-free and fine-grained queues of the case are presented in Figure 2.1. The constructions based on the array lock consistently achieve the worst throughput value in the case of 48 threads in all the studied programming environments. Since this is more than the number of the system's hardware threads, i.e. the hardware limit, any thread waiting in the array might be swapped out by the scheduler. This forces the remaining threads in the array to wait, until the former is swapped back in. Of course this also affects the fairness index of the method besides the throughput value. Due to the above, the results are in fact so low that we consider this solution inapplicable for this number of threads.

At first, for low numbers of threads and/or low contention, all methods show a high index of fairness. An interesting observation that occurs as the number of threads increases, and particularly in the high contention setting, is the sensitivity of the fairness values along the different time intervals. It is quite reasonable that during a small time interval even the slightest scheduling unfairness would affect the measured value. This is even more visible the more the threads are, since the one with the maximum or minimum number of operations affects less the average fairness.

The fairness experiments are also studied for the AMD system, to gain better understanding of the influence of the hardware architecture. The methods where major differences were observed are presented in Figure 2.3. We should also point out that while the 48 threads exceed the hardware limit on the Intel system, this is not the case on the AMD system, which can support up to 48 hardware threads.

2.6.2 Queue: Environment Specific Discussion

As mentioned in Section 2.4.2, in C++ the option to pin specific threads to specific processors is used. That explains the drop of throughput values showed in Table 2.2. When the number of competing threads is up to 12, our pinning strategy schedules them in one processor in a socket. When the number exceeds 12, the next 12 threads, i.e. threads number 12 to 24, are scheduled on a second processor which do not share the same L3 cache with the first one. This increases the possibility of cache conflicts among threads, which results in the throughput drop at 24 threads.

Continuing in the C++ case, the TAS based and TTAS based queues are among the queues which achieve the highest throughput in the cases of up to 4 competing threads. This advantage comes from the fact that the lock is constructed from just one atomic operation. However, as the number of threads increases, the two end points of the queue become hot spots. The cost of dealing with high contention, such as cache conflicts, becomes higher, making such simplicity less important to the throughput results. As a result, the difference in throughput between the TAS and TTAS based queues, and the remaining queues, except for the PMutex one, is relatively small when the number of threads is above 4 up to the hardware limit.

The trend of the PMutex based queue's throughput when increasing the number of threads differs from the other implementations. It is lower than the other queues for thread counts between 4 and 12, but keeps almost the same throughput value in the case of 24 and 48 threads. The internal design of PMutex based is different from the other locking methods. In contended cases, a thread goes to sleep if it fails to acquire the lock. We can observe that this mechanism, which is a form of backoff, penalizes the throughput in the cases of lower number of threads, i.e. below the hardware limit. However it helps the PMutex based queue deal with extreme contention cases, i.e. 24 and 48 threads, better than other implementations. The results show that both throughput and fairness benefit by this.

The thread pinning in specific processors also affects fairness. We observe that the fairness values at 8 threads are lower than those at 6 or 12 for most implementations. The reason is that in the case of 6 or 12 threads, all cores are scheduled to run either one or two threads, respectively. While in the case of 8 threads, some cores run one and some run two threads, which causes more fairness differences among the threads.

In Java, the throughput of the Reentrant lock and its difference from the rest is the most noticeable. This happens due to the Reentrant lock's inherent backoff mechanism – described in Section 2.2 – similar of which are not inherent in the other locks (e.g. exponential backoff). However, the overhead of the Reentrant lock's mechanism does not pay off in lower contention conditions as both versions of the lock are the lowest, with the fair one being by far the worst.

The C# implementations were tested in both Mono and the .NET Framework. The throughput results were consistently in favour of the latter. Furthermore, the low throughput of the Mutex based constructions is justified by its design, which is heavyweight due to the requirement that it should also provide interprocess synchronization. However this low throughput for Mutex, as well as for the `lock` construct, come in benefit of fairness.

2.6.3 Hash table: General Discussion

The throughput of all hash table implementations in different programming languages is presented in Figure 2.4. A summary of the main observations for throughput, fairness and their relation can be found in Table 2.3.

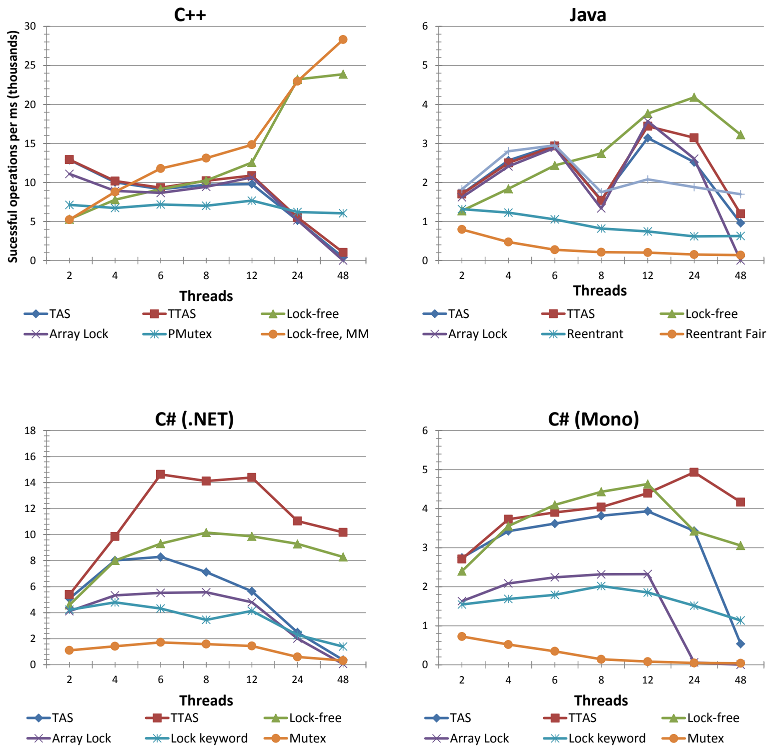


Figure 2.4: Throughput of all hash tables on the Intel system under high contention

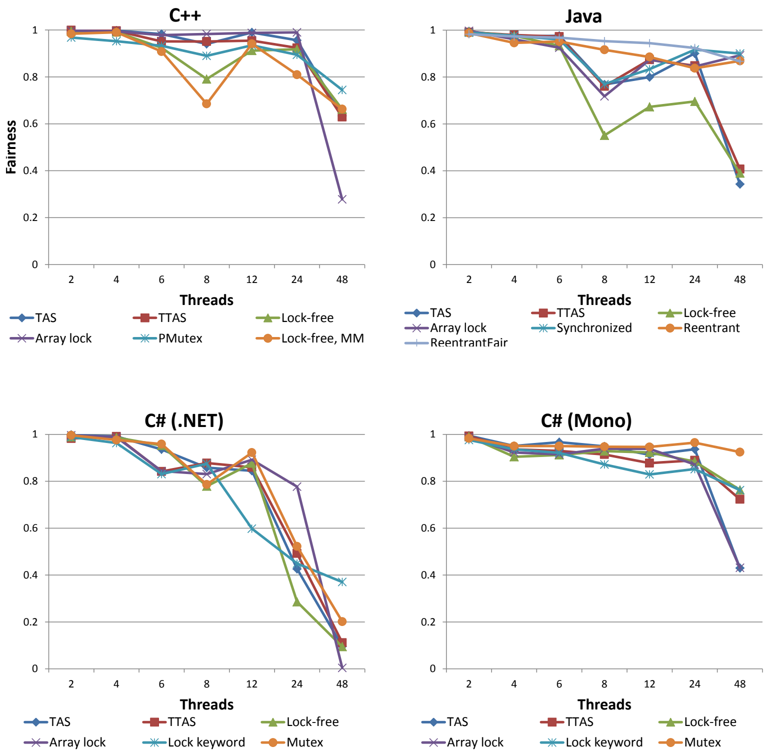


Figure 2.5: Fairness of all hash tables on the Intel system (600 ms time interval)

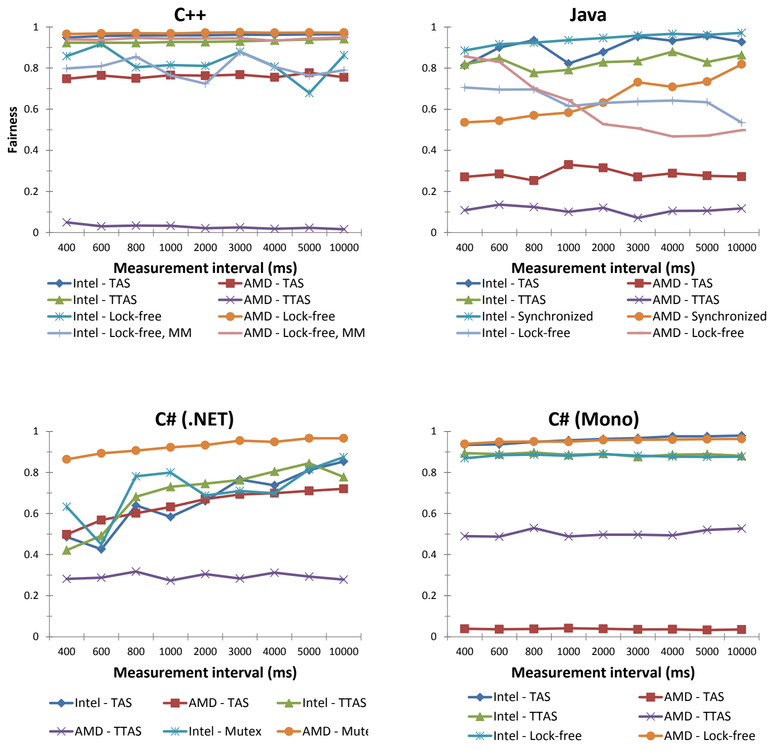


Figure 2.6: Hash tables which have major differences in fairness across platforms at 24 threads

Table 2.3: A summary of the main observations regarding the *hash table* case study

Throughput	**Fairness**	**Throughput versus Fairness**
	All languages	
- The lock-free implementations perform better than most of the lock based implementations, on average, and show scalability as well.	- The fairness indices are generally quite high. The differences become more visible when the number of threads is greater than or equal to the number of buckets.	- Different synchronization methods excel in throughput or fairness than in the queue case.

continued on next page

Table 2.3 – continued from previous page

Throughput	**Fairness**	**Throughput versus Fairness**
	C++	
High Contention - TTAS, TAS and array lock based hash tables have similar throughput values with the first one usually performing slightly better. However they do not scale beyond 12 threads where their values drop significantly. - The PMutex based hash table achieves higher throughput than the previous group beyond 12 threads, but not before. - The lock-free implementations scale all the way up to 48 threads, achieving the highest throughput values in 8 or more threads. The one with the lock-free memory manager performers better than the simple one in most of the cases. *Low Contention* - All the implementations show higher values and better scalability than in the high contention case.	*Intel* - Below the hardware limit (24 threads) all the cases behave fair except for the 8 threads case. - At and above the hardware limit the values of lock-free implementations drop lower, though still about 0.75. - TAS and TTAS lock are very unfair in the 48 threads case. However they recover and achieve high values for longer time intervals. The PMutex construction maintains high values consistently throughout all the time intervals. *AMD* - TAS and TTAS locks are heavily influenced by the change of the architecture.	- When the number of threads is larger than the number of buckets, the lock free implementations achieve high throughput but moderate or low fairness. - TAS, TTAS and to some extent array lock based hash tables show the opposite trend with high fairness and lower throughput values. - The throughput of the PMutex based hash table is usually the lowest and its fairness, though decent, is not among the top. Nevertheless, it keeps steady performance at higher numbers of threads in terms of both throughput and fairness.

continued on next page

Table 2.3 – continued from previous page

Throughput	**Fairness**	**Throughput versus Fairness**
	C#	
- The .NET implementations on Windows perform significantly better (2x - 2.5x) than the Mono implementations on Linux. - The TTAS locks perform better than other locking methods. The lock-free implementation is the one that follows. - The Mutex based locking constructs gives the lowest throughput. - The methods scale up to 8 threads, the number of buckets, and after 12 threads the throughput starts decreasing. The exception is the Mutex lock on Mono which does not scale at all. - The relative order, with respect to absolute throughput, remains largely unchanged by the change in runtime system. *Low Contention* - Increasing the number of buckets causes an increase in throughput across the board.	- For up to 6 threads, all methods are highly fair, regardless of architecture and environment. *Intel* - No single algorithm is always the most fair one for thread counts ranging from 12 to 48. *AMD* - The TTAS locks drop in fairness after 8 threads. - For more than 12 threads, the Mutex lock is the most fair.	- The TTAS lock shows high throughput, but poor fairness. - The Mutex lock is very fair, but lacks in throughput. - The lock-free and the `lock` based hash table provide a good tradeoff between throughput and fairness.

continued on next page

Table 2.3 – continued from previous page

Throughput	**Fairness**	**Throughput versus Fairness**
	Java	
Low Contention - The highest throughput is usually achieved by the hash tables based on Synchronized blocks, array locks and the lock-free ones. However in 48 threads specifically the lock-free construction keeps increasing its performance while the array lock severely drops. - Both the Reentrant locks consistently show low values. *High Contention* - The behavior is similar to the low contention case except for the lock-free hash tables which performs 20-30% lower. Despite that, when the number of threads increases it achieves the highest performance.	*Intel* - The highest fairness values are achieved by both the Reentrant locks based hash tables. Closely follows the one built on the Synchronized blocks. - TAS and TTAS and the array lock are relatively fair in most of the cases. In 48 threads they are the least fair, with TAS and TTAS improving though their values in longer time intervals. - The lock-free hash table is the least fair. *AMD* - The fairness indices of TAS and TTAS based hash tables are heavily influenced. The change also hinders, in a smaller scale, the Synchronized block construction and slightly the lock-free one.	- The lock-free method sacrifices fairness for higher throughput. - The Reentrant locks provide high fairness values without managing decent throughput. - The implementations based on TAS, array lock and the Synchronized block manage to balance the tradeoff in a very efficient manner.

The hash table is a data structure with many points where operations can be performed independently – the different buckets. Thus it allows more threads to be served concurrently and, since the keys that were used were uniformly distributed, it also allows for fairer executions. In fact we observe interesting variations of the fairness values between the different synchronization mechanisms in the cases where the number of competing threads is bigger than the number of the available buckets. Still though, concerningly low fairness values occur when the number of threads exceeds the hardware limit.

Due to the different nature of the hash table's methods, we first checked the values of the fairness index per operation, i.e. Insert, Remove, Search and also for the total number of operations regardless their kind. Since the patterns are similar, unless explicitly mentioned, the observations stand for any kind of operation.

As it can be seen in Table 2.3, different synchronization mechanisms than in the queue case have to pay the tradeoff between throughput and fairness.

In fact the pattern that can be observed is that all the synchronization methods that achieved high throughput in the low contention cases of the queue are the ones that manage the best throughput performance in the hash table. This is because the hash table consists of multiple linked lists where the hashed values are stored, i.e. the same basic component as the queue. And since the contention and the requested operations of the competing threads is now uniformly distributed along the different linked lists, the contention is lowered in each of them. Therefore the best performing solutions locally form the final result for the hash table. Similarly we can see the local fairness behaviour of the queues magnified in the total fairness index of the hash table.

2.6.4 Hash table: Environment Specific Discussion

Again in the case of C++ we can see the advantages and disadvantages of specific thread pinning to cores. While generally when the number of competing threads is less than the hardware limit, i.e. 24 threads, all the hash tables behave very fair, this observation can not be applied for the case of 8 threads. The reason is that scheduling 8 threads into 6 cores with hyperthreading causes unfairness when some cores run only one thread and the other running two. In the case of 6 or 12 threads, they are scheduled evenly to cores.

We also observe that, as the TAS- and TTAS based hash tables achieve very low throughput, even a small unfairness in the scheduling of threads can cause a negative effect on their fairness measures, especially at short time intervals. It is interesting though that the values can recover in longer time intervals.

The tradeoff between throughput and fairness appears when the number of threads is over a threshold, at which point we start to get contention at the sharing points in the data structure, i.e. the behaviour associated with the queue. These thresholds are usually at 8 and 24 threads in high and low contention scenarios, respectively. This result agrees with the fact that the hash tables have 8 or 32 buckets in each respective scenario. When the number of threads goes beyond the threshold, we see that some implementations, which achieve high throughput, might have to sacrifice the fairness. TAS- and TTAS based (and array lock based, to some extent) hash table represent this trend with high fairness, but low throughput. Lock-free hash tables also show a clear trend, but with high throughput and lower fairness results. Between the lock-free implementations with and without lock-free memory management, the former achieves higher throughput, but also gets lower fairness result than the latter, and vice versa. PMutex, the language specific construct in C++ that we tested, surprisingly does not perform well in this case study in the cases of less than 24 threads.

The different runtime systems for C# do not cause any change in the relative order of the methods as of throughput performance, but still the values in .NET are consistently higher than the ones in Mono.

Regarding fairness, no single algorithm is always the most fair for the higher numbers of threads on the Intel machine. On the contrary, considerable differences occur when changing to the AMD architecture, leaving the Mutex construct as the most fair one.

Solutions with high overhead like the Reentrant locks do not pay off for the hash table in Java either. The throughput is the lowest, however their inherent queue structure benefits fairness. More lightweight solutions manage to balance this tradeoff.

2.7 Conclusions

In this paper we evaluated different types of lock-based (from fine-grained to coarse-grained), as well as lock-free, synchronization methods with regard to their potential for high throughput and fairness.

Selecting the best synchronization constructs to achieve the desired behavior is a non-trivial task, which requires thorough consideration of different aspects. Besides languages and their features, the selection is also governed by several other parameters and factors, and the interplay among them, e.g. the system architecture of the implementation platforms; possible run-time environments, virtual machine options and memory management support; and the characteristics of the applications.

Our results show that the implicit synchronization constructs provided at the language level, for the managed languages used in our experiments, provide decent throughput and fairness for many scenarios. Much can however be gained by using more complex designs and implementations in C++, that does not rely on automatic garbage collection. This is especially true for data structures with a fine-grained design, where operations are not just simply serialized, but can actually take place concurrently. In general, it is clear that the more fine-grained the designs is, the higher the potential to achieve a higher degree of throughput, because of their high potential for parallelism. A fine-grained design also leads to increased fairness between the actors involved, as multiple operations can be performed in parallel without conflicts.

We observed that most synchronization methods show reasonable fairness and throughput when used by a low number of threads, or for scenarios with very little contention. However, when the contention increases, and the number of threads that are executed concurrently passes the number that can be scheduled on a single socket, the behaviour starts to deviate. This can be mitigated by having a data structure design that supports more parallelism, allowing for a wider choice of concurrency mechanisms. Some lock constructs, that performed poorly in queues under high contention, worked fine when used in hash tables under high contention. The cause of this is the inherent distribution of data accesses in a hash table. Methods that use backoff were shown to work very well during high contention scenarios, but the extra overhead lowered the throughput during lower contention. Some constructs such as array locks are very fair, but drops quickly in throughput when faced with increased contention. In most cases, a trade-off between throughput and fairness has to be made, no matter the language or architecture. A reasonable such trade-off for many scenarios could be made using lock-free algorithms, which in most cases manages to pair good fairness with high throughput.

More knowledge about the specific execution environment could lead to more fine-tuned decisions on which synchronization mechanism to select. Our experimental observations shed some light in this direction.

The results in this paper allows us to take a step towards improving methodologies for choosing the programming environment and synchronization methods in connection to the application and the system characteristics.

Bibliography

[1] T. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, jan 1990.

[2] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT)*, pages 319–330, New York, NY, USA, 2010. ACM.

[3] S. Benkner, S. Pllana, J. Träff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov. PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems. *IEEE Micro*, 31(5):28–41, sept.-oct. 2011.

[4] B. N. Bershad. Practical Considerations for Non-Blocking Concurrent Objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 264–274, 1993.

[5] M. Butler, L. Barnes, D. Sarma, and B. Gelinas. Bulldozer: An Approach to Multithreaded Compute Performance. *IEEE Micro*, 31(2):6–15, march-april 2011.

[6] J. Chen and W. W. III. Multi-Threading Performance on Commodity Multi-core Processors. In *Proceedings of 9th International Conference on High Performance Computing in Asia Pacific Region (HPC Asia)*, 2007.

[7] T. Craig. Building FIFO and Priority-Queuing Spin Locks from Atomic Swap. Technical report, University of Washington, Technical Report 93-02-02, 1993.

[8] K. M. Franke Hu., Russell R. Futexes and furwocks: Fast userlevel locking in Linux. In *Proceedings of the 2002 Ottawa Linux Summit*, 2002.

[9] K. Fraser and T. L. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, 25(2), 2007.

[10] P. H. Ha, M. Papatriantafilou, and P. Tsigas. Efficient self-tuning spin-locks using competitive analysis. *Journal of Systems and Software*, 80(7):1077–1090, July 2007.

[11] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 413–422, New York, NY, USA, 2009. ACM.

[12] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[13] C. A. R. Hoare. Towards a theory of parallel programming. In P. B. Hansen, editor, *The origin of concurrent programming*, pages 231–244. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[14] H. Inoue and T. Nakatani. Performance of multi-process and multi-thread processing on multi-core SMT processors. In *2010 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, dec. 2010.

[15] R. Jain, D.-M. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR*, cs.NI/9809099, 1998.

[16] A. Lamarca. A performance evaluation of lock-free synchronization protocols. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 130–140. ACM Press, 1994.

[17] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 165–171, apr 1994.

[18] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9:21–65, 1991.

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

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

[21] M. M. Michael and M. L. Scott. Relative Performance of Preemption-Safe Locking and Non-Blocking Synchronization on Multiprogrammed Shared Memory Multiprocessors. In *Proceedings of the 11th International Parallel Processing Symposium (IPPS)*, 1997.

[22] V. Nazaruk and P. Rusakov. Blocking and non-blocking process synchronization: Analysis of implementation. *Scientific Journal of Riga Technical University, Computer Science. Applied Computer Systems*, 44:145–150, 2011.

[23] Oracle. *Java Standard Edition Documentation*. 2012.

[24] H. Sundell and P. Tsigas. NOBLE: A Non-Blocking Inter-Process Communication Library. In *Proceedings of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*, Lecture Notes in Computer Science. Springer Verlag, 2002.

[25] M. E. Thomadakis. The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms. Technical report, A research report of Texas A&M University, 2011.

[26] P. Tsigas and Y. Zhang. Evaluating the Performance of Non-Blocking Synchronization on Shared-Memory Multiprocessors. *ACM SIGMETRICS Performance Evaluation Review*, 29(1):320–321, 2001.

[27] P. Tsigas and Y. Zhang. Integrating Non-Blocking Synchronisation in Parallel Applications: Performance Advantages and Methodologies. In *Proceedings of the 3rd international workshop on Software and performance*, pages 55–67, New York, NY, USA, 2002. ACM.

[28] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA*, pages 392–403, 1995.

PAPER II

Nhan Nguyen, Philippas Tsigas

Progress Guarantees when Composing Lock-free Objects

A shortened version of this paper appeared in the proceedings of
the 17th *International European Conference on Parallel and Distributed Computing (Euro-Par)*
Bordeaux, France. August 29 - September 2, 2011
LNCS Vol.: 6853, pp. 148 - 159, Springer-Verlag 2011.

3

PAPER II - Progress Guarantees when Composing Lock-free Objects

Abstract
Highly concurrent and reliable data objects are vital for parallel programming. Lock-free shared data objects are highly concurrent and guarantee that at least one operation, from a set of concurrently executed operations, finishes after a finite number of steps regardless of the state of the other operations. Lock-free data objects provide progress guarantees on the object level. In this paper, we first examine the progress guarantees provided by lock-free shared data objects that have been constructed by composing other lock-free data objects. We observe that although lock-free data objects are composable when it comes to linearizability, when it comes to progress guarantees they are not. More specif-

ically we show that when a lock-free data object is used as a component (is shared) by two or more lock-free data objects concurrently, these objects can no longer guarantee lock-free progress. This makes it impossible for programmers to directly compose lock-free data objects and guarantee lock-freedom. To help programmability in concurrent settings, this paper presents a new synchronization mechanism for composing lock-free data objects. The proposed synchronization mechanism provides an interface to be used when calling a lock-free object from other lock-free objects, and guarantees lock-free progress for every object constructed. An experimental evaluation of the performance cost that the new mechanism introduces, as expected, for providing progress guarantees is also presented.

3.1 Introduction

A concurrent data object is lock-free if it guarantees that at least one, among all concurrent operations, finishes after a finite number of steps. Lock-free data objects are immune to deadlocks and livelocks, and typically provide high scalability and performance [11] [10] [20] [22], especially in shared memory multiprocessor architectures. Several lock-free implementations of fundamental data structures have been introduced in the literature, such as queues [15] [21] [8], priority queues [18], linked-lists [23] [19] [18] [9], and hashtables [6] [17] [3]. Moreover, the problem of composing lock-free data objects has been considered recently in an effort to support the use of lock-free objects in the context of complex software development. Composite data structures, which are built by nesting multiple basic data structures, were first studied by Cohen and Campell [4]. Recently, Gidenstam et al. [7] and Cederman and Tsigas [2] studied the problem of composing two operations from two different lock-free objects into one compound atomic operation. These results made it possible to perform complex atomic operations such as *moves* that could move an item from one lock-free data object to another lock-free data object in a lock-free way.

Petrank and Steensgaard [16] also studied the problem of composing lock-free programs and services. They provided new formal definitions of lock-freedom, the bounded and unbounded lock-freedom and they extended them to programs and services. These new definitions allowed the authors to formally state and prove the composition theorem. The theorem guarantees lock-free progress for a lock-free program when composing with a service supporting lock-freedom, using the new definitions. This contribution is a step towards formally studying lock-freedom. However, the paper did not consider the case when multiple programs share a service and compete with each other to use it. This way of composing programs and services can affect their progress guarantees.

In this work, we address the lock-free composition problem but from the perspective of object-oriented programming and we do not consider changing the definition of lock-freedom in order to guarantee composition. In object-oriented programs, one lock-free object can be concurrently shared by other lock-free objects. In this setting, composition of several lock-free objects in one object is possible. When examining progress guarantees provided by these objects, we found that they can not provide the lock-free progress guarantee offered by the shared objects that compose them. To help solve this problem, a synchronization mechanism is proposed for a lock-freedom progress guarantee. By applying this mechanism when composing lock-free objects, we can compose as many objects as possible without fear of losing lock-freedom of the individual participants.

The rest of this paper is organized as follows. Section 3.2 examines the progress guarantees for lock-free objects in a composition. Then, the new synchronization mechanism for composing lock-free objects is proposed in section 3.3. Section 3.4 presents a set of experiments to evaluate our synchronization mechanism in practice. A conclusion of our work and discussions about future improvements come last in the section 3.5.

3.2 Progress Guarantees when Composing Lock-free Data Objects

This section examines progress guarantees by lock-free objects used in an object-oriented program. The program can also contain blocking objects. However, since we are considering composing lock-free objects, blocking objects can be taken away without degradation of generality. In the remainder of this paper, all objects mentioned are lock-free.

```
class LF
  word *ptr
  public op(args)
   while (true)
    oldVal ← *ptr
    newVal ← calculate(args)
    if (CAS(ptr, oldVal, newVal))
      return
```

Algorithm 3.1: A template of a lock-free object

3.2.1 Lock-free Implementations of Data Objects

Lock-free objects are objects that provide lock-free progress guarantee for their operation executions. The guarantee ensures that some among its concurrent operations succeed after a finite number of steps of their own execution. To provide such a guarantee, lock-free objects usually use non-blocking synchronization primitives to synchronize concurrent accesses to shared memory among the concurrent operations. Two synchronization primitives that are commonly used are Compare-And-Swap (*CAS*), Load-Link/Store-Conditional (*LL/SC*). *CAS* [11] takes three arguments: an address, an expected value, and an update value. If the value at the address is equal to the expected value, it is replaced by the update value; otherwise the value is left unchanged. *LL/SC* is a pair of instructions. The *LL* instruction reads from an address. A later *SC* instruction attempts to store a new value at the address. The instruction succeeds if content of the address are unchanged since that thread issued the earlier *LL* instruction to it. The instruction fails if the content has changed in the interval. These instructions are equally powerful since they both have an infinitive consensus number [11].

By observing several lock-free implementation of fundamental data structures such as queues [15] [21], linked-lists [23], and memory allocators [14], we found a common template that most of these implementations followed presented in Algorithm 3.1. The template object *LF* offers one operation *op*, which takes generalized arguments *args*. This operation computes a *newVal* (line 6) and updates it to *ptr* variable. In a multi-threaded environment, several threads can try to update *ptr* concurrently. Therefore, the *CAS* primitive is used to keep each update atomic. Examples of an *LF* object and an operation *op* that it supports are a lock-free $Queue$ [15] and its *enqueue* operation, respectively. The *enqueue* operation creates a new node containing the new value and inserts it to the $head$ of the queue (by a *CAS*) to become the new $head$ node.

3.2.2 Examining Lock-free Progress Guarantees in Object-Oriented Programs

An object-oriented program comprised by three lock-free objects is examined as an example. Among the objects, one, O_{21}, is concurrently shared by the other objects: O_{11} and O_{12}. All are assumed to be implemented by using the above template.

During the executions of O_{11} and O_{12}'s operations, they invoke operations in O_{21} and wait for the returned results. Object O_{21} is lock-free and therefore, always has some executed operations, invoked by O_{11} or O_{12}, finish and return after a finite number of executed steps. But, O_{21} provides no mechanism to ensure fairness among the executions invoked by different objects. As a result, that only executed operations called by one object (e.g O_{11}) succeed while those called by the other object fail to succeed is possible. Consequently, the former object progresses while the latter does not and fails to provide lock-freedom. So, composition causes a lock-free conflict point at O_{21} for O_{11} and O_{12}. When it is the case, lock-freedom of objects that conflict can be violated.

This lock-free conflict concept can be generalized. There can be several objects sharing another object. An object sharing another object can also be shared by other objects and become itself a conflict point. This sharing scenario creates a hierarchy of sharing lock-free objects together with the respective hierarchy of lock-free conflicts..

Our objective is to introduce a new synchronization mechanism enhancing the shared object so that it supports the lock-free property of the sharing objects.

3.3 A Synchronization Mechanism for Composing Lock-free Objects

3.3.1 Overview of Our Proposed Approach

A new synchronization mechanism for sharing lock-free objects is proposed. Application of this mechanism enhances objects with the capability to maintain fairness among all the objects that invoke its operations. This fairness ensures that any invoking object has at least one operation returned after a finite number of steps. In other words, no object starves because of performing operations at the shared object.

In detail, the proposed synchronization mechanism keeps track of all invocations by sharing objects to the shared object's operations. When those by an object are unsuccessful to execute the instruction(s) at the linearization point many times, the mechanism will announce one of the operations. When such an announcement is made, later invocations help finish the announced operation before performing their expected operations. Completion of the announced operation allows the sharing object to progress.

The description of the proposed synchronization mechanism are introduced in the two next subsections. A correctness proof for the mechanism is also presented.

3.3.2 The Operation Descriptor

The new synchronization mechanism is introduced so that an unfinished operation can be helped to finish. The operation can be executed by more than one thread but the mechanism guarantees that only at most one execution can successfully complete. To make this helping scheme possible, a description of the operation and its execution status is needed. Any thread can read the description and execute the operation it describes.

The data structure *OpDesc* illustrated in Algorithm 3.2 is such an operation descriptor. *OpDesc* contains a function pointer **oper* to the operation, along with arguments for the operation; a boolean variable *done* records the status of the operation (finished or unfinished); *src* is a unique identity of the object that invokes this operation.

An $OpDesc$ object encapsulates an operation (e.g $enqueue$ operation) provided by shared lock-free object. The mechanism introduces a special kind of operation which can help executing other operations. In other words, operations that can read $OpDesc$ and execute the operation it described. We call them “super-operations”. The term “operation”, from this point, refer to an operation representing functionality that other objects want to perform at the shared object, which is described as an $OpDesc$ object.

3.3.3 The Synchronization Mechanism

The implementation of our synchronization mechanism for the lock-free object *LF* is presented in Algorithm 3.2. The new object *CLF* provides the same interface as that *LF* does to other objects. However each method in the interface is associated with a super-operation instead of an operation.

Any operation op in LF is re-written into a pair of one public method op (a super-operation) and one private one op_m (an operation). The operation *CLF.op_m* executes steps to make changes to the *CLF* object similar to that *LF.op* does to the *LF* object. The difference between *CLF.op_m* and *LF.op* is additional steps required by the synchronization mechanism that will be discussed later. *CLF.op*, is to provide the same interface as that *LF* but the content is to-

```
struct OpDesc
  void *oper(void *args)
  void *args
  bool done
  Object src

class CLF
  word *ptr
  OpDesc hlps[M], EMPTY;      //EMPTY.done=true

  public op(src, args)
   OpDesc me(src,&op_m, (void*)args), hlp
   for(int i ← 0; i < M; i++) {
     hlp ← hlps[i];
     hp_x ← hlp;             //protect hlp with hazard pointer
     if (hlp != hlps[i]) continue;
     if (!hlp.done) *hlp.oper(me, hlp)
   if (¬me.done) op_m(me, me)

  private op_m(OpDesc me, OpDesc hlp)
   while (¬hlp.done)
     for (tries=0; tries<$T_{MAX}$∧¬hlp.done; tries++)
       oldVal ← *ptr
       newVal ← calculate(hlp.args)
       tmp ← hlps[hlp.src]
       if (DCAS(ptr, oldVal, newVal, &hlp.done, false, true))
         counter[hlp.src] ← 0;
         CAS(hlps[hlp.src], tmp, EMPTY);
         break;
     if (¬hlp.done)
       if (++counter[me.src]≥$O_{MAX}$)
         announce(me)

  void announce(OpDesc me)
    curr ← hlps[me.src]
    if(curr.done)
      CAS(hlps[me.src], curr, me)
```

Algorithm 3.2: Implementation of our synchronization mechanism

tally new. When *CLF.op* is invoked, it is expected to perform modifications on *CLF* similar to functionality of operation $LF.op$. The functionality is now implemented in $CLF.op\_m$. In addition, *CLF.op* can help finish other *CLF.op_m* operations that other objects want to perform.

When *CLF.op* is invoked (assuming by object O_i) to perform the operation $CLF.op\_m$, it does not perform the operation immediately. Instead, it first creates an $OpDesc$ describing the operation (line 20) which it can perform by itself (line 26) or any thread can help finishing the operation. Then it checks if there are operations of any object needing help to finish (line 21). If there are such operations, the super-operation will execute these operations (line 25). The checking for any object that needs help is performed through a newly introduced array *hlps[]*. When one among the objects needs help, one of the concurrent operations the object performs will be placed in *hlps[]* at a dedicated position for the object. Other concurrent super-operation executions then can help to finish that one. We assume that there are M objects sharing *CLF* object. Therefore, $hlps[]$ can have M elements that one is assigned to an object.

The operation *CLF.op_m* introduces two main changes compared to *LF.op*. The first change is that a Double-Compare-And-Swap (*DCAS*) is used instead of a *CAS* in *LF.op* (line 7). *DCAS* atomically compares and exchanges values at two separate memory locations. Lock-free implementations of *DCAS* have been introduced in [5] and [2]. In $CLF.op\_m$, the *DCAS* performs modification of **ptr* and a status variable atomically. The former is similar to *CAS* in *LF.op*. The latter is to set the execution status variable of $OpDesc$. This status variable, which is allowed to be changed only once, makes sure that an $OpDesc$ only succeeds once even when multiple threads are executing it.

The second change in *CLF.op_m* is the introduction of a counter array $counter[]$ to record the numbers of times invocations by sharing objects try (but fail) to commit the changes to the shared object *CLF*. The counter at position i is increased after a failed *DCAS* execution (line 34) in an operation invoked by object O_i. When this number reaches a threshold, an executed operation invoked by O_i will be announced in *hlps[]* to be helped.

Due to this change, the loop inside this operation is also modified. Our algorithm could have followed the idea of increasing the counter after every failed *DCAS*. In this case, the counter at any position would be shared among several threads and need synchronization for every update which decreases the performance. To avoid this high overhead, in our design, this counter was split into two counters. One local counter *tries* for each operation execution and a shared one ($counter[]$) to record number of tries the executions invoked by the object have made. When $tries$ reach a threshold T_{MAX}, an update to *counter[me.src]* is made. And if this counter reaches its threshold O_{MAX}, one of the operation executions whose *src* is the same as *me.src* is announced.

In addition to those changes, a *CAS* is added to remove the reference from the announcement array $hlps[]$ to a successful operation hlp. This avoids any unsafe reference to hlp in the future when its hazard-pointer protection (line 23) is removed. The memory used by hlp can safely be reclaimed later by a memory reclamation scheme.

In short, the synchronization mechanism guarantees that new invocations of *CLF*'s operations helps finish on-going executed operations that need help. Then they executes the operation they are supposed to perform. With this mechanism, objects invoking operations of *CLF* always has one of the invocations finish after a finite number of steps. Therefore, these objects make progress.

3.3.4 Addressing the ABA Problem

Similar to other lock-free implementation based on CAS, our mechanism also encounters the ABA problem. The ABA problem happens when the content at an address changes from A to B, and then changes back to A. *CAS* cannot distinguish this case and the case where the content is unchanged. A number of methods have been introduced to tackle the ABA problem such as tagging [12], hazard pointers [13]. In addition, memory words used by lock-free objects must be protected from deletion by concurrent threads when they are in use and reclaimed when they are no longer used. We use Safe Memory Reclamation powered by hazard pointers introduced in [13] to safely manage memory and ABA prevention.

3.3.5 Linearizability

We are presenting the proof for the linearizability and lock-freedom property of *CLF*.

Lemma 3.1. *Regardless of the number of threads executing an operation op_m with the same value of hlp argument, only one can succeed.*

Proof. The DCAS has been proved for its linearizability in [2]: when there are multiple threads that have the possibility to modify the same ptr, only one thread succeeds to change its value from $oldVal$ to $newVal$, and $hlp.done$ from $false$ to $true$. When an $OpDesc$ me is executed by the thread which created it or other threads which are helping it, only one thread can successfully finish the DCAS to change the corresponding ptr. Threads that fails executing the $DCAS$ exit the loops in op_m when they realize the operation has been completed ($hlp.done = true$). □

Lemma 3.2. *CLF is linearizable with the linearization point at line 34.*

Proof. When *op* is executed by a thread t, it first creates an OpDesc object $me = X$ to describe operation op_m that it wants to perform (line 23). If t executes X at line 26 and successfully performs $DCAS$ at line 34, it is the linearization point of *op*. If another helping thread manage to successfully help X (line line 25), the linearization point of the operation is also at line 34 which is executed by the helping thread. When the helping thread is doing that, t is either (i) busy helping another operation (line 25, or (ii) t is also execute X but fails in competing to finish the $DCAS$ with the helping thread. In either case, t knows that its operation me is completed by checking the condition at line 31 and returns. □

Lemma 3.3. *The presented object CLF is lock-free.*

Proof. When *op* is invoked, it creates a descriptor me, then it (i) invokes op_m to help any hlp which needs help (line 25), (ii) invokes op_m to execute me (line 26). Which means there are multiple threads executing op_m. If we assume that the maximum number of threads in the system is N, at a certain point there are

$N_1 < N$ threads executing op_m as helping threads and $N_2 \leq N$ threads executing op_m to finish their own operation; and $N_1 + N_2 \leq N$. For all threads reaching line 34 and compete to finish the DCAS, at least one thread t, succeeds and finish op_m operation this thread is executing returns.

If t is among the N_2, it finishes its execution in *op* and returns. However, t is among the N_1 helping threads, t continues the execution in *op* to execute its operation at line 26. Now, t join the set of N_2 threads. After at most $N_1 + 1$ competing rounds at DCAS, there must be at least one among N_2 threads that succeeds in performing DCAS. This leads to the completion of an operation *op*. In short, there is always one *op* operation in object CLF that finishes after at most N DCAS competing rounds. $\square$

3.3.6 Progress Guarantees

When a lock-free object is concurrently used by other lock-free objects $O_1 \ldots O_M$, it can become a lock-free conflict and block the progress of those objects. This section will prove that when there is such a conflict point at *CLF*, our mechanism can resolve the conflict. Therefore, *CLF* does not block lock-free progress of the objects using it.

A scenario of using *CLF* is a program containing M lock-free objects $O_1 \ldots O_M$ and one *CLF* object. An object O_i can have at most n concurrent invocations (executed by n threads) to *CLF.op* to perform an intended *CLF.op_m* (referred to as *me*). Each invocation creates an execution of operation *CLF.op*. We seek a bound of the maximum number of steps (a step is one execution of *DCAS*) performed by these executions between any two successful operations. If this bound is finite, it guarantees that any object that uses *CLF* progresses. The lemmas and theorem below figure out this bound.

Lemma 3.4. *An object O_i can make at most n concurrent invocations to super-operation $CLF.op$. Starting from when the last invocation returns (or when the program starts, if there is no such invocation), if any of these invocations has executed:*

$$U\_BOUND = T_{MAX} . O_{MAX} \tag{3.1}$$

steps, one of the following condition must hold:

- *at least one invocation finished. Or*
- *one of these concurrent CLF.op_m operations has been announced.*

where a step is a modification at the shared variable ptr by the DCAS.

Proof. Considering a thread executing op which is invoked by an object O_i, the thread eventually executes op_m to complete its own operation or to help another operation. For every T_{MAX} steps, the thread exit the for loop in op_m because the condition $tries < T_{MAX}$ is satisfied. If the operation hlp that the thread is trying to complete is not done, either by itself or by any other helping thread, the thread increases the $counter[O_i]$, but to a value no more than O_{MAX}. Which means, after the thread has executed at $T_{MAX}.O_{MAX}$ steps and hlp still has not completed yet, it announces itself me at line 40. □

Lemma 3.5. *When an operation me is announced in hlps, either me or another operation that has the same src as $me.src$ finishes after it has executed at most*

$$HELP\_BOUND = n(M-1)+1$$

steps since when the announcement is made.

Proof. This lemma is proved by contradiction. After me is announced and has executed $n(M-1)$, if no operation whose $src = me.src$ succeeds, all operations invoked by objects other than $me.src$ have completed. At this point, the executing operation are either those which are invoked by the object $me.src$ or those which are helping me. One of them succeeds after me has executed $n(M-1)+1$. □

Theorem 3.1. *When CLF is shared by several objects by invoking to CLF's super-operation op, there is always one, among all invocations by one object, finishing after executing a finite number of steps.*

Proof. From lemma 3.4, there must be one among the invocations from O which finishes before any of them has executed U_BOUND steps. Otherwise, one of the invocations has its operation me announced.

If me is announced, lemma 3.5 stated that one of the operations whose src is the same as $me.src$ (including me) finishes after it has executed at most $HELP\_BOUND$ steps since the announcement is made. Therefore, one of the invocations from one object returns after executing at most:

$$U\_BOUND + HELP\_BOUND = n(M-1) + T_{MAX}.O_{MAX} + 1$$

steps; where:

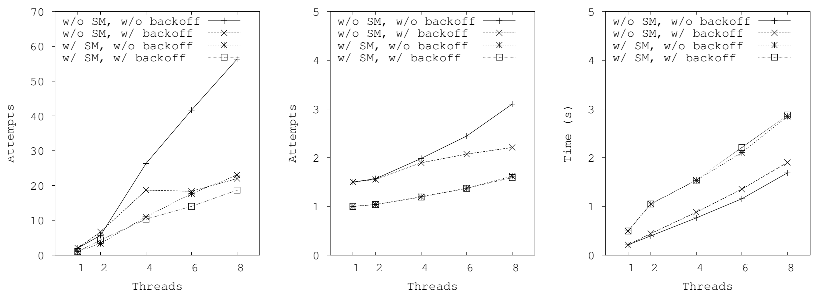
- T_{MAX} is the number of steps executed by an operation before it checks if it should announce itself.
- O_{MAX} is the number of times T_{MAX} was reached by all invocations from one object.
- n is the maximum number of concurrent operations of *CLF* that can be executed.
- M is the number of objects that are sharing *CLF*.

□

3.4 Experimental Evaluation

For our experimental evaluation we considered the composition scenario where a program containing a number of pseudo objects sharing one queue. The queue is an implementation of the Michael-Scott Queue [15] enhanced with the proposed synchronization mechanism. A set of experiments to evaluate the effectiveness and performance cost of our synchronization mechanism was performed and the results are presented.

In our experiments, the program was executed to perform queue's operations at three contention levels. In high contention, each thread performed one

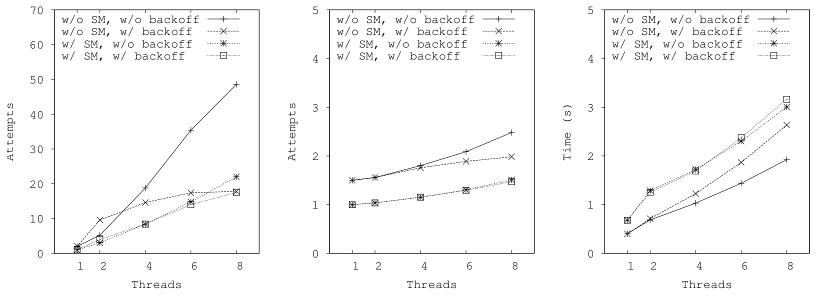


(a) attempts(max)/op (b) attempts(avg)/op (c) Execution time

Figure 3.1: Measurement results in high contention level

operation right after another. In medium contention, “other work” with a ratio following the normal distribution between 0 and 1 was performed between two consecutive operations. The “other work” was a fixed-times spin loop of a simple calculation. In low contention, “other work” was always performed between two consecutive operations. An exponential back-off was also used after any failed *DCAS*. The program can be run by one to 8 threads and each thread performs 1 000 000 queue operations. Each experiment is the program configured to one contention level and with or without back-off, and set up with a specific number of threads. Each experiment ran five times on a platform with two Intel Core i7 quad-core processors and the average result of the runs was reported. When running the experiments, no other users were using the system.

Three measurements were recorded. The first two were the maximum and average number of attempts between two consecutive successful operations invoked by one object. The maximum number of attempts is an indicator to know whether the proposed synchronization mechanism helped the sharing objects before they starved. The lower this number, the more likely an object is to be helped. On the other hand, the average number of attempts, helps answer a question: does the synchronization mechanism cause the total number of attempts to perform the set of operations increasing? The third measurement was the time it took to finish a run.

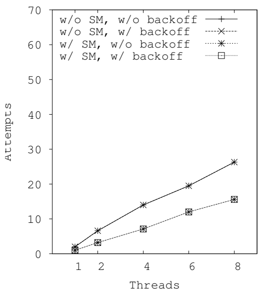


(a) attempts(max)/op (b) attempts(avg)/op (c) Execution time

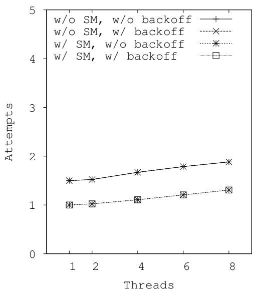
Figure 3.2: Measurement results in medium contention level

Figure 3.1 presents the experimental results for the case of high contention. Figure 3.1a shows that our synchronization mechanism (w/ SM) significantly reduced the maximum number of attempts to finish one operation when there was no back-off. In the case where no synchronization mechanism was used (w/o SM), the maximum number of attempts when back-off is used (w/ back-off) is much lower than when it is not (w/o backoff). The reason is that back-off reduces the contention among threads and, therefore, lowers the number of attempts. Even though, in this case, there is no lock-free progress guarantee for the sharing objects. The average number of attempts in fig. 3.1b shows that when our synchronization mechanism is used, one queue operation needs, on average, about only two thirds of the number of attempts compared to when it is not used. Similar improvements when the synchronization mechanism was used are also observed in medium and low contention levels as shown in figs. 3.2a and 3.2b and figs. 3.3a and 3.3b.

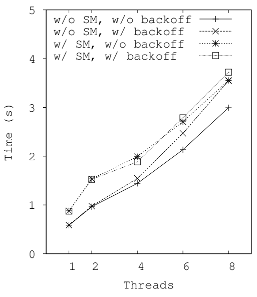
Figure 3.1c shows the time to finish all operations at high contention level. Either with or without back-off, the execution time of the runs where our synchronization mechanism was used took about 1.7 of those where the original queue is used. This degradation in performance is because of the overhead cost when applying our synchronization mechanism to achieve the lock-freedom property. In medium and low contention levels, our synchronization performed



(a) attempts(max)/op



(b) attempts(avg)/op



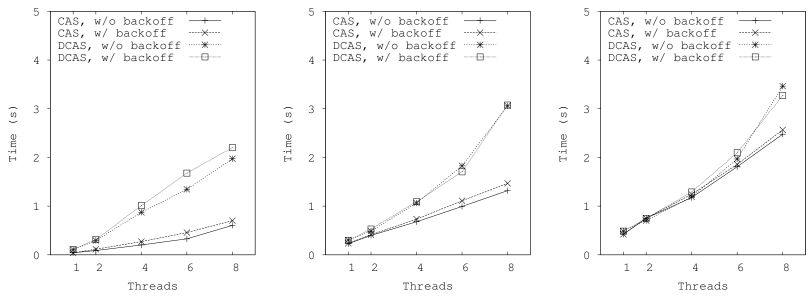
(c) Execution time

Figure 3.3: Measurement results in low contention level

better which reduced the ratios to 1.5 (fig. 3.2c) and 1.2 (fig. 3.3c) respectively. Especially, in low contention level with back-off, the performance of the queue where our synchronization was used is closer to that when it was not used. Our synchronization mechanism performed better in these contention levels than in high contention levels. This is consistent with the previous result that fewer attempts were performed to finish one queue operation in lower contention level. In addition, when the number of attempts were fewer, the number of cases that the synchronization mechanism was activated to help "unlucky object" were fewer too.

We performed additional experiments to analyze the overhead cost by measuring the performance of *DCAS* comparing to that of *CAS*. The experimental setup was similar to the one described in previous experiments. The only difference was that the queue operations were replaced by an operation containing a simple mathematical calculation and a *DCAS* (or *CAS*). The performance result in fig. 3.4 shows that *DCAS* is much more expensive than *CAS* especially in high and medium contention levels. In low contention level, execution time of a *DCAS* operations is quite comparable to that of a *CAS*. These results support a claim that *DCAS* contributes a big portion to the overhead cost of our synchronization mechanism.

In brief, the experimental results demonstrate that our synchronization mech-



(a) High contention (b) Medium contention (c) Low contention

Figure 3.4: Performance of DCAS and CAS

anism reduces the maximum number of attempts in all the contention level cases. The presented experimental results support the theoretical proofs. The results also show, as expected, that there is a performance overhead cost in order to achieve lock-freedom when composing. The software-implemented *DCAS* mainly contributes to this cost. We expect that with the use of a hardware-supported *DCAS* such as the Advanced Synchronization Facility by Advanced Micro Devices [1], this cost will be reduced significantly.

3.5 Conclusions

This paper presents our observation on progress guarantees provided by lock-free objects that concurrently share other lock-free objects. We found that these sharing objects can not provide lock-free progress guarantee as expected. A new synchronization mechanism for composing lock-free objects is proposed in order to provide lock-free progress guarantees for each individual. The experimental results show the effectiveness of the new mechanism. A preliminary study for the performance cost introduced by the new mechanism is also presented.

The assumption of the fixed number M of sharing objects should be studied further and if possible removed. Additional experiments can be performed to investigate the influence of choosing T_{MAX} and O_{MAX} on the performance of the mechanism. In addition, an implementation of the mechanism that uses a hardware-supported *DCAS* such as Advanced Synchronization Facility by Advanced Micro Devices is expected to reduce the performance cost.

Bibliography

[1] AMD. *Advanced Synchronization Facility - Proposed Architectural Specification*. Number 45432/rev 2.1. AMD, 2009.

[2] D. Cederman and P. Tsigas. Supporting lock-free composition of concurrent data objects. In *Proceedings of the 7th Conference on Computing Frontiers*, pages 53–62. ACM, 2010.

[3] C. Click. A lock-free wait-free hash table. `http://www.stanford.edu/class/ee380/Abstracts/070221_LockFreeHash.pdf`, 2007. Lecture notes in Course EE380 (2006-2007), Stanford University.

[4] D. Cohen and N. Campbell. Automatic composition of data structures to represent relations. In *Proceedings of KBSE 1992*, pages 182–191, sep. 1992.

[5] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007.

[6] H. Gao, J. Groote, and W. Hesselink. Almost wait-free resizable hashtables. In *Proceedings. 18th International Parallel and Distributed Processing Symposium, 2004*, page 50a, 2004.

[7] A. Gidenstam, M. Papatriantafilou, and P. Tsigas. Allocating memory in a lock-free manner. *Algorithmica*, 58:304–338, 2005.

[8] A. Gidenstam, H. Sundell, and P. Tsigas. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In *OPODIS*, volume 6490 of *Lecture Notes in Computer Science*, pages 302–317. Springer, 2010.

[9] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Lecture Notes in Computer Science*, pages 300–314. Springer-Verlag, 2001.

[10] M. Herlihy. A methodology for implementing highly concurrent objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.

[11] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[12] IBM. *IBM System/370 Extended Architecture, Principles of Operations*. Number SA22-7085. IBM Publication, 1983.

[13] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[14] M. M. Michael. Scalable lock-free dynamic memory allocation. *SIGPLAN Not.*, 39(6):35–46, 2004.

[15] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, 1996.

[16] E. Petrank, M. Musuvathi, and B. Steesngaard. Progress guarantee for parallel programs via bounded lock-freedom. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 144–154, 2009.

[17] C. Purcell and T. Harris. Non-blocking hashtables with open addressing. In *Proceedings of the 19th International Conference on Distributed Computing*, DISC '05, pages 108–121, 2005.

[18] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.*, 65(5):609–627, 2005.

[19] H. k. Sundell and P. Tsigas. Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. *Principles of Distributed Systems*, 3544:240–255, 2005.

[20] P. Tsigas and Y. Zhang. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. *SIGMETRICS Perform. Eval. Rev.*, 29:320–321, June 2001.

[21] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '01, pages 134–143, 2001.

[22] P. Tsigas and Y. Zhang. Integrating non-blocking synchronisation in parallel applications: performance advantages and methodologies. In *Proceedings of the 3rd International Workshop on Software and Performance*, WOSP '02, pages 55–67, 2002.

[23] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 214–222. ACM, 1995.

PAPER III

Nhan Nguyen and Philippas Tsigas

Lock-free Cuckoo Hashing

To appear in the proceedings of
the 34th International Conference on Distributed Computing Systems (ICDCS)
Madrid, Spain. June 30 - July 3, 2014.

4

PAPER III - Lock-free Cuckoo Hashing

Abstract
This paper presents a lock-free cuckoo hashing algorithm; to the best of our knowledge this is the first lock-free cuckoo hashing in the literature. The algorithm allows mutating operations to operate concurrently with query ones and requires only single word compare-and-swap primitives. Query of items can operate concurrently with others mutating operations, thanks to the two-round query protocol enhanced with a logical clock technique. When an insertion triggers a sequence of key displacements, instead of locking the whole cuckoo path, our algorithm breaks down the chain of relocations into several single relocations which can be executed independently and concurrently with other op-

erations. A fine tuned synchronization and a helping mechanism for relocation are designed. The mechanisms allow high concurrency and provide progress guarantees for the data structure's operations. Our experimental results show that our lock-free cuckoo hashing performs consistently better than two efficient lock-based hashing algorithms, the chained and the hopscotch hash-map, in different access pattern scenarios.

4.1 Overview

A hash table is a fundamental data structure which offers rapid storage and retrieval operations. Hash tables are widely used in many computer systems and applications. Papers in the literature have studied several hashing schemes which differ mainly in their methods to resolve hash conflicts. As multi-core computers become ubiquitous, many works have also targeted the parallelization of hash tables to achieve high performance and scalable concurrent ones.

Cuckoo hashing [12] is an open address hashing scheme which has a simple conflict resolution. It uses two hash tables that correspond to two hash functions. A key is stored in one of the tables but not in both. The addition of a new key is made to the first hash table using the first hash function. If a collision occurs, the key currently occupying the position is "kicked out", leaving the space for the new key. The "nestless" key is then hashed by the second function and is inserted to the second table. The insertion process continues until no key is "nestless". Searching for a key involves examining two possible slots in two tables. Deletion is performed in the table where the key is stored. Search and delete operations in cuckoo hashing have constant worst case cost. Meanwhile, insertion operations with the cuckoo approach have been also proven to work well in practice. Cuckoo hashing has been shown to be very efficient for small hash tables on modern processors [16].

In cuckoo hashing, the sequence of the evicted keys is usually referred to as "cuckoo path". It might happen that the process of key evictions is a loop, causing the insertion to fail. If this happens, the table needs to be expanded or rehashed with two new hash functions. The probability of such insertion failure is low when the load factor[1] is lower than 0.49 but increases significantly beyond that [12]. Recent improvements address this issue by either using more hash functions [4] or storing more than one key in a bucket - known as *bucketized cuckoo hashing* [13] [7].

A great effort has been made to build high performance concurrent hash tables running on multi-core systems. Lea's hash table from Java Concurrency Package [10] is an efficient one. It is a closed address hash table based on chain hashing and uses a small number of locks to synchronize concurrent accesses. Hopscotch hashing [6] is an open address algorithm which combines linear probing with the cuckoo hashing technique. It offers a constant worst case look-up but insertion might requires a sequence of relocation similar to the cuckoo hashing. The concurrent hopscotch hashing synchronizes concurrent accesses using locks, one per bucket. A concurrent version of cuckoo hashing found in [5] is a bucketized cuckoo hash table using a stripe of locks for synchronization. As lock-free programming has been proved to achieve high performance and scalability [15] [2], a number of lock-free hash tables have also been introduced in the literature. Micheal, M. [11] presented an efficient lock-free hash table with separated chaining using linked lists. Shalev O. and Shavit N. [14] designed another high performance lock-free closed address resizable hash table. In [3], a lock-free/wait-free hash table is introduced, which does not physically delete an item. Instead, all the live items are moved to a new table when the table is full.

To the the best of our knowledge, there has not been any lock-free cuckoo hashing introduced in the literature. There are several reasons which can explain this fact. Because a key can be stored in two possible independent slots in two tables, synchronization of different operations becomes a hurdle to overcome when using lock-freedom. As an example, two insertion operations of a key with different data can simultaneously and independently succeed; this can cause both of them to co-exist, which is not aligned with the common semantics of hash tables in the literature. In addition, a relocation of a key from

[1]Load factor: the ratio between the total number of elements currently in the table over its capacity.

one table to another is a combination of one remove and one insert operations, which need to be combined in a lock-free way. While taking care of that, the relocation of a key when it is being looked up can cause the look-up operation to miss the key, though it is just relocated between tables.

In this work, we address these challenges and present a lock-free cuckoo hashing algorithm. We do not consider bucketized cuckoo hashing. To the best of our knowledge, this is the first lock-free cuckoo hashing algorithm in the literature. Our algorithm tolerates any number of process failures. The algorithm offers very high query throughput by optimizing the synchronization between look-up and modification operations. Concurrency among insertions is also high thanks to a carefully designed relocation operation. The sequence of relocations during insertion is broken down into several single relocations to allow higher concurrency among operations. In addition, a fine tuned helping mechanism for relocation operations is designed to guarantee progress. Our evaluation results show that the new cuckoo hashing outperforms the state-of-the-art hopscotch and lock-based chained hash tables [8].

The rest of this paper is organized as follows. Section 4.2 introduces our algorithm in a nutshell. The full design together with a pseudo-code description is presented in Section 4.3. Section 4.4 provides the proof of correctness of the algorithm. Experiments and evaluation results are presented in 3.4. Finally, Section 4.6 concludes our paper.

4.2 Lock-free cuckoo hashing algorithm

Our concurrent cuckoo hashing contains two hash tables, hereafter called sub-tables, which correspond to two independent hash functions. Each key can be stored at one of its two possible positions, one in each sub-table. To distinguish the two sub-tables, we refer to one as the primary and to the other as the secondary. The look-up operation always starts searching in the primary sub-table and then in the secondary one. Because there are different use cases of looking-up operations, we divide them into two types. One, *search*, is a query-only one which asks for the existence of a key without modifying the hash table.

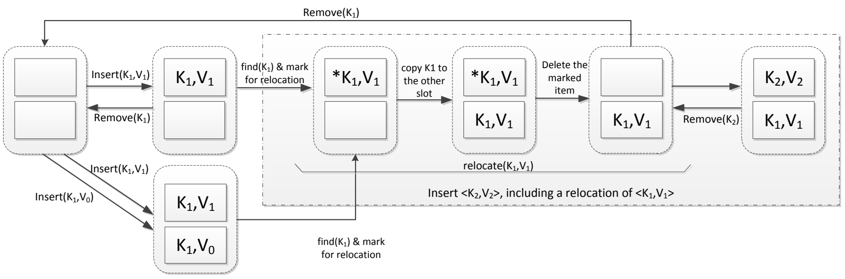


Figure 4.1: State transition of two possible positions of a key in primary (upper) and secondary sub-tables

The other one is a query as a part of another operation such as a deletion or an insertion. We refer to it as *find* to distinguish it from the “real” *search*.

A *search* operation starts by examining the possible slots in the primary sub-table first, and then in the secondary one, and reports if the searched key is found in one of them. Such a simple search, however, can miss an existing key and report the key as not found. The reason is that the reading from two slots is not performed in one atomic step and a relocation operation might interleave in between. The searched key can be relocated from the secondary to the primary sub-table but is missed by the above reading operations. We called such key a “moving key”. To deal with this issue, we design a two round query protocol enhanced with a logical clock based counter technique. Each hash table slot has a counter attached to it to count the number of relocations at the slot. The first round of the two round query reads from the two possible slots and check for the existence of the searched key like the mentioned simple search does. In addition, it records the slot’s counter values. If the key is not found, the second round does similar readings and examination. The second round can discover the key if it was relocated from the secondary to the primary sub-table, and was missed by the first round query. However, it might also miss the key if it has been relocated back and forth between sub-tables several times and interleaved with the readings. Therefore, the second round also records the counter values and compares them with the values of the first round. If the new values are at least two units higher than the previous ones, there is a possibility that even the two round query misses the key because two or more interleaving relocations have happened. In this case, the search is reexecuted.

The *insert* operation of a key starts by invoking *find* to examine if the key exists. If it does not, the insertion is made to the primary sub-table first and, only if a collision occurs, to the secondary sub-table. If both positions are occupied, a relocation process is triggered to displace the existing key to make space for the new key. The original cuckoo approach [12] inserts the new key to the primary sub-table by evicting a collided key and re-inserting it to the other sub-table, as described in Section 4.1. This approach, however, causes the evicted key to be "nestless", i.e. absent from both sub-tables, until the re-insertion is completed. This might be an issue in concurrent environments: the "nestless" key is unreachable by other concurrent operations which want to operate on it. One way to deal with this issue is to make the whole relocation process atomic, which is not efficient and scalable since it is going to result in coarse grained synchronization. We approach the relocation process differently. If an insertion requires relocation, an empty slot is created before the new key is actually added to the table. Our approach contains two steps. First, we search for the cuckoo path, to find a vacant slot. Thereafter, the vacant slot is "moved" backwards to the beginning of the cuckoo path by "swapping" with the last key in the cuckoo path, then the second last key and so forth. The new key is then inserted to the empty slot using an atomic primitive. Each "swap" step involves modifications of two slots in two sub-tables. We can design a fine tuned synchronization for the "swap" using single word Compare-And-Swap (CAS)[2] primitives by using the pointer's Least-Significant-Bit (LSB) marking technique. This technique, which takes advantages of aligned memory addresses and is widely used in the

[2]CAS is a synchronization primitive available in most modern processors. It compares the content of a memory word to a given value and, only if they are the same, modifies the content of that word to a given new value.

literature, set the unused LSB for certain purposes. In our case, it indicates one thread's intention to relocate a table's entry. A helping mechanism is also designed so that other concurrent operations can help finishing an on-going relocation.

Another issue with insertion operations is that a key can be inserted to two sub-tables simultaneously. Since concurrent insertions can operate independently on two sub-tables, they can both succeed. This results in two existing instances of one key, possibly mapped to different values. To prevent such a case to happen, one can use a lock during insertion to lock both slots so that only one instance of a key is successfully added to the table at a time. As we aim for high concurrency and strong progress guarantees, we solve this problem differently. Our table allows, in some special cases, two instances of a key to co-exist in two sub-tables. The special cases are when the two instances have been inserted to two sub-tables simultaneously. (It is noticed that, concurrent insertions to the same sub-table operate and linearize normally, which guarantees that only one instance of a key exists in one sub-table). We design a mechanism to delete one of the instances internally and as soon as possible so that our table still provides the conventional semantic in which one key is mapped to one value. The question is: which instance between the two is to be deleted? Since the two successful insertions which lead to the co-existence must be concurrent, it is always possible to order them so that the insertion to the secondary sub-table is linearized before the insertion to the primary one. In this way, the latter insertion "overwrites" the data inserted by the former one. As a result, if the same key is found in both sub-tables at a certain time, the key in the primary table is the only valid one. Our mechanism to realize such duplication and remove the overwritten key, i.e the one in the secondary sub-table, will be described after discussing below the consequence of such co-existence to the hash table's operations.

We now analyze the consequence of the co-existence of two instances of a key to the operations of the hash table, beginning with the look-up operations. The query-only *search* first examines the primary hash table, and can report if the searched key is stored there immediately. In the cases that two instances of a key co-exist in two sub-tables, *search* always returns the one in the primary sub-table, which agrees with the semantics described in the previous paragraph. The *find* operation, on the other hand, is used at the beginning of any *insert* or *remove* operation. The result of the invoked *find*, i.e the key exists in one or both sub-tables, affects the way the invoker behaves. Therefore, if *find* discovers that two instances of a key exist, it deletes the one in the secondary sub-table, as being described in detail in the next section. When an *insert* or a *remove* proceeds after *find* returns, there is going to be one instance of the key in the table.

The remove operation of a key starts by invoking *find* to locate the table's slot where the key is being stored. Then the key is removed by means of a CAS primitive.

Figure 4.1 shows the states of two possible positions on two sub-tables where a key K_1 can be hashed to. The states change according to the operations performed. The operations are, for example, an *insert*(K_1, V_1), a *remove*(K_1, V_1), two concurrent *insert*(K_1, V_0) and *insert*(K_1, V_1), and an *insert*(K_2, V_2) (in the dashed rectangle) which requires a relocation of $< K_1, V_1 >$. When *find()* is invoked, it can also delete the duplicated key, i.e $< K_1, V_0 >$ stored in the second sub-table.

4.3 Detailed Algorithmic Description

We are now presenting the detailed description and pseudo-code for the functions of our cuckoo hash table. The pseudo-code follows the C/C++ language conventions.

4.3.1 Search operation

Searching for a key in our lock-free cuckoo hash table includes querying for the existence of the key in two sub-tables, as in Algorithm 4.2. A key is available if it is found in one of them. A search operation starts with the first query round by reading from the possible slots in the primary sub-table table[0] (lines 69-71), and then in the secondary sub-table table[1] (lines 72-74). If the key is found in one of them, the value mapped to key is returned.

```
46 HashEntry:
47   word key,
48   word value;

50 CountPtr:
51   <HashEntry*,int> <entry,counter>

53 int hash1(key)
54 int hash2(key)
55 bool checkCounter(int ts1, int ts2, int ts1x, int ts2x)
56   /*check the counter values to see if 2 relocations may have
         taken place*/

58 class CuckooHashTable
59   EntryType *table[2][]      //2 sub-tables
60   word find(word key, CountPtr &<e1,ts1>, CountPtr &<e2,ts2>)
61   word search(word key)
62   insert(word key, word value)
63   remove(word key)
64   relocate(int which, int index)
65   help_relocate(int which, int index, bool initiator)
66   del_dup(idx1, <e1,ts1>, idx2 , <e2,ts2>)
```

Algorithm 4.1: Data structure and support functions

As mentioned in section 4.2, the above query can miss the searched key which happens to be a "moving key". The key present in table[1] is relocated to table[0], meanwhile *search* reads from table[0] and then table[1]. To avoid such missing, *search* performs the second round query (lines 77-79).

This two-round query, however, still can miss an existing key if the key is relocated back and forth between table[1] and table[0] repetitively and alternatively when *search* reads from each sub-table. The possibility of such continuously relocation of a key is very rare but can not be ruled out. To deal with that, we employ a technique based on Lamport's logical clocks [9]. The idea of this technique is to attach a counter to each slot of the hash table to record the number of relocations happening at that slot. Similar to a logical clock whose value

```
67 //h1=hash1(key) and h2=hash2(key)
68 while (true)
69   <e1,ts1> ← table[0][h1] //read the element \& counter
70   if (e1≠NULL ∧ e1→key = key)
71     return e1→value
72   <e2,ts2> ← table[1][h2]
73   if (e2≠NULL ∧ e2→key = key)
74     return e2→value

76   //second round query
77   <e1x,ts1x> ← table[0][h1]
78   ...
79   <e2x,ts2x> ← table[1][h2]
80   ...

82   if (checkCounter(ts1, ts2, ts1x, ts2x))
83     continue
84   else
85     return NIL
```

Algorithm 4.2: *word* ***search*** (*word* key)

changes when a local event happens or when a message is received, the value of the counter is changed on the event of relocations. The counter is initialized to 0. When an element stored in a slot is relocated, the slot's counter is incremented. When a slot serves as the destination of a relocation, its counter is updated with the maximum of its current counter value and the source's counter value, plus 1. The counter value of a slot remains even when the element stored in that slot is deleted or relocated to the other sub-table. For example, consider a key associated with counter value t is stored at table[1][h2]. When key is relocated to table[0][h1] which has counter t_1, the new counter value of table[0][h1] is $max(t, t_1) + 1$ and that of table[1][h2] is incremented to $t + 1$.

By examining the above counters after the second round query, a search can detect if it might have missed an existing key. Such missing happens if: (i) Before the execution of line 69, the key is stored in the secondary table at table[1][h2], then (ii) the key is relocated from table[1] to table[0] before the second read at line 72, then (iii) relocated back to table[1] before the next read at line 77, and finally (iv) relocated again back to table[0] before line 79 . If it is so, the counter value read at line 77 should be at least two units higher than the one read at line 69. Similar condition is applied for the counter values read at line 79 and line 72. In addition, the counter value read at line 79 is at least 3 units higher than the one read at line 69 because the counter increases its value like a logical clock when a relocation happens. If these conditions are satisfied, i.e *checkCounter* returns $true$, the two-round query probably misses an item because of alternative relocations, so the search restarts.

In practice, as each slot in the hash table is a pointer to a table element, we can use the unused bits of pointer values on x86_64 to store the counter value. Currently pointers on x86_64 use only 48 lower bits of the available 64 bits. We can use the 16 highest bits of the 64-bit pointer to store the counter value, an approach which has been used in literature [1]. This approach is efficient as the pointer to an element and its counter can be loaded in one read operation. The disadvantage of this technique is that the counter which has been increased by $2^{16} + k$ can be misinterpreted to be increased by just k, where k is any counter value. However, such increment of $2^{16}+k$ can only be made by many thousands of relocation operations happening at the same slot. Moreover, it must have happened in a very short period of time of a search operation to cause such misinterpretation. With a good choice of hash functions, the possibility that such misinterpretation happens is practically impossible. Therefore, 16 bits are sufficient to store the counter value.

4.3.2 Find operation

Algorithm 4.3 shows the pseudo code of the *find* operation, which functions similar to the *search*. The *find* takes an argument key and answers if, and in which sub-table, the key exists. In addition, it also reports the current values (and their associated counter values) stored at the two possible positions of key. The logic flow of the *find* is similar to that of the *search*, in the sense that it also uses a two round query. However, it has 3 main differences compared to the *search*.

```
//h1=hash1(key) and h2=hash2(key)
word result; int counter

while (true)
  <e1,ts1> ← table[0][h1]
  if (e1 ≠ NULL)
    if (e1 is marked)
      help_relocate(0, h1, false)
      continue
    if (e1→key = key)
      result ← FIRST

  <e2,ts2> ← table[1][h2]
  if (e2 ≠ NULL)
    if (e2 is marked)
      help_relocate(1, h2, false)
      continue;
    if (e2→key = key)
      if (result = FIRST)
        del_dup(h1,<e1, ts1>,h2,<e2, ts2>)
      else
        result ← SECOND

  if (result=FIRST ∨ result=SECOND)
     return result
  /*second round query*/
  <e1,ts1x> ← table[0][h1]
  ...
  <e2,ts2x> ← table[1][h2]
  ...

  if (checkCounter(ts1, ts2, ts1x, ts2x))
    continue
  else return NIL
```

Algorithm 4.3: *word* ***find***(*word* key, CountPtr& <e1,ts1>, CountPtr& <e2,ts2>)

```
//h1=hash1(key); h2=hash2(key)
HashEntry *newNode(key,value)
CountPtr *<ent1,ts1>, *<ent2,ts2>
start_insert:
  int result ← find(key, <ent1,ts1>, <ent2,ts2>)
  if (result=FIRST ∨ result=SECOND)
    Update the current entry with new value
    return

  if (ent1=NULL)
    if (¬CAS(&table[0][h1],<ent1,ts1>,<newNode,ts1>))
      goto start_insert
    return
  if (ent2=NULL)
    if (¬CAS(&table[1][h2],<ent2,ts2>,<newNode,ts2>))
      goto start_insert
    return

  result ← relocate(0, h1)
  if (result=true) goto start_insert
  else
    //rehash()
```

Algorithm 4.4: *insert*(*word* key, *word* value)

First, if it reads an entry who LSB is marked, indicating an on-going relocation operation, it helps the operation (lines 92-93, and 100-101). Secondly, it examines both sub-tables instead of returning immediately when key is found. This is to discover if two instances of the key exist in two sub-tables. When the same key is found on both sub-tables, the one in the secondary sub-table table[1] is deleted (line 105), as described in Section 4.2. Finally, *find* returns also current items which are stored at two possible slots where key should be hashed to. This information is used by the invokers, i.e *insert* or *delete* operations, as described in next subsections.

4.3.3 Insert operation

The insertion of a key, Algorithm 4.4, works as follows. First, it invokes the *find* at line 124 to examine the state of the key: if it exists in the sub-tables and what are the current entries stored at the slots where the key can be hashed to. If the key already exists, the current value associated with it is updated with the new value and the *insert* returns (line 127). Otherwise, the *insert* operation proceeds to store the new key. If one of the two slots is empty (lines 129 and 133), the new entry is inserted with a CAS. If both slots are occupied by other keys, relocation process is triggered at line 138 to create an empty slot for the new key. The relocation operation is described in detail in Section 4.3.5. If the relocation succeeds to create an empty slot for the new key, the insertion retries. Otherwise, which means the length of the relocation chain exceeds the THRESHOLD, the insertion fails. In this case, typical approaches in the literature of cuckoo hashing such as a rehash with two new hash functions or an extension of the size of the table can be used.

4.3.4 Remove operation

```
void remove(word key)
  //h1=hash1(key); h2=hash2(key)
  CountPtr *<ent1,ts1>, *<ent2,ts2>
  while (true)
    ret ← find(key, <ent1,ts1>, <ent2,ts2>)
    if (ret = NULL) return
    if (ret = FIRST)
      if (CAS(&table[0][h1], <ent1,ts1>, <NULL,ts1>))
            return
    else if (ret = SECOND)
      if (table[0][h1] ≠ <ent1,ts1>)
        continue
      if (CAS(&table[1][h2], <ent2,ts2>, <NULL,ts2>))
        return
```

Algorithm 4.5: Remove operation

The *remove* operation also starts by invoking *find* at line 146. If the key is found, it is removed by a CAS, either at line 149 or 154.

4.3.5 Relocation operation

When both slots which can accommodate a newly inserted key are occupied by existing keys, one of them is relocated to make space for the new key. This can trigger a sequence of relocations as the other slot might be occupied too. The *relocate* method presented inAlgorithm 4.6 performs such a relocation process. As mentioned earlier, we use a relocation strategy which can retain the presence of a relocated key in the table without the need for expensive atomicity of the whole relocation process. First, the cuckoo path is discovered, lines 158-177. Then, the empty slot is moved backwards to the beginning of the path, where the new key is to be inserted, lines 179-195.

The path discovery starts from a slot index of one of the sub-tables identified by which and runs at most THRESHOLD steps along the path. If table[which][index] is an empty slot, the discovery finishes (line 176). Otherwise, i.e the slot is occupied by a key (line 169), the key should be relocated to its other slot in the other sub-table. The discovery then continues with the other slot of key. If this slot is empty, the discovery finishes. Otherwise, the discovery continues similarly as before. Each element along the path is identified by a sub-table and an index on that sub-table. Along the path, the sub-tables that elements belong to alternatively change between the primary and secondary ones. Therefore, the data of the path which need to be stored are the indexes of the elements along the path and the sub-table of the last element.

Once the cuckoo path is found, the empty slot is moved backwards along the path by a sequence of "swaps" with the respective preceding slot in the path. Each swap is actually a relocation of the key in the latter slot, a.k.a the source, to the empty slot, a.k.a the destination. Because of the concurrency, the entry stored in the source might have changed. Thus, the relocation operation needs to update the destination and check for its emptiness (lines 189-190), and retry the path discovery if the destination is no longer empty (line 194). If the destination

```
int route[THRESHOLD] //storing cuckoo path
int start_level=0, tbl=which, idx=index
path_discovery:
  bool found ← false
  int depth ← start_level
  do {
    <e1,ts1> ← table[tbl][idx];
    while (e1 is marked)
      help_relocate(tbl, idx, false)
      <e1,_> ← table[tbl][idx]
    if (<pre,tsp>=<e1,ts1> ∨ pre→key=e1→key)
      if (tbl = 0) del_dup(idx,<e1,ts1>,pre_idx,<pre,tsp>)
      else del_dup(pre_idx,<pre,tsp>,idx,<e1,ts1>)
    if (e1 ≠ NULL)
      route[depth] = idx
      key ← e1→key;
      <pre,tsp> ← <e1,ts1>
      pre_idx ← idx
      tbl ← 1 - tbl
      idx ← tbl = 0 ? hash1(key) : hash2(key)
    else found ← true
  } while (!found ∧ ++depth<THRESHOLD)

if (found)
  tbl ← 1 - tbl;
  for (i ← depth-1; i>=0; --i, tbl ← 1-tbl)
    idx ← route[i].index
    <e1,ts1> ← table[tbl][idx]
    if (e1 is marked)
      help_relocate(tbl, idx, false)
      <e1,ts1> ← table[tbl][idx]
    if (e1 = NULL) continue
    dest_idx ← tbl=0?hash2(e1→key):hash1(e1→key)
    <e2,ts2> ← table[1-tbl][dest_idx]
    if (e2 ≠ NULL)
      start_level ← i+1
      idx ← dest_idx
      tbl ← 1 - tbl
      goto path_discovery
    help_relocate(tbl, idx, false)
return found
```

Algorithm 4.6: *int* ***relocate***(*int* which, *int* index)

```
void help_relocate(int which, int index, bool initiator)
  while (true)
    <src,ts1> ← table[which][index]
    while (initiator && src is not marked)
      if (src = NULL) return
      CAS(&table[which][index)],<src,ts1>,<src|1,ts1>
      <src,ts1> ← table[which][index]
    if (src is not marked) return
    /*hd=hash(src.key) where hash is hash function used for
        table (1-which)*/
    <dst,ts2> ← table[1-which][hd])
    if (dst = NULL)
      nCnt ← ts1>ts2 ? ts1 + 1 : ts2 + 1
      if (<src,ts1> ≠ table[which][index])
        continue
      if (CAS(&table[1-which][hd],<dst,ts2>,<src,nCnt>))
        CAS(&table[which][index],<src,ts1>,<NULL,ts1+1>)
        return
    //dst is not null
    if (src = dst)
      CAS(&table[which][index)],<src,ts1>,<NULL,ts1+1>)
      return
    CAS(&table[which][index],<src,ts1>,<src&~1,ts1+1>)
    return false;

void del_dup(idx1, <e1,ts1>, idx2 , <e2,ts2>)
  if (<e1,ts1>≠table[0][idx1] ∧
      <e2,ts2>≠table[1][idx2])
    return
  if (e1→key ≠ e2→key)
    return
  CAS(&table[1][idx2],<e2,ts2>,<NULL,ts2>)
```

Algorithm 4.7: Help relocation and delete duplication operations

is empty, the relocation is performed in three steps in the *help_relocate* operation presented in Algorithm 4.7. First, the source entry's LSB is marked to indicate the relocation intention (line 202). Then, the entry is copied to the destination slot (line 211), which has been made empty. Finally, the source is deleted (line 212). Marking the LSB allows other concurrent threads to help the on-going relocation, for example at lines 164 and 185.

After a slot is marked in *help_relocation*, the destination of the relocation might have been changed and is no longer empty. This can be because either other threads successfully help relocating the marked entry, or a concurrent insertion has inserted a new key to that destination slot. If it is the former case (line 215), the source of the relocation is deleted either by that helping thread (line 212) or by the current thread (line 216). If it is the latter case, the *help_relocation* fails, unmarks the source (line 218) and returns. The relocation process then continues but might need to retry the path discovery in the next loop.

4.4 Proof of correctness

In this section, we is going to prove that our cuckoo hash table is linearizable and lock-free. At first, we prove the linearizability under the assumption that a key exists in only one sub-table. Later on, we prove that if there are two instances of a key in two sub-tables, the linearizability are not violated. Then we continue to prove the lock-freedom. Due to space constraints, we defer part of the proof to the appendix. In the following, we assume a key can be stored in two possible positions: table[0][h1] and table[1][h2].

Lemma 4.1. *When help_relocation is invoked with arguments which and index, either it succeeds relocating the element pointed to by table[which][index] to the other table, i.e table (1-which) and unmarks the source slot; or if it fails doing that, the source slot, i.e table[which][index] is unmarked.*

Proof. If *help_relocation* succeeds to mark table[which][index], it continues to check the destination slot, line 207. At this point, other threads can see the marked element and help the relocation. If the destination is not empty, and no other thread has successfully helped the relocation, the current thread unmarks table[which][index] (line 218). If the current thread or any other helping threads successfully changes the destination to the same pointer value that table[which][index] holds (line 211 or 215), the relocation succeeds. The source is then unmarked at line 212 or 216. Between the time the content is copied to the destination till when the source is unmarked, other threads can perform update operations to the same slot. As both slots are occupied by the same key, any operations touching one of the slots can detect the marked pointer and perform the helping before continuing with their own operations. □

Corollary 4.1. *If help_relocation succeeds relocating entry in table[which][index] to the other sub-table, the counter values of both source and destination increases by at least* 1.

Proof. The destination is written by means of a CAS at line 211 with a new counter value nCnt, which is one unit more than the maximum of the counter value of the source and destination. The source's counter value is increased by one at line 212 or 216. □

Following Corollary 4.1, it is easy to see that if there are two relocations which has happens at one slot, the slot's counter has been increased by at least 2 units.

Lemma 4.2. *The search is linearizable.*

Proof. The *search* operation returns a value only when one of the keys in e1, e2, e1x, e2x (lines 69, 72, 77 or 79) matches the searched key. In this case, *search* is linearized at the respective line. We now consider the loop of the *search* to see when it returns NIL.

As we discussed in Section 4.3.1, during a search for key, the two round query protocol misses the searched key only when there are a sequence of relocations of key from table[1] to table[0], back to table[1] and again to table[0], as described in subsection 4.3.1. Condition at line 85 can detect if such a scenario may happen by examining the counter values of table[0][h1] and table[1][h2], as described also in subsection 4.3.1. If the condition is satisfied, the *search* restarts. We note that the condition is also satisfied if there are two independent relocations of other keys which are stored in the same slots; or if there are only one relocation at each slot but the relocation increases the counter by two or more units. In these cases, the search might restart unnecessarily but its correctness is not violated.

Therefore, the *search* returns NIL when the key is not found in any of the read entries: e1, e2, e1x, e2x and there is no possibility that the key is relocated which forces the *search* to reexecute the loop. Even though, there are cases that key appears in one of the sub-tables at the time *search* performs a reading from the other sub-table. In such cases, *search* might still return NIL, but we can argue that it is totally correct. We consider, as an example, a key exist in one sub-table as *search* perform reading for the second round query at lines 77 and 79, and show the correct linearization points. Other readings, e.g lines 69 and 72, can be argued in a similar manner. If the key exists in the table when search executes lines 77 and 79, and the *search* returns NIL:

- key must be inserted to table[0] after the reading from that sub-table at line 77. The *search* can be linearized at line 77 where key has not been inserted to the table.

- Or key exists in table[1] before the *search* starts and is deleted before the *search* reads from it (line 79). The *search* can then be linearized to line 79, when key has been deleted.

- Or key exists in table[1] when the *search* reads from table[0] (line 77), is deleted and then re-inserted to table[0] before the *search* reads from table[1] (line 79). In such scenario, neither line 77 or line 79 can be the correct linearization point of the *search*. Because key exists in the table at those points of time, in particular, in the other sub-table than the one *search* reads from. Even though, we notice that there is an interval between when key is deleted from table[1] and when it is inserted to table[0] (if these operations overlap, the re-insertion would have failed), this period of time

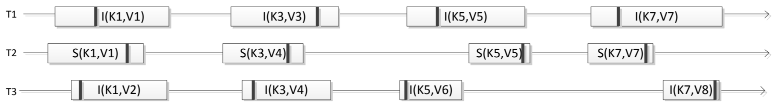


Figure 4.2: Concurrent inserts can create the existence of two instances of a key in the table

is inside the duration that *search* executes line 77 to line 79. In this interval, key does not exist anywhere in the table. Therefore, we can always linearized *search* to a point of time in that interval. This satisfies the requirement that linearization point must be between the time when *search* is invoked and when it responds.

Henceforth, *search* is linearizable. □

Lemma 4.3. *The find operation is linearizable.*

Proof. The arguments are similar to those of the *search* operation. The linearization points is either in lines 90, 98, 112 or 114 if it returns true. Otherwise, it is linearized similar to the cases that the *search* returns NIL. □

Lemma 4.4. *The remove operation is linearizable.*

Proof. If the *remove* operation returns from line 147, the linearization point is the same as the linearization point of the *find* operation it has invoked at line 146. Otherwise, the linearization point is the CAS at either line 149 or 154, depending on where (in table[0] or table[1]) the *find* operation found the key. □

Lemma 4.5. *The insert operation is linearizable.*

Proof. The arguments are similar to the *remove* operation. If the *insert* operation returns from line 127, the linearization point of the *insert* operation is the same as that of the *find* operation invoked at line 124. Otherwise, the linearization points are at CAS at line 130 or 134. □

Now we consider the scenario where two instances of a key co-exist in the table. When two concurrent insertions try to insert the same key to two sub-tables, they might both succeed and store it in two possible positions of that key. As described in Section 4.2, our hash table allows such physical co-existence and then removes the instance in the second sub-table before any mutating operations can operate on these two positions. The subsequence operations can only see one valid instance, i.e the one in the primary table.

Lemma 4.6. *The correctness of the* search *operation is immune of the concurrent physical existence of key in both sub-tables.*

Proof. As we noted before that a key existing in both sub-tables only happens when two or more insertions of that key are concurrent. We now consider the interleaving between these concurrent insertions and a search operation. If a *search* starts after the insertion to the primary sub-table has finished, the search always returns the value associated with the key stored in this primary sub-table as it reads from the primary table first, as we described earlier. The case where the search is concurrent with the insertion to the primary sub-table is divided into three small sub-cases illustrated in Figure 4.2 with correct linearization points:

- the other insertion, i.e to the secondary sub-table, is also concurrent with *search*: *I(K1,V2)* or *I(K3,V4)* are concurrent with other insertion and search of the same key.

- the other insertion starts after the *search* returns: *I(K7,V8)* is concurrent with *I(K7,V7)*, but after the search for *K7*. As *I(K7,V8)* invokes *find(K7)*, it finds out that $< K7, V7 >$ exists in the table. In this case, *I(K7,V8)* overwrites the previous value associated with *K7*, i.e *V7*, with the new value *V8*. At this point, only one instance of *K7* exists in the table.

- the other insertion finishes before the search starts, e.g operations on key K5. The search start examine the primary sub-table and return the stored value $< K5, V5 >$.

In all these cases, we can always find a correct history of the operations satisfying the linearizability with respect to the semantic of the hash table. □

The below propositions can be derived from the pseudo code of *find* and *relocate*.

Proposition 4.1. *If two instances of* key *co-exist in the table when a* find *on* key *is invoked, it removes the instance of the key in the second sub-table.*

Proposition 4.2. *If two instances of* key *co-exist in the table when a* relocate *of* key *happens, it removes the instance of the key in the second sub-table.*

These two propositions can be easily proved as below. Any subsequence *find* or *relocate*, after two instances of the same key have been inserted to the table, always performs reads from two possible positions of key and easily see if those positions store the same key (line 104 or 166). In that case, the one in the secondary table is removed at either line 105, 167 or 168.

Now, we examine the effect of the existence of a duplicated key to the correctness of *insert/remove* operations.

Lemma 4.7. *The correctness of a* remove *and* insert *operation of a key in Lemma 4.4 and 4.5 holds even when two instances of a key exist concurrently.*

Proof for Lemma 4.7. Without loosing generalization, we assume that there are two instances of a key K stored in the table.

We consider first the *insert* and *remove* operations of K. If an *insert* or a *remove* operation of K starts, *find* operation invoked at the beginning of that operation will delete the duplicated instance stored in the second sub-table (Proposition 4.1). The operation then proceeds with only one instance of K. Therefore the correctness of the operations stated in Lemmas 4.4 and 4.5 still holds.

We now consider the *insert/remove* operations of a key K' where K'≠K. A *remove* operation of an existing key K' does not modify the slots which are already occupied by the two instances of K. Therefore, the correctness proof in Lemma 4.4 still holds for the *remove* operations. Regarding the *insert* operations, the insertion of K' is affected by the existence of duplicates of K only if the insertion requires a relocation of K. However, the *relocate* operation remove one of the duplications as stated in Proposition 4.2. Therefore, the insertion of K' is not affected by a duplication of K, and the correctness proof in Lemma 4.5 still holds. □

Theorem 4.1. *The hash table algorithm is linearizable.*

Proof. Each operation of the hash table is linearizable follows Lemmas 4.2, 4.4, 4.5, and Lemmas 4.6, 4.7. □

Now, we are ready to prove that the algorithm is lock-free.

Lemma 4.8. *Either the help_relocation operation finishes after a finite number of steps or another operation must have finished in a finite number of steps.*

Proof. The *help_relocation* has two loops. The inner loop, which only runs by the initiating thread, repeats when table[which][index] changes its value to an unmarked, non-NULL pointer. Such a case happens only when the item in that slot is deleted or relocated, and then a new item is inserted to the same place. Either case means progress of those operations. The outer loop in *help_relocation* repeats at line 209 when the value stored in table[which][index] has been changed to a different value than the one that the current thread has read, i.e $< src, ts1 >$. Since table[which][index] stores a marked pointer, other threads modifying it must have helped the on-going relocation before they can change it to a new value. Therefore, when current thread detects that the value has changed value and repeats the outer loop because, the helping has already finished. Table[which][index] either is empty or carries another element. If it is empty, *help_relocation* finishes and returns at line 201. If it stores another element, *help_relocation* loops again to relocate the new one. Even though, the operation which stores the new element has already made progress. □

Lemma 4.9. *If a help_relocation operation finishes but fails to relocate table[which][index], there must be another operation making progress during that execution of the help_relocation.*

Proof. The *help_relocation* can be invoked by an initiating thread which intends to relocate table[which][index] item to the other, i.e (1-which), table. It can also be invoked by a helping thread if this thread see a table[which][index] which has been marked. If one of the threads succeeds to relocate item stored in table[which][index] to the other table, the relocation is successful. Otherwise, it fails. According to arguments in Lemma 4.1, the *help_relocation* fails because the destination of the relocation has been changed to non-empty, either by another insertion or relocation of another key. Either case means progress of another operations. □

We observe that *help_relocation* can encounter an ABA problem[3] [5], even when we have a proper memory management to handle the hash table's element. This scenario can take place when there are threads executing line 207 to relocate the same table[which][index]. Meanwhile, a new key can be inserted to dst, and then deleted from dst. Some of the threads doing relocation can observe that dst pointing to the inserted element, which leads to the CAS at line 211 to fail and the source of the relocation is unmarked at line 218. Meanwhile, other threads doing relocation might perform the CAS at line 211 after the deletion, and therefore succeed to copy table[which][index] to dst. As a result, the key exists in two sub-tables. However, such co-existence caused by the ABA does not hurt the correctness of our algorithm. This is because our algorithm is capable of tolerating such co-existence and can soon remove the one in the second subtable. Such removal is done by the calling thread performing this *help_relocation*, or any other thread doing *find* or *relocate* which involves the slots storing the duplicated key (at line 105, 167 or 168, as discussed in Section 4.2).

Lemma 4.10. *The search operation finishes after a finite number of steps, or another operation must have finished after a finite number of steps.*

Proof. The *search* operation only have one `while` loop. According to the proof for Lemma 4.2, the `while` loop repeats only when there are relocations of keys stored in table[0][h1] and table[1][h2], which means progress of operations performing the corresponding relocations. This observation also holds even when the relocations move key(s) back and forth between two sub-tables. In this case,

[3]ABA problem happens when an operation succeeds because the memory location it read has not changed; but in fact, it has changed its value from A to B and then back to A.

the *search* might not make progress but the relocation progresses towards the THRESHOLD number of relocation steps. When it reaches the threshold and returns false, the insertion calling such a relocation fails and proceeds with rehashing or resizing the hash table. □

Lemma 4.11. *A find operation finishes after a finite number of steps, or another operation must have finished after a finite number of steps.*

Proof. The *find* operation repeats its loop when it finds a marked pointer indicating an on-going relocation and must help it, e.g. line 93 or 101. The helped *help_relocate* operation succeeds which means progress for the operation initiates such relocation. The helped relocation can fail, but as in Lemma 4.9, it also means progress of another operation. The *find* operation also repeats the loop when the `if` condition at line 117 fails. This case is similar to that of the *search* operation, which is proved in Lemma 4.10. □

Lemma 4.12. *A insert operation finishes after a finite number of steps or another operation must have finished after a finite number of steps.*

Proof. The insertion is restarted if it fails to execute the CAS at line 130. This CAS is executed only if the *find* at line 146 has found that the key does not exist in the table and table[0][h1] holds the value NULL. Therefore, the reason that this CAS fails is only because table[0][h1] has been changed to a new value, which is caused by an insertion of a new key or a relocation of another key to that slot. This means progress of other operations. Similar argument holds for CAS at line 134.

The last case causing an insertion to restart is when it needs to trigger (and succeeds performing) a sequence of relocations at line 138. After relocation(0,h1) returns true, table[0][h1] is an empty slot and holds NULL. The insertion restarts and is going to succeed as table[0][h1] is now empty to accommodate the inserted key. □

Lemma 4.13. *A remove operation finishes after a finite number of steps or another operation must have finished after a finite number of steps.*

Proof. The *remove* operation contains one loop and is restarted when the `if` condition checked at line 152 fails. This check is executed in the case when he previous *find* found the key in the secondary sub-table but not in the primary sub-table. The check is required because there might be the case that the key was inserted to the primary sub-table after the *find* read from that sub-table, and then is relocated to the secondary sub-table where the *find* found it. If the relocation has not finished yet, i.e the key has not been deleted from the primary sub-table, and *remove* operation proceeds to delete the key from the secondary sub-table, the hash table becomes inconsistent. Therefore, checking if table[0][h1] has been changed since the previous *find* reads from it (line 152) helps discovering such cases. If it is so, the *remove* operation restarts. In this case, the mentioned insertion and relocation both make progress. □

Theorem 4.2. *The hash table algorithm is lock-free.*

Proof. According to Lemma 4.10, 4.12, 4.13, our cuckoo hash table always makes progress after a finite number of steps. □

4.5 Experimental Evaluation

This section evaluates the performance of our lock-free cuckoo hash table and compares it with current efficient hash tables. We use micro-benchmarks with several concurrent threads performing hash table's operations, a standard evaluation approach taken in the literature.

4.5.1 The Experimental Setup

We compare our lock-free cuckoo hashing with:

- A lock-based chained one: that uses a linked-list to store keys hashed to the same bucket. A number of locks, equal to the number of table segments, are used [8].

- Hopscotch hashing: a concurrent version of hopscotch hashing, with each lock for a segment [6]. Thanks to the kindness of the authors, we could obtain the original source code of hopscotch hashing.
- LF Cuckoo: our new lock-free cuckoo hashing.

All the algorithms were implemented in C++ and compiled with the same flags. No customized memory management was used. In all the algorithms, each bucket contains either two pointers to a key and a value, or an entry to a hash element, which contains a key and a value.

The experiments were performed on a platform of two 8-core Xeon E5-2650 at 2GHz with HyperThreading, 64GB DDR3 RAM. In our evaluation, we sampled each test point 5 times and plotted the average. To make sure the tables did not fit into the cache, we used a table-size of 2^{23} slots. Each test used the same set of keys for all the hash tables.

4.5.2 Results

Figure 4.3 presents the throughput result of the hash tables in different distributions of actions. The commonly found distribution is $90s/5i/5r$, i.e 90% *search*, 5% *insert* and 5% *remove*. Other less common distributions were also evaluated. One with more query-only operations: $94s/3i/3r$. Two others with more mutating operations: $80s/10i/10r$ and $60s/20i/20r$. As it is commonly known that original cuckoo hashing works with load factors lower than 49%, we used the load-factor of 40%. The concurrency increased up to 32 threads, the maximum number of concurrent hardware threads supported by the machines.

Our lock-free cuckoo hashing performs consistently better than both the lock-based chained and the hopscotch hashing in all the access distribution patterns. This is because positive searches need to examine either one or two table's slots, and negative searches need 3 read operations in most cases. Cases that *search* might need to perform more read operations do happen but not often. This is because the possibility that a relocation of a key happens concurrently as the key is being queried is not high. In addition, our algorithm is designed so

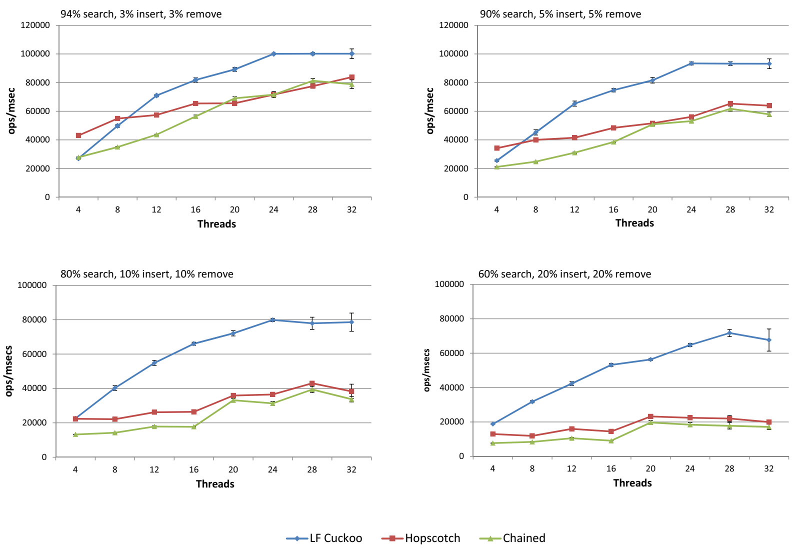
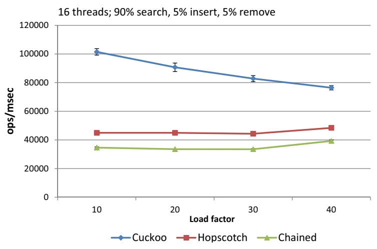


Figure 4.3: Throughput as a function of concurrency at load factor 40%

that the search operations still make progress concurrently with any other mutating operations. In contrast, the lock-based chained and the hopscotch hashing lock the bucket during insertion or removal.

Our lock-free cuckoo hashing maintains very high throughput in scenarios with more mutating operations, i.e $10i/10r$ or $20i/20r$, respectively. The insert operation in cuckoo hashing might require relocations of existing keys to make space for the new key. The algorithm, however, has been fine designed to allow high concurrency between relocation operations and other operations. Meanwhile, the lock-based chained and the hopscotch hashing degrade quickly when the percentage of mutating operations increases, mainly because of their blocking designs.

Figure 4.4 presents throughput results as a function of load factor. In cuckoo hashing, higher load factor means more relocations of existing keys during insertion. Therefore, we can observe that the throughput of our lock-free cuckoo hashing decreases when the load factor increases. Nevertheless, our cuckoo hashing algorithm always achieve throughput $1.5 - 2$ times as much as other



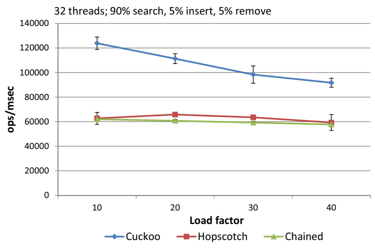


Figure 4.4: Throughput as a function load factor at 16 and 32 threads

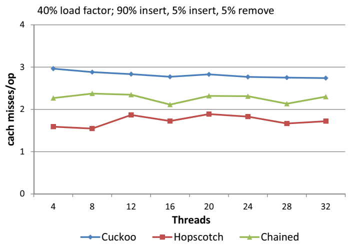


Figure 4.5: The number of cache-misses per operation.

algorithms, in both cases of 16 and 32 concurrent threads.

We now analyze the cache behavior of our lock-free cuckoo hashing. A positive search operation usually requires reading 1 or 2 references and a negative one often requires reading 3 references in the two-round query protocol, if no concurrent relocation happens at the read slots. Otherwise, search operations might need to perform more read operations, which might cause more cache misses. A removal of a key often needs one additional CAS compared to a search operation to delete the found element from the tables. Insertion operations are more complicated. If an insertion does not trigger any relocation, its behavior is similar to the deletion operation. Otherwise, it can cause more cache misses. We have recorded that the number of relocations is approximately 2% of the total performed operations, in the distribution of $90s/5i/5r$. Therefore, we expect a higher number of cache misses in our cuckoo hashing compared to other hash tables. A measurement of number of cache misses of our lock-free cuckoo hashing is presented in Figure 4.5. Our lock-free cuckoo table triggers about 3 cache misses per operation, a bit higher than hopscotch hashing and lock-based chained hashing. Regardless of a slightly higher number of cache misses, our lock-free cuckoo hashing has maintained a good performance over the other algorithms, thanks to the fine designed mechanism to handle concurrency.

4.6 Conclusions

We have presented a lock-free cuckoo hashing algorithm which, to the best of our knowledge, is the first lock-free cuckoo hashing in the literature. Our algorithm uses atomic primitives which are widely available in modern computer systems. We have performed experiments that compares our algorithm with the efficient parallel hashing algorithms from the literature, in particular hopscotch hashing and optimized lock-based chained hashing. The experiments show that our implementation is highly scalable and outperform the other algorithms in all the access pattern scenarios.

Bibliography

[1] M. Brunink, M. Susskraut, and C. Fetzer. Boundless memory allocations for memory safety and high availability. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 13–24, 2011.

[2] D. Cederman, A. Gidenstam, P. H. Ha, H. Sundell, M. Papatriantafilou, and P. Tsigas. Lock-free concurrent data structures. In S. Pllana and F. Xhafa, editors, *Programming Multi-Core and Many-Core Computing Systems*. Wiley-Blackwell, 2014.

[3] C. Click. A lock-free wait-free hash table. `http://www.stanford.edu/class/ee380/Abstracts/070221_LockFreeHash.pdf`. Accessed: 2013-11-14.

[4] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space efficient hash tables with worst case constant access time. In *The Proceedings of the 20th Annual Sympo-*

sium on Theoretical Aspects of Computer Science (STACS), volume 2607 of *LNCS*, pages 271–282. Springer Berlin Heidelberg, 2003.

[5] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[6] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *The Proceedings of the 22nd International Symposium on Distributed Computing (DISC)*, volume 5218 of *Lecture Notes in Computer Science*, pages 350–364. Springer Heidelberg, 2008.

[7] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.*, 39(4):1543–1561, Dec. 2009.

[8] D. E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[10] D. Lea. Hash table util.concurrent.concurrenthashmap. `http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/`, 2003.

[11] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA'02, pages 73–82. ACM, 2002.

[12] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122 – 144, 2004.

[13] K. Ross. Efficient hash probes on modern processors. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1297–1301, 2007.

[14] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM*, 53(3):379–405, May 2006.

[15] P. Tsigas and Y. Zhang. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. *SIGMETRICS Perform. Eval. Rev.*, 29(1):320–321, June 2001.

[16] M. Zukowski, S. Héman, and P. Boncz. Architecture-conscious hashing. In *Proceedings of the 2nd International Workshop on Data Management on New Hardware*. ACM, 2006.

PAPER IV

Nhan Nguyen, Philippas Tsigas and Håkan Sundell

ParMarkSplit: A Parallel Mark-Split Garbage Collector Based on a Lock-Free Skip-List

A shortened version of this paper appeared
in the proceedings of
the 27^{th} *The International Symposium on DIStributed Computing (DISC)*
Jerusalem, Israel. October 14-18, 2013
LNCS Vol.: 8205, pp. 557 - 558, Springer-Verlag 2013.

5

PAPER IV - ParMarkSplit: A Parallel Mark-Split Garbage Collector Based on a Lock-Free Skip-List

Abstract
Garbage collection is an important component of many modern programming languages and runtime systems. Mark-split is a garbage collection algorithm that combines advantages of both the mark-sweep and the copying collection algorithms. With the switch to multi-core and many-core microprocessors, parallelism has become a core issue in the design of any algorithm or software system. In this paper, we present a parallel mark-split garbage collector. Our parallel design introduces and makes use of an efficient concurrency control

mechanism for handling the list of free memory intervals. This mechanism is based on a lock-free skip-list design which supports an extended set of operations. Beside basic operations, it can perform a composite one that can search and remove and also insert two intervals atomically. We have implemented the parallel mark-split garbage collector in OpenJDK HotSpot as a parallel and concurrent garbage collector for the old generation. We present an experimental evaluation of our parallel collector and compare it with the default concurrent mark-sweep garbage collector present in OpenJDK HotSpot, using the DaCapo benchmarks. The experiments were done on two contemporary multiprocessor systems, one has 12 Intel Nehalem cores with HyperThreading and the other has 48 AMD Bulldozer cores. The evaluation shows that our parallel mark-split keeps the characteristics of the sequential mark-split, that it performs better than the concurrent mark-sweep garbage collector in applications that have low live/garbage ratio, and have live objects that often reside adjacent to each other. The experimental results also show that our parallel mark-split performs significantly better than a trivial parallelization based on locks in terms of both collection time and scalability.

5.1 Introduction

Garbage collection mechanisms have evolved dramatically since the first garbage collection algorithm was introduced by McCarthy [10] and now become an important features offered by many modern programming languages. Mark-sweep [10] and copying [4], and their derivations are among the algorithms widely used in garbage collection.

Mark-sweep and copying collectors have been extensively studied in the literature and their pros and cons have been identified in a range of scenarios. The time complexity of the mark phase is proportional to the amount of live data, while that of the sweep phase is proportional to the size of the heap. Lazy sweeping [8] improves mark-sweep by doing sweeping concurrently with the execution of the mutator. Recent mark-region algorithm [3] tries to reduce the fragmentation of the collected heap by dividing it to several regions and compacting objects to one end of the regions. Differing from mark-sweep, copying collectors only need time proportional to the amount of live data. However, copying collectors waste half of the heap space reserved for the need of the collector and it moves objects between semispaces. Copying collectors perform better than mark-sweep ones when the amount of live data is small compared to the size of the heap. This is the case where mark-sweep is penalized by the complexity of its sweep phase.

Sagonas and Wilhelmsson [14] introduced a garbage collection technique called mark-split that can combine advantages of mark-sweep and copying collection. Mark-split does not move objects, uses little extra space and has time complexity proportional to the size of the live data set. Mark-split evolves from mark-sweep but removes the sweep phase. Instead, it builds the list of free spaces during marking. At the beginning, mark-split assumes that the heap is free. The free list contains one big free interval of the whole heap. During the mark phase, the list is continuously repaired by using a special operation called *split* to remove spaces occupied by live objects. When the mark phase ends, the list contains only memory spaces that are not used by any live object and therefore, can be used for memory allocation.

Mark-split brings a new way of looking into the mark-sweep algorithm. It reduces the time complexity to the size of the live data set from the size of the heap. As mark-split removes the sweep phase by introducing to the mark phase additional split work, its performance depends on the data structure used to store the free intervals. The data structure should provide search operation for a free interval with sub-linear cost. Moreover, the list of free intervals should be translated to the allocator's free list with a low cost after the collection finishes. The original mark-split uses a sequential balanced search tree for this purpose. Data structures with similar properties such as splay trees, or skip lists can also be used [14].

While mark-split is comparable to mark-sweep and even outperforms mark-sweep in some situations, to the best of our knowledge, this is the first effort to design a mark-split collector for multicore systems. Our effort is to empower mark-split with a highly concurrent data structure to handle the free intervals and reuse the concurrent mark-sweep's marking to achieve a parallel mark-split collector. We consider using lock-free data structures as they provide scalability and high performance in shared memory multiprocessor architectures [7] [17]. Lock-free data structures guarantee progress and therefore, immune to deadlocks and livelocks. Several lock-free implementations of fundamental data structures have been introduced in the literature [7]. Even though, they couldn't be used directly to parallelize mark-split. This is because all concurrent data structures that support basic modification operations are not strong enough to build a list of free intervals in mark-split. The *split* operation in mark-split is a combined operation in the sense that it contains multiple basic operations: finding a correct free memory interval and performing the actual *splitting* on the interval, which is also a combination of two operations: i) removing one interval and possibly ii) adding two intervals. Concurrent environments require that *split* operations must be performed on the found interval in an atomic way to prevent other threads from modifying the interval. Building such a complicated concurrency control for a lock-free data structure is a challenge. We opt for skip-lists as they provide expected logarithmic time search without the need for a complex rebalance operation like balanced trees. The lock-free skip-list algorithm introduced by Sundell and Tsigas [16] which has high performance and good scalability offers a good mechanism for designing the concurrency controls needed for parallelizing mark-split.

The rest of this paper is organized as follows. Section 5.2 revisits the mark-split algorithm and gives a short introduction to The HotSpot Java Virtual Machine garbage collectors along with the challenge of parallelizing mark-split. We present our extended skip-list algorithm to meet the requirement of designing parallel mark-split in Section 5.3. The implementation of parallel mark-split algorithm with the design of a lazy splitting mechanism are presented in Section 5.4. Section 5.5 shows our evaluation of the garbage collector in the HotSpot along with discussions about the the result. The conclusion is given is Section 5.6.

5.2 Related Work

This section presents the mark-split algorithm. Then it introduces garbage collectors of the HotSpot.

5.2.1 The Mark-Split Algorithm

Mark-split [14], as its name suggests, has the same mark phase as mark-sweep, but does not perform the sweep phase. Instead, it builds a list of free memory intervals during the mark phase using one additional step called *split*. The mark-split algorithm is described briefly next and the reader is referred to the original article for more details.

Mark-split starts by creating the list of free intervals containing only a big free interval spanning the whole heap. Then it proceeds to the mark phase. For each unmarked live object, it marks the object and calls *split* to update the free intervals. The *split* operation splits a free interval containing an object into two smaller free intervals. One interval to the left of the object, which starts from the starting point of the free interval and ends at the beginning of the object. The other interval to the right of the object, which starts after the end of the object and ends at the end of the free interval. At this point, the algorithm decides to keep an interval if its size is bigger than a parameterized global threshold:

- Case 1: keep both the intervals, the original interval is replaced by two intervals: the left and the right one.
- Case 2/3: keep only the left/right interval, the original interval is replaced by the left/right interval.
- Case 4: keep none of the two intervals, the original interval is removed from the list of free intervals.

The most frequent operations that mark-split performs in the data structure storing free intervals are search for an interval and *split*. It is important that the search operation has sub-linear performance and the free intervals can be easily translated to a free list at the end of the collection. Data structures that satisfy these properties can be balanced search trees, splay-trees or skip-lists. In the original mark-split algorithm, a general balanced search tree [1] is used.

The removal of the sweep phase makes the complexity of mark-split proportional to the size of the live data set. This improvement though, comes with an overhead cost of maintaining a set of free intervals during the marking phase. The computational cost of this overhead has complexity of $O(N * logN)$ where N is the number of free spaces in the heap. Usually N is much smaller than the number of live objects because some live objects (or some dead objects) reside adjacent to each other. Therefore, paying this overhead cost to avoid the sweep phase, in certain situations, is beneficial. The overhead cost depends on the distribution of live objects but also depends highly on the data structure selected to store the free intervals. In a multi-core processor, a highly concurrent data structure to handle the free intervals can boost the performance of mark-split.

5.2.2 Garbage Collection in Java Virtual Machine

In HotSpot Java Virtual Machine, heap is mainly divided into two, young and old, generations. Newly allocated objects are placed in the young generation. Garbage in the young generation is collected by a young generation garbage collector. Objects in young generation that survive some collection cycles will be promoted to the old generation.

HotSpot offers various garbage collectors tailored to different environments and applications. Our parallel mark-split shares many features and properties with a parallel mark-sweep, similar to the the sequential implementations. But HotSpot does not contain any pure parallel mark-sweep collector. Therefore, our parallel implementation of mark-split is based on the Concurrent Mark-Sweep (a.k.a CMS) in HotSpot which can perform marking task in parallel. It is also noticed that a recent work by Gidra et al. [6] improves the scalability of the HotSpot's ParallelScavenge, a throughput oriented, copying collector, in NUMA architecture. Meanwhile, our work targets to improve the CMS - a low-pause collector.

The concurrent mark-sweep collector in HotSpot uses an algorithm introduced by Printezis and Detlefs [13]. The algorithm performs mark and sweep in four phases:

Initial Mark: All mutators are suspended. Mark and record all objects directly reachable from the roots.

Concurrent Mark: Resume mutator operation. At the same time, initiate a Concurrent Mark phase, which marks a transitive closure of reachable objects. By the end of this phase, most live objects but those referred to by references which are modified during this phase, are marked.

Final Mark (or **Remark**:) The mutators are suspended. Mark from the roots and consider modified reference fields in marked objects as additional roots. All the transitive closures contain all live objects by the end of this phase.

Concurrent Sweep: Resume the mutators once again, and sweep over the heap, deallocate unmarked objects.

5.2.3 Parallelizing Mark-Split

In mark-split, the most frequent operation is searching for and then splitting a free interval that contains a live object. The task involves in removing one free interval from and inserting up to two intervals to the data structure as described in section 5.2.1. The good concurrency and low latency that lock-free data structures exhibit over their blocking counter-parts make them a good candidate for the design of a parallel mark-split algorithm.

In the original mark-split [14], the use of a search data structure for mark-split is quite straight-forward, thanks to the simplicity of the sequential environment. Such simplicity is not present in a parallel environment. A lock-free skip-list such as the one introduced by Sundell et al. [16] can satisfy the performance but not the capability requirements of mark-split. We need to extend its functionality so that it can handle the list of free intervals in a parallel environment.

In a parallel design, the list of free intervals must allow performing a combination of one search, one remove and two insert operations in one atomic step. Our advantage is that, *split* on an interval, these removal and insertion are performed almost at the same location on the skip-list. Nevertheless, it is a challenge to design a concurrency control mechanism to meet these requirements. In the next section, we present our extended skip-list algorithm with such concurrency control. The following section presents our parallel mark-split using the extended skip-list.

5.3 Concurrent Skip-List with Extended Functionality

We are presenting a skip-list with extended functionality offering significant extensions over the original lock-free skip-list in [16]. A skip-list is a search data structure which stores elements in different layers of ordered linked lists with different densities to achieve tree-like behaviour. The original skip-list [16] can insert a new element, search for or remove an exiting element, but not a combination of those in one atomic operation as required by mark-split. The use of recursion in that skiplist also made its memory management complicated and not efficient. Our extensions of the new skip-list are significant both when it comes to operations that it supports and in the algorithmic design. The new *replace2* operation gives the ability to atomically replace a node with one or two new nodes; making the skip-list usable in the context of mark-split. Regarding the performance, we redesigned the data structure to make use of hazard pointers [11] for memory reclamation purposes and local-thread-storage. The modifications were not trivial as the original algorithm used recursion, something that does not work with a fixed number of hazard pointers.

The *split* procedure described in mark-split algorithm operates on an abstract free-list representing a set of free intervals. A free interval can be represented by a node in a skip-list, where key represents the start address S of the interval and the corresponding value represents its end address E. As used in [16], the

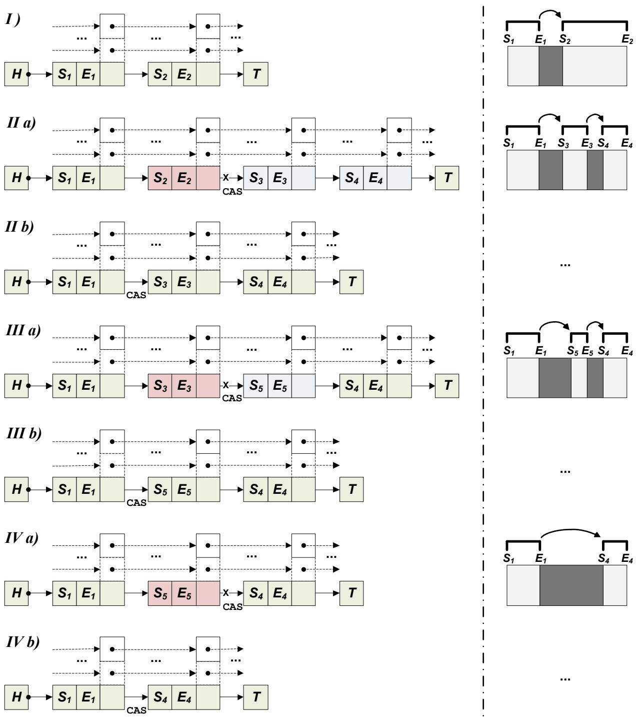


Figure 5.1: Multiple-step process for marking and deleting blocks simultaneously with inserting new nodes, thus fulfilling the corresponding (to the right) abstract operations on the free-list.

skip-list is basically made out of a singly-linked list with the nodes ordered by their keys. To allow probabilistic logarithmic expected time complexity for searching a particular node, nodes are inserted with a varying height such that several auxiliary lists are created with several layers of decreasing density with increasing height. For modifications to the abstract state of the free-list, only changes on the lowest layer's linked list are representative, i.e., changes are first performed atomically on the lowest layer and then modifications of the other layers can be performed concurrently with other operations. All necessary additional steps of the operation are eventually completed by making use of a suitably designed helping scheme. The helping scheme is designed to allow concurrent operation help another on-going operation when the former want to access the data that the latter is processing. A node in the skip-list can be defined to be *present* as soon as it is inserted on the lowest layer (i.e., there is another present node with a next pointer on the lowest level pointing to it) and *deleted* whenever the next pointer on the lowest layer for the corresponding node is marked (e.g. bit 0 set to 1). Atomic changes to the state of each node being present or deleted can be made using the *CAS* primitive.

The *split* procedure can result in four distinct changes on the abstract free-list. Each of these four changes must be possible to perform atomically with respect to each other. The possible changes are to either change S or E of an interval, replace the interval with two new intervals, or remove the interval altogether. To facilitate the representation of these abstract changes in the skip-list, an important observation is that it is possible to extend the skip-list to actually allow atomic deletion *and* insertion. The *CAS* primitive has the capability to both mark the next pointer and change it in the same operation. Thus, it is possible to atomically replace a node in the skip-list with one or more new nodes. The way that this modified skip-list is made to represent the abstract changes on the free-list, is shown in Figure 5.1.

Step I, illustrates how a free-list containing the intervals $\langle S_1, E_1\rangle$ and $\langle S_2, E_2\rangle$ can be represented with two corresponding nodes in the skip-list.

In Step IIa, the interval $\langle S_2, E_2\rangle$ is split into two intervals $\langle S_3, E_3\rangle$ and $\langle S_4, E_4\rangle$, where $S_3 = S_2$ and $E_4 = E_2$. By means of a *CAS*, the pointer on the lowest level of node $[S_2, E_2]$ is atomically marked and made to point to the new node $[S_3, E_3]$ which is already pointing to the new node $[S_4, E_4]$. The deleted node is then finally removed (also part of the helping scheme) in **step IIb**, with the *CAS* operating on the previous node's corresponding next pointer. The remaining layers are then handled in a similar manner.

In Step IIIa, the interval $\langle S_3, E_3 \rangle$ is modified to become $\langle S_5, E_5 \rangle$ where either $S_5 = S_3$ or $E_5 = E_3$. By means of a *CAS*, the pointer on the lowest level of node $[S_3, E_3]$ is atomically marked and made to point to the new node $[S_5, E_5]$. The deleted node is then finally removed (also part of the helping scheme) in **step IIIb**, with the *CAS* operating on the previous node's corresponding next pointer. The remaining layers are then handled in a similar manner.

In Step IVa, the interval $\langle S_5, E_5 \rangle$ is removed altogether from the free-list. By means of a *CAS*, the pointer on the lowest level of node $[S_5, E_5]$ is atomically marked. The deleted node is then finally removed (part of the helping scheme) in **step IVb**, with the *CAS* operating on the previous node's corresponding next pointer. The remaining layers are then handled in a similar manner.

The lock-free property is fulfilled by properly designing the helping scheme so that whenever an attempt made to perform a *CAS* for the a-part of the steps fails, the helping scheme makes sure that the b-part is being performed before attempting the a-part again.

5.3.1 Implementation

The implementation of the extended skip-list is described in Algorithms 5.1, 5.2, 5.3, 5.4 and 5.5. The operation *find_and_split* removes a given interval (i.e., the $start$ and end memory addresses of the live object) from the list of free intervals represented by the skip-list. The node that contains the given interval is searched for, with the search starting from the $head$ node at the highest level. As

```
void find_and_split(void *start, void *end)
  do
    Node *node;
    Node *prev = head;
    for(i=MAX_HEIGHT;i>=0;i--)
      for(;;)
        node = prev→next[i];
        if(node & 1)
           Go backwards in path using $savedNodes[i+1]$ or higher
              and perform helping $prev$ if necessary
        if(node  matches  interval) break;
        prev = node;
      savedNodes[i]=prev;
    bool keepLeft=(start-node→start) $\geq$ T;
    bool keepRight=(node→end-end) $\geq$ T;
    int height = log2random(1,MAX_HEIGHT);
    if(keepLeft && keepRight)
      ok=replace2(node, new Node(node→start,start,height), new
          Node(end, node→end,height));
    else if(keepLeft)
      ok=replace1(node, new Node(node→start,start,height));
    else if(keepRight)
      ok=replace1(node, new Node(end,node→end,height));
    else
      ok=remove(node);
  while(!ok);
```

Algorithm 5.1: Finding and splitting intervals in the skip-list

```
bool replace2(Node *node, Node *node1, Node *node2)
   Connect all $next[]$ of $node1$ to $node2$
  do
    Node *next = node→next[0];
    if(next & 1) return false;
    node2→next = next;
    ok=CAS(&node→next[0], next, (node1 | 1));
  while(!ok);
  do_remove(node);
  for(i=1;i<node1→height;i++)
    do
      Node *prev = savedNodes[i];
      Node *next = prev→next[i];
       If $prev$ is deleted or not the previous node according to
           $node1$, update $savedNodes[i]$ while applying helping if
           necessary, and repeat
      node2→next[i]=next;
      ok=CAS(&prev→next[i],next,node1);
    while(!ok);
     if $node1$ or $node2$ has been marked for deletion, perform
         helping if necessary and exit for-loop
  return true;
```

Algorithm 5.2: Replacing a node with two new nodes in the skip-list

```
bool replace1(Node *node, Node *node1)
  do
    Node *next = node→next[0];
    if(next & 1) return false;
    node1.next = next;
    ok=CAS(&node→next[0], next, (node1 | 1));
  while(!ok);
  do_remove(node);
  for(i=1;i<node1→height;i++)
    do
      Node *prev = savedNodes[i];
      Node *next = prev→next[i];
       If $prev$ is deleted or not the previous node according to
          $node1$, then update $savedNodes[i]$ while applying helping if
          necessary, and repeat
      node1→next[i]=next;
      ok=CAS(&prev→next[i],next,node1);
    while(!ok);
     if $node1$ has been marked for deletion, perform helping if
        necessary and exit for-loop
  return true;
```

Algorithm 5.3: Replacing a node with a new node in the skip-list

```
bool remove(Node *node)
  do
    Node *next = node→next[0];
    if(next & 1) return false;
    ok=CAS(&node→next[0], next, (next | 1));
  while(!ok);
  do_remove(node);
  return true;
```

Algorithm 5.4: Removing a node in the skip-list

```
struct Node
  void * start, end;
  int height;
  Node* next[height];

static Node *head=new Node($-\infty$,$-\infty$,MAX_HEIGHT);
static Node *tail=new Node($\infty$,$\infty$,MAX_HEIGHT);
thread static savedNodes[MAX_HEIGHT];

void do_remove(Node *node)
   Mark $node.next[x]$ on all levels $x$ using CAS
  for(i=node→height-1;i>=0;i--)
    Node *prev = savedNodes[i];
    Node *next = node→next[i]&(~1);
    bool ok = CAS(&prev→next[i],node,next);
    if(!ok)
       Update $savedNodes[i]$ such that it is the previous node of
          $node$ and perform helping if necessary of deleted nodes
          in the path, and repeat.  If previous node cannot be
          found, perform next lap in the for-loop
```

Algorithm 5.5: Data structures and auxiliary procedures for the skip-list.

the search is done in the skip-list level by level downwards, the previous node on each level is stored in the thread-local-storage $savedNodes$ array. These remembered previous nodes are later used when deciding to either replace or remove the found node, according to the rules described in Section 5.2.1. If the found node, represented by $node$, is concurrently modified, the corresponding replace or remove attempts will fail, and the whole *find_and_split* procedure is repeated.

Operation *replace2* describes how $node$ can be atomically replaced by two new nodes $node1$ and $node2$. First the next pointer of $node$ on the lowest level is atomically modified using *CAS*, to both contain the deletion mark (represented by the pointer value of 1) and instead point to $node1$. Thereafter, $node$ is fully removed from the skip-list, and then $node1$ and $node2$ are inserted together, starting from level 1 and going upwards. During this insertion, $node1$ or $node2$ can have been concurrently deleted, in which case the insertion is aborted and helping is applied to make sure the deleted node is fully removed. *Replace1* operation follows the same way as *replace2* but only one new node, $node1$, atomically replaces $node$. *Remove* operation deletes a $node$ as follow. First, the next pointer of $node$ on lowest level is atomically modified using *CAS*, to contain the deletion mark. Thereafter, $node$ is fully removed from the skip-list by *do_remove*. Before actually starting modifying next pointers of previous nodes, the deletion mark is propagated upwards on all levels of the next pointer of $node$ using *CAS* operations. This step is also required to be done by all concurrent operations that apply helping. The next step is then to modify the next pointer of all previous nodes of $node$ such that they should instead point to the next node of $node$, starting with the highest level of the next pointers of $node$ and going downwards. This is done by using *CAS* to atomically update the next pointer of the previous node, possibly given by $savedNodes[i]$, from originally pointing to $node$ to instead point to the next node. As concurrent helping can have been applied, it is important to notify this state when trying to update a possibly outdated pointer in $savedNodes[i]$.

For internal memory management, hazard pointers [11] is preferably used. The thread-local-storage $savedNodes$ can then be replaced by a corresponding number of hazard pointers. To also allow the search part of *find_and_split* to pass through next pointers that are marked, without applying helping, the same hand-over trick used in [15] can be applied to maintain the safety of hazard pointers while de-referencing the marked next pointer.

5.3.2 Correctness

We now sketch the proof of correctness for the linearizability and lock-free criteria.

Lemma 5.1. *The implementation of the find_and_split operation, described in Algorithm 5.1, is linearizable with respect to other concurrent find_and_split operations.*

Proof. Linearizability is demonstrated by giving the respective linearizability points for the corresponding executions of the *find_and_split* operations, described as four cases in Section 5.2.1.

- A *find_and_split* operation that results in Case 1 (split into two intervals), takes effect at the successful *CAS* in line 31. Before the *CAS* takes effect, the nodes $node1$ and $node2$ cannot be reached by the search part of any concurrent *find_and_split* invocation, and $node$ is not marked for deletion. After the *CAS* takes effect, the nodes $node1$ and $node2$ can clearly be reached by the search part of a concurrent *find_and_split*, as $node$ is now referring to $node1$ as being the next node, and $node$ has been logically deleted.

- A *find_and_split* that results in Case 2 (keep the left interval), takes effect at the successful *CAS* in line 49. Before the *CAS* takes effect, the node $node1$ (containing the left interval) cannot be reached by the search part of any concurrent *find_and_split*, and $node$ is not marked for deletion. After the *CAS* takes effect, the node $node1$ can clearly be reached by the search part of a concurrent *find_and_split* as $node$ is referring to $node1$ as being the next node, and $node$ has been logically deleted.

- A *find_and_split* that results in Case 3 (keep the right interval), takes effect at the successful *CAS* in line 49. Same arguments holds as for Case 2.
- A *find_and_split* that results in Case 4 (remove the interval), takes effect at the successful *CAS* in line 66. Before the *CAS* takes effect, $node$ is not marked for deletion. After the *CAS* takes effect, $node$ has been logically deleted, which will be noted by any concurrent *find_and_split* operations that will fail to modify $node$, as the *CAS* in lines 66 and 83 requires the mark to not be set of the next pointer.

□

Lemma 5.2. *The implementation of the find_and_split operation, described in Algorithm 5.1, is lock-free.*

Proof. The lock-free property of the *find_and_split* operation is maintained if a not finite execution of a loop for one invocation of the operation, is a result of a progress of another concurrent invocation. Assuming that the searched interval exists, the lines 7-11 are indefinitely repeated due to concurrent deletions. These deletions are due to successful concurrent *CAS* in lines 31, 49, and 66, all resulting in progress for the corresponding invocations. The lines 2-24 are repeated due to failed *replace2*, *replace1*, or *remove* functions. These functions fail in lines 29, 47, or 65, due to concurrent deletion of $node$. These deletions are due to successful concurrent *CAS* in lines 31, 49, and 66, all resulting in progress for the corresponding invocations. The lines 36-41 can indefinitely repeat due to concurrent deletions or insertions, which is progress for the corresponding invocations. Same arguments can be applied for the loops in lines 35-41 and lines 53-59. □

5.4 Parallel Mark-Split

In this section, we present the design of a lazy splitting mechanism for our parallel mark-split algorithm and then, the implementation of our parallel mark-split, a.k.a ParMarkSplit.

5.4.1 Lazy Splitting

In this subsection, we continue the presentation of our parallel mark-split algorithm by presenting the *lazy splitting* design. This is introduced to improve the efficiency of the splitting part of the mark-split algorithm. Originally, whenever a live object is marked, an interval is split to exclude the space occupied by the marked, i.e live, object from the space of the free intervals. We called this design *aggressive splitting*. Splitting for every marked object is inefficient in multi-threaded environment as it causes high contention at the shared data structure storing free intervals. We observe that: it is usually during the multi-threaded marking phases, marking threads consecutively mark some objects that are located adjacent to each other in the memory. We called them adjacent marked objects. The number of those objects are observed to range from 10% to 61% of the total number of live objects in some applications such as xalan, lusearch, tomcat, sunflow and avrora in the DaCapo benchmarks. Based on this observation, we propose a lazy splitting mechanism for our parallel mark-split algorithm to improve the efficiency of the splitting work.

The lazy splitting mechanism works as follows. Instead of doing *split_interval* right after an object is marked, a marking thread waits for the next object to be marked. If that object is not located adjacent to the previous object, the thread performs *split_interval* for the former object and keeps the latter object for splitting later. But if one or more consecutive marked objects are located adjacent to the object, the whole space occupied by them is excluded from the list of free intervals by only one call to *split_interval*, instead of one for each object. The work of booking objects for lazy splitting can be applied by each marking thread independently.

The lazy splitting mechanism reduces the number of accesses by marking threads to the list of free intervals, compared to the original approach, i.e aggressive splitting. To achieve this reduction, the mechanism have to bookkeep consecutively marked objects and check if they are adjacent in the memory. This computation is performed locally at each marking thread, without the need for synchronization. The lazy splitting mechanism benefits the parallel mark-split algorithm when the performance gain by the reduction of the number of calls to *split_interval* can cover the cost of bookkeeping all marked objects for lazy splitting: $(N - M).C_1 > N.C_2$, where N is the total number of of live objects; M is the total number of *split_interval* operations that the algorithm with lazy splitting performs; C_1 and C_2 are the average costs of performing a *split_interval* operation and bookkeeping a marked object, respectively. It is reasonable to assume that, for a specific application on a certain platform, these costs are constants. Therefore, whether the lazy splitting mechanism benefits ParMarkSplit collector mainly depends on the $(N - M)/N$ ratio.

An auto switch mechanism for determining when ParMarkSplit should use lazy splitting is easy to design by using a threshold T to decide when to use lazy splitting. Based on the evaluation results in section 5.5, we recommend $T = 10\%$. By default, lazy splitting is applied, as we record that in most applications, lazy splitting benefits the parallel mark-split GC. But, lazy splitting is not going to be applied when the GC finds, while collecting, that $(N - M)/N < T$.

Lazy splitting mechanism takes advantages of the fact that marking threads often consecutively mark live objects which reside adjacent to each other. It can improve the efficiency of mark-split in parallel environment. It can be included in a parallel mark-split collector along with a switching mechanism to control the use of lazy splitting.

5.4.2 Implementation

We can now use the extended skip-list to store the list of free intervals. For the rest of this paper, we use the terms *the skip-list* and *the list of free intervals* interchangeably. This list can be created outside the heap that is being collected. However, since parallel mark-split is expected to support concurrency and allow concurrent allocation, the list of free intervals can be stored in heap that is being collected as well. A parallel version of mark-split based on the concurrent mark-sweep can be achieved with following modification to the concurrent mark-sweep algorithm:

1. When GC starts, reset the list of free intervals to contain one interval of the whole space to be collected.

2. Whenever a thread successfully mark an object obj during the mark phases: If *aggressive splitting* is used, the thread calls *find_and_split(obj)* to remove space occupied by obj from the skip-list. When all the mark phases finish, all the marked objects are also excluded from the list of free intervals.
   If *lazy splitting* is used, the thread book-keeps the objects for the lazy splitting mechanism.

3. At the end of the Remark phase (i.e the mutator is still suspended), convert the list of free intervals to the format of the free list usable for allocation.

4. Remove the Sweep phase from CMS.

The correctness of the algorithm in the presence of all these possible concurrent interleavings can be achieved thanks to the design of the extended skip-list which allows *find_and_split* to be performed atomically and in a lock-free manner. The lock-free property of the skip-list, in our context where the number of objects to be marked is finite, also guarantees the termination of all the *find_and_split* operations, and therefore, the mark phases.

The parallel mark-split is then implemented as a garbage collector in the HotSpot, the Java Virtual Machine of OpenJDK. OpenJDK is an open source implementation of the Java Platform Standard Edition. It is contributed and supported by Oracle. We use OpenJDK 7 64-bit in our implementation. We named our implementation Parallel Mark-Split, or in short ParMarkSplit and placed it as a collector for the old generation in HotSpot, like CMS.

One implementation issue of ParMarkSplit based on the CMS is that CMS is dedicated to work for the old generation. This brings difficulty for a plain comparison of the two algorithms in which ParMarkSplit and CMS are used to collect a whole heap. We considered disabling the generational option in HotSpot and having ParMarkSplit or CMS as the only garbage collector for the whole heap. However, disabling the generational option would have required thorough changes for the whole memory management system that would have touched many parts of the HotSpot. We saw that it was extremely complicated to perform such changes. Moreover, keeping the heap divided into two generations allowed us to compare the two collectors in an industrial standard environment and application.

5.5 Evaluation

We present an experimental evaluation of our parallel collector and compare it with the default CMS collector present in OpenJDK HotSpot, using the DaCapo benchmarks. Subsection 5.5.1 sketches the methodology of our evaluation, followed by two subsections presenting our experimental evaluation results in two scenarios. Then we discuss about the memory overhead and characterization of applications that can benefit from ParMarkSplit in the last two subsections.

5.5.1 Evaluation Methodology

ParMarkSplit was implemented based on CMS. So it was natural to compare ParMarkSplit with CMS. We also considered comparing ParMarkSplit with its sequential implementation in [14]. However, we could not do so because we do not have access to that sequential implementation due to licence restrictions.

We evaluated ParMarkSplit and CMS in two scenarios. In the first scenario, we set up HotSpot with ParMarkSplit or CMS garbage collector working in a stop-the-world mode, in which the mutator was stopped during collection. This setting allowed us to exclude the synchronization cost of the old GC with the mutator and the young generation collection. Such an execution provides a better look at the performance of the design itself. In the second scenario, the garbage collectors were evaluated in concurrent mode in which the mutator running concurrently with garbage collection. The younger generation can promote objects and applications can produce new garbage during the old generation collection. The garbage collection worked as described in subsection 5.2.2.

The DaCapo suite [2] was used for benchmarking. DaCapo contains a set of open-source, general-purpose Java Virtual Machine benchmarks. DaCapo is representative of real-world Java applications. We ran experiments on several benchmarks and present the results from five benchmarks which have more memory accesses, as tested by Gidra L. et.al [5] and Kalibera T. et al. [9]: *avrora*, *lusearch*. *sunflow*, *tomcat* and *xalan*. For other benchmarks, most of their garbage are young and collected by the young generation collector. We did not observe any significant performance change of those benchmarks when applying our GC for the old generation. Most Java applications nowadays use large heaps and, in a long run, generate a lot of garbage. When we ran some preliminary experiments, we however found that the benchmarks use much less memory than our available memory and do not produce a lot of garbage in the old generation. Too big heap might never trigger any old generation collection, though the young generation collection could be triggered several times. In order to focus on garbage collection of the old generation, we chose the heap sizes that are close to the benchmark's working set size. They were 50 megabytes (MB) for avrora, 100MB for lusearch, tomcat, sunflow and 400MB for xalan. CMS collector and multi-threaded concurrent phases are enforced by setting the corresponding flags. The other flags were left on default values.

The experiments were run on two contemporary multiprocessor platforms. The first one has two Intel Nehalem 6-core processors running at 2.4GHz with HyperThreading, which can support up to 24 concurrent hardware threads, has 48GB of RAM. The second one has four AMD Bulldozer 12-core processors running at 2.6GHz, which can support up to 48 concurrent hardware threads, has 64GB of RAM. Both machines ran Ubuntu Linux with kernel 3.0.0. We compared 4 collectors: ParMarkSplit with and without lazy-splitting, ParMarkSplit based on coarse-grained locking balanced binary tree and the OpenJDK's CMS collector. In each experiment, we iterated a benchmark six times so that the old generation's collector can perform collecting for some cycles. We replicated each experiment 5 times and plotted the average result. As we are mostly discussing garbage collection in the old generation part, we use the term *garbage collection* or *old GC* in the following text. Other use cases will be clearly stated.

App Threads	avrora	lusearch	sunflow	tomcat	xalan
6	52.5%	28.2%	45.6%	24.8%	6.1%
12	49.3%	29.6%	54.7%	24.3%	4.4%
18	48.7%	32.1%	60.9%	24.2%	5.2%

Table 5.1: The reduced amount of *s*plit_interval operations performed when applying the lazy splitting mechanism to ParMarkSplit

5.5.2 Stop-the-world Scenario

In the stop-the-world scenario, we evaluate the lazy splitting mechanism and collection time that our ParMarkSplit spends on collecting garbage in five applications in the DaCapo benchmarks. We varied the number of threads running the application (App Threads) from 6 to 18. The number of threads that collect garbage (GC Threads) was also varied but was always not more than the number of application threads. Two ParMarkSplit implementations were used, one without lazy splitting (PMS) and the other with lazy splitting (PMS_O). Since collection works when the mutator was stopped, all GC threads do marking and splitting concurrently.

We first evaluate the lazy splitting mechanism by measuring the number of *split_interval* operations performed by ParMarkSplit in each collection cycles before and after adopting lazy splitting mechanism. In general, ParMarkSplit with the lazy splitting mechanism performed fewer *split_interval* operations. In avrora and sun flow, lazy splitting can reduce the number of *split_interval* operations by around 50%. But in xalan applications, the reduction is only about $4 - 6\%$ (see Table 5.1). It was because live objects in xalan interleave with garbage. Therefore lazy splitting can not reduce the number of calls to *split_interval* as much as in other applications. We expect that the lazy splitting mechanism benefits ParMarkSplit, in term of collection time, the most in avrora and sunflow. Similar benefits of lazy splitting is observed on both AMD and Intel systems.

The benefits of lazy splitting are reflected in the performance of ParMarkSplit collector. Figure 5.2 presents the collection time of different garbage collectors in the HotSpot in both Intel and AMD systems. In four out of five bench-

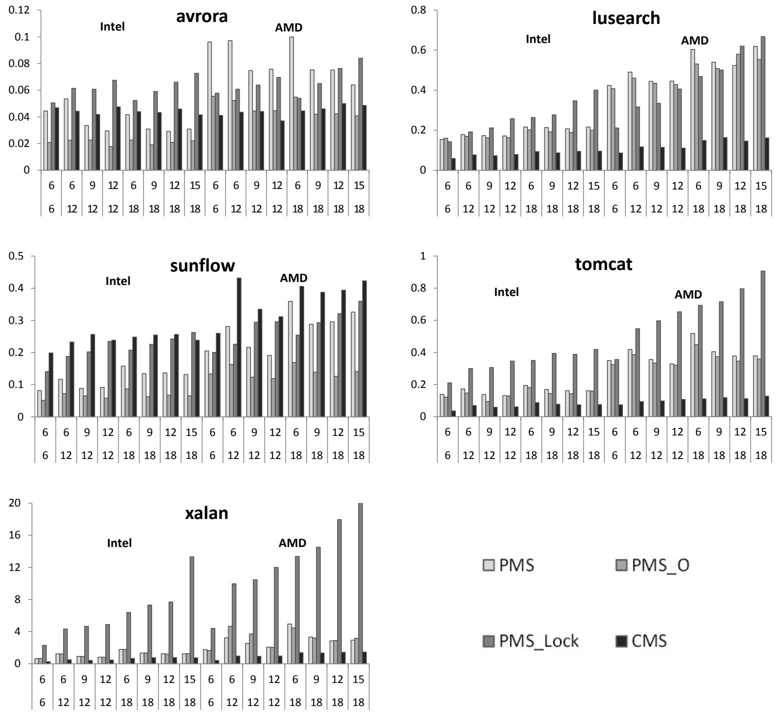
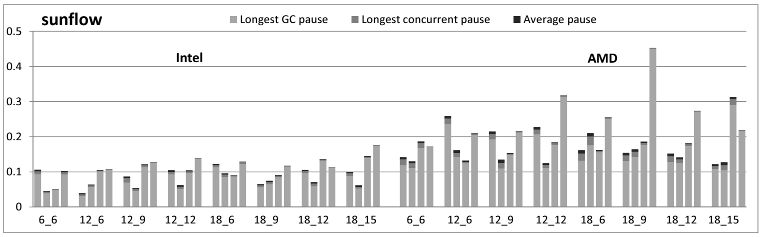


Figure 5.2: Garbage collection time in the stop-the-world scenario
(X-Axis, lower label: App Threads, upper label: GC Threads.)

marks on the Intel system, lazy splitting helps reducing the collection time of ParMarkSplit, especially in avrora and sunflow. Only in xalan, the improvement of lazy splitting are not clear as the benefit of lazy splitting is not enough to pay-off for its overhead cost. We also compared the collection time of ParMarkSplit with the trivial lock-based implementation of parallel mark-split (PMS_Lock) and with the HotSpot's CMS. In all the applications, the two ParMarkSplit implementations, with or without lazy splitting, perform significantly better than the lock-based one. ParMarkSplit performs better than CMS in avrora and sunflow, but not in the other applications. We notice that avrora and sunflow are the two applications that have higher ratios of adjacent marked objects over the total number of live objects compared to the other DaCapo applications. This result in accessing the same intervals for a short time which ca benefit ParMarkSplit from the caching effect. All above observations are also hold for the AMD system.

In addition, the scalability of garbage collectors can be observed from Figure 5.2. We track the collection time as we increase the number of GC threads. At the same time we vary the number of application threads. The lock-based parallel mark-split GC does not scale at all on both Intel and AMD systems. Meanwhile, the ParMarkSplit collectors, with and without lazy splitting, are scalable up to 12 GC threads in most applications. Both collectors have their collection time decreasing when the number of GC threads increases from 6 to 9, 12 in avrora, lusearch, tomcat and xalan applications on both Intel and AMD systems. We have not observed HotSpot's CMS collection time to decrease when the number of GC threads is increased.

Comparing the performance of ParMarkSplit between the two systems, we found that ParMarkSplit performs better on the Intel one. One possible reason can be that the AMD system has four NUMA nodes. GC threads which were scheduled on different processors need more time to access the skip-list storing free intervals on the system with more NUMA nodes.



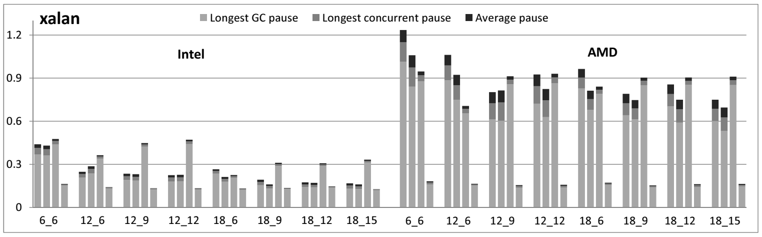


Figure 5.3: Pause time when the old generation's garbage collectors work concurrently with the mutator
- *Longest concurrent pause* when GC work concurrently with the mutator; *Longest GC pause* includes pauses when the collector switches to stop-the-world; *Average pause:* average of all the pauses by the old GC
- Labels on the x-axis are formatted as *n1_n2* where $n1$ and $n2$ are the numbers of application and GC threads, respectively. At each label, four columns represent four collectors (from left to right): PMS, PMS_O, PMS_Lock and CMS.

5.5.3 Concurrent Scenario

In the concurrent scenario, the garbage collector was configured to collect garbage concurrently with the mutator. This is the scenario that CMS was built for. ParMarkSplit is built based on CMS, which is optimized to reduce pause time of the garbage collector. We evaluate the pause times of our GC during the concurrent collection, in addition to the execution times of the benchmarks.

CMS suspends applications during the initial mark and remark phase. ParMarkSplit, which derives from CMS and adds the splitting part to these phases, is expected to have longer pauses than the corresponding CMS's pauses. This reflects in *longest concurrent pause* and *average pause*, which are pauses during concurrent collection, in Figure 5.3. The *longest GC pause* of the old generation garbage collector, including the pauses when the collector switches to working in stop-the-world mode [1] is also presented in the same figure. Due to the lack of space, we include only the results of sunflow and xalan applications, representing for applications which may or may not benefit from ParMarkSplit. We can observe that both average and longest concurrent pauses of ParMarkSplit are longer than the respective ones from CMS, as expected from the design. In current HotSpot, the initial mark phase was implemented to run sequentially. The remark phase, though can run multi-threaded, has many parts which were still running single-threaded. As these two phases were running mostly sequentially, the pause time in ParMarkSplit, which uses lock-free synchronization based on compare-and-swap operation, were penalized dramatically. We can expect that when these two phases are fully parallelized in the HotSpot, pause time of ParMarkSplit will be improved significantly, at least proportionally to the speedup of the lock-free skip-list. Regarding the garbage collection pause time, we also notice that the longest GC pause time does not follow the trends of the longest concurrent pause time across the applications. In sunflow, the ParMarkSplit with or without lazy splitting usually achieves shorter longest GC pauses than both the lock-based one and CMS. However, in xalan, the ParMarkSplit collectors have shorter longest GC pauses than the lock-based one, but longer than CMS. This observation can be drawn from both the AMD and Intel platforms. There are also different in term of absolute values between the two architecture. The AMD system usually have longer pauses than the Intel one.

Regarding the relation between the application's response time and the GC's pause time, it is noticeable that GC pause time is not necessarily the same as the application response time, which means how long it takes an application to responds to a request by users or by other applications. Even though pause time is an indicator for the maximum application response time in the worst case,

[1]GCs switch to stop-the-world mode in a collection cycle when it can not continue in concurrent mode because, for example, the old generation is full. In such cases, our customized HotSpot uses the same algorithm as in concurrent mode but running without mutator intervention

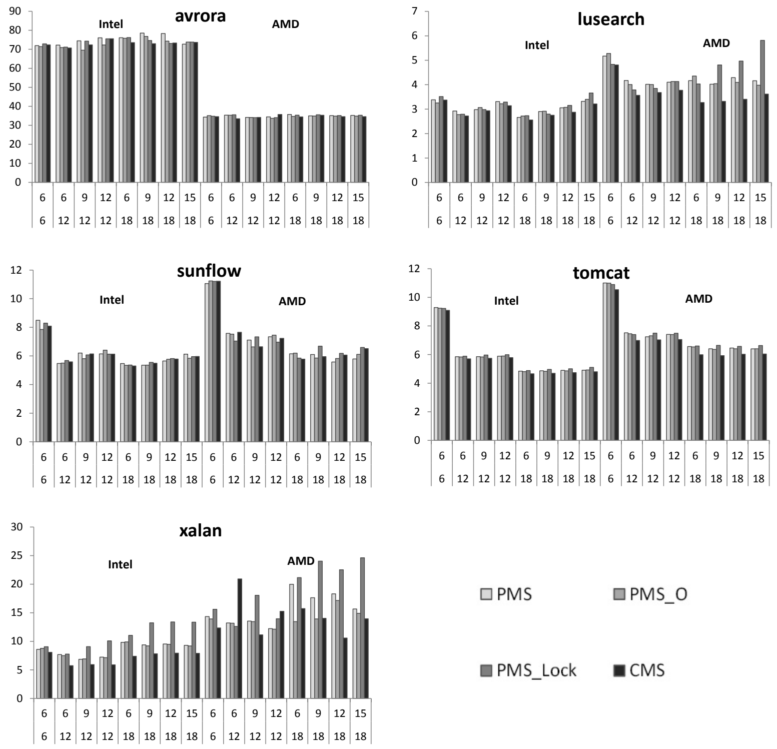


Figure 5.4: Benchmark time for the HotSpot with different concurrent GCs (X-Axis: upper label: App Threads, lower label: GC Threads.)

the contribution of the GC's pause time to the mean application response time is less and less important in systems with heavy loads, as studied by Persson M. and Cummins H. from IBM [12].

ParMarkSplit brings the split part to the mark phase but it also removes the sweep phase. Does this change reflect in the overall throughput of the applications? Figure 5.4 shows the average time to complete each benchmarks in different configurations of application threads and GC threads. We see that on the Intel system, the ParMarkSplit GC with lazy splitting performs better than CMS in sunflow and lusearch in many cases, and performs comparably well as

	avrora	lusearch	sunflow	tomcat	xalan
	Number of nodes (thousands) / Size (MB)				
Intel	2.0/ 0.3	14.4/ 2.1	4.9/ 0.7	49.1/ 7.1	55.0/ 7.9
AMD	2.2/ 0.3	16.6/ 2.3	4.3/ 0.7	46.4/ 6.7	57.6/ 8.3
	Estimated size of bitmap (MB)				
Bitmap	0.78	1.56	1.56	1.56	6.25

Table 5.2: The maximum size of the skiplist and estimated size of bitmap (Printesiz's technique)

CMS does in avrora and tomcat. Similar observations can also be drawn on the AMD system.

In this concurrent scenario, ParMarkSplit has shown that it works well in avrora and sunflow, both in terms of pause time and throughput. In addition to that, it achieves comparable to CMS in lusearch and tomcat on the Intel platform. ParMarkSplit also achieve higher benchmark results than the lock-based parallel mark-split in most cases.

5.5.4 Memory Usage

ParMarkSplit uses the lock-free skip-list to store free intervals during garbage collection, which introduces a memory overhead compared to CMS. In this subsection, we provide an estimation of this overhead on different applications.

We can estimate the memory used by the skip-list based on the number of free intervals stored in it. In the skip-list, each free interval is stored as a linked-list's node. A node needs 18 memory words: two words for the start and the end of the free interval, one for the node's level in the skip-list, and at most max_level pointers pointing to the next nodes in the linked-list at each level of the skip-list. max_level is decided when the skip-list is constructed so that 2^{max_level} is approximately the average size of the skip-list. As our estimated average number of free intervals is 32000, max_level is 15. The estimated memory used by the skip-list in a 64bit system is presented in Table 5.2.

We observe that avrora and sunflow have the lowest number of free intervals among the benchmarks. This is because their live objects often reside adjacent to each other as we discuss in the above evaluation. The memory overhead in avrora and sunflow is less than 1% over the heap size (0.3/50MB and 0.7/100MB, respectively), which is negligible. This cost is higher in applications that the numbers of free intervals are high, approximately 2% in lusearch and xalan, and 7% in tomcat, where the heap size are 100MB, 400MB and 100MB respectively. The size of the skip-list varies depending on the running applications. It is small in applications that their live objects often reside adjacent to each others. In those applications, the memory overhead is negligible. Comparing to the memory overhead of Printezis's technique which uses a bitmap to skip over contiguous unmarked objects while sweeping [13], ParMarkSplit uses less memory in avrora and sunflow, but more in other benchmarks.

We also observe that the fragmentation behavior of ParMarkSplit is similar that of CMS, as it is expected by design. It is possible to check the fragmentation level during or after a collection cycle by checking the size of the skip-list. When the heap is considered too fragmented, a compaction algorithm can be applied in a similar way as it is applied in CMS garbage collector in the HotSpot.

5.5.5 Characterization of Applications that Benefit from ParMarkSplit

Comparing our Parallel Mark-Split garbage collector with CMS present in the HotSpot, ParMarkSplit performs better in some applications in term of collection time. In this subsection, we try to characterize the applications in which ParMarkSplit performs better so that the system can use this characterization to select the best garbage collector based on the characteristics of the application.

We have observed, from the experimental results on the two hardware platforms, that ParMarkSplit outperforms CMS in sunflow and the avrora applications. We have also observed that CMS performs better than ParMarkSplit in tomcat, lusearch and xalan. ParMarkSplit performance is highly dependent on its most frequent operation, i.e *split_interval*. As a consequence, ParMarkSplit usually performs better in applications where ratio of the number of live objects to the total number of objects are low compared to those of garbages objects. Analysis on the live/garbage ratio of those applications shows that, sunflow and avrora have low live/garbage ratios which are 15% and 20% on average respectively. Tomcat have higher live/garbage ratio of 40% on average. ParMarkSplit maintains a property of sequential mark-split algorithm that the algorithm performs better in applications that have low live/garbage ratio. However, this property could not be applied to explain about ParMarkSplit's performance in other applications with the same characteristic.

Xalan and lusearch also have similar live/garbage ratios as sunflow and avrora but ParMarkSplit does not perform well in those applications. We need to distinguish the former from the latter and characterize better the application that clearly benefit from ParMarkSplit. We observed that our lazy splitting design that splits space occupied by consecutive marked objects in one atomic operation brings significant performance gains to ParMarkSplit garbage collector in applications that it already performs better than CMS, i.e. sunflow and avrora. The benefit of the design in xalan and lusearch is not as much as it is in sunflow and avrora. A characteristic that differentiates the two groups is the ratio of the number of adjacent marked objects over the total number of marked objects. This ratio is high in the case of sunflow and avrora; and lower in xalan and lusearch. When this ratio is high, doing splitting interval operation in ParMarkSplit benefits in two ways. First one is cache benefit when a free interval that is previously split can be cached and reused in the next splitting. Second benefit is that adjacent objects can be collected together and one split interval is needed for all of them. These advantages from ParMarkSplit are more in sunflow and avrora than in xalan and lusearch. As a consequence, ParMarkSplit performs better than CMS in the two former applications but not in the two latter ones in our experimental evaluation. All above observations regarding the characterization of applications that benefit from ParMarkSplit are consistent across the two different hardware platforms.

To conclude, ParMarkSplit has been shown to perform better than CMS in applications where the ratio of the number of live objects over that of garbage objects is low and live objects often reside adjacent to each other. ParMarkSplit can be used as a complement to other garbage collection mechanisms to target applications with such characteristics.

5.6 Conclusion

We present a parallel design of the mark-split garbage collector. To the best of our knowledge this is the first parallel mark-split design. This design is based on a lock-free data structure that extends the functionality of a skip-list to meet the requirements of the mark-split algorithm augmented with a lazy splitting design. A complete implementation of the parallel mark-split collector was developed and integrated in the OpenJDK HotSpot. We evaluated experimentally the behavior of our parallel mark-split collector and compared it with the default concurrent marks-sweep garbage collector present in the HotSpot, using the DaCapo benchmarks. The experiments were performed on two multiprocessor systems of different architectures: Intel's Nehalem and AMD's Bulldozer. The results are encouraging in applications where the ratio of the number of live objects over that of garbage objects is low and live objects often reside adjacent to each other. ParMarkSplit can be a complement to other garbage collection mechanisms when used for applications with such characteristics.

Bibliography

[1] A. Andersson. Balanced search trees made simple. In *Proceedings of the Third Workshop on Algorithms and Data Structures*, pages 60–71, London, UK, 1993. Springer-Verlag.

[2] S. M. Blackburn and et al. The dacapo benchmarks: java benchmarking development and analysis. *SIGPLAN Not.*, 41:169–190, October 2006.

[3] S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. *SIGPLAN Not.*, 43(6):22–32, June 2008.

[4] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13:677–678, November 1970.

[5] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. Assessing the scalability of garbage collectors on many cores. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, PLOS '11, pages 7:1–7:5, New York, NY, USA, 2011. ACM.

[6] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A study of the scalability of stop-the-world garbage collectors on multicores. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 229–240, 2013.

[7] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[8] R. J. M. Hughes. A semi-incremental garbage collection algorithm. *Software: Practice and Experience*, 12(11):1081–1082, 1982.

[9] T. Kalibera, M. Mole, R. Jones, and J. Vitek. A black-box approach to understanding concurrency in dacapo. Presented at the UK MM-NET workshop on Memory Management for Many-/Multicore, April 2012.

[10] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3:184–195, April 1960.

[11] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(8), Aug. 2004.

[12] M. Persson and H. Cummins. Java technology, ibm style: Garbage collection policies.

[13] T. Printezis and D. Detlefs. A generational mostly-concurrent garbage collector. *SIGPLAN Not.*, 36:143–154, October 2000.

[14] K. Sagonas and J. Wilhelmsson. Mark and split. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 29–39. ACM, 2006.

[15] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. A lock-free algorithm for concurrent bags. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 335–344, New York, NY, USA, 2011. ACM.

[16] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.*, 65(5):609–627, 2005.

[17] P. Tsigas and Y. Zhang. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. *SIGMETRICS Perform. Eval. Rev.*, 29:320–321, June 2001.

PAPER V

Nhan Nguyen, Lokesh Gidra, Gäel Thomas, Julien Sopena, Marc Shapiro.

A NUMA-Aware Parallel Mark-Compact Garbage Collector

Technical Report 2014:04, ISNN 1652-926X,

Department of Computer Science and Engineering Chalmers University of Technology, 2014.

6

PAPER V - A NUMA-Aware Parallel Mark-Compact Garbage Collector

Abstract

The lack of awareness of Non-Uniform Memory Access (NUMA) architectures can prevent stop-the-world throughput-oriented garbage collectors from scaling in large-scale multicore architectures. An introduction of NUMA-awareness to Parallel Scavenge, a young generation garbage collector of HotSpot virtual machine, has resulted in a scalable performance of the collector up to 48 cores. However, the old generation and its collectors remains NUMA-oblivious which causes memory imbalance and bad locality of both the collector and the application. In this work, we address performance issues of throughput-oriented garbage collector for the old generation in OpenJDK 7, called Parallel Mark-

Compact, in NUMA multicores. We then augment the old generation with the fragmented space policy and Parallel Mark-Compact with NUMA-awareness. Together they help maintain memory balance in the old generation and ensure good locality for both the collector and the application.

6.1 Introduction

Multicore processors with many cores have been developed by different multiprocessor manufacturers. The new architecture with high core counts and large, possibly distributed, memory create several challenges to the garbage collection. The challenges appear in different aspects such as in concurrency control, e.g contention, synchronization bottlenecks; in memory management, e.g memory placement, locality and balance; as well as the interplays among them. To exploit the parallelism, throughput-oriented garbage collectors (GC) opt for the parallel and stop-the-world design, in which the applications are suspended to stop their memory accesses while parallel GC threads are collecting the garbage. Meanwhile, concurrent collectors try to garbage collect concurrently with application execution, and therefore have to synchronize with the application's memory accesses. Such a design is more complicated as it requires fine grained and continuous synchronization between the GCs and the applications.

A recent study by Gidra et al. [7] discovered that the performance of the garbage collectors (GC) in OpenJDK7, both stop-the-world and concurrent ones, degrades beyond about 8 GC threads. The follow-up study [8] by the same authors introduced a set of improvements which can be simply implemented yet bring scalability of the stop-the-world GC in OpenJDK's HotSpot virtual machines. In more detail, the study which is based on the Parallel Scavenge collector concludes that the lack of NUMA-awareness and heavy synchronization based on contended locks in the GC are the bottlenecks for the scalability of PS. The lack of NUMA-awareness of the GC causes memory access imbalances and bad locality for both the application and the GC. The authors proposed three space policies, called interleaved, fragmented and segregated spaces to address those issues. A NUMA-aware Parallel Scavenge collector was introduced based on those policies together with the replacement of contended locks with lock-free solutions. The new collector is able to scale up to 48 GC threads in a contemporary 48 cores NUMA machines and significantly decreases the GC pause time.

While generational GC has been widely employed, especially in Java Virtual Machine, the study addresses only the scalability bottlenecks in the young generation. The old generation and the Parallel Mark-Compact (PMC) collector, which is used for the whole-heap, i.e. major, collection, remain NUMA-oblivious. The old generation consists of a single contiguous memory space which is initialized to a large virtual address range. Under the kernel's memory allocation policy, virtual memory ranges are mapped to physical memory pages from different NUMA nodes when being accessed for the first time. Thereafter, the association between a virtual address range and a physical memory page remains unchanged. The PMC collector performs compaction by moving all live objects to one end of the old generation space, without taking either the object's location or physical distribution of old space's memory into consideration. As a results, GC threads usually perform unnecessary remote copies, in which either the source or the destination, or both are located on a remote node. In addition, an object's new location can be on an arbitrary memory node which might be different from the node it is allocated or is mainly accessed from, resulting in poor locality for the application.

The lack of NUMA-awareness of the old generation also has a negative impact on the young generation collector. HotSpot uses a card table to record references from the old to the young generation. The card table is treated as a root set during the young generation collection. As PMC moves old objects to one end of the space, the GC threads may have to process objects remotely during young generation collection.

In short, the lack of NUMA-awareness of the PMC causes poor locality for both the garbage collector and the application. In this work, we present a NUMA-aware memory placement to the old generation and a NUMA-aware PMC collector. The rest of the paper is organized as follows. Section 6.2 de-

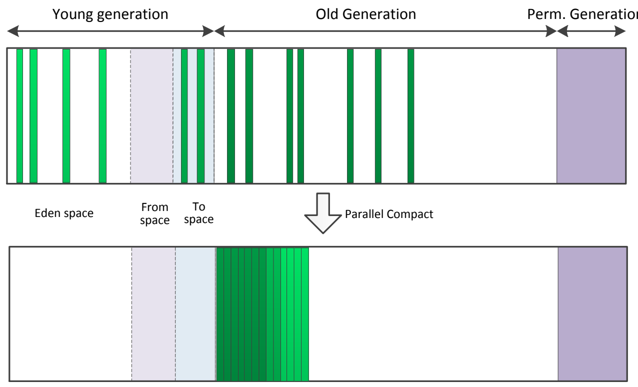


Figure 6.1: Heap Layout of the HotSpot for Parallel Collector

scribes the Parallel Throughput Collector (PC), the throughput-oriented collector of the OpenJDK's HotSpot. In Section 6.3, we summarize the space policies for the heap layout used for NUMA multicores introduced by Gidra et al. [8]. Our proposed NUMA-aware PMC collector is introduced in Section 6.4. Section 6.5 presents some preliminary evaluation results before we conclude the paper in Section 6.7.

6.2 Parallel Throughput Collector

This section describes the Parallel Throughput Collector (PC) in the HotSpot. We start with some overview information regarding the heap arrangement and memory allocation policy. Parallel Scavenge (PS) - the GC for the young generation and PMC - the GC for whole-heap, i.e major, collection are described next.

6.2.1 Heap layout

The PC operates on a generational heap layout consisting of three generations as presented in fig. 6.1. The young generation has one eden space and two survivors spaces, called from-space and to-space. This arrangement suits the young-generation copying collector, i.e. PS, which is usually triggered when the young generation is full. When a mutator, i.e an application thread, allocates a new object, it resides in the eden space. Objects in eden space which survive the first young generation GC are copied to one of the survivor spaces. Objects already in the survivor spaces which survive a GC are promoted to the old generation.

The old and the permanent generations consist of one single space each. The old space contains objects that were promoted from the young generations and the permanent space is used for class definitions and associated meta-data.

6.2.2 Allocation

Each mutator thread allocates a large memory chunk for its exclusive use called Thread Local Allocation Buffer (TLAB) from the eden space. From then on, it allocates objects from its TLAB without having to synchronize with other mutator threads. Surviving objects promoted to the survivor spaces or to the old spaces are allocated in a similar manner, i.e from the Promotion-Local Allocation Buffer (PLAB) which has been allocated from the to-space or the old generation.

6.2.3 Young Generation Collection

The young generation collection which uses Parallel Scavenge is triggered when a mutator fails to allocate a new TLAB. At this point, the eden space is mostly full of recently allocated objects; the from-space contains objects that have survived the previous collection cycle and the to-space is empty. The young collection copies live objects from the eden space to the to-space, and from the from-space to the old generation. When the collection is completed, the eden space and the from-space contains only dead objects and are considered empty. The from-space and the to-space then flip their roles: the from-space before the collection becomes the to-space after that and vice versa. Further details regarding Parallel Scavenge and the synchronization among GC threads, as well as between GC and the mutator has been mentioned in literature, such as [8].

6.2.4 Old Generation Collection

When the old generation is full, a full collection is triggered to garbage collect the whole heap. The HotSpot's throughput-oriented GC uses the Parallel Mark-Compact (PMC) collector, which implements a parallel mark-compact algorithm.

PMC collects garbage by moving all live objects to one end of the old space. First, GC threads trace the object graph and mark the live objects in parallel starting from the root sets. Then, they move all the live objects of the old generation to the beginning end (aka left end) of the old space to fill up the holes created by dead objects. Live objects from the young generation are then moved next to those old live objects in the old generation. The other memory chunks not occupied by live objects are free and can be combined with the pre-collection free memory in order to use for allocation.

In detail, the collector works in four phases:

- The marking phase: marks all the reachable, i.e. live, objects starting from root sets.
- The summary phase: calculates the destination, i.e new location, of each object at the end of the collection.
- The compaction phase: moves objects to their destination and updates references to point to new locations.
- The clean-up phase: updates the remaining references which have not been processed in previous phases to point to new locations.

Regarding the parallelization of the collector, the marking phase works in parallel because the marking of objects and processing their references to other objects can be done independently. The summary phase works in single-threaded mode. It is reasonable as the amount of work is small and the calculation of the new locations of objects are dependent on each other. After the

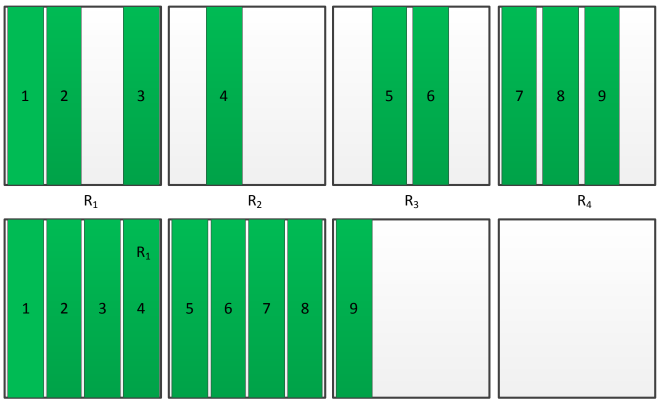


Figure 6.2: An example of regions of Type A (R_1), Type B (R_2, R_3) and type C (R_4) before (upper) and after (lower) compaction

summary phase, all the objects' new locations are determined. The compaction then moves the objects to their new locations accordingly. Parallelization of this phase is more complicated as new locations of objects can be currently occupied by other objects which should be moved else where.

To allow compaction to work in parallel, PMC divides the space into several fixed size regions (512 bytes each in PMC). Before compaction, the regions contains both live objects and garbage objects. When the summary phase finishes, each region can be categorized into one of three types (see fig. 6.2):

- Type A: data from the region is compacted completely into itself, or the region is empty. The region can be claimed and, probably, is filled with objects.
- Type B: data from the region is compacted into 1 other region; some data from the region may also be compacted into the region itself.
- Type C: data from the region is copied to 2 other regions.

The compaction then processes the regions. Type-A regions can be claimed and processed, i.e filled, first. Objects which fills type-A regions are from regions of type-B and type-C. As type-A regions are filled, more type-B and -C regions become type-A ones. Eventually, all regions are processed.

As the number of collected regions are massive, parallelism is achieved by assigning an equal amount of type-A regions to each GC thread at the beginning of the compaction phase. Each thread maintains a region stack to store the assigned regions. As a thread processes the assigned type-A regions, it actually moves data away from type-B/-C (to the processed regions), which results in the transformation of type-B/C regions to type-A regions. It pushes these new type-A regions into its stack. When it finishes processing regions in its stack, a GC thread randomly chooses another GC thread to steal unprocessed regions. Compaction finishes when all regions are processed, either filled or claimed. At this point, all live objects have already moved to their new locations. The free regions claimed are coalesced with the pre-collection free memory.

6.3 The Old Generation NUMA Space

This section starts with a description of current issues of the old generation space. It continues by describing the NUMA-aware space policy called fragmented spaces and how it is applied to achieve a NUMA-aware old generation.

6.3.1 Current Old Generation Space

PMC reserves a large virtual address space for the old generation. The actual physical memory is lazily mapped to a virtual memory using, in the case of Unix-like operating systems, *mmap* system calls. A virtual memory page is mapped to a physical memory page when it is first accessed. The mapping prefers the physical memory page on the node where the fault is triggered. Thereafter, the virtual address range of the page remains associated to the same physical node.

In most cases, objects in the old generation are allocated by the GC threads to promote young objects to the old generation during young GC. To avoid synchronization among GC threads at every allocation, a GC thread allocates a memory chunk in the old generation called Promotion-Local Allocation Buffer (PLAB). Thereafter, it allocates objects from this buffer, without synchronization, using a bump pointer. Only the allocation of PLAB requires synchronization among all GC threads, therefore uses an atomic compare-and-swap.

6.3.2 Performance Issues

The combination of the mapping policy and the use of PLABs results in a memory locality issue during the collection of the young generation. In normal circumstances, GC threads are distributed among nodes and they roughly promote the same amount of data; therefore, PLABs are allocated from different nodes, roughly, in a round robin manner. This behavior, together with the above mapping policy, lead to a result that address ranges of the old space are mapped to different memory nodes in an interleaving manner. Once mapped, the association of a virtual address range to a physical memory page remains unchanged. When a GC thread copies an object to a PLAB, the possibility that physical memory of the PLAB locates at a remote node is high; because a GC thread in large-scale multicore systems usually has more than one remote node. As a result, most copies performed by GC threads when promoting young objects to the old generation are remote.

In addition, the unawareness of object locality affects negatively the performance of the young GC, i.e. Parallel Scavenge. HotSpot uses a card table to store all references from old to young generation. During the PS collection, the card table is divided evenly into several chunks and each GC thread processes one of the chunks, without any consideration of object locality. As each range contains physical memory located on different nodes, a GC thread often has to scan remote objects, resulting in a saturation of the inter-connection among nodes.

PMC also encounters a memory locality issue during compaction when old generation contains only a single space. A GC thread performing compaction fills a type-A region with objects from other, i.e. type-B/-C, regions. When either the destination or both source and destination locate on a node other than where the GC thread is executing, the thread performs remote copies. The likelihood of this happening is high because, as mentioned, a GC thread has one local node but usually more than one remote node.

6.3.3 Fragmented Space

Gidra et al. [8] addressed memory locality and access imbalance in the young generation by introducing a NUMA space layout called fragmented spaces. The following paragraph describes the policy.

Fragmented space policy divides a space into multiple fragments where each fragment gets all it physical memory from a single node. Under this policy, a thread always allocates objects from the fragment on the node where it is executing. This policy helps to improve the locality for the mutator as a thread accesses mostly the recently allocated objects. It also improves locality for the GC as an object is copied to the memory node where the thread performing copy is executing. Fragmented spaces improve memory access imbalance in two ways. Firstly, when the allocation policy for each segments is fixed, a space has its physical memory from multiple nodes instead of one single node. Secondly, work-stealing during GC ensures that each GC thread copies roughly the same number of bytes, therefore balances the distribution of objects among nodes.

6.3.4 NUMA-aware Old Space

The issues discussed in subsection 6.3.1 are partly solved by introducing a NUMA-aware old space based on fragmented spaces. The space consists of a number of fragments where each fragment is a virtual address range that get its physical pages only from a single node. A GC thread allocates PLAB from the fragment associated with the node where it is executing. As a result, any GC thread promoting objects from the young to the old generation always copies them to the local memory node. Scanning the card table to discover references from the old to the young generation can also be adapted to perform in a per-node manner. GC threads running on a node process only the part of the table that associates with its local fragment.

The main objective of the introduction of the NUMA-aware old space layout is to create a NUMA-aware GC for the old generation. In the next section, we present our NUMA-aware Parallel Mark-Compact, which is built on top of the current PMC in the HotSpot.

6.4 NUMA-aware Parallel Mark-Compact

In this section, we present our modification to the PMC collector to make it NUMA-aware. As current PMC treats all spaces as a single contiguous fragment, a GC thread usually performs a remote copy where either or both the source and destination addresses are on some remote nodes, other than where the thread is executing. In addition, PMC also has a negative impact on memory locality of the mutator. An object, after the compaction of a collection cycle, resides on a node different from where it was allocated or where it resided, before the collection cycle. With the transformation of all spaces to NUMA-aware fragmented spaces, the memory locality of the compaction can be improved. The idea is that objects from a fragment associated to a node are copied, i.e "compacted", to the old fragment on the same node. Under this policy, copying of objects during a PMC collection is always local. Memory locality of the mutator are also improved because objects locates on the same nodes before and after collection.

The NUMA-aware PMC works in the same phases as the original PMC, which we are going to described next.

6.4.1 Marking Phase

GC threads perform breadth-first-traversal of the graph of live objects, starting from the root set. When reaching an object, a GC thread marks the object as *alive* by setting the associated bit in the bit map. While traversing, GC threads follow the references to objects in any nodes. Restricting the traversal to be local can help the marking phase achieves perfect locality. When a GC thread discovers a remote object, it send a message to GC threads running on the respective remote node so that they can process, i.e marking, scanning and/or copying, the object locally. However, this approach, which was proposed in [8], did not bring significant improvement to PS in practice. In the case of PMC, marking threads only perform marking and scanning objects, without copying; the possible benefit must, therefore, be even lower or none at all compared to achieved benefit in PS. This motivates us to keep the marking phase unchanged.

6.4.2 Summary Phase

The summary phase collects information to prepare for compaction. The fundamental idea of the compaction is to divide the heap into several small fixed-size regions so that the compaction can process several regions in parallel. In PMC, the heap space is divided into 512 byte regions and the summary phase calculates summary data for every region. The most important summary data for a region are the *source_address* (i.e., where the live data to fill in the region comes from) and the *destination_count* (i.e., the number of regions that live data in this region will be copied to). As each space is divided into fragments and an object is only moved among fragments associated with the same node, the summarization can be done on each node separately. The NUMA-aware PMC compacts a fragment similar to the way the original PMC treats a space. All live objects in the old fragment associated with a node are compacted to the beginning of the fragment. The objects on fragments associated to the same node on eden space and from-space are then copied next.

For each node, summary data is calculated for, in order, old, to, and eden fragments. Each fragment is divided into several fixed size regions and the calculation of the new addresses works in the manner of per region, rather than per object. A GC threads scans all regions from the beginning to the end of a fragment. The live data in a scanned region is expected, during the compaction, to be copied next to the live data of the previous region. As a result, it is simple to update a region that is expected to contains live data after compaction with the starting address of that live data, i.e., *source_address*. When the summary phase is completed, the *source_address* of any region that are supposed to contain live data after compaction are identified. Each scanned region is also identified as Type-A, -B or -C (see Section 6.2), according to its *destination_count*.

When the heap is almost full, an old fragment on a node sometimes cannot accommodate all live objects on that node. The excess data should then be placed in another, e.g., eden or to, fragment. In this situation, special care is required to make sure an object is not separated into two fragments. The summary phase also identifies the *dense prefix* area, which is a part at the beginning of an old fragment that consists of consecutive regions with very high density of live data. Compacting these regions brings less benefit than the cost, and is therefore skipped.

The summary phase can be parallelized easily by assigning one task on each node to perform summarization for memory fragments associated with the node. On each node, the summary data of each fragment, i.e old, eden and from, can also be calculated in parallel. However, considering the amount of calculation work is small, we did not parallelize this phase in our implementation.

6.4.3 Compaction Phase

In this phase, a GC thread fetches a task from a global task queue which is synchronized using a GC monitor. Similar to the original PMC, the NUMA-aware PMC performs three kinds of tasks: fill-region tasks, update-dense-prefix-region tasks and steal tasks. However, the data fed for each task are different. We want that the regions processed by a task executed on a node locates on fragments associated to the same node. In this way, all copying is local.

Each fill-region task or update-dense-prefix-region task associates with a stack containing regions that the task should process. Before the parallel GC threads are woken up to perform compaction tasks, regions that are ready to be processed are pushed, in round robin manner, to stacks of the tasks. According to the task execution model of the virtual machine, tasks are queued in a global task queue and later, can be fetched by any active GC thread. Therefore, when regions are pushed to a task's region stack, the fact that which GC thread will perform the task is unknown. Neither is the node that the task will be executed on. Using the current task execution model, it is not possible to fix a node that a task will be executed on. In order to ensure that a task running on a node will process regions from the same NUMA-node, we need some modifications on the task execution model.

The modified model uses a two-level task queue and each task is attached with a preference executing node that indicates the node that the task should be executed on. The two-level task queue contains a global task queue accessible by any threads and multiple per-node task queues. Tasks in the global queue can be performed by any GC threads while tasks in a per-node task queue can only be performed by GC threads executed on the same node. The task manager will assign a task with a preference executing node to the appropriate task queue. The task is then executed by a GC thread running on that node. Our model distributes tasks among all nodes so that each node has a roughly equal amount of tasks. For convenience, we refers to a task with a preference executing node X as *X's task*.

Fill-region tasks

Fill-region tasks perform copying objects to their destination by filling in type-A regions and updating all references in the objects to new addresses.

Before the tasks start, we assign all type-A regions located on a node to fill-region tasks of that node in a round robin manner. When a GC thread executes a task, it pops the regions from the associated region stack and fills them with data from other, type-B/-C regions locating on the same nodes. The copying are totally local. When data of a type-B region is moved away, the region becomes type-A. The GC thread that performs the last data movement for such a type-B region pushes that region to its region stack. Similarly, a type-C region gradually becomes a type-B, and then a type-A region. When all these tasks are finished, all regions which needed to be filled with live objects are filled and empty regions are reclaimed.

When a GC thread copies data among regions, it processes references in the copied objects. If the thread copy the whole object, it updates all references in the object to new, i.e. post-compaction, addresses. It might also copy a partial object that spans over two or more regions. In such a case, updates of references of that object is deferred to the end of the compact phase, when all objects have been moved to their new locations. To reduce the amount of stored information for each object, PMC does not store the new address for any object. Instead, the new address of an object is calculated based on the information of a region in a dedicated routine.

Update-dense-prefix-region tasks
Similar to fill-region task, dense prefix regions on a node are assigned to tasks that are executed on the same node. Each task is assigned with an equal set of continuous regions. Each dense prefix task updates references in the assigned regions to the referenced objects' post-compaction addresses.

Steal task
The steal task performs stealing regions from other fill-region tasks, similar to PMC, to balance the load among GC threads and/or nodes.

6.5 Discussions

We have introduced a NUMA-aware policy to the old generation space and have also transformed the PMC collector to become NUMA-aware. The collector achieves almost perfect locality as most tasks, except for the marking tasks, are done locally. There is still room to improve the work. The summary phase can be easily parallelized, though the benefit might be marginal.

The NUMA-aware young GC roughly promotes the same amount of data to each fragment of the NUMA-aware old space and memory in the old space are allocated in a balance manner. However, when the old GC works, it compacts all fragments on a node locally. In case that the mutator does not allocate memory in a balanced manner, slight imbalances of memory among nodes is expected. Memory among node can be balanced by allowing compacting memory across nodes. If the calculation during the summary phase indicates memory imbalance, objects can be taken from heavy loaded nodes and moved to less loaded nodes.

6.6 Related Works

A great effort has been made to design parallel GCs, especially improving the load balancing mechanism. Halstead [9] developed a parallel version of Baker's semi-space copying GC for Multilisp on shared memory multiprocessors. During collection, the heap is logically partitioned into per-thread from-space and to-space, and a thread traces objects from its set of roots and copies them to its to-space. Since then, parallel copying GCs have been improved in many ways especially on the work-stealing mechanism among GC threads to balance the load, by Imai and Tick [10], Siegwart and Hirzel [12], Flood et al. [6], Attanassio et al. [2], Cheng and Blelloch [4].

Endo et al. [5] developed a parallel mark-sweep collector for share memory multiprocessors which balances the loads among GC threads in the mark phase by per-object work-stealing, and in the sweep phase by fine-grained partitioning of heap into several small blocks that are processed in parallel. Ben-Yitzhak et al [3] augmented a parallel mark-sweep collector with periodically selecting a certain area to clear, which reduces heap fragmentation. The mark-compact algorithm which performs compaction in three heap passes was first parallelized by Flood et al. [6], who divided the heap into several areas to be compacted in parallel by GC threads. The number of heap passes in the compaction has been improved to two passes by Aboaiadh et al. [1]. Kermany and Petrank [11] reduced the number of heap passes in the compaction to one, which is similar to the algorithm used in HotSpot's PMC collector.

Zhou and Demsky [13] propose a NUMA-aware parallel mark-compact collector targeting the TILE-Gx microprocessor family. The key design principle is that each core independently manages its own memory partition to minimize the coordination overheads and to improve the memory locality. While this algorithm focuses on memory locality, our proposal works together with the NUMA-aware young generation collector to balance the load while maintaining good memory locality.

6.7 Conclusions

In this paper, we have presented a NUMA-aware Parallel Mark-Compact garbage collector. The collector performs compaction in a per-node manner, allowing most of the work to be done on local memory nodes. The collector maintains the locality of memory after compaction the same as pre-compaction. In addition, the application of the NUMA-aware space to the old generation helps to keep memory allocation in old generation balance among nodes. The NUMA-aware old space brings positive impact for the scanning of the old to young references during young generation collection. There are other remaining bottlenecks for the scalability of the GC to be addressed such as memory imbalance. We are expecting a paper with all the improvements completed together with evaluation results soon.

Bibliography

[1] D. Abuaiadh, Y. Ossia, E. Petrank, and U. Silbershtein. An efficient parallel heap compaction algorithm. *SIGPLAN Not.*, 39(10):224–236, Oct. 2004.

[2] C. R. Attanasio, D. F. Bacon, A. Cocchi, and S. Smith. A comparative evaluation of parallel garbage collector implementations. In *Proceedings of the 14th international conference on Languages and compilers for parallel computing*, LCPC'01, pages 177–192, Berlin, Heidelberg, 2003. Springer-Verlag.

[3] O. Ben-Yitzhak, I. Goft, E. K. Kolodner, K. Kuiper, and V. Leikehman. An algorithm for parallel incremental compaction. *SIGPLAN Not.*, 38(2 supplement):100–105, June 2002.

[4] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. *SIGPLAN Not.*, 36(5):125–136, May 2001.

[5] T. Endo, K. Taura, and A. Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, Supercomputing '97, pages 1–14, New York, NY, USA, 1997. ACM.

[6] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 21–21, Berkeley, CA, USA, 2001. USENIX Association.

[7] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. Assessing the scalability of garbage collectors on many cores. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, PLOS '11, pages 7:1–7:5, New York, NY, USA, 2011. ACM.

[8] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A study of the scalability of stop-the-world garbage collectors on multicores. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 229–240, 2013.

[9] R. H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 9–17. ACM, 1984.

[10] A. Imai and E. Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *IEEE Trans. Parallel Distrib. Syst.*, 4(9):1030–1040, Sept. 1993.

[11] H. Kermany and E. Petrank. The compressor: Concurrent, incremental, and parallel compaction. *SIGPLAN Not.*, 41(6):354–363, June 2006.

[12] D. Siegwart and M. Hirzel. Improving locality with parallel hierarchical copying gc. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 52–63, New York, NY, USA, 2006. ACM.

[13] J. Zhou and B. Demsky. Memory management for many-core processors with software configurable locality policies. *SIGPLAN Not.*, 47(11):3–14, June 2012.