CHALMERS

Efficient Implementation of Concurrent Data Structures on Multi-core and Many-core Architectures

BAPI CHATTERJEE

Thesis to be defended in public on **March 13th**, **2015** at **13:15** in **room EA**, **Rännvägen 6**, Chalmers for the Degree of Licentiate of Engineering.
The defense will be conducted in English.

Discussion leader: Dr. Neeraj Mittal
Department of Computer Science, The University of Texas at Dallas, USA

The thesis is available at:
Division of Networks and Systems
Chalmers University of Technology
SE-412 96 GÖTEBORG, Sweden
Phone: +46 (0)31-772 10 00

Efficient Implementation of Concurrent Data Structures on Multi-core and Many-core Architectures

BAPI CHATTERJEE

Division of Networks and Systems
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Synchronization of concurrent threads is the central problem in order to design efficient *concurrent data-structures*. The compute systems widely available in market are increasingly becoming heterogeneous involving multi-core Central Processing Units (CPUs) and many-core Graphics Processing Units (GPUs). This thesis contributes to the research of efficient synchronization in concurrent data-structures in more than one way. It is divided into two parts. In the first part, a novel design of a Set Abstract Data Type (ADT) based on an efficient lock-free Binary Search Tree (BST) with improved amortized bounds of the time complexity of set operations - ADD, REMOVE and CONTAINS, is presented. In the second part, a comprehensive evaluation of concurrent Queue implementations on multi-core CPUs as well as many-core GPUs are presented.

Efficient Lock-free BST To the best of our knowledge, the lock-free BST presented in this thesis is the first to achieve an amortized time complexity of $O(H(n) + c)$ for all Set operations where $H(n)$ is the height of a BST on n nodes and c is the contention measure. Also, the presented lock-free algorithm of BST comes with an improved disjoint-access-parallelism compared to the previously existing concurrent BST algorithms. This algorithm uses single-word compare-and-swap (CAS) primitives. The presented algorithm is linearizable. We implemented the algorithm in Java and it shows good scalability.

Evaluation of concurrent data-structures We have evaluated the performance of a number of concurrent FIFO Queue algorithms on multi-core CPUs and many-core GPUs. We studied the portability of existing design of concurrent Queues from CPUs to GPUs which are inherently designed for SIMD programs. We observed that in general concurrent queues offer them to efficient implementation on GPUs with faster cache memory and better performance support for atomic synchronization primitives such as CAS. To the best of our knowledge, this is the first attempt to evaluate a concurrent data-structure on GPUs.

Keywords: Lock-free, Concurrent Data Structures, Lock-free Binary search tree, Synchronization Primitives

THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

Efficient Implementation of Concurrent Data Structures on Multi-core and Many-core Architectures

BAPI CHATTERJEE

Division of Networks and Systems
Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2015

Efficient Implementation of Concurrent Data Structures on Multi-core and Many-core Architectures
Bapi Chatterjee

Technical report 120L
ISSN 1652-876X
Department of Computer Science and Engineering
Distributed Computing and Systems Research Group

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

Author e-mail: bapic@chalmers.se

Printed by Chalmers Reproservice
GÖTEBORG, Sweden 2015

Efficient Implementation of Concurrent Data Structures on Multi-core and Many-core Architectures

Bapi Chatterjee
Division of Networks and Systems, Chalmers University of Technology

ABSTRACT

Synchronization of concurrent threads is the central problem in order to design efficient *concurrent data-structures*. The compute systems widely available in market are increasingly becoming heterogeneous involving multi-core Central Processing Units (CPUs) and many-core Graphics Processing Units (GPUs). This thesis contributes to the research of efficient synchronization in concurrent data-structures in more than one way. It is divided into two parts. In the first part, a novel design of a Set Abstract Data Type (ADT) based on an efficient lock-free Binary Search Tree (BST) with improved amortized bounds of the time complexity of set operations - ADD, REMOVE and CONTAINS, is presented. In the second part, a comprehensive evaluation of concurrent Queue implementations on multi-core CPUs as well as many-core GPUs are presented.

Efficient Lock-free BST To the best of our knowledge, the lock-free BST presented in this thesis is the first to achieve an amortized complexity of $O(H(n) + c)$ for all Set operations where $H(n)$ is the height of a BST on n nodes and c is the contention measure. Also, the presented lock-free algorithm of BST comes with an improved disjoint-access-parallelism compared to the previously existing concurrent BST algorithms. This algorithm uses single-word compare-and-swap (CAS) primitives. The presented algorithm is linearizable. We implemented the algorithm in Java and it shows good scalability.

Evaluation of concurrent data-structures We have evaluated the performance of a number of concurrent FIFO Queue algorithms on multi-core CPUs and many-core GPUs. We studied the portability of existing design of concurrent Queues from CPUs to GPUs which are inherently designed for SIMD programs. We observed that in general concurrent queues offer them to efficient implementation on GPUs with faster cache memory and better performance support for atomic synchronization primitives such as CAS. To the best of our knowledge, this is the first attempt to evaluate a concurrent data-structure on GPUs.

Keywords: Lock-free, Concurrent Data Structures, Lock-free Binary search tree, Synchronization Primitives

List of Papers

This thesis is based on the following manuscripts:

I **Bapi Chatterjee**, Nhan Nguyen and Philippas Tsigas, "*Efficient Lock-Free Binary Search Trees*", In the Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC 2014), Technical report no 2014:05, ISSN 1652-926X, Department of Computer Science and Engineering, Chalmers University of Technology, Sweden, Feb 2014.

II Daniel Cederman, **Bapi Chatterjee** and Philippas Tsigas, "*Understanding the Performance of Concurrent Data Structures on Graphics Processors*", In the Proceedings of the 18th International Conference on Parallel Processing, Euro-Par 2012, Lecture Notes in Computer Science Vol.: 7484, pages 883-894, Springer-Verlag 2012.

The following manuscript has been published in preparation of this thesis but not included here:

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 International Parallel and Distributed Symposium (IPDPS 2013), pages 1309-1320, IEEE Press 2013.

Acknowledgments

First of all, I would like to express my gratitude to my supervisor Prof. Philippas Tsigas for his constant support and guidance. Without his generous help, this thesis would not have been possible.

I also thank the committee members: Prof. Thierry Coquand, Prof. Marina Papatriantafilou and Prof. Gerardo Schneider for their kind support and helpful suggestions during the discussions in my PhD study follow-up meetings.

I also take this opportunity to thank my previous supervisors Prof. Subodh Kumar (Dept of CS&E, IIT Delhi, India) and Prof. Aparna Mehra (Dept of Mathematics, IIT Delhi, India), whose guidance during my masters studies helped me shape my thinking and motivated me enough to join the doctoral studies.

Next I would like to thank everybody in the Distributed Computing and Systems group, of which I am proud to be a member: Aras, Bashr, Elad, Ioannis, Iosif, Ivan, Giorgos, Olaf, Magnus, Paul, Thomas, Valentin and Vincenzo.

I thank former members of the group, with whom I spent some of the most enjoyable moments during my ongoing PhD so far: Andreas, Daniel, Farnaz and Oscar. My special thanks to my former office-mates, Nhan and Zhang, for the fun time and interesting discussions on topics confined to uncountable number of domains.

I thank Daniel Cederman for his collaboration and contribution to achieve the second paper included in this thesis.

I also take this opportunity to thank the staff and the PhD students at the Department of Computer Science and Engineering where I am privileged to be a member. It is thanking them for their efforts to make the department such a great place to work. I would like to specially thank Eva, Tiina and Peter for always being helpful and responsive.

Bapi Chatterjee
Göteborg, March 2015

Contents

List of Figures

List of Acronyms

ADT	Abstract Data Type
BST	Binary Search Tree
CAS	Compare-And-Swap
CPU	Central Processing Unit
DCAS	Double-Compare-And-Swap
FAϕ	Fetch-And-ϕ
GPU	Graphics Processing Unit
HTM	Hardware Transactional Memory
HPC	High Performance Computing
LL/SC	Load-Link/Store-Conditional
LSB	Least Significant Bit
NUMA	Non-Uniform Memory Access
SIMD	Single Instruction Multiple Data
TAS	Test-And-Set
TTAS	Test and Test-And-Set
TLAB	Thread Local Allocation Buffer
UMA	Uniform Memory Access

Part I

INTRODUCTION

1 Introduction

It is widely known now that in single-core compute processors higher computation performance could be achieved only by way of increased clock frequency and that would come with the drawbacks ranging from large power requirements to unmanageably high heat dissipation. A ubiquitous device to solve the optimization problem of maximizing the computation under the constraints of power-consumption and heat-dissipation in contemporary computers is a multi-core Central Processing Unit (CPU). To further enhance the processing of data-parallel components in a program, multi-core CPUs are supported by many-core co-processors such as Graphics Processing Units (GPUs). These co-processors can run hundreds of lightweight processing threads concurrently. The processing units in such co-processors often have their independent memory hierarchy to store the data to be processed close to compute units. The computer architecture comprising of CPUs and co-processors like GPUs i.e. heterogenous compute units with added heterogeneity in memory hierarchy is known as *heterogeneous computer architecture*.

However, harnessing the maximum available processing power in the machines equipped with multi-core and many-core processors is not plain sailing. An algorithm may not always lend itself for easy parallelization, specially when it involves *concurrency* of multiple threads to modify shared data. For example, consider the case of two threads that increment a shared counter which broadly

happens in three compute steps - (1) read the counter (2) increment the counter (3) store the incremented value at the shared memory word. We can list out all possible $\binom{6}{3} = 20$ valid interleavings[1] of the instructions by the two threads. But it is not hard to see that there can be only 2 valid interleavings which will increase the counter meaningfully. The problem of finding a valid interleaving that can resolve the conflict between concurrent threads to produce a correct solution of a problem involving concurrent access of shared memory is called *synchronization*. An algorithm is called a *synchronization algorithm* that solves the synchronization problem in a concurrent program. A synchronization algorithm becomes more complex with increasing number of shared memory words to be modified in order to accomplish an operation. And, the challenge of optimizing such an algorithm is immense because there is no guarantee provided by the implementation platform about the relative speeds of the threads.

A data-structure in a concurrent setup, usually, has to deal with the concurrent access of one or more shared memory words that are needed to be modified in an operation. The multi-core and many-core architecture, with more than a single level of hierarchy in the memory access, adds further complexity to it as the size of the cache memory comes to play important role in the performance of such data structures. Thus the design, analysis and implementation of *concurrent data-structures* on multi-core and many-core architectures is a reasonably challenging task. This thesis contributes towards a comprehensive description of the synchronization algorithms from the perspective of efficient implementation of concurrent data structures on the computers with both multi-core and many-core compute units. Rest of this chapter is as following. In section 1.1 a discussion on the design of the synchronization algorithms and concurrent data structure is presented. The section 1.2 discusses the correctness and the complexity of the concurrent data structures. The section 1.3 presents the fundamentals of the heterogeneous computer architecture. And finally, in the section 1.4 the chapter is concluded with a discussion on the related work and the contribution of this thesis. To describe a *shared-memory system*, we borrow terminologies and definitions from [10], however the discussion presented here is self-contend.

[1] There are altogether 6 instructions and a thread executes 3. We pick-up any 3 out of 6 in $\binom{6}{3}$ ways for one thread and 3 out of remaining 3 in $\binom{3}{3}$ for the second thread. Counter will be increased by 2 counts only if the first thread performs all its instructions in order before the second thread starts and does the same or vice-versa.

1.1 Synchronization Algorithm

The Synchronization algorithms are categorized in two classes - (a) *blocking* and (b) *non-blocking*. At the heart of a blocking synchronization algorithm is the *critical section* which is a piece of code that needs to be executed by multiple concurrent threads and in which threads need to access shared memory words. If the critical section is executed by more than one thread, it can result in unexpected and unwanted return by the concurrent program. Given the assumption that a thread can not fail or stop during its critical section and it can take only a finite number of steps in it, a blocking synchronization algorithm satisfies following properties -

(a) *Mutual Exclusion* : Two threads executing concurrently can not be in their critical section simultaneously.

(b) *Deadlock-freedom* : If a thread attempts to enter its critical section, then some thread, not necessarily the same one eventually enters its critical section.

To satisfy the property of mutual exclusion, when a thread is in critical section, all but one threads keep essentially blocked from executing the same and (consequently) the body of the program after the critical section. However, the available machines and popular programming languages for concurrent programs in general do not provide a guarantee that a thread can not become infinitely delayed (if not faulty) during the execution of the critical section. And that may result in a situation in which a thread gets delayed infinitely (i.e. becomes faulty) in the critical section, and other threads may remain blocked for that thread to come out of it. Thus the blocking synchronization algorithm are not very interesting quite often.

Alternatively a non-blocking synchronization method provides the guarantee that even if a thread becomes faulty in course of its execution path at any point, at least one non-faulty thread will complete its execution in finite number of its own steps. Essentially no thread needs to be blocked from executing any (atomic) *step* on any shared memory word in a non-blocking synchronization algorithm. A non-blocking synchronization algorithm is also known as a *lock-free* synchronization algorithm.

In case of blocking synchronization, the mutual exclusion property is called a *liveness property* and the deadlock-freedom is called a *progress property*. Another stronger and quite desirable progress property is

(c) *Starvation-freedom* : A thread, that attempts to enter its critical section, must eventually succeed.

A non-blocking synchronization algorithm that provides starvation-freedom is called a *wait-free* algorithm.

We consider a concurrent shared memory system in which a finite set of threads communicate asynchronously by reading and writing on shared memory words. Asynchrony implies that there is no assumption on the relative speeds of the threads. Each thread is considered to be executing a sequential program over a finite set of variables as memory words each of which consists of one or more bits. A variable is *local* to a thread if the thread holds an exclusive access to it, and is *shared* when two or more threads can access it. The sequential program consists of steps and a step can contain computation on local variables and at most a single access to a shared-variable.

Access to a shared-variable happens by means of an atomic operation. Atomic operations are also known as synchronization primitives in general. Some of the widely used synchronization primitives in concurrent data structures are listed here. The shared-variables (`Sv`) passed as arguments are passed by reference whereas the value type (`Vt`) arguments are passed by value.

1. Read

```
Vt read(Sv r){
    return r;
}
```

2. Write

```
void write(Sv r, Vt v){
    r = v;
}
```

3. Test-and-set (TAS)

```
Vt TAS(Sv r, Vt v){
    initial = r;
    r = v;
    return initial;
}
```

4. Compare-and-swap (CAS)

```
bool CAS(Sv r, Vt old, Vt new){
    if(old = r){
```

```
            r = old; return true;
        } else {
            return false;
        }
    }
```

The synchronization primitives listed above are natively provided by almost all of the multi-core and many-core architectures available in market. Using these primitives blocking and non-blocking concurrent data structures are implemented via the respective synchronization methods. Next we describe the design of these concurrent data structures.

1.1.1 Blocking concurrent data structures

The design of a blocking concurrent data structure is primarily based on mutual exclusion *locks*. A simple test-and-set (TAS) lock works as described below:

```
while(!TAS(Sv lock, Vt 1)){}
    critical_section{}
lock = 0;
```

The shared-variable `lock` is called to be *acquired* by a thread if the thread succeeds to set 1 at it given that it was initially 0. Setting the variable `lock` back to 0 is called *releasing* the lock. In the next section, we shall describe the various hardware components that play role in the performance of synchronization algorithms but here we mention that if the number of threads increases and the memory bus is locked by many threads continuously then the performance drops drastically. Therefore, a better and optimized version can be written as

```
while(lock!=0 || !TAS(Sv lock, Vt 1)){}
    critical_section{}
lock = 0;
```

In the revised format a thread reads the variable `lock` first and if it is available then only tries to acquire the lock. Using the CAS synchronization primitive also we can construct a lock as below

```
while(!CAS(Sv lock, Vt 0, Vt 1)){}
    critical_section{}
lock = 0;
```

All the above formulations of locks essentially bring busy waiting because of high degree of contention for the variable `lock`. A way to lower the con-

tention on a lock is to use some *backoff function*. A backoff function tells a process to wait for a certain amount of time before checking again. A popular backoff function is exponential backoff in which the backoff time is increased exponentially for every failed attempt to acquire the lock. But still the problem to tune the backoff function persists in the sense that some processes might have to wait much longer than other processes before acquiring the lock.

In the above lock designs it is easy to notice that the available cores in a processor are used for useless works during *busy waiting* and so it is imperative to minimize that. An alternative method of lock implementation is *queue lock* in which when a process fails to acquire a lock it adds itself to a queue associated with the lock and does a context switch so that another thread can use the processor while it is waiting to acquire the lock. After a thread finishes its critical section and releases the lock, it notifies the next thread waiting in the associated queue about it and context switches itself. Unfortunately the cost of context switching in multiprocessors could be high and therefore the queue locks get outperformed by the `TAS` or `CAS` locks.

Not just the above problems that arise with locks but also there is problem of *lock conveying* described as the situation in which a thread acquiring the lock gets preempted by the thread scheduler. This causes other threads to wait longer than necessary because the thread that got swapped could not release the lock and this results into overall slowdown of the entire program. A related problem is that of the *priority inversion* when a high priority thread has to wait for a low priority thread holding lock. This problem can be solved by increasing the priority of the lower priority thread that holds the lock to a certain high ceiling priority or to the priority of the higher priority thread. The first method is called the *priority-ceiling-protocol(PCP)* and the second method is called the *priority-inheritance-protocol(PIP)*.

And finally the most commonly known problem that occurs in blocking synchronization algorithms is that of deadlock when we *compose* two or more concurrent data structures using blocking synchronization. Although the individual concurrent data structures could be deadlock-free as described before but in the composed one two threads could keep on waiting for each other to release their respective locks while holding their own locks.

The above described problems make it important to design and implement concurrent data structures which do not use locks and that is what the non-blocking synchronization methods provide.

1.1.2 Non-blocking concurrent data structures

Unlike the blocking method of synchronization, in the non-blocking synchronization there is always a progress guarantee. The basic idea is that instead of holding any kind of lock, the threads copy the value of shared-variable using an atomic read primitive to its local variable, makes the changes as needed locally and then changes the shared-variable using a stronger synchronization primitive like `CAS` in one atomic step. In case it fails to successfully perform `CAS`, because another thread would have applied its own changes on the shared-variable since this thread read the value, it retries after updating its local value. This way a greater fault-tolerance comes with avoiding lock convoying and priority inversion.

Often an operation in a concurrent data structure needs more than a single shared-variable to be modified. As there is no assumption on the relative speeds of the threads therefore if a thread gets delayed then other threads in order to progress without getting blocked in any manner actually helps the delayed thread. This is called *helping mechanism* in a non-blocking synchronization scheme to implement a concurrent data structure. In the core of a helping mechanism is the idea that whenever in an operation more than one shared-variable is needed to be modified then the modifications should be done in an orderly manner and some indicator should be used in order to indicate the progress of the operation. However, helping should always be optimized as it makes the threads perform many atomic accesses to shared-variables even when that is not needed.

`CAS` is the most widely used synchronization primitive for modifying the shared-variables in a non-blocking synchronization method. One reason is that it is available in almost all the available multi-core and many-core architectures. However, use of `CAS` brings one of the most interesting problems in a concurrent setup known as *ABA problem*. `CAS(Sv A, Vt A, Vt C)` is not able to discover whether `A` was changed to `B` and then changed back to `A` between it was read and `CAS` is performed. In many situations it causes problems in which a concurrent data structure can become malformed which we shall see in the next chapter. Not in many architectures present now, but another synchronization primitive used for updating a shared-variable in the similar lines as `CAS` is `Load-Link/Store-Conditional`. Unlike `CAS`, it succeeds only if `A` when changed in Store-Conditional to `C` has not changed since it was read in Load-link.

1.2 Correctness and Complexity

1.2.1 Correctness

The concurrent data structures in which operations are executed by multiple threads concurrently are very hard to debug. The reason is asynchrony in the shared memory systems which makes it difficult to replicate a bug. But even before the implementation a concurrent data structure needs rigourous proof of correctness. Correctness of a concurrent data structure is expressed as consistency of operations with respect to their *sequential specifications* that the data structure provides. A concurrent data structure implementation is *considered* to be correct if the operations performed during an execution can be ordered in a way that it gives an illusion to an observer to have an implementation of a sequential data structure with same sequential specifications of the operations.

The most often used correctness condition to reason about the consistency of concurrent data structures is *linearizability* [8]. In simple terms linearizability can be viewed as a property which ensures that an operation provided by the data structure with a return as expressed in its sequential specification takes effect at a point in time between the invocation and the response point of the operation. The point of time at which the effect of the operation is seen to be taking place is called its *linearization point*. Formally, a *history* H is a sequence of operations in a concurrent data structure. If the invocation point of an operation op' is after the response point of another operation op then in the history H they must be arranged as $H = \{\ldots, op, op', \ldots\}$. The time interval between the invocation and response point of an operation is known as its *execution interval*. If the execution interval of two operations overlap then they can be arranged in any order in H and are said to be concurrent. Hence, the obtained history must satisfy the sequential specifications of the operations with respect to the return of them when applied on the concurrent data structure. Being so, the concurrent data structure is called *linearizable*.

A weaker consistency condition is *sequential consistency*. A sequentially consistent concurrent data structure guarantees that the threads executing different operations provided by the data structure will see the effect of the data structure in their respective program order. It is weaker than the linearizability in the sense that in a linearizable implementation a user looking from outside the threads gets the illusion of operations running in their program order. An even weaker consistency condition is *quiescent consistency* which ensures that to operations separated by a period of quiescence take effect in their real time order but concurrent operations i.e. those whose execution interval overlap can take any order. The sequential consistency and quiescent consistency are less

often used in proving consistency of concurrent data structures. The former is often useful in describing the correctness of low level concurrent systems such as hardware memory interfaces and the latter is used to provide even weaker constraints in object behaviours, mostly in order to obtain a higher computational performance.

1.2.2 Complexity

It is difficult to derive the time complexity in the traditional sense of an algorithm involving synchronization because of the obvious reason that there is no assumption on the relative speeds of the the threads. For concurrent algorithms we count the total number of steps taken by all the operations in an execution. It is referred as the *step complexity* of the execution. In a step, to a thread, at most one atomic access to any shared-variable is allowed. However, depending on the architecture of the machine and the memory hierarchy, access to a shared-variable that is cached in a memory close to the processor is some magnitude faster than the access to a shared-variable that is not cached. Therefore it becomes imperative to count only those steps in which an access to a remote shared-variable is performed. This is called *Remote Memory Access* measure. A remote memory access may refer to an attempt by a thread to access a shared-variable residing at either a central shared memory location or in the local memory of a core in which thread is not running. In both the cases the memory access attempt goes across the memory bus. Depending on the architecture there are two possible remote memory access complexity models

1. Coherent Caching (CC) model - An access to a shared-variable not in the cache memory of the core running the thread is called a remote access.

2. Distributed Shared Memory (DSM) model - An access to a shared-variable in the cache memory of a core that the thread is not running is called a remote access.

Often we need to derive the bounds of operations provided by a concurrent data structure. For this purpose amortized analysis is the most popular method, specifically in the contexts where the operations are not run in isolation. In concurrent data structures the amortized analysis can be used to give the bounds of complexity of operations. For a blocking concurrent data structure, it is impossible to give any upper bound of an operation which follows from the following result due to Alur and Taubenfield [2], we mention it here without its proof.

Theorem 1. *There is no two (or more) process mutual exclusion algorithm, with an upper bound on the number of times a winning process may need to access the shared memory in order to enter its critical section in presence of contention.*

Nevertheless, for a non-blocking concurrent data structure, an amortized analysis of its upper bound can be presented in terms of the size of the data structure and the *measure of contention*. In principle the measure of contention is the number of concurrent threads in the recent history of a thread during an interval. Starting with the size of the data structure at the invocation of the operation, the change in size of the data structure during the execution interval of an operation can be accounted to a measure of contention. Moreover the number of extra steps that a thread incurs on account of helping other threads during a given operation can be measured using a measure of contention and hence on amortization we could count total number of steps taken by all the threads performing the operations in a finite execution. Some of the often used measures of contention *during an operation* are as following.

Let op be an operation with t_i and t_r as its invocation and response points respectively,

- *Interval Contention* [1] - Total number of operations whose execution interval overlaps the interval $[t_i, t_r]$.
- *Point Contention* [3] - Maximum number of operations being executed concurrently at any point in the interval $[t_i, t_r]$.
- *Overlapping-Interval Contention* [9] - Maximum interval contention of any operation whose execution interval overlaps the interval $[t_i, t_r]$.

Some authors name the interval contention as *cumulative contention* and point contention as *concurrent contention* [7]. Point contention is a tighter measure of contention compared to interval contention. As explained before the number of extra steps that a thread needs to take on account of the helping mechanism in the design of a concurrent data structure should always be optimized. However, in some cases a thread may end up taking extra read steps due to extended traversal path in a linked concurrent data structure which may arise because of conservative helping. Overlapping interval contention is used in these cases in which a thread has to incur extra steps during an operation which can be accounted to the operations whose execution interval do not overlap with that of itself. For example, in a double linked-list if the insert operations do not help a concurrent insert then there can be such a situation (example taken directly from [9]). Consider that out of two links connecting two nodes in a double

linked-list, a link gets updated by an insert and the other is pending while the thread performing insert gets delayed. In the meantime many insert operations succeed to insert multiple nodes between the node whose insert is pending and the node that has been connected by one of its link. Now if after the successful insert operations return, a predecessor query travels extra steps from one direction to the other, these extra steps can not be accounted to an operation whose execution interval overlaps the predecessor query. In such cases overlapping interval contention is used.

1.3 Multi-core and Many-core Processors

At the outset we discuss the observations in the Figure 1.1. The figure presents the plotting of average throughput of `CAS` per milliseconds vs number of threads on two different multi-core architectures. We performed our experiments on - (a) An Intel machine consisting of 2 sockets populated with 6-cores of Xeon E5645 (Nehalem) CPU capable of running 12 hardware threads apiece (24 logical cores in total) at 2.4 GHz and (b) An AMD machine consisting of 4 sockets populated with 12-core Opteron 6238 (Bulldozer) CPU capable of running 12 hardware threads apiece (48 logical cores in total) at 2.6 GHz. Both the machines have DDR3 RAM at 1366 MHz. The Intel machine is provided with Quick-Path Interconnect for connectivity between chips and I/O subsystem; the AMD machine has Hyper-Transport for the same. The implementation of Simultaneous Multi-Threading [12] differs between the two machines - in Intel (Nehalem) processors two hardware threads share the resources on each physical core [11], whereas the AMD (Bulldozer) processors follow a modular architecture [4] in which each physical core runs at most one hardware thread at a time. The experiments were coded in C## and run in Mono 2.10.5 open source .Net framework and the host operating systems on both the machines were based on Linux kernel version 3.0.0. From the plot it can be observed that

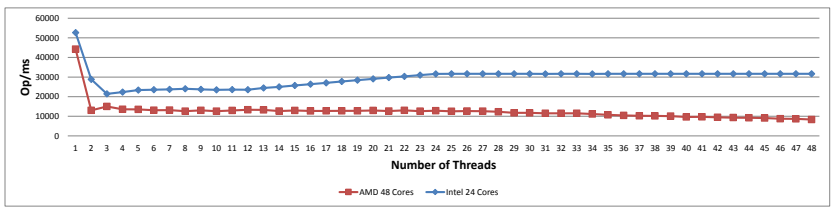


Figure 1.1: *Throughput vs. #threads of* `CAS` *operations on three architectures.*

the performance of `CAS` when implemented on different architectures shows entirely different behaviour. A comprehensive evaluation of various synchronization methods on the above mentioned architectures and implementation on most popular high level programming languages with regard to both throughput and fairness[2] is presented in [5] in the context of performance of concurrent queue and concurrent hash-table data structures. Similar observations can be seen for many-core GPUs in the figure 3.3(a). These observations give an important insight that understanding the behaviour of concurrent data structures on different implementation platforms is as important as their algorithmic design. Therefore, we discuss the hardware parameters of multi-core and many-core processors which play important roles in the scalability of concurrent data structures.

1. Multi-core CPUs

   (a) Cache hierarchy - Cache hierarchy in a given architecture has important role in the performance of a synchronization primitive. In case of local access the latency to access *Last Level Cache (LLC)* determines the throughput of all the primitives. In general, in a single socket the latency of accessing the LLC by all the cores is same. Increasing cache hierarchy improves the local load throughput and in general stores behave similarly regardless of the previous state of the cache line.

   (b) I/O subsystem - If an atomic operation is performed on a shared-variable residing at the LLC of a socket other than the one in which the thread runs in one of its core the performance drops substantially and therefore we can see in the plotting when the number of threads increases and the OS scheduler tries to schedule different threads on different sockets the performance drops. Not only that if the shared-variable is not found in the LLC of the socket then the access goes across the memory bus and that also creates drop in the performance of the synchronization algorithm. So, the I/O subsystem with improved bandwidth can improve the performance of a concurrent data structure.

[2]Fairness of a concurrent data structure is a property that measures the degree of starvation freedom as described before. Fairness is defined as

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

where $n_{i_{\Delta t}}$ is the number of successfully performed operations by the thread i, in the time interval Δt.

(c) Simultaneous Multi-Threading - The mechanism of simultaneous multi-threading (SMT) has its own role in the performance of a synchronization algorithm and hence the concurrent data structure. The SMT implementation differ in the two architectures used in the experiment shown above. The Intel (Nehalem) architecture is equipped with hyper-threading and a physical core is shared by two hardware threads and in the AMD (Bulldozer) architecture there is no such sharing. Clearly the performance difference highlights that sharing the resources of a core favours the performance of synchronization primitives and hence the performance of concurrent data structures.

2. Many-core GPU Architecture - The many-core architecture of a GPU has some more hardware parameters to look into compared to a multi-core CPU, when analysing the behaviour of a concurrent data structure implementation. The GPUs are designed to improve the processing of Single Instruction Multiple Data (SIMD) component of a program. The most popular GPUs are marketed by Nvidia and AMD and the preferred programming technology to write program for these GPUs are CUDA and OpenCL respectively. OpenCL is an open source programming technology which can be used to write programs for GPUs of all the vendors. The terms that we use here are that from CUDA, which we use for coding in our experiments. A *warp* is the minimum number of consecutive threads that should be running the same instruction in the SIMD program. In case of Nvidia GPUs 32 consecutive threads make a warp. To implement a concurrent data structure in which a synchronization primitive can not be executed by more than a single thread, the inherent SIMD model of GPU programming is the biggest hurdle. For that matter to program a concurrent data structure for a GPU, we take a representative thread from a warp to execute the synchronization primitive; and other threads in the same warp either can be used to perform the local computation if needed and suitable to be parallelized or just remain idle. In this way a concurrent data structure implementation on a GPU is scalable according to the number of warps that can be run on a core at a time. The memory hierarchy in a GPU in general has global and shared memory levels. The global memory is shared by all the warps and therefore by the threads running in the GPU at a time. The share memory is shared by threads running in the same multiprocessor. This is the reason that accessing a shared-variable in the global memory is way slower than accessing a shared-variable in the shared memory. This kind of memory hierarchy implies that a concurrent data structure should be implemented in a hierarchical way. We

shall discuss the GPU implementation of the concurrent data structures in more detail in the chapter 3.

Along with the architecture of implementation, the programming language and its concurrency support has an important role in the performance of a concurrent data structure implementation. A structured guideline to choose a programming environment to implement a concurrent data structure is presented in our work in [5].

1.4 Contributions and Future Work

The present thesis contributes to the research of concurrent data structures in two important ways -

(a) Improved algorithm of concurrent lock-free BSTs - We present a lock-free concurrent internal binary search tree which was an improvement upon the existing algorithms in the following ways. To the best of our knowledge, the amortized analysis of the operations in a concurrent search tree was first presented in our work. Our work proposed the first design of a lock-free BST in which the modify operations run in $O(H(n)+c)$, where $H(n)$ is the height of the BST with n number of nodes and c is the contention during the execution. The presented algorithm in our work also improved the disjoint access parallelism in an internal lock-free BST. The thesis presents the first experimental results of the proposed algorithm which was published in [6].

(b) Concurrent Data Structures on GPUs - We present the first evaluation of any concurrent data structure on GPUs. Our work examines the performance portability of concurrent data structures with regard to the move from conventional CPUs to graphics processors and found that they are in general performance portable. This work presented important insight on the programming paradigm of GPUs that present inherent limitations on the implementation of synchronization algorithms on GPUs. Our work also discussed the hardware parameter of a GPU that play role in the performance of concurrent data structure implementations.

Continuing the present work we plan to study the design and analysis of other important concurrent data structures algorithmically as well as from implementation point of view. The present work presented amortized analysis of upper bound of set operations in a lock-free BST, whereas the area of lower bound analysis of any lock-free data structure is an important open area. We plan to study the lower bound of important algorithms that use the concurrent data

structures and hence can be implemented and made scalable using lock-free concurrent data structures. We also plan to implement more complex concurrent data structures on the GPUs and other heterogenous architectures.

Bibliography

[1] Y. Afek, G. Stupp, and D. Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.

[2] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Real-Time Systems Symposium, 1992*, pages 12–21. IEEE, 1992.

[3] H. Attiya and A. Fouren. Algorithms adapting to point contention. *Journal of the ACM*, 50(4):444–468, 2003.

[4] 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.

[5] 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 27th IPDPS*, pages 1309–1320, 2013.

[6] B. Chatterjee, N. Nguyen, and P. Tsigas. Efficient lock-free binary search trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 322–331, New York, NY, USA, 2014. ACM.

[7] M. Herlihy, V. Luchangco, and M. Moir. Space- and time-adaptive nonblocking algorithms. *Electronic Notes in Theoretical Computer Science*, 78(0):260 – 280, 2003. CATS'03, Computing: the Australasian Theory Symposium.

[8] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[9] R. Oshman and N. Shavit. The skiptrie: Low-depth concurrent search without rebalancing. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 23–32, New York, NY, USA, 2013. ACM.

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

[11] 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.

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

Part II

PAPERS

PAPER I

Extended version of

Bapi Chatterjee, Nhan Nguyen and Philippas Tsigas

Efficient Lock-Free Binary Search Trees

In the Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC 2014), Technical report no 2014:05, ISSN 1652-926X, Department of Computer Science and Engineering, Chalmers University of Technology, Sweden, Feb 2014.

2
PAPER I

Abstract
In this paper we present a novel algorithm for concurrent lock-free internal binary search trees (BST) and implement a Set abstract data type (ADT) based on that. We show that in the presented lock-free BST algorithm the amortized step complexity of each set operation - ADD, REMOVE and CONTAINS - is $O(H(n) + c)$, where $H(n)$ is the height of the BST with n number of nodes and c is the contention during the execution. It uses single-word compare-and-swap (CAS) operations. We show that our algorithm has improved disjoint-access-parallelism compared to similar existing algorithms. We prove that the presented algorithm is linearizable. To the best of our knowledge, this is the first algorithm for a concurrent internal binary search tree data-structure in which the modify operations are performed with an additive term of contention measure. The experimental results show that the presented algorithm is scalable and outperforms existing algorithms for similar data structures.

2.1 Introduction

With the wide and ever-growing availability of multi-core processors it is evermore compelling to design and develop more efficient concurrent data-structures.

Being immune to deadlocks due to various fault-causing factors beyond the control of the data-structure designers, the non-blocking concurrent data-structures are more attractive than their blocking counterparts.

In literature, there are lock-free as well as wait-free singly linked-lists [12, 25], lock-free doubly linked-list [24], lock-free hash-tables [18] and lock-free skip-lists [12, 23]. However, not many performance-efficient non-blocking concurrent search trees are available. A multi-word compare-and-swap (MCAS) based lock-free BST implementation was presented by Fraser in [13]. However, MCAS is not a native atomic primitive provided by available multi-core chips and is very costly to be implemented using single-word CAS. Bronson et al. proposed an optimistic lock-based partially-external BST with relaxed balance [3]. Ellen et al. presented lock-free external binary search tree [11] based on co-operative helping technique presented by Barnes [2]. Though their work did not include an analysis of complexity or any empirical evaluation of the algorithm, the contention window of update operations in the data-structure is large. Also, because it is an external binary search tree, REMOVE is simpler at the cost of extra memory to maintain internal nodes without data. Howley et al. presented a lock-free internal BST [16] based on similar technique. A software transactional memory based approach was presented by Crain et al. [7] to design a concurrent red-black tree. While it seems to outperform some coarse-grained locking methods, it easily falls behind in performance when compared to a carefully tailored locking scheme as in [3]. Recently, two lock-free external BSTs [4, 20] and a lock-based internal BST [9] have been proposed. All of these works lack theoretical complexity analysis.

A common predicament for the existing lock-free BST algorithms is that if multiple modify operations contend at a leaf node, and, if a REMOVE operation among them succeeds then all other operations have to restart from the root. It results in the complexity of a modify operation to be $O(cH(n))$ where $H(n)$ is the height of the BST on n nodes and c is the measure of contention. It may grow dramatically with the growth in the size of the tree and the contention. In addition to that, CONTAINS operations have to be aware of an ongoing REMOVE of a node with two children, otherwise it may return an invalid result. Hence in the existing implementations of lock-free internal BST [16], a CONTAINS operation may have to restart from the root on realizing that the return may be invalid if more nodes are not scanned. The external or partially-external BSTs remain immune to this problem at the cost of extra memory used by the routing internal nodes. Our algorithm solves both these problems elegantly. The CONTAINS operations in our BST enjoy oblivion of any kind of modify operation. And, the modify operations after helping a concurrent modify operation restart not from the root, rather from a level in the vicinity of failure. It ensures that

all the operations in our algorithm run in $O(H(n) + c)$. This is our main contribution. Ellen et al. [10] improved the external BST by Ellen et al. [11] using thread local stacks for each thread to achieve similar complexity as ours.

Given the requirements of a concurrent data-structure in terms of number of shared-memory words to modify in order to update, we always strive to exploit maximum possible disjoint-access-parallelism [17], so that if an operation needs to modify multiple shared memory words in order to complete its steps then the progress of concurrent operations could be maximized. The lock-free methods for BST [11, 16], in order to use single-word CAS for atomically modifying the links outgoing from a node, and yet maintain correctness, store a flag as an operation field or some version indicator in the node itself, and hence a modify operation "holds" a node. This mechanism of holding a node, specifically for a REMOVE, can reduce the progress of two concurrent operations which may be otherwise non-conflicting. In [20], a flag is stored in a link instead of a node in an external BST. We found that even in an internal BST it is indeed possible that a REMOVE operation, instead of holding the node, just holds the links connected to and from a node in a predetermined order so that improved progress of concurrent operations working at disjoint memory words corresponding to the links could be achieved. The presented lock-free design using "storing a flag" at a link instead of a node significantly improves the disjoint-access-parallelism compared to the existing design as in [16]. This is our next contribution.

Helping mechanism which ensures non-blocking progress may prove counterproductive to the performance if not used judiciously. In the conference version of this paper [6] we proposed an adaptive helping mechanism to choose depending on the read-write load. However, with empirical observations we found that helping during traversal always reduces the performance and so it is better not to help a pending REMOVE operation during traversal. Our algorithm requires only single-word atomic CAS primitives along with single-word atomic read and write which practically exist in all the widely available multicore processors in the market. Based on our design, we implement a Set ADT. We prove that our algorithm is linearizable [15]. We also present complexity analysis of our implementation which is novel in in the context of concurrent BST algorithms. This is another contribution in this paper.

The body of our paper will further consist of the following sections. In section 2.2, we present the basic tree terminologies. In section 2.3, the proposed algorithm is described. The section 2.4 presents a discussion on the correctness and the progress of our concurrent implementation along with an amortized analysis of its time complexity. The section 2.5 presents the detail of the implementation of the algorithm and a discussion on the ABA problem. The paper is

concluded in section 3.6.

2.2 Preliminaries

A *binary tree* is an ordered tree in which each *node* x has a *left-child* and a *right-child* denoted as $left(x)$ and $right(x)$ respectively, either or both of which may be *external*. When both the children are external the node is called a *leaf*, with one external child a *unary* node and with no external child a *binary* node, and all the non-external nodes are called *internal* nodes. If a node is used to store data it is called a *data node* else it remains in the tree for just facilitating a traversal and is called a *routing-node*. We denote the parent of a node x by $p(x)$ and there is a unique node called *root* s.t. $p(root) = null$. Each parent is connected with its children by links.

We are primarily interested in implementing an ordered *Set* ADT - *binary search tree* using a binary tree in which each node is associated with a unique key k selected from a totally ordered universe. A node with a key k is denoted as $x(k)$ and x if the context is otherwise understood. Determined by the total order of the keys, each node x has a *predecessor* and a *successor*, denoted as $pre(x)$ and $suc(x)$, respectively. We denote height of x by $ht(x)$, which is defined as the distance of the deepest leaf in the *subtree* rooted at x from x. Height of a BST is $ht(root)$.

In an *internal BST*, all the internal nodes are data-nodes and the external nodes are usually denoted by null. There is a *symmetric order* of arranging the data - all the nodes in the *left subtree* of $x(k)$ have keys less than k and those in its *right subtree* have keys greater than k, and so no two nodes can have the same key. To query whether a BST CONTAINS a data with key k, at every *search-step* we utilize this order to look for the desired node either in the left or in the right subtree of the *current node* if the key not matched at it, unless we reach an external node. If the key matches at a node then we return true (or address of the node if needed), otherwise false. To ADD data we query by its key k. If the query reaches an external node we replace this node with a new leaf node $x(k)$. To REMOVE a data-node corresponding to key k we check whether $x(k)$ is in the BST. If the BST does not contain $x(k)$, false is returned. On finding $x(k)$, we perform delete as following. If it is a leaf then we just replace it with an external node. In case of a unary node its only child is connected to its parent. For a binary node x, it is first replaced with $pre(x)$ or $suc(x)$ which may happen to be a leaf or a unary node, and then the replacer is REMOVEd.

In an alternate form - an *external BST*, all the external nodes are data-nodes and the internal nodes are routing-nodes. The symmetric order is different from

an internal BST - all the nodes in the left subtree of a node $x(k)$ have keys less than k whereas, in its right subtree the nodes have keys at least k. It increases the depth of the BST compared to an internal BST for an equal amount of data. Also, because all the data-nodes are external, a query traversal always terminates at an external node making the average traversal length increase for a set of queries comprising large number of search operations. However, because the data-nodes are external, REMOVE of a node becomes simpler compared to an internal BST (in fact, REMOVE of a node here is effectively equivalent to REMOVE of a unary node as in an internal BST). In this paper we focus on internal BSTs, and hence forward by a BST we shall mean an internal BST.

2.3 Our Algorithm

2.3.1 The Efficient Lock-free BST Design

To implement a lock-free BST, we represent it in a threaded format [22]. In this format, if the left or right -child link at a node x points to null and hence points to an external node, it is instead connected to $pre(x)$ or $suc(x)$ respectively. An indicator is stored to indicate whether a child-link is used for such a connection. This is called *threading* of the child-links. In our design, we use the child-links at the leaf and unary nodes as following: right-child link, if pointing to null, is threaded and is used to point to the successor node, whereas a similar left-child link is threaded and pointed to the node itself, see Fig. 2.1(a). In this representation a BST can be viewed as an ordered list with exactly two outgoing and two incoming links per node, as shown in Fig. 2.1(b). Also among the two incoming links, exactly one is threaded and the other is not. In this representation, if $x(k_i)$ and $x(k_j)$ are two consecutive nodes in the corresponding ordered list and hence in the BST and there is no node $x(k)$ such that $k_i \leq k \leq k_j$ then we call the interval $[k_i,\ k_j]$ to be *associated* with the threaded link incoming at k_j. The unique incoming threaded-link on a node is called its *order-link*. We name the node where the order-link of x emanates from - the *order-node* of x.

We utilize this symmetry of the equal number of incoming and outgoing links. A usual traversal in the BST following its symmetric order for a predecessor query, is equivalent to a traversal over a *sublist* of the ordered-list shown in Fig. 2.1(b). This list is essentially produced by an in-order traversal of the BST. This is made possible by the threaded right-links at leaf or unary nodes. Though in this representation, there are two links in both incoming and outgoing directions at each node, a single link needs to be modified to ADD a node in the list. To REMOVE a node we may have to modify up to four links. Therefore, ADD can be as simple as that in a lock-free single linked-list [12], and REMOVE

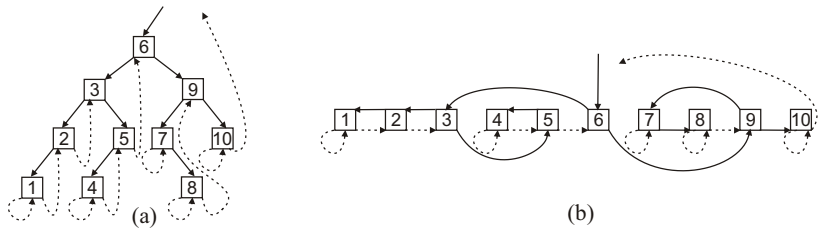


Figure 2.1: *(a) Threaded BST (b) Equivalent ordered list.*

should be no more complex than that in a lock-free double linked-list [24]. A traversal in a lock-free linked-list is not affected by a concurrent REMOVE of a node. In our design of internal BST also, a traversal can remain undeterred by any ongoing modification, unlike that in the existing lock-free implementations of internal BSTs [13, 16].

In designing an internal BST in a concurrent setup, specifically a lock-free one, the most difficult part is to perform an error-free REMOVE of a binary node. To remove a binary node we replace it with its predecessor, which is its order-node, and hence the incoming and outgoing links of the predecessor also need to be updated in addition to the links incoming to the node itself. According to the number of links needed to be modified in order to remove a node, unlike traditional categorization of nodes of a BST into leaf, unary and binary, we categorize them into three categories as shown in Fig. 2.2(b). The categorization characteristic is the position of the order-node of a node. A node belonging to category 1 is that which is order-node of itself; for a category 2 node, the order-node is its left-child; and for a category 3 node its order-node is the rightmost node in its left-subtree.

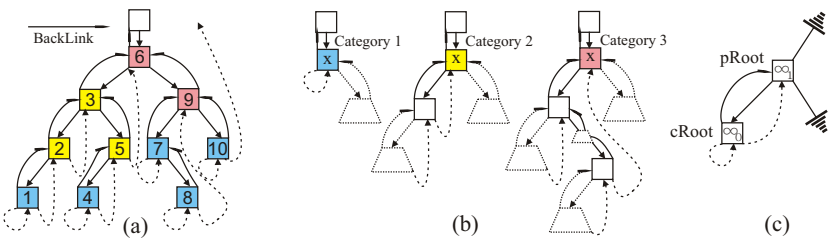


Figure 2.2: *(a) Threaded BST with backlinks (b) Categorization of nodes for* REMOVE *(c) An empty tree.*

To remove a node of category 1, only the incoming parent-link needs to be modified to connect to the node pointed by the right-link. For a category 2 node, the parent-link is updated to connect to the node pointed by the left-link and the order-link is modified to point to the node which the right-link was pointing to. In order to remove a category 3 node, its order-node replaces it and the incoming and outgoing links of the order-node are updated to take the values of that of the REMOVEd node. Parent-link of the order-node is connected to the node which its left-link was pointing to before it got shifted. Observing carefully, the removal of a node is essentially replacing it with its order-node if the order-node is not same as itself, and in order to do that, the corresponding links of the order-node and the parent are updated. If the order-node of the node is same as itself then only the incoming link from the parent is needed to be updated. Also, when a link is updated the thread indicator value of the link, which it updates to, is copied to it. Note that in this categorization a left-unary (i.e. whose right-child is null) or a binary node whose left-child is either a left-unary or a leaf node gets classified in to category 2. Category 1 includes conventional leaf and right-unary nodes (whose left-child is null).

The number of steps required to remove a category 1 or a category 2 node is much less compared to that required in case of a category 3 node because of the greater number of links that need to be modified. Obviously the REMOVE of a category 3 node is much more complex than that of the other two categories of nodes. Also, it can be observed that if the order-node of a category 3 node is shifted up in a way that it is connected in between the left-child of the node and the node, the category 3 node is transformed to a category 2 node with its order-link emanating from its left-child, without violating the BST order. We will use this observation while removing a category 3 node which we will do in two stages - first shift the order-node to make it the left-child of the node under REMOVE and then follow the steps required for the REMOVE of a category 2 node.

In all the existing designs of a lock-free BST, when an operation fails at a link connected to a leaf node because of a concurrent modify operation, it retries from the scratch i.e. it restarts the operation from the root of the tree, after helping the obstructing concurrent operation. In our design an operation restarts from a node at the vicinity of the link where it fails, after the required helping. To achieve that, we need to get hold of the appropriate node(s) to restart at. For that, we use a *backlink* per node similar to [12], that points to a node *present* in the tree from where the failure spot is a single link away. It should be noted that a backlink is not used for the tree traversal, see Fig. 2.2(a). In a category 3 node under REMOVE, we also store the address of its order-node, so that a concurrent operation failing at one of its child-links does not have to

traverse down the tree to search the order-node to replace, while helping after being obstructed. For that we use a *prelink* per node which ensures a single link traversal for recovery due to obstruction by a concurrent REMOVE of a category 3 node.

With the description of the steps to remove nodes of different categories as above, we describe the synchronization of threads performing modifications of links associated with a node undergoing REMOVE. A REMOVE operation first sets indicators using atomic CAS at the links associated with the node (and its order-node) and then updates the incoming links to it in order to correctly synchronize with a concurrent operation. During the steps of setting indicators, if any atomic CAS fails because of that of a concurrent modify operation then it goes to help the obstructing operation before retrying to ensure the lock-freedom in the BST implementation. Also it is likely that when an operation is helping a concurrent operation, it may itself be obstructing another concurrent operation and hence can get helped. Because of this mechanism of interleaved helping and multiple CAS operations needed to complete the REMOVE of a node, it is worthwhile to keep note of the state of the node when REMOVE has been affected so that some unnecessary CAS executions could be saved. To do that we can set an indicator on one of the outgoing links of a node whose parent link gets modified to point to the node that replaces the removed node.

In our design we need to maintain as many states of a link as it could be sufficient to indicate various stages of link updates during a REMOVE. Given that, we need following states of a link: (a) a clean unthreaded link (b) a clean threaded link (c) a threaded / unthreaded link with indicator implying it as outgoing from a node under REMOVE (d) a threaded link with indicator implying it as order-link of a node under REMOVE (e) an unthreaded link with indicator implying it as a parent-link of a node under REMOVE (f) an unthreaded link with indicator implying it as a parent-link of the order node of a node under REMOVE (g) a threaded / unthreaded link with indicator implying it as a left-link of the order-node of a category 3 node under REMOVE. We observed that *eight* different state-indicators of a link are sufficient for maintaining all the aforementioned states effectively. Apart from the above states we also maintain one more state of a threaded / unthreaded right-link of a node that has been removed to save some helping steps of an obstructed operation. Hence we need *ten* state-indicators for the links in our design.

We use the terms - (a) *mark*, (b) *flag* and (c) *tag* to denote the state change of a link by setting indicators on it using an atomic CAS. They respectively denote that a link from a clean unthreaded or threaded state modifies to become a link (a) outgoing from a node under REMOVE (b) incoming as parent-link or order-link of a node under REMOVE and (c) incoming parent-link or outgoing

left-link of the order-node of a category 3 node under REMOVE. The order-link of a category 1 node is though outgoing from the node itself but it is flagged. And we call the state change of a marked right-link as *finalize* to indicate the completion of a REMOVE. Once a link is marked, flagged or tagged it can not be a *point of injection* of a *new* ADD or REMOVE operation. Also no state change over such a state is permitted except that a marked right-link changes to finalized only after the all the incoming links to the removed node are updated. The flagging and marking are performed in such a predetermined order so that a malformed structure of the BST is avoided. Following are the order of state changes of links associated with a node undergoing REMOVE.

(a) Category 1: (I) flag the order-link of the node (II) mark the right-link of the node (III) flag the incoming parent-link of the node (IV) update the parent link to connect to the right-child (V) finalize the marked right-link.

(b) Category 2: (I) flag the order-link of the node (II) mark the right-link of the node (III) mark the left-link of the node (IV) flag the incoming parent-link of the node (V) update the order link to connect to the right-child (VI) update the parent link to connect to the left-child (VII) finalize the marked right-link.

(c) Category 3: As explained before, a category 3 node can be transformed to a category 2 node without violating the symmetric order of the BST by shifting its order-node to connect in between the node and its left-child. Therefore, to remove a category 3 node, we shall proceed in two stages. Following are the steps in the first stage - (I) flag the order-link of the node (II) tag the incoming parent-link of the order-node (III) tag the left-link of the order-node (IV) set the pre-link of the node to point to the order-node (V) mark the right-link of the node (VI) mark the left-link of the node (VII) update the parent-link of the order node to connect it to the left-child of the order-node (VIII) update the left-link of the order node to connect it to the left-child of the node under REMOVE (IX) update the left-link of the node to connect it to the order-node and keep it marked. After the first stage is over, the category 3 node under REMOVE gets converted to a category 2 node under REMOVE with the first three steps of the state change applied to it. In the second stage the remaining steps as described for a category 2 node are completed.

In the very first step of flagging the order-link of a node to remove, if the atomic CAS succeeds then eventually that node is removed from the BST. Whenever a parent-link or an order-link is updated to connect to the right-child of a node that is removed, the thread indicator status of the right-link is also

copied to the updated link. Also, before a flagged or tagged link is updated to connect to a new node, the backLink of the new node is updated to connect to the node where the link emanates from, if the corresponding marked or tagged link was not threaded. See Fig. 2.3. Please note that, there can be concurrent REMOVE of the order-node of a category 3 node undergoing REMOVE itself. In that case the order-link of the category 3 node is the right-link of the order-node and the order-link of the order-node is the left-link of the order-node of the category 3 node. Therefore, if the REMOVE operations at both the nodes have successfully flagged the order-link of the respective nodes, they are synchronized as following: (a) the REMOVE of the order-node, say o_1, is put to help the concurrent REMOVE of the node, say o_2 (b) o_1 or o_2 having tagged the parent-link of the order-node and finding the left-link of the order-node as threaded and flagged, updates the parent-link of the order node to become the new flagged order-link of the category 3 node (c) it then finalizes the flagged right-link of the order-node and thus o_2 would terminate before retrying (d) o_1 reassesses the node category of the node that it was removing before attempting the step (II) of REMOVE now. In all the other cases of helping, on failing at an atomic `CAS` an operation retries the same `CAS` at the same step.

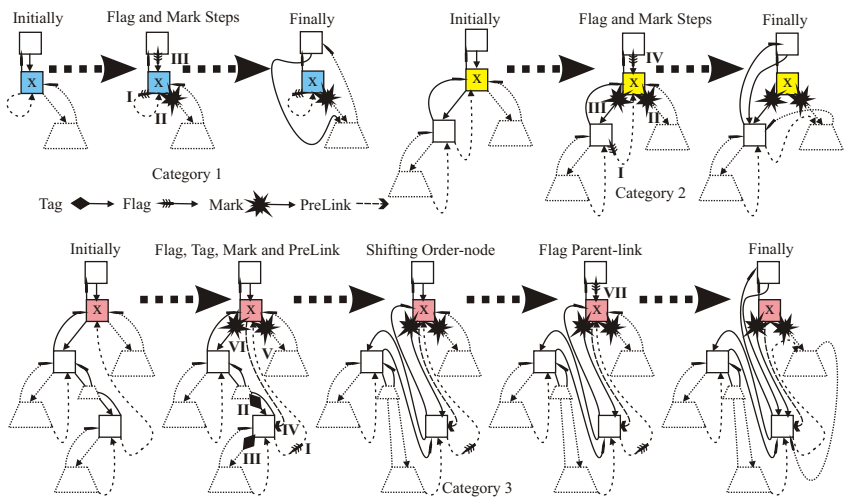


Figure 2.3: *Remove steps of nodes*

Because we follow orderly modifications of the links, it never allows a node to be missed by a traversal in the BST unless both its incoming links are pointed

away. However, because a node may shift "upward" to replace its successor, the interval associated with the order-link of its successor may shift "rightward" i.e. to the right subtree of the node after the successor is REMOVEd. Given that, the intuitive terminal criterion at a threaded link of a traversal may not work. Therefore, in order to terminate a traversal correctly, we use the stopping criterion given in Condition 1. It follows from the fact that the threaded left-link of a node is connected to the node itself and a traversal in the BST uses the order in the equivalent list.

CONDITION **1.** *Let k be the search key and k_{curr} be the key of the current node in the search path. If $(k = k_{curr})$ then stop. Else, if the next link is a threaded left-link then stop. Else, if the next link is a threaded right-link and the next node has key k_{next} then check if $k < k_{next}$. If true then stop, else continue.*

This stopping criterion not only solves the problem of synchronization between a concurrent REMOVE and a traversal for a predecessor query but also enables to achieve bound on the length of the traversal path. We shall explain that in section 2.4. A similar stopping criterion is used in [9] for conventional doubly-threaded BSTs.

We can observe that in our BST design, because we do not store a flag or version indicator at a node, two modify operations that need to change two disjoint memory-words have significantly improved conditions for progress. First, an ADD operation does not store any flag so two ADD operations operating at two outgoing threaded links of a leaf node can progress without any synchronization, which is not the case if a flag is stored at a node by an ADD. Then, to REMOVE a category 2 node the left-link of its predecessor is never marked or flagged, therefore when such a node goes under REMOVE, a concurrent injection of ADD or REMOVE at the left-link of the predecessor is possible. These progress conditions are not possible in the existing algorithms that use "node holding" [11, 16]. It shows that our algorithm has improved disjoint-access-parallelism.

2.3.2 The Algorithm

We consider our concurrent system as a shared memory machine in which threads are fully asynchronous and arbitrary delay of a thread is allowed. The read and write of a single memory-word is guaranteed to be atomic. The system provides atomic single-word compare-and-swap (CAS) primitives. An execution of $\texttt{CAS}(R,\ old,\ new)$ returns true, iff $(old = R)$, after updating R to new, else it returns false without any update. The language specific implementation

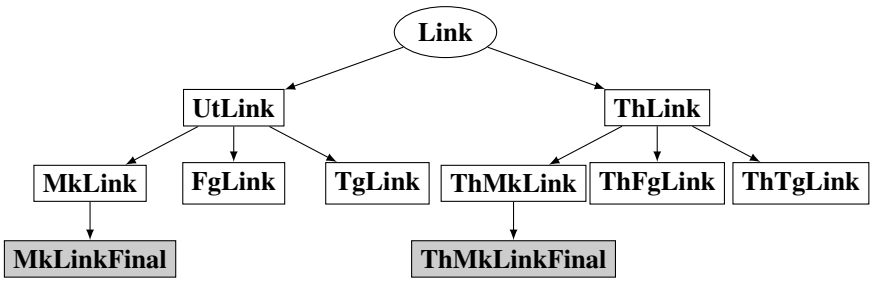


Figure 2.4: *The hierarchy of the class*

of the algorithm will be described in the section 2.5 and here we shall describe the algorithm borrowing terms from a Java implementation.

```
1  class Node {
2      KType key;
3      Link lChild, rChild;
4      Node backLink, preLink;
5  };
6  class Link {
7      KType key;
8      Node ref;
9  };
   /* Global variable with the fixed value of members.  */
10 Node pRoot = Node (∞1, UtLink(∞0, cRoot), null, null, null);
11 Node cRoot = Node (∞0, ThLink(∞0, cRoot), ThLink(∞1, pRoot), pRoot, null);
```

A typical node $x(k)$ in our BST implementation is represented by an instance of the class **Node** consisting of five fields corresponding to - (a) a key key, (b) two child-links : $lChild := left(x)$ and $rChild := right(x)$, (c) a $backLink$ and (d) a $preLink$. See lines 1 to 5. The child-links are instances of the class **Link** which has two fields - (a) the key of the node that it points to as key and (b) the reference of the node that it points to. We could have done with only the reference field in a **Link**, however keeping the key gives an optimization to read the key of the nodes during traversal. The **Link** class has subclasses representing states of a link as described before. The hierarchy of the subclasses of **Link** is shown in the figure 2.4. See lines 6 to 9. We shall use the terms **instanceof** and **typeof** to mean a variable that is an instance of a class or any of its subclasses and a variable that is an instance of a class only (not its subclasses), respectively. Being applied on by the functions $\cdot mark()$, $\cdot flag()$ and $\cdot tag()$ a variable of type of **UtLink** with fields $key = k$ and $ref = r$ gives a new vari-

able of type of **MkLink**, **FgLink** and **TgLink** with fields $key = k$ and $ref = r$; and, similarly for a variable of type of **ThLink**. The function $\cdot finalize()$ when applied on a variable of type of **MkLink** or **ThMkLink** gives a new variable of type of **MkLinkFinal** and **ThMkLinkFinal** respectively. Also, for a variable of type **ThFgLink**, the function $\cdot finalize()$ generates a new **ThMkLinkFinal** with the same key and ref fields. A variable of type of **MkLink** and **ThMkLink** when applied on by the function $\cdot unmark()$ gives a new variable of type **UtLink** and **ThLink** respectively with the same fields of key and ref. And, a variable of type of **TgLink** and **ThTgLink** when applied on by the function $\cdot untag()$ gives a new variable of type **UtLink** and **ThLink** respectively. A link that is **instanceof ThLink**, when applied on by the function $\cdot isThd()$ returns true and that which is **instanceof UtLink** return false for the same.

We use two global variables $pRoot = x(\infty_1)$ and $cRoot = x(\infty_0)$. The keys ∞_1 and ∞_0 satisfy the relation $k<\infty_0<\infty_1$ for all the nodes $x(k)$ present ever in the BST. Effectively, $cRoot$ is always left-child and order node of $pRoot$ and all the nodes ever present in the BST are always in the left-subtree of $cRoot$. Also, the node with the largest key in the BST is the order-node of $cRoot$. See lines 10 and 11. Having mentioned the terminologies used, we next describe the pseudo-code of the functions in our algorithm.

Locating a Node

The set operations - CONTAINS, ADD and REMOVE, need to perform a predecessor query using a given key k to locate either the node $x(k)$ or a threaded order-link incoming at the node $x(k_j)$ such that an interval $[k_i,\ k_j]$ associates with it and $\{k_i \leq k \leq k_j\}$. $x(k_i)$ is either the order-node of $x(k_j)$ or a node in whose right-subtree $x(k_j)$ is the leftmost node. The function LOCATE is used for that, which starts from a node preNode and follows the symmetric order of the internal BST. The return value of LOCATE is an instance of the class **Location**, say l. The class **Location** comprises of two fields pre and cur as address of nodes. The variable $l{\cdot}cur = x(k_{curr})$ satisfies the above requirements in the following way - if the key k is equal to k_{curr} then the desired node is located, if k is less than k_{curr} then the desired link is the left threaded link at $x(k_{curr})$ and for the greater than case it is the right threaded link at the same. The node referred by the variable $l{\cdot}pre$ is connected to the node $x(k_{curr})$ by a single link. The termination criterion for LOCATE implements Condition 1. Lines 22 to 44.

It is important to note here that an operation during the traversal does not help a concurrent modify operation in the BST. However, the respective positions of the nodes can change during traversal causing it to move in a loop. To take care of that we have to do extra checks and adjustments in lines 37 to 44.

```
12 class Location {
13     Node pre, cur;
14 };
```

```
15 Location LOCATE(Node preNode, KType key)
16 begin
17     bool turn = key ≤ preNode·key, lastTurn;/* Convention of turn:  true =
       left, false = right.  */
18     Node prePrev; Link rtPrePrev, rtprePrevNow;
19     Node curr = preNode·(turn ? lChild : rChild)·ref;
20     while true do
21         if (key = curr·key) then
22             return Location (preNode, curr);
23         else
24             lastTurn = turn;
25             prePrev = preNode;
26             rtprePrev = prePrev·rChild;
27             if (key > curr·key) then
28                 next = curr·rChild;
29                 if ((next instanceof ThLink ) and (key < next·ref·key)) then
30                     return Location (preNode, curr);
31             else
32                 next = curr·lChild;
33                 if (next instanceof ThLink ) then
34                     return Location (preNode, curr);
35             preNode = curr;
36             curr = next;
37             if ((lastTurn = false ) and (turn = true )) then
38                 rtPrePrevNow = prePrev·rChild;
39                 if (rtPrePrevNow·ref ≠ preNode ) then
40                     curr = rtPrePrevNow·rchild·ref;
41                     preNode = prePrev;
42                 else if (rtPrePrevNow·isThd() ≠ rtprePrev·isThd() ) then
43                     curr = prePrev;
44                     preNode = curr·backLink;
```

```
45 bool CONTAINS(KType key)
46 begin
47     Location loc = LOCATE(pRoot, key); Node curr = loc·cur;
48     return (curr·key = key ? true : false);
```

Here we first check whether the present move of the traversal is from right to left. If it is so then we need to check whether the last node the we had read has been shifted to replace its successor by a concurrent REMOVE operation. That is indicated by reading the right-child link of the node that we leave behind. If either the right-child node or the thread status of the right-child link changed since we moved from there then it may be possible that the present node has been either shifted or deleted. In both cases if we backtrack by a single link then we do not have to visit the present node again as we meet a threaded right-link whose key will be greater than the query key and so the LOCATE may terminate.

To perform a CONTAINS operation for a key key, we start from the node $pRoot$. Having performed LOCATE, if the node at the returned address has the key equal to the query key key then it indicates that the key was present in the BST at the point of termination of LOCATE. And therefore, CONTAINS returns true. If the query key does not match with that of the node whose address is returned then false is returned by CONTAINS. See lines 45 to 48.

Remove operation

To REMOVE a node $x(k)$ present in the BST, we locate its order-link. In order to do that, starting from the node $pRoot$ we locate the threaded link which the interval containing the key $(k-\epsilon)$ associates with, see line 53. Here the function LOCATE returns the possible node to remove $x(k)$ and its parent as $loc{\cdot}cur$ and $loc{\cdot}pre$ respectively. Having located the threaded link we need to ascertain that the node $x(k)$ is present in the BST. If $x(k)$ is present in the BST then the located threaded link must be pointing to it and if that is not found then false is returned at line 61. If $x(k)$ is located then we try to flag its order-link which is the located threaded link. In the **while** loop, in lines 65 to 90, during the repeated attempt to flag the order-link, we check if the right-link of the node is already marked before attempting the CAS to flag the order-link. It gives a good optimization as the contention grows. If the right-link is found marked then we directly go to finish the remaining steps of REMOVE and false is returned, line 66. This also takes care of the scenario in which if the right-link is finalized then the REMOVE returns readily; this is done in the function HELPMARKRIGHT.

If the CAS execution fails then it may have been that either the link is flagged, marked or tagged, or a new node is added at the threaded link. If the link is flagged then it indicates that a concurrent thread started the REMOVE of the same node and so that is helped. In case of marked order-link, it shows that the order-node is being removed and so that is helped first. Please note that a marked and threaded link can only be the right-link of a node because

```
49 bool REMOVE(KType key)
50 begin
51     bool turn = false;
52     Node preNode = pRoot, delNode;
53     Location loc = LOCATE(preNode, key − ϵ);
54     if (key = loc·cur·key) then
55         preNode = delNode = loc·cur;
56         turn = true;
57     else if (key = (l·cur·rChild)·key) then
58         preNode = loc·pre;
59         delNode = loc·cur;
60     else
61         return false;
62     Link delRtLink, oLink;
63     while (true ) do
64         delRtLink = delNode·rChild;
65         if ((delRtLink instanceof MkLink) or (delRtLink instanceof ThMkLink)) then
66             HELPMARKRIGHT(delNode); return false;
67         oLink = preNode·(turn ? lChild : rChild);
68         if ((oLink typeof ThLink ) and (oLink·ref = delNode)) then
69             if CAS(preNode·(turn ? lChild : rChild), oLink, oLink·flag() ) then
70                 HELPTHFLAGGED(preNode, delNode);
71                 return true;
72         if (oLink·ref = delNode) then
73             if (oLink typeof ThFgLink ) then
74                 HELPTHFLAGGED(preNode, delNode);
75                 return false;
76             else if (oLink typeof ThMkLink ) then
77                 preNode = HELPMARKRIGHT(preNode); continue;
78             else if (oLink typeof ThTgLink ) then
79                 Node delRightNode = delRtLink·ref;
80                 if ((delRtLink instanceof ThFgLink ) and (delRightNode·preLink =
                    delNode)) then
81                     HELPTAGPRELEFT(delRightNode);
82                 preNode = loc·pre; continue;
83         loc = LOCATE(preNode, key);
84         if (l·cur = delNode) then
85             if (delNode·lChild·ref = delNode) then
86                 preNode = delNode; turn = true;
87             else
88                 preNode = loc·pre; turn = false;
89         else
90             return false;
```

a threaded left-link can only be the order-link of a category 1 node which is flagged in order to remove the node or tagged if the successor node of the node is being removed. If the order-link is found marked then the node to restart is returned by the function HELPMARKRIGHT. If the order-link is found tagged then after helping steps, the node $x(k)$ shifts to replace its successor and then node located as its parent, which was $loc{\cdot}pre$, becomes its order-node and its threaded right-link becomes the desired order-link. Therefore we restart from the node $loc{\cdot}pre$. In case a new node is added at the order-link then the order-link itself get shifted down in the BST and therefore we call LOCATE again. It can also happen that the node $x(k)$ is removed from the BST concurrently and it is not located and then false is returned. See lines 50 to 90.

Having flagged the order-link, it becomes ensured that the node $x(k)$ will be eventually removed and we move to the next step of marking right-link in case of category 1 and category 2 nodes and tagging the parent-link of the order node in case of category 3 nodes. This is done in the function HELPTHFLAGGED, see lines 92 to 134. Here also during the repeated attempts to perform the CAS to mark or tag the desired link, the right-link of the node $x(k)$ is checked, and if it is found marked then only the remaining steps are performed or the REMOVE returns in case the right-link is found finalized. If before a reattempt the order-link is found not connected to the node $x(k)$ then it indicates that a concurrent helping operation may already have taken steps up to the update of the order-link and only the final step of the update of the parent-link may have remained, this is taken care of in the lines 129 to 134. Also note that, a category 3 node may get changed to a lower category node before reattempt after the failure to execute the desired CAS at the parent-link of the order-node. These scenarios are taken care of in checking the category of the node before every reattempt at the line 102. Here, effectively it is checked if either the order-node or the parent of the order-node is same as the node to remove. In the former case it is a category 1 node and in the latter case it is a category 2 node. In both these cases the second step is marking the right-link of the node to remove. If both these conditions are not met then it may possibly be a category 3 node and therefore the tagging of the parent-link of its order-node is attempted. During tagging of the parent-link of the order-node, if there is an obstruction then it is appropriately helped. However, if the parent-link of the order-node is found connected to the node under REMOVE and in the state threaded and flagged then it shows that the order-node has been removed concurrently and therefore the parent of the order-node becomes new order-node during reattempt, line 126.

Having performed the marking of right-link of $x(k)$ or tagging of the parent-link of its order-node we take next steps in the functions HELPMARKRIGHT or HELPTAGGEDPREPAR respectively. We first discuss the function HELP-

```
91  void HELPTHFLAGGED(Node preNode, Node dNode)
92  begin
93      Link oLink, rtLink, ptLink, rtRightLink, ptRtLink;
94      Node preParent, rightNode, parent; bool ptDir;
95      while (true ) do
96          preParent = preNode·backLink;
97          oLink = preNode·((preNode = dNode ) ? lChild : rChild);
98          rtLink = dNode·rChild;
99          if ((rtLink instanceof MkLink) or (rtLink instanceof ThMkLink)) then
100             HELPMARKRIGHT(dNode); break;
101         if ((oLink typeof ThFgLink ) and (oLink·ref = dNode )) then
102             if ((preNode = dNode ) or (preParent = dNode )) then
103                 if ((rtLink typeof UtLink ) or (rtLink typeof ThLink )) then
104                     if CAS(dNode·rChild, rtLink, rtLink·mark()) then
105                         HELPMARKRIGHT(dNode ); break;
106                 rightNode = rtLink·ref;
107                 if (rtLink typeof FgLink ) then
108                     HELPFLAGGED(dNode , rtLink, false );
109                 else if (rtLink typeof ThFgLink ) then
110                     HELPTHFLAGGED(dNode , rightNode);
111                 else if (rtLink typeof TgLink ) then
112                     rtRightLink = rightNode·rChild;
113                     if (rtRightLink typeof ThFgLink ) then
114                         HELPTAGGEDPREPAR(dNode , rightNode, rtRightLink·ref);
115             else
116                 ptDir = preNode·key < preParent·key;
117                 ptLink = preParent·(ptDir ? lChild : rChild);
118                 if (!ptDir and (ptLink·ref = preNode ) and (ptLink typeof UtLink )) then
119                     if CAS(preParent·rChild, ptLink, ptLink·tag()) then
120                         HELPTAGGEDPREPAR(preParent, preNode, dNode ); break;
121                 if (!ptDir and (ptLink typeof TgLink ) and (ptLink·ref = preNode )) then
122                     HELPTAGGEDPREPAR(preParent, preNode, dNode ); break;
123                 if (!ptDir and (ptLink typeof MkLink ) and (ptLink·ref = preNode )) then
124                     HELPMARKRIGHT(preParent);
125                 if ((ptLink typeof ThFgLink ) and (ptLink·ref = dNode )) then
126                     preNode = dNode;
127                 else if (ptLink typeof FgLink ) then
128                     HELPFLAGGED(preParent, ptLink, ptDir);
129         else
130             parent = dNode·backLink; ptDir = dNode·key < parent·key;
131             ptLink = parent·(ptDir ? lChild : rChild);
132             if ((ptLink typeof FgLink ) and (ptLink·ref = dNode )) then
133                 HELPFLAGGED(parent, ptLink, ptDir);
134             break;
```

```
135 void HELPTAGGEDPREPAR(Node preParent, Node preNode, Node dNode)
136 begin
137     Link oLink, preParLink, preLtLink;
138     while (true ) do
139         preLtLink = preNode·lChild;
140         preParLink = preParent·backLink;
141         oLink = preNode·rChild;
142         rtLink = dNode·rChild;
143         if ((rtLink instanceof MkLink) or (rtLink instanceof ThMkLink)) then
144             HELPMARKRIGHT(dNode); break;
145         if ((oLink typeof ThFgLink ) and (oLink·ref = dNode )) then
146             if ((preParLink typeof TgLink ) and (preParLink·ref = preNode )) then
147                 if ((preLtLink typeof UtLink ) or (preLtLink typeof ThLink )) then
148                     if CAS(preNode·lChild, preLtLink, preLtLink·tag()) then
149                         if (dNode·preLink = null ) then
150                             dNode·preLink = preNode;
151                         HELPTAGGEDPRELEFT(dNode ); break;
152                 else if (preLtLink typeof ThFgLink ) then
153                     CAS(preParent·rChild, preParLink, oLink·tag());
154                     CAS(preNode·rChild, oLink, oLink·finalize());
155                     HELPTHFLAGGED(preParent, dNode ); break;
156                 else if ((preLtLink typeof TgLink) or (preLtLink typeof ThTgLink)) then
157                     if (dNode·preLink = null ) then
158                         dNode·preLink = preNode;
159                     HELPTAGGEDPRELEFT(dNode ); break;
160                 else if (preLtLink typeof FgLink ) then
161                     HELPFLAGGED(preNode, preLtLink, true );
162             else
163                 if ((preLtLink typeof TgLink ) or (preLtLink typeof ThTgLink )) then
164                     HELPTAGGEDPRELEFT(dNode );
165                 break;
166         else
167             if ((preParLink typeof ThFgLink ) and (preParLink·ref = dNode )) then
168                 HELPTHFLAGGED(preParent, dNode ); break;
169             else
170                 parent = dNode·backLink; ptDir = dNode·key < parent·key;
171                 ptLink = parent·(ptDir ? lChild : rChild);
172                 if ((ptLink typeof FgLink ) and (ptLink·ref = dNode )) then
173                     HELPFLAGGED(parent, ptLink, ptDir);
174                 break;
```

TAGGEDPREPAR which takes care of the next step in the REMOVE of a category 3 node, see lines 136 to 174. Here the next step is to tag the left-link of the order-node only if it is either a clean threaded or a clean unthreaded link. Please note that this link can not be found marked as the right-link of the order-node is flagged. If the link to tag is found unthreaded and flagged then it indicates that the left-child node of the order node is being removed and that is helped. However, if this link is found threaded and flagged then it indicates that the order node itself is undergoing REMOVE. To handle that scenario, we just update the parent-link of the order-node to connect to the node $x(k)$ by copying the flagged order-link of $x(k)$ at the right-link of the parent of the order-node. After that we finalize the right-link of the order-node to indicate that it has been removed from the BST. After that the category 3 node may have changed to a category 2 node or a category 3 node with a different order-node with its order-link flagged. Hence we go back to the function HELPTHFLAGGED to take the next appropriate step. As earlier, here also before any attempt of a desired CAS, the right-link of the node $x(k)$ is checked and if found marked then we directly go to perform the remaining steps. Before an attempt to tag the left-link of the order-node, if the order-link is found not connected to the node $x(k)$ or not threaded and flagged then it indicates that either it has been updated in the final link update steps or it has been finalized indicating the removed order-node. All these scenarios are handled in the lines 166 to 174. Having tagged or having found tagged the left-link of the order-node of $x(k)$ we set its $preLink$ to point to the order-node if it is not already set, lines 150 and 158, before going to mark its right-link as the next step in the function HELPTAGGEDPRELEFT.

The function HELPTAGGEDPRELEFT performs the step of marking the right-link of a category 3 node. As mentioned before, the right-link is marked only if it is a clean link, otherwise appropriate helping is performed before a reattempt. See lines 176 to 192.

The step after marking the right-link of the node $x(k)$ is either to flag its incoming parent-link (category 1 nodes) or to mark its left-link (category 2 or 3 nodes). That is done in the function HELPMARKRIGHT, lines 194 to 217. Here before taking any CAS step it is checked whether the right-link has already been finalized. If that is so then the node to which the right-node is connected, is returned. In case of a category 1 node it is the parent-node referred by the back-link of $x(k)$ and in case of category 2 or 3 nodes it is the order node of $x(k)$ which is referred by the marked left-link before the right-link is finalized. See line 197. If the node is a category 2 or 3 node, which is ascertained by checking whether the left-child of $x(k)$ points to itself, before an attempt to mark the left-link of the node, the right-link is checked and if it is finalized then the function returns accordingly. If the left-link is found flagged then the concurrent

```
175 void HELPTAGGEDPRELEFT(Node dNode)
176 begin
177   Link rtLink, rtRightLink; Node rtNode;
178   while (true ) do
179     rtLink = preNode·rChild; rtNode = rtLink·ref;
180     if ((rtLink typeof UtLink ) or (rtLink typeof ThLink )) then
181       if CAS(dNode·rChild, rtLink, rtLink·mark()) then
182         HELPMARKRIGHT(dNode ); break;
183     else if ((rtLink instanceof MkLink ) or (rtLink instanceof ThMkLink )) then
184       HELPMARKRIGHT(dNode ); break;
185     else if (rtLink typeof FgLink ) then
186       HELPFLAGGED(dNode, rtLink, false );
187     else if (rtLink typeof ThFgLink ) then
188       HELPTHFLAGGED(dNode, rtNode);
189     else if (rtLink typeof TgLink ) then
190       rtRightLink = rtNode·rChild;
191       if (rtRightLink typeof ThFgLink ) then
192         HELPTAGGEDPREPAR(dNode , rtNode, rtRightLink·ref);

193 Node HELPMARKRIGHT(dNode)
194 begin
195   Link rtLink = dNode·rChild, ltLink= dNode·lChild; Node ltNode = ltLink·ref;
196   if ((rtLink typeof MkLinkFinal ) or (rtLink typeof ThMkLinkFinal )) then
197     return (ltNode = dNode ? dNode·backLink : ltNode);
198   if (ltNode = dNode ) then
199     Node parent = FLAGPARENT(dNode );
200     if (parent ≠ null ) then
201       bool parentDir = dNode·key < parent·key;
202       Link parentLink = parent·(ptDir ? lChild : rChild);
203       if ((parentLink typeof FgLink ) and (parentLink·ref = dNode )) then
204         CLEANFIRST(parent, parentLink, dNode, rtLink, parentDir);
205     return dNode·backLink;
206   else
207     while (true ) do
208       rtLink = dNode·rChild, ltLink= dNode·lChild; ltNode = ltLink·ref;
209       if ((rtLink typeof MkLinkFinal ) or (rtLink typeof ThMkLinkFinal )) then break;
210       if (ltLink typeof UtLink ) then
211         if CAS(dNode·lChild, ltLink, ltLink·mark()) then
212           HELPMARKLEFT(dNode ); break;
213       else if (ltLink typeof MkLink ) then
214         HELPMARKLEFT(dNode ); break;
215       else if (ltLink typeof FgLink ) then
216         HELPFLAGGED(dNode , ltLink, true );
217     return dNode·lChild·ref;
```

REMOVE of the left node is helped before reattempting the CAS. Here in case of category 1 nodes, with flagging of the parent-link all its incoming and outgoing traversal links become fixed and the steps of updating the incoming links and finalizing the right-link is done in the function CLEANFIRST, see lines 265 to 270.

```
218 void HELPMARKLEFT(dNode)
219 begin
220   Link ltLink = dNode·lChild;
221   Node ltNode = ltLink·ref, preNode = dNode·preLink ;
222   if ((preNode ≠ null ) and (ltNode ≠ preNode)) then
223     Node preParent = preNode·backLink;
224     Link preParentLink = preParent·rChild;
225     Link preLeftLink = preNode·lChild; oLink = preNode·rChild; Node
        preLeftNode = preLeftLink·ref;
226     if ((oLink typeof ThFgLink ) and (oLink·ref = dNode )) then
227       if ((preLeftLink typeof TgLink ) or (preLeftLink typeof ThTgLink )) then
228         if ((preParentLink typeof TgLink ) and
            (preParentLink·ref = preNode)) then
229           if (preLeftNode ≠ preNode) then
230             CAS(preLeftNode·backLink, preNode, preParent);
231           CAS(preParent·rChild, preParentLink, UtLink(preLeftNode));
232         CAS(ltNode·backLink, dNode, preNode);
233         CAS(preNode·lChild, preLeftLink, UtLink(leftNode));
234       CAS(preNode·backLink, preParentNode,dNode);
235       if ((ltLink typeof MkLink ) and (ltLink·ref = preNode)) or CAS(dNode
          ·lChild, ltLink, MkLink(preNode)) then
236         HELPMARKLEFT(dNode );
237     else
238       Node parent = dNode·backLink;
239       bool parentDir = dNode·key < parent·key;
240       Link parentLink = parent·(ptDir ? lChild : rChild);
241       if ((parentLink typeof FgLink ) and (parentLink·ref = dNode )) then
242         Link rtLink= dNode·rChild;
243         if ((rtLink typeof MkLink ) or (rtLink typeof ThMkLink )) then
244           CLEANSECOND(parent, parentLink, dNode,
              ltNode, rtLink, parentDir);
245   else
246     Node parent = FLAGPARENT(dNode );
247     if (parent ≠ null ) then
248       bool parentDir = dNode·key < parent·key;
249       Link parentLink = parent·(ptDir ? lChild : rChild);
250       if ((parentLink typeof FgLink ) and (parentLink·ref = dNode )) then
251         Link rtLink= dNode·rChild;
252         if ((rtLink typeof MkLink ) or (rtLink typeof ThMkLink )) then
253           CLEANSECOND(parent, parentLink, dNode,
              ltNode, rtLink, parentDir);
```

Having marked the left-link of a category 2 node, the next step is to flag the

incoming parent-link. That is done in the function HELPMARKLEFT, lines 219 to 253. The preLink of a category 2 node is never set and so it is null and that is checked to distinguish between a category 2 and category 3 node whose left-link is marked, 222. Same as before, the flagging of the parent-link, line 246, is the last state change step of a link connected with a category 2 node and after that the update of the incoming links and finalizing the right-link is performed in the function CLEANSECOND, see lines 272 to 280.

```
254 void HELPFLAGGED(Node parent, Link pLink, bool pDir)
255 begin
256     Node delNode = pLink·ref;
257     Link rtLink = delNode·rChild;
258     if ((rtLink typeof MkLink ) or (rtLink typeof ThMkLink )) then
259         ltNode = dNode·lChild·ref;
260         if (ltNode = delNode) then
261             CLEANFISRT(parent, pLink, delNode, rtLink, pDir );
262         else
263             CLEANSECOND(parent, pLink, delNode, ltNode, rtLink, pDir );
```

```
264 void CLEANFISRT(Node parent, Link pLink, Node dNode, Link rtLink, bool pDir)
265 begin
266     Node rtNode = rtLink·ref;
267     if ((rtLink typeof MkLink ) and (rtNode·backLink = dNode )) then
268         CAS(rtNode·backLink, dNode, parent);
269     CAS(parent·(pDir ? lChild : rChild), pLink, rtLink ·unmark());
270     CAS(dNode·rChild, rtLink, rtLink ·finalize());
```

```
271 void CLEANSECOND(Node parent, Link pLink, Node dNode, Node oNode, Link rtLink, bool pDir)
272 begin
273     Node rtNode = rtLink·ref; Link oLink = oNode·rChild;
274     if ((oLink typeof ThFgLink ) and (oLink·ref = dNode )) then
275         if ((rtLink typeof MkLink ) and (rtNode·backLink = dNode )) then
276             CAS(rtNode·backLink, dNode, oNode);
277         CAS(oNode·rChild, oLink, rtLink ·unmark());
278     CAS(oNode·backLink, dNode, parent);
279     CAS(parent·(pDir ? lChild : rChild), pLink, UtLink (oNode));
280     CAS(dNode·rChild, rtLink, rtLink ·finalize());
```

For a category 3 node, before its parent-link is flagged, its order-node is shifted to get connected in between the node and its left-child, lines 229 to 234. This way of converting a category 3 node undergoing REMOVE to a category 2 node with its left-link marked reduces the contention window significantly and the remaining steps of its REMOVE become much simplified. After this modification the function HELPMARKLEFT is again called and the check at line 222 ensures that the next step will be flagging of the incoming parent-link to the node $x(k)$.

In the function CLEANFIRST, if the marked right-link points to the right-

```
281 Node FLAGPARENT(Node dNode)
282 begin
283   Node parent, ptRtNode, ptLtNode; bool ptDir;
284   Link ptLink, rtLink, ptLtLink, ptRtLink ;
285   Link ptLtLtLink, ptLtRtLink, ptRtLtLink, ptRtRtLink;
286   while (true ) do
287     parent = dNode·backLink;
288     ptLtLink = parent·lChild; ptRtLink = parent·rChild;
289     ptLtNode = ptLtLink·ref; ptRtNode = ptRtLink·ref;
290     ptLtLtLink = ptLtNode·lChild; ptLtRtLink = ptLtNode·rChild;
291     ptRtLtLink = ptRtNode·lChild; ptRtRtLink = ptRtNode·rChild;
292     ptDir = dNode·key < parent·key;
293     ptLink = parent·(ptDir ? lChild : rChild);
294     rtLink = dNode·rChild;
295     if ((rtLink typeof MkLinkFinal ) or (rtLink typeof ThMkLinkFinal )) then return null;
296     if (ptLink·ref = dNode ) then
297       if (ptLink typeof UtLink ) then
298         if CAS(parent·(ptDir ? lChild : rChild), ptLink, ptLink·flag()) then
299           return parent;
300       else if (ptLink typeof FgLink ) then
301         return parent;
302       else if (ptLink typeof MkLink ) then
303         if (ptDir) then
304           HELPMARKLEFT(parent);
305         else
306           HELPMARKRIGHT(parent);
307       else if (ptDir and (ptLink typeof TgLink )) then
308         if ((ptRtLink typeof ThFgLink ) and (ptRtNode·preLink = parent)) then
309           HELPTAGGEDPRELEFT(ptRtNode);
310     else
311       if (ptRtLtLink·ref = dNode) then
312         if (ptRtLink typeof ThFgLink and ptRtLtLink typeof MkLink) then
313           HELPMARKLEFT(ptRtNode);continue;
314         else if (ptRtLink typeof FgLink and ptRtLtLink typeof MkLink) then
315           HELPFLAGGED(parent, ptRtLink, false );continue;
316         else if (ptRtLink typeof TgLink and ptRtLtLink typeof TgLink) then
317           If(ptRtRtLink typeof ThFgLink and ptRtRtLink·ref·preLink =
              ptRtNode) HELPTAGGEDPRELEFT(ptRtRtLink·ref); continue;
318       else if (ptRtRtLink·ref = dNode) then
319         if (ptRtLink typeof FgLink and ptRtRtLink typeof MkLink) then
320           HELPFLAGGED(parent, ptRtLink, false);
321         else if (ptRtLink typeof ThFgLink and ptRtRtLink typeof MkLink) then
322           HELPMARKRIGHT(ptRtNode);
323       else if (ptLtLtLink·ref = dNode) then
324         if (ptLtLink typeof FgLink and ptLtLtLink typeof MkLink) then
325           HELPFLAGGED(parent, ptLtLink, true);
326       else if (ptLtRtLink·ref = dNode) then
327         if (ptLtLink typeof FgLink and ptLtRtLink typeof MkLink) then
328           HELPFLAGGED(parent, ptLtLink, true);
329       else
330         return null;
```

child node (and not the successor node had it been threaded), its backLink is connected to the parent of the node $x(k)$, line 268. After that the incoming flagged parent-link is connected to either the right-child or the successor node of the $x(k)$. If it is connected to the successor it becomes clean and threaded otherwise it becomes a clean and unthreaded link, line 269. After that the marked right-link is finalized to indicate the completion of REMOVE operation, line 270. The function CLEANSECOND in the same way is used to update the incoming links of a category 2 node or a category 3 node whose left-link has been already connected to the order-node. Here the flagged parent-link is connected to the order-node and becomes a clean and unthreaded link and the flagged order-link is connected to either the successor or the right-child of the node. And as before, the backLinks of the left-child and right-child nodes are updated before they are respectively connected to the parent and the order node of $x(k)$, see lines 272 to 280. Please note that it ensures that whenever we read the backLink of a node it is guaranteed to be connected to a node which is at most 2 links away from the node. This fact is utilized when we try to flag the incoming parent-link in the function FLAGPARENT. The function HELPFLAGGED, lines 255 to 263, is used to direct a helping operation to the appropriate clean function, if it is obstructed at some state-change step.

The function FLAGPARENT, lines 282 to 330, first reads the node referred by the backLink of the node $x(k)$. Before the CAS is attempted, the outgoing links of both left and right children of the parent are read and stored locally. This is done because on failing the CAS due to the reason that the read parent-link was not connected to the node $x(k)$, we need to help the pending REMOVE operation at the node pointed by the backLink. As explained before, the backLink of a node can point to a node which is at most two links away, we find the appropriate helping to perform by reading outgoing links of the node that is connected with the read parent-link, see lines 310 to 330. However, before going further to attempt the CAS execution to flag the incoming parent-link, the right-link of the node is checked to make sure that the node is still part of the BST if its right-link is not finalized. If on a failed CAS, the parent-link is found connected to the node $x(k)$ then either the required helping is performed or if a concurrent helping operation had already flagged the link then the parent is returned, see lines 300, 302 and 307. The function returns null if either the right-link is found finalized or the node pointed by the backLink is found connected to a node from which the node $x(k)$ is not reachable by traversing up to two nodes. In both the cases the REMOVE operation is considered finished.

The return value of REMOVE($x(k)$) is true only if it could successfully flag the order-link of the node $x(k)$ otherwise false is returned.

Add operation

```
331 bool ADD(KType key)
332 begin
333     Location loc; bool turn; KType curKey;
334     Node par = pRoot, cur, next, curRight, nextLeft, nextRight ;
335     Link curLink, curRtLink, nextRtLink;
336     Node node = new Node(key, ThLink(key, node), null, null, node);
337     while (true) do
338         loc = LOCATE(par, key), cur = loc·cur;
339         curRtLink = cur·rChild, curKey = cur·key;
340         if (curKey = key) then                          /* key exists in the BST */
341             return false;
342         else
343             turn = key < curKey;
344             curLink = cur·(turn ? lChild : rChild); next = curLink·ref;
345             if ((curLink typeof ThLink) and (curLink·ref = next)) then
346                 node·rChild = curLink; node·backLink = cur;
347                 if CAS(cur·(turn ? lChild : rChild), curLink, UtLink(key, node)) then
348                     return true;
349             if (curLink·ref = next) then
350                 if (curLink typeof ThMkLink) then
351                     par = HELPMARKRIGHT(cur);
352                 else if (curLink typeof ThFgLink) then
353                     HELPTHFLAGGED(cur, next);
354                     nextLeft = next·lChild·ref;
355                     if (curLeft = cur) then
356                         par = next·backLink ;
357                     else
358                         par = nextLeft;
359                 else if (curLink typeof ThTgLink) then
360                     nextRtLink = next·rChild; nextRight = nextRtLink·ref ;
361                     if ((nextRtLink typeof ThFgLink) and (nextRight·preLink = next)) then
362                         HELPTAGGEDPRELEFT(nextRight);
363                     par = loc·pre;
```

To ADD a new node with key k in a BST, starting from the node $pRoot$ we LOCATE the target interval $[k_i,\ k_j]$ that k belongs to, associated with a threaded link. If the **Location** variable returned by LOCATE is loc and the node $loc{\cdot}cur$ has the key as the query key then it indicates that the BST already contains a node with the query key and therefore ADD returns false, line 341. If such a node is not found in the BST then the created new node at line 336 is to be ADDed. Having ascertained the left or right link of $loc{\cdot}cur$ which the new node must be attached to, we check the state of the link. If it was a clean and threaded link then the right-link of the new node takes the value of the link to which it needs to be connected, and the backlink is pointed to the node $loc{\cdot}cur$, line 346. Note that when a new node is ADDed to a BST, both its outgoing links

are threaded. The link which it needs to connect to is modified in one atomic step to point to the new node using a CAS. If the CAS succeeds then true is returned, line 347. On failure, if the threaded link, which was the new node to connect to, is found flagged, marked or tagged then appropriate helping is performed. However if the target link was found connected to a new node then another ADD operation succeeded to add a new node after we read the link and therefore we locate the target link before reattempt. See lines 332 to 363.

2.4 Correctness and Complexity

2.4.1 Correctness

In this section we present the formal proof of correctness of the presented algorithm and show that the algorithm is linearizable to a sequential BST and then prove its lock-freedom. First we give some basic definitions that will be used in the proof.

We consider a *shared memory system* U. In U let Λ be the finite set of *threads* and V be the finite set of *shared variables* on which a thread $\lambda \in \Lambda$ executes atomic operations provided by the system i.e. atomic read, write and compare-and-swap (CAS) in order to communicate. Each thread $\lambda \in \Lambda$ performs a sequence of *steps* on shared variables $v \in V$. A step comprises of local computations of the thread and at most a single atomic operation o on a $v \in V$. The *state* of a thread λ, denoted by $S(\lambda)$, is defined as a variable consisting of the values of the thread's local variables, registers and program counter. Let S be the set of states of all the threads. A *configuration* C of the shared memory system U is defined as a set assembly consisting of the set of state of all the threads and the set of shared variables i.e. $C = \{S,\ V\}$, so that $|C| = |S| + |V|$. An *execution* E is a sequence of the form $\{C_k^-, s_k, C_k^+\}_{k \in \mathfrak{K}}$ where the index set $\mathfrak{K}$ can be finite or countably infinite. E starts from an initial configuration C_{k_0} consisting of initial set of shared variable states in the system and all $\lambda \in \Lambda$ in their initial states. We express $C_{k_i}^+ \leftarrowtail s_{k_i}(C_{k_i}^-)$ to mean that on being operated with step s_{k_i} in an execution E, the configuration $C_{k_i}^-$ changes to $C_{k_i}^+$.

In U a binary search tree is an object that supports the set operations ADD ($A(k)$), REMOVE ($R(k)$) and CONTAINS ($C(k)$). The state of a BST Υ at time t is a finite set of nodes denoted as $\Upsilon_t = \{x_i(k_i)\}_{i=0}^{r-1}$ forming a directed graph. The nodes are connected with links and the nodes and the links together have properties as described in the section 2.2. There is a specific node $Root \in \Upsilon_t$ from where every node $x_i(k_i) \in \Upsilon_t$ is reachable following the $lChild$ and $rChild$ links of other nodes. It implies that to remove the node $x_i(k_i)$ from

Υ_t, we need to ensure that it can not be reached from any other node and hence from the node $Root$.

Definition 2.1. *At time t the BST state Υ_t is called valid iff $\forall$ nodes $x(k_x), y(k_y) \in \Upsilon_t$, $x(k_x) \neq y(k_y) \Rightarrow k_x \neq k_y$. Further if $y(k_y)$ is in the left-subtree of $x(k_x)$ then $k_y < k_x$ and if that is in the right-subtree of the same then $k_y > k_x$.*

At time $t = 0$, the BST state $\Upsilon_0 = \{cRoot, pRoot\}$ satisfies the above requirement in which $cRoot$ is the only node in the left-subtree of the node $pRoot$. The right-subtree of $pRoot$ as well as both the subtrees of $cRoot$ are null.

*A valid BST state $\Upsilon_t = \{x_i(k_i)\}_{i=0}^{r-1}$ corresponds to an abstract set $K = \{k_i\}_{i=0}^{r-1}$ with r elements and the set operations $A(k)$, $R(k)$ and $C(k)$ provide following sequential specifications. The **if** clause on the LHS of $\Rightarrow$ is the precondition of the set transition on RHS of the same which itself is the postcondition of the execution in the **then** clause.*

$$\textbf{if } k \in K \textbf{ then } C(k)(\Upsilon_t) = \text{true} \Rightarrow K \rightarrowtail K \tag{2.1}$$

$$\textbf{if } k \notin K \textbf{ then } C(k)(\Upsilon_t) = \text{false} \Rightarrow K \rightarrowtail K \tag{2.2}$$

$$\textbf{if } k \notin K \textbf{ then } A(k)(\Upsilon_t) = \text{true} \Rightarrow K \rightarrowtail K \cup \{k\} \textbf{ and } K \cup \{k\} \textbf{ corresponds to a valid BST} \tag{2.3}$$

$$\textbf{if } k \in K \textbf{ then } A(k)(\Upsilon_t) = \text{false} \Rightarrow K \rightarrowtail K \tag{2.4}$$

$$\textbf{if } k \in K \textbf{ then } R(k)(\Upsilon_t) = \text{true} \Rightarrow K \rightarrowtail K \setminus \{k\} \textbf{ and } K \setminus \{k\} \textbf{ corresponds to a valid BST} \tag{2.5}$$

$$\textbf{if } k \notin K \textbf{ then } R(k)(\Upsilon_t) = \text{false} \Rightarrow K \rightarrowtail K \tag{2.6}$$

Definition 2.2. *A concurrent BST is a shared object in which (address of) nodes and links are shared variables $v \in V$ having the properties of a BST and whose operations in the shared memory system U are linearizable [15] to the operations maintaining a valid BST as defined in* DEFINITION *2.1. Further, a concurrent BST is said to be lock-free if at least one non-faulty thread $\lambda \in \Lambda$ in an execution E in U is guaranteed to finish its operation in finite number of its own steps in E.*

Now we prove some invariants of the presented algorithm in form of lemmas in order to prove that at all time $t \geq 0$ it maintains a valid BST. Essentially, the lemmas will show that no null variable is dereferenced and further if in an execution E, a step s is performed by a $\lambda \in \Lambda$ over a configuration $C \supseteq \Upsilon_t$ so that it changes to a configuration $C' \supseteq \Upsilon'_t$ then $\Upsilon_t \; is \; valid \Rightarrow \Upsilon'_t \; is \; also \; valid$. After that we shall prove the linearizability of the set operations to prove a correct

concurrent linearizable implementation. And finally we shall prove the lock-freedom and hence we shall prove that the efficient lock-free BST algorithm presented in this paper implements a correct linearizable lock-free concurrent BST.

It is trivial to observe that after a node is ADDed in a BST following our algorithm, its key never changes and all the links outgoing from it are modified using atomic CAS only. Because we can not ADD or REMOVE a key k such that $k \geq (\infty_0)$ so the sentinel nodes can neither be REMOVEd nor the right-link of $cRoot$ and the links of $pRoot$ can change and for that matter nor the backLink or preLink of these nodes after once they are set as shown in the Fig. 2.2 (c). We use the node $pRoot$ as the node $Root$ to prove the validity of the abstract BST as described above.

Lemma 2.1. A null is never dereferenced at any step s during an execution E.

Proof. It can be observed that when a node is ADDed in the BST, all its fields except preLink are non-null. To turn a node to be unreachable, depending on the category of node, the incoming pointers are turned away from it in the functions HELPMARKLEFT, CLEANFIRST and CLEANSECOND at lines using atomic CAS. The $preLink$ field of a category 3 node is updated only once in the algorithm in the function HELPTAGGEDPRELEFT. After that when we need to retrieve the fields of the order-node referred by the preLink of a category 3 node in the function HELPMARKLEFT, we first check if it is non-null at line 222 and if not then it is categorised as a category 2 node, whose preLink is never dereferenced. The return of the function LOCATE is a class containing address of two non-null node variables and the HELPMARKLEFT never returns a null value. As the parameters passed to functions other than LOCATE are computed by dereferencing link fields of nodes either in function LOCATE or in other functions, no null pointer is passed to them. And whenever a LOCATE is called in the functions ADD REMOVE or CONTAINS the address passed to it is either that of $pRoot$, which is never REMOVEd as noted before, or the address returned by LOCATE or HELPMARKRIGHT. Hence a null pointer is never dereferenced at any s in E. □

Lemma 2.2. In a BST state Υ_t a link that connects a node to any of its children is not threaded.

Proof. In the initial state Υ_0 it is trivially true. Now suppose it is true in a state Υ_{i-1} and $\text{ADD}(x)(\Upsilon_{i-1}) \rightarrowtail \Upsilon_i$. At line 347 after CAS succeeds, the link between the node to which x is connected and x, is a **UtLink**. Hence, by induction the lemma proves. □

Corollary 2.1. A threaded link outgoing from a node ensures the subtree of the node in that direction is null.

Proof. We observed in the function ADD that a new node that has been initialized at line 336 its left and right outgoing links are threaded at the lines 336 and 346 and it has null subtrees in both the directions. Using the induction as in Lemma 2.1 this is proved. □

Lemma 2.3. In a BST state Υ_t, if a call to LOCATE terminates returning address of a node as $x(k) = loc.cur$, then either $k =$ key or $x(k)\cdot rChild\cdot isThd() = 1$, if key$> k$ or $x(k)\cdot lChild\cdot isThd() = 1$, if key$< k$.

Proof. A LOCATE terminates either at line 22 or 30 or 34. If termination happens at line 22 then $k =$ key. If it happens at line 30 then $x(k)\cdot rChild\cdot isThd() = 1$ and $k <$ key. And if it happens at line 34 then $x(k)\cdot lChild\cdot isThd() = 1$ and $k >$ key. □

Corollary 2.2. An ADD always happens at a clean and threaded link.

Proof. By lemma 2.3, when the LOCATE at line 338 terminates, if the key key does not match at the node $loc\cdot cur$ then the left or right child link at $loc\cdot cur$ must be threaded according to the above lemma. The CAS at line 347 ensures that the link that is modified here is clean and threaded. □

Lemma 2.4. At any step s in an execution E, a call of CONTAINS(key) over a valid BST state Υ_t at a $t\geq 0$ satisfies the sequential specifications (2.1) and (2.2) in Definition 2.1.

Proof. In a step s executed by a thread $\lambda\in\Lambda$, if a call to CONTAINS returns true at line 48, key $= curr\cdot key$ where $curr = loc\cdot cur$ and loc is returned by LOCATE. Now if that is not the case then either key $< curr\cdot key$ or key $> curr\cdot key$. By the validity of BST state Υ_t, and the fact that the locate has exited at a link which was threaded using Corollary 2.1, there can not be a node in the left-subtree if key $> curr\cdot key$ and similarly there can not be a node in the right-subtree if key $< curr\cdot key$. Hence the precondition of the sequential specifications is satisfied. It is trivial to observe that there is no update on a shared variable either in CONTAINS or in LOCATE and hence the postcondition is satisfied. □

Lemma 2.5. At any step s in an execution E, a call of ADD(key) over a valid BST state Υ_t at a $t\geq 0$ satisfies the sequential specifications (2.3) and (2.4) in Definition 2.1.

Proof. The proof of satisfying the precondition is along the lines of proof of Lemma 2.4. A call to ADD can return false only at line 341 and that can happen if either the rest of the lines are not executed or the CAS fails at line 347 and then in course of reattempt the ADD returns at line 341. In both the cases no change is made to Υ_t. If the CAS succeeds at the line 347 then there is an unthreaded link between the node $curr \in \Upsilon_t$ and $x(\mathsf{key})$ and so $x(\mathsf{key})$ can be reached from $pRoot$ via $curr$. If a successful CAS at line 347 is executed then by the validity of Υ_t and using lemma 2.2, $\Upsilon_t \cup x(\mathsf{key})$ is valid. Hence and ADD operation in our algorithm always maintains the validity of a BST. □

Lemma 2.6. At any step s in an execution E, from a valid BST state Υ_t at a $t \geq 0$, the REMOVE of a node $x(k) \in \Upsilon_t$ starts with flagging its order link.

Proof. By Lemma 2.3, if $x(k) \in \Upsilon_t$ then the key $(k\text{-}\epsilon)$ belongs to the interval associated with the order-link of $x(k)$. If $x(k) \in \Upsilon_t$ is a category 1 node then a LOCATE(k-ϵ) will terminate returning address of $x(k) = loc{\cdot}cur$ whose order-link emanates from itself and if it is a category 2 or category 3 node then it will terminate returning address of order-node of $x(k)$ as $loc{\cdot}cur$. Hence the link that is flagged is always order-link of $x(k)$. □

Lemma 2.7. (a) Whenever the function HELPTHFLAGGED is called the link connecting the nodes preNode and dNode is of type **ThFgLink**.

(b) Whenever the function HELPFLAGGED is called the the link pLink is of type **FgLink**.

(c) Whenever the functions HELPMARKLEFT is called the the left-link of the node dNode is of type **MkLink**.

(d) Whenever the functions HELPMARKRIGHT is called the the right-link of the node dNode is of type **MkLink**.

Proof. Trivial by observations. □

Lemma 2.8. For a link belonging to a node $x(k) \in \Upsilon_t$,

(a) once it is marked it can not be updated again.

(b) if it is tagged or flagged then it can only be updated to a clean unthreaded or threaded state.

Proof. Trivial by observations at the CAS operations performed in the algorithm. □

Lemma 2.9. At any step s in an execution E, in a valid BST state Υ_t at a $t \geq 0$, the $rChild$ link of a node $x(k) \in \Upsilon_t$ can not be marked unless its order-link is flagged and further if its $lChild$ link does not point to its order-node then its $preLink$ must point to the correct order-node.

Proof. $rChild$ link of a node is marked only at line 104 in the function HELPTHFLAG or at line 181 in the function HELPTAGGEDPRELEFT. Before these functions could be called in any execution, the function REMOVE must have been called with a successful CAS at line 69. This proves that the order-link must have been flagged for the node. Further the right-link is marked in the function HELPTHFLAG only after passing through the check at the line 102. This ensures that the node to remove either coincides with its order-node or the parent of the order-node. Now order-node of a node is always its predecessor by the design of the BST. So, it must be that the left-child link of the node must have been pointing to its order-node. Now, if the right-link is marked at the line 181 in the function HELPTAGGEDPRELEFT, then it must have been called from the function HELPTAGGEDPREPAR and in that case is $preLink$ must have been pointed to the correct order-node either at line 150 or at line 158. □

Lemma 2.10. At any step s in an execution E, in a valid BST state Υ_t at a $t \geq 0$, for a node $x(k) \in \Upsilon_t$ following hold

1. If $x(k)$ is a category 1 node, then before its incoming parent-link is flagged, its order-link is flagged and its $rChild$ link is marked.

2. If $x(k)$ is a category 2, then before its $lChild$ is marked, its order-link is flagged and its $rChild$ link is marked.

3. If $x(k)$ is a category 2, then before its parent-link is flagged, its order-link is flagged and its $rChild$ and $lChild$ links are marked.

4. If $x(k)$ is the order-node of a category 3 node, then before its incoming parent-link is tagged, its $rChild$ link is flagged and threaded.

5. If $x(k)$ is the order-node of a category 3 node, then before its $lChild$ is tagged, its $rChild$ link is flagged and its incoming parent-link is tagged.

6. If $x(k)$ is a category 2 node, then before its $lChild$ is marked, its order-link is flagged and its $rChild$ link is marked.

7. If $x(k)$ is a category 3 node, then before its incoming parent-link is flagged, its $lChild$ link is marked and points to its order-node.

Proof. The FLAGPARENT function for a category 1 node is called only from a HELPMARKRIGHT function and using the lemma 2.9 before the $rChild$ link of a node is marked its order-link must have been flagged. Therefore, before the parent-link of a category 1 node is flagged, its order-link must have been flagged and its $rChild$ marked. In the same way for a category 2 node before its $lChild$ link is marked in the function HELPMARKRIGHT, its order-link must have been flagged and its $rChild$ link marked. And the FLAGPARENT function on such a node is called only from the function HELPMARKLEFT. That proves that before the incoming parent-link is flagged for a category 2 node its its order-link is flagged and $lChild$ and $rChild$ links are marked. In the same way we can observe the order of call of tagging and flagging of the links connected to a category 3 node and its order-node. Clearly, the incoming parent-link of the order-node of a category 3 node is tagged in the function HELPTHFLAGGED and that ensures a flagged order-link of the node i.e. a flagged and threaded right-link of the order-node. The left-link of the order-node of a category 3 node is tagged only in the function HELPTAGGEDPREPAR at line 148. The call of the function HELPTAGGEDPREPAR ensures a tagged parent-link of the order-node. Similar to the case of a category 2 node, the left-child link of a category 3 node is marked only after its right-link is marked. And finally before the call of FLAGPARENT at the line 246, it must have been that the left-link points to the order-node by check at the line 222 ensuring that the left-link is marked and pointed to the order-node. □

Lemma 2.11. At any step s in an execution E, before any traversal link (i.e. left-link or right-link of a node present in the BST) incoming to a node $x(k) \in \Upsilon_t$ at a $t \geq 0$, is moved away from it, following hold

1. If $x(k)$ is a category 1 node then its (a) order-link is flagged, (b) $rChild$ link is marked, and the (c) incoming parent-link is flagged.

2. If $x(k)$ is a category 2 node then its (a) order-link is flagged, (b) $rChild$ link is marked, (c) $lChild$ link is marked, and the (c) incoming parent-link is flagged.

3. If $x(k)$ is a category 3 node, (a) order-link is flagged, (b) $rChild$ link is marked, c) $lChild$ link is marked, and the (c) incoming parent-link is flagged.

Proof. The first incoming link to be pointed away from a category 1 node is its parent-link and that for a category 2 or 3 node is its order-link. The update of the parent-link of a category 1 node happens in the function CLEANFIRST and the update of the order-link of a category 2 or 3 node happens in the function

CLEANSECOND. These functions are called only if the parent-link is flagged and then using lemma 2.10, it follows. □

Lemma 2.12. At any step s in an execution E, before the parent-link incoming to a node $x(k) \in \Upsilon_t$ at a $t \geq 0$, is moved away from it all other links, including the $backLink$ of its children if any, are moved away from it.

Proof. The update of the flagged parent-link of a node happens in the function CLEANFIRST or CLEANSECOND as the absolutely last `CAS` operation that moves a link away from a node at line 269 or 279. Before the first successful execution of the `CAS` in these lines, all the previous `CAS` operations must have succeeded to move away all the links connected to a node. □

Lemma 2.13. At any step s in an execution E, in a valid BST state Υ_t at a $t \geq 0$, the shifting of the order-node of a category 3 node in the function HELPMARKLEFT does not violate the symmetric order of the BST. Further, order-node remains reachable from the $Root$ during its shifting.

Proof. When the order node is shifted in the function HELPMARKLEFT, the update of traversal links in the BST ensure that the (a) incoming parent-link of the order-node is connected to the left-child node of the order-node if it is in the BST, (b) left-link of the order-node is connected to the left-child node of the category 3 node and (c) left-link of the category 3 node is connected to its order-node, in this order. Clearly, with the validity of the BST before (a), it is ensured that the nodes maintain the symmetric order. After (a), the right-child of the parent of order-node is a node in its right-subtree before (a). So, with the validity before (a), the validity after (a) is guaranteed. Similarly before (b) the order-node is a node in the right-subtree of the left-child node of the node under REMOVE and hence after (b) the left-child of the order-node has a key less than that of itself. Same holds for the change (c). Clearly none of these changes violate the symmetric order of the BST. Other than these three changes, no traversal link is changed due to the shifting of the order-node of a category 3 node. It implies that if the order-link of the order-node was incoming to it, emanating from another node in its left-subtree, then it remains so and if the order-link was emanating from itself then after the shifting, a threaded-link from its previous parent now connects to it. In both these cases, unless the tagged parent-link incoming to the order-node is updated to a clean threaded or unthreaded link, the parent of the order-node can not be removed following the lemma 2.12. That ensures that following the right-link of the previous parent of the order-node it is reachable by any traversal. □

Lemma 2.14. The $backLink$ of a node $x(k) \in \Upsilon_t$ at a $t \geq 0$, always points to a node that is present in Υ_t and is at most two links away.

Proof. It follows from lemma 2.12 that before the flagged parent-link of a node is moved away from it, the $backLink$ of its children pointing to it at the time the parent-link is flagged, are pointed away to either connect to its parent or to its order-node. Also from 2.13, when the order-node of a category 3 node is shifted, the back-link pointer from a possible left-child of it is connected to the parent of the order-node before the tagged parent-link is connected to the left-child of the same. Similarly, the backlink of the left-child node of a category 3 node is connected to its order-node before the left-link of the order-node is connected to it. And then the $backLink$ of the order-node is connected to the node whose parent-link is still to be flagged. This shows that the $backLink$ of a node always points to a node present in the BST.

Moreover, it can easily be observed that the parent-link of the parent of a node emanates from a node from where, it would take at most two links to reach the node. And in case of a order-node shifting, from the order-node it takes one flagged order-link and one marked left-link to reach the left-child node of a category 3 node. It is trivial to observe that when the backLinks are not under update during a REMOVE they always point to the parent of the node from where a single travel is needed to reach the node. Hence it can be seen that from a node which is pointed by a $backLink$ field, a node can be reached by traversing at most two links. □

Corollary 2.3. If the FLAGPARENT function applied on a node $x(k)$ returns null, then $x(k) \notin \Upsilon_t$.

Proof. Observing thorough the function FLAGPARENT, it returns null only if (a) the $rChild$ link of $x(k)$ is found finalized or $x(k)$ is connected to neither of the child-links of the node pointed by the backLink nor to the child-links of the nodes pointed by those child-links. Using the lemma above it directly follows. □

Lemma 2.15. At any step s in an execution E, a call of REMOVE(key) over a valid BST state Υ_t at a $t \geq 0$, satisfies the sequential specifications (2.5) and (2.6) in Definition 2.1.

Proof. In keeping with the proof of lemma 2.4 the sequential specification (2.6) is satisfied by REMOVE(key). By lemma 2.13, the shifting of the order-node of a category 3 node maintains the symmetric order of the BST. By Lemma 2.6, if $x(\text{key}) \in \Upsilon_t$ then REMOVE(key) starts by flagging its order link. By lemma 2.11, before $x(\text{key})$ is REMOVEd, all the incoming links to it are moved away

maintaining the symmetric order of the BST. By the Lemma 2.8, any new ADD or REMOVE operation can not be injected to a link unless it is clean. So, if Υ_t is valid then $\Upsilon_t \backslash x(\text{key})$ is also valid. Moreover, after the order link and parent-links are pointed away $x(\text{key})$ can not be reached from $pRoot$. Hence REMOVE(key) satisfies the sequential specification (2.5). □

Using Lemmas 2.4, 2.5 and 2.15, we arrive at proposition 2.1.

Proposition 2.1. *The the algorithm Efficient Lock Free BST in an execution E starting with initial configuration $C_0 \supseteq \Upsilon_0$ maintains a valid binary search tree state Υ_t, $\forall t \geq 0$.*

2.4.2 Linearizability

Having proved the above invariants of the lock-free BST algorithm we prove the linearizability.

Lemma 2.16. In a step s in an execution E, there exist linearization points of ADD, REMOVE and CONTAINS between their respective invocation and return, satisfying the sequential specifications of a valid BST state Υ_t for $t \geq 0$ given in Definition 2.1.

Proof. CONTAINS - A thread performing a CONTAINS(key) essentially performs a LOCATE with key starting from the node $pRoot$. Now if LOCATE terminates returning $x(\text{key})$ so that CONTAINS(key)(Υ_t) = true then the point at which the link pointing to $x(\text{key})$ was read in LOCATE, is taken as the linearization point. There is no other way that CONTAINS(key)(Υ_t) could return true as proved before. However, if CONTAINS(key)(Υ_t) = false then there could be two possibilities - (a) when CONTAINS(key) was invoked, $x(\text{key}) \in \Upsilon_t$ and (b) when CONTAINS(key) was invoked, $x(\text{key}) \notin \Upsilon_t$. In the first case because LOCATE could not reach the node $x(\text{key})$ although it was in Υ_t at the invocation point shows that a concurrent REMOVE operation REMOVEd the node and in that case the linearization point of CONTAINS(key)(Υ_t) = false is just after the linearization point of the successful concurrent REMOVE operation. In the second case the linearization point can well be taken as the invocation point of CONTAINS(key) returning false.

ADD - A thread performing an ADD(key) returns false if $x(\text{key}) \in \Upsilon_t$ and therefore as in case of CONTAINS, the linearization point of ADD(key)(Υ_t) = false is at the point where the link pointing to $x(\text{key})$ was read, which is between the invocation and return of the LOCATE called from ADD. For a successful ADD operation, LOCATE return the address of a node which does not

have the key key. And with that, the execution of the CAS at line 347 is where it takes effect and therefore it is the linearization point of ADD(key)(Υ_t) = true.

REMOVE - A thread performing a REMOVE(key) can return false in two ways (a) x(key)$\notin\Upsilon_t$ at the invocation of REMOVE and therefore is not located (b) x(key)$\in\Upsilon_t$ at the invocation of REMOVE but got REMOVEd by a concurrent successful REMOVE. These are the cases similar to those in the CONTAINS operation and so the linearization points are same as in that case. For a REMOVE(key) returning true, we should choose a point between its invocation and return such that it is consistent with the return of the concurrent CONTAINS operations. A REMOVE returns true only if it could successfully perform the CAS at line 69. However, key can still be located until the incoming parent-link is pointed away. So we must have the linearization point of the REMOVE(key) returning true at the execution of the successful CAS that swaps the incoming parent-link of x(key)$\in\Upsilon_t$ so that a concurrent CONTAINS returning false linearizes just after that. However, it could be done by a helping operation and that will still be between the invocation and return point of the REMOVE that successfully flags the order-link. □

Using the Proposition 2.1 and the Lemma 2.16, the proposition 2.2 follows.

Proposition 2.2. *The algorithm Efficient Lock Free BST implements a valid and linearizable concurrent binary search tree.*

2.4.3 Lock-Freedom

Lemma 2.17. If REMOVE(x) and REMOVE(y) work concurrently on nodes x and y then without loss of generality

(a) If x is a child of y and the link $[y, x]$ is flagged then REMOVE(x) finishes before REMOVE(y); otherwise if this link is marked, REMOVE(y) finishes before REMOVE(x).

(b) If x is the order-node of y, where y is a category 3 node, and the order-links of both x and y have been successfully flagged then REMOVE(x) finishes before REMOVE(y).

(c) If x is the order-node of y, where y is a category 2 node, and the order-links of both x and y have been successfully flagged then REMOVE(y) finishes before REMOVE(x).

(d) If x is the left-child of the order-node of y, where y is a category 3 node, and the link $[pre(y), x]$ is tagged then REMOVE(x) finishes before REMOVE(y); otherwise if this link is flagged, REMOVE(y) finishes before REMOVE(x).

(e) If x is the left-child of the order-node of y, where y is a category 2 node, and the link $[y, x]$ is marked then REMOVE(y) finishes before REMOVE(x).

(f) If x is the parent of the order-node of y, where y is a category 3 node, and the link $[x, pre(y)]$ is marked then REMOVE(x) finishes before REMOVE(y); otherwise if this link is tagged, REMOVE(y) finishes before REMOVE(x).

(g) In all other cases REMOVE(x) and REMOVE(y) do not obstruct each other.

Proof. Follows from the lemmas 2.8, 2.9 and 2.10. □

Lemma 2.18. *Lock-freedom is guaranteed in the algorithm Efficient Lock Free BST.*

Proof. By the description of the algorithm, a non-faulty thread performing CONTAINS will always return unless its search path keeps on getting longer forever. If that happens, an infinite number of ADD operations would have successfully completed adding new nodes making the implementation lock-free. So, it will suffice to prove that the modify operations are lock-free. Suppose that a thread $\lambda \in \Lambda$ performs a modify operation *op* on a valid BST state Υ_t and takes infinite steps and no other modify operation completes after that. Now, if no modify operation completes then Υ_t remains unchanged forcing λ to retract every time it wants to execute its own modification step on Υ_t. This is possible only if every time λ finds the injection point of *op* flagged, marked or tagged. This implies that a REMOVE operation is pending. It can be observed in our algorithm that in the function ADD if it gets obstructed by a concurrent REMOVE then before retrying after recovery from failure it helps the pending REMOVE by executing all the remaining steps of that. Also from lemma 2.17, whenever two REMOVE operations obstruct each other, one finishes before the other. It implies that whenever two modify operations obstruct each other one finishes before the other and so Υ_t changes. It is contrary to our assumption. Hence, by contradiction we show that no non-faulty thread shall remain taking infinite steps if no other non-faulty thread is making progress. □

The lemma 2.18 leads to the proposition 2.3.

Proposition 2.3. *The algorithm Efficient Lock Free BST implements a valid and linearizable lock-free concurrent binary search tree.*

2.4.4 Complexity

Having proved that our algorithm guarantees lock-freedom, though we can not compute worst-case time complexity of an operation, we can definitely derive

their amortized complexity. We derive the amortized step complexity of set operations in our implementation by the accounting method along the similar lines as in [12, 21]. For an execution E, let $\mathcal{O}$ be the set of operations. First we show that a traversal visits a node only a constant number of times and hence we bound the length of the traversal path. Then in the execution E we amortize the step complexity of the operations $op \in \mathcal{O}$.

Lemma 2.19. A thread $\lambda \in \Lambda$ executing an operation op visits a node $x \in \Upsilon_t$ only an $O(1)$ times during the traversal in the shared memory system U, if it does no modification.

Proof. As we have observed that the key of a node $x \in \Upsilon_t$ is immutable. If the nodes $\{x_i \in \Upsilon_t\}_{i \in I}$, that a thread $\lambda \in \Lambda$ performing a set operation $op \in \mathcal{O}$ traverses through, do not change their respective positions in Υ_t then a node x_i appears only once in the traversal path and λ terminates according to the Lemma 2.3. However, because of concurrent REMOVE operations shifting nodes to replace their successor, the respective position of nodes can change during the traversal. Also, in U it is allowed that a thread λ can get delayed infinitely. Therefore, when λ updates the values of $curr$ and preNode at lines 35 and 36 in LOCATE, it could be that $curr$ is shifted up to replace its successor by a concurrent REMOVE operation and the key which λ was querying for, was in the left-subtree of $curr$ before it got shifted. In that case λ may visit the nodes which it already would have visited. So, for turns as *right to left* when going from preNode to $curr$ to $next$ must be observed for such a change. The lines 37 to 44 take care of that and if after $curr$ has been updated to $next$, it is found that the right child of preNode has changed we update $curr$ to the new right-child of preNode and in case it is found that the thread status of the right-link of preNode changed then we take back $curr$ to preNode. This ensures that if the order-node of a removed category 3 node shifted up and the delayed operation by λ reads the left-link of the order-node after its shifting then it must go back in order to terminate at the threaded order-link of the shifted node. After that the same scenario can occur if the new order-node of the shifted node shifts up when λ was reading it. However, again the same steps will follow and the newly shifted node will not be read again. Clearly, a node is visited only $O(1)$ times (maximum twice). □

Note that, all the set operations have to perform a predecessor query by a key k to LOCATE an interval $[k_i,\ k_j]$ associated with a link s.t. $x(k_i)$ and $x(k_j)$ are two nodes in the BST. Let us define the *access-node* of an interval as the node from which the link associated with the interval, emanates from. We define *distance* of an interval from an operation op as the number of links that

op traverses from its current location ($pRoot\ \forall\ t{\leq}t_i(op)$) to read the access-node of the interval. Suppose that at $t_i(op)$ there are n nodes in the valid BST state $\Upsilon_{t_i(op)}$. Clearly, distance of any interval from op at $t_i(op)$ is $O(H(n))$. Next we prove the following lemma.

Lemma 2.20. If at $t_{ref}(op) \in [t_i(op),\ t_r(op)]$, $x(k_j)$ is a category 1 node and the distance of the interval $[k_i,\ k_j]$ associated with the order-link of $x(k_j)$ from op at $t_{ref}(op)$ is d then to access an interval $[k,\ k'] \subseteq [k_i,\ k_j]$ op traverses no more than $d + ht(x(k_j)) + |\{op'{\in}\mathcal{O} : op'\ is\ a\ concurrent\ \text{ADD}\}|$ links.

Proof. Given that at $t_{ref}(op) \in [t_i(op),\ t_r(op)]$, $x(k_j)$ is a category 1 node and the distance of the interval $[k_i,\ k_j]$ associated with the order-link of $x(k_j)$ from op at $t_{ref}(op)$ is d. We can observe that if at a $t \in [t_{ref}(op),\ t_r(op)]$, $x(k_j)$ is still a category 1 node and it is REMOVEd then the interval associated with its order-link gets subsumed by the interval associated with the order-link of the leftmost child in its right subtree or with the order-link emanating from its parent if the right subtree is null. In the former case the distance of $[k_i,\ k_j]$ from op becomes $d + ht(x(k_j))$ and in the latter it decreases by 1. Also if a node $x(k_l)$ is ADDed by an operation op' then the extra distance apart from d traversed by op to access $[k_i,\ k_l]$ or $[k_l,\ k_j]$ is no more than 1. The observations made above imply that op does not traverse more than $d{+}ht(x(k_j)){+}|\{op'{\in}\mathcal{O} : op'\ is\ a\ concurrent\ \text{ADD}\}|$ links to access a subinterval of $[k_i,\ k_j]$. □

Lemma 2.21. Length of the traversal path of a thread $\lambda{\in}\Lambda$ is bounded by $2H(n) + |\{op'{\in}\mathcal{O} : op'\ is\ a\ concurrent\ \text{ADD}\}|$.

Proof. When op traverses in the left subtree of a category 3 node x and if x gets REMOVEd, the interval associated with its order-link gets subsumed by the interval associated with the order-link of the leftmost node in the right subtree of x which is a category 1 node. On removal of a category 2 node, the interval associated with its order-link is subsumed by the interval associated with the order-link of its parent which can be a category 2 or category 3 node. Lemma 2.19 shows that a node is visited at maximum twice by a thread during a traversal. And, $(d + ht(x)) \leq 2H(n)\ \forall\ x{\in}\Upsilon$ and for any d that is distance of an interval from op its present position. Therefore, lemma 2.21 along with these observations show that the path length of a traversal in our lock-free BST is bounded by $2H(n) + |\{op'{\in}\mathcal{O} : op'\ is\ a\ concurrent\ \text{ADD}\}|$. □

Having shown that the traversal path of a thread for any operation is bounded by $O(H(n) + |\{op'{\in}\mathcal{O} : op'\ is\ a\ concurrent\ \text{ADD}\}|)$ we prove that an obstructed operation incurs only a constant number of extra steps in helping an obstructing operation.

Lemma 2.22. An obstructing operation op makes an obstructed operation op' take only a constant number of extra steps for recovery from failure in order to finish its execution.

Proof. An ADD operation does not have to hold any link and so does not obstruct an operation for itself so only a REMOVE operation can obstruct another operation. Let op be a REMOVE operation. We observe that after flagging the order-link of a node, op takes only a constant number of atomic steps to flag, mark, tag and swap links connected to the node and to its order-node in addition to setting the $preLink$ pointer of the node under REMOVE, if not obstructed by a concurrent operation. To start helping op after an unsuccessful CAS in order to complete an operation op', a thread λ reads either the $preLink$ pointer or the $backLink$ pointer of a node. It is trivial to observe that from a node pointed by $preLink$ the distance of node is no more than a single directed link. Also using lemma 2.14 a node pointed by $backLink$ is no more than two links away. That shows that a thread needs to take only a constant number of extra steps in order to perform helping. Hence the recovery from failure due to a concurrent obstructing operation needs only a constant number of links to traverse. That proves the lemma. □

Now we amortize the step complexities of the operations during an execution E. In the shared memory system U, let $t_i(op)$ be the *invocation point* of op which is the time it reads the $pRoot$, and $t_r(op)$ be the *return point* of op which is the time it reads or writes at the last link before it leaves the BST. The *point contention* $c_p(op)$ during the execution interval of op is defined as the maximum number of threads running concurrently at any point $t \in [t_i(op),\ t_r(op)]$ [1] to execute any operation. Some authors also call it *concurrent contention* [14]. In order to perform an operation $op \in \mathcal{O}$, a number of atomic steps a are taken. The amortized complexity of $op \in \mathcal{O}$ is computed as following

$$
\begin{aligned}
& Amortized\ step\ complexity\ \hat{c}(op)\ of\ op \in \mathcal{O} \\
= \ & Actual\ step\ complexity\ c(op) \\
& + number\ of\ extra\ steps\ charged\ to\ op\ on\ behalf\ of\ op' \\
& - number\ of\ extra\ steps\ charged\ on\ behalf\ of\ op\ to\ op'' \\
& where\ op, op', op'' \in \mathcal{O}\ and\ op' \neq op \neq op''
\end{aligned}
$$

Let $\mathcal{A}$ be the set of atomic steps taken by all $\lambda \in \Lambda$. We define a function $f : \mathcal{A} \mapsto \mathcal{O}$ such that if $f(a) = op$ then a is charged to the account of op and

(a) In case of no contention, all the atomic steps a representing atomic read, write and CAS taken by op is mapped to op by f.

(b) In case of contention, any failed CAS by op is mapped by f to the operation op' whose successful CAS causes the failure.

(c) If an extra read is performed during the traversal in op due to an ADDed node at $t \in [t_i(op),\ t_r(op)]$ to the set of existing nodes by a concurrent ADD operation op' then it is mapped by f to op'.

(d) Any read, write or CAS step a taken by an operation op after the first failed CAS and before retrying at the same link i.e. during helping and recovery from failure is mapped by f to the operation op' that performed the successful CAS in order to make op help it, provided op' further does not help some op'' so that op helps op'' recursively. This includes resetting of prelink, if needed.

(e) In case of recursive helping, the extra atomic steps by all the operations helping op is mapped by f to op.

With the definition of the function for accounting the steps, we prove that the upper bound of amortized complexity of operations in the following Proposition.

Proposition 2.4. *In a BST with n nodes at the start of a finite execution E, the amortized step complexity of each operation op in E is $O(H(n) + c_p(op))$, where $c_p(op)$ is the point contention during the execution of op.*

Proof. Clearly for two operations $op, op' \in \mathcal{O}$ *and* $op \neq op'$, if they are not concurrent, f can not charge any extra step to op or op' on behalf of either. Now we take the three set operations separately.

(a) A CONTAINS operation op, as it does no modification in the BST no extra step can be charged to it other than the essential steps that it would take on account of the traversal. If a node is ADDed by op' to the BST at $t \in [t_i(op),\ t_r(op)]$ and it comes in the traversal path of op then the read of this node is charged to op' by f. In Lemma 2.19 we proved that a traversal visits a node only $O(1)$ times. From the discussion in the proof of Lemma 2.19 it can be seen that during the traversal an operation may possibly visit the node preNode more than once if the node $curr$ is shifted by a concurrent REMOVE operation during its LOCATE. The extra read of preNode is charged to the concurrent REMOVE operation that shifts $curr$ and that is at most once as in a REMOVE the shift of an order-node happens only once. Because the traversal path length is at most $2H(n) + |\{op' \in \mathcal{O} : op'\ is\ a\ concurrent$ ADD$\}|$ and the number of extra reads, if any, counted in $|\{op' \in \mathcal{O} : op'\ is\ a\ concurrent$ ADD$\}|$ is charged to concurrent ADD

operations, the amortized step complexity of a CONTAINS operation during a finite execution E is $O(H(n))$ where n is the number of nodes in the BST in the initial configuration in E.

(b) An ADD operation does not perform any flag or mark of a link which can block a concurrent operation so an extra step can not be charged to an ADD by f on account of getting helped by any concurrent operation. When a node is ADDed to the BST by an operation op which was not at the invocation point of a concurrent operation op' and the new node comes in the path of the traversal of op', the read of the new node by op' is charged to op and it can be at most 1. This infers that at most $c_p(op)$ can be charged to an ADD operation op by f on account of the added node. An ADD operation's successful CAS can cause failure to the CAS of a concurrent ADD or the flagging or marking of a concurrent REMOVE and for that at most 1 CAS step can be charged by any concurrent modify operation. After the failure any concurrent modify will have to travel only one extra link which has been counted before. So on the account of failed CAS of concurrent operations f can charge at most $c_p(op)$ to an ADD operation op. Finally, an ADD operation helps a concurrent REMOVE operation if the threaded link that the new node is required to be added to is found marked or flagged. The failed CAS step and the extra steps in helping the concurrent REMOVE is charged to that. By Lemma 2.22 only a constant number of extra steps are taken in the helping. On summarizing these observations and by the upper bound of the traversal path by Lemma 2.19, the amortized complexity of an ADD operation op during a finite execution E is $O(H(n) + c_p(op))$ where n is the number of nodes in the BST in the initial configuration in E.

(c) For a REMOVE operation other than the essential steps that it takes in the traversal to track the order-link of the node to be deleted, it obstructs a concurrent modify operation and can make a concurrent traversal read extra node because of shifting of the order-node. The steps charged on account of traversal is similar to as discussed in case of CONTAINS. By Lemma 2.22 only a constant number of extra steps are needed by an obstructed concurrent modify operation. Therefore after locating the order-link of the node and its successful flagging a REMOVE operation op can be charged only $O(c_p(op))$ extra steps if it itself is not obstructed by another concurrent REMOVE forcing a recursive helping. In case of recursive helping the charges of the extra steps on behalf of obstructed operations that is taken to help another obstructing operation is passed to that by f. Summarizing these observations the amortized complexity of a REMOVE operation op during a finite execution E is $O(H(n) + c_p(op))$ where n is the number of

nodes in the BST in the initial configuration in E.

Summing up, in any finite execution E with the set of operations $\mathcal{O}$, threads perform at most $O\left(\sum_{op\in\mathcal{O}}(H(n)+c_p(op))\right)$ steps in total where n is the number of nodes in the BST in the initial configuration in E. □

It is straightforward to observe that the number of memory-words used by a BST with n nodes in our design is $5n$. That concludes the amortized analysis of our algorithm.

2.5 Implementation

2.5.1 Implementation

The algorithm for lock-free binary search tree is presented in the pseudo-code from line 1 to 363. The pseudo-code borrows its style from object-oriented programming approach of Java. We have heavily used the mechanisms of multilevel inheritance and polymorphism of classes as practised in Java. Indeed, it is straightforward to use the method of Run Time Type Identification (RTTI) available in Java to implement the presented algorithm. However, traditionally the implementation of lock-free pointer based data-structures is considered to be the best suitable for a language that allows pointer manipulation and bit-stealing from pointers, such as C/C++. It provides good performance, but that comes at the cost of increased programming effort of implementing an additional safe memory reclamation method [5]. An equivalent approach in Java can be found in way of implementation using objects like `AtomicMarkedReference` and `AtomicStampedReference` available in object library Java.util.concurrent.atomic. These are very convenient to use for the purpose of bit-stealing from pointers. Needless to mention that availability of safe garbage collector in Java reduces the programming complexity significantly.

To implement the algorithm using pointers for links, we can use bit-stealing to maintain the states of links. To maintain 10 states, we need to steal four bits from a pointer. If we represent a pointer as a packet of a reference and four boolean variables overlapping four bits that is not used to represent address of a variable as $\{\&ref, b_1, b_2, t, f\}$, where b_1, b_2, t and f are bits, then state of a link can be expressed as following:

- Setting the bit f indicates a finalized link.
- Setting the bit t indicates a threaded link.

- Setting the bits b_1 and b_2 in combination can indicate three states of flagged, marked and tagged and to retrieve the state they must be retrieved together.

The modern C/C++ compilers by default allocate memory aligned at 64 bit boundary that leaves three least significant bits unused. We will need to steal one more bit when using a 64-bit machine and default alignment of memory allocated by popular compilers like GCC(G++). When using the Java library object `AtomicStampedReference`, the state identifiers can be realized with decimal conversion of the four bits combination as a binary number and using stamp for that. Doing logical operations on the stamp simulates the bit stealing operations.

We assume that in the implementation setup a lock-free safe mechanism to reclaim the removed nodes is used. In Java, the garbage collector exists for this purpose. However for C/C++ implementation, where no language provided garbage collectors are available some widely used methods are Hazard-pointer based safe memory reclamation [19] and Reference counting [8]. The basic idea in all these methods is that they keep track of all the objects local to each thread in the system and the shared object is reclaimed only after having ensured that no object local to any thread holds a reference to it. Other than that it is also assumed that for all practical cases a memory allocated for a new node always has a new address. In most of the lock-free data-structures these assumptions are sufficient for a correct implementation. However, this algorithm can face a classical ABA problem as described next.

2.5.2 ABA problem

Consider the case of a delayed thread in the presented algorithm as shown in the Fig. 2.5. The thread T that executes REMOVE(5), performs the steps of (a) flagging order-link, (b) tagging parent-link of the order-node, (c) tagging left-link of the order-node (d) setting preLink, (e) marking right-link and (f) marking left-link successfully, Fig. 2.5 (I). Then it modifies the backLink of the left-child of the order-node, swaps the incoming parent-link of the order-node and also changed the backLink of the left-child of the node $x(5)$ i.e. it executes up to line 232 in the function HELPMARKLEFT in the execution of REMOVE(5), Fig. 2.5 (II). Just after that it gets delayed and some helping thread comes and helps to complete REMOVE(5), Fig. 2.5 (III). Because there is a safe memory reclamation scheme being used, the node $x(5)$ is not reclaimed as its reference is held in the thread T, but for the sake of clarity we have not shown the removed node in Fig. 2.5 (III). After that $x(6)$ is removed by another thread and T keeps on sleeping, Fig. 2.5 (IV). After that $x(2)$ is removed by another thread and

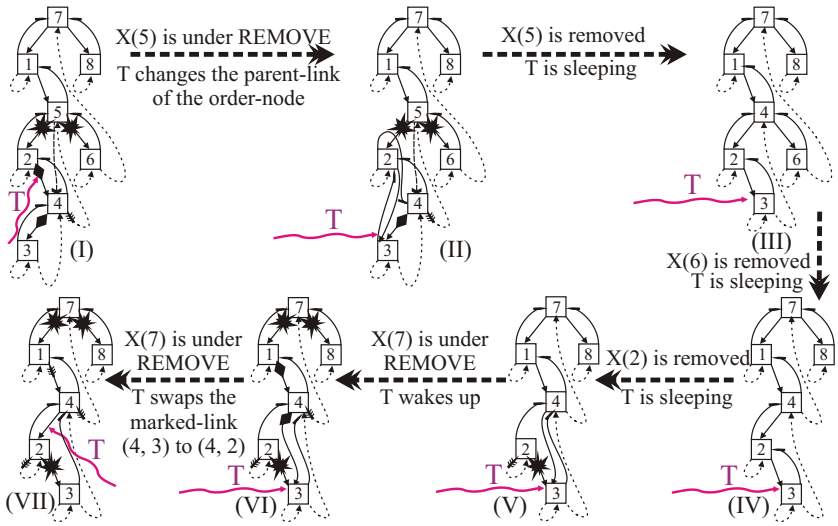
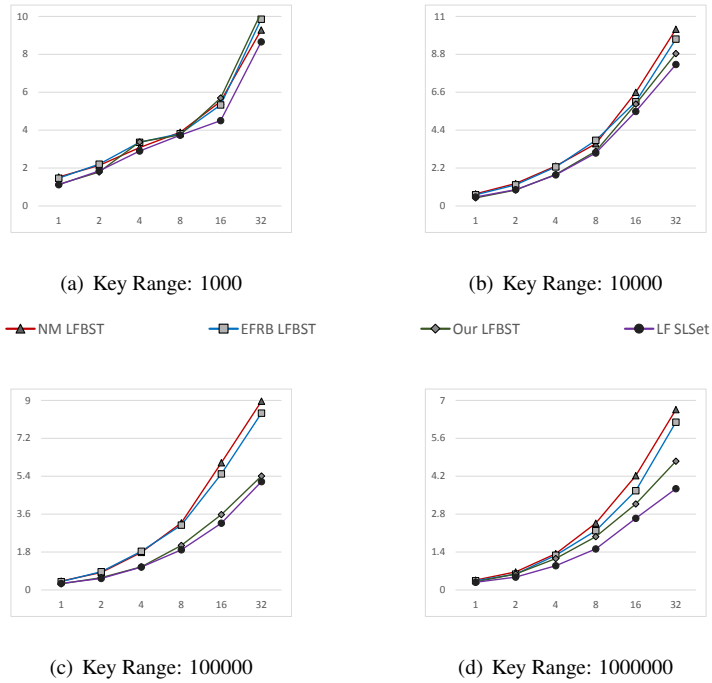


Figure 2.5: *The ABA problem in the algorithm*

T still keeps on sleeping, Fig. 2.5 (V). Note that references to both $x(6)$ and $x(2)$ is held by T so they are not freed, we have not showed these nodes for the same reason as mentioned before. After that $x(7)$ starts being removed by another thread and that completes all the steps up to the line 232 in the function HELPMARKLEFT successfully, that T had done in the course of REMOVE(5). At this very moment T wakes up, Fig. 2.5 (VI). Because $x(2)$ is still not freed and the left child-link of $x(4)$, being the order-node of $x(7)$ now is tagged and pointing to $x(3)$, i.e. in exactly the same state when T went to sleep, T will swap this child-link to connect to $x(2)$ and that formes a malformed structure, Fig. 2.5 (VII). Now at this point a CONTAINS(2) can return true and that is wrong. This is a classical case of the well known ABA problem.

Thus, if we implement this algorithm using pointers as links, we would need to use a version counter in each child pointer in order to prevent the described ABA problem. Whenever a pointer is swapped in order to finally clean a node undergoing REMOVE, the version counter needs to be incremented. To do that we would need to steal few more bits from a pointer used as a link. With the counter in a child pointer the pointer swapping steps will include incrementing the counter as well. Now with that, if a thread after waking up tries to swap the pointer the CAS step will fail.

In case of implementing this algorithm with RTTI, when we shift the order-



(a) Key Range: 1000

(b) Key Range: 10000

(c) Key Range: 100000

(d) Key Range: 1000000

Figure 2.6: *The performance graph in read-heavy case (*ADD-REM-CON *: 2%-2%-96%).*

node of a category 3 node to replace it, a new instance of the **UtLink** class is allocated for the left-child link of the shifted node. Also other links are appropriately given a new instance of a subclass of **Link**. Clearly, the CAS will not succeed more than once in such an implementation. So, the algorithm is not affected by any ABA problem as long as the implementation environment guarantees to allocate a new variable with address different from that of any variable whose reference is held by a thread that is active or idle and has not completed its operation.

2.5.3 Experiments

The experimental evaluations in [26] show that the real-time-type-information (RTTI) based implementation in Java outperforms the implementation of lock-

free single linked-list based on `AtomicMarkedReference`. We also implemented our algorithm using method of RTTI in Java.

The following algorithms are compared in our experiments:

(a) *Our LFBST*: The presented algorithm.

(b) *NM LFBST*: The lock-free external BST of [20].

(c) *EFRB LFBST*: The lock-free external BST of [11][1].

(d) *LF SLSet*: `ConcurrentSkipListSet` available in java concurrency library.

The `ConcurrentSkipListSet` available in java class library is a highly optimized non-linear lock-free set implementation based on skip-lists and is widely used in real life programs. We implemented the algorithm of [20] using the same method of RTTI in Java.

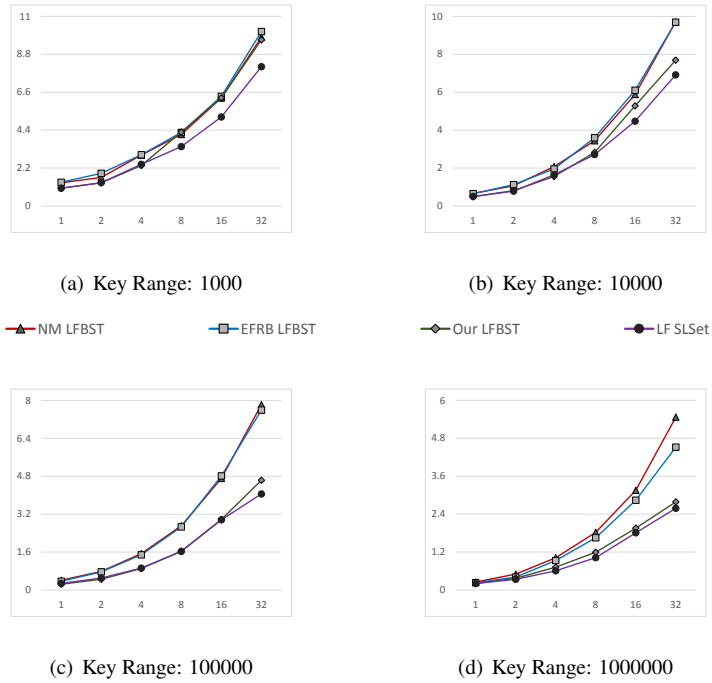
Our experiments were performed on a machine with 2 Intel Xeon E5-2650 processors with 16 hardware threads (8 physical cores with hyper-threading enabled) per processor i.e. 32 hardware threads in total running at 2.0 GHz. The machine has 64 GB of RAM and runs over x86_64 Ubuntu Linux 13.04 (Linux Kernel version: 3.8.0-30-generic) with OpenJDK 64-bit JVM version 1.7.0_51 (with 1 GB initial heap size and 16 GB MaxHeapSize). All the implementations were compiled using `javac` version 1.7.0_51 without any flag.

To assimilate the variation in the contention due to various factors such as range of keys, size of BST and proportion of modify operations we compared the performance of all the implementations varying the following parameters:

- $|\{key \in K\}| \in \{1000, 10000, 100000, 1000000\}$.
- The distribution of (ADD, REMOVE, CONTAINS) $\in \{(02, 02, 96), (10, 10, 80), (25, 25, 50), (50, 50, 00)\}$.
- The number of threads $\in \{1, 2, 4, 8, 16, 32\}$.

All the keys are of type `int` in our experiments. We performed 10 repetitions of 5 seconds runs for each combination of the above parameters and recorded the throughput as total number of operations per milliseconds. We discarded the first 3 observations on account of VM warm-up. When computing the average over the remaining 7 trials, we discarded the maximum and minimum to remove the outliers. The counting of operations start after *pre-populating* the BST with 1000 nodes.

[1] We obtained the code from the page http://www.cs.utoronto.ca/~tabrown/ and modified it to remove the value field from the node class for a fair comparison.



(a) Key Range: 1000

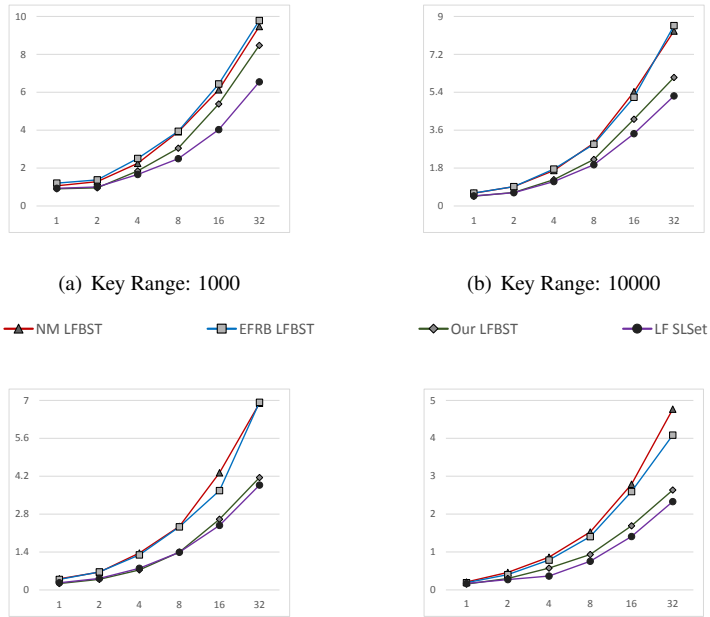
(b) Key Range: 10000

(c) Key Range: 100000

(d) Key Range: 1000000

Figure 2.7: *The performance graph in moderate write dominated case (*ADD-REMCON *: 10%-10%-80%).*

Performance of the implementations in terms of throughput ($\frac{1000 \times \#Ops}{ms}$) vs. $\#threads$ with varying the range of the keys and the distribution of operations in terms of ADD%-REMOVE%-CONTAINS% are plotted in figures 2.6(a) to 2.9(d). The variations is key range vary the contention level in the implementation. The empirical observations show that the presented lock-free BST algorithm shows good scalability with increasing number of threads. Both the internal as well as the external lock-free BST algorithm significantly outperform the java `ConcurrentSkipListSet` as soon as there is concurrency in the system. The external lock-free BSTs perform better than our algorithm experimentally because of the reason that in our algorithm a REMOVE operation takes up to 16 atomic `CAS` executions (17 if $\cdot finalize()$ is used to save some helping steps) even without contention, whereas the *NM LFBST* and *EFRB LFBST* take respectively three and four such executions.



(a) Key Range: 1000

(b) Key Range: 10000

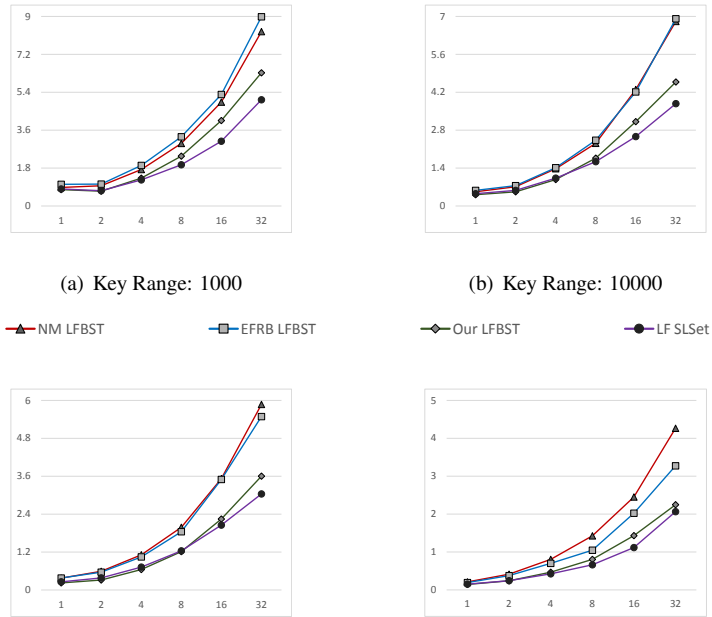
(c) Key Range: 100000

(d) Key Range: 1000000

Figure 2.8: *The performance graph in mixed case (*ADD-REM-CON *: 25%-25%-50%).*

2.6 Conclusion and Future Work

In this paper we proposed a novel algorithm for the implementation of a lock-free internal BST. Using amortized analysis we proved that all the operations in our implementation run in time $O(H(n) + c)$, where $H(n)$ is the height of the BST with n nodes and c is the measure of point contention. We solved the existing problem of "retry from scratch" for modify operations after failure caused by a concurrent modify operation, which resulted into an amortized step complexity of $O(cH(n))$. This improvement takes care of an algorithmic design issue for which the time complexity of modify operations increases dramatically with the increase in the contention and the size of the data-structure. This is an important improvement over the existing algorithms. Our algorithm also comes with improved disjoint-access-parallelism compared to similar lock-free BST algorithms. We proved the linearizability and lock-freedom of the pro-



(a) Key Range: 1000

(b) Key Range: 10000

(c) Key Range: 100000

(d) Key Range: 1000000

Figure 2.9: *The performance graph in write-heavy case (*ADD-REM-CON *: 50%-50%-0%).*

posed algorithm. The experiments show that our algorithm is scalable with the number of threads.

Acknowledgments

We would like to thank Dr. Neeraj Mittal (Associate Professor, Department of Computer Science, The University of Texas at Dallas, USA) for many valuable comments on a previous draft of this paper.

Bibliography

[1] Y. Afek, G. Stupp, and D. Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.

[2] G. Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the 5th ACM SPAA*, pages 261–270, 1993.

[3] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM PPoPP*, pages 257–268, 2010.

[4] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM PPoPP*, pages 329–342, 2014.

[5] 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 27th IPDPS*, pages 1309–1320, 2013.

[6] B. Chatterjee, N. Nguyen, and P. Tsigas. Efficient lock-free binary search trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 322–331, 2014.

[7] T. Crain, V. Gramoli, and M. Raynal. A speculation- friendly binary search tree. In *Proceedings of the 17th ACM PPoPP*, pages 161–170, 2012.

[8] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

[9] D. Drachsler, M. Vechev, and E. Yahav. Practical concurrent binary search trees via logical ordering. In *Proceedings of the 19th ACM PPoPP*, pages 343–356, 2014.

[10] F. Ellen, P. Fatourou, J. Helga, and E. Rupert. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 2013 ACM PODC*, 2014.

[11] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM PODC*, pages 131–140, 2010.

[12] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the 23rd ACM PODC*, pages 50–59, 2004.

[13] K. Fraser. *Practical lock-freedom*. PhD thesis, Cambridge University, Computer Laboratory, 2004.

[14] M. Herlihy, V. Luchangco, and M. Moir. Space- and time-adaptive nonblocking algorithms. *Electronic Notes in Theoretical Computer Science*, 78(0):260 – 280, 2003. CATS'03, Computing: the Australasian Theory Symposium.

[15] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[16] S. V. Howley and J. Jones. A non-blocking internal binary search tree. In *Proceedinbgs of the 24th Annual ACM SPAA*, pages 161–171, 2012.

[17] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th ACM PODC*, pages 151–160, 1994.

[18] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th ACM SPAA*, pages 73–82, 2002.

[19] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6):491–504, 2004.

[20] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM PPoPP*, pages 317–328, 2014.

[21] R. Oshman and N. Shavit. The skiptrie: low-depth concurrent search without rebalancing. In *Proceedings of the 2013 ACM PODC*, pages 23–32, 2013.

[22] A. J. Perlis and C. Thornton. Symbol manipulation by threaded lists. *Communications of the ACM*, 3(4):195–204, 1960.

[23] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, 2005.

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

[25] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank. Wait-free linked-lists. In R. Baldoni, P. Flocchini, and R. Binoy, editors, *Principles of Distributed Systems*, volume 7702 of *LNCS*, pages 330–344. Springer Berlin Heidelberg, 2012.

[26] K. Zhang, Y. Zhao, Y. Yang, Y. Liu, and M. F. Spear. Practical non-blocking unordered lists. In *Proceedings of the 27th DISC*, pages 239–253, 2013.

PAPER II

Extended version of

Daniel Cederman, **Bapi Chatterjee** and Philippas Tsigas

Understanding the Performance of Concurrent Data Structures on Graphics Processors

In the Proceedings of the 18th International Conference on Parallel Processing, Euro-Par 2012, Lecture Notes in Computer Science Vol.: 7484, pages 883-894, Springer-Verlag 2012.

3
PAPER II

Abstract

In this paper we revisit the design of concurrent data structures – specifically queues – and examine their performance portability with regard to the move from conventional CPUs to graphics processors. We have looked at both lock-based and lock-free algorithms and have, for comparison, implemented and optimized the same algorithms on both graphics processors and multi-core CPUs. Particular interest has been paid to study the difference between the old Tesla and the new Fermi and Kepler architectures in this context. We provide a comprehensive evaluation and analysis of our implementations on all examined platforms. Our results indicate that the queues are in general performance portable, but that platform specific optimizations are possible to increase performance. The Fermi and Kepler GPUs, with optimized atomic operations, are observed to provide excellent scalability for both lock-based and lock-free queues.

3.1 Introduction

While multi-core CPUs have been available for years, the use of GPUs as efficient programmable processing units is more recent. The advent of CUDA [11]

and OpenCL [16] made general purpose programming on graphics processors more accessible to the non-graphics programmers. But still the problem of efficient algorithmic design and implementation of generic concurrent data structures for GPUs remains as challenging as ever.

Much research has been done in the area of concurrent data structures. There are efficient concurrent implementations of a variety of common data structures, such as stacks [17], queues [3, 4, 7, 9, 13, 18] and skip-lists [15]. For a good overview of several concurrent data structures we refer to the chapter by Cederman et al. [1].

But while the aforementioned algorithms have all been implemented and evaluated on many different multi-core architectures, very little work has been done to evaluate them on graphics processors. Data structures targeting graphics applications have been implemented on GPUs, such as the kd-tree [20] and octree [19]. A C++ and Cg based template library [8] has been provided for random access data structures for GPUs. Load balancing schemes on GPUs [2] using different data structures have been designed. A set of blocking synchronization primitives for GPUs [14] has been presented that could aid in the development or porting of data structures.

With the introduction of atomic primitives on graphics processors, we hypothesize that many of the existing concurrent data structures for multi-core CPUs could be transferred to graphics processors, perhaps without much change in the design. To evaluate how performance portable the designs of already existing common data structure algorithms are, we have, for this paper, implemented a set of concurrent FIFO queues with different synchronization mechanisms on both graphics processors and on multi-core CPUs. We have performed experiments comparing and analyzing the performance and cache behavior of the algorithms. We have specifically looked at how the performance changes by the move from NVIDIA's Tesla architecture to the newer Fermi [10] and Kepler (GK104) [12] architectures.

The paper is organized as follows. In section 3.2, we introduce the concurrent data structures and describe the distinguishing features of the algorithms considered. Section 3.3 presents a brief description of the CUDA programming model and different GPU architectures. In section 3.4, we present the experimental setup. A detailed performance analysis is presented in section 3.5. Section 3.6 concludes the paper.

3.2 Concurrent Data Structures

Depending on the synchronization mechanism, we broadly classify concurrent data structures into two categories, namely *blocking* and *non-blocking*. In blocking synchronization, no progress guarantees are made. For non-blocking synchronization, there are a number of different types of progress guarantees that can be assured. The two most important ones are known as *wait-free* and *lock-free*. Wait-free synchronization ensures that all the non-faulty processes eventually succeed in finite number of processing steps. Lock-free synchronization guarantees that at least one of the non-faulty processes out of the contending set will succeed in a finite number of processing steps. In practice, wait-free synchronization is usually more expensive and is mostly used in real-time settings with high demands on predictability, while lock-free synchronization targets high-performance computing.

Lock-free algorithms for multiple threads require the use of atomic primitives, such as Compare-And-Swap (CAS). CAS can conditionally set the value of a memory word, in one atomic step, if at the time, it holds a value specified as a parameter to the operation. It is a powerful synchronization primitive, but is unfortunately also expensive compared to normal read and write operations.

In this paper we have looked at different types of queues to evaluate their performance portability when moved from the CPU domain to the GPU domain. The queue data structures that we have chosen to implement are representative of several different design choices, such as being array-based or linked-list-based, cache-aware or not, lock-free or blocking. We have divided them up into two main categories, Single-Producer Single-Consumer (SPSC) and Multiple-Producer Multiple-Consumer (MPMC).

3.2.1 SPSC Queues

In '83, **Lamport** presented a lock-free array-based concurrent queue for the SPSC case [6]. For this case, synchronization can be achieved using only atomic read and write operations on shared head and tail pointers. No CAS operations are necessary. Having shared pointers cause a lot of cache thrashing however, as both the producer and consumer need to access the same variables in every operation.

The **FastForward** algorithm lowered the amount of cache thrashing by keeping the head and tail variables private to the consumer and producer, respectively [3]. The synchronization was instead performed using a special empty element that was inserted into the queue when an element was dequeued. The producer would then only insert elements when the next slot in the array con-

tained such an element. Cache thrashing does however still occur when the producer catches up with the consumer. To lower this problem it was suggested to use a small delay to keep the threads apart. The settings used for the delay function are however so application dependant that we decided not to use it in our experiments.

The **BatchQueue** algorithm divides the array into two batches [13]. When the producer is writing to one batch, the consumer can read from the other. This removes much of the cache thrashing and also lowers the frequency at which the producer and consumer need to synchronize. The major disadvantage of this design is that a batch must be full before it can be read, leading to large latencies if elements are not enqueued fast enough. A suggested solution to this problem was to at regular intervals insert null elements into the queue. We deemed this as a poor solution and it is not used in the experiments. To take better advantage of the graphics hardware, we have also implemented a version of the BatchQueue where we copy the entire batch to the local shared memory, before reading individual elements from it. We call this version **Buffered BatchQueue**.

The **MCRingBuffer** algorithm is similar to the BatchQueue, but instead of having just two batches, it can handle an arbitrary number of batches. This can be used to find a balance between the latency caused by waiting for the other threads and the latency caused by synchronization. As for the BatchQueue we provide a version that copies the batches to the local shared memory. We call this version **Buffered MCRingBuffer**.

3.2.2 MPMC Queues

For the MPMC case we used the lock-free queue by Michael and Scott, henceforth called the **MS-Queue** [9]. It is based on a linked-list and adds items to the queue by using CAS to swap in a pointer at the tail node. The tail pointer is then moved to point to the new element, with the use of a CAS operation. This second step can be performed by the thread invoking the operation, or by another thread that needs to help the original thread to finish before it can continue. This helping behavior is an important part of what makes the queue lock-free, as a thread never has to wait for another thread to finish.

We also used the lock-free queue by Tsigas and Zhang, henceforth called the **TZ-Queue**, which is an array-based queue [18]. Elements are here inserted into the array using CAS. The head and tail pointers are also moved using CAS, but it is done lazily, after every x:th element instead of after every element. In the experiments we got the best performance doing it every second operation.

To compare lock-free synchronization with blocking, we used the **lock-based** queue by Michael and Scott, which stores elements in a linked-list [9].

We used both the standard version, with separate locks for the enqueue and dequeue operation, and a simpler version with a common lock for both operations. For locks we used a basic spinlock, which spins on a variable protecting a critical section, until it can acquire it using CAS. As CAS operations are expensive, we also implemented a lock that does not use CAS, the bakery-lock by Lamport [5].

3.3 GPU Architectures

Graphics processors are massively parallel shared memory architectures excellently suitable for data parallel programs. A GPU has a number of stream multiprocessors (SMs), each having many cores. The SMs have registers and a local fast shared memory available for access to threads and thread blocks (group of threads) respectively, executing on them. The global memory, the main graphics memory, is shared by all the thread blocks and the access is relatively slow compared to that of the local shared memory.

In this work we have used CUDA for all GPU implementations. CUDA is a mature programming environment for programming on GPUs. In CUDA threads are grouped into blocks where all threads in a specific block execute on the same SM. Threads in a block are in turn grouped into so called warps of 32 consecutive threads. These warps are then scheduled by the hardware scheduler. Exactly how the scheduler schedules warps is unspecified. This is problematic when using locks, as there is a potential for deadlocks if the scheduler is unfair. For lock-free algorithms this is not an issue, as they are guaranteed to make progress regardless of the scheduler.

The different generations of CUDA programmable GPUs are categorized in compute capabilities (CC) and are identified more popularly by their architecture's codename. CC 1.x are Tesla, 2.x are Fermi and 3.x are Kepler. The architectural features depend on the compute capability of the GPU. In particular the availability of atomic functions has been varying with the compute capabilities. In CC 1.0 there were no atomic operations available, from CC 1.1 onwards there are atomic operations available on the global memory and from CC 1.2 also for the shared memory. An important addition to the GPUs in the Fermi and Kepler architectures is the availability of a unified L2 cache and a configurable L1 cache. The performance of the atomic operations significantly increased in Fermi, with the atomic unit working on the L2 cache, instead of on the global memory [14]. The bandwidth of L2 cache increased in Kepler so that it is now 73% faster than that in Fermi [12]. The speed of atomic operations has also been significantly increased in Kepler as compared to Fermi.

3.4 Experimental Setup

The experiments were performed on four different types of graphics processors, with different memory clock rates, multiprocessor counts and compute capabilities. To explore the difference in performance between CPUs and GPUs, the same experiments were also performed on a conventional multi-core system, a 12-core Intel system (24 cores with HyperThreading). See Table 3.1 for an overview of the platforms used.

Name	Clock speed	Memory clock rate	Cores	Cache	Architecture (CC)
GeForce 8800 GT	1.6 GHz	1.0 GHz	14	0	1.1 (Tesla)
GeForce GTX 280	1.3 GHz	1.1 GHz	30	0	1.3 (Tesla)
Tesla C2050	1.2 GHz	1.5 GHz	14	786 kB	2.0 (Fermi)
GeForce GTX 680	1.1 GHz	3.0 GHz	8	512 kB	3.0 (Kepler)
Intel E5645 (2x)	2.4 GHz	0.7 GHz	24	12 MB	

Table 3.1: *Platforms used in experiments. Counting multiprocessors as cores in GPU.*

In the experiments we only consider communication between thread blocks, not between individual threads in a thread block.

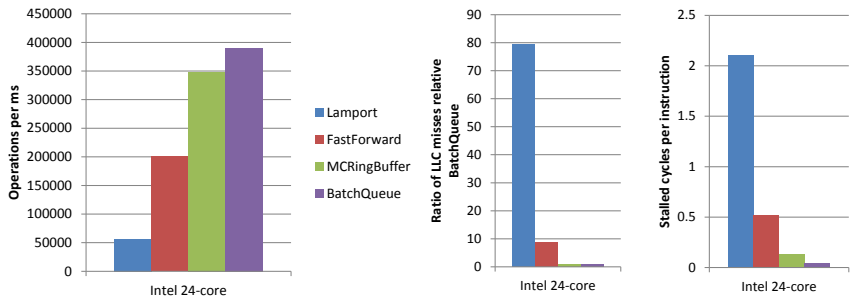
For the SPSC experiments, a thread from one thread block was assigned the role of the producer and another thread from a second block the role of the consumer. The performance was measured by counting the number of successful enqueue/dequeue operations per ms that could be achieved when communicating a set of integers from the producer to the consumer. Enqueue operations on full queues or dequeue operations on empty queues were not counted. Local variables, variables that are only accessed by either the consumer or the producer, are placed in the shared memory to remove unnecessary communication with the global memory. For buffered queues, 32 threads were used for memory transfer between global and shared memory to take advantage of the hardware's coalescing of memory accesses. All array-based queues had a maximum length of 4096 elements. The MCRingBuffer used a batch size of 128 whereas the BatchQueue by design has batches of size as of half the queue size, in this case 2048. For the CPU experiments care was taken to place the consumer and producer on different sockets, to emphasize the penalty taken by using an inefficient memory access strategy.

For the MPMC experiments a varying number of thread blocks were used, from 2 up to 60. Each thread block performed 25% enqueue operations and 75% dequeue operations randomly, using a uniform distribution. Two scenarios were used, one with high contention, where operations were performed one

after another, and one with low contention, in which a small amount of work was performed between the operations. The performance was measured in the number of successful operations per ms in total.

3.5 Performance Analysis

3.5.1 SPSC Queues



(a) Comparison of SPSC queues. (b) Cache profile.

Figure 3.1: *Comparison of SPSC queues on the CPU based system.*

Figure 3.1(a) depicts the result from the experiments on the CPU system. It is clear from the figure that even the small difference in access pattern between the Lamport and the FastForward algorithms has a significant impact on the performance. The number of operations per ms differ by a factor of four between the two algorithms. The cache access profile in Figure 3.1(b) shows that the number of cache misses goes down dramatically when the head and tail variables are no longer shared between processes. It goes down even further when the producer and the consumer are forced to work on different memory locations. The figure also shows that the number of stalled cycles per instructions matches the cache misses relatively well. The reason for the performance difference between the BatchQueue and the MCRingBuffer, which both have a similar number of cache misses, lies in the difference between the size of the batches. This causes more frequent reads and writes of shared variables compared to the BatchQueue. It was observed that increasing the batch size lowers the synchronization overhead and the number of stalled cycles and improves the performance of the MCRingBuffer and brings it close to that of the BatchQueue.

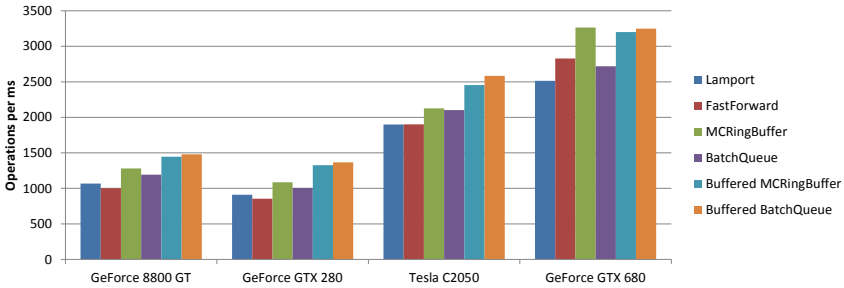
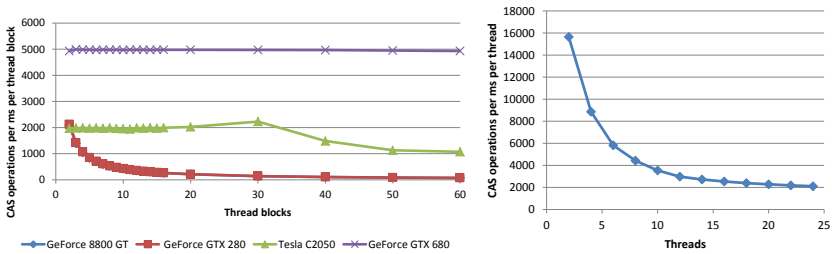


Figure 3.2: *Comparison of SPSC queues on four different GPUs.*

Figure 3.2 shows the results for the same experiment performed on the graphics processors. On the Tesla processors there are no cache memories available, which removes the problem of cache thrashing and causes the Lamport and FastForward algorithms to give similar results. In contrast to the CPU implementations, here the MCRingBuffer is actually faster than the BatchQueue. This is due to the fact that the BatchQueue enqueuer is faster than the dequeuer and has to wait for a longer time for the larger batches to be processed. The smaller batch size in MCRingBuffer thus has an advantage here. The two buffered versions lower the overhead, as for most operations the data will be available locally from the shared memory. It is only at the end of a batch that the shared variables and the elements stored in the queue need to be accessed. This access is done using coalesced reads and writes, which speeds up the operation. When the queues are buffered, the BatchQueue becomes faster than the MCRingBuffer. Thus the overhead of the more frequent batch copies became more dominant. The performance on the Fermi and Kepler graphics processor is significantly better compared to the other processors, benefiting from the faster memory clock rate and the cache memory. The speed of the L2 cache is however not enough to make the unbuffered queues comparable with the buffered ones on the Fermi processor. On the Kepler processor, on the other hand, with its faster cache and higher memory clock rate, the unbuffered MCRingbuffer performs similarly to the buffered queues. The SPSC queues that we have examined thus need to be rewritten to achieve maximum performance on most GPUs. This might however change with the proliferation of the Kepler architecture.



(a) CAS behavior on the GPUs.

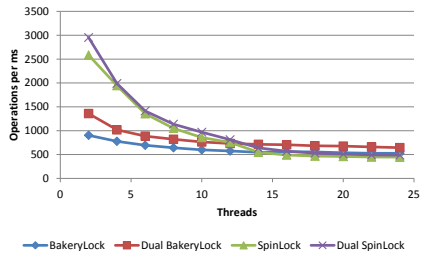
(b) CAS behavior on the CPU.

Figure 3.3: *Visualization of the CAS behavior on the GPUs and the CPU.*

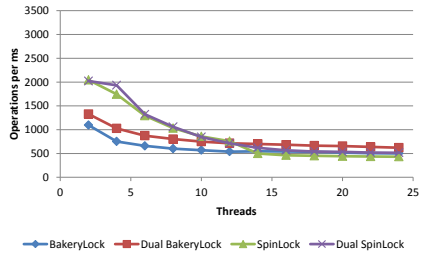
3.5.2 MPMC Queues

All MPMC queue algorithms, except the ones that used the bakery-lock, make use of the CAS primitive. To visualize the behavior of the CAS primitive we measured the number of CAS operations that could be performed per thread block per ms for a varying number of thread blocks. The result is available in Figure 3.3. We see in Figure 3.3(a) that when the contention increases for the Tesla processors the number of CAS operations per ms drops quickly. However, it is observed that the CAS operations scale well on the Fermi, for up to 40 thread blocks, at high speed. The increased performance of the atomic primitives was one of the major improvements done when creating the Fermi architecture. The atomic operations are now handled at the L2 cache level and no longer need to access the global memory [14]. The Kepler processor has twice the memory clock rate of the Fermi processor and we can see that the CAS operations scales perfect despite increased contention. Figure 3.3(b) shows that on the conventional system the performance is quite high when few threads perform CAS operations, but the performance drops rapidly as the contention increases.

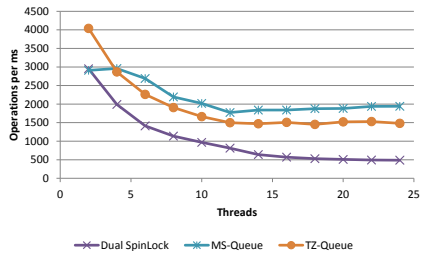
Figure 3.4 shows the performance of the MPMC queues on the CPU-based system. Looking first at the topmost graphs, which shows the result using just lock-based queues, we see that for a low number of threads the dual spinlock based queue clearly outperforms the bakery lock based queues. The bakery lock does not use any expensive CAS operation, but the overhead of the algorithm is still too high, until the number of threads goes above the number of cores and starts to use hyperthreading. The difference between dual and single spinlock is insignificant, however between the dual and the single bakery lock there is a noticeable difference.



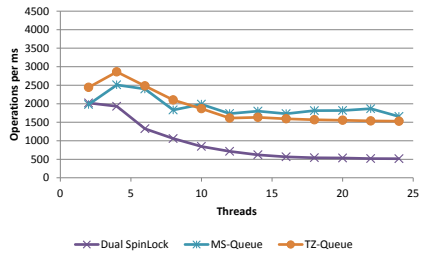
(a) Lock-based queues (*High* contention).



(b) Lock-based queues (*Low* contention).



(c) Best lock-based and lock-free queues (*High* contention).

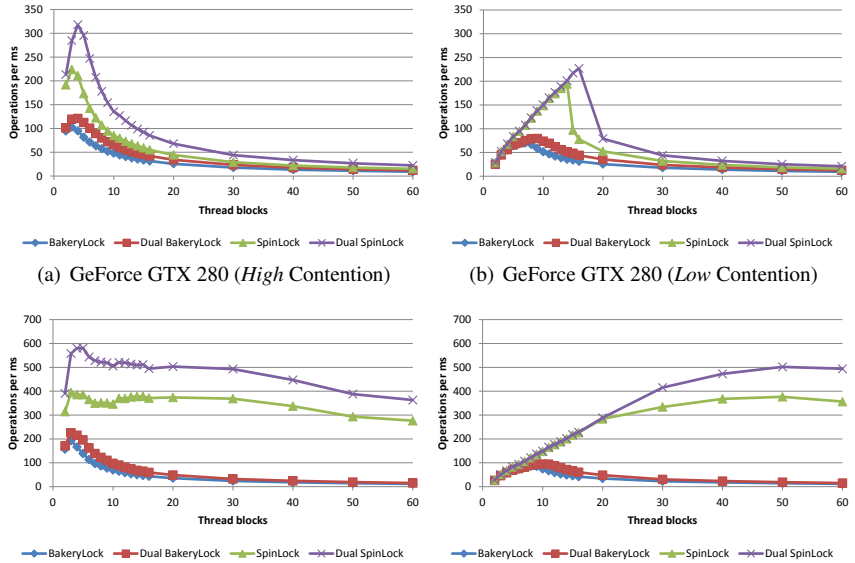


(d) Best lock-based and lock-free queues (*Low* contention).

Figure 3.4: *Comparison of MPMC queues on the Intel 24-core system under* high *and* low *contention scenarios.*

The lower two graphs show the comparison results for the two lock-free queues together with the best lock-based one, the dual spinlock queue. The lock-free queues clearly outperform the lock-based one for all number of threads and for both contention levels. The array-based TZ-queue exhibits better results for the lower range of threads, but it is quickly overtaken by the linked-list based MS-queue. When hyperthreading kicks in, the performance does not drop any more for any of the queues.

The measurements taken for the lock-based queues on the Fermi and one of the Tesla graphics processors are shown in Figure 3.5. Just as in the CPU experiments the dual spinlock queue excels among the lock-based queues. There is however a much clearer performance difference between the dual and single spinlock queues in all graphs, although not for the low contention cases when using few thread blocks. The peak in the result in Figure 3.5(a) is due to the overhead of the benchmark and the non-atomic parts of the queue. When contention is lowered, as in Figure 3.5(b), the peak moves to the right. After the peak the cost of the atomic operations become dominant, and the performance



(a) GeForce GTX 280 (*High* Contention)

(b) GeForce GTX 280 (*Low* Contention)

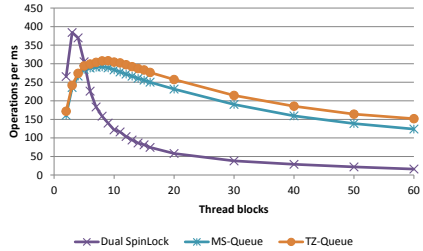
(c) Tesla C2050 (*High* Contention)

(d) Tesla C2050 (*Low* Contention)

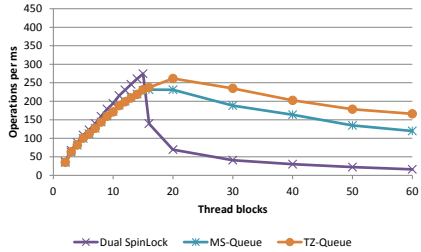
Figure 3.5: *Comparison of lock-based MPMC queues on two GPUs under* high *and* low *contention scenarios.*

drops. For the Fermi-processor, in Figure 3.5(c), the performance for the spin-lock based queues is significantly higher, while at the same time scaling much better. As we could see in Figure 3.3(a), this is due to the much improved atomic operations of the Fermi-architecture.

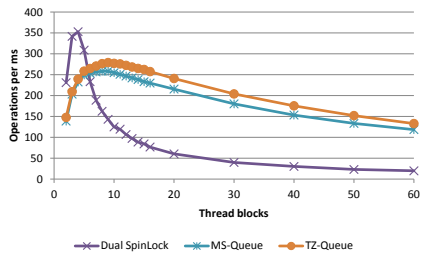
Comparing the dual spinlock queue with the lock-free queues, in Figure 3.6 we see that the lock-free queues scale much better than the lock-based one and provides the best performance when the thread block count is high. The spin-lock queue does however achieve a better result on all graphics processors for a low number of thread blocks. As the contention is lowered, it remains useful for a higher number of threads. The array-based TZ-queue outperforms the linked-list based MS-queue on both the Tesla processors, but falls short on the Fermi and Kepler processors, Figure 3.6(e). When contention is lowered on the Fermi-processor, Figure 3.6(f), there is no longer any difference between the lock-based and the lock-free queues.



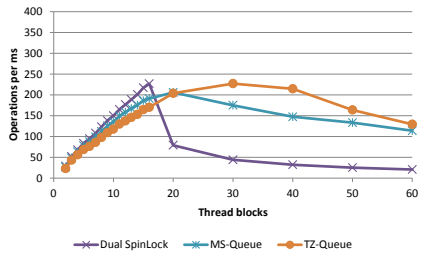
(a) GeForce 8800 GT (*High* Contention)



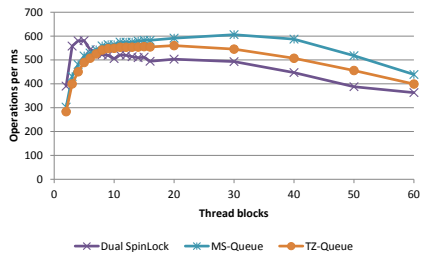
(b) GeForce 8800 GT (*Low* Contention)



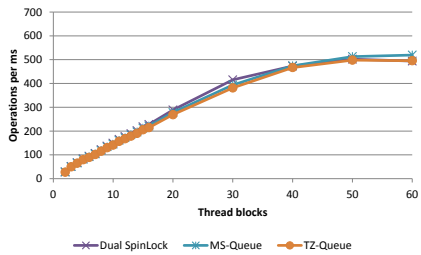
(c) GeForce GTX 280 (*High* Contention)



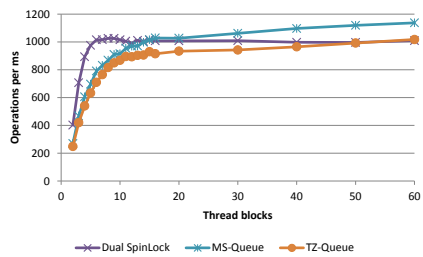
(d) GeForce GTX 280 (*Low* Contention)



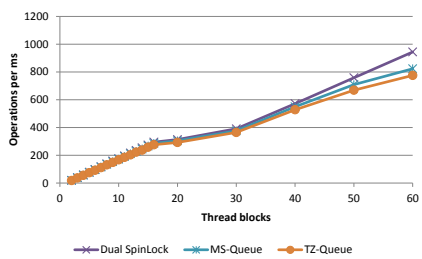
(e) Tesla C2050 (*High* Contention)



(f) Tesla C2050 (*Low* Contention)



(g) GeForce GTX 680 (*High* Contention)



(h) GeForce GTX 680 (*Low* Contention)

Figure 3.6: *Comparison of the best lock-based and lock-free MPMC queues on four GPUs under* high *and* low *contention scenarios.*

3.6 Conclusion and Future work

In this paper we have examined the performance portability of common SPSC and MPMC queues. From our experiments on the SPSC queues we found that the best performing queues on the CPU were also the ones that performed well on the GPUs. It was however clear that the cache on the Fermi-architecture was not enough to remove the benefit of redesigning the algorithms to take advantage of the local shared memory. For the MPMC queue experiments we saw similar results in scalability for the GPU-versions on the Tesla processors as we did for the CPU-version. On the Fermi processor the result was surprising however. The scalability was close to perfect and for low contention there was no difference between the lock-based and the lock-free queues. The Fermi architecture has significantly improved the performance of atomic operations and this is an indication that new algorithmic designs should be considered to more properly take advantage of this new behavior. The Kepler architecture has continued in this direction and now provides atomic operations with performance competitive to that of conventional CPUs.

We will continue this work by studying the behavior of other concurrent data structures with higher potential to scale than queues, such as dictionaries and trees. Most queue data structures suffer from the fact that only two operations can succeed concurrently in the best case, whereas for a dictionary there are no such limitations.

Bibliography

[1] D. Cederman, A. Gidenstam, P. Ha, H. Sundell, M. Papatriantafilou, and P. Tsigas. Lock-free Concurrent Data Structures. In Sabri Pllana et al., editor, *Programming Multi-core and Many-core Computing Systems*. John Wiley & Sons, 2012.

[2] D. Cederman and P. Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd symposium on Graphics hardware*, GH '08, pages 57–64. Eurographics Association, 2008.

[3] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 43–52. ACM, 2008.

[4] A. Gidenstam, H. Sundell, and P. Tsigas. Cache-Aware Lock-Free Queues for Multiple Producers/Consumers and Weak Memory Consistency. In *Proceedings of the 14th International Conference on Principles of Distributed Systems*, pages 302–317, 2010.

[5] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17:453–455, August 1974.

[6] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5:190–222, April 1983.

[7] P. P. C. Lee, T. Bu, and G. Chandranmenon. A lock-free, cache-efficient shared ring buffer for multi-core architectures. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '09, pages 78–79, New York, NY, USA, 2009. ACM.

[8] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 25(1):60–99, Jan. 2006.

[9] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.

[10] NVIDIA. *Whitepaper NVIDIA Next Generation CUDATM Compute Architecture: FermiTM*, 1.1 edition, 2009.

[11] NVIDIA. *NVIDIA CUDA C Programming Guide*, 4.0 edition, 2011.

[12] NVIDIA. *Whitepaper NVIDIA GeForce GTX 680*, 1.0 edition, 2012.

[13] T. Preud'homme, J. Sopena, G. Thomas, and B. Folliot. BatchQueue: Fast and Memory-Thrifty Core to Core Communication. In *22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 215 –222, 2010.

[14] J. Stuart and J. Owens. Efficient synchronization primitives for gpus. *Arxiv preprint arXiv:1110.4623*, 2011.

[15] H. Sundell and P. Tsigas. Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. In *Proceedings of the 17th IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS)*, pages 84–94. IEEE press, 2003.

[16] The Khronos Group Inc. *OpenCl Reference Pages*, 1.2 edition, 2011.

[17] R. Treiber. System programming : Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, 1986.

[18] 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*, pages 134–143. ACM, 2001.

[19] K. Zhou, M. Gong, X. Huang, and B. Guo. Data-Parallel Octrees for Surface Reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 17(5):669 –681, may 2011.

[20] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics*, 27(5):1–11, Dec. 2008.