# Αποτελεσματικές Τεχνικές Συγχρονισμού για Συστήματα Διαμοιραζόμενης Μνήμης

Η ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Πληροφορικής Εξεταστική Επιτροπή

από τον

Νικόλαο Καλλιμάνη

ως μέρος των Υποχρεώσεων για τη λήψη του

ΔΙΔΑΚΤΟΡΙΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

Πανεπιστήμιο Ιωαννίνων

Μάιος 2013

Τριμελής Συμβουλευτική Επιτροπή (αλφαβητικά)

- **Βασίλειος Δημακόπουλος**, Αναπληρωτής Καθηγητής του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- **Λεωνίδας Παληός**, Αναπληρωτής Καθηγητής του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- **Παναγιώτα Φατούρου**, Επίκουρη Καθηγήτρια του Τμήματος Επιστήμης Υπολογιστών του Πανεπιστημίου Κρήτης

Επταμελής Εξεταστική Επιτροπή (αλφαβητικά)

- **Βασίλειος Δημακόπουλος**, Αναπληρωτής Καθηγητής του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- **Απόστολος Ζάρρας**, Επίκουρος Καθηγητής του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- **Κωνσταντίνος Μαγκούτης**, Ερευνητής Γ΄ του Ινστιτούτου Πληροφορικής, Ίδρυμα Τεχνολογίας και Έρευνας
- **Δημήτριος Νικολόπουλος**, Καθηγητής της Σχολής Electronics Engineering and Computer Science του Πανεπιστημίου Queens University of Belfast
- **Λεωνίδας Παληός**, Αναπληρωτής Καθηγητής του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- **Ευαγγελία Πιτουρά**, Καθηγήτρια του Τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων
- **Παναγιώτα Φατούρου**, Επίκουρη Καθηγήτρια του Τμήματος Επιστήμης Υπολογιστών του Πανεπιστημίου Κρήτης

# Dedication

This dissertation is dedicated to my family.

# Acknowledgments

First, I would like to sincerely thank my supervisor Panagiota Fatourou for motivating and encouraging me during the entire period that I was conducting my PhD. I would also like to thank the members of my advisory committee, Vassilios Dimakopoulos and Leonidas Palios for their support. Many thanks to Dimitris Nikolopoulos for arranging the provision of access to some of the multi-core machines of the Department of Computer Science at Virginia Tech where I ran some of the experiments of this dissertation, and to Michael Scott for providing me access to the Rochester's Niagara 2 machine. I would also like to thank the rest of the members of my examination committee, Kostas Magoutis, Evaggelia Pitoura, and Apostolos Zarras.

Special thanks go to my friends Spiros Agathos, Eytychia Datsika, Vasilis Kagias, Kostas Lillis, Thanos Mpiliris, Odysseas Petrocheilos, and Kostas Ramantas for supporting and encouraging me.

Finally, I would like to thank Empirikion Foundation for the moral and financial support.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Abstract

Nikolaos D. Kallimanis.

Highly-Efficient Synchronization Techniques in Shared-Memory Distributed Systems.

PhD, Department of Computer Science, University of Ioannina, Greece.

May, 2013.

Thesis Supervisor: Vassilios Dimakopoulos.

Overcoming the difficulty of concurrent programming has never become more urgent due to the proliferation of multicore machines and the imperative necessity of exploiting their computational power. One way to achieve this is by designing efficient concurrent data structures; common structures, like stacks and queues, are the most widely used inter-thread communication mechanisms. Additionally, synchronization techniques are required to efficiently execute, in a concurrent environment, those parts of modern applications that require significant synchronization. Although the efficient parallelization of these parts is not an easy task, Amdhal's law implies that achieving this is necessary in order to avoid significant reductions in speed-up.

In this dissertation three families of highly efficient synchronization algorithms, called RedBlue, Sim and Synch are presented for executing concurrently blocks of code that have originally been programmed to be executed sequentially in asynchronous shared-memory distributed systems.

We start by presenting the RedBlue family of adaptive synchronization algorithms that use common base objects (`LL/SC` or `CAS` and `Read-Write`) provided by the majority of the real-world machines. The first of these algorithms achieves better time complexity than all previously presented algorithms and it matches a lower bound presented by Jayanti in PODC 1998. This algorithm uses large `LL/SC` base objects and it comprises the keystone for the design of the other RedBlue algorithms that use smaller base objects. Specifically,

the second algorithm significantly reduces the size of the required base objects. The last two algorithms have been designed for large objects improving previously presented work for large objects.

In the Sim family of synchronization algorithms, we aim at (1) getting better time complexity by using base objects other than LL/SC and read-write (i.e. Swap, Add, etc) and (2) competing in terms of performance with the state-of-the-art synchronization algorithms (i.e. high performance spin-locks, etc), while having the nice theoretical properties that RedBlue algorithms have. Sim algorithms achieve these goals.

Sim is a simple synchronization algorithm with constant step complexity using an Add additional to an LL/SC object. Sim answers the open problem that was mentioned by Jayanti in PODC 1998: "If shared-memory supports all of Read, Write, LL/SC, Swap, CAS, Move, Add, Fetch&Multiply, would the $\Omega(logn)$ lower bound still hold?". Sim has been implemented for a real shared-memory machine architecture. Its practical version, called P-Sim, outperforms several state-of-the-art lock-based and lock-free synchronization algorithms, while being *wait-free*, i.e. satisfying a stronger progress condition than all the algorithms that it outperforms.

The Sim and RedBlue families of synchronization algorithms can be considered as efficient wait-free implementations of the combining technique in which, one thread (the combiner) in addition to its own operation, serves the operations of other active threads. The RedBlue synchronization algorithms are adaptive and employ LL/SC (or CAS) and read-write base objects, whereas Sim are much simpler algorithms that are highly-efficient in practice and require Add base objects.

We further study blocking implementations of the combining technique with the goal of discovering where their real performance power resides and whether or how performance is impacted by ensuring some desired properties (e.g. fairness in serving requests). This is accomplished by presenting two new blocking implementations of the combining technique; the first (CC-Synch) is highly-efficient in systems that support coherent caches, whereas the second (DSM-Synch) works better in cache-less NUMA machines. In comparison to previous blocking implementations, the new implementations (1) provide bounds on the number of remote memory references (RMRs) that they perform, (2) support a stronger notion of fairness, and (3) use simpler and fewer base objects. CC-Synch and DSM-Synch achieve better performance than P-Sim as well as any other algorithm provided in the past. The experimental analysis sheds light to the questions that were aimed to be answered.

Several modern multicore systems organize the cores into clusters and provide fast communication within the same cluster and much slower communication across clusters. A hierarchical version of CC-Synch, called H-Synch, is presented, which exploits the hierarchical communication nature of such systems to achieve better performance. Experiments show that H-Synch significantly outperforms previous state-of-the-art hierarchical approaches.

Based on P-Sim, CC-Synch, DSM-Synch, and H-Synch, we provide very efficient implementations of common shared data structures like stacks and queues. Specifically, the implementations SimStack and SimQueue that are based on P-Sim are wait-free, whereas those based on CC-Synch, DSM-Synch and H-Synch are blocking but achieve better performance than SimStack and SimQueue as well as any other algorithm provided in the past. SimStack and SimQueue are the first stack and queue implementations that satisfy both wait-freedom and high performance.

The results of this dissertation have been published in the following conferences/journals: ACM PPoPP 2012, ACM SPAA 2011, DISC 2009 and Theory of Computing Systems Special Issue on SPAA 2011.

# Εκτεταμενη Περιληψη στα Ελληνικα

Νικόλαος Καλλιμάνης του Δημητρίου και της Νικολέττας.

PhD, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων.

Μάιος, 2013.

Αποτελεσματικές Τεχνικές Συγχρονισμού για Συστήματα Διαμοιραζόμενης Μνήμης.

Επιβλέπων: Βασίλειος Δημακόπουλος.

Η εξάπλωση των πολυπύρηνων επεξεργαστών τα τελευταία χρόνια έχει καταστήσει εξαιρετικά αναγκαία την εκμετάλλευση της υπολογιστικής ισχύος τους. Ένας τρόπος για την αποδοτική χρήση συστημάτων που βασίζονται σε πολυπύρηνους επεξεργαστές είναι ο σχεδιασμός αποδοτικών διαμοιραζόμενων (παράλληλα προσπελάσιμων από πολλά νήματα) δομών δεδομένων (π.χ. στοιβών και ουρών), οι οποίες χρησιμοποιούνται ως ένας θεμελιώδης μηχανισμός επικοινωνίας και συγχρονισμού μεταξύ των νημάτων του συστήματος. Η αποτελεσματική παράλληλη εκτέλεση πολλών εφαρμογών επιβάλει την ανάπτυξη αποδοτικών αλγορίθμων συγχρονισμού που θα συγχρονίζουν αποτελεσματικά τα τμήματα των εφαρμογών που εκτελούνται σε διαφορετικά επεξεργαστικά στοιχεία. Ο νόμος του Amdhal υποδεικνύει ότι η χρήση αποδοτικών τεχνικών συγχρονισμού είναι απαραίτητη για την επίτευξη της μέγιστης δυνατής ταχύτητας υπολογισμών.

Σε αυτή τη διατριβή παρουσιάζονται τρεις οικογένειες νέων αλγορίθμων συγχρονισμού εξαιρετικά υψηλής απόδοσης, οι οποίες ονομάζονται RedBlue, Sim και Synch. Οι εν λόγω αλγόριθμοι συγχρονισμού χρησιμοποιούνται για την παράλληλη εκτέλεση κώδικα που έχει προγραμματιστεί να εκτελείται σειριακά.

Αρχικά παρουσιάζονται οι προσαρμοστικοί αλγόριθμοι συγχρονισμού RedBlue (οι προσαρμοστικοί αλγόριθμοι έχουν χρονική πολυπλοκότητα ανάλογη του αριθμού των ενεργών νημάτων), οι οποίοι πληρούν την ιδιότητα ελεύθερη-αναμονής (wait-free) και είναι κατάλληλοι για ασύγχρονα συστήματα διαμοιραζόμενης μνήμης. Ο πρώτος από αυτούς τους αλγορίθμους, ο οποίος ονομάζεται F-RedBlue, επιτυγχάνει την καλύτερη χρονική πολυπλοκότητα από τους αλγορίθμους που είχαν παρουσιαστεί παλιότερα και είναι χρονικά βέλτιστος αφού επιτυγχάνει το κάτω όριο χρονικής πολυπλοκότητας που παρουσιάστηκε από τον Jayanti στο PODC 1998. Ο δεύτερος αλγόριθμος της οικογένειας RedBlue χρησιμοποιεί βασικά αντικείμενα μικρότερου μεγέθους από ότι ο F-RedBlue ενώ οι δύο τελευταίοι αλγόριθμοι της οικογένειας RedBlue βελτιώνουν τεχνικές που είχαν παρουσιαστεί παλιότερα.

Κατά την ανάπτυξη των Sim αλγορίθμων συγχρονισμού, στόχος ήταν (1) η περαιτέρω μείωση της χρονικής πολυπλοκότητας χρησιμοποιώντας βασικά αντικείμενα διαφορετικά των `LL/SC` και `Read-Write` (όπως `Swap` και `Add` βασικά αντικείμενα) και (2) η βελτίωση της απόδοσής τους, ώστε οι επιδόσεις τους να είναι τέτοιες που να ανταγωνίζονται ή και να ξεπερνούν τις επιδόσεις των γρηγορότερων αλγορίθμων συγχρονισμού (κλειδώματα, κτλ) έχοντας παράλληλα όλα τα καλά θεωρητικά χαρακτηριστικά των RedBlue αλγορίθμων. Η οικογένεια των Sim αλγορίθμων επιτυγχάνει όλους αυτούς τους στόχους.

Ο αλγόριθμος συγχρονισμού Sim χρησιμοποιεί ένα `Add` και ένα `LL/SC` βασικό αντικείμενο και επιτυγχάνει $O(1)$ χρονική πολυπλοκότητα. Ο Sim αλγόριθμος απαντά στο ανοιχτό πρόβλημα που τέθηκε από τον Jayanti στο PODC 1998, για το αν το κάτω όριο $\Omega(log(n))$ ισχύει στην περίπτωση που η διαμοιραζόμενη μνήμη υποστηρίζει όλα τους τύπους βασικών αντικειμένων `Read`, `Write`, `LL/SC`, `Swap`, `CAS`, `Move`, `Add` και `Fetch&Multiply`. Η χρονική πολυπλοκότητα του Sim είναι σταθερή, και επομένως η απάντηση στο ερώτημα αυτό είναι αρνητική. Η πρακτική έκδοση του Sim αλγορίθμου, που ονομάζεται P-Sim, ξεπερνά σε επιδόσεις τους γρηγορότερους αλγορίθμους συγχρονισμού, ενώ ταυτόχρονα πληροί την ισχυρότερη συνθήκη τερματισμού (ελεύθερη αναμονής).

Οι οικογένειες των RedBlue και Sim αλγορίθμων συγχρονισμού είναι ουσιαστικά αποδοτικές υλοποιήσεις της συνεργατικής τεχνικής (combining technique), στην οποία ένα νήμα είναι δυνατό να εφαρμόζει λειτουργίες και άλλων νημάτων βοηθώντας τα να τελειώσουν την εκτέλεσή τους. Οι RedBlue αλγόριθμοι είναι προσαρμοστικοί και χρησιμοποιούν `LL/SC` (ή `CAS`) βασικά αντικείμενα, ενώ οι Sim αλγόριθμοι είναι απλούστεροι αλγόριθμοι που στην πράξη επιτυγχάνουν πολύ υψηλές επιδόσεις, αλλά χρησιμοποιούν `Add` βασικά αντικείμενα.

Σε αυτή τη διατριβή μελετήθηκε σε βάθος η συνεργατική τεχνική με στόχο την ανάπτυξη εμποδιστικών (blocking) αλγορίθμων συγχρονισμού με βελτιωμένη απόδοση και με χαρακτηριστικά δικαιότερης εξυπηρέτησης. Αναπτύχθηκαν δύο νέοι εμποδιστικοί αλγόριθμοι συγχρονισμού που ανήκουν στην οικογένεια Synch. Ο πρώτος ονομάζεται CC-Synch και είναι κατάλληλος για μηχανές που υποστηρίζουν συνεπείς κρυφές μνήμες (coherent NUMA machines), ενώ ο δεύτερος ονομάζεται DSM-Synch και είναι κατάλληλος για πολυεπεξεργαστές χωρίς κρυφές μνήμες (cache-less NUMA machines). Σε αντίθεση με παλαιότερους εμποδιστικούς συνεργατικούς αλγορίθμους, οι παραπάνω αλγόριθμοι (1) προσφέρουν άνω όρια στον αριθμό των απομακρυσμένων αναφορών στη μνήμη, (2) προσφέρουν περισσότερη δικαιοσύνη κατά την προσπέλαση στο κοινόχρηστο αντικείμενο, και (3) χρησιμοποιούν απλούστερα βασικά αντικείμενα. Ο CC-Synch και ο DSM-Synch επιτυγχάνουν καλύτερη απόδοση από τον P-Sim, αλλά και όλους τους παλαιότερους αλγόριθμους συγχρονισμού.

Πολλά πολυπύρηνα συστήματα οργανώνουν τα επεξεργαστικά στοιχεία σε ομάδες και παρέχουν γρήγορη επικοινωνία μεταξύ των επεξεργαστικών στοιχείων που βρίσκονται στην ίδια ομάδα, ενώ παρέχουν αργή επικοινωνία μεταξύ των επεξεργαστικών στοιχείων διαφορετικών ομάδων. Σε αυτή τη διατριβή παρουσιάζεται μια ιεραρχική έκδοση του CC-Synch που ονομάζεται H-Synch. Ο H-Synch εκμεταλλεύεται την ιεραρχική φύση της επικοινωνίας τέτοιων συστημάτων και η πειραματική του μελέτη έδειξε ότι ξεπερνά κατά πολύ σε απόδοση όλες τους παλαιότερους ιεραρχικούς και μη αλγόριθμους συγχρονισμού.

Σε αυτή τη διατριβή παρουσιάζονται υλοποιήσεις διαμοιραζόμενων ουρών και στοιβών πολύ υψηλών επιδόσεων που βασίζονται στους P-Sim, CC-Synch, DSM-Synch και H-Synch. Ειδικότερα, οι υλοποιήσεις SimStack και SimQueue που βασίζονται στον P-Sim ικανοποιούν τη συνθήκη τερματισμού ελεύθερη-αναμονής, ενώ εκείνες που βασίζονται στους CC-Synch, DSM-Synch και H-Synch είναι εμποδιστικές αλλά επιτυγχάνουν καλύτερες επιδόσεις από τον SimStack και SimQueue αλλά και όλες τις παλαιότερες υλοποιήσεις. Οι SimStack και SimQueue είναι οι πρώτες υλοποιήσεις κοινόχρηστων στοιβών και ουρών που πληρούν την ιδιότητα ελεύθερη-αναμονής και ταυτόχρονα επιτυγχάνουν υψηλή απόδοση.

Τα ερευνητικά αποτελέσματα αυτής της διατριβής έχουν παρουσιασθεί στα διεθνή συνέδρια/περιοδικά: ACM PPoPP 2012, ACM SPAA 2011, DISC 2009 και Theory of Computing Systems Special Issue on SPAA 2011.

# Chapter 1

# Introduction

The last decade, the computer industry has made a significant turn towards developing multicore systems which nowadays, are used in any computing device (from smartphones to large scale multiprocessor machines). A wide variety of low cost commercial computing devices are equipped with processors containing a dozen or more processing cores. Even smartphones are equipped with multicore processors. In all of these devices, increased performance can be achieved by exploiting parallelism; thus, harnessing the difficulty of concurrent programming is currently very important.

Multicore systems are typical examples of distributed systems. A *distributed system* consists of a set of computing entities (threads), which have the ability to communicate. Distributed systems are distinguished in two main types depending on how the threads communicate. The first type consists of systems where threads communicate through a shared memory (*shared memory systems*), while the second type consists of systems that their threads communicate by exchanging messages (*message passing systems*). In recent years, a lot of research is conducted in shared memory systems due to the proliferation of the multicore systems. A multicore system is usually a shared memory system, since it consists of many tightly connected processing cores that communicate through shared memory.

Several applications that could be parallelized contain parts whose parallelization requires significant synchronization and coordination. Amdhal's law [9] implies that failing in parallelizing these parts may result in a significant limitation on the speed-up that could be achieved. However, these parts usually require accesses to shared data and thus, parallelizing them demands the design of low-overhead synchronization mechanisms; without such efficient mechanisms the synchronization cost may overshadow any performance gain that could result from the parallelization of these parts.

In a shared memory system, threads use shared atomic objects (or briefly atomic objects) as main communication mechanism. Every atomic object stores some information, which is accessible to system's threads via atomic operations. Intuitively, an atomic operation is an operation that seems to be executed instantly at some point in time. Some objects, called base objects, are provided by the hardware and therefore the hardware guarantees that the supported operations are executed atomically.

The most common type of base objects are the `Read-Write` ones. A `Read-Write` base object supports two operations for accessing and modifying the stored data: a) `Read`($O$), which returns the stored data of $O$ without modifying it, and b) `Write`($O$, $v$), which stores value $v$ in $O$ and returns an acknowledgment.

Other types of base objects are `CAS`, `LL/SC`, `Add`, `Swap`, etc. Specifically, a `CAS` *base object* $O$ supports two operations: a) `Read`($O$) that returns the stored value in $O$ without modifying it and b) `CAS`($O$, $v_{old}$, $v_{new}$). `CAS`($O$, $v_{old}$, $v_{new}$) compares the current value of $O$ with $v_{old}$ and if they are equal it stores the value $v_{new}$ in $O$ and returns *true*. Otherwise, the contents of $O$ remain unchanged and *false* is returned. An `Add` object supports, in addition to `Read`, the operation `Add`($O, x$) that atomically adds some (positive or negative) value $x$ to object $O$. A `Swap` *object* $O$ supports in addition to `Read`, the operation `Swap`($O, v$) which (atomically) writes in $O$ the value $v$ and returns the previous value of $O$. An `LL/SC` *object* $O$ supports the atomic operations a) `LL`($O$) which returns the current value of $O$, and b) `SC`($O, v$) whose execution by a thread $p_i$ must follow the execution of `LL`($O$) by $p_i$ and changes the value of $O$ to $v$ if no other `SC` (by some other thread) has changed the value of $O$ since the execution of $p_i$'s latest `LL` on $O$. If the value of $O$ changes to $v$ by `SC`($O, v$), *true* is returned; otherwise, the value of $O$ does not change and *false* is returned.

Apparently, common base objects as those described above offer very simple operations for accessing stored data. The design of more complex atomic objects significantly simplifies the parallel programming of most modern applications. Thus, the design and implementation of such complex objects in software using simpler objects provided by the hardware is of high importance.

Any atomic object can be easily implemented using locks. A thread that wishes to perform an operation to the shared object, acquires the lock that is associated to the shared object, executes the sequential code of the operation and releases the lock. This methodology has been widely used in several real-world applications systems (e.g. data base applications, etc). However, this technique has a serious drawback; a thread may fail (i.e. stops its execution due to a software or hardware failure) while holding the lock leading the system to a total failure. Properties that guarantee system's progress are desirable, since it is very important for a system to be fault tolerant. A property that guarantees high tolerance in thread failures is *wait-freedom* [12, 16]. Wait-freedom ensures that each thread finishes the execution of the code block it wants to execute within a finite number of its own steps independently of the speed or the state of the other threads.

Atomic objects are arguably useful; however they are practical only in the case that they are implemented efficiently. From a theory perspective, the main complexity measures are the step complexity of an implementation, and the number and size of the base objects it employs. The *step complexity* of an operation is the maximum number of shared memory accesses that any thread executes in order to complete the operation. Some desirable properties when designing atomic objects in software are the following:

- The step complexity of every operation of the object should be as low as possible.
- The used base objects should support as fewer complex operations as possible.
- The needed base objects should have size equivalent to the size of hardware base objects (usually less or equal to 128 bits).
- The implementation should be fault tolerant, thus wait-freedom property should be satisfied.

A *universal synchronization algorithm* is a generic mechanism to implement any shared object; it supports an operation, called ApplyOp, that takes as a parameter the sequential implementation of any operation of the simulated object, and simulates its execution in a concurrent environment. A universal algorithm provides the implementation of any shared object for free. So, if efficient implementations of universal algorithms are provided then the programming effort is highly reduced and high performance is achieved.

In the first part of this dissertation, a family of wait-free universal synchronization algorithms, called RedBlue, is presented. In shared memory systems it is often the case that the total number of threads $n$ taking part in a computation is much larger than the actual number of threads that concurrently access the shared object. For this reason, a flurry of research [2, 3, 13, 14, 38] has been devoted to the design of *adaptive* algorithms whose time complexity depends on $k$, the maximum number of threads that concurrently access the shared object. All RedBlue algorithms are adaptive.

All RedBlue algorithms use two perfect binary trees of $\lceil \log_2 n \rceil + 1$ levels each. The first tree (*red* tree) is employed for the estimation of any encountered contention, while the second tree (*blue* tree) is used for the synchronization with other threads when applying an operation. In each of these trees, a thread is assigned a leaf node (and therefore also a path from this leaf to the root node, or vice versa). A thread that wants to apply an operation to the simulated object, traverses first its path in the red tree from the root downwards looking for an unoccupied node in this path. Once it manages to occupy such a node, it starts traversing the blue tree upwards from the isomorphic blue node to the occupied red node, transferring information about its operation (as well as about other active operations) towards the tree's root. In this way, each operation traverses at most $O(\min\{k, \log n\})$ nodes in each of the two trees. Once information about the operation reaches the root, the operation is applied to the simulated object.

The first algorithm of the RedBlue family, which is called F-RedBlue, has time complexity $O(\min\{k, \log n\})$ which is better than any previously presented algorithm using LL/SC and read-write base objects. However, F-RedBlue uses big LL/SC base objects; thus it is mainly of theoretical interest. A lower bound of $\Omega(\log n)$ on the time complexity of wait-free universal synchronizations algorithms that use LL/SC base objects is presented in [42]. It holds even if an infinite number of unbounded-size base objects is employed. Therefore, F-RedBlue is optimal in terms of time complexity.

The second algorithm (S-RedBlue) of the RedBlue family is a slightly modified version of F-RedBlue that uses smaller base objects and it is therefore practical in many cases. S-RedBlue uses $O(n)$ LL/SC base objects, one for each of the trees' nodes and $n+1$ single-writer base objects per thread. Each base object of the red tree has size $\lceil \log_2 n \rceil + 1$. Each base object of the blue tree stores $n$ bits, one for each thread. One of the base objects (the base object corresponding to the blue root) is big. This base object is implemented by single-word `LL/SC` objects using the technique presented in [44]. In current systems where base objects of 128 bits are available, S-RedBlue works with single-word `LL/SC` objects for up to 128 threads. In fact, even if $n/128 = c > 1$, where $c$ is any constant, the algorithm can be implemented by single-word `LL/SC` base objects with the same time complexity (increased by a constant factor) using the implementation of multi-word LL/SC from single-word `LL/SC` of [44].

Most of the universal algorithms presented in the past, as well as F-RedBlue and S-RedBlue, copy the entire state of the object each time an update is to be performed on it by some thread. This is not practical for large objects whose states may require a large amount of storage to maintain. Anderson and Moir [11] presented a lock-free and a wait-free synchronization algorithm that is practical for large objects. Their algorithms assume that the object state is represented as a continuous array which requires $B$ data blocks of size $S$ each for its storage. Each operation can modify at most $T$ blocks and each thread can help at most $M \geq 2T$ other threads. We combine some of the techniques introduced in [11] with the techniques employed by the RedBlue algorithms in order to design two simple wait-free synchronization algorithms which have the nice properties of the constructions in [11] while achieving better time complexity and being adaptive. The time complexity of the first algorithm is better than the synchronization algorithm presented in [11] but it does not assume an upper bound on the number of threads a thread may help as the wait-free construction in [11] does. BLS-RedBlue exhibits all the properties of the wait-free construction in [11] and still achieves better time complexity. In particular, its time complexity is similar to the time complexity of the wait-free algorithm in [11] but with $k$ replacing $n$ and thus the algorithm is adaptive. The space complexity of the algorithm is the same as that of the wait-free algorithm in [11]. RedBlue algorithms are much simpler than the constructions presented in [11], and they improve on time complexity upon these algorithms. Table 1.1 provides the exact time complexities and the space overheads of all of the algorithms presented in this dissertation.

In the Sim family of synchronization algorithms, we aim at (1) getting better time complexity by using base objects other than `LL/SC` and read-write (i.e. `Swap`, `Add`, etc) and (2) competing in terms of performance with the state of the art synchronization algorithms

| Algorithm | Base objects | Progress property | Published in |
|---|---|---|---|
| Synchronization Algorithms | | | |
| F-RedBlue | CAS, rw | wait-free | DISC '09 |
| S-RedBlue | CAS, rw | wait-free | DISC '09 |
| LS-RedBlue | CAS, rw | wait-free | DISC '09 |
| BLS-RedBlue | CAS, rw | wait-free | DISC '09 |
| Sim | Add, CAS, rw | wait-free | SPAA '11 |
| P-Sim | Add, CAS, rw | wait-free | SPAA '11 |
| L-Sim | Add, CAS, rw | wait-free | unpublished |
| CC-Synch | Swap, rw | blocking | PPoPP '12 |
| DSM-Synch | Swap, CAS, rw | blocking | PPoPP '12 |
| H-Synch | Swap, rw | blocking | PPoPP '12 |
| Shared Stacks | | | |
| SimStack | Add, CAS, rw | wait-free | SPAA '11 |
| CC-Stack | Swap, rw | blocking | PPoPP '12 |
| DSM-Stack | Swap, CAS, rw | blocking | PPoPP '12 |
| H-Stack | Swap, rw | blocking | PPoPP '12 |
| Shared Queues | | | |
| SimQueue | Add, CAS, rw | wait-free | SPAA '11 |
| CC-Queue | Swap, rw | blocking | PPoPP '12 |
| DSM-Queue | Swap, CAS, rw | blocking | PPoPP '12 |
| H-Queue | Swap, rw | blocking | PPoPP '12 |

Table 1.1: Algorithms and their properties proposed in this dissertation.

(i.e. high performance spin-locks, etc), while having the nice theoretical properties that RedBlue algorithms have. The family of Sim synchronization algorithms achieve these goals.

The Sim synchronization algorithm follows the simple idea presented by Herlihy in [37]: a thread $p$ starts by recording the request that it wants to execute in a shared struct that it owns. This struct additionally contains a toggle bit. A set of toggle bits, one for each thread, are also stored as part of the simulated state. Based on the values of the toggle bits, $p$ finds out which other requests are active and serves them by executing their code on a local copy of the simulated state. Finally, $p$ tries to change a shared reference, stored in an LL/SC object, to point to this local struct. Process $p$ may have to apply these steps twice to ensure that its request has been served. An array containing $n$ response values is also stored as part of the simulated state. Once $p$ ensures that its request has been served, it finds its response value in the LL/SC object.

We start with Sim, a simplified version of this technique that allows us to derive some theoretical results. In Sim, the announcement of the requests and the discovery of the active requests by each thread have been abstracted using a collect object. A *collect object* consists of $n$ components $A_1, ..., A_n$, one for each thread, where each component stores a value from some set and supports two operations UPDATE($v$) and COLLECT. When executed by thread $p_i$, $1 \leq i \leq n$, UPDATE($v$) stores the value $v$ in $A_i$; COLLECT returns a vector of $n$ values, one for each component. It is remarkable that a collect object is not atomic (see Section 3 for a description of the correctness condition that needs to be ensured by an implementation of a collect object). A *snapshot* object is an atomic version of a collect object.

We describe simple implementations of collect and snapshot objects using a single atomic `Add` (or `XOR`) object. An `Add` (`XOR`) object supports the operation `Add` (`XOR`) in addition to `Read`; `Add`$(O, x)$ adds some (positive or negative) value $x$ to object $O$ (`XOR`$(O, x)$ computes $O$ `XOR` $v$ and stores it into $O$). These implementations exhibit constant step complexity (under the standard theoretical model of shared memory computation where even if the size of the `Add` object is large, an `Add` can be executed atomically as a single step). Using these simple implementations, one could get improved performance for several previously presented algorithms [7, 15, 40, 57].

By plugging in to Sim the implementation of collect discussed above, the step complexity of Sim becomes constant as well. Jayanti [42] has proved a lower bound of $\Omega(\log n)$ on the step complexity of any oblivious universal synchronization algorithm using `LL/SC` objects; an *oblivious* universal synchronization algorithm does not exploit the semantics of the object being simulated. This lower bound holds even if the size of the base objects used by the universal synchronization algorithm is unbounded. One of the open problems mentioned in [42] is the following: *"If shared-memory supports all of* `Read`*,* `Write`*,* `LL/SC`*,* `Swap`*,* `CAS`*,* `Move`*,* `Fetch&Add`*, would the* $\Omega(\log n)$ *lower bound still hold?"* Sim has constant step complexity and it uses a single `Add` (or `XOR`) object in addition to an `LL/SC` object, thus proving that the lower bound in [42] can be beaten if we use just a single `Add` (or `XOR`) object in addition to an `LL/SC` object. So, an $\Omega(\log n)$ lower bound can be derived for the step complexity of any implementation of an `Add`, `XOR`, collect, or a snapshot object, from `LL/SC` objects.

Sim is an efficient wait-free implementation of the well-known combining technique [29, 34, 37, 52, 54, 56, 60]. Most of the previous implementations of this technique, including the algorithm presented in [52] (which we will call OyamaAlg from now on) and flat-combining [34], employ locks and therefore they are *blocking* (i.e. threads may have to wait for actions performed by other threads in order to make progress). Specifically, in those algorithms, a thread, called the *combiner*, holding a coarse-grain lock, serves, in addition to its own request, active requests announced by other threads while they are waiting by performing local spinning (and possibly periodical checking of the lock status).

We present a practical version of Sim, called P-Sim, which we have implemented and experimentally tested on a real shared memory machine. We provide a detailed experimental analysis illustrating that P-Sim is highly-efficient in practice. Specifically, our experiments show that P-Sim outperforms several state-of-the-art synchronization algorithms, both lock-based (like local spinning) and lock-free (Figures 5.2-5.12). Moreover, the performance of P-Sim is as good as that of the best-known implementations [34, 52] of the combining technique, and in some cases even better than them. More specifically, we experimentally compare P-Sim with OyamaAlg [52], flat-combining [34], CLH spin locks [23, 47], and a simple lock free algorithm. Our experiments (Figure 5.2) show that P-Sim outperforms all these algorithms in several cases. Besides that, P-Sim is wait-free whereas all other algorithms ensure only weaker progress properties. P-Sim proves that the common belief that ensuring wait-freedom is too expensive to be practical is in many cases wrong.

We have used P-Sim to design new highly-efficient *wait-free* implementations of common concurrent data structures like queues and stacks. We experimentally prove that our stack implementation, called SimStack, outperforms most well-known previous shared stack algorithms, like the lock-free stack implementation of Treiber [58], the elimination back-off stack [35], a stack implementation based on a CLH spin lock [23, 47], and a linked stack implementation based on flat-combining [34]. Similarly, our queue implementation, called SimQueue significantly outperforms the following previous queue implementations: a lock-based algorithm [50] which uses two CLH locks [23, 47], the lock-free algorithm presented in [50], and the implementation using flat-combining provided by Hendler *et. al* [34].

In this dissertation, a further investigation of the combining technique is provided aiming at discovering where its real performance power resides, understanding the performance implications of using different primitives when implementing it, and investigating whether and how ensuring some desired properties (e.g., fairness in serving requests) would impact performance. We do so by presenting two new blocking implementations of this technique. The first, called CC-Synch, is suitable for *cache coherent* (CC) shared memory systems where accesses to shared objects are performed via cached copies of them; an access to a shared object is a *remote memory reference* (RMR) if the cached copy of this object is invalid, so the access causes a cache miss*. The vast majority of modern parallel architectures follow the CC shared memory model. The second implementation, called DSM-Synch, is better suited for the *cache-less NUMA* shared memory systems, where a part of the shared memory is associated with each processor; so, each shared object is allocated (and resides) in the part of the shared memory that is associated to a specific processor. Processors do not have access to local caches, so a thread $p$ performs a *remote memory reference* (RMR) if it accesses a shared object residing in the shared memory part of some processor other than that where $p$ is being executed. Since an RMR is significantly more costly than a local memory reference [49], it is highly desirable to design algorithms that perform as few RMRs as possible; CC-Synch and DSM-Synch perform a bounded number of RMRs.

CC-Synch and DSM-Synch use a single FIFO queue to both implement the lock and store the active synchronization requests. Therefore, the synchronization needed for implementing the list of active requests comes for free. Specifically, each newly activated thread adds a node to the tail of the queue to announce its request and participate to the implementation of the lock. Thus, each active thread is assigned one of the nodes of the queue. The active thread $q$ that owns the first node of the queue becomes the combiner and undertakes the responsibility of applying some (or all) of the requests listed in the queue. Each active thread whose node is not first in the queue performs local spinning.

The experimental analysis (Section 6.4) reveals that the use of a highly-efficient queue-like lock which, in addition to its low synchronization overhead, provides the implementation of the list of announced requests for free, significantly reduces the synchronization required to implement the combining technique. Moreover, the new implementations are simpler to program than previous combining-based synchronization approaches [34, 52].

* Once the cache miss is served and as long as the data item is not updated by threads that are being executed on other processors, future accesses to the data item by threads that are being executed on this processor are local.

These result in a performance benefit in comparison to P-Sim as well as to any other algorithm provided in the past. Additionally, the new implementations exhibit several nice properties, not ensured by previous blocking combining implementations [34, 52]. First, they provide stronger fairness guarantees in serving the requests. Second, they provide bounds on the number of remote memory references that are executed. Specifically, in CC-Synch, the combiner thread performs $O(h + t)$ RMRs, where $h$ is an upper bound on the number of synchronization requests that the combiner may serve, and $t$ is the size of the shared data that should be accessed in order to execute these $h$ requests; we remark that $h$ is a parameter that can be determined by the user and it can be chosen to be constant. The combiner in DSM-Synch performs $O(dh)$ RMRs, where $d$ is the average number of RMRs required to serve a single request. In both algorithms, all threads, other than the combiner, perform local spinning and cause only a constant number of RMRs. Thus, the amortized number of performed RMRs is $O(d)$. Moreover, no thread may ever starve. Finally, the new implementations do not employ any form of backoff and they need minimal tuning to achieve the best performance.

CC-Synch uses a `Swap` object in addition to `Read-Write` base objects. DSM-Synch uses an object that supports `CAS` and `Swap` in addition to `Read-Write` base objects; a `CAS`$(O, u, v)$ (atomically) checks if the current value of $O$ is $u$ and if this is so, it changes the value of $O$ to $v$ and returns *true*, otherwise the value of $O$ remains unchanged and *false* is returned. CC-Synch and DSM-Synch use just one primitive stronger than `Read-Write` base objects and in CC-Synch this is a `Swap` object which is weaker than `CAS`. In CC-Synch, each thread maintains a single node to insert in the list, and therefore the total space overhead of CC-Synch is $O(n)$, where $n$ is the number of threads; this is no more than that of previous combining-based synchronization approaches. The total space overhead for DSM-Synch is also $O(n)$.

We experimentally compare CC-Synch and DSM-Synch with several state-of-the-art synchronization approaches, like P-Sim, flat-combining [34], CLH spin locks [23, 47], and a simple lock free algorithm. The experiments (Figures 6.1-6.10) show that CC-Synch outperforms all these approaches in most cases. DSM-Synch outperforms all algorithms other than CC-Synch. DSM-Synch has the advantage over CC-Synch that it is designed to be efficient even in machines that support the DSM model; so, it can be executed efficiently by architecture unaware applications.

The experimental analysis reveals that the number of cache misses incurred per request is smaller in the new implementations than in previous algorithms and the same is true for the cycles invested in memory stalls. Based on experiments, we conclude that the algorithm of repeatedly performing CAS until it succeeds, even if it comes together with an appropriately-tuned back-off scheme, causes more cache misses and more branch mispredictions than employing Swap or other non-comparison primitives. Experiments also show that the average number of requests served by a combiner in CC-Synch and DSM-Synch is larger than in other algorithms, so the synchronization overhead paid to serve an amount of requests in these implementations is closer to the ideal than in previous approaches. So, the achieved combining degree has a significant impact on the performance of combining implementations.

We used CC-Synch and DSM-Synch to implement shared stacks and queues (Section 6.5). The stack implementation (CC-Stack) based on CC-Synch, outperforms all state-of-the-art shared stack implementations like SimStack, the linked stack implementation based on flat-combining [34] where elimination has also been applied [35], and the stack implementation based on CLH spin locks [23, 47]. The stack implementation (DSM-Stack) based on DSM-Synch, outperforms all implementations other than CC-Stack. We also use CC-Synch and DSM-Synch to get two highly efficient shared queue implementations, called CC-Queue and DSM-Queue. More specifically, these implementations are derived by simply replacing the ordinary locks in the two-locks queue implementation presented by Michael and Scott in [50] with two instances of either CC-Synch or DSM-Synch. These implementations were experimentally compared to SimQueue, the two-locks implementation [50], and the queue implementation based on flat-combining presented in [34]. CC-Queue performs up to 2.5 times faster than the queue implementation of [34] and outperforms SimQueue by a factor of up to 1.5.

For modern multi-core systems that organize the cores into clusters and provide fast communication (via shared caches) to the threads running in the same cluster and much slower communication across clusters, we present an hierarchical version of CC-Synch, called H-Synch, which exploits the hierarchical communication nature of such systems to achieve better performance. Experiments show that in such systems, H-Synch significantly outperforms CC-Synch and DSM-Synch as well as the state-of-the-art flat-combining NUMA locks recently presented by Dice *et. al* in [24]. H-Synch is used to design highly efficient implementations of stacks and queues for such machines. These implementations outperform by far, in such machines, CC-Stack, DSM-Stack, CC-Queue and DSM-Queue, respectively, as well as all other concurrent stack and queue implementations with which these implementations have been compared.

Many hardware manufactures have been influenced by the universality result [36], and they have equipped their machines with strong atomic primitives (like CAS and LL/SC). Sim shows that machines that additionally support Add instructions, have important performance advantages, and can ensure wait-freedom. CC-Synch and DSM-Synch show that machines that support Swap objects have even better performance benefits. We believe that the results of this dissertation provide some motivation for seeing primitives such as Add provided in the instruction set of more architectures in the future.

Note that CC-Synch, similarly to Sim and flat-combining [34], cannot be trivially applied in an efficient way for designing data structures such as search trees, where $m$ lookups can be executed in parallel performing just a logarithmic number of shared memory accesses each. In such cases, it is expected that CC-Synch will perform well, only if several instances of it are employed. It is an interesting open problem to find efficient ways to synchronize these instances. It is also not obvious how to use the combining technique to implement data structures, like shared linked lists, if several instances of the combining implementation should be employed to achieve good speed-up.

The synchronization algorithms of the RedBlue synchronization algorithms have been presented in DISC '09 [27], synchronization algorithms based on Sim have been presented in SPAA 2011 [28] and an extended version will appear to Theory of Computing Systems Special Issue on SPAA 2011, while the Synch synchronization algorithms are presented in PPoPP [29].

This dissertation is organized as follows. The related work is discussed in Chapter 2. The model of the system is described in Chapter 3. In Chapter 4, the family of RedBlue algorithms is presented. The family of Sim algorithms is provided in Chapter 5. Finally, the family of Synch algorithms is presented in Chapter 6.

# Chapter 2

# Related Work

In [36], Herlihy provides the first wait-free universal synchronization algorithm using `Read-Write` base objects and consensus objects. This universal algorithm can be used to simulate any other shared object in a system of $n$ threads. Herlihy's algorithm uses $O(n^2)$ `Read-Write` base objects and $O(n^2)$ consensus objects of size $s$, where $s$ is the size of the state of the simulated object. The consensus objects can be easily implemented by using `CAS` or `LL/SC` base objects [36]. The step complexity of Herlihy's synchronization algorithm is $O(n)$.

Afek, Dauber and Touitou [4] have presented algorithm GroupUpdate which also uses a tree technique to keep track of the list of active threads. They then combine this tree construction with Herlihy's universal algorithm [36, 37] to get a universal construction with time complexity $O(k \log k + W + kD)$, where $W$ is the size (in words) of the simulated object state and $D$ is the time required for performing a sequential request on it. F-RedBlue retains the basic structure of GroupUpdate but achieves better time complexity ($O(\min\{k, \log n\})$) by employing a faster mechanism to discover the encountered contention and by using large `LL/SC` base objects. S-RedBlue addresses the problem of using large base objects still achieving better time complexity than GroupUpdate.

Although the first of the RedBlue algorithms shares a lot of ideas with GroupUpdate, it also exhibits several differences: (1) it employs two complete binary trees each of which has one more level than the single tree employed by GroupUpdate; in each of these trees, each thread is assigned its own leaf node which identifies a unique path (from the root to this leaf) in the tree for the thread; (2) threads traverse the red tree first in order to occupy a node and this procedure is faster than a corresponding procedure in GroupUpdate. More specifically, GroupUpdate performs a $BFS$ traversal of its employed tree in order for a thread to occupy a node of the tree, while each thread in any of the RedBlue algorithms always traverses appropriate portions of its unique path. This results in reduced time complexity for some of the RedBlue algorithms.

Afek, Dauber and Touitou [4] present a technique that employs indirection to reduce the size of the base objects used by GroupUpdate (each tree base object stores a thread id and a pointer to a list of ids of currently active threads). A similar technique can be applied to the RedBlue algorithms in case $n$ is too large to have $n$ bits stored in a constant number of LL/SC base objects. The resulting algorithms will have just a pointer stored in each of the blue nodes (thus using smaller base objects than GroupUpdate which additionally stores a thread id in each of its LL/SC base objects). However, employing this technique would cause an increase to the step complexity of our algorithms by an $O(k \log n)$ additive term.

Afek, Dauber and Touitou present in [4] a second universal construction, called IndividualUpdate, that has time complexity $O(k(W + D))$. IndividualUpdate stores sequence numbers in base objects and therefore it requires unbounded size base objects or base objects that support the VL request in addition to LL and SC. The first two RedBlue algorithms achieve better time complexity than IndividualUpdate. Some of our algorithms use single-word base objects (however, they also employ LL/VL/SC objects).

Afek, Dauber and Touitou [4] discuss a method similar to that presented in [17] to avoid copying the entire object's state in IndividualUpdate. The resulting algorithm has time complexity $O(kD \log D)$. The work of Anderson and Moir on universal constructions for large objects [11] follows this work. Our last two algorithms improve in terms of time complexity upon the constructions presented in [11]. They achieve this using single-word base objects (and the last algorithm with the same space complexity as the wait-free construction in [11]).

Jayanti [43] presented f-arrays, a generalized version of a snapshot object which allows the execution of any aggregation function $f$ on the $m$ elements of an array of $m$ memory cells that can be updated concurrently. As F-RedBlue, f-arrays has time com-

| Algorithm | Primitives | Shared Memory Accesses | Required Space |
|---|---|---|---|
| Wait-free synchronization algorithms presented in this dissertation | | | |
| F-RedBlue | LL/SC | $O(\min\{k, \log n\})$ | $O(n^2 + s)$ |
| S-RedBlue | LL/VL/SC, r/w regs | $O(k + s)$ | $O(n^2 + ns)$ |
| LS-RedBlue | LL/VL/SC, r/w regs | $O(B + k(w + TL))$ | $O(n^2 + n(B + kTL))$ |
| BLS-RedBlue | LL/VL/SC, r/w regs | $O((k/\min\{k, M/T\})\ (B + ML + k + \min\{k, M/T\}w))$ | $O(n^2 + n(B + ML))$ |
| Sim | LL/SC or CAS, FAD | $O(1)$ | $O(n + s)$ |
| P-Sim | LL/SC or CAS, Fetch&Add | $O(n + s)$ | $O(n^2 + ns)$ |
| Related Work | | | |
| Herlihy [36] | consensus objects, r/w regs | $O(n)$ | $O(n^3 s)$ |
| Herlihy [37] | LL/VL/SCor CAS, r/w regs | $O(n + s)$ | $O(ns)$ |
| GroupUpdate [4] | LL/SC r/w regs, consensus objects | $O(\min\{n, k \log k\})$ | $O(n^2 s \log n)$ |
| IndividualUpdate [4] | LL/VL/SC | $O(kw \log w)$ | $O(nw + s)$ |
| Anderson & Moir [10] | LL/VL/SC | $O((n/\min\{k, M/T\})\ (B+ ML + nw))$ | $O(n^2 + n(B + ML))$ |
| Chuong, *et. al* [20] | CAS, r/w regs | $O(nw)$ | $O(n + s)$ |

Table 2.1: Wait-free universal algorithms and their complexities.

plexity $O(\min\{k, \log n\})$; the algorithm uses a tree structure similar to that employed by GroupUpdate and our algorithm. F-RedBlue is universal, thus achieving wider functionality than f-arrays. Constructions for other restricted classes of objects with polylogarithmic complexity are presented in [19].

Afek *et al.* [5, 6] and Anderson and Moir [10] have presented universal algorithms for multi-object requests that support access to multiple objects atomically. The main difficulty encountered under this setting is to ensure good parallelism in cases where different requests perform updates in different parts of the object's state. In Table 2.1, a comparison between the wait-free algorithms proposed in this dissertation and previous work is displayed. Notice that $w$ is the maximum number of different memory words accessed by an operation on the sequential data structure. In [10, 27], $B$ is the number of blocks, each of size $L$, required to store the object's state, and each thread is allowed to modify at most $T$ blocks and help at most $M/T$ other threads, where $M \geq 2T$ is some integer.

P-Sim uses an efficient implementation of the Add-based collect object. This allows a thread to read only the announcement records of those requests that are active improving upon the technique described in [37] where threads read all $n$ such records. Furthermore, in P-Sim each thread uses its own pool of structs to store the simulated state. In the recycling technique of [37] threads share the same pool which leads to a significantly higher number of cache misses. P-Sim validates not only whether the copied state is consistent (as does the validation mechanism in [37]), but also if the reference to the shared state has changed in the meantime. If the validation fails, P-Sim avoids performing unnecessary work. Moreover, P-Sim uses a simple backoff scheme to guarantee that a thread executing a request will help a large number of other active threads; in [37], the employed backoff scheme aims at reducing the contention in updating the LL/SC object.

Herlihy [37] starts by presenting a lock-free version of the universal synchronization algorithm where each thread does not help requests initiated by other threads. It rather performs LL on the reference to the simulated state, copy the state locally and apply its SC; these actions are applied repeatedly until the SC succeeds. Experiments presented in [37] show that Herlihy's wait-free implementation does not perform well in comparison to this lock-free version and a Test-And-Test-And-Set spin lock (with backoff).

The combining technique is old. It was first been introduced by Gottlieb *et. al* on network switches [31] that connect processors to memory; messages with the same destination were merged to reduce memory traffic and contention. Software combining was first realized in combining trees [30, 60], at which requests to modify a concurrent data structure are transferred from the leaves of the tree towards the root applying combining at every internal node. However, each thread applies $\Theta(\log n)$ CAS operations per access and therefore the synchronization overhead is high. To reduce this overhead a lot of research work has focused on designing adaptive versions of combining trees [32, 49] (e.g., for implementing barriers) or decentralized algorithms for dynamically changing tree size [55, 56]. For some of these algorithms [32, 49], it is not clear how they can be used to design general concurrent data structures, others [55] satisfy weaker consistency conditions than linearizability, and for others, experiments [34] have shown that the synchronization overhead they introduce is still high.

Oyama, Taura and Yonezawa present in [52] a different approach for implementing the combining technique. Their algorithm uses a coarse-grain lock implemented with a CAS object $O$, and a list of announced requests implemented as a stack. Each thread has a record that it uses to announce its request by pushing it in the stack. The CAS object may store the values free, locked, or a pointer to some of the threads' records. A thread with a newly-activated request first performs a CAS in an effort to change the value of $O$ from free to locked. If this CAS succeeds, the thread becomes the combiner and starts executing its request. Otherwise, the thread tries to announce its request by inserting its record in the stack; this is done by repeatedly performing CAS on $O$ to change its value from locked to a pointer to its record. If this succeeds, the thread performs local spinning on its record until the combiner thread notifies it that its request has been served. When the combiner finishes the execution of its own request, it tries to change the value of $O$ from locked to free by performing a CAS. If this CAS is successful, the stack is empty, i.e. no other thread has an active request, so the work of the combiner is done and the combiner can return. On the opposite case, the combiner performs a Swap to store in $O$ the value locked and get a pointer to the stack of announced requests. Then, it serves all the requests listed in the stack. After this, it tries again to change the value of $O$ from locked to free using CAS. In the meantime, other newly-activated threads may have created a new stack pointed to by $O$. So, the combiner thread must apply the procedure described above again until it manages to change the value of $O$ from locked to free which would mean that there are no more active requests in the system.

OyamaAlg [52] has several drawbacks. First, the list of announced requests is treated in a LIFO way, so requests are not served in the order they enter it. CC-Synch and DSM-Synch provide a stronger notion of fairness since they serve requests in FIFO order of entering the list. Moreover, in OyamaAlg, the combiner may starve since there may always be newly-activated requests, so the combiner may never manage to change the value of $O$ from locked to free; other threads may also starve when they repeatedly try to insert their record in the list. In our algorithms, the combiner can choose how many requests it will serve and no thread ever starves. Finally, the algorithm in [52] has significant performance overheads in comparison to CC-Synch and DSM-Synch for the following reasons. First, threads need to succeed on a CAS in order to have their requests announced; this causes a lot of contention and leads to a significant performance degradation. Second, the number of RMRs performed by any thread is unbounded since threads may starve as described above.

Flat-combining has been presented by Hendler *et. al* in [34]. As in OyamaAlg, flat-combining employs a global lock that protects the shared data and a list of announced requests; the global lock is again implemented using a CAS object $O$. There are however two main differences between flat-combining and OyamaAlg. First, the list of announced requests usually contains one record for each thread independently of whether it has a currently active request; this reduces the number of insertions in the list. However, it increases the work of the combiner which should now traverse a longer list than necessary. To avoid this extra overhead, the combiner cleanups the list periodically keeping in it only records of threads that have recently initiated a request. A thread that initiates a request starts by checking if its record is in the list; if not, it first tries to insert it (as the first record of the list). Second, the CAS object $O$ is not used to manipulate the head of the list as in OyamaAlg. This results in less overhead since the combiner does not interfere with threads that are trying to insert their records in the list.

Dice, Marathe and Shavit [24] have recently presented an hierarchical spin-lock implementation, called flat-combining NUMA lock; this hierarchical lock is based on flat-combining and exploits the cache hierarchies in order to provide good performance. As it is shown in [24], this lock implementation greatly outperforms the previous (hierarchical and non-hierarchical) spin-lock implementations presented in [23, 46, 47, 49, 53]. H-Synch exhibits a significant performance improvement over the flat-combining NUMA locks [24], since it (1) is simpler, (2) employs combining to serve the thread requests in each cluster, whereas this is not the case in the hierarchical lock presented in [24], and (3) is based on CC-Synch which performs better than flat combining. Our experiments show that not only H-Synch, but also CC-Synch outperforms flat-combining NUMA locks, CC-Synch by a factor of up to 1.65 (Figure 6.2) and H-Synch by a factor of up to 2.65 (Figure 6.2). Our stack and queue implementations based on H-Synch outperform the stack and queue implementations based on flat-combining NUMA locks by similar factors.

CC-Synch implements a combining-friendly version of the CLH queue lock [23, 47]; in contrast to the CLH implementation where the maintained queue is implicit, the queue maintained by CC-Synch is explicit so that the combiner can traverse it. DSM-Synch implements a combining-friendly version of the MCS spin lock [49].

CC-Synch and DSM-Synch provide a stronger notion of fairness than flat-combining since in flat-combining requests, that have been inserted in the list later than other requests, may be served first. In flat combining, the combiner cannot starve but this does not come without an extra performance overhead; specifically, the combiner may choose to return without serving all the active requests in the list, but this makes it necessary to have each active thread checking regularly whether the coarse-grain lock has been released by the combiner and if yes, trying to become a combiner itself instead of performing just local spinning. Finally, flat combining experiences performance overheads in comparison to CC-Synch and DSM-Synch. First, in CC-Synch and DSM-Synch no extra cost is paid for maintaining the list of active requests since this list is provided for free by the implementation of the lock. Second, the cost paid by the combiner is larger in flat combining since the combiner usually has to traverse a longer list than necessary. Finally, threads in flat combining may perform an unbounded number of RMRs when trying to insert their records in the list.

Treiber has presented a lock-free shared stack implementation in [58]. Treiber's shared stack is implemented as a linked list and there is a CAS base object that points to the topmost element. Each thread that wants to Push/Pop an element to the stack repeatedly performs CAS instructions to the CAS object trying to Push/Pop a node to the top of the linked list. The performance of Treiber's stack is significantly enhanced by employing a simple adaptive backoff scheme.

Hendler, Shavit and Yerushalmi [35] use an elimination layer on top of Treiber's lock-free stack. The elimination layer offers the ability to eliminate concurrent Push/Pop operations instead of competing to modify the CAS object that points to the topmost element of the stack. In cases of high contention, this results to a better scaling compared to Treiber's lock-free implementation.

Hendler *et. al* in [34], use an instance of flat-combining in order to implement a scalable blocking shared stack. This results to better performance comparing to the lock-free stack implementation presented in [58] and the elimination scheme presented in [35]. SimStack achieves much better performance than the lock-free stack implementation presented in [58], the elimination algorithm algorithm of [35] and the shared stack based on flat-combining [34] (see Section 5.3). Furthermore, SimStack ensures stronger notion of progress, since it is wait-free. CC-Stack, DSM-Stack and H-Stack offer even better performance but they are blocking.

Michael and Scott [50] present a lock-free implementation of shared queue that is implemented using a linked-list. There are two CAS base objects; the first object points to the head of the queue, while the second one points to the tail of the queue. Threads that want to Enqueue/Dequeue elements to the shared queue repeatedly perform CAS instructions to these CAS objects. In the same paper, Michael and Scott present a blocking alternative of the lock-free queue. Similarly to the lock-free queue, the blocking queue is implemented using a linked-list. Two ordinary locks are used, the first one protects the head of the list and the second protects the tail of the list. This gives the ability to enqueuers and dequeuers to run independently.

Hendler *et. al* [34], uses an instance of flat-combining in order to implement a scalable blocking shared queue implementation, which is called FCQueue. In the experiments presented in [34], it is shown that FCQueue outperforms prior work (i.e the lock-free queue implementation of [50], a queue implementation based on OyamaAlg, etc). SimQueue achieves much better performance than the lock-free queue implementation presented in [50] and the shared queue based on flat-combining [34] (see Section 5.3). Furthermore, SimQueue ensures stronger notion of progress, since it is wait-free. CC-Queue, DSM-Queue and H-Queue offer even better performance but they are blocking.

# Chapter 3

# Model

## 3.1 General

We consider an *asynchronous* system of $n$ *threads*, $p_1, \ldots, p_n$, each of which may fail by *crashing*. In case of thread failure, the thread simply stops executing its algorithm, i.e. it stops taking steps. Threads communicate by accessing shared base objects. Each *shared base object* stores some information and provides operations to threads to read and modify the stored information; these operations may be executed by threads concurrently.

The most basic shared object is a `Read-Write` base object $R$ which stores a value and supports two atomic operations: `Read`($R$) which returns the current value of $R$ and leaves its content unchanged, and `Write`($R, v$) which writes the value $v$ into $R$ and returns an acknowledgment. A base object is *multi-writer* if all threads can change its content; on the contrary, a *single-writer* base object can be modified only by one thread. A base object is *unbounded* if the set of values that can be stored in it is unbounded; otherwise, the base object is *bounded*.

An Add object $O$ supports two atomic operations Read($O$) and Add($O, v$). Operation Add($O, v$) adds the (positive or negative) value $v$ to the value of $O$ and returns an acknowledgment, while operation Read($O$) returns the value stored in $O$.

An LL/SC *object* $O$ supports the atomic operations LL and SC. LL($O$) returns the current value of $O$; the execution of SC($O, v$) by a thread $p$ must follow the execution of LL($O$) by $p$, and changes the contents of $O$ if $O$ has not changed since the execution of $p$'s latest LL on $O$. If SC($O, v$) changes the value of $O$ to $v$, *true* is returned and we say that the SC is successful; otherwise, the value of $O$ does not change, *false* is returned and we say that the SC is not successful or it is failed. Some LL/SC base objects support the operation VL in addition to LL and SC; when executing by some thread $p$, VL returns *true* if no thread has performed a successful SC on $O$ since the execution of $p$'s latest LL on $O$, and *false* otherwise.

A CAS object $O$ supports in addition to operation Read($O$), the atomic operation CAS($O, v_{old}, v_{new}$) which stores $v_{new}$ to $O$ if the current value of $O$ is equal to $v_{old}$ and returns *true*; otherwise the contents of $O$ remain unchanged and *false* is returned. A Swap object stores a value from a set $V$ and in addition to Read($O$), it supports the atomic operation Swap($O, v$), which stores $v$ to $O$ and returns the current value of $O$.

An *active set* is a shared object that identifies the set of threads that participate in some computation; it supports the operations (1) JOIN which is called by a thread to identify its participation to the computation, (2) LEAVE to request removal from the set of participating threads, and (3) GETSET which returns the set of the currently participating threads.

A *collect* object consists of $n$ components $A_1, ..., A_n$, one for each thread, where each component stores a value from some set; it supports the operations (1) UPDATE($v$) which, when executed by $p_i$, it stores the value $v$ in $A_i$, and (2) COLLECT which returns a vector of $n$ values, one for each component.

A *universal* object simulates any other shared object. It supports an operation, called APPLYOP(*operation op*), which simulates the execution of operation *op* on the simulated object; APPLYOP returns the return value of operation *op*.

An *implementation* of a (high-level) object from base objects provides, for each operation of the simulated object and for each thread, an algorithm that uses the base objects to implement the operation. An implementation of a universal object is called *oblivious* if it does not exploit the semantics of the type of the simulated object.

A *configuration* consists of a vector of $n+r$ values, where $r$ is the number of base objects in the system; the first $n$ attributes of this vector describe the state of the threads, and the last $r$ attributes are the values of the $r$ base objects of the system. Thus, a configuration $C$ describes the system at some point in time. At an *initial* configuration, the base objects contain initial values and each thread is in an initial state. A thread completes the execution of a *step* each time it executes an operation on a shared object (i.e. a step consists of a single operation on a base object and possibly some local computation). An *execution* $\alpha$ is a sequence of steps executed by threads. An *execution fragment* of $\alpha$ is a part of $\alpha$ consisting of any number of consecutive steps. A thread starts the execution of a (simulated) operation by invoking its algorithm and finishes it when it gets back a response; thus, for each instance of an operation that is executed in $\alpha$, there is an *invocation* and a *response*. A thread is *active* at some configuration $C$, if it has invoked an operation *op* at $C$ but it has not yet issued a response for *op*; in this case, we also say that *op* is active at $C$. The *execution interval* of *op* is the part of the execution that starts with *op*'s invocation and ends with *op*'s response.

An implementation is *blocking* if a thread may have to wait for some event caused by another thread. An implementation is *wait-free* [36] if in each execution, *each* thread finishes the execution of every operation it initiates within a finite number of steps independently of the speed or the failure of other threads. Wait-freedom is the strongest non-blocking progress property. A weaker such property is *lock-freedom*, which ensures that *some* thread finishes the execution of an operation within a finite number of steps.

For the sake of studying the performance of blocking algorithms, we consider that the system's shared memory is divided into memory chunks. Each memory chunk stores a number of base objects and is associated with some processor. We consider two shared memory models. In cache-less NUMA machines (this model is also known as DSM), a thread $p$ performs a *remote memory reference* (RMR) if it accesses a base object residing in the memory chunk associated with some processor other than $p$; all other memory accesses by $p$ are called *local*. In *cache-coherent* (CC) machines, accesses in shared memory are performed on cached copies of the base objects. In this case, if the cached copy of the base object is invalid, the memory reference is called *remote*. Then, a cache miss occurs and a valid copy of the base object should first be fetched in the local cache before it can be accessed. It is worth pointing out however that once this occurs and as long as the base object is not updated by other processors, all future accesses to the base object by the processor are local. This is not the case in the DSM model, where every access to a base object that resides in a remote memory is *remote*. We remark that since an RMR is significantly more costly than a local memory reference, our goal when designing blocking algorithms, is have them to perform as few RMR as possible.

For the sake of studying the performance of non-blocking algorithms, we define the *step complexity* of an operation, to be the maximum number of steps that a thread performs to complete such an operation in any execution. The *step complexity* of an implementation is the maximum between the step complexities of the operations of the simulated object.

The *interval contention* of an instance of an operation in an execution $\alpha$ is the maximum number of threads that are active in the execution interval of this instance. The *interval contention* of an operation in an execution $\alpha$ is the maximum number of threads that are active in the execution interval of any instance of the operation in $\alpha$. The *total contention* of an execution $\alpha$ is the total number of threads that have taken steps in $\alpha$. If the step complexity of an implementation depends on the interval (or the total) contention of the simulated operations in each execution, and not on the total number of threads $n$, then the implementation is called adaptive.

Let $\alpha$ be any execution. *Linearizability* [39] ensures that, for each operation *op* on the simulated object in $\alpha$, there is some point, called *linearization point*, within the execution interval of *op*, such that *op* appears as if it has executed instantaneously (or as so called *atomically*) at this point; more specifically, the response returned by *op* in $\alpha$ should be the same as the response *op* would return if all operations in $\alpha$ were executing sequentially in the order determined by their linearization points. When this holds, we say that the response of *op* is *consistent*. An implementation is *linearizable* if all its executions are linearizable.

We remark that an implementation of a collect object does not have to be linearizable. In such an implementation, the vector returned by each COLLECT *Col* must contain, for each component $A_i$, the value written by the last UPDATE $U$ on $A_i$ by $p_i$ that has finished its execution before the invocation of *Col* (or the initial value if such an UPDATE does not exist), given that $p_i$ has not started the execution of a new UPDATE $U'$ in the interval between the end of $U$ and the end of $Col$. If $p_i$ has started the execution of a new update $U'$, then $Col$ may return either the value of $U'$ or that of $U$ (or the initial value, if $U$ does not exist) for $A_i$. Moreover, an instance of COLLECT $G$ which has started its execution after the response of some other instance of COLLECT $G'$, must return, for each component $A_i$, either the same value $v$ with that returned by $G$ or some value $v'$ that has been written in $A_i$ by an UPDATE $U'$ which has started its execution after the response of the UPDATE $U$ that writes into $A_i$ the value $v$ read by $G$. A similar correctness property must be ensured by any implementation of an active set. A *snapshot* object is a linearizable version of a collect object (we then use the term SCAN instead of COLLECT). A snapshot implementation is *single-writer*, if each component can be updated only by a specific thread, so it is only this thread that can apply UPDATE operations to the component. In *multi-writer* snapshots, each thread can execute UPDATE operations to any component.

## 3.2 Pseudocode conventions

The code description (pseudocode) of an algorithm for shared memory machines provides pseudocode for every thread. The pseudocode of an algorithm for shared memory machines is similar to the pseudocode of serial algorithms. The pseudocode includes accesses to local variables (thread's local variables consist thread's state) and accesses to shared base objects. Shared base objects are similar to local variables, but the keyword *shared* is inserted in the beginning of their declaration (e.g. the declaration *shared int* $X$; means that base object $X$ is a shared base object of type int, on the contrary *int* $x$; declares a local variable). The first character of shared base object's name is usually capitalized.

For simplicity, instead of using `Read` and `Write` instructions in the pseudocode, the following conventions are used: (i) a reference to the shared base object at the left of an assignment means that a `Write` instruction is executed on the base object, (ii) and a reference to the shared base object at the right of an assignment means that a `Read` instruction is executed on the base object. For example, if $X$ and $Y$ are shared base objects, then the $X = Y$ expression means that the value of $Y$ is read and its value is written to base object $X$. A simple expression of the pseudocode may contain a lot of references to shared base objects. In such a case, the `Read` instructions to shared base objects that are placed at the right of the expression are executed from left to right, the returned values from the `Read` instructions are stored to temporary local variables, the calculation of the expression is executed on the local variables and the result is written to the base object appearing to the left part of the expression. For example, if $X$, $Y$ and $Z$ are shared base objects, then $X = Y + Z$ means that initially the value of $Y$ is read and its value is stored into a temporary local variable, after that $Z$ is read and its value is stored to a local variable, and finally the sum of local variables is written to base object $X$. Comments in the pseudocode start with //.

# Chapter 4

# Adaptive Wait-Free Synchronization Algorithms

In this chapter, the family of RedBlue synchronization algorithms is presented. In Section 4.1, we present F-RedBlue. In Section 4.2, we present S-RedBlue, which is a modified version of F-RedBlue that uses small base objects. Section 4.3 presents two algorithms, LS-RedBlue and BLS-RedBlue. These algorithms combine techniques presented in [11] and the techniques employed by S-RedBlue algorithms in order to achieve the nice properties of the algorithms presented in [11] with better time complexity.

## 4.1 The F-RedBlue algorithm

In this section, we present the first algorithm of the RedBlue family, which is called F-RedBlue. F-RedBlue has time complexity $O(\min\{k, \log n\})$, uses `LL/SC` and read-write base objects, and is optimal in terms of time complexity.

### 4.1.1 Algorithm description

We first describe a relatively simple wait-free algorithm that F-RedBlue is based on. This simple algorithm uses a perfect binary tree (called the *blue tree*) of $\lceil \lg n \rceil + 1$ levels, each node of which is a LL/SC object. We assume that the root node is placed at level $\lceil \lg n \rceil + 1$ and that the leaf nodes are placed at level 1. Each thread $p$ owns one of the tree leaves and it is the only thread capable of modifying this leaf. Thus, there is a unique path $pt(p)$ (called *blue path* for $p$) from the leaf node assigned to $p$ up to the root. The LL/SC object of each node stores an array of $n$ request types (and their parameters), one for each thread to identify the request that the thread is currently executing. The root node additionally stores the state of the simulated object and the return value for the last request applied to the simulated object by each thread. We denote by $\hat{s}$, the initial value of the state field.

Whenever thread $p$ wants to apply a request *req* to the object, it moves up its path until it reaches the root node and ensures that its request *req* is recorded in all nodes of the path by executing two LL/SC on each of them. If any of the LL/SC that $p$ executes on some node succeeds, *req* is successfully recorded in it; otherwise, the algorithm guarantees that *req* is recorded for $p$ in the node by some other thread before the execution of the second of the two SC instructions executed by $p$. In this way, *req* is propagated towards the root where *req* is applied to the object. Besides that, $p$ records in each node the requests that are being executed by other active threads in an effort to help them finish their executions. Successful SC instructions that are executed at the root node may result the application of several requests to the simulated object. In this way, the algorithm guarantees wait-freedom.

Once $p$ ensures that *req* has been applied, it traverses its path from its leaf up to the root once more to eliminate any evidence of its last request by overwriting *req* with the special value $\perp$. This allows $p$ to execute additional requests later on the simulated object. A new request $req'$ by $p$, is applied to the simulated object only if $req'$ is propagated to the root node and the thread $p$ thats stores it, finds that the previous value for $p$ is $\perp$.

This relatively simple algorithm requires $O(\log n)$ steps to execute. In order to make it adaptive, F-RedBlue uses one more tree (the *red tree*), isomorphic to the blue tree (Figure 4.1). Each thread $p$ is assigned a leaf node of the red tree which identifies a

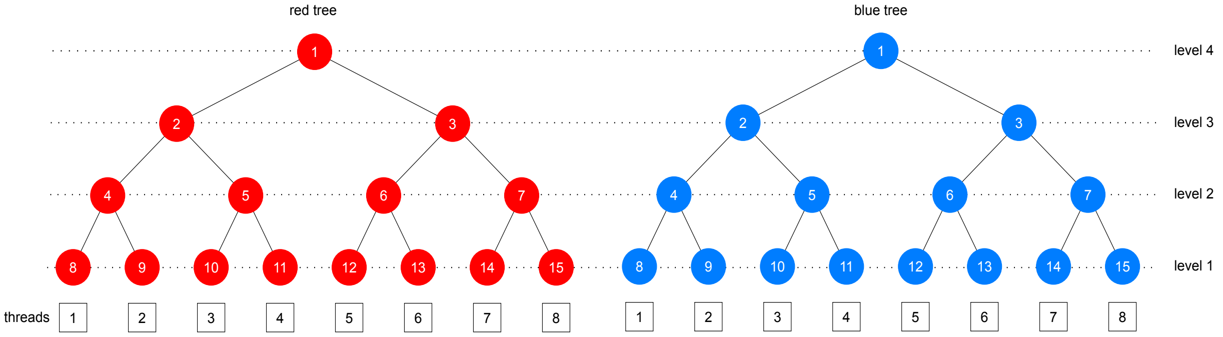


Figure 4.1: The red and the blue tree of F-RedBlue for $n = 8$.

unique path from the root to this leaf (*red path for p*). The red tree allows threads to obtain information about the encountered contention which is then used to shorten the paths that threads traverse in the blue tree (i.e. a thread starts its traversal of its blue path possibly from some internal node of blue the tree (instead of starting from its assigned leaf) which is at a level that depends on the encountered contention).

Each node of the red tree stores information about the request $req$ of a single (active) thread. We then say that $p$ "occupies" the node. More specifically, each thread $p$ first tries to occupy a node of the red tree and then starts traversing (part of) its blue path. In order to occupy a red node, $p$ traverses its red path downwards starting from the root, until it finds a *clean node* (i.e. a currently unoccupied node, such a node stores the value $< \perp, -1 >$) and manages to occupy it by recording its request type and its id in it. We prove that each red node is occupied by at most one thread at any point in time. An occupied node identifies a thread that is currently active, so as long as $p$ reaches occupied nodes, it encounters more contention. We prove that $p$ will eventually reach an unoccupied node and record the appropriate information there. In the worst case, $p$ will occupy its leaf node. Once $p$ occupies some red node with id $z_r$, it performs two traversals of its blue path from the node of the blue tree that corresponds to $z_r$ up to the root. By employing the red tree, threads traverse shorter paths in the blue tree. This improves the time complexity of the algorithm to $O(\min\{k, \log n\})$, where $k$ is the interval contention of $req$.

We continue to provide a more technical description of F-RedBlue (Algorithm 1). Since the blue (red) tree is perfect and there is only one such tree with $\lceil \lg n \rceil + 1$ levels, we implement it using an array $bn$ ($rn$) of $2n-1$ elements. The nodes of the tree are numbered so that node $z$ is stored in $bn[z]$ ($rn[z]$, respectively). The root node is numbered with 1,

```
   struct RedNode{
      request req;
      int pid;
   };
   struct BlueNode{
      state st;                        // this field is used only at the root node
      RetVal rvals[n];                 // this field is used only at the root node
      request reqs[n];
   };

   shared RedNode rn[1..2n-1] = {$< \perp, -1 >, ..., < \perp, -1 >$};
   shared BlueNode bn[1..2n-1]={<$\hat{s}$,$< 0, ..., 0 >$,<$\perp$,...,$\perp$>>, ..., <$\hat{s}$,$< 0, ..., 0 >$,<$\perp$,...,$\perp$>>};

   RetVal F-RedBlue(request req){      // pseudocode for thread $p$
      int direction = n/2, levels = lg(n) + 1;
      int z = 1, l, j;
      RetVals rv;

1     for (l=levels; l $\geq$ 1; l--) {    // traversal of red path from the root node
2        LL(rn[z]);
3        if(rn[z] == $< \perp, -1 >$)
4           if(SC(rn[z], <req, p>)) break;
5        if (p $\leq$ direction) {        // find the next node in the path
6           direction = direction - $2^{l-3}$;
7           z = 2 * z;                 // move to the left child of $z$
8        } else{
9           direction = direction + $2^{l-3}$;
10          z = 2 * z + 1;             // move to the right child of $z$
         }
      }
11    Propagate(z, p);                 // first traversal of blue path: propagating the request
12    rv = bn[1].vals[p];
13    LL(rn[z]);
14    SC(rn[z], $< \perp$, p>);           // the request occupying $rn[z]$ starts its deletion phase
15    Propagate(z, p);                 // second traversal of blue path: propagating $\perp$
16    LL(rn[z]);
17    SC(rn[z], $< \perp$, -1>);          // re-initialize the occupied red node to $\perp$
18    return rv;                       // return the appropriate value
   }
```

Algorithm 1: Pseudocode for F-RedBlue.

and the left and right children of any node $z$ are nodes $2z$ and $2z + 1$, respectively. Red and blue trees for $n = 8$ are illustrated in Figure 4.1. Thread $p$, $1 \leq p \leq n$, is assigned to the leaf node numbered $n + p - 1$. We remark that traversing up the path from any node $z$ to the root can be implemented in a straightforward manner: the next node of $z$ in the path is node numbered $\lfloor z/2 \rfloor$. However, the downward traversal of the path requires some more calculations which are accomplished by the lines $5 - 10$ of the pseudocode.

When a thread $p$ wants to execute a request $req$, it first traverses its red path (lines $1 - 10$). For each node $z$ of this path, it checks if the node is unoccupied (line 3) and if this is so, it applies an SC instruction to it in an effort to occupy it (line 4). If the SC is

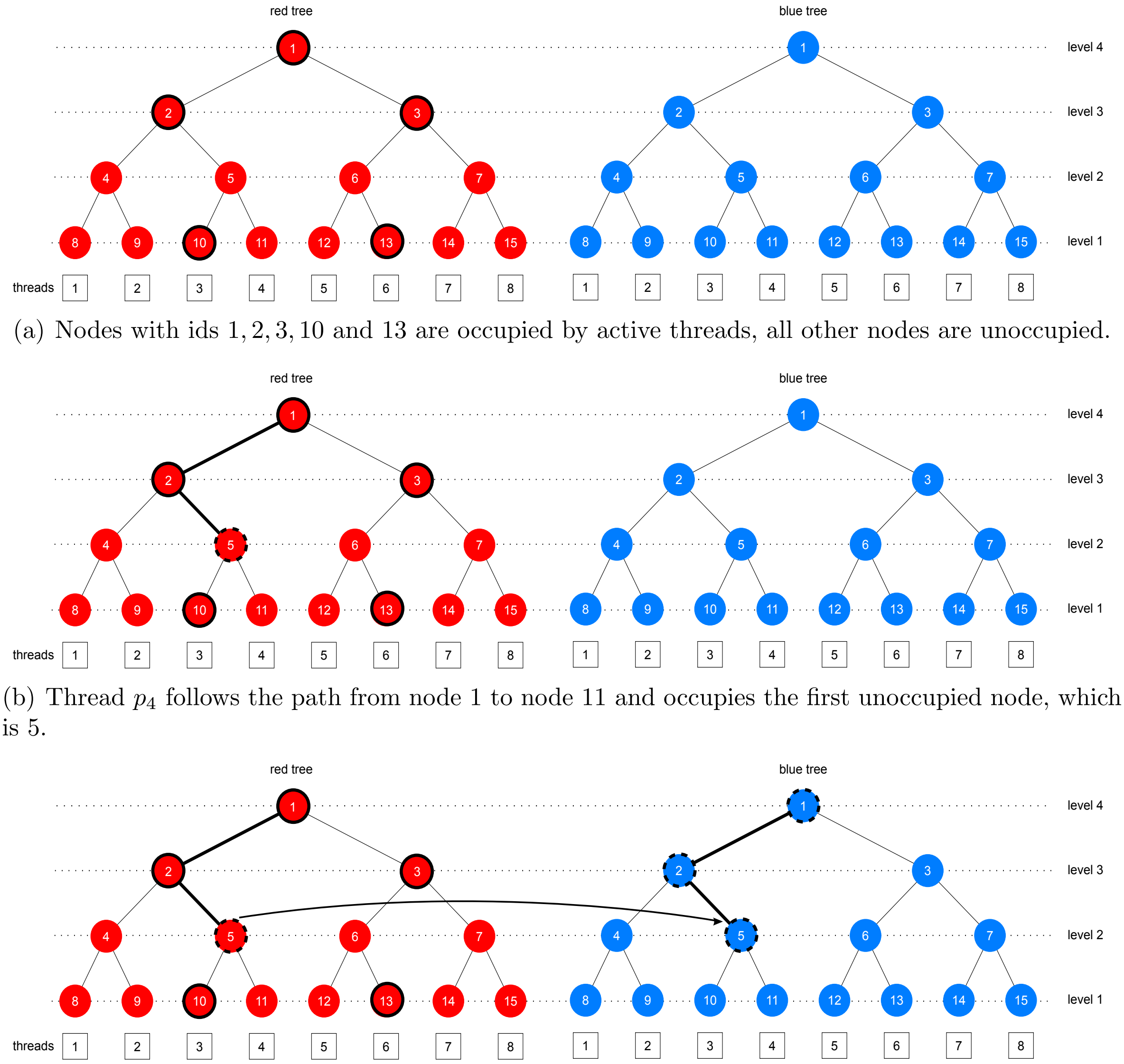
```
    void Propagate(int $z$){
       BlueNode bt;
19     while (z!=0){                          // traversal of the blue path
20        for (j=1 to 2) do{                  // two efforts to store appropriate information at each node
21           LL(bn[z]);
22           bt=Calculate(z);
23           SC(z, bt);
          }
24        z =$\lfloor$z/2$\rfloor$;
       }
    }

    BlueNode Calculate(int z) {
       BlueNode tmp=$< \bot, < 0, \ldots, 0 >, < \bot, \ldots, \bot >>$;
       BlueNode blue=bn[z], lc, rc;
       RedNode red = rn[z];
       int q, range;
25     if (2*z+1 < 2n) {                      // in case that $z$ is not a leaf node
26        lc = bn[2*z];                       // read the contents of the left child in the blue tree
27        rc = bn[2*z+1];                     // read the contents of the right child in the blue tree
28        range = $2^{\lg(n)-\lceil \lg(z) \rceil}$;         // compute the number of leaves that each subtree has
          // copy the requests placed on the left subtree
29        tmp.reqs[2q-range..2q] = lc.reqs[2q-range..2q];
          // copy the requests placed on the right subtree
30        tmp.reqs[2q+1..2q+1+range] = rc.reqs[2q+1..2q+1+range];
       }
31     if (red.pid $\neq$ -1) tmp.reqs[red.pid]=red.req; // if thread $q$ occupies a red node
32     if (z == 1) {
33        tmp.rvals[1..n] = blue.rvals[1..n]; // copy the return values
34        tmp.st = blue.st;                   // copy object's state
35        for q=1 to n do{                    // local loop
             // apply any pending request
36           if (tmp.reqs[q] $\neq \bot$ AND tmp.reqs[q] $\neq$ blue.reqs[q])
37              apply tmp.reqs[q] to tmp.st and store into tmp.rvals[q] the return value;
          }
       }
38     return tmp;                            // a blue node is returned
    }
```

Algorithm 2: Pseudocode for Calculate and Propagate of F-RedBlue.

successful, the traversal of the red path ends (line 4). Otherwise, the next node in the path is calculated (lines $5-10$) and one more iteration of the loop is performed.

Once a red node $z_r$ has been occupied, *req* performs two traversals of (a part of) its blue path starting from the node in the blue tree corresponding ro $z_r$, up to the root. This is accomplished by the two calls to Propagate (lines 11 and 15). Each of these traversals propagates the request type written into $z_r$ to the root node. Notice that $p$ records $\bot$, as its request type, into $z_r$ (lines $13-14$) before it starts its second traversal (we remark that



(a) Nodes with ids $1, 2, 3, 10$ and $13$ are occupied by active threads, all other nodes are unoccupied.

(b) Thread $p_4$ follows the path from node 1 to node 11 and occupies the first unoccupied node, which is 5.

(c) Thread $p_4$ traverses the path from its occupied node 5 to the root node.

Figure 4.2: An example of an execution of F-RedBlue, where thread $p_4$ applies an operation to the simulated object.

this occurs by performing one more LL/SC since we assume that an LL/SC base object supports only Read, LL, and SC and not Write).

On each node $z$ of the traversed path, Propagate performs twice the following: (1) an LL instruction on $z$ (line 21); (2) calculates the appropriate information to write into $z$ by calling function Calculate (line 22); (3) an SC to store the result of Calculate into $z$ (line 23). Finally, it moves up to the next node of the blue path (line 24). Thread $p$ re-initializes its occupied red node by writing in it the value $(\perp, -1)$ (lines $16 - 17$) just before it returns.

Figure 4.2 shows an example of the application of some operation by thread $p_4$ in a system of 8 threads. Assume that $p_4$ wants to apply some operation *op* and assume that some nodes of the red tree are already occupied. More specifically, red nodes with black circles are occupied by active threads (see Figure 4.2(a)). At first, $p_4$ traverses the path from the root node to its leaf until it occupies a red node (see Figure 4.2(b)). The first unoccupied node from the root node to the leaf of $p_4$, is node 5. After occupying this node, $p_4$ continues its execution with the traversal of the blue path. Now, $p_4$ traverses the path from the occupied node, which is 5, to the root node upwards (see Figure 4.2(c)). In each node of this path, $p$ tries to store the values read in both child-nodes and also the value that is stored to the isomorphic red node.

We finally discuss the details of function `Calculate`. Function `Calculate` applies a (potentially new) request for each thread $q$ (lines $35-37$) as described below. If $q$ occupies node with id $z$ (line 31), then $q$'s new request is the one which is recorded into the red node. Otherwise, the request for $q$ is found in the previous node of that with id $z$ in $q$'s blue path. In case $z$ is the root node ($z = 1$) and the calculated request for $q$ is not already written in z and it is different than $\perp$ (line 36), then the request of $q$ is a new one and should be applied to the simulated object (line 37). This is simulated by calling function `apply`.

### 4.1.2 Correctness proof

In this section we prove the correctness of F-RedBlue and analyze its time and space complexity. In order to prove that F-RedBlue is correct, we first study the execution fragment of a request *req* executed by thread $p$, where $p$ traverses the red tree. Intuitively, we prove that *req* manages to occupy exactly one red node, and as long as *req* is executed, no other request succeeds in occupying this red node. After this point and once *req* finishes its execution, it stores into this red node its initial value in order to allow its re-occupation by some other request. We then study the properties of the execution fragment of *req* that traverses the blue tree. We prove that if *req* occupies a red node with id $z_r$, *req* will be recorded into all nodes of the path starting from the blue node with id $z_r$ up to the blue root. Therefore, *req* eventually reaches the root node and it is applied to the object. We also prove that the application of each request occurs only once and that the calculated response values are correct.

We first study the properties of the execution fragment of an F-RedBlue request *req* that traverses the red tree. Intuitively, we prove that *req* manages to occupy a red node (Lemma 4.1) from that point and as long as *req* is executed, no other request succeeds in updating this red node (Lemmas 4.2 and 4.4). Once *req* finishes its execution, it stores into its red node its initial value in order to allow its re-occupation by some other request.

Call the SC instructions of line 4 SC of type 1, the SC instructions of line 14 SC of type 2, and the SC instructions of line 17 SC of type 3. Let $p$ be any thread. By the pseudocode, only $p$ executes `SC` instructions on $rn[n-1+p]$, the red leaf associated to $p$. Thus, all these instructions succeed. Therefore, the condition of line 3 of the pseudocode is evaluated to *true* when executed on $rn[n-1+p]$ and, by the pseudocode, the sequence of SC executed on $rn[n-1+p]$ alternates between SC of type 1, SC of type 2, and SC of type 3.

**Observation 4.1.** *Let $p$ be any thread. Then,*

1. *only thread $p$ executes SC instructions on base object $rn[n-1+p]$;*
2. *the condition of line 3, when executed on $rn[n-1+p]$, is evaluated to* true;
3. *all SC on $rn[n-1+p]$ succeed;*
4. *the sequence of SC executed on $rn[n-1+p]$ alternates between SC of type 1, SC of type 2, and SC of type 3.*

Based on Observation 4.1, it is easy to prove that any request *req*, executed by some thread $p$, performs a successful `SC` of type 1 at some node of the red tree, since, in the worst case, this will occur at $p$'s red leaf, the last node of $p$'s red path.

**Lemma 4.1.** *Any instance req of* ApplyOp *executes a successful SC instruction of type 1 at some node of the red tree.*

*Proof.* Let $p$ be the thread that executes *req*. Assume, by the way of contradiction, that *req* does not execute a successful SC of type 1 on any node of the red tree. Then, by the pseudocode (lines 1-10), *req* executes an `SC` instruction of type 1 on any node of the red path of $p$. Since the last node in this path is $rn[n-1+p]$, it follows that the SC executed by $p$ on $rn[n-1+p]$ is not successful, which contradicts Observation 4.1 (Claim 3). ∎

Let $z$, $1 \leq z \leq 2n-1$, be the id of a node of the red tree. For any $j \geq 1$, denote by $SC_1^j(z)$ the $j$-th successful SC of type 1 executed on $z$ (i.e. on the node with id $z$), and let $req_j(z)$ be the request that executes $SC_1^j(z)$. We often abuse notation and omit $z$, whenever it is clear from the context. Notice that, by definition, there are no successful SC instructions of type 1 between $SC_1^j$ and $SC_1^{j+1}$.

We say that a red node with id $z$ *is occupied* by a thread $p$ at some configuration $C$, if it holds that $rn[z].pid = p$ at $C$. If $p$ is executing request $req$ at $C$, we also say that $z$ is occupied by $req$ at $C$. We continue to prove that each red node, occupied by some request $req$, should first be released by $req$ before it can be occupied again by some other request.

**Lemma 4.2.** *For each $j \geq 1$, $req_j$ executes a successful* SC *instruction of type 3, which we denote by $SC_3^j$, on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$.*

*Proof.* First, we prove that at least one successful SC of type 3 is executed on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$. We let $SC_3^j$ be the first of these successful SC instructions. Then, we prove, by induction on $j$, that $SC_3^j$ is executed by $req_j$ (i.e. the same request that executes $SC_1^j$).

1. Assume, by the way of contradiction, that no successful SC of type 3 is executed on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$. Recall that $req_j$ is the request that executes $SC_1^j$ and $req_{j+1}$ is the request that executes $SC_1^{j+1}$.

   First, we prove that the read of $rn[z]$ (line 3) by $req_{j+1}$ follows $SC_1^j$. Assume, by the way of contradiction, that this read occurs before $SC_1^j$. The execution of the $LL$ of line 2 by $req_{j+1}$ precedes this read, so the execution of this $LL$ occurs before $SC_1^j$. Since the corresponding SC to this $LL$ is $SC_1^{j+1}$, and occurs after the successful $SC_1^j$ instruction, $SC_1^{j+1}$ cannot be successful, which is a contradiction. Therefore, the read at line 3 by $req_{j+1}$ follows the execution of $SC_1^j$.

   Since $req_{j+1}$ executes $SC_1^{j+1}$, by the pseudocode (lines $3-4$), it follows that $req_{j+1}$ has read $-1$ into $rn[z].pid$ (line 3). Let $SC_x$ be the last successful SC on $rn[z]$ preceding $SC_1^{j+1}$. Recall that we have assumed that no successful SC of type 3 is executed on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$. Moreover, by definition of $SC_1^j$ and $SC_1^{j+1}$, no successful SC of type 1 occurs between $SC_1^j$ and $SC_1^{j+1}$. Thus, $SC_x$ must be either $SC_1^j$ or some successful SC of type 2. In either case, it follows by the pseudocode (lines 4, 14), that $SC_x$ writes a value different than $-1$ into $rn[z].pid$, which is a contradiction. Thus, there is at least one successful SC of type 3 executed on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$.

Let $SC_3^j$ be the first successful SC of type 3 executed on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$.

2. We prove, by induction on $j$, that $SC_3^j$ is executed by $req_j$. Fix any $j \geq 1$ and assume that the claim holds for any $j', 1 \leq j' < j$.

   We prove that the claim also holds for $j$. Assume, by the way of contradiction, that $SC_3^j$ is executed by some request $req \neq req_j$. By the pseudocode (lines 4, 14) and by Lemma 4.1, $req$ executes a successful SC instruction of type 1 on some node of the red tree before $SC_3^j$; let $SC_1$ be this SC instruction. By the pseudocode (lines 4, 17), $SC_1$ and $SC_3^j$ are executed on the same node, namely on node $z$.

   By the definitions of $SC_1^j$ and $SC_1^{j+1}$, no other successful SC of type 1 is executed on $z$ between $SC_1^j$ and $SC_1^{j+1}$. Moreover, $SC_1^j \neq SC_1$ since $SC_1^j$ is executed by $req_j \neq req$. Thus, $SC_1$ is executed before $SC_1^j$.

   If $j = 1$, this is a contradiction, since, by definition, $SC_1^j$ is the first successful SC of type 1 on $rn[z]$. If $j > 1$, let $SC_1'$ be the first successful SC of type 1 on $rn[z]$ following $SC_1$. Notice that $SC_1'$ is either $SC_1^j$ or some earlier successful SC of type 1 on $z$. As proved above (item 1), there is at least one successful SC of type 3 executed on $rn[z]$ between $SC_1$ and $SC_1'$. Let $SC_3$ be the first such SC; obviously, $SC_3$ precedes $SC_3^j$. Then, by the induction hypothesis, $SC_3$ is executed by $req$. By the pseudocode, $req$ executes only one SC of type 3, which contradicts the fact that $req$ executes both $SC_3$ and $SC_3^j$. ∎

We continue to prove that the SC instructions executed on $rn[z]$ by any request $req \neq req_j$ between $SC_1^j$ and $SC_1^{j+1}$ fail.

**Lemma 4.3.** *Let $req \neq req_j$ be any request. Then, no successful* SC *is executed by $req$ on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$.*

*Proof.* By definition, no successful SC of type 1 is executed between $SC_1^j$ and $SC_1^{j+1}$. Assume, by the way of contradiction, that *req* executes a successful SC of type 2 or 3 on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$. Let $SC_h$ be the first of these successful SC instructions.

Lemma 4.1 implies that *req* executes a successful SC of type 1 on some node of the red tree before $SC'$; let $SC_1$ be this instruction. By the pseudocode (lines 4, 17), $SC_1$ is executed on the same node as $SC_h$, namely on node $rn[z]$. Since $req \neq req_j$ and $SC_1^j$ is executed by $req_j$, $SC_1 \neq SC_1^j$. Since no successful SC of type 1 is executed between $SC_1^j$ and $SC_1^{j+1}$, it follows that $SC_1$ is executed before $SC_1^j$.

If $j = 1$, this is a contradiction since, by definition, $SC_1^j$ is the first successful SC of type 1 on $rn[z]$. If $j > 1$, let $SC_1'$ be the first successful SC of type 1 on $rn[z]$ following $SC_1$. Then, $SC_1'$ is either $SC_1^j$ or some earlier successful SC of type 1 on $rn[z]$. Lemma 4.2 implies that *req* executes a successful SC of type 3 on $rn[z]$ between $SC_1$ and $SC_1'$; denote by $SC_3$ this SC instruction. Then, $SC_3$ precedes $SC_h$, so $SC_3 \neq SC_h$. By the pseudocode, $SC_3$ is the only SC of type 3 executed by *req*. Moreover, by the pseudocode, *req* executes only one SC of type 2 and it does so between $SC_1$ and $SC_3$; let $SC_2$ be this SC instruction. Then, $SC_2 \neq SC_h$. It follows that $SC_h$ cannot be an SC of the type 2 or of type 3 executed by *req*. This contradicts our assumption. ∎

Recall that, by the pseudocode, *req* executes exactly one SC of type 3. Thus, Lemmas 4.2 and 4.3 immediately imply the following observation.

**Observation 4.2.** *For each $j \geq 1$, $SC_3^j$, executed by $req_j$, is the only successful* SC *of type* 3 *executed on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$.*

It is now easy to prove that between any successful SC of type 1 and the following successful SC of type 3 on $rn[z]$, there is exactly one successful SC of type 2 on $rn[z]$ executed by the same thread.

**Lemma 4.4.** *For each $j \geq 1$, there is exactly one successful SC of type* 2*, namely $SC_2^j$, on the red node with id $z$ between $SC_1^j$ and $SC_1^{j+1}$, and $SC_2^j$ is executed by $req_j$ between $SC_1^j$ and $SC_3^j$.*

*Proof.* By Observation 4.2, $SC_3^j$ is executed by $req_j$. By the pseudocode (line 14), $req_j$ executes exactly one SC of type 2, namely $SC_2^j$, and this happens between $SC_1^j$ and $SC_3^j$. Moreover, the only SC of type 3 executed by $req_j$ is $SC_3^j$. Let $LL_2^j$ be the matching $LL$

instruction to $SC_2^j$. By the pseudocode (lines 4, 13, 14), it follows that $LL_2^j$ is executed after $SC_1^j$. Lemma 4.3 implies that no successful `SC` is executed on $rn[z]$ by any request $req \neq req_j$ between $SC_1^j$ and $SC_1^{j+1}$. It follows that $SC_2^j$ succeeds and it is the only successful `SC` of type 2 executed on $rn[z]$ between $SC_1^j$ and $SC_1^{j+1}$. Since in addition $SC_2^j$ is executed by $req_j$ between $SC_1^j$ and $SC_3^j$, the claim follows. ∎

Lemmas 4.2 and 4.4 and the pseudocode (line 4) imply that each request *req* occupies exactly one red node during its execution. We denote by $z_r(req)$ the id of this red node; whenever *req* is clear from the context, we abuse notation and use $z_r$ instead of $z_r(req)$.

**Observation 4.3.** *The following claims hold: (1) Assume that $C$ is some configuration at which a thread $p$, performing some request req at $C$, has executed the type 1 SC instruction of req but it has not yet executed its type 3 SC instruction. Then, there exists exactly one integer $z_r$, $1 \leq z_r \leq 2 * n - 1$, such that $p$ occupies the red node with id $z_r$ at $C$. (2) Assume that $C$ is some configuration at which a thread $p$ does not execute any request. Then, for each integer $z$, $1 \leq z \leq 2*n-1$, $p$ does not occupy the red node with id $z$ at $C$.*

We continue to study the properties of the execution fragment of a request *req* executed by thread $p$, where $p$ traverses the blue tree. Intuitively, we prove that for each request *req* that occupies a red node with id $z_r$; *req* will be recorded into all nodes of the path starting from the blue node with id $z_r$ up to the blue root (Lemma 4.5). Therefore, *req* is eventually recorded into the root node of the blue tree.

Consider any integer $z$, $1 \leq z \leq 2n-1$, and let $level(z) = lg(n) - \lfloor lg(z) \rfloor + 1$, i.e. $level(z)$ is the level of the node with id $z$ in any of the trees. For the rest of this section, let *req* be any instance of ApplyOp executed by some thread $p$. By Lemma 4.1, *req* executes a successful SC of type 1 on some node with id $z_r(req)$ of the red tree. By the pseudocode (line 4), this is the only SC of type 1 executed by *req*. Let $pt(req)$ be the path of the blue tree from the node with id $z_r$ to the root. For each $h, level(z_r) \leq h \leq lg(n)+1$, denote by $z_h$ the id of the node of $pt(req)$ at level $h$; notice that when $h = level(z_r)$, $z_h = z_r$. The following observation is an immediate consequence of the pseudocode (lines 20, 23).

**Observation 4.4.** *Let $\pi$ be the execution of any instance of* `Propagate` *by req. Then, $\pi$ executes two SC instructions on every node of $pt(req)$.*

Let $\pi_1(req)$ and $\pi_2(req)$ be the two instances of function `Propagate` executed by *req*, in order. By Observation 4.4, for each $i \in \{1, 2\}$, $\pi_i(req)$ executes two SC instructions

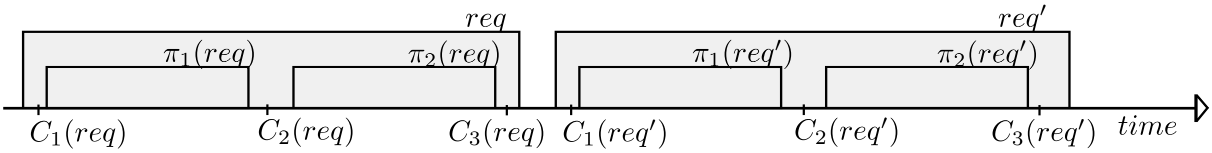


Figure 4.3: An example of an execution of F-RedBlue.

on each node of $pt(req)$. For each $h$, $level(z_r) \leq h \leq \log n + 1$, denote by $C'_{i,h}(req)$ the configuration immediately following the execution of the second of these SC instructions (line 23) on node $bn[z_h]$ by $\pi_i(req)$. Let $reqt[1] = req$ and $reqt[2] = \perp$. Denote by $C_1(req)$ the configuration just after the successful `SC` of type 1 by $req$ (that writes the pair $< req, p >$ into $rn[z_r]$), let $C_2(req)$ be the configuration just after the successful `SC` of type 3 by $req$ (that writes the pair $< \perp, p >$ into $rn[z_r]$) and let $C_3(req)$ be the configuration just after the successful `SC` of type 3 by $req$ (that writes the pair $< \perp, -1 >$ into $rn[z_r]$). In case $h = level(z_r)$, let $C_{1,h-1}(req) = C_1(req)$ and $C_{2,h-1}(req) = C_2(req)$. For simplicity of presentation, we sometimes omit $req$ from the above notation if it is clear from the context. An example of the above notation is shown on Figure 4.3.

**Lemma 4.5.** *For each $i \in \{1, 2\}$, and for each $h$, $level(z_r) \leq h \leq \log n + 1$, there is a configuration $C_{i,h}(req)$ such that:*

1. *$C_{1,h}(req)$ is the first configuration at which req is contained in $bn[z_h].reqs[p]$;*

2. *$C_{2,h}(req)$ is the first configuration following $C_{2,h-1}(req)$ at which $\perp$ is returned;*

3. *$C_{i,h}(req)$ follows $C_{i,h-1}(req)$ and comes before or at $C'_{i,h}(req)$;*

4. *at each configuration between $C_{1,h}(req)$ and $C_{2,h}(req)$, $bn[z_h].reqs[p]$ contains req;*

5. *at each configuration between $C_{2,h}(req)$ and $C_3(req)$, $bn[z_h].reqs[p] = \perp$.*

*Proof.* The proof is by induction on $h$. Fix any integer $h$, $level(z_r) \leq h \leq \log n + 1$ and assume that the claim holds for each $h', level(z_r) \leq h' < h$. We prove that the claims hold for $h$.

Fix any $i \in \{1, 2\}$. Recall that $p$ is the thread executing $req$. By Observation 4.4, $\pi_i$ executes two SC instructions on the blue node with id $z_h$. Let $SC_{i,1}$, $SC_{i,2}$ be these two SC instructions and let $LL_{i,1}$, $LL_{i,2}$ be the matching $LL$ instructions, respectively. By the pseudocode, $\pi_i$ reads $rn[z_h]$ and $bn[z_{h-1}]$ during the execution of any of the instances of its `Calculate`. We prove that $req$ calculates the value $reqt[i]$ as the new value of $bn[z_h].reqs[p]$.

Assume first that $h = level(z_r)$. By definitions of $z_r$ and $C_{i,h-1}$ (when $h = level(z_r)$), and by the pseudocode (lines 4, 14), the pair $(reqt[i], p)$ was written into $rn[z_r]$ at $C_{i,h-1}$, and therefore before the execution of $LL_{i,1}$ and $LL_{i,2}$. By the pseudocode, and by Lemmas 4.2 and 4.4, it follows that $rn[z_r]$ contains the pair $(reqt[i], p)$ until the execution by $req$ of the SC of type $(i+1)$ which comes after the final configuration of $\pi_i$. Therefore, each of the reads of $rn[z_r]$ by $\pi_i$ following $LL_{i,1}$ and $LL_{i,2}$ returns $reqt[i]$ for thread $p$ and, by the pseudocode (lines $27-28$), `Calculate` writes $reqt[i]$ for $p$ in the new set of request types it calculates.

Assume now that $h > level(z_r)$, so $z_h \neq z_r$. Then, Observation 4.3 (Claim 1) implies that $rn[z_h].pid \neq p$. By the pseudocode (lines $29-30$), it follows that `Calculate` will consider, as the new request type for $p$, the value read for $p$ in $bn[z_{h-1}]$. By the induction hypothesis, there is a configuration $C_{i,h-1}(req)$ in which $reqt[i]$ is written in $bn[z_{h-1}].reqs[p]$, and $C_{i,h-1}(req)$ precedes $C'_{i,h-1}$. We argue that request $reqt[i]$ is contained into $bn[z_{h-1}].reqs[p]$ from $C_{i,h-1}(req)$ until the execution of the SC instruction of type $(i+1)$ by $req$ (which occurs after the final configuration of $\pi_i$). If $i = 1$, this is implied by the induction hypothesis (Claim 2) and because $C_{2,h-1}(req) = C_2(req)$ when $h = level(z_r)$. If $i = 2$ this is immediate from the induction hypothesis (Claim 3). By the definition of $C'_{i,h-1}$, and by the pseudocode, it follows that $LL_{i,1}$ and $LL_{i,2}$ occur between $C'_{i,h-1}$ and the final configuration of $\pi_i$. Thus, when $bn[z_{h-1}]$ is read between $LL_{i,1}$ and $SC_{i,1}$ (or $LL_{i,2}$ and $SC_{i,2}$), $reqt[i]$ is found in $bn[z_{h-1}].reqs[p]$.

If any of the $SC_{i,1}$ or $SC_{i,2}$ is successful, then $reqt[i]$ is recorded into $bn[z_h].reqs[p]$ by this successful SC.

Assume now that both $SC_{i,1}$ and $SC_{i,2}$ fail. Since $SC_{i,2}$ fails, it must be that, between $LL_{i,2}$ and $SC_{i,2}$ there is at least one successful SC instruction on $bn[z_h]$. Let $SC'_{i,2}$ be the first of these instructions, and let $req'_i \neq req$ be the request that executes $SC'_{i,2}$. Let $LL'_{i,2}$ be the matching $LL$ instruction to $SC'_{i,2}$. Since $SC_{i,1}$ fails, it must be that between $LL_{i,1}$ and $SC_{i,1}$, there is at least one successful SC instruction on $bn[z_h]$; let $SC'_{i,1}$ be any of them. $LL'_{i,2}$ follows $LL_{i,1}$, since otherwise $SC'_{i,1}$, which follows $LL_{i,1}$, would cause $SC'_{i,2}$ to fail.

The pseudocode implies that $req'_i$ reads $rn[z_h]$ and $bn[z_{h-1}]$ during the execution of its `Calculate` between $LL'_{i,2}$ and $SC'_{i,2}$. Recall that $req$ occupies $z_r$. We prove that request $req'_i$ calculates the value $reqt[i]$ as the new value of $bn[z_h].reqs[p]$.

Assume first that $h = level(z_r)$, so $z_h = z_r$. Recall that the pair $(reqt[i], p)$ was written into $rn[z_r]$ before the execution of $LL_{i,1}$ and $LL_{i,2}$; moreover, $rn[z_r]$ contains the pair $(reqt[i], p)$ until the execution of the SC instruction of type $(i+1)$ by $req$ which comes after the final configuration of $\pi_i$. By the pseudocode (line 20), $req'_i$ reads $rn[z_r]$ after $LL'_{i,2}$ (which follows $LL_{i,1}$) and before $SC'_{i,2}$ (which precedes $SC_{i,2}$ and the final configuration of $\pi_i$) . Thus, $req'_i$ reads the pair $(reqt[i], p)$ in $rn[z_r]$. So, by the pseudocode (lines $27-28$), the instance of `Calculate` executed by $req'_i$ between $LL'_{i,2}$ and $SC'_{i,2}$, stores $reqt[i]$ for $p$ in the new set of request types it calculates and $req'_i$ writes $reqt[i]$ into $bn[z_h].reqs[p]$ when it executes $SC'_{i,2}$.

Assume now that $h > level(z_r)$, so $z_h \neq z_r$. Then, Observation 4.3 (claim 1) implies that $rn[z_h].pid \neq p$. By the pseudocode (lines $28-30$), it follows that the instance of `Calculate` executed by $req'_i$ between $LL'_{i,2}$ and $SC'_{i,2}$, will consider as the new request type of $p$ the value read in $bn[z_{h-1}].reqs[p]$. By the induction hypothesis, there is a configuration $C_{i,h-1}$ in which $reqt[i]$ is written into $bn[z_{h-1}].reqs[p]$, and $C_{i,h-1}$ precedes $C'_{i,h-1}$; moreover, $reqt[1]$ is contained in $bn[z_{h-1}]$ from $C_{1,h-1}$ until $C_{2,h-1}$ (which follows the final configuration of $\pi_1$), and $reqt[2]$ is contained in $bn[z_{h-1}]$ from $C_{2,h-1}$ until the execution of the SC instruction of type $(i+1)$ by $req$ (which follows the final configuration of $\pi_2$). Thus, in either case $reqt[i]$ is contained in $bn[z_{h-1}].reqs[p]$ from $C_{i,h-1}$ until the final configuration of $\pi_i$. By the definition of $C'_{i,h-1}$, and by the pseudocode, it follows that $LL_{i,1}$ and $SC_{i,2}$ occur between $C'_{i,h-1}$ and the final configuration of $\pi_i$. Thus, when $bn[z_{h-1}]$ is read between $LL'_{i,2}$ (which follows $LL_{i,1}$) and $SC'_{i,2}$ (which precedes $SC_{i,2}$), $reqt[i]$ is found in $bn[z_{h-1}].reqs[p]$. So, by the pseudocode (lines $27-28$), $reqt[i]$ i stored for $p$ in the new set of request types calculated by $req'_i$ and is written into $bn[z_h].reqs[p]$ by $SC'_{i,2}$.

In both cases, we conclude that there is at least one configuration between $SC_{i,1}$ and $SC_{i,2}$ at which the value $reqt[i]$ is written into $bn[z_h].reqs[p]$.

Let $SC_{i,h}(req)$ be the first successful SC that writes $reqt[i]$ into $bn[z_h].reqs[p]$ and follows $C_{i,h-1}(req)$; let $C_{i,h}(req)$ be the configuration immediately following the execution of $SC_{i,h}(req)$. Therefore, it follows that $SC_{i,h}(req)$ precedes $SC_{i,2}$. (We remark that

$SC_{i,h}(req)$ may also precede $SC_{i,1}$.) Since, by definition, $C'_{i,h}$ follows $SC_{i,2}$, $SC_{i,h}(req)$ precedes $C'_{i,h}$. Thus, $C_{i,h}(req)$ comes before or at $C'_{i,h}$. By definition, $SC_{i,h}(req)$ follows $C_{i,h-1}(req)$, so $C_{i,h}(req)$ follows $C_{i,h-1}(req)$. This concludes the proof of Claim 3.

We continue to prove Claim 1. We argue that the first time that $reqt[1]$ appears in $bn[z_{h-1}].reqs[p]$ is at $C_{i,h-1}$. If $h = lever(z_r)$, this is implied by the pseudocode and by the fact that $C_{1,h-1}(req) = C_1(req)$ in this case. If $h > level(z_r)$, this is implied by the induction hypothesis (Claim 1); moreover, in this case, Observation 4.3 implies that $rn[z_h].pid \neq p$ at all configurations starting from $C_1(req)$ until $C_3(req)$ (which comes after $C_{i,h-1}(req)$). Thus, by the pseudocode, it follows that no SC can write $reqt[1]$ into $bn[z_h].reqs[p]$ before $C_{i,h-1}(req)$. Since (by definition) $SC_{1,h}(req)$ is the first successful SC that writes $reqt[1]$ into $bn[z_h].reqs[p]$ after $C_{i,h-1}(req)$, $C_{1,h}(req)$ is the first configuration at which $reqt[1]$ is contained in $bn[z_h].reqs[p]$.

By Claim 3 proved above, $C_{1,h}$ precedes the final configuration of $\pi_1$; moreover, $C_{2,h}$ follows $C_2(req)$ which comes after the final configuration of $\pi_1$. Thus, $C_{2,h}$ follows $C_{1,h}$.

Assume, by the way of contradiction, that there is a configuration $C_l$ between $C_{1,h}$ and $C_{2,h}$, such that $bn[z_h].reqs[p]$ contains a value $x \neq reqt[1]$ at $C_l$. By definition of $C_l$, there is at least a successful SC instruction that writes $x$ into $bn[z_h].reqs[p]$ and occurs between $C_{1,h}$ and the configuration that precedes $C_{2,h}$. Let $SC'_1$ be the first of these instructions, let it be executed by request $req''_1 \neq req$ and let $LL'_1$ be its matching $LL$ instruction. Recall that $SC_{1,h}$ is the successful SC instruction executed just before $C_{1,h}$. Since $SC'_1$ is a successful SC instruction, $LL'_1$ must follow $SC_{1,h}$. By the pseudocode, $req''_1$ reads $rn[z_h]$ and $bn[z_{h-1}]$ during the instance of `Calculate` executed between $LL'_1$ and $SC'_1$. Recall that $p$ occupies $z_r$.

Assume first that $h = level(z_r)$. Then, the pseudocode (lines 4 or 14), and Lemmas 4.2 and 4.4 imply that $req''_1$ reads either the pair $(reqt[1], p)$ or the pair $(reqt[2], p)$ into $rn[z_r]$ (depending on whether the read happens between $C_{1,h}$ and $C_{2,h-1}$ or between $C_{2,h-1}$ and the configuration that precedes $C_{2,h}$, respectively) when executing `Calculate` between $LL'_1$ and $SC'_1$. Assume now that $h > level(z_r)$, so $z_h \neq z_r$. Then, Observation 4.3 (Claim 1) implies that $rn[z_h].pid \neq p$. By the pseudocode (lines $28 - 30$), it follows that $req''_1$ will consider as the new request type for $p$, the value read in $bn[z_{h-1}].reqs[p]$ when executing `Calculate` between $LL'_1$ and $SC'_1$. By the induction hypothesis (Claim 5), $req$ is contained in $bn[z_{h-1}].reqs[p]$ at all configurations between $C_{1,h-1}(req)$ and $C_{2,h-1}(req)$.

Thus, $req''_1$ reads either the pair $(reqt[1], p)$ or the pair $(reqt[2], p)$ (depending on whether the read happens between $C_{1,h}$ and $C_{2,h-1}$ or between $C_{2,h-1}$ and the configuration that precedes $C_{2,h}$, respectively) in $bn[z_h]$ when executing `Calculate` between $LL'_1$ and $SC'_1$. We conclude that this is so in either case.

If $req''_1$ reads the pair $(reqt[1], p)$, by the pseudocode, it follows that $req'$ writes the value $reqt[1]$ into $bn[z_h].reqs[p]$ when it executes $SC'_1$. This is a contradiction to our assumption that $SC'_1$ writes $x \neq reqt[1]$ into $bn[z_h].reqs[p]$. Thus, assume that $req''_1$ reads the pair $(\perp, p)$. Then, by the pseudocode, it follows that $req''_1$ writes $\perp$ into $bn[z_h].reqs[p]$ when it executes $SC'_1$. Recall that, in this case, the read by $req''_1$ that occurs between $LL'_1$ and $SC'_1$ must take place after $C_{2,h-1}(req)$. Therefore, $SC'_1$ occurs after $C_{2,h-1}(req)$ and before $SC_{2,h}(req)$. Then, it is $SC'_1$ (and not $SC_{2,h}(req)$) the first `SC` after $C_{2,h-1}(req)$ that writes $\perp$ into $bn[z_h].reqs[p]$, which contradicts the definition of $SC_{2,h}(req)$.

We continue to prove Claim 5. By definition of $C_{2,h}$, $\perp$ is contained in $bn[z_h].reqs[p]$ at $C_{2,h}$. So, we continue to prove that $\perp$ is contained in $bn[z_h].reqs[p]$ at each configuration between $C_{2,h}$ and $C_3(req)$.

By the definitions of $C'_{2,h}$ and $C_3(req)$, and by the Claim 3 that is proved above, $C_3(req)$ follows $C_{2,h}$. Assume, by the way of contradiction, that there is a configuration $C_l$ between $C_{2,h}$ and $C_3(req)$, such that $bn[z_h].reqs[p]$ contains a value $x \neq \perp$ at $C_l$.

By definition of $C_l$, there is at least a successful SC that writes $x$ into $bn[z_h].reqs[p]$ and occurs between $C_{2,h}$ and the configuration that precedes $C_3(req)$. Let $SC'_2$ be the first of these instructions, let it be executed by request $req''_2 \neq req$ and let $LL'_2$ be its matching $LL$ instruction. Recall that $SC_{2,h}$ is the successful SC instruction executed just before $C_{2,h}$. Since $SC'_2$ is a successful SC instruction, $LL'_2$ follows $SC_{2,h}$.

If $h = level(z_r)$, Lemmas 4.2 and 4.4 imply that the read of $rn[z_h]$ by $req''_1$, which occurs between $LL'_2$ and $SC'_2$ (at the beginning of the execution of its `Calculate`) returns $(\perp, p)$. Thus, by the pseudocode, $req''_2$ writes $\perp \neq x$ into $bn[z_h].reqs[p]$ by executing $SC'_2$, which is a contradiction.

Assume now that $h > level(z_r)$. Observation 4.3 (Claim 1) implies that $rn[z_h].pid \neq p$. By the pseudocode (lines $28 - 30$), it follows that the instance of `Calculate` executed by $req''_2$ between $LL'_2$ and $SC'_2$, will consider as the new request type of $p$ the value read in $bn[z_{h-1}].reqs[p]$. By the induction hypothesis, there is a configuration $C_{2,h-1}$ at which $\perp$ is written into $bn[z_{h-1}].reqs[p]$, and $C_{2,h-1}$ precedes $C'_{2,h-1}$; moreover, $\perp$ is contained in

$\mathit{bn}[z_{h-1}].\mathit{reqs}[p]$ from $C_{2,h-1}(\mathit{req})$ until $C_3(\mathit{req})$. Thus, the read of $\mathit{bn}[z_h]$ by $\mathit{req}_2''$, which occurs between $LL_2'$ and $SC_2'$ returns the value $\perp$ for $p$. Thus, by the pseudocode, $\mathit{req}_2''$ writes $\perp \neq x$ into $\mathit{bn}[z_h].\mathit{reqs}[p]$ by executing $SC_2'$, which is a contradiction. ∎

We remark that information about *req* (namely, *req* and the id $p$ of the thread that executes it) is recorded for the first time into one of the base objects when *req* occupies its red node $z_r$. Lemma 4.5 implies that it is then transferred to the blue node with id $z_r$ (by *req* or some other request); moreover, only when it is written there, it can be forwarded to the next node of the blue path of *req*. This transfer continues up to each node of *req*'s blue path until the request type of *req* eventually reaches the root. Recall that for each $h$, $\mathit{level}(z_r) \leq h \leq \log n + 1$, *req* is written into node $z_h$ at level $h$ of $\mathit{pt}(\mathit{req})$ by $SC_{1,h}(\mathit{req})$ just before $C_{1,h}(\mathit{req})$, and $\perp$ is written into $\mathit{bn}[z_h]$ by $SC_{2,h}(\mathit{req})$ just before $C_{2,h}(\mathit{req})$.

The following claim is an immediate consequence of Lemma 4.5 when $h = \log n + 1$.

**Corollary 4.1.** *The request type of any request req is successfully recorded in the blue root by* $SC_{1,\log n+1}(\mathit{req})$.

We continue to prove that the value $\perp$ is contained for some thread $p$ in a blue node $z$ from the time that a request by $p$ writes $\perp$ into $z$ until the time that the subsequent request by $p$ (for which $z$ is contained in its path) writes its request type into $z$ (or until the final configuration if such a request does not exist).

**Lemma 4.6.** *Consider any request req executed by some thread p. For each* $z_h \in \mathit{pt}(p)$, *let* $C_h = C_{1,h}(\mathit{req}_{m_h})$ *if p executes a subsequent request* $\mathit{req}_{m_h}$ *such that* $z_h \in \mathit{pt}(\mathit{req}_{m_h})$; *let* $C_h$ *be the final configuration if such a request does not exist. Then,* $\mathit{bn}[z_h].\mathit{reqs}[p] = \perp$ *at each configuration between* $C_3(\mathit{req})$ *and* $C_h$.

*Proof.* Assume, by the way of contradiction, that the claim does not hold and let $C$ be the first configuration at which the claim is violated. Let *req* be the request (let it be executed by some thread $p$) and $z_h \in \mathit{pt}(p)$ be the node that causes this violation. More specifically, if $\mathit{req}_{m_h}$ is the first request executed by $p$ after *req* for which $z_h \in \mathit{pt}(\mathit{req}_{m_h})$, then $C$ is between $C_3(\mathit{req})$ and $C_{1,h}(\mathit{req}_{m_h})$ and $\mathit{bn}[z_h].\mathit{reqs}[p] = x \neq \perp$ at $C$ (if such a request does not exist, then $C$ is between $C_3(\mathit{req})$ and the final configuration). Assume that $SC'$ is the SC instruction executed just before $C$ which writes the value $x$ in $\mathit{bn}[z_h].\mathit{reqs}[p]$ and let

$req'$ be the request that executes $SC'$. Denote by $LL'$ the corresponding $LL$ instruction to $SC'$.

Assume first that $req'$ reads $x$ in $rn[z_h].req$. Notice that $req'$ performs this read (let it be $r'$) at some configuration $C'$ that precedes $C$. By the pseudocode, it follows that $rn[z_h].pid = p$ and $rn[z_h].req = x$ at $C'$. Thus, $p$ is active executing some request $req''$ at $C'$ such that $z_h$ is the first node in $pt(req'')$.

If $z_h \in pt(req)$, let $req_{b_h} = req$; otherwise, let $req_{b_h}$ be the last request by $p$ preceding $req$ such that $z_h \in pt(req_{b_h})$. (Since $z_h \in pt(req'')$, $req_{b_h}$ is well-defined.) By the pseudocode, $LL'$ happens before $r'$. If $C'$ precedes $C_{2,h}(req_{b_h})$, then $LL'$ precedes $C_{2,h}(req_{b_h})$. Since, by Lemma 4.5, $SC_{2,h}(req_{b_h})$ happens just before configuration $C_{2,h}(req_{b_h})$ and it is a successful SC on $bn[z_h]$, $SC'$ cannot be successful. This is a contradiction. Thus, $C'$ follows $C_{2,h}(req_{b_h})$. Lemmas 4.2 and 4.4 imply that $rn[z_h] = (\perp, p)$ at $C_{2,h}(req_{b_h})$. By definition of $req_{b_h}$, no other request following $req_{b_h}$ and containing $z_h$ in its path is executed by $p$ before $req_{m_h}$. It follows that, at each configuration between $C_{2,h}(req_{b_h})$ and $C'$, $rn[z_h] \neq (x, p)$. This is a contradiction (since we have assumed that $req'$ reads $(x, p)$ in $rn[z_h]$ at $C'$).

Assume now that $req'$ reads $x$ in $bn[z_{h-1}].reqs[p]$ (where $z_{h-1}$ is the node preceding $z_h$ in $pt(p)$); let $r''$ be this read. Notice that $r''$ results in some configuration $C''$ which precedes $C$. Let $req_{m_{h-1}}$ be the first request executed by $p$ after $req$ such that $z_{h-1} \in pt(req_{m_{h-1}})$. If $z_{h-1} \in pt(req)$, let $req_{b_{h-1}} = req$; otherwise, let $req_{b_{h-1}}$ be the last request by $p$ preceding $req$ such that $z_{h-1} \in pt(req_{b_{h-1}})$. In case $req_{b_{h-1}}$ does not exist, denote by $C_{2,h-1}(req_{b_{h-1}})$ the initial configuration. Similarly, in case $req_{b_h}$ does not exist, let $C_{2,h}(req_{b_h})$ be the initial configuration. If $req_{m_{h-1}}$ does not exist, denote by $C_{1,h-1}(req_{m_{h-1}})$ the final configuration. Similarly, in case $req_{m_h}$ does not exist, let $C_{1,h}(req_{m_h})$ be the final configuration.

Since $z_h$ is an ancestor of $z_{h-1}$ in the blue tree, it follows that $z_h \in pt(req_{m_{h-1}})$ and $z_h \in pt(req_{b_{h-1}})$. Thus, either $req_{b_{h-1}} = req_{b_h}$ or $req_{b_{h-1}}$ precedes $req_{b_h}$. Similarly, either $req_{m_{h-1}} = req_{m_h}$ or $req_{m_{h-1}}$ follows $req_{m_h}$.

Assume first that $req_{m_{h-1}}$ follows $req_{m_h}$, so that $C$ precedes $C_{1,h-1}(req_{m_{h-1}})$. By the pseudocode, $LL'$ happens before $r''$. Obviously, $C''$ follows the initial configuration. If $C''$ precedes $C_{2,h}(req_{b_h})$, then $LL'$ precedes $C_{2,h}(req_{b_h})$. Since, by Lemma 4.5, $SC_{2,h}(req_{b_h})$ happens just before $C_{2,h}(req_{b_h})$ and it is a successful SC on $bn[z_h]$, $SC'$ cannot be successful. This is a contradiction. Thus, $C''$ follows $C_{2,h}(req_{b_h})$.

We now prove that, at each configuration between $C_{2,h}(req_{b_h})$ and $C_{1,h-1}(req_{m_h})$, $bn[z_{h-1}].reqs[p] = \perp$. In case that $req_{b_{h-1}} = req_{b_h}$, Lemma 4.5 implies that $C_{2,h-1}(req_{b_h})$ precedes $C_{2,h}(req_{b_h})$, and at each configuration between $C_{2,h-1}(req_{b_h})$ and $C_3(req_{b_h})$, it holds that $bn[z_{h-1}].reqs[p] = \perp$. Otherwise, recall that $req_{b_{h-1}}$ precedes $req_{b_h}$ and they are both executed by $p$, so $C_3(req_{b_{h-1}})$ precedes $C_{2,h}(req_{b_h})$. Since $C$ is the first configuration at which the claim of the lemma is violated, it must be that, at each configuration between $C_3(req_{b_{h-1}})$ and $C_{1,h-1}(req_{m_{h-1}})$, $bn[z_{h-1}].reqs[p] = \perp$.

Assume that $C$ precedes $C_{1,h-1}(req_{m_{h-1}})$. It follows that at each configuration between $C_3(req_{b_{h-1}})$ and $C$, it holds that $bn[z_{h-1}].reqs[p] = \perp$. It follows that $r''$ reads $\perp \neq x$ in $bn[z_{h-1}].reqs[p]$ and writes $\perp \neq x$ in $bn[z_h].reqs[p]$, which is a contradiction.

Assume now that $C$ follows $C_{1,h-1}(req_{m_{h-1}})$. In case that request $req_{m_{h-1}}$ follows request $req_{m_h}$, it holds that configuration $C$ precedes $C_{1,h-1}(req_{m_{h-1}})$, so it must be that $req_{m_{h-1}} = req_{m_h}$. Since (1) it holds that $bn[z_{h-1}].reqs[p] = \perp$ at each configuration between $C_{2,h}(req_{b_h})$ and $C_{1,h-1}(req_{m_{h-1}}) = C_{1,h-1}(req_{m_h})$, (2) $r''$ occurs after $C_{2,h}(req_{b_h})$, and (3) $r''$ returns $x \neq \perp$, it must be that $r''$ occurs after $C_{1,h-1}(req_{m_h})$. Thus, $r''$ occurs between $C_{1,h-1}(req_{m_h})$ and $C_{1,h}(req_{m_h})$. Lemma 4.5 implies that, at all configurations between $C_{1,h-1}(req_{m_h})$ and $C_{1,h}(req_{m_h})$, it holds that $bn[z_{h-1}].reqs[p] = req_{m_h}$. Thus, $r''$ reads $req_{m_h}$ in $bn[z_{h-1}].reqs[p]$ and writes the same value in $bn[z_h].reqs[p]$ by executing $SC'$. However, Lemma 4.5 implies that the first configuration after $C_{1,h-1}(req_{m_h})$ at which $req_{m_h}$ is written into $bn[z_h].reqs[p]$ is $C_{1,h}(req_{m_h})$. Since we have assumed that $C$ precedes $C_{1,h}(req_{m_h})$, this is a contradiction. ∎

We are now ready to assign linearization points to the requests of F-RedBlue. By Observation 4.1, there is at least one successful SC that records the request type of *req* in the blue root node. Recall that $SC_{1,\log n+1}(req)$ is the first of these SC instructions. We place the linearization point of *req* at $SC_{1,\log n+1}(req)$; ties are broken by the order that thread identifiers impose.

**Lemma 4.7.** *For each request req, the linearization point of req is placed in its execution interval.*

*Proof.* By Lemma 4.5, the request type of *req* is recorded in the root node by some SC instruction and this occurs before the execution of the type 2 SC instruction by *req* (line 14). Thus, the linearization point of *req* precedes the end of its execution interval.

Let $SC(req)$ be the first successful SC that records the request type of *req* in the blue root node. Notice that *req* is invisible to all threads until it performs its first store request, writing its information into the red node that it occupies. Thus, $SC(req)$ must follow the beginning of the execution interval of *req*. ∎

We say that a request *req is applied* on the simulated object if (1) procedure `Calculate` executed by some request $req'$ (that might be *req* or another request), reads in the appropriate child node of the blue root or in the red root node, a value equal to *req* (i.e. the request type written there for *req*) and records it as the new request type for $p$, (2) `Calculate` by $req'$ applies this request type with its parameters, and (3) the execution of the SC of line 23 (let it be $SC_r$) on $bn[1]$ by $req'$ succeeds (thus writing there the value *req* for $p$). When these conditions are satisfied, we sometimes also say that $req'$ applies *req* on the simulated object or that $SC_r$ applies *req* on the simulated object. We next prove that each request *req* is applied on the simulated object exactly once.

**Lemma 4.8.** *Each request req is applied on the simulated shared object exactly once by* $SC_{1,\log n+1}(req)$.

*Proof.* Assume that *req* is executed by thread $p$. We first prove that *req* is applied on the simulated object at least once. Lemma 4.5 implies that *req* is successfully recorded in the root node of the blue tree at least once and that $SC_{1,\log n+1}$ is the first `SC` instruction that stores *req* into $bn[1].reqs[p]$. Let $req'$ be the request that executes $SC_{1,\log n+1}$.

In case $p$ executes a request before *req*, let $C_{b_h} = C_{2,\log n+1}(req_{b_h})$, where $req_{b_h}$ is the last preceding to *req* request executed by $p$; otherwise, let $C_{b_h}$ be the initial configuration of the algorithm. If $req'$ performs the read of $bn[1]$ that precedes $SC_{1,\log n+1}$ before $C_{b_h}$, then $SC_{1,\log n+1}$ fails. This is so because, by the pseudocode, the corresponding $LL$ to $SC_{1,\log n+1}$ precedes this read, and, by the definition of $C_{b_h}$ and Lemma 4.5, a successful `SC` on $bn[1]$ (namely $SC_{2,\log n+1}(req_{b_h})$) occurs at $C_{b_h}$, thus causing the failure of $SC_{1,\log n+1}$. This is a contradiction.

Thus, $req'$ performs its read after $C_{b_h}$. Lemmas 4.5 and 4.6, imply that, at each configuration between $C_{b_h}$ and $C_{1,\log n+1}(req)$, $bn[1].reqs[p] = \perp$. It follows that $req'$ reads $\perp$ into $bn[1].reqs[p]$ during the execution of `Calculate` that precedes $SC_{1,\log n+1}(req)$. Since $req \neq \perp$, $req'$ evaluates the condition of the *if* statement of line 31 to true. We conclude that $req'$ applies *req*.

We now prove that *req* is applied at most once on the simulated object. Assume, by the way of contradiction, that *req* is applied at least twice, and let $SC'$ be the first `SC` after $SC_{1,\log n+1}(req)$ that applies *req*. Let $req''$ be the request that executes $SC'$ and let $r'$ be the last read of $bn[1]$ executed by $req''$ before $SC'$.

If $r'$ occurs before $SC_{1,\log n+1}(req)$, then the corresponding $LL$ to $SC'$, which precedes $r'$, precedes also $SC_{1,\log n+1}(req)$. Thus, $SC_{1,\log n+1}(req)$ causes $SC'$ instruction to fail, which is a contradiction. Therefore, $r'$ follows $SC_{1,\log n+1}(req)$.

If $r'$ occurs between configurations $C_{1,\log n+1}(req)$ and $C_{2,\log n+1}(req)$, then Lemmas 4.5 and 4.6 imply that $r'$ reads the value *req* in $bn[1]$. Since $req''$ apply *req*, it must be that $tmp.reqs[p] = req$ when $req''$ executes line 31 of the instance of `Calculate` that precedes $SC'$. Thus, by the pseudocode (line 31), it follows that the condition of the `if` statement of line 31 is evaluated to *false*. Thus, $req'$ does not call ApplyOp which contradicts the fact that $req'$ applies *req*.

Assume finally that $r'$ follows $C_{2,\log n+1}(req)$. Lemma 4.4 and Observation 4.3 imply that $rn[1] \neq req$ at all configurations after the configuration at which *req* executes its type 2 `SC` instruction. Moreover, if any of the root children belongs to $pt(req)$, then Lemma 4.5 imply that $C_{2,\log n}(req)$ precedes $C_{2,\log n+1}(req)$, and $bn[\log n].reqs[p] \neq req$ after $C_{2,\log n}(req)$. By the pseudocode, it therefore follows that procedure `Calculate` executed by $req''$ before $SC'$, calculates as the new value of $bn[1].reqs[p]$ a value other than *req* and therefore it does not apply *req*, which is a contradiction. ∎

In order to prove consistency, we use the following notation. Denote by $SC_m$, $m > 0$, the $m$th successful `SC` instruction on $b[1]$, which is the root node of the blue tree, and let $LL_m$ be its matching `LL`. Obviously, between $SC_m$ and $SC_{m+1}$, $b[1]$ is not modified.

Denote by $\alpha_m$, the prefix of $\alpha$ which ends at $SC_m$ and let $C_m$ be the first configuration following $SC_m$. Let $\alpha_0$ be the empty execution. Denote by $L_m$ the order defined by the linearization points, assigned as described above, of the requests in $\alpha_m$. We remark that $b[1].st$ stores a copy of the simulated state at each point in time. Moreover, each thread applies requests on its local copy of the simulated state sequentially, the one after the other. We say that $b[1].st$ is *consistent* at $C_m$ if it is the same as the state that results if the requests of $\alpha_m$ are executed sequentially in the order specified by $L_m$.

**Lemma 4.9.** *For each $m \geq 0$, (1) $b[1].st$ is consistent at $C_m$, and (2) $L_m$ is a linearization order for $\alpha_m$.*

*Proof.* We prove the claim by induction on $m$.

**Base case ($m$=0):** The claims hold trivially: by the initialization of $b[1]$, $b[1].st$ contains $\hat{s}$, which is the initial state of the simulated object, and $\alpha_0$ is empty.

**Induction hypothesis:** Fix any $m > 0$ and assume that the claims hold for $m - 1$.

**Induction step:** We prove that the claim holds for $m$. By the induction hypothesis, it holds that: (1) $b[1].st$ is consistent at $C_{m-1}$, and (2) $L_{m-1}$ is a linearization order for $\alpha_{m-1}$. Let $req$ be the request that executes $SC_m$. Assume that $req$ applies $j > 0$ requests on the simulated object. Denote by $req_1, ..., req_j$ the sequence of these requests ordered in increasing order of the identifiers of the threads that initiate them.

Notice that $req$ performs $LL_m$ after $C_{m-1}$ since otherwise $SC_m$ would not be successful. By the induction hypothesis, $b[1].st$ is consistent at $C_{m-1}$. Thus, the local copy of $b[1]$ that is last stored by $req$ in $tmp$, represents a consistent state of the simulated object. Lemma 4.8 implies that $req_1, \ldots, req_j$ are applied only once. This is realized when $SC_m$ is executed. Thus, none of these requests have been applied in the past.

Given that the application of $req_1, ..., req_j$ is simulated by the thread executing $req$ sequentially, in the order mentioned above, starting from the state stored in $tmp$, it is a straightforward induction to prove that (1) for each $f$, $0 \leq f \leq j$, a consistent response is calculated for $req_f$, and the new state of the simulated object is calculated in a correct way in the local variable $tmp$ of the `Calculate` executed by $req$. Therefore, $b[1].st$ is consistent after the execution of $req$'s successful `SC`. Notice that, by the way linearization points are assigned, $L_m = L_{m-1}, req_1, \ldots, req_j$. It follows that $L_m$ is a linearization order for $\alpha_m$. ∎

Pseudocode (lines 1 and 19-24) implies that an operation *op* takes as many steps as the path length it traverses in the red and blue tree. Lemma 4.1 and the pseudocode (lines 4 and 17) imply that when a process occupies some red node, then the interval contention is at least equal with the depth of the occupied node. Since the maximum path length of the red and blue trees is $\lg n + 1$, the time complexity of F-RedBlue is $O(\min\{k, \lg n\})$, where $k$ is the point contention. Thus, the following theorem holds.

**Theorem 4.1.** F-RedBlue *is a linearizable, wait-free implementation of a universal object that uses* $2n-2$ LL/SC *objects. Its step complexity is* $O(\min\{k, \lg n\})$.

## 4.2 Modified version of F-RedBlue that uses small base objects

In this section, we present S-RedBlue, a modified version of F-RedBlue that uses small base objects. Now, each red node stores $\lceil \log n \rceil + 1$ bits. A blue node other than the root stores $n$ bits. The blue root stores $n$ bits, a thread id and the state of the object. This LL/SC base object is implemented by single-word LL/SC base objects using the implementation presented in [44].

In S-RedBlue, a thread $p$ uses a single-writer base object to record its current request (line 1). As in F-RedBlue, the thread starts the execution of any of its requests by traversing the red tree. However, to occupy a red node, the thread just records its id and sets the bit of the node to *true*.

Similarly, each thread, moving up the path to the root of the blue true, just sets a bit in each node of the path to identify that it is currently executing a request. Thus, the bit array of the root identifies all threads that are currently active.

To avoid storing the return values in the root node, each thread $p$ maintains an array of $n$ single-writer base objects, one for each thread. When $p$ reaches the root (during the application of one of its requests), it first records the responses for the currently active threads in its appropriate single-writer base objects (lines $25-26$). Then, it tries to store the new state of the object in the blue root together with its id and the set (bit vector) of active threads. A thread finds the response for its current request in the appropriate single-writer base object of the thread whose id is recorded in the root node.

The state is updated only at the root node and only when the bit value for a thread changes from *false* to *true* in the blue root's bit array (line 23). This guarantees that the request of each thread is applied only once to the simulated object. However, all threads reaching the root, record responses for each currently active thread $p$ in their single-writer base objects, independently of whether they also apply $p$'s request to the simulated object. This is necessary, since the request of $p$ may be applied to the object by some thread $q$ and later on (and before $p$ reads the root node for finding its response) another thread $q'$

```
struct RedNode{
    boolean req;
    int pid;
};
struct BlueNode{
    state st;                       // this field is used only at the root node
    RetVal rvals[n];                // this field is used only at the root node
    boolean reqs[n];
};

shared RedNode rn[1..2n-1] = {<F, -1>, ..., <F, -1>};
shared BlueNode bn[1..2n-1]={<F, <0, ...,0>, <F,...,F>>,...,$< \perp, < 0, ..., 0 >, < \perp, ..., \perp >>$};
shared RetVal rvals[1..n][1..n] = {{$\perp, \ldots, \perp$}, ..., {$\perp, \ldots, \perp$}};
share request Announce[1..n];

RetVal ApplyOp(Request req){        // pseudocode for thread $p$
    int direction = n/2, z = 1, levels = lg(n) + 1, l, j;
    RetVals rv;
    announce[i] = req;              // $p$ announces its request
    for(l=levels;l≥1;l--){          // traversal of red path
        LL(rn[z]);
        if(rn[z] == <F, -1>)
            if(SC(rn[z], <req, p>)) break;
        if(p≤direction){            // find the next node in the path
            direction = direction - $2^{l-3}$;
            z = 2 * z;              // move to the left child of $z$
        } else{
            direction = direction + $2^{l-3}$;
            z = 2 * z + 1;          // move to the right child of $z$
        }
    }
    Propagate(z);                   // first traversal of blue path: propagating the request
    rv = bn[1].rvals[p];
    LL(rn[z]);
    SC(rn[z], <F, p>);              // the request occupying $rn[z]$ starts its deletion phase
    Propagate(z);                   // second traversal of blue path: propagating $\perp$
    LL(rn[z]);
    SC(rn[z], <F, -1> );            // re-initialize the occupied red node to $\perp$
    return rv;                      // return the appropriate value
}
```

Algorithm 3: Pseudocode for S-RedBlue.

may overwrite the root contents. thread $q'$ will include $p$ in its calculated active set but it will not re-apply $p$'s request to the object, since it will see that $p$'s bit in the active set of the root node is already set. Still $q'$ should record a response for $p$ in its single-writer base objects since $p$ may read $q'$ and not $q$ in $bn[1].pid$ when seeking for its response.

The proof that S-RedBlue is correct closely follows the correctness proof of F-RedBlue. The main difference of the two algorithms is on the way that response values are calculated. If $q$ is the thread that applies some request $req$, the response for $req$ is originally stored

```
    void Propagate(int $z$){                        // pseudocode for thread $p$
20     while (z!=0){                                // traversal of the blue path
21        for (j=1 to 2) do{                        // two efforts to store the appropriate information
22           LL(bn[z]);
23           bt=Calculate(z, i);
24           SC(z, bt);
          }
25        z =$\lfloor$z/2$\rfloor$;
       }
    }

    BlueNode Calculate(int z) {
       BlueNode tmp=$< \perp, < 0, \ldots, 0 >, < \perp, \ldots, \perp >>$;
       BlueNode blue=bn[z], lc, rc;
       RedNode red = rn[z];
       int q, range;
26     if (2*z+1 < 2n) {
27        lc = bn[2*z];
28        rc = bn[2*z+1];
       }                                            // if $z$ is an internal node
29     range = $2^{\lg(n)-\lceil \lg(z) \rceil}$;                    // compute the number of leaves of each subtree
       // copy the requests placed on the left subtree
30     tmp.reqs[2q-range..2q] = lc.reqs[2q-range..2q];
       // copy the requests placed on the right subtree
31     tmp.reqs[2q+1..2q+1+range] = rc.reqs[2q+1..2q+1+range];
32     if (red.pid $\neq$ -1) tmp.reqs[red.pid]=red.req; // if thread $q$ occupies node red
33     if (z == 1) {
34        tmp.rvals[1..n] = blue.rvals[1..n];       // copy the return values
35        tmp.st = blue.st;                         // copy object's state
36        for q=1 to n do{                          // local loop
37           if (tmp.reqs[q]==T AND blue.reqs[q]==F) // apply any pending request
38              apply tmp.reqs[q] to tmp.st and store into tmp.rvals[q] the return value;
39           else if(tmp.reqs[q]==T)                // store the return value for pending request q
40              rvals[p][q]=rvals[b.pid][q];
          }
       }
41     return tmp;
    }
```

Algorithm 4: Pseudocode for Propagate and Calculate of S-RedBlue.

in $rvals[q][p]$ and the id of $q$ is written into the root node. The next thread to update the root node will find the id of $q$ in the root node and (as long as $req$ has not yet read its response by executing line 8), it will see that $tmp.reqs[p] = T$. Therefore, it will copy the response for $req$ from $rvals[q][p]$ (line 26) to its appropriate single-writer base object. So, when $p$ seeks for the response of $req$ it will find the correct answer in the single-writer base object of the thread recorded at the root node.

S-RedBlue uses $O(n)$ multi-writer LL/SC objects and $O(n^2)$ single-writer read/write base objects. One of the multi-writer base objects is large and it is implemented using the implementation of a $W$-word LL/SC object from single-word LL/VL/SC objects presented in [44]. This implementation achieves time complexity $O(W)$ for both LL and SC and has space complexity $O(nW)$. Thus, the number of base objects used by S-RedBlue is $O(n^2 + nW)$. In common cases where $n$ bits fit in a constant number of single-word base objects, the time complexity of S-RedBlue is $O(k + W)$ since Calculate pays $O(k)$ to record $k$ response values in the single-writer base objects and $O(W)$ for reading and modifying the root node.

**Theorem 4.2.** S-RedBlue *is a linearizable, wait-free implementation of a universal object that uses $O(n^2 + nW)$ base objects and its step complexity is $O(k + W)$.*

## 4.3 Adaptive synchronization algorithms for large objects

In the universal constructions for large objects presented by Anderson and Moir in [11] the object is treated as if it were stored in a contiguous array. Moreover, the user is supposed to provide sequential implementations of the object's requests which call appropriate Read and Write procedures (described in [11]) to perform read or write requests in the contiguous array (see [11, Section 4] for more information on what the user code should look like and a example). The universal constructions partition the contiguous array into $B$ blocks of size $S$ each, and during the application of a request to the object, only the block(s) that should be modified are copied locally (and not the entire object's state). The authors assume that each request modifies at most $T$ blocks.

S-RedBlue can easily employ the simple technique of the *lock-free* construction presented in [11] in order to provide a simple, adaptive, *wait-free* algorithm (called LS-RedBlue) for large objects. As illustrated in Algorithm 5, only routine Propagate requires some modifications. Also, data structures are similar to those of [11] are needed for storing the array blocks, having threads making "local" copies of them and storing back the changed versions of these blocks. More specifically, array $BLK$ stores the $B$ blocks of the object's state, as well as a set of *copy* blocks used by the threads for performing their updates without any interference by other threads. Since each request modifies at most $T$ blocks, a thread reaching the blue root, requires at most $kT$ copy blocks in order to make copies of the $kT$ state blocks that it should possibly modify. So, $BLK$ contains $nkT + B$

blocks; initially, the object's state is stored in $BLK[nkT+1], \ldots, BLK[nkT+B]$ (the blocks storing the state of the object at some point in time are called *active*). The blue root node stores an array named $BANK$ of $B$ indices; the $i$th entry of this array is the pointer (i.e. the index in $BLK$) of the $i$th active block. Each thread $p$ has a private variable $ptrs_p$, which it is used to making a local copy of the $BANK$ array (line 9).

The application of an active request to the object is now done by calling (in `Calculate`) the appropriate sequential code provided by the user. The codes of the `Read` and `Write` routines (used by the user code) are also presented in Algorithm 5 (although they are the same as those in [11]). These routines take an index *addr* in the contiguous array as a parameter. From this index, the block number $blkidx_p$ that should be accessed is calculated as $blkidx_p = addr$ div $S$, and the offset in this block as $addr$ mod $S$. The actual index in $BLK$ of the $blkidx_p$-th block can be found through the $BANK$ array. However, the thread uses its local copy $ptrs_p$ of $BANK$ for doing so. Thus, line 15 simply access the appropriate word of $BLK$. If the execution of the $VL$ instruction of line 16 by some thread $p$ does not succeed, the $SC$ instruction of line 11 by $p$ will also not succeed. So, we use the `goto` to terminate the execution of its `Calculate` .

The first time that thread $p$ executes a `Write` to the $blkidx_p$-th block, it copies it to one of its copy blocks (line 21). Array $dirty_p$ is used to identify whether a block is written for the first time by $p$. In this case, the appropriate block is copied into the appropriate copy block of $p$ (line 21). Indices to the $kT$ copy blocks of $p$ are stored in $p$'s private array $copy_p$. The dirty bit for this block is set to *true* (line 22). Counter $dcnt_p$ counts the number of different blocks written by $p$ thus far in the execution of its current request (line 25). The appropriate entry of $ptrs_p$ changes to identify that the $blkidx_p$-th block is now one of the copy blocks of $p$ (line 23). The write is performed in the copy block at line 27. A thread $p$ uses its copy blocks to make copies of the blocks that it will modify. If later $p$'s $SC$ at line 11 is successful, some of $p$'s copy blocks become active blocks (substituting those that have been modified by $p$). These old active blocks (that have been substituted) consist the new copy blocks of $p$ which it will use to perform its next request. This is accomplished with the code of line 12.

LS-RedBlue is a wait-free algorithm; it has space overhead $\Theta(n^2 + n(B + kTS))$ and its time complexity is $\Theta(B + k(D + TS))$. The wait-free universal construction presented in [11] assumes that each thread has enough copy blocks to perform at most $M/T$ other

```
type INDEX $\{1, \ldots, nkT + B\}$;
struct BlueNode {
    INDEX BANK[B];
    int pid;
    boolean reqs[n]
};
shared word $BLK[1..B + kN * T][1..S]$;

INDEX $copy_p[1..kT]$, $oldlst_p[1..kT]$, $dcnt_p$, $blkidx_p$;
pointer $ptrs_p[1..B]$;
boolean $dirty_p[1..B]$;
word $v_p$;

void Propagate(int $z$) {                                  // pseudocode for thread $p$
    BlueNode $b$;
    while(z $\neq$ 0) {
        for(int i=1 to 2) do {
            if(z == 1) {
                for(int j=1 to B) do
                    $dirty_p$[j]=$false$;
                    $dcnt_p$ = 0;
            }
            b=LL(bn[z]);
            if (z == 1) $ptrs_p$ = b.BANK;
            bt = Calculate(z);
            if (SC(bn[z], bt) AND z==1)
                for (int l=1 to $dcnt_p$) do
                    $copy_p$[l]=$oldlst_p$[l];
        }
        $z$ =$\lfloor$z/2$\rfloor$;
    }
}

wordtype Read(int addr) {
    $v_p$=BLK[$ptrs_p$[addr div S]][addr mod S];
    if (VL(BANK)==$false$)
        goto line 41 of Calculate (Algorithm 4);
    else return $v_p$;
}

void Write(int addr, wordtype val) {
    $blkidx_p$=addr div S;
    if ($dirty_p$[$blkidx_p$]==$false$) {
        memcpy(BLK[$copy_p$[$dcnt_p$]], BLK[$ptrs_p$[$blkidx_p$]], sizeof(blktype));
        $dirty_p$[$blkidx_p$]=$true$;
        $oldlst_p$[$dcnt_p$]=$ptrs_p$[$blkidx_p$];
        $ptrs_p$[$blkidx_p$]=$copy_p$[$dcnt_p$];
        $dcnt_p$=$dcnt_p$+1;
    }
    BLK[$ptrs_p$[$blkidx_p$]][addr mod S]=val;
}
```

Algorithm 5: Pseudocode for LS-RedBlue.

```
struct BlueNode{
   int BANK[B];                        // this field used only at root
   PINDEX pid;                         // this field used only at root
   PINDEX help;                        // this field used only at root
   boolean reqs[n];
};
void Propagate(int z){                 // pseudocode for thread $p$
   BlueNode b;
   while(z!=1){
      for (int d=1 to 2) do {
         b=LL(bn[z]);
         bt=Calculate (z);
         SC(bn[z], bt);
      }
      z =$\lfloor$z/2$\rfloor$;
   }
   b=LL(bn[1]);                        // requests to perform at root
   while (b.reqs[p] == false) {
      for (int j=1 to B) do
         $dirty_p$[j]=false;
      $dcnt_p$ = 0;
      b=LL(bn[1]);
      $ptrs_p$ = b.BANK;
      bt=Calculate (1);
      if (SC(bn[1], bt))
         for (l=1 to $dcnt_p$) do
            $copy_p$[i] = $oldlst_p$[i];
   }
}
```

Algorithm 6: Pseudocode for BLS-RedBlue.

requests in addition to its own where $M \geq 2T$ is any fixed integer. The algorithm uses a quite complicated helping mechanism with return values written into return blocks which should then be recycled in order to keep the memory requirements low. This universal construction has time complexity $O((n/min\{n, M/T\})(B + nD + MS))$. LS-RedBlue achieves much better time complexity ($\Theta(B + k(D + TS))$) and is adaptive. However, it assumes that threads have enough copy blocks to help any number of other active threads.

LS-RedBlue can be slightly modified to disallow a thread to help more than $M/T$ other threads. The resulting algorithm (BLS-RedBlue) is much simpler than the wait-free construction of [11] since it does not require the complicated mechanisms of [11] for returning values and verifying the application of a request. These tasks are performed in BLS-RedBlue in the same way as in S-RedBlue.

The BLS-RedBlue algorithm is presented in Algorithm 6. Propagate executes the same code as in S-RedBlue for all nodes other than the root. The code executed by a thread $p$

```
    BlueNode Calculate(int z) {                    // pseudocode for thread $p$
       BlueNode tmp=$< \perp, < 0, \ldots, 0 >, < \perp, \ldots, \perp >>$;
       BlueNode blue=bn[z], lc, rc;
       RedNode red = rn[z];
       int q, range, d, help = 0;
18     if (2*z+1 < 2n) {
19        lc = bn[2*z];
20        rc = bn[2*z+1];
       }                                           // if $z$ is an internal node
21     range = $2^{\lg(n)-\lceil \lg(z) \rceil}$;                   // compute the number of leaves of each subtree
       // copy the requests placed on the left subtree
22     tmp.reqs[2q-range..2q]=lc.reqs[2q-range..2q];
       // copy the requests placed on the right subtree
23     tmp.reqs[2q+1..2q+1+range] = rc.reqs[2q+1..2q+1+range];
24     if (red.pid $\neq$ -1) tmp.reqs[red.pid]=red.req; // if thread $q$ occupies node red
25     if (z == 1) {                               // in case z is the root node
26        q = blue.help;                           // start helping form thread q
27        tmp.pid = $p$;                             // set thread's id
28        tmp.rvals[1..n] = blue.rvals[1..n];      // copy the return values
29        tmp.st = blue.st;                        // copy object's state
30        for d=1 to n do {                        // local loop
31           if (tmp.reqs[q]==true AND blue.reqs[q]==false)          // apply any pending request
32              if (help < M/T) {                  // help at most M/T requests
33                 apply tmp.reqs[q] to tmp.st and store into tmp.rvals[q] the return value;
34                 help = help + 1;                // increase the number of helped requests
35              } else tmp.reqs[q] = false;        // mark that thread $q$ has an unapplied request
36           else tmp.reqs[q]==true)               // store the return value for pending request q
37              rvals[p][q]=rvals[b.pid][q];
          }
       }
38     return tmp;
    }
```

Algorithm 7: Pseudocode for Calculate of BLS-RedBlue.

when it reaches the blue root (lines $18-37$) is similar to the one of LS-RedBlue. However, lines $31-34$ may have to execute more times in order to ensure that $p$'s request has been applied to the object. Only when this has occurred, $p$'s Propagate returns. To speed up this thread, we store one more field, called *help*, in the blue root node. Each thread, applying a successful $SC$ on the root node, writes there the index of the last active thread it has helped, and next time threads start their helping effort from the next to this thread. This has as a result, the body of the while loop (line 8) to execute at most $\min\{k, 2M/T\}$ times. Each time that the loop is executed twice, $M/T$ more active threads are helped. Therefore, after $2k/(\min\{k, M/T\})$ iterations, the request of $p$ will have been applied to the object.

Each iteration of the loop requires $O(B)$ time to execute lines $9 - 10$ and 13. Each execution of `Calculate` applies at most $\min\{k, M/T\}$ requests. The cost of applying these requests is $O(MS + \min\{k, M/T\}D)$. Finally, the cost of calculating the return values at each execution of `Calculate` is $O(k)$. So, the cost of executing the while loop is $O(k/(\min\{k, M/T\})(B + MS + k + \min\{k, M/T\}D))$. Given that each thread requires only $O(\log k)$ steps to reach the root node, it follows that the time complexity of BLS-RedBlue algorithm is $O((k/\min\{k, M/T\})(B + MS + k + \min\{k, M/T\}D))$. Obviously, BLS-RedBlue achieves better time complexity than the wait-free construction of [11] and it is adaptive. This is achieved without any increase to the required space overhead which is $O(n^2 + n(MS + B))$ for both algorithms.

In case a return value has size larger than a single word, i.e. it is at most $R$ words, our algorithms can still work with single-word base objects by substituting the array of single-writer base objects held by each thread with a bi-dimensional array of $nR$ words. Then, the time complexity of BLS-RedBlue becomes $O((k/\min\{k, M/T\})\ (B + MS + kR + \min\{k, M/T\}D))$. The wait-free universal construction of [11] has time complexity $O(n/min\{n\ M/T\}(B + nR + nD + MS))$ under this assumption.

If $n$ is very large, a technique like the one used by GroupUpdate [4] can be employed to store a single pointer instead of the bit vector in each blue node. Then, the time complexity of BLS-RedBlue becomes $O(k \log k + (k/\min\{k, M/T\})(B + MS + kR + \min\{k, M/T\}D))$. We expect that $k \log k \in O((k/\min\{k, M/T\})\ (B + MS + kR + \min\{k, M/T\}D))$ for large objects in most cases.

# Chapter 5

# Practical Wait-Free Synchronization Algorithms

In this chapter, the family of Sim synchronization algorithms is presented. Specifically, in Section 5.1, we present the Sim wait-free synchronization algorithm based on the simple algorithm presented in [37]. In Section 5.2, a practical and efficient implementation of P-Sim is discussed, while its performance is evaluated in Section 5.3. In Section 5.4, we present the L-Sim synchronization algorithm, which is suitable for simulating objects with large state. Section 5.5 presents a wait-free stack implementation based on P-Sim, while Section 5.6 presents a wait-free shared queue implementation also based on P-Sim.

## 5.1 The Sim algorithm

In this section, we present the Sim algorithm. Sim is a wait-free synchronization algorithm with $O(1)$ step complexity using an Add and an LL/SC object.

### 5.1.1 Algorithm description

Sim (Algorithm 8) uses an LL/SC object $S$ and a collect object $Col$. The LL/SC object stores the simulated state $st$, a vector, called *applied*, of $n$ bits with initial value 0, and an array *rvals* of $n$ elements containing the return values. We remark that the size of $S$ could be reduced to a single pointer using indirection (see Section 5.2 which describes how to build a practical version of Sim).

Each thread maintains a persistent local variable $toggle_i$, initially 0, which it toggles each time it performs a new request. The collect object consists of $n$ components, one for each thread. The $i$th component of $Col$ stores the last request $req_i$ initiated by $p_i$ (or $\perp$ if no such request exists) and a toggle bit *toggle* which stores the value contained in $toggle_i$ at the time that $req_i$ was initiated (or 0 if no such request exists). Whenever $p_i$ wants to perform some request $req_i$, it first announces $req_i$ by updating component $i$ of $Col$ with the value $\langle req_i, toggle_i \rangle$ (line 1). It then toggles $toggle_i$. Finally, $p_i$ executes a routine (line 3), called Attempt, to ensure that its request has been executed.

A request by $p_i$ is applied only if the *toggle* field of the $i$th component of $Col$ differs from the $i$th bit of $S.applied$ (lines 13, 15). In more detail, when $p_i$ wants to execute its first request $req_1$, it writes in the *toggle* field of the $i$th component of $Col$ the value 1 (line 3). Each thread $q$ that sees the value 1 in $Col[i].toggle$ and 0 in $S.applied[i]$, will apply $req_1$ on the copy of the simulated state that it works on. However, only one of them will succeed in updating $S$ on line 18. This update changes $S.applied[i]$ to 1 which identifies that $req_1$ has been applied. When $p_i$ initiates its second request, it changes the *toggle* field of the $i$th component of $Col$ to 0, thus storing in it a different value than that of $S.applied[i]$; this indicates that a new request by $p_i$ has been announced.

When a thread $p_i$ executes Attempt, it first creates a copy of $S$ (line 6) which contains the state of the simulated object. Then, it discovers which requests are currently active (line 7) by executing COLLECT, and performs locally, one after the other, those of them that have not been applied yet (lines 8 - 11) by using its local copy $ls$ of $S$. By doing so,

```
    typedef struct {
       State st;
       boolean applied[1..n];
       RetVal rvals[1..n];
    } StRec;

    typedef struct {
       Request req;
       boolean toggle;
    } CollectRec;

    // $\hat{s}$ is the initial state of the simulated object
    shared StRec S = $\langle \hat{s}, \langle 0, \ldots, 0\rangle, \langle \perp, \ldots, \perp\rangle\rangle$;

    // Col is a collect object that stores n structs of type CollectRec
    shared CollectRec Col(n) = $<< \perp, 0 >, \ldots, < \perp, 0 >>$;

    Boolean $toggle_i$ = 1;                        // Persistent variable of thread $p_i$

    RetVal SimApplyOp(Request req, ThreadId i){
1      update(Col, i, <req, $toggle_i$>);          // Announce $req$
2      $toggle_i$ = 1 - $toggle_i$;
3      Attempt(i);                                 // Call Attempt to perform $req$
4      return S.rvals[i];
    }

    void Attempt(ThreadId i) {                     // Code for Attempt
       StRec ls;
       CollectRec v[n];                            // v stores a copy of the collect object

5      for j=1 to 2 do{
6         ls = LL(S);                              // create a local copy of S in ls
7         v = collect(Col);                        // discover the active requests
8         for l=1 to n do {
9            if(v[l].toggle $\neq$ ls.applied[l]) {      // if $p_l$ has a request not applied yet
10              apply v[l].req on ls.st and store the return value into ls.rvals[l];
             }
11           ls.applied[l] = v[l].toggle;
          }
12        SC(S, ls);                               // Try to change the contents of S
       }
    }
```

Algorithm 8: Pseudocode for Sim.

it calculates a new state for the simulated object and a return value for each of the active requests (line 10); finally, Attempt attempts to write the value of $ls$ into $S$ by executing an SC (line 12). If, in the mean time, some other thread managed to replace $S$ with its local copy of the simulated state, then $p_i$'s SC will not succeed and the actions it performed during the execution of the current loop of for (of line 8) will be discarded.

Recall that thread $p_i$ computes the return values (line 8) for the requests that it attempts to perform and stores them in $ls.rvals$ (line 8). We remark that $ls.rvals$ contains return values for all active requests (and not only for those that $p_i$ attempts to perform) since all return values recorded in $S$ are copied in $ls$ by executing the `LL` of line 6.

The instance of `Attempt` executed by $p_i$ performs the above steps twice (lines 6-12) to ensure that its currently active request $req_i$ has been applied to the simulated object before the instance of SimApplyOp that is currently executed by $p_i$ responds. We remark that executing lines 6-12 just once is not enough. This is so since, if this was the case, there could be another request $req_j$ executed by some thread $p_j$ whose COLLECT on line 7 occurred before the execution of the UPDATE with parameter $\langle req_i, toggle_i \rangle$ (line 1) and so it did not return $req_i$ for the $i$th component[*]. If the `SC` instruction, let it be $SC_1$, that was executed by $req_j$ on $S$ was successful, it may have caused $req_i$'s `SC` on $S$ to fail. In this case, $p_i$ would return without ensuring that its request has been served.

We finally explain why this problem is overcome if the instance of `Attempt` executed by $p_i$ performs lines 6-12 twice. Let $SC_2$ be the first successful `SC` instruction on $S$ executed after $SC_1$. Notice that $SC_2$ is either the second `SC` executed by $p_i$ or $SC_1$ is executed before this `SC`. Then, the thread $p$ which executes $SC_2$ sees $req_i$ and performs it, if this has not already been done. This is so since the matching `LL` of $SC_2$ (and the COLLECT that follows it) are executed by $p$ after $SC_1$ and $req_i$ is announced before $SC_1$.

### 5.1.2 Correctness proof

In this section, we prove that `Sim` is linearizable. We start by introducing some useful notation. Fix any execution $\alpha$ of `Sim`. Assume that some thread $p_i$, $i \in \{1, ..., n\}$, executes $m_i > 0$ requests in $\alpha$. Let $req^i_j$ be the argument[†] of the $j$th invocation of SimApplyOp by $p_i$ and let $\pi^i_j$ be the instance of `Attempt` executed by $req^i_j$. Let $U^i_j$ be the last UPDATE executed by $p_i$ before $\pi^i_j$ and let $Q^i_j$ be the configuration just before the first step of $U^i_j$; let $Q^i_0$ be the initial configuration $C_0$ and let $v^i_j$ be the value written by $U^i_j$. The notation of this proof is summarized in Table 5.1.

[*] For simplicity we sometimes say that a request $req$ by a thread $p_i$ executes `Attempt` (or any other line of the pseudocode) meaning that the instance of SimApplyOp that is called by $p_i$ for $req$ executes `Attempt` (or any code line in reference). Moreover, when we refer to the execution interval of some request $req$, we mean the execution interval of the instance of SimApplyOp that is invoked with parameter $req$.

[†] For clarity of the proof, we consider each $req^i_j$ as distinct.

| Notation | Description |
|---|---|
| $\alpha$ | Any execution of $\mathsf{Sim}$ |
| $C$ | Any configuration in $\alpha$ |
| $C_0$ | The initial configuration of $\alpha$ |
| $p_i$ | The thread which its id is equal to $i$, $i \in \{1, \ldots, n\}$ |
| $m_i$ | Thread $p_i$ executes $m_i$ requests in $\alpha$ |
| $req^i_j$ | The argument of the $j$th invocation of SIMAPPLYOP |
| $\pi^i_j$ | The instance of `Attempt` executed by $reg^i_j$ |
| $U^i_j$ | The last update executed by $p_i$ before $\pi^i_j$ |
| $Q^i_j$ | The configuration just before the first step of $U^i_j$ |
| $Q^i_0$ | The initial configuration $C_0$ |
| $v^i_j$ | The value written by $U^i_j$ |
| $C^i_1$ | The first configuration between $C_0$ and the end of $\pi^i_1$ at which $S.applied[i]$ is equal to 1 |
| $C^i_j$ | The first configuration between the end of $\pi^i_{j-1}$ and the end of $\pi^i_j$ such that $S.applied[i]$ is equal to $j \mod 2$ |
| $SC^i_j$ | The `SC` instruction executed just before $C^i_j$ |
| $LL^i_j$ | The matching `LL` instruction of $C^i_j$ |
| $SC_m$ | The $m$th successful $SC$ instruction on $S$ in $\alpha$ |
| $LL_m$ | The matching `LL` of $SC_m$ |
| $C_m$ | The configuration just after the execution of $SC_m$ |
| $\alpha_m$ | The prefix of $\alpha$ which ends at $SC_m$ |
| $\alpha_0$ | The empty execution |

Table 5.1: Notation used in the proof of $\mathsf{Sim}$.

We first present the following lemma which is an immediate consequence of the pseudocode (lines 5, 6 and 12) and of the semantics of the `LL/SC` operation.

**Lemma 5.1.** *Consider any $j$, $0 < j \leq m_i$. There are at least two successful* `SC` *instructions in the execution interval of $\pi^i_j$.*

The next lemma also follows from the pseudocode (lines 1 and 2). It states that $U^i_j$ updates the $i$th component of $Col$ with the value $\langle req^i_j, j \mod 2\rangle$ and this value does not change until the next UPDATE on the $i$th component starts its execution.

**Lemma 5.2.** *For each $j$, $0 \leq j \leq m_i$, the following claims hold:*

1. *$v^i_j = \langle req^i_j, j \mod 2\rangle$;*

2. *no* UPDATE *occurs on the $i$th component of $Col$ between the end of $U^i_{j-1}$ and $Q^i_j$.*

We next prove that, at the end of the execution of $\pi^i_j$, it holds that $S.applied[i]$ is equal to $j \mod 2$.

**Lemma 5.3.** *Consider any $j$, $0 < j \leq m_i$. It holds that $S.applied[i]$ is equal to $j \mod 2$ at the end of the execution of $\pi^i_j$.*

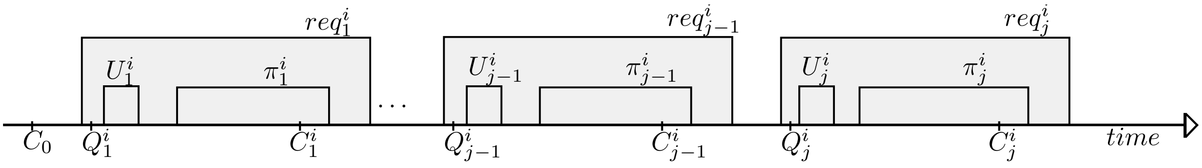


Figure 5.1: An example execution of the Sim algorithm.

*Proof.* Assume, by way of contradiction, that $S.applied[i] = 1 - (j \mod 2)$ at the end of $\pi_j^i$. By Lemma 5.1, there are at least two successful SC instructions in the execution interval of $\pi_j^i$. It follows that the last successful SC instruction that is executed in $\pi_j^i$ writes $1 - (j \mod 2)$ into $S.applied[i]$. Let $\texttt{SC}_x$ be this SC instruction, let $\texttt{LL}_x$ be the matching LL instruction of $\texttt{SC}_x$, let $p_x$ be the thread that executes $\texttt{LL}_x$ and $\texttt{SC}_x$, and let $G_x$ be the instance of COLLECT executed by $p_x$ between $\texttt{LL}_x$ and $\texttt{SC}_x$. Since the execution interval of $\pi_j^i$ contains at least two successful SC instructions and $\texttt{SC}_x$ is the last one and it is successful, it follows that $\texttt{LL}_x$ follows the beginning of $\pi_j^i$.

By the pseudocode (lines 1, 6 and 7), it follows that $G_x$ begins its execution after the end of the execution interval of $U_j^i$. Thus, Lemma 5.2 implies that $G_x$ returns $j \mod 2$ for the *toggle* field of the $i$th component of $Col$. By the pseudocode (line 11), $\texttt{SC}_x$ writes the value $j \mod 2$ into $S.applied[i]$ which is a contradiction. ∎

At $C_0$, $S.applied[i]$ is equal to 0. If $m_i > 0$, Lemma 5.3 implies that at the end of $\pi_1^i$, $S.applied[i]$ is equal to 1. Let $C_1^i$ be the first configuration between $C_0$ and the end of $\pi_1^i$ at which $S.applied[i]$ is equal to 1. Consider any integer $1 < j \leq m_i$. Lemma 5.3 implies that at the end of $\pi_{j-1}^i$, $S.applied[i]$ is equal to $(j-1) \mod 2$, whereas at the end of $\pi_j^i$, $S.applied[i]$ is equal to $j \mod 2$. Let $C_j^i$ be the first configuration between the end of $\pi_{j-1}^i$ and the end of $\pi_j^i$ such that $S.applied[i]$ is equal to $j \mod 2$; let $C_0^i = C_0$. Obviously, $C_j^i$ precedes the end of $\pi_j^i$. Figure 5.1 illustrates the above notation.

By the definition of $C_j^i$, it follows that just before $C_j^i$ a successful SC on $S$ is executed. Let $\texttt{SC}_j^i$ be this SC instruction and let $\texttt{LL}_j^i$ be its matching LL instruction. Denote by $G_j^i$ the instance of COLLECT that is executed between $\texttt{LL}_j^i$ and $\texttt{SC}_j^i$ by the same thread (line 7).

We continue to prove that $G_j^i$ returns the value $v_j^i$ written by $U_j^i$ for thread $p_i$. Moreover, we prove that $\texttt{SC}_j^i$ is executed after $Q_j^i$ (i.e. after the execution of the first step of $U_j^i$).

**Lemma 5.4.** *Consider any $j$, $0 < j \leq m_i$. It holds that: (1) $\texttt{SC}_j^i$ is executed after $Q_j^i$, and (2) $G_j^i$ returns $v_j^i$ for the $i$th component.*

*Proof.* Assume first that $j = 1$. Then, $\texttt{SC}_1^i$ writes 1 to *S.applied*$[i]$; the code (lines 7, 11 and 12) implies that, in this case, $G_1^i$ returns the value 1 for the *toggle* field of the $i$th component of *Col*. However, since the initial value of this field is 0, and $U_1^i$ is the only UPDATE that is executed on the $i$th component between $C_0$ and the end of $\pi_1^i$, it follows that $G_1^i$ returns the value written to the $i$th component by $U_1^i$. Thus, the execution of $G_1^i$ ends after the beginning of $U_1^i$, i.e. after $Q_1^i$, and $G_1^i$ returns $v_1^i$.

Consider now any $j > 1$. Suppose first that $G_j^i$ starts executing before the beginning of $\pi_{j-1}^i$. By the pseudocode, it follows that $\texttt{LL}_j^i$ is executed before $G_j^i$ and, by its definition, $\texttt{SC}_j^i$ is executed after the end of $\pi_{j-1}^i$. By Lemma 5.1, at least two successful SC instructions are executed in the execution interval of $\pi_{j-1}^i$. It follows that $\texttt{SC}_j^i$ is not successful, which is a contradiction. Thus, $G_j^i$ starts its execution after the beginning of $\pi_{j-1}^i$.

We continue to prove that the value $v$ returned by $G_j^i$ for the *toggle* field of the $i$th component of *Col* is not equal to $v_{j-1}^i.toggle$. By definition, $\texttt{SC}_j^i$ writes $j \mod 2$ to *S.applied*$[i]$. Then, by the code (lines 7, 11 and 12), it follows that $G_j^i$ returns the value $v = j \mod 2$ for the *toggle* field of the $i$th component. By Lemma 5.2, $v_{j-1}^i.toggle = (j-1) \mod 2$. Thus, $v \neq v_{j-1}^i$.

By the code (lines 1-4), no UPDATE other than $U_j^i$ is executed on the $i$th component between the end of $U_{j-1}^i$ and the end of $\pi_j^i$. Since $G_j^i$ starts after the beginning of $\pi_{j-1}^i$ (and therefore after the end of $U_{j-1}^i$), ends before the end of $\pi_j^i$, and returns a value not equal to $v_{j-1}^i$, it follows that $G_j^i$ must return the value $v_j^i$ written by $U_j^i$. Therefore, the execution of $G_j^i$ ends after the beginning of the execution of $U_j^i$, and the same is true for $\texttt{SC}_j^i$ which is executed right after $G_j^i$. ∎

We next prove that the value of *S.applied*$[i]$ remains the same between $\texttt{SC}_{j-1}^i$ and $\texttt{SC}_j^i$.

**Lemma 5.5.** *Consider any $j$, $0 < j \leq m_i$. At each configuration $C$ following $C_{j-1}^i$ and preceding $C_j^i$, it holds that S.applied$[i] = (j-1) \mod 2$.*

*Proof.* By definition of $C_j^i$, no successful SC writes the value $j \mod 2$ to *S.applied*$[i]$ between the end of $\pi_{j-1}^i$ and $C_j^i$. Assume, by way of contradiction, that there is some configuration between $C_{j-1}^i$ and the end of $\pi_{j-1}^i$ such that *S.applied*$[i]$ is equal to $j \mod 2$.

Let $C_x$ be the first of these configurations. Since only SC instructions change the value of $S$, there is a successful SC instruction, $\texttt{SC}_x$, which occurs just before $C_x$ and writes the value $j \mod 2$ to $S.applied[i]$. Let $\texttt{LL}_x$ be the matching LL instruction to $\texttt{SC}_x$, let $\pi_x$ be the instance of Attempt that executes $\texttt{SC}_x$, and let $G_x$ be the instance of COLLECT executed by $p_x$ between $\texttt{LL}_x$ and $\texttt{SC}_x$.

We continue to prove that the value $v$ returned by $G_x$ for the *toggle* field of the $i$th component of $Col$ is not equal to $v^i_{j-1}$. By definition of $\texttt{SC}_x$, $\texttt{SC}_x$ writes the value $j \mod 2$ into $S.applied[i]$. Then, by the code (lines 7, 11 and 12), it follows that $G_x$ returns a value $v = j \mod 2$ for the *toggle* field of the $i$th component. By Lemma 5.2, $v^i_{j-1}.toggle = (j-1) \mod 2$. Thus, $v \neq v^i_{j-1}$.

Since $\texttt{SC}_x$ is successful, $\texttt{LL}_x$ must have occurred after $C^i_{j-1}$. Since $G_x$ occurs between $\texttt{LL}_x$ and $\texttt{SC}_x$, and $U^i_{j-1}$ is executed before $\pi^i_{j-1}$, $G_x$ occurs after the end of the execution of $U^i_{j-1}$. Since no other UPDATE occurs on component $i$ between the end $U^i_{j-1}$ and the end of $\pi^i_{j-1}$, it follows that $G_x$ returns $v^i_{j-1}$ for the $i$th component, which contradicts our argument above that $G_x$ returns $v \neq v^i_{j-1}$. ∎

We say that a request $req^i_j$ *is applied* on the simulated state if there is some request $req'$ (that might be $req^i_j$ or not) for which all the following conditions hold: (1) the last COLLECT that is executed by $req'$ returns $\langle req^i_j, toggle\rangle$ as the value of the $i$th component of $Col$, where *toggle* is a value different from the value returned by the last Read on $S.applied[i]$ (line 6) that is executed by the Attempt of $req'$ (so that line 10 is executed), and (2) the execution of the last SC of line 12 on $S$ by $req'$ succeeds. When these conditions are satisfied, we sometimes also say that $req'$ *applies* $req^i_j$.

We continue to prove that $req^i_j$ is applied on the simulated object exactly once and this occurs just before $C^i_j$.

**Lemma 5.6.** *For each $j$, $0 < j \leq m_i$, $req^i_j$ is applied exactly once.*

*Proof.* By the pseudocode (lines 7, 8, 9, and 11) and by definition, it follows that when some request $req$ initiated by $p_i$ is applied, there is some successful SC on $S$ which toggles the value of $S.applied[i]$. Lemmas 5.3 and 5.5 imply that there should be at least one $m > 0$ such that this SC is $\texttt{SC}^i_m$. Since the requests initiated by $p_i$ are distinct, Lemmas 5.2 and 5.4 imply that $req^i_j$ is applied if $m = j$. Thus, $req^i_j$ is applied exactly once. ∎

We are now ready to assign linearization points. For each $i \in \{1, ..., n\}$ and $0 < j \leq m_i$, we place the linearization point of $req^i_j$ at $C^i_j$; ties are broken by the order imposed by threads' identifiers.

**Lemma 5.7.** *Each request $req^i_j$, $0 < j \leq m_i$, is linearized within its execution interval.*

*Proof.* Lemma 5.4 implies that $\texttt{SC}^i_j$ follows $Q^i_j$. By its definition, $\texttt{SC}^i_j$ occurs before the end of $\pi^i_j$. Thus, $C^i_j$ is in the execution interval of $req^i_j$, as needed. ∎

In order to prove consistency, we use the following notation. Denote by $\texttt{SC}_m$, $m > 0$, the $m$th successful `SC` instruction on $S$ and let $\texttt{LL}_m$ be its matching `LL`. Obviously, between $\texttt{SC}_m$ and $\texttt{SC}_{m+1}$, $S$ is not modified.

Denote by $\alpha_m$, the prefix of $\alpha$ which ends at $\texttt{SC}_m$ and let $C_m$ be the first configuration following $\texttt{SC}_m$. Let $\alpha_0$ be the empty execution. Denote by $L_m$ the order defined by the linearization points, assigned as described above, of the requests in $\alpha_m$. We remark that $S.st$ stores a copy of the simulated state at each point in time. Moreover, each thread applies requests on its local copy of the simulated state sequentially, one after the other. We say that $S.st$ is *consistent* at $C_m$ if it equals the state resulting from executing the requests of $\alpha_m$ sequentially in the order specified by $L_m$.

**Lemma 5.8.** *For each $m \geq 0$, (1) $S.st$ is consistent at $C_m$, and (2) $L_m$ is a linearization order for $\alpha_m$.*

*Proof.* We prove the claim by induction on $m$.

**Base case ($m$=0):** The claims hold trivially: by the initialization of $S$, $S.st$ contains $\hat{s}$, which is the initial state of the simulated object, and $\alpha_0$ is empty.

**Induction hypothesis:** Fix any $m > 0$ and assume that the claims hold for $m - 1$.

**Induction step:** We prove that the claim holds for $m$. By the induction hypothesis, it holds that: (1) $S.st$ is consistent at $C_{m-1}$, and (2) $L_{m-1}$ is a linearization order for $\alpha_{m-1}$. Let $req$ be the request that executes $\texttt{SC}_m$. Assume that $req$ applies $j > 0$ requests on the simulated object. Denote by $req_1, ..., req_j$ the sequence of these requests ordered in increasing order of the identifiers of the threads that initiate them.

Notice that $req$ performs $\texttt{LL}_m$ after $C_{m-1}$ since otherwise $\texttt{SC}_m$ would not be successful. By the induction hypothesis, $S.st$ is consistent at $C_{m-1}$. Thus, the local copy of $S$ that is last stored by $req$ in $ls$, represents a consistent state of the simulated object. Lemma 5.6

implies that $req_1, \dots, req_j$ are applied only once. This is realized when $\texttt{SC}_m$ is executed. Thus, none of these requests have been applied previously.

Given that the application of $req_1, ..., req_j$ is simulated by the thread executing $req$ sequentially, in the order mentioned above, starting from the state stored in $ls$, it is a straightforward induction to prove that (1) for each $f$, $0 \leq f \leq j$, a consistent response is calculated for $req_f$, and the new state of the simulated object is calculated in a correct way in the local variable $ls$ of the `Attempt` executed by $req$. Therefore, $S.st$ is consistent after the execution of $req$'s successful `SC`. Notice that, by the way linearization points are assigned, $L_m = L_{m-1}, req_1, \dots, req_j$. It follows that $L_m$ is a linearization order for $\alpha_m$. ∎

**Theorem 5.1.** Sim *is a linearizable implementation of a universal object.*

### 5.1.3 An efficient implementation of COLLECT

We present an implementation of a collect object, called SimCollect, which uses a single `Add` object and has step complexity $O(1)$. However, the size of the `Add` object it employs is large. In Section 5.2, we describe a practical version of SimCollect which has been used by P-Sim to achieve high performance and scalability.

Recall that a collect object consists of $n$ components. Suppose that each of the components stores a value from some set $D$. Suppose that $d$ is the number of bits that are needed for the representation of any value in $D$. SimCollect uses an `Add` object $O$ of $nd$ bits. $O$ is partitioned into $n$ chunks of $d$ bits each, one for each thread. Thread $p_i$ owns the $i$th chunk of $d$ bits, and stores there the value of the component that has been assigned to it. An UPDATE $U$ with value $v$ by $p_i$ first performs an `Add` to ensure that $v$ is written into the $i$th chunk of $O$, and then keeps a copy of $v$ into a local variable; this copy is maintained by $p_i$ to discover the appropriate value that should be added to the $i$th chunk of $O$ during its next UPDATE (which will be the new value minus $v$). Whenever $p_i$ executes a COLLECT, it simply reads the value stored in $O$ and returns for each component the value stored in the corresponding chunk. It is apparent that the number of shared memory accesses performed by SimCollect is 1.

If the size $b$ of an `Add` object is less than $nd$ bits, then we can employ $\lceil nd/b \rceil$ `Add` objects, $O_1, \dots, O_{\lceil nd/b \rceil}$. In this case, the value last written by $p_i$ is represented by the

$(i \cdot d \mod b)$th chunk of $O_{\lceil i \cdot d/b \rceil}$[‡]. An UPDATE by $p_i$ adds an appropriate value to $O_{\lceil i \cdot d/b \rceil}$, and COLLECT reads every Add object once and returns the set of values written in the chunks. This version of the algorithm has step complexity 1 for UPDATE, and $O(nd/b)$ for COLLECT. Notice that this version is not linearizable (but recall that linearizability is not necessary for implementations of collect objects). In case $b \geq nd$, the implementation is linearizable; in this case, SimCollect can serve as a single-writer snapshot implementation.

We remark that the same techniques, as in SimCollect, can be used to get an implementation of an active set, called SimActSet, by using an Add object of $n$ bits (one for each thread); this implementation has step complexity 1 if $b < n$, and $\lceil n/b \rceil$ if $b > n$.

It is apparent that similar implementations of collect, snapshot and active set can be derived if a XOR object is used instead of an Add object.

### 5.1.4 Space and step complexity

The step complexity of Sim is $O(sc)$, where $sc$ is the step complexity of the implementation of the collect object it employs. If this implementation is SimCollect, Sim exhibits constant step complexity. In this case, it uses an Add object of $nd$ bits and an LL/SC object of size $O(n + s)$, where $s$ is the size of the simulated state.

**Theorem 5.2.** *By using* SimCollect*, the step complexity of* Sim *is* $O(1)$ *and* Sim *uses one* Add *object of* $nd$ *bits and one* LL/SC *object of size* $O(n + s)$*.*

### 5.1.5 Derived lower bounds

Jayanti [42] has proved that any oblivious implementation of a universal object from LL/SC objects has step complexity $\Omega(\log n)$. This lower bound holds even if the size of the LL/SC objects is unbounded. Sim is oblivious. So, the lower bound can be beaten if just one Add object (or a collect object) is used in addition to an LL/SC object. Thus, the following theorem holds:

**Theorem 5.3.** *A lower bound of* $\Omega(\log n)$ *holds on the step complexity of any implementation of (1) a single-writer snapshot object, (2) a collect object, (3) a* XOR *object, and (4) an* Add *object, from* LL/SC *objects.*

[‡] For simplicity, we assume that $d$ is a divisor of $b$, so that the $d$ bits allocated to each thread are not split across two Add objects.

## 5.2 P-Sim: A practical version of Sim

In this section, we present a practical version of Sim, which is called P-Sim. P-Sim uses $O(n/b)$ `Add` objects of size $b$ bits each, one `LL/SC` object storing a single pointer, and $O(n)$ `Read-Write` structs each of size $O(n+s)$. The step complexity of P-Sim is $O(n+s)$.

### 5.2.1 Algorithm description

First, we discuss the techniques applied to Sim in order to port it to a real-world machine architecture, like x86_64. Applying these techniques leads to a practical version of Sim, called P-Sim. In P-Sim, the information stored in struct *StRec* is now maintained using indirection; we employ recycling to reduce the space requirements. Each thread $p_i$ maintains a pool of two structs of type *StRec*. These pools are implemented by allocating an array *Pool* of type *StRec* which consists of $n+1$ rows of two elements each. Thread $p_i$'s pool is comprised by the $i$th row of *Pool*[§]. Variable $S$ is now a pointer to one of the elements of *Pool*[¶], initially pointing to $Pool[n+1][1]$ (where the (n+1)st row is used for initialization).

The collect object is implemented by a set of $n$ single-writer `Read-Write` structs of type `Request`, called *Announce*, and a shared bit vector *Toggles* of $n$ bits, one for each thread. A struct of type `Request` contains two fields, a pointer *func* which points to a function containing the code of the simulated operation, and an argument. *Toggles* is implemented using `Add` in a way similar to SimCollect. Specifically, when a thread $p_i$ initiates a new request, it toggles $Toggles[i]$ by performing an atomic `Add` (lines 3-4). More specifically, *Toggles* is implemented as an integer (or as an array of $\lceil n/b \rceil$ integers, if $n$ is larger than the size $b$ of an integer); to toggle bit $i$, $p_i$ atomically adds $2^i$ or $-2^i$ to this integer (or $2^{i \mod b}$ or $-2^{i \mod b}$ to $Toggles[\lceil i/b \rceil]$, respectively). Initially, all bits of *Toggles* are equal to 0.

When $p_i$ wants to execute a request *req*, it announces it by writing *req* (and its parameters) in $Announce[i]$ (line 2). Thread $p_i$ discovers the requests that other active threads want to perform by reading the appropriate entries of *Announce* (lines 14-16)

[§] We remark that in the real code we use a pool of $nC+1$ structs, where $C > 1$ is a small constant, for performance reasons. However, using a pool of just $2n+1$ structs is enough to prove correctness. For code simplicity, Algorithm 9 uses $2n+2$ such structs.

[¶] In the real code, *Pool* is implemented as a one-dimensional array, and $S$ is an index indicating one of its elements.

```
     typedef struct {
         void *func;                              // Function pointer to push or pop
         ArgVal arg;
     } Request;

1    typedef struct {
         State st;
         boolean applied[1..n];                   // applied is implemented as an integer
         RetVal rvals[1..n];
     } StRec;

     shared Integer Toggles = 0;                  // Toggles implements a vector of $n$ bits
     shared StRec Pool[1..n+1][1..2];             // Initially, $Pool[n+1][1] = \langle \perp, 0, \langle \perp, \ldots, \perp \rangle \rangle$
     shared StRec *S = &Pool[n+1][1];             // Initially, $S$ points to $Pool[n+1][1]$
     shared Request Announce[1..n];
```

Algorithm 9: Data structures used in P-Sim.

based on the information Read in *Toggles* and in the struct pointed to by $S$. This increases the step complexity of P-Sim but it decreases the size of the Add object which now stores $n$ bits instead of $nd$ bits that are used in SimCollect. A simplified version of P-Sim is shown in Algorithms 9-10.

The VL instruction of line 11 guarantees that the copied state (line 10) is consistent. A slow thread $p_j$ may read the state of the simulated object form some struct $r$ while thread $p_i$, which owns $r$, reuses this struct. This will have as a result $p_j$ reading an inconsistent state. Notice that, in this case, the SC instruction of $p_j$ on line 18 will fail. Still, $p_j$ may simulate locally the application of several requests , while executing lines 14-17, using an inconsistent state. Successful execution of the VL of line 11 guarantees that $r$ has not yet been reused, so the state that was read is consistent. Additionally, the existence of the VL enhances the performance of P-Sim. Making a local copy of the state (line 10) is slow, since it usually causes one or more cache misses. In the mean time, another thread may have successfully updated the state of the simulated object. In this case, having $p_j$ executing lines 14-17 is useless and may cause cache misses due to the Read operations that are performed on the *Announce* array. The use of the VL ensures that this unnecessary overhead is avoided.

The majority of the commercially available shared memory machines support CAS rather than LL/SC. P-Sim simulates an LL on $S$ with a Read($S$). VL is implemented by reading $S$ and checking whether its timestamp has changed since the most recent previous

```
// Private local variables for thread $p_i$
Integer $toggle_i = 2^i$;
Integer $index_i = 0$;

RetVal PSimApplyOp(Request req, ThreadId i){ // Code for thread $p_i$
    Announce[i] = req;                          // Announce $req$
    FAD(Toggles, $toggle_i$);                   // toggle $p_i$'s bit in $Toggles$
    $toggle_i$ = -$toggle_i$;
    Backoff();
    Attempt(i);
    return S.rvals[i];
}

void Attempt(ThreadId i) {                      // Code for Attempt
    boolean ltoggles[1..n];                     // $ltoggles$ is implemented as an integer
    StRec *ls_ptr;

    for j=1 to 2 do {
        ls_ptr = LL(S);                         // read the pointer stored in S
        Pool[i][$index_i$] = *ls_ptr;           // Create a copy of current state
        if (VL(S) == 0)
            continue;
        ltoggles = Toggles;                     // Read the vector of toggles
        for l=1 to n do {
            // If $p_i$ has a request that is not applied yet
            if(ltoggles[l] $\neq$ Pool[i][$index_i$].applied[l]) {
                // Apply the request and compute return value
                apply Announce[l] on Pool[i][$index_i$].st
                and store the return value into Pool[i][$index_i$].rvals[l];
            }
            Pool[i][$index_i$].applied[l] = ltoggles[l];
        }
        if(SC(S, &Pool[i][$index_i$]))          // Try to change the contents of S
            $index_i$=($index_i + 1$) mod 2;    // If success, $p_i$ uses the next struct
        BackoffCalculate();
    }
}
```

Algorithm 10: Pseudocode of P-Sim.

LL executed by the same thread. Finally, an SC is simulated with a CAS on a timestamped version of $S$ to avoid the ABA problem[‖]. In the real code $S$ stores just an index to *Pool* (and not a full 64 bit pointer), so there are enough bits (in our experiments 48) in a word to store the timestamp. In systems with more threads, we could use 128 bit words; we remark that x86_64 machines support 128 bit words.

[‖] This problem occurs when a thread $p$ reads some value $A$ from a shared variable and then some other thread $p'$ modifies the variable to the value $B$ and back to $A$; when $p$ begins execution again, it sees that the variable has not changed and continues executing normally which might be incorrect.

We remark that the performance of P-Sim becomes better when a combining thread manages to help a large number of other threads while performing its request. For exploiting this property, we use an adaptive backoff scheme. A thread $p_i$ backoffs, after it has announced its request (line 5) and has indicated in *Act* that it is active. P-Sim does not use backoff for reducing the contention on accessing a shared CAS object, as it is usually the case in previous algorithms [37, 50]. It rather employs backoff in an effort to achieve a better combining degree. The backoff scheme of P-Sim is simple: it uses a single parameter which is a backoff upper bound (by default, the backoff lower bound is set to 1). During backoff, a thread executes $t$ noop instructions (where $t$ is initialized to 1). Each time a new request is initiated $t$ is re-calculated as follows. If the SC instruction (line 18) of this request succeeds, $t$ is doubled (until it reaches its upper bound); otherwise $t$ is halved until it reaches its lower bound. In Section 5.3, we discuss the impact of backoff in the performance of P-Sim.

The full source code of P-Sim is provided at http://code.google.com/p/sim-universal-construction/.

### 5.2.2 Correctness proof

The correctness proof of P-Sim is similar to that of Sim presented in Section 5.1.2. We follow the same notation as in Section 5.1.2 where we consider as $U_j^i$ the $j$th Add executed by $p_i$ (line 3) and as the analog of the COLLECT executed on line 7 of Sim, the Read of *Toggles* on line 13 of P-Sim. Let $v_j^i$ be equal to 1 if the argument of the $j$th Add by $p_i$ is positive and 0 otherwise. It is easy to see that, in this way, $v_j^i$ plays the same role as $v_j^i$*.toggle* in the proof of Sim. The notation of this proof is summarized in Table 5.2.

We focus on those parts of the proof of Sim that are different than those of the proof of P-Sim. We start with Lemma 5.1 whose proof is now simpler.

**Lemma 5.9.** *Consider any $j$, $0 < j \leq m_i$. There are at least two successful* SC *instructions in the execution interval of $\pi_j^i$.*

*Proof.* We prove that during the execution of each iteration of the for loop of lines 8-17, at least one successful SC instruction is performed. If the iteration is completed on line 12 of the pseudocode, the VL instruction (line 11) returns 0. This implies that at least one successful SC instruction occurred between the LL of line 9 and the execution of

| Notation | Description |
|---|---|
| $\alpha$ | Any execution of P-Sim |
| $C$ | Any configuration in $\alpha$ |
| $C_0$ | The initial configuration of $\alpha$ |
| $p_i$ | The thread which its id is equal to $i$, $i \in \{1, \ldots, n\}$ |
| $m_i$ | Thread $p_i$ executes $m_i$ requests in $\alpha$ |
| $req_j^i$ | The argument of the $j$th invocation of PSimApplyOp |
| $\pi_j^i$ | The instance of `Attempt` executed by $reg_j^i$ |
| $U_j^i$ | The $j$th `Add` executed by $p_i$ |
| $Q_j^i$ | The configuration just before $U_j^i$ |
| $Q_0^i$ | The initial configuration $C_0$ |
| $v_j^i$ | The value written by $U_j^i$ |
| $C_1^i$ | The first configuration between $C_0$ and the end of $\pi_1^i$ at which $S.applied[i]$ is equal to 1 |
| $C_j^i$ | The first configuration between the end of $\pi_{j-1}^i$ and the end of $\pi_j^i$ such that $S.applied[i]$ is equal to $j \mod 2$ |
| $SC_j^i$ | The `SC` instruction executed just before $C_j^i$ |
| $LL_j^i$ | The matching `LL` instruction of $C_j^i$ |
| $r_j^i$ | The `Read` of $Toggles$ that is executed between $LL_j^i$ and $SC_j^i$ |
| $SC_m$ | The $m$th successful $SC$ instruction on $S$ in $\alpha$ |
| $LL_m$ | The matching `LL` of $SC_m$ |
| $C_m$ | The configuration just after the execution of $SC_m$ |
| $\alpha_m$ | The prefix of $\alpha$ which ends at $SC_m$ |
| $\alpha_0$ | The empty execution |

Table 5.2: Notation used in the proof of P-Sim.

`VL` on line 11. Suppose that the iteration executes the `SC` instruction on line 18. If this `SC` is successful, the claim follows. Otherwise, at least one successful `SC` instruction was performed between the execution of line 9 and line 18. ∎

We next present the analog of Lemma 5.2 of Sim. Its proof is a straightforward induction on $j$.

**Lemma 5.10.** *For each $j$, $0 \leq j \leq m_i$, the following claims hold:*

1. $v_j^i = j \mod 2$ *(i.e. $Toggles[i] = j \mod 2$ after the execution of $U_j^i$);*

2. *no* `Add` *instruction executed between the end of $U_{j-1}^i$ and $Q_j^i$ changes the ith bit of $Toggles$.*

It is easy to prove a lemma similar to Lemma 5.3 for P-Sim. Its proof follows the same arguments as those in the proof of Lemma 5.3.

**Lemma 5.11.** *Consider any $j$, $0 < j \leq m_i$. It holds that $S \rightarrow applied[i]$ is equal to $j \mod 2$ at the end of $\pi_j^i$.*

As in the proof of Sim, we let $C_1^i$ denote the first configuration between $C_0$ and the end of $\pi_1^i$ at which $S \rightarrow \mathit{applied}[i]$ is equal to 1, and we let $C_j^i$ to be the first configuration between the end of $\pi_{j-1}^i$ and the end of $\pi_j^i$ such that $S \rightarrow \mathit{applied}[i]$ is equal to $j \mod 2$; let $C_0^i = C_0$.

We continue to prove that no field of the structure pointed to by $S$ may change its value as long as $S$ points to it. Thus, $S \rightarrow \mathit{applied}[i]$ takes different values only by executing successful `SC` instructions on $S$. It follows that recycling does not cause any implication to the proof.

**Lemma 5.12.** *Let* $\texttt{SC}_1$ *and* $\texttt{SC}_2$ *be two successful* `SC` *instructions on* $S$ *such that no successful* `SC` *on* $S$ *is executed between* $\texttt{SC}_1$ *and* $\texttt{SC}_2$*. Let* $v_1$ *be the value of the structure pointed to by* $S$ *after* $\texttt{SC}_1$*. Then, the value of the structure pointed to by* $S$ *is always* $v_1$ *between* $\texttt{SC}_1$ *and* $\texttt{SC}_2$*.*

*Proof.* Let $C_1$ and $C_2$ be the configurations resulting from the application of $\texttt{SC}_1$ and $\texttt{SC}_2$, respectively. Let $p_i$ be the thread that executes $\texttt{SC}_1$. By the pseudocode (lines 10, 18), it follows that $S$ points to $\mathit{Pool}[i][l]$, for some $l \in \{1, 2\}$ at $C_1$, so $\mathit{Pool}[i][l] = v_1$.

Assume, by way of contradiction, that there is a configuration $C_x$ between $\texttt{SC}_1$ and $\texttt{SC}_2$ at which $\mathit{Pool}[i][l] \neq v_1$. By the pseudocode (lines 17 and 18), it follows that only $p_i$ can write to $\mathit{Pool}[i][l]$. Since $p_i$'s pool contains two structures and $p_i$ uses a different structure each time it performs a successful `SC` on $S$, it follows that $p_i$ can use $\mathit{Pool}[i][l]$ again only if it performs a successful `SC` instruction between $\texttt{SC}_1$ and $C_x$. However, this would contradict our assumption that no successful `SC` instruction is executed on $S$ between $\texttt{SC}_1$ and $\texttt{SC}_2$. ▮

Lemma 5.12 implies that $S \rightarrow \mathit{applied}[i]$ takes different values only when successful `SC` instructions are executed on $S$. It follows that just before $C_j^i$ a successful `SC` on $S$ is executed. Let $\texttt{SC}_j^i$ be this `SC` instruction and let $\texttt{LL}_j^i$ be its matching `LL` instruction. Let $r_j^i$ be the `Read` of $\mathit{Toggles}$ that is executed between $\texttt{LL}_j^i$ and $\texttt{SC}_j^i$ by the same thread (line 13) for the $i$th bit. The following two lemmas are the analogs of Lemmas 5.4 and 5.5 of Sim. Their proofs follow similar arguments as those of these lemmas.

**Lemma 5.13.** *Consider any* $j$*,* $0 < j \leq m_i$*. It holds that* $r_j^i$ *is executed after* $Q_j^i$ *and reads* $j \mod 2$ *in* $\mathit{Toggles}[i]$*.*

**Lemma 5.14.** *Consider any $j$, $0 < j \leq m_i$. At each configuration $C$ following $C^i_{j-1}$ and preceding $C^i_j$, it holds that $S \rightarrow applied[i] = (j-1) \mod 2$.*

We say that a request $req^i_j$ *is applied* if there is some request $req'$ (that might be $req^i_j$ or some other request) for which all the following conditions hold: (1) the last `Read` on *Toggles* that is executed by $req'$ returns a value for its $i$th bit which is different from the value returned by the last `Read` on $S.applied[i]$ (line 6) executed by the `Attempt` of $req'$, (2) the `Read` on $Announce[i]$ (line 16) by $req'$ returns $req^i_j$, and (3) the execution of the `SC` of line 12 on $S$ by $req'$ succeeds. If these conditions hold, we sometimes say that $req^i_j$ is applied when the `SC` of line 12 on $S$ by $req'$ is executed.

Following similar arguments as those in the proof of Lemma 5.6, we can prove that $req^i_j$ is applied on the simulated object exactly once and this occurs just before $C^i_j$.

**Lemma 5.15.** *For each $j$, $0 < j \leq m_i$ $req^i_j$ is applied to the simulated object only once and this occurs just before $C^i_j$.*

*Proof.* By the pseudocode (lines 13, 14, 15, and 17) and by definition, it follows that when some request $req$ initiated by $p_i$ is applied, there is some successful `SC` on $S$ which toggles the value of $S.applied[i]$. Lemmas 5.11 and 5.14 imply that there should be at least one integer $m > 0$ such that this `SC` is $\mathtt{SC}^i_m$. We argue that $req^i_j$ is applied when $\mathtt{SC}^i_j$ is executed. By Lemma 5.13, $r^i_j$ is executed after $Q^i_j$. By definition of $C^i_j$ and by the pseudocode (lines 13 and 18), it follows that $r^i_j$ is executed before the end of $\pi^i_j$. By the pseudocode, it follows that the read of $Announce[i]$ on line 16 by the instance of `Attempt` that executes $\mathtt{SC}^i_j$ occurs between $Q^i_j$ and $C^i_j$. Since $req^i_j$ is active between $Q^i_j$ and $C^i_j$, this read returns $req^i_j$. Thus, $req^i_j$ is applied at least once when $\mathtt{SC}^i_j$ is executed. Since the requests initiated by $p_i$ are distinct, $req^i_j$ is not applied any other time. ∎

We assign linearization points for P-Sim in the same way as we do for Sim. We can then argue, as in Sim, that P-Sim is linearizable. Thus, the following theorem holds for P-Sim.

**Theorem 5.4.** P-Sim *is a linearizable implementation of a universal object.*

### 5.2.3 Space and step complexity

P-Sim performs $O(n + s)$ shared memory accesses. More specifically, the Read of the structure of type *State* which is performed on line 10, results in reading the array *rvals* of $n$ return values, the bit vector *applied* which is stored in $O(n/b)$ memory words, and the entire state of the object, i.e. $s$ memory words. Moreover, the Read of *Toggles* on line 13 requires $O(n/b)$ additional shared memory accesses. The algorithm performs $O(k)$ memory accesses to read the appropriate elements of *Announce*. Thus, the shared memory accesses performed by P-Sim is $O(n + s)$. P-Sim uses a pool of $O(n)$ structures of type *State*, each of size $O(n + s)$. The algorithm also employs a bit vector of size $n$ and an array of $n$ values. Thus, the space complexity of P-Sim is $O(n^2 + ns)$.

**Theorem 5.5.** P-Sim *uses* $O(n/b)$ Add *objects of size* $b$ *bits each, one* LL/SC *object storing a single pointer, and* $O(n)$ Read-Write *structures each of size* $O(n + s)$*. The step complexity of* P-Sim *is* $O(n + s)$*.*

### 5.2.4 Making P-Sim adaptive

In this section, we discuss how we could modify P-Sim in order to get an adaptive version of it in terms of both space and step complexity. The step complexity of P-Sim is determined based on the following: (1) *Toggles* is a vector of $n$ bits, so a Read on it (line 13) causes $O(n/b)$ shared memory accesses, (2) each struct of type *State* contains the simulated state and a vector of $n$ return values, so a Read on it (line 10) causes $O(n + s)$ shared memory accesses, and (3) *Pool* contains $O(n)$ structs. Below, we discuss how we can redesign each of these data structures to get an adaptive version of P-Sim in terms of both space and step complexity.

Herlihy, Luchangco and Moir present in [38, Algorithm 1], an adaptive implementation of a collect object. The step complexity of this implementation is $O(k)$ for COLLECT and $O(1)$ for UPDATE, where $k$ is the total contention. Its space complexity is $O(k)$.

To avoid maintaining a vector of $n$ bits, we can replace *Toggles* with the collect implementation of [38, Algorithm 1]. Whenever a thread $p$ wants to perform a request, it calls UPDATE to write to its component the value of its bit and its request instead of recording its request on *Announce* and executing an Add on *Toggles* to update the value of its assigned bit (line 3). The Read of the bit vector performed on line 13 is replaced with a COLLECT on the collect object. This COLLECT returns the set of bits and the requests of all active threads.

Instead of storing an array of $n$ return values, a set of structs (one for each thread that has taken steps thus far) is maintained. Each of these structs contain a return value, a thread identifier, and a toggle bit which is used to identify if a new request has been initiated by this thread. Whenever a thread executes a request for the first time, the set is updated with a struct containing the thread's id. The set can be trivially implemented as a linked list given that each thread works on its own copy of this list. By applying these techniques the size of struct *State* is reduced to $O(k + s)$ and making a copy of it costs $O(k + s)$.

By applying the two techniques discussed above, the step complexity of P-Sim becomes $O(k + s)$. However, the space complexity is still a function of $n$ since *Pool* still contains $n+1$ structs. Instead of allocating *Pool* statically at the beginning of the execution, each thread dynamically allocates its two structs when it executes its first request. Instead of storing an array of $n$ return values, each struct stores a pointer which will point to the first element of the list of return values. Thus, the number of such structs that are allocated is $2k+1$, each of size $O(s)$. $S$ is a pointer to one of the elements of *Pool*. When a thread makes a local copy of a struct pointed by $S$, it should also make a local copy of the current list of return values which is pointed to by the appropriate field of $S$. Thus, the memory overhead for each thread is $O(k+s)$. Therefore, by applying these techniques the space complexity becomes $O(k(k + s))$.

The collect implementation presented in [38, Algorithm 1] has the disadvantage that its step and space complexity is a function of the total contention since it cannot free the memory that is not used any more. The collect implementation presented in [38, Algorithm 2] could be used instead, the step and space complexity of which adapt to operation's complexity. However, the last implementation uses primitives that are not supported by real machines. In case primitives are simulated using CAS instructions, the resulting algorithm would not satisfy the wait-freedom property.

## 5.3 Performance evaluation of P-Sim

We run our experiments on a 32-core machine consisting of four AMD Opteron 6134 processors (Magny Cours). Each processor consists of two dies and each of them contains four processing cores and an L3 cache shared by its cores. Dies and thus processors are connected to each other with Hyper Transport Links creating a topology with an average diameter of 1.25 hops [21]. For the experiments presented here, we used the latest version of the code of P-Sim (version 1.2) [45]. All codes were compiled with *gcc* 4.3.4, and the Hoard memory allocator [18] was used to eliminate any bottlenecks in memory allocation. The operating system was Linux with kernel 2.6.18. Thread binding was used in order to get more reliable performance results; the $i$-th thread was bound to the $i$-th core of the machine. In this way, we exploited first multi-core, then multi-chip and then multi-socket configuration.

We first focus on a micro-benchmark which shows the performance advantages of P-Sim over well-known synchronization algorithms. We have chosen to simulate a simple `Fetch&Multiply` operation as a case study; each algorithm has simulated the execution of $10^7$ `Fetch&Multiply` requests for different values of $n$, where each thread initiated $10^7/n$ such requests. We measured the average throughput (i.e. the number of `Fetch&Multiply` simulated per second) that each algorithm has exhibited. Specifically, the horizontal axis of Figures 5.2-5.12 represents the number of threads $n$, and the vertical axis represents the throughput (in millions of requests executed per second) that each synchronization algorithm has performed. For each value of $n$, the experiment has been performed 10 times and averages have been calculated. Between two `Fetch&Multiply` requests by the same thread, we have inserted a random number (up to 512) of dummy loop iterations in order to simulate a random work load large enough to avoid unrealistically low cache miss ratios and long runs; we remark that this load is not big enough to reduce contention. A similar technique is employed by Michael and Scott in [50] for the same reasons. The performance behavior of our algorithms for different values of random work (Figure 5.6) will be discussed later.

We have performed this experiment to measure the performance of the following algorithms: CLH spin locks[**] [23, 47], a simple lock-free algorithm with exponential backoff,

[**] We experimentally saw that MCS spin locks [49] have similar performance to CLH spin locks, so we present our results only for CLH locks.

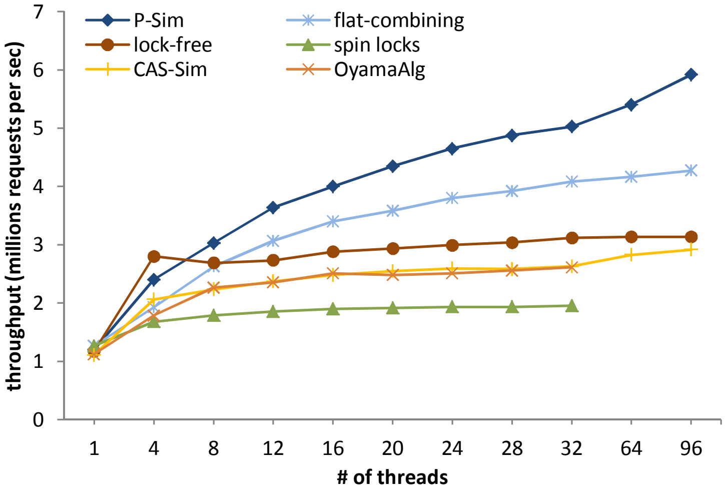


Figure 5.2: Performance of P-Sim.

flat-combining [33, 34], and OyamaAlg [52]. The simple lock-free algorithm uses a single CAS object $O$, and executes CAS on $O$ repeatedly until it successfully stores the new value into it; the algorithm employs an exponential backoff scheme to reduce contention. We also implemented a version of P-Sim, called CAS-Sim, where Add is simulated in a lock-free way using a CAS object, as in the simple lock-free algorithm discussed above.

We carefully optimized these algorithms in our computing environment; for those that use backoff schemes, we performed a large number of experiments to select the best backoff parameters in each case. CLH spin locks and OyamaAlg have been evaluated for only up to 32 threads (so that each thread runs on a distinct core), since otherwise they result in poor performance. We used the flat-combining implementation provided by its inventors [33, 34] and we applied similar optimizations on its code as for that of P-Sim; we also carefully chose its parameters (i.e. polling level, number of combining rounds) to optimize its performance in our computing environment.

Figure 5.2 shows the results of our experiment. P-Sim has been proved to be up to 2.5 times faster than spin-locks, and up to 1.7 times faster than the simple lock-free algorithm. We remark that both P-Sim and flat-combining implement the combining technique, so we expect both of them to enjoy the performance benefits of this technique. However, flat-combining is blocking whereas P-Sim is wait-free. Since wait-freedom is expected to come with some overhead, our first goal was to design a wait-free implementation of the combining technique that performs the same well as flat-combining (which is however blocking). Figure 5.2 shows that P-Sim achieves this goal and even performs slightly better

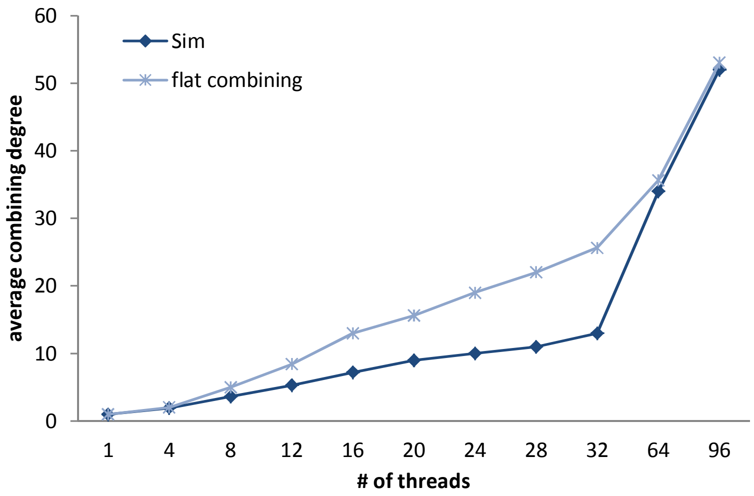


Figure 5.3: Average combining degree of P-Sim and flat-combining for different numbers of threads.

than flat-combining; it exhibits up to 1.20 times better throughput than flat-combining. Finally, P-Sim outperforms OyamaAlg by a factor of up to 1.9.

As illustrated in Figure 5.2, when $n > 4$, the simple lock-free algorithm causes a lot of contention and exhibits performance much worse than P-Sim or flat-combining. However, it behaves well for up to 4 threads since then all the communication occurs within the same die, which is much faster than achieving intra-communication between dies. As expected for queue-locks, the performance of CLH remains almost the same as $n$ increases. For up to some number of threads, the performance of P-Sim and flat-combining is getting better as the number of $n$ increases. This is so, since the average degree of combining that is achieved increases with the number of active requests in the system. We remark that this enhancement in performance is noticed even for values of $n > 32$ where the processing cores are over-subscribed. In contrast, OyamaAlg achieves lower throughput. This is so since in OyamaAlg threads need to succeed on a CAS in order to have their requests announced; this causes a lot of contention and leads to a significant performance degradation. It is worth pointing out that P-Sim is at least 2 times faster than CAS-Sim which, not surprisingly, exhibits similar performance to OyamaAlg; in CAS-Sim, as in OyamaAlg, a thread repeatedly executes CAS to announce a request and therefore CAS-Sim faces a similar performance penalty for the announcement of the requests as OyamaAlg.

Figure 5.3 shows the average number of requests, called average *combining degree*, that are executed by the combiners in P-Sim and flat-combining. Specifically, to calculate

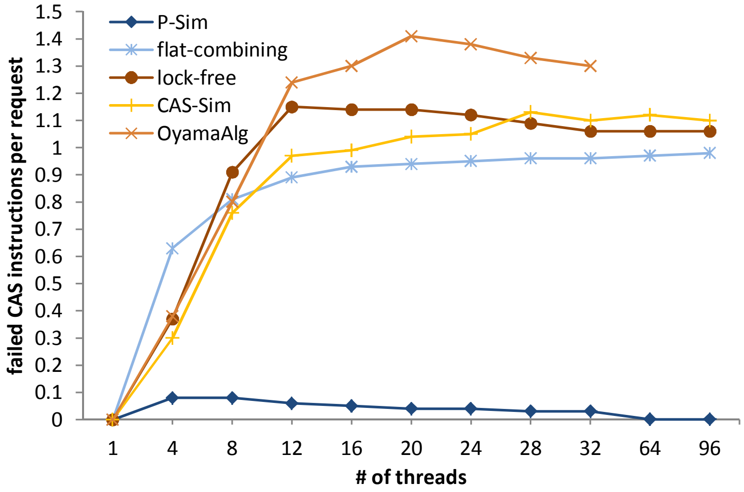


Figure 5.4: Average number of failed CAS instructions per request for different numbers of threads.

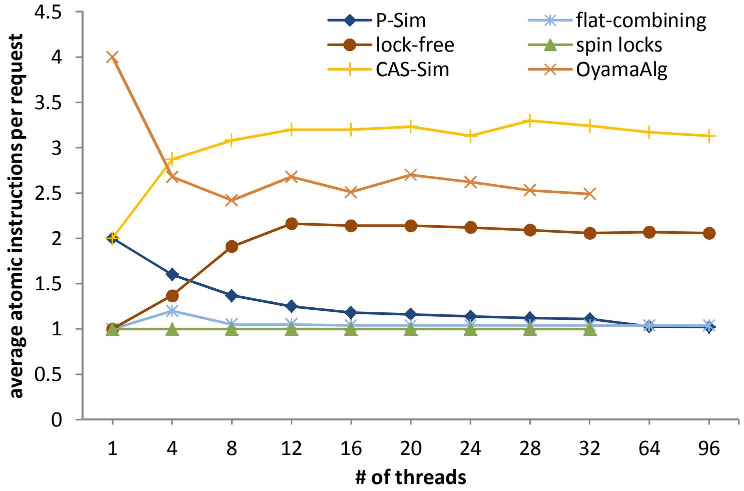


Figure 5.5: Average number of atomic instructions (excluding Read and Write operations) per request performed by P-Sim for different numbers of threads.

the average combining degree of Sim, we add the number of requests that are applied each time a successful CAS on $S$ is executed and we divide this sum by the total number of successful CAS instructions. As shown in Figure 5.3, flat-combining achieves better combining degree in some cases. This is expected since no more than $n$ requests (one per thread) can be applied in P-Sim, each time a CAS on $S$ succeeds. In contrast, in flat-combining, a combiner may apply several requests of the same thread; this may happen, if the thread initiates a new request before the combiner processes all other requests of the request list. Because of this, it is reasonable to expect that flat-combining avoids moving cache lines between the processing cores which is good in terms of performance. However, the performance of flat-combining is not better than that of Sim, since the advantage in the achieved combining degree of flat-combining is counterbalanced by other performance factors that are discussed below. Moreover, as shown in Figure 5.3, when the processing cores are lightly over-subscribed, the combining degree of P-Sim matches the combining degree of flat-combining. As shown in Figure 5.2, this results in better performance for P-Sim.

Figure 5.4 shows the average number of failed CAS instructions executed per request. Notice that a large number of requests in P-Sim do not execute any unsuccessful CAS instructions; this is mainly due to the validation that is performed on line 11. So, the average number of unsuccessful CAS per request in P-Sim is very small. In contrast, this number is close to one (or larger) for all other algorithms. The large number of unsuccessful CAS instructions executed in flat-combining occur during the acquisition of the global lock. This results in a performance degradation. We remark that our efforts to overcome this problem by increasing the number of combining rounds[††] (which reduces the number of times the global lock is acquired) did not result in better performance. This was so because after the first few combing rounds, flat-combining spent a lot of time reading records of threads with no announced requests. On the contrary, P-Sim does not perform non-useful Read operations.

Figure 5.5 shows the average number of atomic instructions (other than Read and Write operations) per request that each algorithm executes. For large values of $n$, P-Sim and flat-combining execute almost the same number of atomic instructions per request on average. For smaller values of $n$, flat-combining executes slightly less atomic instructions per request on average than P-Sim. However, all the atomic instructions executed in flat-combining are CAS instructions on a single shared variable that implements the global lock. In contrast, the atomic instructions that are executed by any request in P-Sim are not applied on the same base object (one of them is an Add and if there is any other it is a CAS on $S$). Moreover, the release of the global lock in flat-combining have not been taken into consideration in the diagram of Figure 5.5 (since it is implemented with a Write). However, these Write instructions cause more contention on the global lock and additional cache misses.

[††] The number of combining rounds determines how many times the combiner (in flat-combining) traverses the request list before it gives up serving other requests.

| Algorithm | average cpu cycles spent in cpu stalls per request |
|---|---|
| P-Sim | 6121 |
| flat-combining | 6810 |

Table 5.3: Average cpu cycles spent in cpu stalls per request for P-Sim and flat-combining for $n = 16$.

It is worth pointing out that the failed `CAS` instructions may cause branch miss-predictions which are expensive since modern microprocessors usually have deep pipelines. Table 5.3 shows that flat-combining pays more (in cpu cycles) for stalls due to cache misses and branch miss-predictions.

In the experiment illustrated in Figure 5.6, we study the behavior of the evaluated algorithms for different amounts of random work, i.e. for different numbers of dummy loop iterations inserted between the executions of two `Fetch&Multiply` by the same thread. We fix the number of threads to 32 and we perform the experiment for several different random work values ($0-8192$). Figure 5.6 shows that, for a wide range of values ($64-2048$), there are no big differences on the throughput exhibited by the evaluated algorithms. The reason for this is that for all these values the synchronization cost is the dominant performance factor. For small values of random work ($0 - 64$), the simple lock-free algorithm achieves unrealistically high throughput. The reason for this is that a thread can uninterruptedly perform thousands of `Fetch&Multiply`. This phenomenon is known as a *long run*; as discussed in previous work [50], such runs are unrealistic workloads. A similar behavior, but in smaller scale, is observed in flat-combining. In cases that the random work is too high (greater than 4096), the throughput of all algorithms degrades and the performance differences among them become minimal since the amount of random work becomes then the dominant performance factor.

Table 5.4 studies the throughput of P-Sim for different values of the backoff upper bound. The performed experiment is the same `Fetch&Multiply` experiment studied in Figure 5.2, where $n = 32$ and for a maximum workload of 512. The first row of Table 5.4, shows the throughput achieved by P-Sim for different values of the backoff upper bound as well. Notice that the best throughput is achieved when the backoff upper bound is equal to 1000 dummy loop iterations. The second row of this table The third row shows how much each of these values diverge from the optimal backoff upper bound value (of 1000) and line 4 shows the performance degradation in each case. Notice that the performance

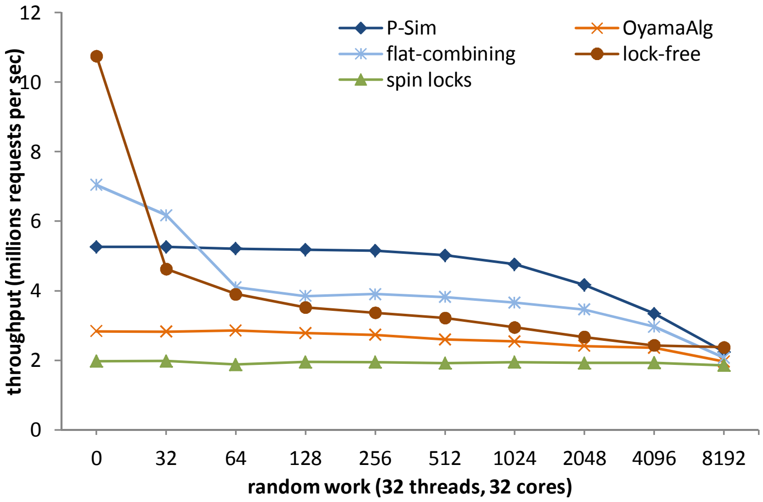


Figure 5.6: Performance of P-Sim for different values of random work.

of P-Sim is very tolerant to overestimated values for the backoff upper bound. More specifically, even a value greater by 120% than the optimal backoff upper bound cause a performance drop of just 12.5%. However, P-Sim is less tolerant to smaller backoff values. Notice that a value for the backoff upper bound smaller by 40% than the optimal causes a 22.7% performance drop. Thus, if the amount of workload cannot be determined precisely, overestimated backoff upper bounds is the preferable choice.

Our next experiment, illustrated in Figure 5.7, studies the performance of P-Sim and flat-combining (the best two algorithms in terms of performance) when $n$ takes values larger than 96, i.e. when a large number of threads are active and the system is heavily over-subscribed. The active threads execute $10^7$ `Fetch&Multiply` in total, as in the previous experiments. Figure 5.7 shows that both P-Sim and flat-combining scale well up to thousands of threads.

Figure 5.8 illustrates how P-Sim performs in cases where an application initiates a large number of threads from which only a small percentage are active, at any given point in time. Specifically, we consider systems where the total number of threads ranges from

| backoff upper bound (in hundreds of dummy loop iterations) | 6 | 8 | 10 | 12 | 14 | 18 | 22 |
|---|---|---|---|---|---|---|---|
| throughput | 4.10 | 5.00 | 5.03 | 5.02 | 4.95 | 4.70 | 4.47 |
| % divergence from backoff upper bound | −40% | −20% | 0% | +20% | +40% | +80% | +120% |
| % performance drop | 22.7% | ≈ 0% | 0% | ≈ 0% | 1.6% | 7% | 12.5% |

Table 5.4: Sensitivity of P-Sim to the backoff upper bound parameter.

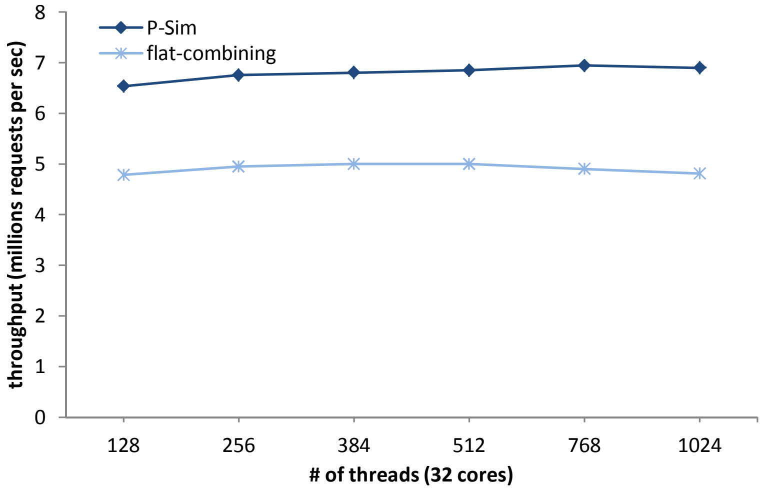


Figure 5.7: Performance of P-Sim for large numbers of threads.

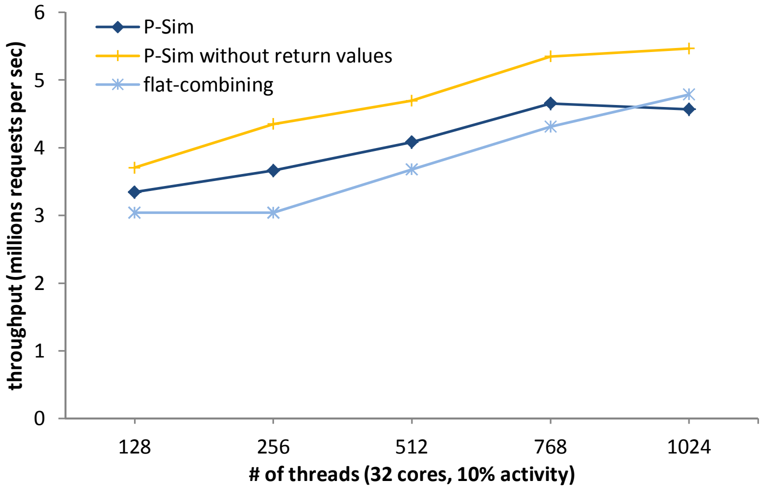


Figure 5.8: Performance of P-Sim when a large number of threads are initiated but only 10% are active.

128 to 1024 and only 10% of them are active at each point in time. The active threads execute $10^7$ `Fetch&Multiply` in total, as previously. We remark that this experiment is in favor of flat-combining: P-Sim requires to read all $n$ bits of the `Add` object and copy locally $n$ return values independently of how many of the threads are active, whereas in flat-combining inactive threads cause no overhead. Figure 5.8 shows there is indeed a small performance advantage (by a factor of 1.05) of flat-combining in this case. In order to discover the main overheads of P-Sim in this case, we have implemented an additional version of it where no return values are calculated. Figure 5.8 shows that the calculation of the return values is indeed an expensive part of the computation performed by P-Sim. This shows that the overhead of having each thread performing an `Add` per request does

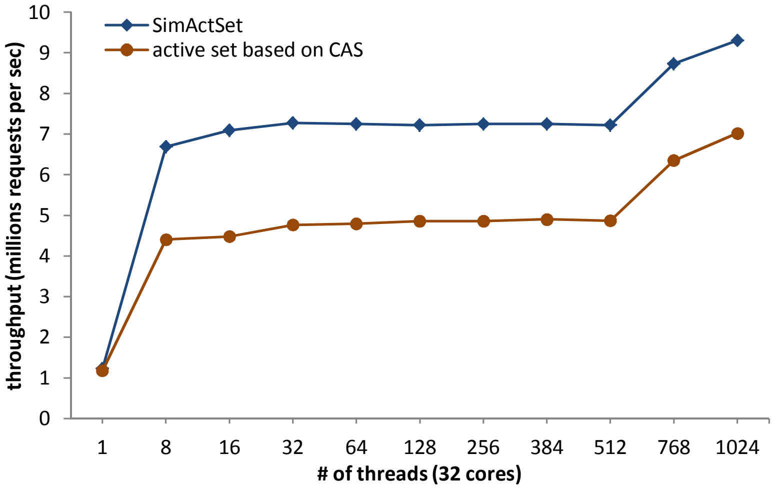


Figure 5.9: Performance of SimActSet.

not cause any significant overhead even for large values of $n$. We remark that in the implementation of several shared objects some of the simulated operations do not have a return value. For instance, a push on a stack or an enqueue on a queue, etc., do not require the calculation of a return value. We remark that in cases where the percentage of active threads is larger than 10%, P-Sim achieves much better performance than that shown in Figure 5.8.

We next explore in more detail the performance characteristics for the Add instruction. We compare the performance of SimActSet (discussed in Section 5.1.3) to that of a simple lock-free active set implementation that uses CAS objects to store a set of $n$ bits, one for each thread. Specifically, as in SimActSet, the algorithm uses $n/b$ CAS objects. Whenever a thread wants to apply a JOIN (LEAVE), it repeatedly executes CAS on the appropriate object until it succeeds to change its bit to 1 (0, respectively). GETSET simply reads the CAS objects. An exponential backoff scheme is used to increase the performance of the lock-free implementation.

We executed $10^7$ JOIN and LEAVE requests, and $10^7$ GETSET requests in total; each thread executed $10^7/n$ JOIN or LEAVE requests and $10^7/n$ GETSET requests. We measured the average throughput exhibited by each technique. To study the scalability of our technique, we consider systems where the total number of threads is large, i.e. it ranges from 128 to 1024. Again, a random number of (up to 512) dummy loop iterations are executed between the execution of two requests by the same thread. Figure 5.9 illustrates that SimActSet outperforms the active set based on CAS by a factor of up to 1.7. This is due to the fact that GETSET causes a small number of cache misses in SimActSet, whereas the repeated execution of a CAS results in a larger number of cache misses.

Several modern shared memory machines (e.g. those that employ the x86_64 architecture) include an atomic Add (CAS) instruction on up to 64 bit words in their instruction set. In order to cope with more than 64 threads efficiently, we have implemented the multi-word bit vector of Add (and CAS, for the lock-free algorithm) by storing its words to the minimum possible number of cache lines. In this way, GETSET causes a minimum number of cache misses. Notice that the size of a typical cache line is usually 64 bytes; thus, a single cache line can store one bit for each of up to 512 threads. So, GETSET causes more than one cache miss only if the number of threads is more than 512.

As illustrated in Figure 5.9, the throughput of both algorithms does not change for values of $n$ greater than 16 and smaller than 512, whereas it improves for values larger than 512. This is because all toggle bits that comprise the active set fit in one cache-line in case that $n < 512$. Thus, all processing cores access the same cache line in this case which results in a lot of contention. On the other hand, when $n > 512$, the processing cores work on two different cache-lines which reduces the contention (but increases the number of cache misses). Figure 5.9 shows that in this experiment the contention is the dominant performance factor.

## 5.4 L-Sim: A synchronization algorithm for large objects

In this section, we present a variation of Sim, called L-Sim, which avoids copying the entire state and it can be used to handle objects with large (or even unbounded) state.

### 5.4.1 Algorithm description

Similarly to P-Sim, L-Sim (Algorithms 11 and 12) employs a shared vector *Toggles* of $n$ bits (one bit for each thread) and during its $j$th request, $j > 0$, thread $p_i$ adds $2^i$ or $-2^i$ to *Toggles* depending on whether $j \mod 2 = 1$ or not. L-Sim also employs a set of $n$ single-writer base objects (*Announce* array on Algorithm 11), one base object for each thread. Each thread starts the execution of a request *req* by announcing *req* in its single-writer base object on *Announce* array (line 1) and by adding $2^i$ or $-2^i$ to *Toggles* (line 3).

The main difficulty in designing L-Sim was to ensure that at each point in time, all "up-to-date" threads (i.e. those that have read the current version of *State*) that are active and execute some request will help the same set of requests. This is achieved by storing in *State* ($S$) two versions of the *applied* bit vector (the first one is called *applied*, while the second one is called *papplied*). Each time an instance $A$ of `Attempt` is executed, *papplied* is updated to store the values found in *applied* at the beginning of $A$ (line 15); *applied* is updated based on the values recorded in *Toggles* (line 16). Whether a request by a thread $p_i$ should be applied or not is determined based on the values read in the $i$-th entry of the arrays *applied* and *papplied* of $S$; if the $i$-th entry of *applied* is different than the $i$th entry of *papplied* (i.e. $applied[i] \neq papplied[i]$) then the request of thread $p_i$ has not been applied yet and it should be simulated (line 18); otherwise, the request (if any) has already been applied. In the initial value of $S$, both *applied* and *papplied* contain *false* in all their entries. Thus, the first application of a successful *Attempt* will result in the simulation of no requests. However, the execution of a successful *Attempt* stores in $S$ information about the requests that should be simulated by the threads that will read the new value of $S$. Therefore, all these threads will try to simulate the same set of requests.

In contrast to Sim and P-Sim, the state of the simulated data structure is now shared and it can be updated directly by any thread. For each data item $x$, L-Sim maintains a struct of type *ItemSV*. This struct stores the old and the current value of the data, a toggle bit that identifies the position in the *val* array of the struct where the current data for $x$ should be read from, and a sequence number. Consider that two threads $p$ and $q$ simulate the same request *req*. It may happen that $p$ is at some earlier point of its execution (e.g., just before executing line 29), whereas $q$ has finished the simulation of *req* (lines 39-43) and has started updating the shared data structure. Then, it could happen that $p$ reads the updated version for a data item although it should have read the old version. For this reason, $q$ stores the old value (additionally to the current value) in one of the entries of *val* array and uses the toggle bit appropriately to indicate the updated version. If $p$ discovers that this bad scenario has occurred (line 33), it reads the old value of the data item found in the $1 - toggle$ entry of its val array. Notice that $p$ should continue executing *req* to ensure wait-freedom (i.e. to help $q$ in case it fails

```
 struct NewVar {                        // list of newly allocated variables
    ItemSV *var;                        // points to the actual struct of the variable
    NewVar *next;                       // points to the next element of the list
};
struct NewList {                        // a stack object
    ItemSV *first;
};
struct State {                          // this struct is stored in a single base object
    boolean applied[1..n], papplied[1..n];
    int seq;
    NewList *var_list;                  // indirection to a shared stack
    RetVal RVals[1..n];                 // return values
};
struct DirectoryNode {
    Name name;                          // variable name
    ItemSV *sv;                         // data item
    Value val;                          // new value of the data item
};
struct ItemSV {
    Value val[0..1];                    // old and new values of data item
    int toggle;                         // toggle shows which of val[0..1] is the current value
    int seq;
};

shared Integer Toggles = $< 0, ..., 0 >$;      // Toggles is implemented as an integer of $n$ bits
shared State S = $< F, ..., F >, < F, ..., F >, 0, < \perp >, < \perp, ..., \perp >>$;
shared OpType Announce[1..n] = {$\perp$, ..., $\perp$};

// Private local variables for thread $p_i$
Integer $toggle_i = 2^i$;

    RetVal ApplyOp(request req){               // Pseudocode for thread $p_i$
1       Announce[i] = req;                     // Announce request $req$
2       $toggle_i$ = -$toggle_i$;
3       Add(Toggles, $toggle_i$);              // $2^i$ is added to toggle $p_i$'s bit
4       Attempt();
5       Attempt();                             // call Attempt to perform req
6       return S.rvals[i];                     // $p_i$ returns
    }
```

Algorithm 11: Data structures used in L-Sim and pseudocode for LSimApplyOp.

without having performed all the required updates). The *seq* field is used to discover whether a helper is already obsolete. ‘ For each set of simulated requests “listed” in $S$, the required updates are first performed by each thread $p_i$ in local copies of the data items accessed (lines 19-37), and only later they are applied to the shared data structure (lines 39-43). To implement this, each thread $p_i$ uses a local directory $D$ containing structs of type $DirectoryNode$, where it stores information about each item it accesses during the execution of its current instance of Attempt (lines 33, 34), and performs all its updates first on these copies (line 36). Only after it has finished the simulation of the

```
void Attempt()(request req){                               // pseudocode for thread $p_i$
        Pindex q, j;
        State ls, tmp;
        Set lact;
        DirectoryNode $D$;
        NewVar *pvar = new NewVar(), *ltop;
        ItemSV sv, *psv = new ItemSV();
7       psv→ ⟨val, toggle, seq ⟩ = $<< \perp, \perp >, 0, 0 >$;
8       pvar→ ⟨var, next, ⟩ = <psv, null>;
9       for j=1 to 2 do {
10          D = ∅;                                         // initialize direcory D
11          ls = LL(S);                                    // read State struct
12          lact = Toggles;                                // read active set
13          ltop = ls.var_list→first;                      // read pointer to the current variable list
14          tmp.seq = ls.seq + 1;
15          tmp.papplied[1..n] = ls.applied[1..n];
16          tmp.applied[1..n] = lact[1..n];                // p will attempt to update S with tmp
17          for q=1 to n do {                              // local loop
18              if (ls.applied[q] ≠ ls.papplied[q]) {      // apply request of thread q
19                  foreach access of a variable x while applying request Announce[q]{
20                      if (x is a newly allocated variable) {
21                          if(CAS(ltop→next, null, pvar)){
22                              psv = new ItemSV();
23                              psv→ ⟨val, toggle, seq ⟩ = $<< \perp, \perp >, 0, 0 >$;
24                              pvar = new NewVar();
25                              pvar→ ⟨var, next, ⟩ = <psv, null>;
                            }
26                          ltop = ltop→next;              // in any case, use ltop → next as the new variable's metadata
27                          add <x, ltop→var, ltop→var.val[0]> to D;              // add variable to local dictionary
28                      } else {                                                  // x is not a newly allocated variable
29                          let svp be a pointer to the ItemSV struct for x;
30                          if (this access is a read instruction) {
31                              if (x exists in D) read x from D; // perform the request on the local copy of x (if any)
32                              else {sv = LL(*svp);
33                                  if (tmp.seq == sv.seq) add <x, svp, sv.val[1-sv.toggle]> to D;
34                                  else if (tmp.seq > sv.seq) add <x, svp, sv.val[sv.toggle]> to D;
35                                  else goto Line 38;     // the State read by p is obsolete, start from scratch
                                }
36                          } else if (this access is a write instruction) update x in D; // perform request on local copy
                        }
                    }
37                  store into tmp.rvals[q] the return value;
                }
            }
38          if (!VL(S)) continue;                          // the State read by p is obsolete, start from scratch
39          foreach record <x, svp, v> in D {
40              if(svp→seq > tmp.seq) break;               // if all requests have been applied, return
41              else if(svp→seq == tmp.seq) continue;                             // if the variable is modified, continue
42              else if(svp→toggle == 0) SC(*svp, <<svp→val[0],v>, 1, tmp.seq>);// make update visible
43              else SC(*svp, <<v, svp→val[1]>, 0, tmp.seq>);                     // make update visible
            }
44          tmp.var_list = new List(); tmp.var_list→first = null;                 // re-initiate tmp.var_list
45          SC(S, tmp);                                                           // try to modify S
        }
    }
```

Algorithm 12: Pseudocode for L-Sim.

set of requests described in *Announce*, it applies the changes listed in the elements of its directory to the shared data structure (lines 39-43).

Some additional synchronization that should be achieved between different helpers of the same set of requests is when new data items are allocated by these requests; Then, all helpers should use the same allocated ItemSV struct for each of these data items. To solve this problem, $S$ stores a pointer (called *var_list*) to a list of newly created data items shared by all threads that read this instance of $S$. Each time a thread $p_i$ needs to allocate the $k$-th, $k \geq 1$, such data item, it tries to add a struct of type $NewVar$ as the $k$-th element of the list (line 21). If it does not succeed, some other thread has already done so, so $p$ uses this struct (by moving pointer *ltop* to this element on line 13, and by inserting $ltop \rightarrow var$ in its dictionary on line 27).

### 5.4.2 Correctness proof

In this section, we present the correctness proof of L-Sim. We start by introducing a similar notation to that of Section 5.2.2. Let $\alpha$ be any execution of L-Sim and assume that some thread $p_i$, $i \in \{1, ..., n\}$, executes $m_i > 0$ requests in $\alpha$. Let $req^i_j$ be the argument of the $j$th call of L-Sim by $p_i$ and let $\pi^i_j$ be the $j$th instance of `Attempt` executed by $p_i$ in $\alpha$ (see Figure 5.10). Define as $Q^j_i$ `Add` instruction of line 3; let $Q^i_0 = C_0$. We use $Toggles[i]$, $i \in \{1, \dots, n\}$, to denote the $i$-th bit of $Toggles$. We denote by $toggle^i_j$ the value of $p_i$'s persistent local variable $toggle_i$ at the end of $req^i_j$. The notation of this proof is summarized in Table 5.5.

**Lemma 5.16.** *Consider any $j$, $0 < j \leq m_i$. There are at least two successful* `SC` *instructions in the execution interval of $\pi^i_j$.*

*Proof.* We prove that during the execution of each iteration of the for loop of line 9, at least one successful `SC` instruction is performed. If the iteration is completed on line 38 of the pseudocode, the `VL` instruction returns *false*. This implies that at least one successful `SC` instruction occurred between the `LL` of line 11 and the execution of `VL` on line 38. Now, suppose that the iteration executes the `SC` instruction on line 45. If this `SC` is successful, the claim follows. Otherwise, at least one successful `SC` instruction was performed between the execution of line 11 and line 45. ∎

The following observation is an immediate consequence of the pseudocode (line 2).

**Observation 5.1.** *Consider any $j$, $0 \leq j \leq m_i$. The following claims hold:*

1. *if $j \mod 2 = 0$, $toggle^i_j = 2^i$;*

2. *if $j \mod 2 = 1$, $toggle^i_j = -2^i$.*

| Notation | Description |
|---|---|
| $\alpha$ | Any execution of L-Sim |
| $C$ | Any configuration in $\alpha$ |
| $C_0$ | The initial configuration of $\alpha$ |
| $p_i$ | The thread which its id is equal to $i$, $i \in \{1, \ldots, n\}$ |
| $m_i$ | Thread $p_i$ executes $m_i$ requests in $\alpha$ |
| $req_j^i$ | The argument of the $j$th invocation of LSimApplyOp |
| $\pi_j^i$ | The $j$th instance of `Attempt` executed by $p_i$ in $\alpha$ |
| $Q_j^i$ | The `Add` instruction of line 3 |
| $Q_0^i$ | The initial configuration $C_0$ |
| $Toggles[i]$ | The $i$-th bit of $Toggles$ |
| $toggle_j^i$ | The value of $p_i$'s persistent local variable $toggle_i$ at the end of $req_j^i$ |
| $C_l$ | The configuration just after the execution of the $l$th `Add` in $\alpha$ |
| $C_1^i$ | The first configuration between $C_0$ and the end of $\pi_1^i$ at which $S.applied[i]$ is equal to 1 |
| $C_j^i$ | The first configuration between the end of $\pi_{2j-2}^i$ and the end of $\pi_{2j-1}^i$ such that $S.applied[i]$ is equal to $j \mod 2$ |
| $SC_j^i$ | The `SC` instruction executed just before $C_j^i$ |
| $LL_j^i$ | The matching `LL` instruction of $C_j^i$ |
| $T_j^i$ | The `Read` on $Toggles[i]$ executed between $LL_j^i$ and $SC_j^i$ by the same thread that executes $LL_j^i$ and $SC_j^i$ |
| $\tilde{C}_j^i$ | The first configuration after $C_j^i$ such that a successful `SC` instruction is executed |
| $SC_l$ | The $l$th successful $SC$ instruction on $S$ in $\alpha$ |
| $LL_l$ | The matching `LL` of $SC_l$ |
| $C_l$ | The configuration just after $SC_l$ |
| $\alpha_l$ | The prefix of $\alpha$ which ends at $SC_l$ |
| $\alpha_0$ | The empty execution |

Table 5.5: Notation used in the proof of L-Sim.

For each $l > 0$, let $C_l$ be the configuration resulting after the execution of the $l$th `Add` instruction in $\alpha$.

**Lemma 5.17.** *For each $l \geq 0$, and for each $i \in \{1, \ldots, n\}$, if $p_i$ has executed $j_i^l \geq 0$ `Add` instructions by $C_l$, it holds that $Toggles[i] = j_i^l \mod 2$ at $C_l$.*

*Proof.* We prove the claim by a (straightforward) induction on $l$.

**Base case ($l = 0$).** Fix any $i \in \{1, \ldots, n\}$. By the way $Toggles$ is initialized, it follows that $Toggles[i] = 0$ at $C_0$. Since $p_i$ has not performed any request at $C_0$, it follows that $j_i^0 = 0$ at $C_0$, so that $j_i^0 \mod 2 = 0$ and the claim follows.

**Induction hypothesis.** Fix any $l > 0$ and assume that the claim holds for $C_{l-1}$.

**Induction step.** We prove that the claim holds for $C_l$. Assume that the $l$th `Add` is executed by some thread $p_i$ and let $j_i^l$ be the number of `Add` that has been executed by

$p_i$ until $C_l$. At $C_{l-1}$, $p_i$ has executed $j_i^{l-1} = j_i^l - 1$ Add. By the induction hypothesis, $Toggles[i] = j_i^{l-1} \mod 2 = (j_i^l - 1) \mod 2$ at $C_{l-1}$.

Assume first that $j_i^l \mod 2 = 1$. In such a case, it follows that $j_i^l - 1 \mod 2 = 0$. Induction hypothesis implies that $Toggles[i] = 0$ at $C_{l-1}$. By Observation 5.1, it follows that $toggle^i_{j_i^l-1} = 2^i$. During the $l$th Add instruction, $toggle^i_{j_i^l-1}$ is added to $Toggles$. Notice that $Toggles$ is updated only by executing Add (line 3). Thus, $Toggles$ remains unchanged between $C_{l-1}$ and the $l$th Add. It follows that $Toggles[i] = 0$ just before the execution of the $l$th Add. Thus, the only change that the $l$th Add causes on $Toggles$ is to set the $i$th bit to 1; all other bits remain unchanged.

Fix any $k \neq i$, $k \in \{1, \ldots, n\}$. Since the $l$th Add is executed by $p_i$, it follows that $j_k^l = j_k^{l-1}$. By the induction hypothesis, $Toggles[k] = j_k^{l-1} \mod 2 = j_k^l \mod 2$, as needed.

The case where $j_i^l \mod 2 = 0$ is symmetric. ∎

The following is an immediate consequence of Lemma 5.17.

**Corollary 5.1.** *For each $j$, $0 \leq j \leq m_i$, the following claims hold:*

1. *$Toggles[i] = j \mod 2$ at $Q_j^i$;*

2. *$Toggles[i]$ has the same value between $Q_{j-1}^i$ and $Q_j^i$.*

**Lemma 5.18.** *Consider any execution $\pi_j^i$, $j > 0$, of function* Attempt *by some thread $p_i$. $S.applied[i]$ is equal to $v = \lceil j/2 \rceil \mod 2$ just after the end of $\pi_j^i$.*

*Proof.* Assume, by the way of contradiction, that $S.applied[i] \neq v$ at the end of $\pi_j^i$. Since $S.applied[i]$ is a binary variable, it follows that $S.applied[i] = 1 - v$ at the end of $\pi_j^i$. By Lemma 5.16, there are at least two successful SC instructions in the execution interval of $\pi_j^i$. It follows that the last successful SC instruction executed in $\pi_j^i$ writes $1 - v$ into $S.applied[i]$. Let $SC_x$ be this SC instruction, let $LL_x$ be its matching LL instruction, let $p_x$ be the thread that executes $LL_x$ and $SC_x$, and let $T_x$ be the read instruction of line 12 executed by $p_x$ between $LL_x$ and $SC_x$. Lemma 5.16 implies that there are at least two successful SC instructions in the execution interval of $\pi_j^i$. Since $SC_x$ is a successful SC instruction, it follows that all $LL_x$, $T_x$ and $SC_x$ are executed in the execution interval of $\pi_j^i$. By the definition of $\pi_j^i$, it follows that $\pi_j^i$ is executed by the request $req^i_{\lceil j/2 \rceil}$. Corollary 5.1, implies that $Toggles[i] = \lceil j/2 \rceil \mod 2$ between $Q^i_{\lceil j/2 \rceil}$ and before the end

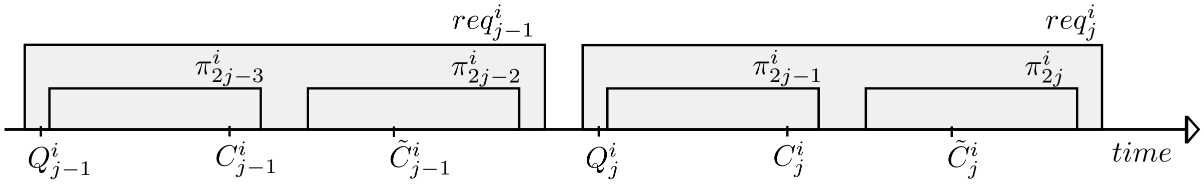


Figure 5.10: An example of an execution of L-Sim.

of $\pi^i_j$. Since $T_x$ is executed after $Q^i_{\lceil j/2 \rceil}$ and before the end of $\pi^i_j$, it follows that $T_x$ returns $v = \lceil j/2 \rceil \mod 2$ for the $i$th component. The pseudocode (lines 12 and 45) implies that $SC_x$ writes $v = \lceil j/2 \rceil \mod 2 \neq 1 - v$ at $S.applied[i]$, which is a contradiction. ∎

For the rest of the proof we introduce the following notation. Let $C_0$ be the initial configuration. At $C_0$, $S.applied[i]$ is equal to *false*. Lemma 5.18 implies that just after $\pi^i_1$, $S.applied[i]$ is equal to *true*. Let $C^i_1$ be the first configuration between $C_0$ and the end of $\pi^i_1$ at which $S.applied[i]$ is equal to *true*. Lemma 5.18 implies that just after the end of $\pi^i_3$, $S.applied[i]$ is equal to *false*. Let $C^i_2$ be the first configuration after $C^i_1$ such that $S.applied[i]$ is equal to *false*. Obviously, $C^i_2$ precedes the end of $\pi^i_3$. Consider any request $req^i_j$, $j > 1$. Lemma 5.18 implies that just after $\pi^i_{2j-2}$, $S.applied[i]$ is equal to $\lceil (j-2)/2 \rceil \mod 2 = (j-1) \mod 2$, while just after $\pi^i_{2j-1}$, $S.applied[i]$ is equal to $\lceil (2j-1)/2 \rceil \mod 2 = j \mod 2 \neq (j-1) \mod 2$. Let $C^i_j$ be the first configuration between the end of $\pi^i_{2j-2}$ and the end of $\pi^i_{2j-1}$ such that $S.applied[i]$ is equal to $j \mod 2$. Lemma 5.18 implies that just after $\pi^i_{2j-2}$, $S.applied[i]$ is equal to $(j-1) \mod 2$. Let $C^i_j$ be the first configuration after the end of $\pi^i_{2j-2}$ such that $S.applied[i]$ is equal to $j \mod 2$. Obviously, $C^i_j$ precedes the end of $\pi^i_{2j-1}$. Figure 5.10 illustrates the above notation.

Since the value of $S.applied[i]$ can change only by the execution of `SC` instructions on $S$, it follows that just before $C^i_{j-1}$ a successful `SC` on $S$ is executed. Let $SC^i_j$ be this `SC` instruction and let $LL^i_j$ be its matching `LL` instruction. Let $T^i_j$ be the read of $Toggles$ that is executed between $LL^i_j$ and $SC^i_j$ by the same thread.

**Lemma 5.19.** *Consider any $j$, $0 < j \leq m_i$, it holds that $T^i_j$ is executed after $Q^i_j$ and reads $j \mod 2$ in $Toggles[i]$.*

*Proof.* Assume, by the way of contradiction, that $T^i_j$ is executed before $Q^i_j$. Let $\pi_x$ be the `Attempt` that executes $T^i_j$.

Assume first that $j = 1$. Then, by its definition, $SC^i_1$ (which is executed by $\pi_x$ after $T^i_1$) writes to $S \rightarrow applied[i]$ a value equal to $\lceil j/2 \rceil \mod 2$; the code (lines 12, 16) implies that, in this case, $T_1^i$ reads 1 in $Toggles[i]$. Corollary 5.1 implies that $Toggles[i] = 0$ between $C_0$ and $Q_1^i$. Thus, $T_1^i$ could not read 1 in $Toggles[i]$, which is a contradiction.

Assume now that $j > 1$. By our assumption that $T_j^i$ is executed before $Q_j^i$, it follows that $LL_j^i$, which is executed before $T_j^i$, precedes $Q_j^i$. In case that $T_j^i$ follows $Q_{j-1}^i$, Corollary 5.1 implies that $T_j^i$ reads $(j-1) \mod 2 \neq j \mod 2$ in $Toggles[i]$. By the pseudocode (lines 12, 16 and 45), it follows that $\pi_x$ writes the value $(j-1) \mod 2$ into $S.applied[i]$. By its definition, $SC_j^i$ stores $j \mod 2$ into $S.applied[i]$, which is a contradiction. Thus, $T_j^i$ is executed before $Q_{j-1}^i$. By its definition, $\pi_{2j-3}^i$ starts its execution after $Q_{j-1}^i$ and finishes its execution before $C_j^i$. Lemma 5.16 implies that at least two successful `SC` instructions are executed in the execution interval of $\pi_{2j-3}^i$. Recall that $LL_j^i$ precedes $T_j^i$ and therefore also the beginning of $\pi_{2j-3}^i$, while by definition $SC_j^i$ follows the end of $\pi_{2j-3}^i$. It follows that $SC_j^i$ is not a successful `SC` instruction, which is a contraction. ∎

We next prove that the value of $S.applied[i]$ remains the same between $SC_{j-1}^i$ and $SC_j^i$.

**Lemma 5.20.** *Consider any $j$, $0 < j \leq m_i$. At each configuration $C$ between $C_{j-1}^i$ and $C_j^i$, it holds that $S.applied[i] = (j-1) \mod 2$.*

*Proof.* Assume, by the way of contradiction, that there is at least one configuration between $C_{j-1}^i$ and $C_j^i$ such that $S \rightarrow applied[i]$ is equal to some value $v_x \neq (j-1) \mod 2$. Let $C_x$ be the first of these configurations. Since only `SC` instructions of line 45 write on base object $S$, it follows that there is a successful `SC` instruction, let it be $SC_x$, executed just before $C_x$ that stores $v_x$ at $S.applied[i]$. Let $\pi_x$ be the `Attempt` that executes $SC_x$ and let $T_x$ be the read instruction that $\pi_x$ executes on line 12 of the pseudocode. By the definition of $C_{j-1}^i$ and $Q_{j-1}^i$, it is implied that $C_{j-1}^i$ follows $Q_{j-1}^i$ and precedes $Q_j^i$. Corollary 5.1 implies that $Toggles[i] = (j-1) \mod 2 \neq v_x$ in any configuration between $Q_{j-1}^i$ and $Q_j^i$. Since $SC_x$ writes $v_x$ into $S.applied[i]$, the pseudocode (lines 12 and 45) imply that $T_x$ precedes $Q_{j-1}^i$. It follows that $LL_x$ precedes $Q_{j-1}$, since $LL_x$ precedes $T_x$. Therefore $LL_x$ precedes $C_{j-1}$. This implies that there is a successful `SC` instruction, which is $SC_{j-1}^i$, between $LL_x$ and $SC_x$. Thus, $SC_x$ is a failed `SC` instruction, which is a contradiction. ∎

By Lemma 5.20 and the pseudocode (line 15), it follows that $S.papplied[i] = 1 - (j \mod 2)$ at $C_j^i$. Denote by $\tilde{C}_j^i$ be the first configuration after $C_j^i$ such that a successful `SC` instruction is executed.

**Lemma 5.21.** *$\tilde{C}^i_{j-1}$ precedes $C^i_j$ and follows $C^i_{j-1}$.*

*Proof.* By the definition of $\tilde{C}^i_{j-1}$, it is implied that $\tilde{C}^i_{j-1}$ follows $C^i_{j-1}$. Lemma 5.19 implies that $C^i_j$ follows $Q^i_j$. By its definition, $Q^i_j$ follows $\pi_{2j-2}$. By Lemma 5.18, it follows that $C^i_{j-1}$ precedes the end of $\pi_{2j-3}$. Thus, $\pi^i_{2j-2}$ begins its execution after $C^i_{j-1}$ and ends its execution before $C^i_j$. By Lemma 5.16, there are at least two successful SC instructions in the execution interval of $\pi^i_{2j-2}$. Since, $\tilde{C}^i_{j-1}$ is the first successful SC just after $C^i_{j-1}$, it follows that $\tilde{C}^i_{j-1}$ precedes the end of $\pi^i_{2j-2}$. Therefore, $\tilde{C}^i_{j-1}$ precedes $C^i_j$. ∎

**Lemma 5.22.** *$S.papplied[i] = S.applied[i]$ in any configuration between $\tilde{C}^i_{j-1}$ and $C^i_j$ ($C^i_j$ is not included).*

*Proof.* By Lemma 5.20, it follows that $S.applied[i] = (j-1) \mod 2$ between $C^i_{j-1}$ and $C^i_j$. Assume by the way of contradiction that there at least one configuration between $\tilde{C}^i_{j-1}$ and $C^i_j$ such that $S.papplied[i] \neq (j-1) \mod 2$ and let $C_x$ be the first of them. Let $SC_x$ be the SC instruction executed just before $C_x$ and let $LL_x$ be its matching LL instruction. Since, $SC_x$ is a successful SC instruction, $LL_x$ follows $\tilde{C}^i_{j-1}$. By Lemma 5.20, it follows that $S.applied[i] = (j-1) \mod 2$ between $C^i_{j-1}$ and $C^i_j$. Thus, $LL_x$ reads $(j-1) \mod 2)$ in $S.applied[i]$. The pseudocode (lines 11 and 15) implies that the SC instruction at $\tilde{C}^i_j$ stores a value equal to $(j-1) \mod 2$ into $S.papplied[i]$, which is a contradiction. ∎

By Lemma 5.22, and by the pseudocode (line 15), the following observation is derived.

**Observation 5.2.** *$S.papplied[i] = 1 - S.applied[i]$ at $C^i_j$.*

The following observation is an immediate derivation of the definition of $\tilde{C}^i_j$ and Observation 5.2.

**Observation 5.3.** *$S.papplied[i] = 1 - S.applied[i]$ in any configuration between $C^i_j$ and $\tilde{C}^i_j$, $\tilde{C}^i_j$ is not included.*

We say that an request *req* by some thread $p_i$ *is applied* on the simulated object if (1) the Read instruction on $Toggles$ (line 12), executed by some request *req′* (that might be *req* or any other request), includes $p_i$ in the set of threads it returns, (2) procedure Attempt, executed by *req′* reads in $Announce[i]$, the request type written there by $p_i$ for *req* and considers it as the new request type for $p_i$, (3) Attempt by *req′* calls apply for *req* (lines 19 - 37), and the execution of the `SC` at line 45 (let it be $SC_r$) on $S$ succeeds. When these conditions are satisfied, we sometimes also say that *req'* applies *req* on the simulated object or that $SC_r$ applies *req* on the simulated object. We next prove that each request *req* is applied on the simulated object exactly once.

**Lemma 5.23.** *Request* $req_j^i$ *is applied to the simulated object at configuration* $C_{3j-1}^i$.

*Proof.* Let $p_h$ be the `Attempt` that executes the successful `SC` instruction (let it be $SC_h$ this `SC` instruction) just before $\tilde{C}_j^i$. Let $LL_h$ be the matching `LL` of $SC_h$. Since, $SC_h$ is a successful `SC` instruction, it is implied that $LL_h$ follows $C_j^i$. Observation 5.3 implies that $LL_h$ reads for *S.applied*$[i]$ a value different from that stored in *S.papplied*$[i]$. Therefore, the if statement of line 18 returns *true*. Thus, a request for thread $p_i$ is applied at $\tilde{C}_j^i$. Let *req'* be this request and assume, by the way of contradiction, that $req' \neq req_j^i$. Lemma 5.19 implies that $\pi_h$ executes its read $T_h$ on $Toggles$ after $Q_j^i$. By the pseudocode (lines 12, 19), $\pi_h$ reads *Announce*$[i]$ after $T_h$, thus the reading of *Announce*$[i]$ by $\pi_h$ is executed between $Q_j^i$ and $\tilde{C}_j^i$. Since $req_j^i$ writes its request to *Announce*$[i]$ before $Q_j^i$, the reading of *Announce*$[i]$ by $\pi_h$ returns $req_j^i$. Thus, $\pi_h$ applies $req_j^i$ as the request of $p_i$ in the simulated object. ∎

The following corollary is an immediate consequence of Lemma 5.22, Observation 5.3, Lemma 5.23, and of the definition of $\tilde{C}_j^i$.

**Corollary 5.2.** *Each request req is applied exactly once.*

We are now ready to assign linearization points. Let $a$ be any execution. For each $i \in \{1, ..., n\}$ and $j \geq 1$, we place the linearization point of $req_j^i$ at $\tilde{C}_j^i$; ties are broken by the order imposed by identifiers of threads.

**Lemma 5.24.** *Each request* $req_j^i$ *is linearized within its execution interval.*

*Proof.* Lemma 5.19 implies that $Q_j^i$ precedes $C_j^i$. Thus $Q_j^i$ precedes $\tilde{C}_j^i$. Since $\tilde{C}_j^i$ is the first sucessfull `SC` on $S$ after $C_j^i$ and (by its definition and by Lemma 5.18) $C_j^i$ precedes the end of $\pi_{2j-1}^i$, $\tilde{C}_j^i$ precedes the end of $\pi_{2j}^i$. Thus, $\tilde{C}_j^i$ is in the execution interval of $req_j^i$. Thus, $req_j^i$ is executed in its execution interval. ∎

In order to prove consistency, we introduce the following notation. Denote by $SC_l$ the $l$-th successful $SC$ instruction on base object $S$. Obviously, between $SC_{l-1}$ and $SC_l$,

$l > 1$ base object $S$ is not modified. Let $it_i$ be some iteration of for loop of line 9 executed by some thread $p_i$. Let $SV_r(it_i)$ be the sequence of base objects read by the LL instructions of line 32 by $it_i$. Denote by $|SV_r(it_i)|$ the number of elements of $SV_r(it_i)$. For each $1 \leq j \leq |SV_r(it_i)|$, denote by $SV_r^j(it_i)$ the prefix of $SV_r(it_i)$ containing the $j$ first elements of $SV_r(it_i)$, i.e. $SV_r^j(it_i) = \langle sv_r^1(it_i), \ldots, sv_r^j(it_i)\rangle$, where $sv_r^j(it_i)$ is the $j$th LL instruction performed by $it_i$ on some base object $r$. Let $SV_r^0(it_i) = \lambda$ be the empty sequence. Let $V_r(it_i)$ be the sequence of insertions in directory $D$ (lines 33-34) by $it_i$. Denote by $|V_r(it_i)|$ the number of elements of $V_r(it_i)$. Obviously, it holds that $|SV_r(it_i)| = |V_r(it_i)|$. For each $1 \leq j \leq |V_r(it_i)|$, denote by $v_r^i(it_i)$ the prefix of $V_r(it_i)$ containing $j$ first elements of $V_r(it_i)$, i.e. $V_r^j(it_i) = \langle v_r^1(it_i), \ldots, v_r^j(it_i)\rangle$, where $v_j(it_i)$ is the $j$th value inserted to directory $D$. Let $V_r^0(it_i) = \lambda$ be the empty sequence. Let $SV_w(it_i)$ be the sequence of shared base objects accessed by $it_i$ while executing lines 41-42. Denote by $|SV_w(it_i)|$ the number of elements of $SV_w(it_i)$. For each $1 \leq j \leq |SV_w(it_i)|$, denote by $SV_w^j(it_i)$ the prefix of $SV_w(it_i)$ that contains the $j$ last elements of $SV_w(it_i)$, i.e. $SV_w^j(it_i) = \langle svw_1(it_i), \ldots, svw_j(it_i)\rangle$, where $svw_j(it_i)$ is the $j$th operation (lines 41-42) by $it_i$. Let $SV_w^0(it_i) = \lambda$ be the empty sequence. Let $SV_a(it_i)$ be the sequence of shared base objects allocations during $it_i$ iteration (lines 20-26). Denote by $|SV_a(it_i)|$ the number of elements of $SV_a(it_i)$. For each $1 \leq j \leq |SV_a(it_i)|$, denote by $SV_a^j(it_i)$ the prefix of $SV_a(it_i)$ that contains the $j$ first elements of $SV_a(it_i)$, i.e. $SV_a^j(it_i) = \langle sva_1(it_i), \ldots, sva_j(it_i)\rangle$, where $sva_j(it_i)$ is the $j$th base object allocation by $it_i$.

Let $SV_{arw}(it_i)$ be the sequence of allocations/reads/writes that $it_i$ performs nf base objects in lines 20-43 of the pseudocode. Denote by $|SV_{arw}(it_i)|$ the number of elements of $SV_{arw}(it_i)$. Obviously, it holds that $|SV_{arw}(it_i)| = |SV_a(it_i)| + |SV_r(it_i)| + |SV_w(it_i)|$. For each $1 \leq j \leq |SV_{arw}(it_i)|$, denote by $SV_{arw}^j(it_i)$ the prefix of $SV_{arw}(it_i)$ that contains the $j$ first elements of sequence $SV_{arw}(it_i)$ (i.e. $SV_{arw}^j(it_i) = \langle svarw_1(it_i), \ldots, svarw_j(it_i)\rangle$) where $svarw_j(it_i)$ is the $j$th base object allocations/reads/writes of base objects performed by $it_i$.

**Lemma 5.25.** *Let $l > 0$ be any integer such that S.applied$[i] \neq$ S.papplied$[i]$ at configuration $C_{l-1}$. Let $req_j^i$ be the value of Announce$[i]$ at $C_{l-1}$. In any configuration between $C_{l-1}$ and $C_l$, it holds that Announce$[i] = req_j^i$.*

*Proof.* Assume, by the way of contradiction, that there is at least one configuration between $C_{l-1}$ and $C_l$, such that $Announce[i] = req^i_{j'} \neq req^i_j$. Let $C$ be the first of these configurations. The pseudocode (line 1) implies that $p_i$ is the only thread that modifies base object $Announce[i]$. Thus, $p_i$ starts the execution of a new request $req^i_{j'}$ at $C$, and it holds that $j' = j + 1$. Since the write on $Announce[i]$ by $p_i$ is executed between $C_{l-1}$ and $C_l$, it is implied that either $C^i_{j+1} = C_l$ or $C^i_{j+1}$ follows $C_l$. Since the end of $req^i_j$ precedes $C$, it follows that either $\tilde{C}^i_j = C_{l-1}$ or $\tilde{C}^i_j$ precedes $C_{l-1}$. Lemma 5.22 implies that $S.applied[i] = S.papplied$ in any configuration between $\tilde{C}^i_j$ and $C^i_{j+1}$ ($C^i_{j+1}$ is excluded). Thus, it holds that $S.applied[i] = S.papplied$ in any configuration between $C_{l-1}$ and $C_l$, which is contracts our claim that $S.applied[i] \neq S.papplied[i]$ at $C_{l-1}$. ∎

By the pseudocode (lines 9, 38 and 45) the following observation holds.

**Observation 5.4.** *For each $j > 0$, it holds that $S.seq = j - 1$ in every configuration between $C_{j-1}$ and $C_j$.*

By lines 32, 39-42 and 45 of the pseudocode the following observation is derived.

**Observation 5.5.** *Let $it_i$ be an iteration executed by $p_i$ such that the execution of the* SC *instruction $SC_j$ on line 45 is successful. Let $r$ be the base object accessed by $svw_j(it_i)$, $1 \leq j \leq |SV_w(it_i)|$. There is at least one successful* SC *on $r$ instruction between $SC_{j-1}$ and $SC_j$.*

**Lemma 5.26.** *Let $r$ be any shared base object other than $S$. For any $l > 0$, the following claims are true:*

1. *At most one successful* SC *instruction is executed on $r$ between $C_{l-1}$ and $C_l$.*

2. *In case that a successful* SC *instruction $SC_w$ is executed on $r$, it holds that $r.seq < l$ just before $SC_w$ and $r.seq = l$ just after $SC_w$.*

3. *Let $it_i$ be some iteration of the loop of line 9 executed by a thread $p_i$ that executes at least one successful* SC *instruction $SC_r$ on $r$. If $LL_r$ is the* LL *instruction of line 11 executed by $it_i$, then $LL_r$ is executed after $C_{l-1}$.*

4. *Let $it_i$, $it_{i'}$ be two iterations of the for loop of line 9 executed by threads $p_i$ and $p_{i'}$ respectively, such that that both $it_i$, $it_{i'}$ execute their* LL *instructions of line 11*

*somewhere between $C_{l-1}$ and $C_l$, $l > 0$, and $|SV_{arw}(it_i)| \geq |SV_{arw}(it_{i'})|$. If both $it_i$, $it_{i'}$ execute line 39, just before $C_l$ it holds that $SV_{arw}(it_i) = SV_{arw}(it_{i'})$.*

*Proof.* We prove the claims by induction on $l$.

**Induction hypothesis.** Fix any $l \geq 1$ and assume that the claims hold for $l$

**Induction step.** We prove that the claims hold for $l + 1$. We first prove Claim 1. Let $SC'$ be the first of the successful `SC` instruction on $r$ between $C_{l-1}$ and $C_l$. We prove that $r.seq = l$ just after the execution of $SC'$. Assume by the way of contradiction that $r.seq = l' \neq j$. Let $it_h$ be the iteration of line 11 executed by some thread $p_h$ that executes $SC'$. Let $LL'$ be the matching `LL` instruction of $SC'$. Since $it_i$ executes successfully line 42 of the pseudocode, the pseudocode (lines 38 and 42) implies that the `VL` instruction of line 38 returns *true*. Since $LL'$ is executed by $it_i$ before this `VL` instruction, it follows that $LL'$ precedes $SC_{j'}$. Thus, the `VL` instruction of line 38 is executed before $SC_{j'}$. Let $it_{i'}$ be the iteration of the loop of line 11 at which $SC_{j'}$ is executed and let $p_{i'}$ be the thread that executes $SC_{j'}$. Obviously, $LL_{j'}$ has been executed between $C_{l'-1}$ and $C_{l'}$. Since $LL'$ is also executed between $C_{l'-1}$ and $C_{l'}$, the induction hypothesis (Claim 2.ii) implies that $SV_w(it_h) = SV_w(it_q)$. Thus, $it_q$ has also executed an `SC` instruction on $r$. By Observation 5.5, there is a successful `SC` instruction on $r$ between $SC_{l'-1}$ and $SC_{l'}$. Let $SC_r$ be this instruction. By induction hypothesis (claim 1), it follows that $r.seq = j'$ just after the execution of $SC_r$. Since $SC'$ is a successful `SC` instruction, $LL'$ follows $SC_r$. By the pseudocode (lines 41-42), it follows that $SC'$ is not executed, which is a contradiction. Therefore $r.seq = j$ just after the execution of $SC_r$. We now prove that there is no other successful `SC` instruction between $SC'$ and $C_l$ on $r$. Assume by the way of contradiction that at least one successful `SC` instruction takes place between $SC'$ and $C_l$. Let $SC''$ be the first of these instructions. Since, $SC''$ is a successful `SC` instruction, it follows that its matching `LL` instruction $LL''$ follows $SC'$. By the pseudocode (lines 41-42), it follows that $SC''$ is not executed since $r.seq = S.seq$, which is a contradiction.

Claim 2 is proved by following similar arguments to those of Claim 1.

We now prove Claim 3. Assume by the way of contradiction that $LL_p$ is executed between $SC_{j'-1}$ and $SC_{j'}$, $j' < j$. Let $p_i$ be the thread that executes $SC_{j'}$ on some iteration $it_i$. By Claim 1 and by Claim 2, it follows that $r.seq \leq j'$ just before $SC_{j'}$. Thus $SC_r$ is not executed, which is a contradiction. Thus, Claim 3 holds.

We now prove Claim 4. It is enough to prove that $svarw_{l'}(it_i) = svarw_{l'}(it_{i'})$, for any $l' \leq |SV_{arw}(it_i)|$. we prove this claim by induction on the number $l' \leq |SV_{arw}|$ of elements of $SV_{arw}(it_i)$. *Induction hypothesis.* Fix any $l' \geq 1$ and assume that the claims hold for $l' - 1$.

*Induction step.* We prove that the claim holds for $l'$. Distinguish the following cases.

1. In case that $svarw_{l'}(it_i) = sva_j(it_i), j \leq |SV_a(it_i)|$, the induction hypothesis, the fact that both $it_i$ and $it_{i'}$ simulate the same deterministic object and the pseudocode (lines 20-27) imply this.

2. In case that $svarw_{l'}(it_i) = svw_j(it_i), j \leq |SV_w(it_i)|$, the claim is an immediate implication of the induction hypothesis and the fact that both $it_i$ and $it_{i'}$ simulate the same deterministic object.

3. In case that $svarw_{l'}(it_i) = svr_j(it_i), j \leq |SV_r(it_i)|$, the induction hypothesis and the fact that both $it_i$ and $it_{i'}$ simulate the same deterministic object imply that $svarw_{l'}(it_i) = svr_j(it_i) = svr_j(it_{i'}) = svarw_{l'}(it_{i'})$. Claims 1 and 2 and the pseudocode (lines 39-43) imply that $V_r^{l'}(it_i) = V_r^{l'}(it_{i'})$. Thus, the claim holds. ∎

Let $\alpha$ be any execution of the algorithm. Denote by $\alpha_i$, the prefix of $\alpha$ which ends at $SC_i$ and let $C_i$ be the first configuration following $SC_i$. Let $\alpha_0$ be the empty execution. Denote by $l_i$ the linearization order of the requests in $\alpha_i$.

**Lemma 5.27.** *For each $i \geq 0$, (1) object's state is consistent at $C_i$, and (2) $\alpha_i$ is consistent.*

*Proof.* We prove the claim by induction on $i$.

**Base case (i=0):** The claim holds trivially; we remark that $\alpha_i$ is empty in this case.

**Induction hypothesis:** Fix any $i > 0$ and assume that the claim holds for $i - 1$.

**Induction step:** We prove that the claim holds for $i$. By the induction hypothesis, it holds that: (1) object's state is consistent at $C_{i-1}$, and (2) $\alpha_{i-1}$ is consistent with linearization $l_{i-1}$. Let *req* be the request that executes $SC_i$. If *req* applies no request on the simulated object, the claim holds by induction hypothesis. Thus, assume that *req* applies $j > 0$ requests on the simulated object. Denote by $req_1, ..., req_j$ the sequence of these requests ordered with respect to the identifiers of the threads that initiate them.

Notice that *req* performs $LL_i$ after $C_{i-1}$ since otherwise $SC_i$ would not be successful. By the induction hypothesis, object's is consistent at $C_{i-1}$. Lemma 5.26 and Corollary 5.2 imply that all threads that are trying to apply a set of requests between $C_{i-1}$ and $C_i$ do the following (1) apply the same set of requests with the same order, (2) all read the same consistent state of the object, (3) write the same set of base objects with the same values (although only one write succeeds), and (4) none of $req_1, \ldots, req_j$ have been applied in the past.

Given that $req_1, ..., req_j$ are executed by *req* sequentially, the one after the other in the order mentioned above, it is a straightforward induction to prove that (1) for each $f$, $0 \leq f \leq j$, request $req_f$ returns a consistent response; moreover, $S \rightarrow st$ is consistent and once line 14 has been executed by *req* for all these requests. Therefore, $S \rightarrow st$ is consistent after the execution of *req*'s successful SC. This concludes the proof of the claim. ∎

**Theorem 5.6.** L-Sim *is a linearizable, wait-free implementation of a universal object. The number of shared memory accesses performed by* L-Sim *is* $O(kW)$.

## 5.5 SimStack: A wait-free implementation of a shared stack

In this section, we present a wait-free implementation of a stack based on P-Sim, which is called SimStack (Algorithm 13). Performance evaluation of SimStack, is also provided.

### 5.5.1 Algorithm description

In SimStack, the stack is implemented as a linked list of nodes; a pointer *top* points to the topmost element of this list. P-Sim is employed to atomically manipulate *top*. Thus, the state *st* of the simulated object stores just this pointer and not the entire stack state. This is accomplished by defining `State` to be a pointer to a struct of type node (line 2).

When a thread initiates a Push or a Pop request, it allocates a struct of type `Request`, initiates it appropriately, and simply calls P-Sim with this struct as a parameter. The pseudo-code for the sequential operations, push and pop, is also presented in Algorithm 13.

**Theorem 5.7.** SimStack *is a linearizable wait-free implementation of a concurrent stack.*

### 5.5.2 Performance Evaluation

We compare the experimental performance of SimStack with that of state-of-the-art concurrent stack implementations, like the lock free stack implementation presented by Treiber in [58], the elimination back-off stack [35], a stack implementation based on CLH spin lock [23, 47], and a linked stack implementation based on flat-combining [33, 34]. We remark that the implementation based on flat-combining uses the same pseudo-code for Push and Pop, i.e. that presented in Algorithm 13.

Our experiment is of the same nature as that performed by Michael and Scott for queues in [50]. More specifically, $10^7$ pairs of a Push and a Pop, in total, were executed

```
typedef struct {
    Data data;
    Node *next;
} Node;

typedef struct {
    Node *top;
} State;

void Push(ArgVal arg, ThreadId i){                 // Code for Push
    PSimApplyOp(<push, arg>, i);
}

Node *Pop(ThreadId i){                             // Code for Pop
    Node *ret;

    ret = PSimApplyOp(<pop, $\perp$>, i);
    return ret;
}

void push(State *pst, ArgVal arg){
    nd = allocate a new node;                      // Allocate a new node
    nd→data = arg;                                 // Write node's information
    nd→next = pst→top;                             // top→next points to the top of stack
    pst→top = nd;
}

Node *pop(State *pst){
    Node *ret;

    ret = pst→top;                                 // Compute the return value for thread $p_i$
    if (pst→top $\neq \perp$)
        pst→top=pst→top→next;                      // Pop a node from the list
    return ret;
}
```

Algorithm 13: Implementation of Pop and Push for SimStack.

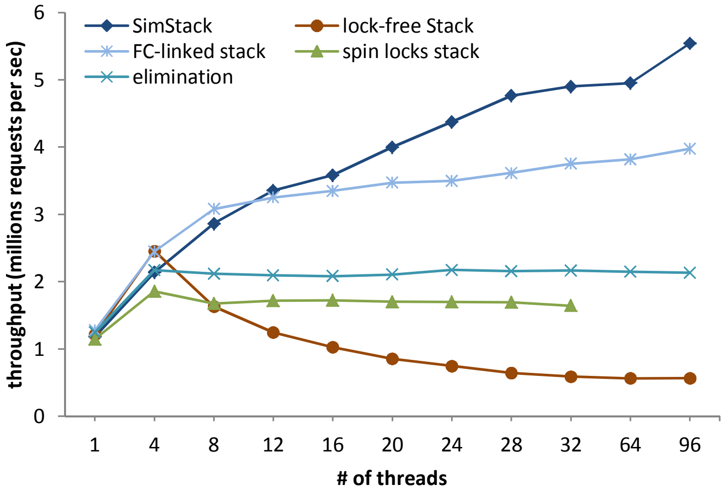


Figure 5.11: Performance of SimStack.

as the number of threads $n$ increases. The average throughput was measured; the results are illustrated in Figure 5.11. Again, for each value of $n$, the experiments have been performed 10 times and averages have been taken. As in previous experiments for P-Sim, we have simulated a random workload by executing a random number of (at most 512) iterations of a dummy loop after the completion of each request. To reduce the overheads for the memory allocation of the stack nodes, we use the Hoard memory allocator [18] to allocate an entire pool of nodes (instead of allocating one node each time); when all the elements of a pool have been used, we ask for the allocation of a new pool of nodes.

As shown in Figure 5.11, all algorithms scale well up to 4 threads but SimStack outperforms all other implementations for $n > 12$. More specifically, SimStack is up to 2.94 times faster than the lock-free stack, up to 2.58 times faster than the spin-lock based stack, up to 2.57 times faster than the elimination back-off stack, and up to 1.35 times faster than flat-combining.

Similarly to the experiment of the `Fetch&Multiply` object (Figure 5.2), CLH spin locks achieve almost constant throughput for different values of $n$. The lock-free implementation suffers from increased contention, so its performance degrades as $n$ increases. As expected, the elimination backoff stack achieves much better performance than the lock-free implementation and the spin-lock based implementations in most cases. SimStack and flat-combining significantly outperform the other stack implementations. For small numbers of $n$, flat-combining performs a little better than SimStack, but for $n > 12$, SimStack exhibits better performance than flat-combining (for similar reasons to those presented in Section 5.3).

## 5.6 SimQueue: A wait-free implementation of a shared queue

In this section, we present a wait-free implementation of a queue based on P-Sim, which is called SimQueue (Algorithms 14 and 15). Performance evaluation of SimQueue, is also provided.

### 5.6.1 Algorithm description

Designing a queue implementation using P-Sim is not as simple as implementing a stack, basically because an efficient such implementation should allow the enqueuers and the dequeuers to run independently. To achieve this, we employ two instances of a slightly modified version of P-Sim (Algorithms 14 -16), one for the enqueuers, called Enq-PSim, and one for the dequeuers, called Deq-PSim. The queue is linked using a *next* pointer field in each node. We denote by *EnqS* and *DeqS* the variable $S$ in each of the instances of P-Sim for the enqueuers and the dequeuers, respectively (lines 3-4). Pointer $DeqS \rightarrow st.head$ points to the first element of the simulated queue. The queue initially contains a dummy node; the dummy node is always the first node of the queue. This allows the dequeuers to work independently of the enqueuers. Specifically, the dequeuers manipulate the head of the queue which is never updated by any enqueuer. Thus, *DeqState* is a struct which contains just a pointer to the head of the queue.

Whenever a thread $p$ performs an enqueue request, it helps only other active enqueuers (ignoring currently active dequeuers). Thread $p$ creates a local queue of nodes, one for each enqueuer it helps. If the SC of line 35 by $p$ is successful, a pointer to the first element of $p$'s local queue and a pointer to its last element are stored in $EnqS \rightarrow st$. These pointers are $EnqS \rightarrow st.first$ and $EnqS \rightarrow st.last$. Moreover, the previous value of $EnqS \rightarrow st.last$ is stored in $EnqS \rightarrow st.tail$. Thus, *EnqState* is a struct containing these three pointers.

We remark that at configuration $C$ resulting from the execution of the successful SC by $p$, the simulated queue is supposed to contain not only the elements of the list

```
typedef struct {
    Data data;
    Node *next;
} Node;

typedef struct {                 // Implementation of State for enqueuers
    Node *tail;                  // pointer to previous value of queue's tail
    Node *first;                 // pointer to the first element of the newly enqueued nodes
    Node *last;                  // pointer to the last element of the newly enqueued nodes
} EnqState;

typedef struct {                 // Implementation of State for dequeuers
    Node *head;                  // pointer to the head of the queue
} DeqState;

// Initially, EnqS points to a struct with value $\langle\langle nd_0, \perp, \perp\rangle, 0, \langle\perp, \ldots, \perp\rangle\rangle$,
// where $nd_0$ is the dummy node that is initially placed in the queue
shared StRec *EnqS;
// Initially, DeqS points to a struct with value $\langle\langle nd_0, \perp, \perp\rangle, 0, \langle\perp, \ldots, \perp\rangle\rangle$
shared StRec *DeqS;

void ENQUEUE(ArgVal arg, ThreadId i){             // Code for ENQUEUE
    Enq-PSim(<enqueue, arg>, i);                  // Call an instance of P-Sim for enqueuers
}

Node *DEQUEUE(ThreadId i){                        // Code for DEQUEUE
    Node *ret = Deq-PSim(<dequeue, $\perp$ >, i); // Call an instance of P-Sim for dequeuers
    return ret;
}

void enqueue(EnqState *pst, ArgVal arg) {
    Node *new_node;

    new_node = allocate a new struct Node;
    new_node→data = arg;
    new_node→next = $\perp$;
    if (pst→first == $\perp$) pst→first = new_node;
    else pst→last→next = new_node;
    pst→last = new_node;
}

Node *dequeue(DeqState *pst) {
    Node *ret = $\perp$;

    ret = pst→head→next;
    if (ret != $\perp$)
        pst→head = ret;
    return ret;
}
```

Algorithm 14: Data structures for SimQueue, the implementation of ENQUEUE and DEQUEUE in SimQueue, and the implementations (enqueue and dequeue) of the sequential versions of enqueue and dequeue.

```
   void Attempt(ThreadId i) {                // Code for Attempt
      boolean ltoggles[1..n];                // $ltoggles$ is implemented as an integer
      StRec *ls_ptr;
      boolean dFlag;
18    for j=1 to 2 do{
19       ls_ptr = LL(S);                     // read the pointer stored in S
20       Pool[i][$index_i$] = *ls_ptr;         // Create a copy of current state
21       if (VL(S) == 0)
22          continue;
23       ltoggles = Toggles;                 // Read the vector of toggles
24       if (Announce[i].func==enqueue) EnqLinkQueue(&Pool[i][$index_i$].st);
25       dFlag = $true$;
26       for l=1 to n do {
            // If $p_i$ has a request that is not applied yet
27          if(ltoggles[l] $\neq$ Pool[i][$index_i$].applied[l]) {
28             if (Announce[i].func== dequeue) {
29                if (dFlag == $true$) dFlag = DeqLinkQueue(Pool[i][$index_i$].st);
30                if (dFlag == $true$) {
                     // Apply the request and compute return value
31                   apply Announce[l] on Pool[i][$index_i$].st
                     and store the return value into Pool[i][$index_i$].rvals[l];
32                } else store $\perp$ to Pool[i][$index_i$].rvals[l];
               } else {
                  // Apply the request and compute return value
33                apply Announce[l] on Pool[i][$index_i$].st
                  and store the return value into Pool[i][$index_i$].rvals[l];
               }
34             Pool[i][$index_i$].applied[l] = ltoggles[l];
            }
         }
35       if(SC(S, &Pool[i][$index_i$]))          // Try to change the contents of S
36          $index_i$=($index_i + 1$)  mod 2;      // If success, $p_i$ uses the next struct
37       BackoffCalculate();
      }
   }
```

Algorithm 15: Pseudocode for the Attempt in SimQueue.

addressed by $DeqS \rightarrow st.head$, but also the elements that are stored in $p$'s local queue (addressed by $EnqS \rightarrow st.first$), in the order they are met in these two lists with the first element being that pointed to by $DeqS \rightarrow st.head$. This is so, despite the fact that at $C$, $EnqS \rightarrow st.tail \rightarrow next$ does not point to the node pointed to by $EnqS \rightarrow st.start$. However, an update on $EnqS \rightarrow st.tail \rightarrow next$ to point to this node must occur before the application of the next set of simulated requests. To achieve this, the enqueuers call EnqLinkQueue, where they try to update (with the CAS of line 39) the next field of the node pointed to by $EnqS \rightarrow st.tail$ to point to $EnqS \rightarrow st.start$ (connecting in this way the two parts that store the state of the simulated queue).

```
    void EnqLinkQueue(State *pst) {
38     if (pst→first $\neq \perp$) {
39        CAS(pst→tail→next, $\perp$, pst→first);
40        pst→tail = pst→last;
41        pst→first = $\perp$;
42        pst→last = $\perp$;
       }
    }

    boolean DeqLinkQueue(State *pst) {
       StRec *tmpS;
       Node *first, *tail;
43     if (pst → head → next == $\perp$) {
44        tmpS = LL(EnqS); // EnqS is the variable S of P-Sim's instance for enqueuers
45        tail = tmpS→st.tail;
46        first = tmpS→st.first;
47        if (VL(EnqS))
48           if (first $\neq \perp$) CAS(tail→next, $\perp$, first);
       }
49     if (pst → head → next == $\perp$) return *false*;
50     else return *true*;
    }
```

Algorithm 16: Pseudocode for EnqLinkQueue and DeqLinkQueue in SimQueue.

A dequeue helps only active dequeuers. To ensure consistency, each dequeue request also executes a CAS (line 48 of DeqLinkQueue) to link the two parts of the simulated queue in a way similar to what enqueue requests do (line 39). The pseudocode for SimQueue appears in Algorithms 14-16. Notice that Enqueue simply calls Enq-PSim with parameters a pointer to enqueue, which is a function containing the enqueue sequential implementation, its argument, and the thread id. Similarly, Dequeue calls Deq-PSim with parameters a pointer to dequeue, which is a function containing the dequeue sequential implementation, and the thread id.

### 5.6.2 Correctness proof

Let $\alpha$ be any execution of SimQueue. Denote by $\texttt{SC}_m$, $m \geq 0$, the $m$th successful SC instruction on $EnqS$ executed in $\alpha$, denote by $\texttt{LL}_m$ its matching LL, let $p_{i_m}$ be the thread that executes $\texttt{SC}_m$, and let $ls_{i_m}$ be the element of $Pool$ used by $p_{i_m}$ during the instance of Attempt that executes $\texttt{SC}_m$. Denote by $C_m$ the configuration that results from the execution of $\texttt{SC}_m$. Let $tail_m$, $first_m$, and $last_m$ be the values of $EnqS \rightarrow st.tail$, $EnqS \rightarrow st.first$, and $EnqS \rightarrow st.last$, respectively, at $C_m$. Let $tail_0$ be a pointer to the dummy node $nd_0$ that is initially placed in the queue, and let $first_0$ and $last_0$ be $\perp$. Obviously, between $\texttt{SC}_m$ and $\texttt{SC}_{m+1}$, $EnqS$ is not modified. Denote by $\texttt{CAS}_m$ the $m$th successful `CAS` executed in $\alpha$.

We remark that the proof of P-Sim up to Lemma 5.15 which states that each request is applied exactly once, does not depend on the state of the object. Since `EnqLinkQueue` does not access *Toggles* or the *applied* field of $EnqS$, it follows that each ENQUEUE request $req$ in $\alpha$ is applied exactly once. Let $req'$ be the ENQUEUE request that applies $req$. By definition and by the pseudocode (line 33), it follows that $req'$ calls `enqueue` for $req$. We call *node allocated for req in* $\alpha$ the node that is allocated by this instance of `enqueue`.

As in the correctness proof of P-Sim, we linearize each ENQUEUE at the point that the successful `SC` of the `Attempt` of the request that applies the ENQUEUE is executed; ties are broken by the order imposed by threads' identifiers.

Denote by $\alpha_m$ the prefix of $\alpha$ which ends at $\texttt{SC}_m$. Let $\alpha_0$ be the empty execution. Denote by $E_m$ the sequence of the ENQUEUE requests applied up until $C_m$, in the order defined by their linearization points; let $E_0 = \lambda$, i.e. $E_0$ is the empty sequence. Denote by $E_m - E_{m-1}$ the suffix of $E_m$ that does not contain any of the instances of ENQUEUE in $E_{m-1}$.

**Lemma 5.28.** *No thread executes lines 39-42 and lines 44-48 of the code between* $C_0$ *and* $C_1$.

*Proof.* Fix any request $req$ (ENQUEUE or DEQUEUE) that is initiated between $C_0$ and $C_1$ and let $req'$ be the request that applies $req$; denote by $p_i$ the thread that initiates $req'$. Let $ls_i$ be the *Pool* element used by $p_i$ during the execution of the `Attempt` of $req'$. By initialization, $ls_i \rightarrow st.tail$ stores a pointer to the dummy node at $C_0$; moreover, $ls_i \rightarrow st.first = \perp$ and $ls_i \rightarrow st.last = \perp$ at $C_0$.

Assume first that $req$ is an ENQUEUE request. By the pseudocode, thread $p_i$ calls `EnqLinkQueue` before executing the for loop of line 26. Since $ls_i \rightarrow first$ can change only if $p_i$ calls `enqueue` and this occurs only in the body of the for loop of line 26, it follows that the condition of line 38 of `EnqLinkQueue` is evaluated by $p_i$ to *false*. Therefore, $p_i$ does not execute lines 39-42 between $C_0$ and $C_1$.

Assume now that *req* is a Dequeue request. By initialization, $ls_i \rightarrow st.head$ stores a pointer to the dummy node at $C_0$; moreover, the *next* field of this dummy node is equal to $\perp$. Since *head* changes only by executing line 16, it is a straightforward induction to show that each time the if statement of line 15 is executed between $C_0$ and $C_1$, its condition is evaluated to *false*. It follows that the condition of line 48 of `DeqLinkQueue` is evaluated to *false*. Therefore, lines 44-48 of `DeqLinkQueue` are not executed by $p_i$ between $C_0$ and $C_1$. ∎

**Lemma 5.29.** *Fix any index $m > 0$. The following claims hold for $C_m$:*

1. *If $m > 1$, $\texttt{CAS}_{m-1}$ is the only successful* `CAS` *executed between $\texttt{SC}_{m-1}$ and $\texttt{SC}_m$; $\texttt{CAS}_{m-1}$ is performed on $tail_{m-1} \rightarrow next$ and writes the value $first_{m-1}$ there.*

2. *Let $nd_0, \ldots, nd_{q_m}$ be the nodes that are traversed, in order, if, at $C_m$, the next pointers are followed starting from node $nd_0$. Then, for each $j$, $1 \leq j \leq q_m$, $nd_j$ is the node allocated in $\alpha$ for the* Enqueue *that corresponds to the $j$th* Enqueue *in $E_{m-1}$; moreover, $tail_m$ points to $nd_{q_m}$ at $C_m$.*

3. *Let $nd'_1, \ldots, nd'_{f_m}$ be the nodes that are traversed, in order, if, at $C_m$, the next pointers are followed starting from the node pointed to by $first_m$. Then, $f_m > 0$ and for each $j$, $1 \leq j \leq f_m$, $nd'_j$ is the node allocated in $\alpha$ for the* Enqueue *that corresponds to the $j$th* Enqueue *in $E_m - E_{m-1}$, and $nd'_{f_m}$ is the node pointed to by $last_m$.*

*Proof.* We prove the claim by induction on $m$.

**Induction Base (m = 1).** Claim (1) holds trivially. We continue to prove claim (2). By the pseudocode and by Lemma 5.28, it follows that the value of $ls_{i_1} \rightarrow st.tail$ does not change until $\texttt{SC}_1$. Recall that, initially, $ls_{i_1} \rightarrow st.tail$ points to the dummy node. Since each thread works on distinct elements of the *Pool* array, it follows that $ls_{i_1} \rightarrow st.tail$ points to the dummy node at $C_1$. So, $tail_1$ points to $nd_0$ at $C_1$. Moreover, Lemma 5.28 and the pseudocode (lines 8-10, 39, and 48) imply that the *next* field of the dummy node points to $\perp$ at $C_1$, thus $q_1 = 0$. This concludes the proof of claim (2).

We finally prove claim (3). Recall that $ls_{i_1} \rightarrow st.first = \perp$ and $ls_{i_1} \rightarrow st.last = \perp$ at $C_0$. Lemma 5.28 implies that the values of $ls_{i_1} \rightarrow st.first$ and $ls_{i_1} \rightarrow st.last$ do not change by executing lines 41 and 42 of `EnqLinkQueue`. Thus, claim (3) is implied by the correctness of P-Sim.

**Induction hypothesis.** Fix any index $m > 1$ and assume that the claim holds for every $0 < m' < m$.

**Induction step.** We prove the claim for $m$.

We start by proving claim (1). We first prove that $p_{i_m}$ executes the `CAS` of line 39 during the execution of `EnqLinkQueue`. By induction hypothesis (claim (3)), $f_{m-1} > 0$, so $first_{m-1} \neq \perp$ at $C_{m-1}$. By the definition of $C_{m-1}$ and $C_m$, $EnqS$ is not modified between $C_{m-1}$ and $C_m$. Since $\texttt{SC}_m$ is a successful `SC` instruction, it follows that $p_{i_m}$ executes $\texttt{LL}_m$ between $C_{m-1}$ and $C_m$. By the pseudocode (lines 19 and 20), it follows that at the configuration $C$ that results from the execution of line 21 by $p_{i_m}$, it holds that $ls_{i_m} \rightarrow st.first = first_{m-1}$, $ls_{i_m} \rightarrow st.tail = tail_{m-1}$ and $ls_{i_m} \rightarrow st.last = last_{m-1}$; moreover, the value of $ls_{i_m} \rightarrow st.first$ does not change between $C$ and the execution of line 38 of `EnqLinkQueue` by $p_{i_m}$. Thus, the condition of the if statement of line 38 is evaluated to *true* and $p_{i_m}$ executes the `CAS` of line 39 (which we denote by $\texttt{CAS}'$ in the rest of the proof).

We next argue that $\texttt{CAS}'$ is executed on $tail_{m-1} \rightarrow next$. Recall that $ls_{i_m} \rightarrow tail = tail_{m-1}$ at $C$. By the pseudocode, it follows that between $C$ and the execution of line 39 of `EnqLinkQueue` by $p_{i_m}$, the value of $ls_{i_m} \rightarrow st.tail$ does not change. Thus, by the pseudocode (lines 19), it follows that $\texttt{CAS}'$ is executed on $tail_{m-1} \rightarrow next$.

By the induction hypothesis (claim (2)), it follows that $tail_{m-1} \rightarrow next = \perp$ at $C_{m-1}$. If $\texttt{CAS}'$ fails, it follows that at least one successful `CAS` is executed on $tail_{m-1} \rightarrow next$ between $C_{m-1}$ and the execution of $\texttt{CAS}'$; let $\texttt{CAS}_x$ be the first such `CAS`.

We next argue that $\texttt{CAS}_x = \texttt{CAS}_{m-1}$ by proving that each `CAS` on any variable other than $tail_{m-1} \rightarrow next$ which is executed between $C_{m-1}$ and the execution of $\texttt{CAS}_x$ fails. Let $\texttt{CAS}_y \neq \texttt{CAS}_x$ be any `CAS` that is executed between $C_{m-1}$ and the execution of $\texttt{CAS}_x$. By the pseudocode (lines 19-21, and 44-47), it follows that $\texttt{CAS}_y$ is executed on $tail_j$ for some $j$, $1 \leq j < m$. Notice that once the *next* field of a node changes to a value which is not $\perp$, then it never becomes $\perp$ again (since the structures of type $Node$ are not recycled). This, and induction hypothesis (claims (1), (2), and (3)) imply that for each $j$, $1 \leq j < m$, $tail_j$ points to a distinct node; moreover, if $j < m-1$, $tail_j \rightarrow next \neq \perp$ after $C_{j+1}$. So, for each $j$, $1 \leq j < m-1$, $tail_j \rightarrow next \neq \perp$ after $C_{m-1}$. It follows that $\texttt{CAS}_y$ is not successful. Thus, $\texttt{CAS}_x = \texttt{CAS}_{m-1}$. Therefore, $\texttt{CAS}_{m-1}$ is executed on $tail_{m-1} \rightarrow next$ and writes the value $first_{m-1}$ there. Once $tail_{m-1} \rightarrow next$ changes to a non-$\perp$ value, no other `CAS` on it can succeed. Recall that the same is true for all other `CAS` instructions that are executed on variables other than $tail_{m-1} \rightarrow next$. This concludes the proof of claim (1).

We continue to prove claim (2). Recall that $p_{i_m}$ executes the `CAS` of line 39 of `EnqLinkQueue`, and therefore it also executes lines 40-42 of `EnqLinkQueue`. Let $C'$ be the configuration that results when $p_{i_m}$ finishes the execution of `EnqLinkQueue`. Since $ls_{i_m}$ points to one of the *Pool* elements owned by $p_{i_m}$ (so no other thread can change $ls_{i_m}$), the pseudocode (lines 19-20) implies that $ls_{i_m} \rightarrow st.tail = last_{m-1}$, $ls_{i_m} \rightarrow st.first = \perp$, and $ls_{i_m} \rightarrow st.last = \perp$, at $C'$; moreover, the value of $ls_{i_m} \rightarrow st.tail$ does not change from $C'$ to $C_m$. By the pseudocode (lines 39, 48) it follows that $last_{m-1} \rightarrow next$ does not change from $C_{m-1}$ to $C_m$. Thus, $ls_{i_m} \rightarrow st.tail \rightarrow next = \perp$ at $C_m$. Notice that $tail_m = ls_{i_m} \rightarrow st.tail$ at $C_m$. Claim (2) now follows by induction hypothesis (claims (1), (2), and (3)) and by the way linearization points are assigned to the Enqueue requests.

We finally prove (3). Recall that $ls_{i_m} \rightarrow st.first = \perp$, and $ls_{i_m} \rightarrow st.last = \perp$, at $C'$. By definition, $C'$ precedes the execution of the for loop of line 26 by $p_{i_m}$. It follows that claim (3) can be derived by the correctness of P-Sim and by the way linearization points are assigned to Enqueue requests. ∎

We continue to study the behavior of the dequeuers. We first describe how to assign linearization points to each instance of Dequeue executed in $\alpha$.

Recall that $DeqS \rightarrow st.head$ initially points to $nd_0$, i.e. to the initial dummy node. Lemma 5.29 implies that, for each $m$, at $C_m$, the nodes which can be reached by following *next* pointers, starting from the initial dummy node, contain the same values, in order, as those of the queue that would result if the Enqueue requests in $E_{m-1}$ were applied sequentially to a queue that initially contains only a dummy node initialized in the same way as $nd_0$. Moreover, the pseudocode of `Attempt` is different than that of P-Sim in the following: (1) for each Dequeue that is simulated locally by any thread, `DeqLinkQueue` may be called, and (2) if for some Dequeue request *req*, the execution of `DeqLinkQueue` returns *false*, then the response value for it and for all Dequeue requests that are simulated by the same `Attempt` after *req* are set to $\perp$.

We say that a dequeue request *req* initiated by some thread $p_i$ *is applied* if there is some request $req'$ (that might be *req* or some other request) for which all the following conditions hold: (1) the last `Read` on *Toggles* that is executed by $req'$ returns a value for its $i$th bit which is different from the value returned by the last `Read` on $DeqS \rightarrow applied[i]$ (line 20) executed by the `Attempt` of $req'$, (2) *req* is recorded in Announce[i] when the last read of *Toggles* is executed by the `Attempt` of $req'$, and (3) the execution of the `SC` of line 35 on *DeqS* by the `Attempt` of $req'$ succeeds. When these conditions hold, we sometimes say that $req'$ applies *req*.

Since the new version of `Attempt` handles *Toggles* and the *applied* field of *DeqS* in the same way as the `Attempt` of P-Sim, it can be proved, by using the same arguments as those presented for P-Sim up until Lemma 5.15, that each instance of Dequeue in $\alpha$ is applied exactly once.

Fix any request $req'$ such that $req'$ applies a bunch of Dequeue requests all of which return values different from $\perp$. We linearize the bunch of requests applied by $req'$ at the point that the successful `SC` (line 35) is executed by $req'$; ties are broken by the order imposed by threads' identifiers.

Fix any request $req'$ such that $req'$ applies a set $\Delta$ of Dequeue requests that return $\perp$ and possibly some other Dequeue requests that have a non $\perp$ response. Let *req* be the Dequeue request from $\Delta$ that has been initiated by the thread with the smallest identifier; let $p_i$ be this thread. Denote by *DLQ* the instance of `DeqLinkQueue` that is executed during the $i$th iteration of the last execution of the for loop of line 26 performed by the instance of `Attempt` executed by $req'$; the definition of *req* and the pseudocode imply that `DeqLinkQueue` is indeed called during the $i$th iteration of this for loop, so *DLQ* is well-defined. We linearize all the Dequeue requests applied by $req'$ (independently of whether they return a value equal to $\perp$ or not) at the point that line 43 of *DLQ* is executed by $req'$; ties are broken by the order imposed by threads' identifiers.

In order to prove consistency, we introduce the following notation. Denote by $\texttt{SC}'_m$, $m > 0$, the $m$th successful `SC` instruction in $\alpha$ and let $\texttt{LL}'_m$ be its matching `LL`; notice that $SC'_m$ may be an `SC` on either *EnqS* or on *DeqS*. Obviously, between $\texttt{SC}'_m$ and $\texttt{SC}'_{m+1}$, neither *EnqS* nor *DeqS* is modified. Denote by $\alpha_m$, the prefix of $\alpha$ which ends at $\texttt{SC}'_m$ and let $C'_m$ be the configuration that results from the execution of $\texttt{SC}'_m$; let $C'_0 = C_0$. Let

$\alpha_0$ be the empty execution. Denote by $L_m$ the sequence of the requests in $\alpha_m$ in the order defined by their linearization points; let $L_0 = \lambda$, i.e. $L_0$ is the empty sequence.

Let $EnqS_m$ and $DeqS_m$ be the values of variables $EnqS$ and $DeqS$, respectively, at $C_m$. Let $H'_m = nd_1 \dots nd_{q_m}$ be the sequence of nodes that are traversed, in order, if, at $C_m$, we follow the *next* pointers, starting from the node pointed to by $DeqS_m \rightarrow st.head$, up until we reach a node whose *next* field is equal to $NULL$; nodes $nd_1, \dots, nd_{q_m}$ are the *reachable* nodes from the node pointed to by $DeqS_m \rightarrow st.head$ at $C_m$. Let $H''_m = nd'_1 \dots nd'_{f_m}$ be the sequence of nodes that are traversed, in order, if, at $C_m$, we follow the *next* pointers starting from the node pointed to by $first_m$ up until we reach a node whose *next* field is equal to $NULL$; nodes $nd'_1, \dots, nd'_{f_m}$ are the *reachable* nodes from the node pointed to by $first_m$ at $C_m$. If at $C_m$, $EnqS_m \rightarrow st.tail \rightarrow next$ points to $EnqS_m \rightarrow st.first$, then let $H_m$ be the sequence of values, in order, contained in the nodes in $H'_m$ (notice that in this case $H''_m$ is a suffix of $H'_m$); otherwise, let $H_m$ be the sequence of values, in order, contained in the nodes in $H'_m \cdot H''_m$.

Let $Q_m$ be the queue that is created if the requests in $L_m$ are applied sequentially on a queue that initially contains a dummy node initialized in the same way as the initial dummy node in $\alpha$. Let $S_m$ be the sequence of values, in order, contained in the nodes of $Q_m$. Let $H_0$ and $S_0$ be sequences containing only one value each, that of the initial dummy node.

**Lemma 5.30.** *For each $m \geq 0$, the following claims hold: (1) $H_m = S_m$, and $L_m$ is a linearization order for $\alpha_m$.*

*Proof.* By induction on $m$.

**Base Case.** The claims hold trivially for $m = 0$.

**Induction Hypothesis.** Let $m > 0$, and assume that the claims hold for $m - 1$.

**Induction Step.** We prove that the claims hold for $m$. Suppose that $L_{m-1} - L_m$ contains ENQUEUE requests only. Then, $\texttt{SC}'_m$ is a successful `SC` on $EnqS$. In this case, the claims hold by induction hypothesis, Lemma 5.29, and the fact that ENQUEUE returns `ack`.

Assume next that $L_{m-1} - L_m$ contains a set $D$ of DEQUEUE requests. Let $\texttt{SC}'_d$ be the first successful `SC` on $DeqS$ after $C'_{m-1}$. Notice that $d \geq m$. Let $req_d$ be the DEQUEUE request that executes $\texttt{SC}'_d$, let $p_d$ be the thread that initiated $req_d$, and let $ls_d$ be the element of *Pool* used by $p_d$ during the instance of `Attempt` that executes $\texttt{SC}'_d$.

Assume first that all Dequeue requests in $D$ return a value other than $\perp$. Then, by the way linearization points are assigned, it follows that $d = m$. In this case, the induction hypothesis, Lemma 5.29, and the correctness of P-Sim, imply that the claims hold.

Assume finally that some requests applied by $req_d$ return $\perp$. We first argue that all Dequeue requests in $D$ are applied by $req_d$. Assume, by contradiction, that there is at least one Dequeue request $req_{d'} \neq req_d$ that applies some of the Dequeue requests in $D$. By definition of $\texttt{SC}'_d$, $req_{d'}$ applies its SC which we denote by $\texttt{SC}'_{d'}$, after $\texttt{SC}'_d$. However, by the pseudocode (lines 19, 29, and 35) and by the way that linearization points are assigned, it follows that $req_{d'}$ executes the LL that matches $\texttt{SC}'_{d'}$ before $\texttt{SC}'_m$ and therefore before $\texttt{SC}'_d$. Thus, $\texttt{SC}'_{d'}$ cannot be successful. This contradicts the assumption that $req_{d'}$ applies some of the Dequeue requests in $D$. Therefore, all Dequeue requests in $D$ are applied by $req_d$.

Let $req$ be the Dequeue request among those that return $\perp$ in $D$ that has been initiated by the thread with the smallest identifier; let $p_i$ be this thread. Denote by $DLQ$ the instance of DeqLinkQueue that has been executed during the $i$th iteration of the last execution of the for loop of line 26 performed by the instance of Attempt of $req_d$; the definition of $i$ and the pseudocode imply that DeqLinkQueue is indeed called during the $i$th iteration of this for loop, so $DLQ$ is well-defined. Let $C$, $C'$, and $C''$ be the configurations just before the execution of lines 43, 48, and 49, respectively, of $DLQ$ by $req_d$. By the way linearization points are assigned, $C$ precedes the execution of $\texttt{SC}'_m$.

Denote by $h_i$ the value of $ls_d.st.head$ after the $(i-1)$st iteration of the for loop of line 26 has been executed by $req_d$. The induction hypothesis, Lemma 5.29, the correctness of P-Sim, and the pseudocode, imply that after the execution of the first $(i-1)$ iterations of the for loop of line 26 by $req_d$ (i.e. after those iterations that cope with Dequeue requests that return a value not equal to $\perp$), the claims hold. Since the request that is processed during the $i$th iteration is $req$ which returns $\perp$, the pseudocode (lines 43-49, and 14-16) implies that $ls_d.st.head$ is equal to $h_i$ at $C$.

Let $\texttt{SC}_e$ be the last successful SC on $EnqS$ preceding $SC'_m$. If there is no such SC, then all Dequeue requests in $D$ return $\perp$ and are linearized before the first Enqueue request is linearized. This and the induction hypothesis imply that the claims hold.

So, assume that $\texttt{SC}_e$ is well defined. Suppose that $\texttt{SC}_e$ writes the value $EnqS_e$ in $EnqS$. Since $C$ occurs between $\texttt{SC}'_{m-1}$ and $\texttt{SC}'_m$, it follows that $\texttt{SC}_e$ precedes $C$. We first argue that at least one successful `CAS` is executed between $\texttt{SC}_e$ and $C'$. Assume, by contradiction, that this is not the case. Since the response value for *req* is $\perp$, the pseudocode (lines 43, 49, 25, 28 -30, and 14-16) implies that $DLQ$ returns *false*. Since $p_d$ is the only thread updating $ls_d$ and $ls_d.st.head \rightarrow next = \perp$ at $C''$, the pseudocode implies that $ls_d.st.head \rightarrow next = \perp$ at $C$. Thus, $DLQ$ evaluates the condition of the if statement of line 43 to *true*. Since no successful `CAS` is executed after $\texttt{SC}_e$, Lemma 5.29 implies that $EnqS \rightarrow st.tail \rightarrow next = \perp$ and $EnqS \rightarrow st.first \neq \perp$ at each configuration between $\texttt{SC}_e$ and $C'$. Since the `LL` of line 44 comes after $C$, $DLQ$ reads the value for $EnqS$ written by $\texttt{SC}_d$. Thus, the condition of the if statement of line 48 is evaluated to *true* and the `CAS` of line 48 is executed by $DLQ$ and it is successful. This contradicts our assumption that no successful `CAS` is executed on between the execution of $\texttt{SC}_e$ and $C'$. Thus, there is at least one successful `CAS` that is executed between the execution of $\texttt{SC}_e$ and $C'$. We remark that this `CAS` is executed on $EnqS_e \rightarrow st.tail \rightarrow next$. Notice that once a successful `CAS` is executed on $EnqS_e \rightarrow st.tail \rightarrow next$, no other `CAS` on it can succeed. Thus, there is exactly one successful `CAS` on it between $\texttt{SC}_e$ and $C'$. Denote by $\texttt{CAS}_e$ this successful `CAS`.

We argue that $\texttt{CAS}_e$ is executed before $C$. Assume, by contradiction, that $\texttt{CAS}_e$ is executed between $C$ and $C'$. Recall that the condition of the if statement of line 43 executed by $DLQ$ is evaluated to *true*. Recall that $ls_d.st.head$ is equal to $h_i$ at $C$. Lemma 5.29 implies that only $EnqS_e \rightarrow st.tail \rightarrow next$ and $EnqS_e \rightarrow st.last \rightarrow next$ can be equal to $\perp$ at $C$. Since $\texttt{CAS}_e$ occurs after $C$, the induction hypothesis, the pseudocode, and Lemma 5.29 imply that $ls_d.st.head \neq EnqS_e \rightarrow st.last$ at $C$. Therefore, $ls_d.st.head = EnqS_e \rightarrow st.tail$ at $C$. Lemma 5.29 implies that $\texttt{CAS}_e$ changes $EnqS_e \rightarrow st.tail \rightarrow next$ to point to $EnqS_e \rightarrow st.first$. Recall that $EnqS_e \rightarrow st.first \neq \perp$. Thus, in all configurations between the execution of $\texttt{CAS}_e$ and $C''$, it holds that $ls_d.st.head \rightarrow next \neq \perp$. It follows that the condition of the if statement of line 49 is evaluated to *false* by $DLQ$, so $DLQ$ returns *true*. This contradicts the fact that the response for *req* is $\perp$. It follows that $\texttt{CAS}_e$ is executed before the execution of line 43 by $DLQ$. Then, Lemma 5.29 imply that $ls_d.st.head = EnqS_e \rightarrow st.last$ at $C$. Recall that we argued that the claims hold until $C$.

We conclude that the sequential queue which is formed by applying sequentially all the requests in $L_{m-1}$, in order, as well as those requests applied by $req_d$ up until *req* (excluding *req*), in the order of thread ids, is empty (i.e. contains only the dummy node).

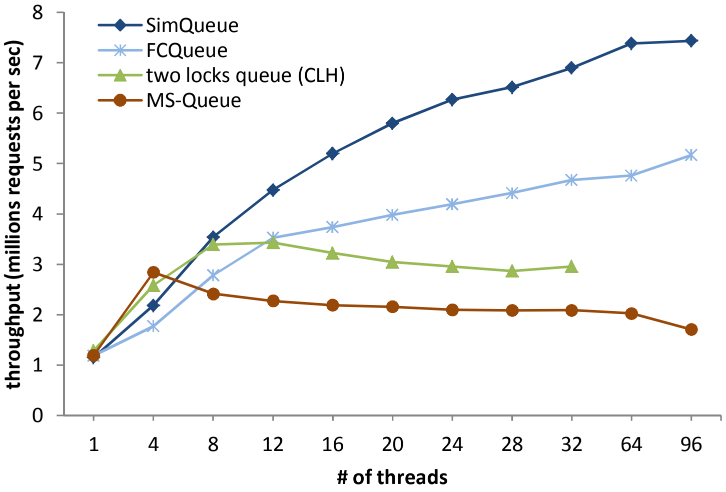


Figure 5.12: Performance of SimQueue.

Thus, linearizing $req$ and all other requests applied by $req_d$ after $req$ at $C$ does not violate the claims.

We remark that if there are ENQUEUE requests in $L_m - L_{m-1}$ as well, they are all linearized after the DEQUEUE requests in $D$ because of the way that linearization points are assigned. Given that the DEQUEUE requests in $D$ return a consistent response, the induction hypothesis, Lemma 5.29, and the fact that ENQUEUE returns ack imply that the consideration of these ENQUEUE requests does not violate the claims.

This concludes the proof of the induction step and thus also the proof of the lemma.

∎

**Theorem 5.8.** SimQueue *is a linearizable wait-free implementation of a concurrent queue.*

### 5.6.3 Performance evaluation

We now experimentally compare the performance of SimQueue with that of state-of-the-art concurrent queue implementations, like the lock-based implementation using two CLH locks by Michael and Scott [50], the lock-free algorithm (MSQueue) presented in [50], and the queue implementation using flat-combining (FCQueue) presented in [33, 34]. Similarly to the experiment performed in [50], $10^7$ pairs of an enqueue and a dequeue were executed as the number of threads $n$ increases. The average throughput of each algorithm was measured and the results are illustrated in Figure 5.12. As in previous experiments, we simulate a random workload after the completion of each request.

As shown in Figure 5.12, SimQueue significantly outperforms all other implementations for $n > 4$. More specifically, SimQueue is up to 3.6 times faster than the lock-free implementation, up to 2.25 times faster than the spin-lock based implementation, and up to 1.5 times faster than flat-combining.

Similarly to the experiment of Figure 5.11, the queue implementation based on CLH spin locks outperforms the lock-free algorithm. We remark that the queue implementation based on CLH locks performs much better than the CLH lock-based stack implementation, since in the queue implementation there are two CLH locks (one handling enqueues and one handling dequeues) that are employed; this leads to increased parallelism. Flat-combining outperforms all queue implementations other than SimQueue. However, SimQueue achieves much better performance than flat-combining for almost any number of threads. In addition to the points discussed for the performance of P-Sim and flat-combining in Section 5.3, this is due to the fact that SimQueue uses two instances of P-Sim, thus achieving increased parallelism by having enqueuers and dequeuers run concurrently.

# Chapter 6

# Highly-Efficient Blocking Synchronization Algorithms

## 6.1 CC-Synch: An efficient synchronization algorithm for the CC model

In this section, we present the CC-Synch synchronization algorithm. The time complexity of CC-Synch is $O(h+t)$ RMRs, where $h$ is an upper bound of the requests that a combiner may serve and $t$ is the size of the shared memory data that the combiner should access in order to serve these $h$ requests. The amortized time complexity is $O(d)$, where $d$ is the average number of RMRs required to serve a single request. The space overhead of CC-Synch is $O(n)$.

### 6.1.1 Algorithm description

CC-Synch (Algorithm 17) maintains a list which contains, in addition to a dummy node which is always the last node of the list, one node for each thread that has initiated an active request. Each thread first announces its request by recording it in the last node of the list (i.e. in the current dummy node) and by inserting a new node as the last node of the list (which will comprise the new dummy node). We say that a node *is assigned* to the thread that has written the request recorded in it; i.e. each thread is assigned the previous node to the node that it inserts.

The thread that is assigned the head node of the list plays the role of the combiner, so it is the only thread that is allowed to access the shared data. The combiner starts by serving its own request. Other threads that have announced requests perform spinning on the *locked* field of their assigned node. The combiner does not give up the lock when it completes the execution of its request; it rather continues accessing the next elements of the list, it serves the requests announced in these nodes, and sets the *locked* field of these nodes to *false* to stop the threads that have been assigned these nodes from spinning. It also changes their *completed* field to *true* to identify that their requests have been completed.

The combiner completes its execution when it serves either all requests in the list or a pre-specified number $h$ of such requests. In the later case, the combiner identifies the thread, which owns the next to the last node that the combiner helps, as the new combiner; this is done by changing the *locked* field of this node to *false* while leaving its *completed* field equal to *false*.

We now give a more detailed description of CC-Synch. Pointer $Tail$ is a Swap object which initially points to a dummy node. Whenever thread $p_i$ wants to announce a request $req$, it executes a Swap operation to $Tail$ (line 5) in order to read the pointer to the dummy node pointed to by $Tail$ and update $Tail$ to point to its node (i.e. to the node pointed to by $p_i$'s local variable $next_i$). Once this has been performed, $p_i$ has been assigned the node that was previously pointed to by $Tail$, so it announces its request by recording $req$ in the $req$ field of this node (line 6) and then it sets the $next$ field of this node to point to the new dummy node (line 7). After that, $p_i$ starts spinning on field *locked* of its assigned node until this field becomes *false*. When $p_i$ reads *false* in *locked*, either its request has

```
struct Node {
   Request req;
   RetVal ret;
   boolean wait;
   boolean completed;
   Node *next;
};

// Tail initially points to a dummy node
// with value $\langle \perp, \perp, \mathit{false}, \mathit{false}, \text{null} \rangle$
shared Node *Tail;

// The following variable is private to each thread $p_i$; it is a pointer to a
// struct of type Node; it initially points to a struct with
// initial value $\langle \perp, \perp, \mathit{true}, \mathit{true}, \text{null} \rangle$
private Node *$node_i$;

RetVal CC-Synch(Request req) {                  // Pseudocode for thread $p_i$
   Node *nextNode, *tmpNode, *tmpNodeNext;
   int counter = 0;
   $node_i \rightarrow wait = true$;
   $node_i \rightarrow next =$ null;
   $node_i \rightarrow completed = false$;
   $nextNode = node_i$;
   $node_i = \texttt{Swap}(Tail, node_i)$;              // curNode is assigned to $p_i$
   $node_i \rightarrow req =$ req;                     // $p_i$ announces its request
   $node_i \rightarrow next =$ nextNode;
   while ($node_i \rightarrow wait == true$)          // $p_i$ spins until it is unlocked
      nop;
   if ($node_i \rightarrow completed == true$)        // if $p_i$'s req is already applied
      return $node_i \rightarrow ret$;                // $p_i$ returns its return value
   tmpNode = $node_i$;                         // otherwise $p_i$ is the combiner
   while (tmpNode $\rightarrow$ next $\neq$ $null$ AND counter $< h$){
      counter = counter + 1;
      tmpNodeNext=tmpNode$\rightarrow$next;
      apply tmpNode$\rightarrow$req to object's state
        and store the return value to tmpNode$\rightarrow$ret;
      tmpNode$\rightarrow$completed = $true$;           // tmpNode's req is applied
      tmpNode$\rightarrow$wait = $false$;               // unlock the spinning thread
      tmpNode = tmpNodeNext;                    // and proceed to the next node
   }
   tmpNode$\rightarrow$wait = $false$;                  // unlock next node's owner
   return $node_i$ $\rightarrow$ret;
}
```

Algorithm 17: Pseudocode for CC-Synch.

been executed by the combiner or $p_i$'s record is the first in the list and therefore it owns the lock. In the former case, $p_i$ simply returns (line 10), whereas in the later, $p_i$ becomes the combiner.

We remark that the list could grow forever while the combiner thread $p$ traverses it since a thread may add a node at the end of the list more than once after its request has been served by $p$. In order to prevent $p$ from traversing a continuously growing list, an upper bound $h$ (lines 12 and 13) on the number of requests that $p$ may serve is employed; once $p$ serves $h$ requests, it identifies the thread that has been assigned the next node of the list as the new combiner, and returns. Our experiments show that the choice of $h$ does not significantly impact the performance of the algorithm. Specifically, setting $h$ to a value equal to $cn$, where $c > 0$ is a small constant, is a good choice in terms of performance.

### 6.1.2 Time and space complexity

By the pseudocode (Algorithm 17), it follows that each thread returns either on line 10 or on line 20. In case that $p_i$ returns on line 10, it follows that $p_i$ executes a constant number of RMRs. Assume now that $p_i$ returns on line 20. By the pseudocode (lines 12 and 13), $p_i$ executes at most $h$ iterations of the while loop (lines 13-18). Lines 14-18 contribute just a constant number of RMRs, and line 14 is a local request. Thus, $p_i$ executes $O(h + t)$ RMRs, where $t$ is the size of the shared memory data that they should be accessed in order to serve these $h$ requests, where we have assumed that the cache size of $p_i$'s processor is greater than $t$. Notice that the amortized time complexity is $O(d)$, where $d$ is the average number of RMRs required to serve a single request. We remark that in most cases, $d$ equals a small constant. The space overhead of CC-Synch is $O(n)$, since each thread maintains a struct of type $Node$.

### 6.1.3 Required memory barriers

When implementing CC-Synch, memory barriers may need to be inserted in the code to ensure its correct execution. In architectures that implement either the TSO (Total Store Order) or the PO (Process Order) consistency model, we need to insert just one store memory barrier. These memory consistency models are very common and they are used in many contemporary multiprocessors, among which those that we used for our experiments. The first model is implemented on SPARC machines of version v8 and newer [59], while the second is implemented on AMD64 [1] and on Intel64 [22] architectures. SPARC processors support weaker consistency memory models as well, but they are rarely used, and the TSO model is the default option for the Solaris operating system [48]. Both of these consistency models do not reorder two read operations, and the same holds for two store operations [48]. However, a read can be reordered with an older store only in case that the read and the store instructions access different memory locations [48]. Thus, for the correct execution of lines 6-7 and lines 17-18, no store barrier is needed. Similarly, no load barrier is needed for lines 9 and 10. A store memory barrier is inserted just before the return instruction of line 21. In cases where a weaker memory model is considered, additional memory barriers may have to be inserted; however, this is not the case in the architectures we employed for our experiments.

### 6.1.4 Correctness proof

In this section, we present the correctness proof of CC-Synch. Let $\alpha$ be any execution. Consider any configuration $C$ in $\alpha$. Let $Tail(C)$ be the value of $Tail$ at $C$. For each $i$, $1 \leq i \leq n$, denote by $node_i(C)$ the value of variable $node_i$ at $C$. Denote by $C^-$ the configuration just preceding $C$ and let $C_0$ be the initial configuration. The notation of this proof is summarized in Table 6.1.

We start by proving the following lemma which states that at each configuration $C$, $node_i$ points to a distinct node other than $Tail(C)$.

**Lemma 6.1.** *For any configuration $C$, the following claims hold:*

1. *for each $i$, $Tail(C) \neq node_i(C)$;*

2. *for each $i, j$, $i \neq j$, $node_i(C) \neq node_j(C)$.*

*Proof.* We prove the lemma by induction on $C$.

**Base case ($C = C_0$).** Recall that $Tail(C_0)$ points to a dummy node. Also, for each $i$, $1 \leq i \leq n$, $node_i(C_0)$ points to a distinct node (allocated for thread $p_i$) other than that pointed to by $Tail(C_0)$. Thus, the claim holds.

**Induction hypothesis.** Let $C \neq C_0$ be any configuration in $\alpha$ and assume that the claim holds at configuration $C^-$.

**Induction step.** We now prove that the claim holds for $C$. Denote by $s$ the step taken at $C^-$ (that results in $C$) and let $p_j$ be the thread that executes $s$. If $s$ is not the execution of a `Swap` operation (line 5), the claim holds at $C$ by the induction hypothesis since neither $Tail$ no any of the $node_i$ variables, $1 \leq i \leq n$, change their values from $C^-$ to $C$. Thus, assume that $s$ is the execution of a `Swap` operation (line 5).

Then, by the pseudocode we get the following: (1) $node_j(C) = Tail(C^-)$ and (2) $Tail(C) = node_j(C^-)$.

By the induction hypothesis, for each $i$, $1 \leq i \leq n$, (1) $Tail(C^-) \neq node_i(C^-)$ and (2) for each $l$, $1 \leq l \leq n$, $l \neq i$, $node_i(C^-) \neq node_l(C^-)$. Since $s$ is a step of thread $p_j$, $node_i(C) = node_i(C^-)$, for each $i$, $1 \leq i \leq n$, $i \neq j$.

From the above, we get that (1) $Tail(C) = node_j(C^-) \neq node_i(C^-) = node_i(C)$, for each $i$, $1 \leq i \leq n$, $i \neq j$; also, $Tail(C) = node_j(C^-) \neq Tail(C^-) = node_j(C)$. Thus, for each $i$, $1 \leq i \leq n$, $Tail(C) \neq node_i(C)$, as needed by Claim 1.

By the induction hypothesis, for each $i, l$, $1 \leq i, l \leq n$, $i \neq l$ and $i, l \neq j$, it holds that $node_i(C) = node_i(C^-) \neq node_l(C^-) = node_l(C)$. Moreover, $node_j(C) = Tail(C^-) \neq node_i(C^-) = node_i(C)$, for each $i$, $1 \leq i \leq n$, $i \neq j$. This concludes the proof of Claim 2. ▮

We next prove that as long as a thread $p_i$ is executing an instance of CC-Synch, no other thread can write the *next* field of the node pointed to by $node_i$; notice that $node_i$ may not point to the same node during the course of the execution of an instance of CC-Synch by $p_i$.

**Lemma 6.2.** *Consider any instance $A$ of* CC-Synch *executed by some thread $p_i$. Let $C_f$ and $C_l$ be the first and the last configurations, respectively, of the execution of $A$. Then, for each configuration $C$, $C_f < C < C_l$, no thread $p_j$, $1 \leq j \leq n$, $j \neq i$, writes into $node_i(C) \rightarrow next$.*

*Proof.* Assume that there is a configuration $C_w$ between $C_f$ and $C_l$ at which a thread $p_j$, $j \neq i$, changes $node_i \rightarrow next$. By the pseudocode, it follows that $p_j$ executes either line 2 or line 7 at $C_w$ and therefore $p_j$ writes $node_j(C_w)$ at $C_w$. Lemma 6.1 implies that

| Notation | Description |
|---|---|
| $\alpha$ | Any execution of CC-Synch |
| $C$ | Any configuration in $\alpha$ |
| $C_0$ | The initial configuration of $\alpha$ |
| $C^-$ | The configuration just preceding $C$ |
| $Tail(C)$ | The value of $Tail$ at configuration $C$ |
| $p_i$ | The thread which its id is equal to $i$, $i \in \{1, \ldots, n\}$ |
| $node_i(C)$ | The value of $node_i$ at configuration $C$ |
| $m$ | The number of Swap operations executed in $\alpha$ |
| $S_l$ | The $l$th Swap in execution $\alpha$ |
| $A_l$ | The instance of CC-Synch that executes $S_l$ |
| $p_{i_l}$ | The thread that executes $S_l$ |
| $C_l$ | The configuration just after $S_l$ |
| $nd_l$ | The value returned by $S_l$ |

Table 6.1: Notation used in the proof of CC-Synch.

$node_j(C_w) \neq node_i(C_w)$. Thus, $p_j$ does not change $node_i$ at $C_w$, which is a contradiction. ▮

We next prove that the *next* field of variable $Tail$ is always equal to $\perp$.

**Lemma 6.3.** *In any configuration $C$ of $\alpha$, it holds that $Tail(C) \rightarrow next = \perp$.*

*Proof.* Assume, by the way of contradiction, that there is a configuration at which the *next* field of $Tail$ is not equal to $\perp$. Denote by $C_w$ the first such configuration. By the pseudocode, it follows that the step applied at $C_w^-$ must be the execution of one of the lines 2, 5, or 7. Lemma 6.1 implies that for each $i$, $1 \leq i \leq n$, $Tail(C_w^-) \neq node_i(C_w^-)$. Thus, $s$ cannot be the execution of line 2 or line 7. So, it must be that $s$ is the execution of a Swap of line 5; let $p_j$ be the thread that executes this Swap. By the pseudocode (lines 2 and 5), $Tail(C_w) = node_j(C_w^-)$ and $p_j$ sets the *next* field of $node_j$ to $\perp$ by executing line 2 of its current instance of CC-Synch. By Lemma 6.1 and the pseudocode (lines 2-5), it follows that $node_j \rightarrow next$ does not change after it is set to $\perp$ and until $C_w^-$. Thus, $Tail(C_w) \rightarrow next = node_j(C_w^-) \rightarrow next = \perp$, which contradicts our assumption that $Tail(C_w) \rightarrow next = \perp$. ▮

Let $m \geq 0$, be the number of Swap operations that are executed in $\alpha$[*]. Denote by $S_l$, $0 \leq l \leq m$, the $l$th Swap operation executed in $\alpha$, let $A_l$ be the instance of CC-Synch that executes $S_l$, let $p_{i_l}$ be the thread that executes $A_l$, and denote by $nd_l$ the return value of $S_l$. Let $C_l$ be the configuration just after the execution of $S_l$ and let $Q_0 = C_0$ be the initial configuration.

[*] We remark that $m$ may be $\infty$ if $\alpha$ is an infinite execution.

**Lemma 6.4.** *The following claims hold:*

1. *for each $l$, $0 \leq l < m$, and for each configuration $C$ such that $C$ follows $C_l$ and $A_l$ is active at $C$, it holds that either $nd_l \rightarrow next = \perp$ or $nd_l \rightarrow next = nd_{l+1}$ at $C$;*

2. *if $m$ is finite, at each configuration $C$ following $C_m$, it holds that either $nd_m \rightarrow next = \perp$ or $nd_m \rightarrow next = Tail(C)$.*

*Proof.* Fix any $l >\geq 1$. If $m$ is finite, $l$ is chosen so that $l \leq m$. By the pseudocode (line 5), it follows that $nd_l = Tail(C_l^-)$. By Lemma 6.3, $Tail(C_l^-) \rightarrow next = \perp$; since it is a `Swap` that is executed at $C_l^-$ and $nd_l = Tail(C_l^-)$, it follows that $nd_l \rightarrow next = \perp$ at $C_l$.

By the pseudocode (line 5), if follows that $nd_l = node_{i_l}(C_l) = node_{i_l}(C)$; it also follows that $Tail(C_l) = node_{i_l}(C_l^-)$. Lemma 6.2 implies that no thread other than $p_{i_l}$ can change $node_{i_l} \rightarrow next$ between $C_l$ and $C$. By the pseudocode, it is only through the execution of line 7 that $p_{i_l}$ changes $node_{i_l} \rightarrow next$. Thus, if $p_{i_l}$ has not executed line 7, it holds that $nd_l \rightarrow next = \perp$ at $C$ (and the claim follows). Assume now that $p_{i_l}$ executed line 7 in $A_l$ at some configuration $C'$. By the pseudocode (lines 14-18), $node_{i_l} \rightarrow next$ is set to be equal to $node_{i_l}(C_l^-) = Tail(C_l)$ at $C'$.

By the pseudocode, it follows that $nd_{l+1} = Tail(C_l)$. Thus, if $l < m$, $node_{i_l} \rightarrow next = Tail(C_l) = nd_{l+1}$ at $C'$. Lemma 6.2 implies that no thread can change $node_{i_l} \rightarrow next$ between $C'$ and $C$. Thus, $nd_l \rightarrow next = nd_{l+1}$ at $C$, as needed by Claim 1.

We continue to consider the case that $l = m$. Recall that $node_{i_m} \rightarrow next$ is set to be equal to $Tail(C_m)$ at $C'$. Since no `Swap` operation is executed from $C_m$ to $C$, $Tail(C) = Tail(C_m)$. Thus, at $C$, $node_{i_m} \rightarrow next = Tail(C)$, as needed by Claim 2. ∎

Consider a thread $p_i$ that executes an instance $A$ of CC-Synch at some configuration $C$. We say that $p_i$ is the combiner at $C$ if there is a configuration $C'$ in $A$ such that: (1) $C'$ precedes $C$, and (2) it holds that $node_i(C') \rightarrow wait = false$ and $node_i(C') \rightarrow completed = false$. Let $C_f$ be the first such configuration in $A$. We also say that $p_i$ is a combiner from $C_f$ until (and including) the execution of line 19 in $A$ (we show below that a combiner always returns on line 20).

We say that an instance $A$ of CC-Synch visits a node $nd$, if $A$ executes line 11 or 18 and sets its $tmpNode$ variable to point to $nd$; if $A$ is executed by thread $p_i$, we sometimes say that $p_i$ visits $nd$ (if $A$ visits $nd$). If $A$ visits a node $nd$, then there is an execution fragment starting from the configuration at which $A$ executes line 11 (or 18) to set *tmpNode* to point to $nd$ until the configuration that $A$ executes line 18 for the next time (or until $A$ executes line 19 if this was the last time that line 18 was executed by $A$ or if line 18 was not executed by $A$).

**Lemma 6.5.** *In each configuration $C$,*

1. *exactly one of the following conditions (i or ii) holds:*

   **(i)** *$Tail(C)$ points to a node $nd$ such that $nd \rightarrow completed =$ false and $nd \rightarrow wait =$ false, there is no combiner at $C$ and there is no thread poised to execute any of lines 11-19.*

   **(ii)** *$Tail(C)$ points to a node $nd$ such that $nd \rightarrow completed =$ false and $nd \rightarrow wait =$ true, there is exactly one combiner at $C$ and only the combiner is poised to execute any of lines 11-19.*

2. *if there is a combiner $p_i$ at $C$, the following claims hold:*

   **(i)** *$p_i$ is poised to execute one of the lines 6-19 at $C$ and it is not poised to execute line 10 at $C$;*

   **(ii)** *no thread other than $p_i$ executes lines 11-19 at $C$;*

   **(iii)** *suppose that $p_i$ is poised to execute one of the lines 11-19 at $C$, let $k$ be the number of nodes that have been visited by $p_i$ until $C$, denote by $nd'_l$, $1 \leq l \leq k$, the $l$th such node, and let $\beta_l$ be the execution fragment at which $p_i$ is visiting $nd'_l$; then, for each $l$, $1 \leq l \leq k$ and for each configuration $C'$ in $\beta_l$, if $nd'_l \neq Tail(C')$ there is one active thread $p_l$ such that $node_l(C') = nd'_l$, and either $p_l = p_i$ or $p_l$ executes one of the lines 6-10 at $C'$;*

   **(iv)** *if $C$ is the configuration just after $p_i$ has executed line 19 of the pseudocode and $k$ is the number of nodes that have been visited by $p_i$ until $C$, then the following hold: if $nd'_k \neq Tail(C^-)$, then $p_k$ is the unique combiner in the system at $C$, otherwise there is no combiner in the system at $C$.*

3. *if lines 16-17 have been executed $m$ times in total until $C$, then for each $l$, $1 \leq l \leq m$, lines 15-17 for the $l$th time were executed by a combiner that had its tmpNode variable equal to $nd_l$.*

*Proof.* We prove the claim by induction on $C$.

**Base Case ($C = C_0$).** No node is active at $C_0$ so there is no combiner at $C_0$. Moreover, $Tail(C_0)$ points to a dummy node which has its *completed* and *wait* fields equal to *false*, since no `Swap` operation has been executed at $C_0$. Thus, Claim 1 holds. Claim 2 trivially holds, since there is no combiner ar $C_0$. Furthermore, Claim 3 holds, since no thread has executed lines 15-17.

**Induction Hypothesis.** Let $C$ be any reachable configuration and assume that the claim holds in all configurations that precede $C$.

**Induction Step.** We prove that the claim holds at $C$. The induction step is proved by a case analysis on the step $s$ that is applied from $C^-$ to get $C$. Let $p_j$ be the thread that executes $s$.

1. $s$ is the execution of any of the lines 1, 2 and 4.

   In case $s$ is the execution of line 1, Lemma 6.1 implies that $node_j(C^-) \neq Tail(C^-)$. Thus, Claim 1.i holds by induction hypothesis. The rest of the claims also hold by the induction hypothesis.

   In case $s$ is the execution of either line 2 or line 4, the claims hold by the induction hypothesis.

2. $s$ is the execution of line 3.

   Lemma 6.1 implies that $node_j(C) \neq Tail(C)$. Thus, Claim 1 holds by the induction hypothesis. To prove Claim 2, it suffices to argue that $p_j$ does not become the combiner by executing $s$. Let $C_1$ be the configuration at which $p_j$ executes line 1 of the pseudocode. Assume by the way of contradiction that $node_j(C) \rightarrow wait = false$ in some configuration $C_w$ between $C_1$ and $C$. By the pseudocode (lines 1 and 2), $node_j$ does not change value between $C_1$ and $C$ by $p_j$. Since $node_j(C) \rightarrow wait = false$, there must be a thread that writes $node_j \rightarrow wait = false$ between $C_1$ and $C$. Let $p_m$ be a thread that does so. By the pseudocode, it follows that $p_m$ must execute either line 17 or line 19 at $C_w$. Since $p_m$ is active at $C_w$, Claim 1.i does not hold at $C_w$ and by the induction hypothesis (for $C_w$) it follows that Claim 1.ii must hold at $C_w$. Moreover, the induction hypothesis (Claim 2.ii) implies that $p_m$ must be the combiner at $C_w$. However then, the induction hypothesis (Claim 2.iii) implies that $p_j$ should be a thread executing lines 6-10 at $C_w$. This is a contradiction since $p_j$ executes line 3 at $C_w$. We conclude that $p_j$ is not a combiner at $C$, which is a contradiction.

3. $s$ is the execution of line 5.

   By the pseudocode (line 5), it follows that $node_j(C) = Tail(C^-)$ and $Tail(C) = node_j(C^-)$. Let $C_1$ be the configuration at which $p_j$ executes line 1. By the pseudocode, it follows that the value of the $node_j$ variable does not change from the configuration $C_1$ until $C^-$ by thread $p_j$. We start by proving that no thread other than $p_j$ can change the fields $node_j \rightarrow wait$ and $node_j \rightarrow completed$ from $C_1$ to $C^-$. Assume, by the way of contradiction, that there is some configuration $C_w$ following $C_1$ and preceding $C^-$ at which some thread $p_m \neq p_j$ changes one of the *next* or *completed* fields of the node pointed to by $node_j$. Lemma 6.1 and the pseudocode imply that this may happen only if $p_m$ executes any of lines 16, 17, or 19 at $C_w$ (with its $tmpNode$ variable equal to $node_j$); since $p_m$ is active at $C_w$, Claim 1.i does not hold at $C_w$. Thus, by the induction hypothesis (for $C_w$) it follows that Claim 1.ii must hold at $C_w$. Thus, there is a combiner at $C_w$. The induction hypothesis (Claim 2.ii) implies that $p_m$ must be the combiner at $C_w$. However then, the induction hypothesis (Claim 2.iii) implies that $p_j$ should be a thread executing lines 6-10 at $C_w$. This is a contradiction since $p_j$ is poised to execute line 5 at $C_w$. Thus, no thread other than $p_j$ writes $node_j(C^-) \rightarrow wait$ and $node_j(C^-) \rightarrow completed$ from $C_1$ to $C^-$, so $node_j(C^-) \rightarrow wait = true$ and $node_j(C^-) \rightarrow completed = false$.

   We now prove that Claim 1 holds. By the induction hypothesis (Claim 1), one of the following conditions hold at $C^-$: (Claim 1.i) $Tail(C^-)$ points to a node $nd$ such that $nd \rightarrow completed = false$ and $nd \rightarrow wait = false$, there is no combiner at $C^-$, and no active thread is executing any of the lines 6-19 at $C^-$, or (Claim 1.ii) $Tail(C^-)$ points to a node $nd$ such that $nd \rightarrow completed = false$ and $nd \rightarrow wait = true$, and there is exactly one combiner at $C^-$.

- Assume first that (1.i) is *true* at $C^-$. Since $Tail(C^-) \rightarrow \mathit{completed} = \mathit{false}$, $Tail(C^-) \rightarrow \mathit{wait} = \mathit{false}$, and $\mathit{node}_j(C) = Tail(C^-)$, it follows that $p_j$ is a combiner at $C$. Recall that there was no combiner at $C^-$, so $p_j$ is the only combiner in the system at $C$. Moreover, $p_j$ is poised to execute line 6 at $C$ (as needed by 2.i). Recall that no active thread is executing any of the lines 11-19 at $C^-$, so no thread other than the combiner is executing these lines at $C$ (as needed by 2.ii). Recall that $\mathit{node}_j(C^-) \rightarrow \mathit{wait} = \mathit{true}$ and $\mathit{node}_j(C^-) \rightarrow \mathit{completed} = \mathit{false}$. Since $Tail(C) = \mathit{node}_j(C^-)$, it follows that $Tail(C) \rightarrow \mathit{wait} = \mathit{true}$ and $Tail(C) \rightarrow \mathit{completed} = \mathit{false}$ (as needed by 1).
- Assume now that condition (1.ii) is *true* at $C^-$. Since $Tail(C^-)$ points to a node $nd$ such that $nd \rightarrow \mathit{completed} = \mathit{false}$ and $nd \rightarrow \mathit{wait} = \mathit{true}$, and $\mathit{node}_j(C) = Tail(C^-)$, it follows that $p_j$ is not a combiner at $C$; notice that $p_j$ could not be a combiner at $C^-$ since by induction hypothesis (Claim 2.i), the combiner is poised to execute one of the lines 6- 18 at $C^-$ (whereas $p_j$ is poised to execute line 5 at $C^-$).

  Recall that $\mathit{node}_j(C^-) \rightarrow \mathit{wait} = \mathit{true}$ and $\mathit{node}_j(C^-) \rightarrow \mathit{completed} = \mathit{false}$. Since $Tail(C) = \mathit{node}_j(C^-)$, it follows that $Tail(C) \rightarrow \mathit{wait} = \mathit{true}$ and $Tail(C) \rightarrow \mathit{completed} = \mathit{false}$. This completes the proof of Claim 1.

The rest of the claims in each case, hold by the induction hypothesis.

4. If $s$ is the execution of any of the lines 6-8, the claim holds trivially by induction hypothesis and Lemma 6.1.

5. If $s$ is the execution of line 9. We distinguish the following two cases.

   - Assume first that $p_j$ is the combiner at $C^-$. It suffices to argue that $p_j$ is not poised to execute line 10 at $C$ (as needed by Claim 2.i). By definition, there is some configuration $C''$ that precedes $C$ (and occurs during the course of the execution of the current instance of CC-Synch by $p_j$) at which $\mathit{node}_j(C'') \rightarrow \mathit{wait} = \mathit{false}$ and $\mathit{node}_j(C'') \rightarrow \mathit{completed} = \mathit{false}$. By the pseudocode, the *wait* or *completed* field of $\mathit{node}_j$ can change only if some thread $p_m$ executes one of the lines 1, 3, 16, 17, or 19 after $C''$. Lemma 6.1 implies that the *wait* or *completed* field of $\mathit{node}_j$ cannot change by threads other than $p_j$ that execute line 1 or line 3. Moreover, the induction hypothesis (Claims 1 and 2.ii) implies that no thread other than $p_j$ can change the *wait* or *completed* fields of $node_j$ by executing lines 16, 17, or 19. Thus, no thread other than $p_j$ can change these fields of $node_j$ from $C''$ to $C^-$. It follows that $p_j$ evaluates the condition of line 9 to *false* and therefore, it is not poised to execute line 10 at $C$ (as needed by 2.i).

- Assume now that $p_j$ is not a combiner at $C^-$. Let $C_5$ be the configuration at which $p_j$ executed its `Swap` instruction of line 5. Obviously, $C_5$ precedes $C^-$. This and the fact that $p_j$ is not a combiner at $C^-$ implies that $p_j$ was not a combiner at configuration $C_5$. Since $node_j(C_5) = Tail(C_5^-)$, it follows that one of the *wait*, *completed* fields pointed to by $Tail(C_5^-)$ was equal to *true*. Since $C_5$ precedes $C$, the induction hypothesis (Claim 1) implies that $Tail(C_5^-) \rightarrow wait = true$ and $Tail(C_5^-) \rightarrow completed = false$. Thus, $node_j(C_5) \rightarrow wait = true$. Since $p_j$ executes line 9 at $C^-$, it must be that there is some configuration $C_8$ preceding $C^-$ at which $p_j$ evaluates the condition of the `while` statement of line 8 to *true*. It follows that there must be a configuration between $C_5$ and $C_8$, at which $node_j \rightarrow wait$ becomes equal to *false*. By the pseudocode and by Lemma 6.1, it follows that the only way for this to happen, is if a thread executes line 17 just before that configuration (with its $tmpNode$ variable equal to $node_j$). By the induction hypothesis (Claims 1 and 2.ii), this thread (let it be $p_m \neq p_j$) should be a combiner. By the pseudocode, it follows that $p_m$ executes line 16 just before executing line 17; let $C''$ be the configuration at which this occurs. The execution of line 16 by $p_m$ results in changing $node_j \rightarrow completed$ to *true*. By the induction hypothesis (Claim 2.iii), $p_j$ should execute one of the lines 6-10 at $C''$. By the pseudocode, it follows that $p_j$ does not change $node_j \rightarrow completed$ to *false* from $C''$ to $C^-$. Also, Lemma 6.1 imply that no thread other than $p_j$ may change $node_j \rightarrow completed$ to *false* between $C''$ and $C^-$ by executing line 3. Thus, $node_j \rightarrow completed = true$ at $C^-$. Therefore, $p_j$ evaluates the condition of the `if` statement of line 9 to *true* and is poised to execute line 10 (and return) at its next step.

The rest of the claims in each case, hold by the induction hypothesis.

6. $s$ is the execution of line 11.

   By the induction hypothesis (Claims 1 and 2), it follows that it is only the combiner $p_i$ that can be poised to execute line 11 at $C^-$; thus, $p_j = p_i$ and $p_j$ has not executed any iteration of the while loop (lines 12-18) yet. By the pseudocode (line 11) it follows that the node assigned to *tmpNode* is equal to $node_j$. Thus, Claim 2.iii holds. The rest of the claims hold by the induction hypothesis.

7. $s$ is the execution of one of the lines 12-15. In this case the claim holds by induction hypothesis.

8. $s$ is the execution of line 16. It is enough to prove that Claim 3 holds at $C$. Suppose that this is the $k$th time, $k \geq 1$, that $p_i$ executes line 16 during $A_i$ and assume that line 16 has been executed $m$ times in total until $C$. By the induction hypothesis (Claim 2.iii) there is a thread $p_k$ such that $node_k(C^-) = nd'_k$, and either $p_k = p_i$ or $p_k$ is executing one of the lines 6-10 at $C^-$.

   - Assume first that $k = 1$. Let $C_5$ be the configuration resulted when $p_i$ executes line 5. By the induction hypothesis (Claim 2.i), $p_i$ cannot be a combiner before $C_5$.

     – Assume first that $p_i$ is not a combiner at $C_5$.
     Since $p_i$ is active at all configurations between $C_5$ and $C$, by the induction hypothesis (Claim 1), it follows that there is always some combiner in the system between $C_5^-$ and $C$. Assume that $p_i$ became the combiner at some configuration $C'$ preceding $C$, and let $p_j$ be the thread that was the combiner at $C'^-$; let $A_j$ be the instance of CC-Synch executed by $p_j$ at $C'^-$. Since $p_i$ becomes the combiner after $p_j$ and $p_i$ is executing line 16 for the last time at $C$, it follows that the $(m-1)$st time that lines 15-17 were executed was the last time that $p_j$ executed those lines during $A_j$. Suppose that $p_j$ visits $k', k' \geq 1$ nodes. Let $nd'_{k'}$ be the last node visited by $A_j$. The induction hypothesis (Claim 2.iv) implies that $nd'_{k'} = node_i(C')$ (since $p_i$ is the unique combiner in the system right after $p_j$). The pseudocode (lines 11-14) imply that $nd'_1 = node_j(C'^-) \neq node_i(C'^-)$; thus, $k' > 1$. By the pseudocode, $p_i$ becomes a combiner

when $p_j$ executes line 19 (i.e. at configuration $C'$). By the pseudocode, $node_i(C') = nd'_{k'} = nd'_{k'-1} \rightarrow next$. We distinguish the following two cases. In case that $p_{k'-1} = p_j$, the induction hypothesis (Claim 3), implies that $nd'_{k'-1} = nd_{m-1}$. Pseudocode (lines 14, 18 and 19) implies that $nd'_{k'} = nd'_{k'-1} \rightarrow next = nd_{m-1} \rightarrow next$ and $nd'_{k'} = node_i(C) \neq \perp$, it follows that $nd_{m-1} \rightarrow next \neq \perp$. Thus, Lemma 6.4 implies that $node_i(C) = nd'_1 = nd_{m-1} \rightarrow next = nd_m$, as needed (by Claim 3). In case that $p_{k'-1} \neq p_j$, by the induction hypothesis (Claims 2.iii), there is some thread $p_{k'-1}$ such that at each configuration $\tilde{C}$ in $\beta_{k'-1}$, $p_{k'-1}$ was executing one of the lines 6-10 at $\tilde{C}$ and $node_{k'-1}(\tilde{C}) = nd'_{k'-1}$; specifically, $p_{k'-1}$ was active at configuration $C_{14}$ at which line 14 was executed by $p_j$ during $\beta_{k'-1}$. Also, by the induction hypothesis (Claim 3), it follows that $nd'_{k'-1} = nd_{m-1}$. Since $nd'_{k'} = nd'_{k'-1} \rightarrow next = nd_{m-1} \rightarrow next$ and $nd'_{k'} = node_i \neq \perp$, it follows that $nd_{m-1} \rightarrow next \neq \perp$. Thus, Lemma 6.4 implies that $node_i(C) = nd'_1 = nd_{m-1} \rightarrow next = nd_m$, as needed (by Claim 3).

- Assume now that $p_i$ is a combiner at $C_5$. Thus, $node_i(C_5) \rightarrow wait = \mathit{false}$ and $node_i(C_5) \rightarrow completed = \mathit{false}$. Since $node_i(C_5) = Tail(C_5^-)$, it follows that $Tail(C_5^-) \rightarrow wait = \mathit{false}$ and $Tail(C_5^-) \rightarrow completed = \mathit{false}$. By the induction hypothesis (Claim 1), it follows that there is no combiner at $C_5^-$ and no active thread is executing any of the lines 11-19 at $C_5^-$. Let $C'$ be the last configuration preceding $C_5^-$ at which there was a combiner $p_c$ in the system; let $A_c$ be the instance of CC-Synch executed by $p_c$ at $C'$. Notice that $p_c$ has executed line 19 in $A_c$ just before configuration $C'$. By the induction hypothesis (Claim 2.iv), it follows that there is no combiner in the system at $C'$. By definition of $p_c$, there is no combiner in the system between $C'$ and $C_5^-$. We first prove that there no `Swap` operation is executed between $C'$ and $C_5^-$. Assume by the way of contradiction that at least one `Swap` is executed between $C'$ and $C_5^-$. Let $s$ be the first such `Swap` and let $C''$ be the configuration just after the execution of $s$. Assume that $s$ is executed by some thread $p_s$. Since there is no combiner between $C'$ and $C_5^-$, induction hypothesis (Claim 1.i) implies that $Tail(C'^-) \rightarrow completed{=}\mathit{false}$ and $Tail(C'^-) \rightarrow wait{=}\mathit{false}$. The pseudocode (line 5) and the definition of combiner implies that $node_s(C') \rightarrow$ *completed=false* and $node_s(C') \rightarrow$ *wait=false*. Therefore, $p_s$ is a combiner at $C''$, which is a contradiction. Thus, the `Swap` instruction executed by $p_i$ at $C_5^-$ is the first `Swap` instruction executed between $C'$ and $C_5^-$. Assume that $p_c$ visited $k'$ nodes in $A_c$; and denote by $nd'_{k'-1}$ and $nd'_{k'}$ the last two nodes visited by $p_c$ in $A_c$. Induction hypothesis (Claim 3) implies that $nd'_{k'-1} = nd_{m-1}$. It is enough to prove that $nd'_{k'} = Tail(C'_5) = nd'_1 = nd'_m$. Induction hypothesis (Claim 1.i) implies that $Tail(C'^-) \rightarrow$ *completed =false* and $Tail(C'^-) \rightarrow$ *wait =true*. Induction hypothesis (Claim 1.ii) also implies that $Tail(C') \rightarrow$ *completed =false* and $Tail(C') \rightarrow$ *wait =false*. Since $p_c$ executes line 19 at node $nd'_{k'}$ at $C'$, it follows that $nd'_{k'} = Tail(C')$. Since there is no `Swap` executed between $C'$ and $C_5$ other than that of $p_i$ at $C_5$, it follows that $Tail(C') = Tail(C_5)$. Lemma 6.1 implies that $Tail(C_5) = nd'_1$. Thus, $nd'_{k'} = Tail(C_5) = nd'_1 = nd_m$, as needed.

- Assume now that $k > 1$. By the pseudocode, $nd'_{k-1} \rightarrow next$ (in $A_i$ instance) cannot be equal to $\perp$ since otherwise the $k$th (and the $(k-1)$th) iteration of the `while` loop would not be executed by $p_i$. By the induction hypothesis (Claims 2.iii), either $p_{k-1} = p_i$ or there is some thread $p_{k-1} \neq p_i$ such that at each configuration $C'$ in $\beta_{k-1}$, $p_{k-1}$ was executing one of the lines 6-10 at $C'$. By the induction hypothesis (Claim 3), it follows that $nd'_{k-1} = nd_{m-1}$. By the pseudocode, it follows that $nd'_k = nd'_{k-1} \rightarrow next = nd_{m-1} \rightarrow next$. Since $nd'_k \neq \perp$, it follows that $nd_{m-1} \rightarrow next \neq \perp$. Thus, Lemma 6.4 implies that $nd'_k = nd_{m-1} \rightarrow next = nd_m$, as needed (by Claim 3).

9. $s$ is the execution of line 17.

Assume that line 17 is executed for the $k$th time, $k \geq 1$. Recall that it is only the combiner that can be poised to execute line 16 at $C^-$. By the induction hypothesis (Claim 2.iii) there is a thread $p_k$ such that $node_k(C^-) = nd'_k$, and either $p_k = p_i$ or $p_k$ is executing one of the lines 6-10 at $C^-$. Lemma 6.1 implies that $Tail(C^-) \neq node_k(C^-)$. Since $Tail(C) = Tail(C^-)$ and $node_k(C) = node_k(C^-)$, it follows that $Tail(C) \neq node_k(C^-)$. Since $s$ changes either the *wait* field of $node_k(C^-)$, Claim 1 holds by induction hypothesis. The rest of the claims hold by induction hypothesis.

10. $s$ is the execution of line 18.

It is enough to argue that Claim 2.iii holds after the execution of $s$. The rest of claims hold by induction hypothesis. Assume that the execution of $s$ by $p_i$ identifies the $k$th node visited by $p_i$, $k > 1$ (notice that the first node visited by $p_i$ is identified by executing line 11 and not line 18; thus, $k > 1$). If $nd'_k = Tail(C)$, Claim 2.iii holds (by induction hypothesis for each $l < k$). Thus, assume that $nd'_k \neq Tail(C)$. Denote by $S_1$ the Swap operation executed by $p_i$ in $req$ and denote by $S_k$ the Swap operation executed by $p_k$. We first prove that $S_1$ precedes $S_k$. By the pseudocode, $nd'_{k-1} \rightarrow next = nd'_k$ cannot be equal to $\perp$ since otherwise the $(k-1)$th iteration of the while, where the $k$th node to be visited by $p_i$ is identified (this occurs when $p_i$ executes line 18 of that iteration), would not be executed. By the induction hypothesis (Claim 2.iii), either $p_{k-1} = p_i$ or there is some thread $p_{k-1}$ such that at each configuration $C'$ in $\beta_{k-1}$, $p_{k-1}$ was executing one of the lines 6-10 at $C'$ and $node_{k-1}(C') = nd_{k-1}$. In case that $p_{k-1} = p_i$, the pseudocode implies that $nd'_1 \rightarrow next = nd'_k$. Lemma 6.4 implies that the Swap operation $S_1$ executed by $p_i$ precedes the execution of the Swap operation $S_k$ by $p_k$. In case that $p_{k-1} \neq p_i$, (Claim 2.iii) implies that $p_{k-1}$ was active at configuration $C_{14}$ at which line 14 was executed by $p_i$ during $\beta_{k-1}$. The pseudocode implies that $nd'_{k-1} \rightarrow next = node_{k-1}(C_{14}) \rightarrow next$. Since we have assumed that $nd'_k \neq Tail(C)$, it follows that $node_{k-1}(C_{14}) \rightarrow next \neq \perp$. Thus, Lemma 6.4 implies that there is a thread $p_k$ that has executed line 5 before $C_{14}$ such that $node_k(C) = nd'_k$; moreover, Lemma 6.4 implies that the Swap operation $S_{k-1}$ executed by $p_{k-1}$ precedes the execution of the Swap operation $S_k$ by $p_k$. By the pseudocode, $p_i$ executes lines 15-17 for itself during the first iteration of the while loop of lines 12-18, i.e. before executing it for $p_{k-1}$. Since $p_i$ executes lines 15-17 later on for $p_{k-1}$, the induction hypothesis (Claims 2.iii and 3) implies that $S_1$ has occurred before $S_{k-1}$. It follows that $S_1$ has occurred before $S_k$.

We continue to prove that $p_k$ is active at $C^-$. Assume, by the way of contradiction, that the instance $A_k$ of CC-Synch executed by $p_k$ is not active at $C^-$. Recall that $p_k$ executed $S_k$ after $S_1$. Thus, $p_i$ was active while executing one of the lines 6-18 when $S_k$ was executed; let $C_k$ be the configuration just after the execution of $S_k$.

Apparently, $C_k$ precedes $C$. Since $p_i$ is active between $C_1$ and $C$, by the induction hypothesis (Claim 1), it follows that there is always some combiner in the system in all configurations between $C_1$ (which results by applying $S_1$) and $C$. Thus, there is some combiner in all configurations between $C_k^-$ and $C$. By induction hypothesis (Claim 1.ii), $Tail(C_k^-) \rightarrow wait = true$ and $Tail(C_k^-) \rightarrow completed = false$. By the pseudocode, $node_k(C_k) = Tail(C_k^-)$, so $node_k(C_k) \rightarrow completed = false$. By the pseudocode, the *completed* field of $node_k$ must be equal to *true* when $p_k$ terminates. By the pseudocode, this can happen only if there is some thread $p_h$ that executes lines 15-17 with $tmpNode = node_k$ at some configuration before $C$. Suppose that lines 15-17 have been executed $h'$ times in total until the configuration that $p_h$ executed line 17 with $tmpNode = node_k$. Since $S_1$ is performed before $S_k$, the induction hypothesis (Claim 3) implies that lines 17-19 have been executed for $p_i$ before being executed for $p_k$ (assume that this has happened the $h''$th time that lines 15-17 were executed). By the induction hypothesis (Claim 3), $node_i = nd_{h''}$. However, by the pseudocode, it follows that lines 15-17 are executed for $node_i$ by $p_i$ during the execution of the first iteration of the while loop of lines 12-18; let this be the $h$th time that lines 15-17 are executed. Since $p_i$ has not visited $node_k$ before $C$, it follows that $h > h''$, which contradicts the induction hypothesis (Claim 3).

Thus, $p_k$ is still active at $C^-$. By the induction hypothesis (Claim 2.ii), no thread other than the combiner $p_i$ executes lines 11-20 at $C^-$. Thus, $p_i$ executes one of the lines 6-10 at $C^-$. Since it is thread $p_i$ that executes $s$, $p_k$ executes one of the lines 6-10 at $C$. This completes the proof of Claim 2.iii.

11. $s$ is the execution of line 19. Recall that it is only the combiner that can be poised to execute this line at $C^-$. Assume that $p_i$ visits $k$ nodes during the execution of $A_i$.

    - If $nd'_k \neq Tail(C^-)$, the induction hypothesis (Claim 2.iii) implies that $nd'_k = node_k(C^-)$ for some thread $p_k$ that is poised to execute one of the lines 6-10 at $C^-$. By the induction hypothesis (Claim 3), $node_k$ is visited for the first time since $A_k$ was initialized. This and Lemma 6.1 imply that $node_k(C^-) \rightarrow completed = false$. Since $s$ changes $node_k(C^-) \rightarrow wait$ to *false* it follows that $p_k$ is a combiner at $C$. Since $p_i$ was the unique combiner in the system at $C^-$

and by definition, it is not a combiner anymore after the execution of line 19, it follows that the unique combiner in the system at $C$ is $p_k$, as needed by Claim 2.iv. Moreover, Lemma 6.1 implies that $Tail(C^-) \neq node_k(C^-)$, so $Tail(C) = Tail(C^-)$. Since $p_i$ is a combiner at $C^-$, the induction hypothesis (Claim 1) implies that $Tail(C^-) \rightarrow wait = true$ and $Tail(C^-) \rightarrow completed = false$. Thus, $Tail(C) \rightarrow wait = true$ and $Tail(C) \rightarrow completed = false$. So, Claim 1 follows. The rest of claims hold by induction hypothesis.

- We first prove that $Tail(C) \rightarrow completed = false$ and $Tail(C) \rightarrow completed = false$. Assume now that $nd'_k = Tail(C^-)$. By the induction hypothesis (Claim 3), $node_k$ is visited for the first time since $A_k$ was initialized. This and Lemma 6.1 imply that $node_k(C^-) \rightarrow completed = Tail(C^-) \rightarrow completed = false$. The pseudocode (line 19) implies that $tmpNode_k \rightarrow wait = nd'_k \rightarrow wait = Tail(C^-) \rightarrow wait = false$ at $C$. We now prove that no thread is poised to execute any of lines 11-19 at $C$. Induction hypothesis (Claim 1.i), there is no thread other than $p_i$ poised to execute any of lines 11-19 at $C^-$. Since $p_i$ executes line 19 just before $C$, it follows that no thread is poised to execute any of lines 11-19 at $C$. We finally prove that there is no combiner at $C$. Assume, by the way of contradiction that there is a combiner $p_c$ at $C$ that executes some instance $A_c$ of CC-Synch. By combiner's definition, it follows that there is a configuration $C'$ preceding $C$ such that $node_c(C') \rightarrow wait =false$ and $node_c(C') \rightarrow completed =false$. Lemma 6.1 implies that $node_c(C) \neq Tail(C)$. It follows that $C'$ precedes $C^-$. Thus, $p_c$ is also a combiner at $C^-$. Induction hypothesis (Claim 1.ii) implies that the only combiner at $C^-$ is $p_i$, which is a contradiction. So, Claim 1 follows.

The rest of the claims in each case hold by the induction hypothesis. ∎

Let $nd_i$ be the node of the list that is assigned to $p_i$ for $req_i$. Thread $p_i$ completes the execution of CC-Synch for $req_i$ either on line 10 or on line 20. Assume first, that $p_i$ returns on line 10. Lemma 6.5 (Claim 2.iii) implies that a combiner thread $p_j$ has served $req_i$ before the execution of line 10 by $p_i$. Therefore, $p_j$ has executed line 15 for $nd_i$ at some iteration $l > 1$ of its while loop (lines 12-18). Request $req_i$ is linearized just before the execution of this instance of line 15 by $p_j$. Assume now that $p_i$ returns on line 20.

Lemma 6.5 (Claim 2.iii) implies that $p_i$ serves its request on its own when it executes line 15 at the first iteration of its while loop (lines 12-18). In this case, $req_i$ is linearized just before the execution of line 15 of the first iteration of $p_i$'As while loop. Obviously, in both cases the linearization point of $req_i$ is within its execution interval. Consistency is immediately implied by Claims 2.iii and 3 of Lemma 6.5. Thus, the following theorem holds.

**Theorem 6.1.** CC-Synch *is a linearizable synchronization algorithm.*

## 6.2 H-Synch: A hierarchical synchronization algorithm based on CC-Synch

We now discuss how we can modify CC-Synch to get an hierarchical approach of it, called H-Synch. We consider a system of $m$ processors which are partitioned into $C$ clusters; each cluster consists of $m/C$ processors. In such a system, communication among the processors of the same cluster is performed much faster than among processors residing in different clusters. A characteristic example of such a system is the Niagara 2 machine, in which we have executed some of the experiments in Section 6.4.

In H-Synch, the threads use $C$ instances of CC-Synch one per cluster (Algorithm 18). Each instance of CC-Synch is used, as described in Section 6.1, to identify at each point in time, the combiner thread of each cluster and the list of announced requests of the threads that are executed at processors of the cluster. In addition to the $C$ instances of CC-Synch, a queue lock $L$ [23, 47] is used; $L$ is accessed only by the combiner threads of the clusters. The CLH lock [23, 47] is a good choice for implementing $L$ in systems where the intra-cluster communication is achieved with a cache coherent (CC) protocol, whereas the MCS lock [49] is expected to be a better choice in other systems.

Whenever a thread $q$ has a newly activated request, it calls H-Synch. If $q$ does not become the combiner of its cluster, it waits until its request has been served by a combiner of the cluster. Otherwise, before $q$ starts serving requests, it executes an `acquire` operation on $L$ (line 13) in order to ensure that it is the only combiner (among those of the different clusters) that has access to the shared data. A combiner $q$ serves only requests initiated by threads that are running on its local cluster. By doing so, intra-cluster communication is kept low. After finishing its work as a combiner, $q$ releases $L$ (line 21), so that a combiner of some other cluster can acquire $L$ and have access to the shared data. In cases that the communication between clusters is performed through a more complex interconnection network (for instance, one that has an hierarchical structure), H-Synch can be easily modified to exploit the characteristics of the communication hierarchy by using more levels of queue locks (one queue lock per communication level).

```
     // Assume that thread $p_i$ runs at some processor of cluster $c_i$
     shared Node *Tail[1..C] = { $\langle \bot, \bot$, false, false, null$\rangle$ };

     // L is a CLH [23, 47] or MCS [49] queue lock
     shared QueueLock L;

     // The following variable is private to each thread $p_i$; it is a pointer to a
     // struct of type Node with initial value $\langle \bot, \bot$, false, false, null$\rangle$
     private Node *$node_i$ = {$\langle$null, $\bot, \bot$, false, false$\rangle$};

     RetVal H-Synch(Request req) {                //Pseudocode for thread $p_i$
        Node *nd, *cur, *next_node, *ndnext;
        int counter = 0;
1       next_node = $node_i$;                       // $p_i$ uses a (possibly recycled) node
2       next_node $\rightarrow$ next = $\bot$;                  // $p_i$ initializes the fields of this node
3       next_node $\rightarrow$ locked = true;
4       next_node $\rightarrow$ completed = false;
5       cur = Swap(Tail[$c_i$], next_node);
6       cur $\rightarrow$ req = req;                        // $p_i$ announces its request
7       cur $\rightarrow$ next = next_node;
8       $node_i$ = cur;                              // reuse this node in future request
9       while (cur $\rightarrow$ locked == true)           // $p_i$ spins until its request
           nop;                                  // is applied or until is unlocked
10      if (cur $\rightarrow$ completed == true)          // if $p_i$'s request is already applied
11         return cur $\rightarrow$ ret;                    // $p_i$ returns its return value
12      nd = cur;
13      lock(L);                                 // acquire the lock
14      while (nd $\rightarrow$ next $\neq$ null AND counter $< h$) {
15         counter = counter + 1;
16         ndnext = nd$\rightarrow$next;
17         apply nd$\rightarrow$req to object's state and store the return value to nd$\rightarrow$ret;
18         nd $\rightarrow$ completed = true              // nd's request is applied
19         nd $\rightarrow$ locked = false;               // $p_i$ unlocks the spinning thread
20         nd = ndnext;                          // and proceeds to the next node
        }
21      unlock(L);                               // release the lock
22      nd $\rightarrow$ locked = false;                   // unlocks the next node's owner
23      return cur $\rightarrow$ ret;                       // thread returns
     }
```

Algorithm 18: Pseudocode for H-Synch.

H-Synch ensures that requests initiated by threads of the same cluster are served in FIFO order and that the combiners acquire the global lock in FIFO order but, apparently, it does not globally ensure the FIFO property.

## 6.3 DSM-Synch: An efficient synchronization algorithm for the DSM model

In this section, we present DSM-Synch (Algorithm 19) which performs $O(hd)$ RMRs in the DSM model, where $h$ is an upper bound on the number of requests that a combiner may serve and $d$ is the maximum number of RMRs required to serve a single request. The space overhead of DSM-Synch is $O(n)$.

### 6.3.1 Algorithm description

DSM-Synch maintains a list of announced requests which is updated in a manner similar to that of CC-Synch (and which also implements the lock). In contrast to CC-Synch, the list is initially empty and its last node is a valid (not dummy) node. Each thread $p$ maintains two list nodes; $p$ announces each request it wants to perform in one of these nodes, it inserts this node at the end of the list by using Swap, and if its record is not the first in the list, it performs spinning on the *wait* field of its node. If $p$'s node is the first in the list, $p$ becomes the combiner and serves up to $h$ requests of the list in FIFO order. A thread that appends a node in the list updates the next field of the previous node to point to the inserted node. The combiner serves requests recorded in list nodes up to the second last element of the list (see condition of the if statement of line 19). It does so to avoid arriving to a node where its next field has not yet been updated although this node is not the last node in the list any more. Thus, a combiner will execute lines 22-25 only if its node is the only node in the list. In this way, a combiner performs a bounded number of RMRs (whereas spinning on the last node of the list on line 25 would cause an unbounded number of RMRs).

We explain now why it is not enough to have each thread $p_i$ using just one node. Let's assume that $p_i$ has a single node that it reuses each time it initiates a new request. Let

$q$ be a combiner thread that serves $p_i$'s request. Assume that there is a thread $p_j$ whose assigned node is the next node of $p_i$'s node in the list. Assume also that there is no other active request and thus $p_j$'s node is the last node in the list. After serving $p_i$'s request, $q$ sees that less than two nodes are left in the list and stops the execution of its while loop (lines 19-20). Suppose that $q$ stalls before executing line 26 of the pseudocode (pointer $nd$ points to the node assigned to $p_i$). Assume now that $p_i$ wants to immediately apply a new operation. Thus, $p_i$ initializes its node again and inserts it at the end of the list by executing a Swap instruction on $Tail$. Then, $q$ continues by executing line 26 of the pseudocode and makes an invalid memory reference.

### 6.3.2 Time and Space complexity

DSM-Synch performs $O(hd)$ RMRs when executing the while loop of line 14. The only extra piece of code that may cause DSM-Synch to perform more RMRs is that consisting of lines 22-25. However, as explained above a thread $p$ executes these code lines only when the node it inserts is the single node of the list. So, if these lines are executed, $p$'s local variable called $nd$ is equal to $node_i$, and therefore spinning on $p \rightarrow next$ on line 25 is local. The space overhead of DSM-Synch is $O(n)$.

### 6.3.3 Required memory barriers

Similarly to CC-Synch, memory barriers may need to be inserted in the code of DSM-Synch to ensure its correct execution in architectures that implement either the TSO (Total Store Order) or the PO (Process Order) consistency model. Remind that both of TSO an PO consistency models do not reorder two read operations, and the same holds for two store operations [48]. Remind also that a read can be reordered with an older store only in case that the read and the store instructions access different memory locations [48]. Thus, for the correct execution of lines 3-6 and lines 17-18, no store barrier is needed. Similarly, no load barrier is needed for lines 9 and 10. A store memory barrier is inserted just before the return instruction of line 12.

```
struct LockNode {
    Request req;
    RetVal ret;
    boolean wait, completed;
    LockNode *next;
};

shared LockNode *Tail = null;
// The following variables are private to each thread $p_i$
private LockNode $MyNodes_i$[0..1] = {⟨⊥, ⊥, false, false, null ⟩};
private int $toggle_i$ = 0;

RetVal DSM-Synch(Request req) {                // pseudocode for thread $p_i$
    LockNode *nd, *$node_i$, *MyPred;
    int counter = 0;

    $toggle_i$ = 1 - $toggle_i$;                 // $p_i$ toggles its toggle variable
    $node_i$= $MyNodes_i$[$toggle_i$];             // $p_i$ chooses to use one of its nodes
    $node_i$→wait = true;                         // $p_i$ initializes the node
    $node_i$→completed = false;
    $node_i$→next = null;
    $node_i$→req = req;                           // $p_i$ announces its request
    MyPred = Swap(Tail, $node_i$);                 // $p_i$ inserts $node_i$ in the list
    if (MyPred ≠ null) {                        // if a node already exists in the list
        MyPred→next = $node_i$;                   // fix next of previous node
        while($node_i$→wait == true)              // perform spinning
            nop;
        if($node_i$→completed == true)            // if combiner has applied $p_i$'s req
            return $node_i$→ret;
    }
    nd = $node_i$;
    while(true) {                               // $p_i$ is the combiner
        counter++;
        apply nd→req and store the return value to nd→ret;
        nd→completed = true;                    // announce that nd→ req is applied
        nd→wait = false;                        // unlock the spinning thread
        if (nd→next == null or nd→next→next == null or counter $\geq h$)
            break;                              // $p_i$ helped $h$ threads or fewer than 2 nodes are in list
        nd = nd→next;                           // proceed to the next node
    }
    if (nd→next == null) {                      // $p_i$'s req is the single record in list
        if(CAS(Tail, nd, null)==true)           // try to set Tail to null
            return $node_i$→ret;
        while (nd→next == null)                 // some thread is appending a node
            nop;                                // wait until it finishes its operation
    }
    nd→next→wait = false;                       // A new combiner is identified
    return $node_i$→ret;
}
```

Algorithm 19: Pseudocode for DSM-Synch.

| Notation | Description |
|---|---|
| $\alpha$ | Any execution of DSM-Synch |
| $C$ | Any configuration in $\alpha$ |
| $C_0$ | The initial configuration of $\alpha$ |
| $C^-$ | The configuration just preceding $C$ |
| $Tail(C)$ | The value of base object $Tail$ at configuration $C$ |
| $p_i$ | The thread which its id is equal to $i$, $i \in \{1, \ldots, n\}$ |
| $node_i(C)$ | The value of $node_i$ at configuration $C$ |
| $m$ | The number of Swap operations executed in $\alpha$ |
| $S_l$ | The $l$th Swap in execution $\alpha$ |
| $A_l$ | The instance of DSM-Synch that executes $S_l$ |
| $p_{i_l}$ | The thread that executes $S_l$ operation |
| $C_l$ | The configuration just after $S_l$ |
| $nd_l$ | The value returned by $S_l$ |

Table 6.2: Notation used in the proof of DSM-Synch.

### 6.3.4 Correctness proof

In this section, we present the correctness proof of DSM-Synch. Let $\alpha$ be any execution. Consider any configuration $C$ in $\alpha$. Let $Tail(C)$ be the value of $Tail$ at $C$. For each $i$, $1 \leq i \leq n$, denote by $node_i(C)$ the value of variable $node_i$ at $C$. Denote by $C^-$ the configuration just preceding $C$ and let $C_0$ be the initial configuration. The notation of this proof is summarized in Table 6.2.

Let $m \geq 0$, be the number of Swap operations that are executed in $\alpha$[†]. Denote by $S_l$, $0 \leq l \leq m$, the $l$th Swap operation executed in $\alpha$, let $A_l$ be the instance of DSM-Synch that executes $S_l$, let $p_{i_l}$ be the thread that executes $A_l$, and denote by $nd_l$ the return value of $S_l$. Let $C_l$ be the configuration just after the execution of $S_l$ and let $Q_0 = C_0$ be the initial configuration.

Consider a thread $p_i$ that executes an instance $A$ of DSM-Synch at some configuration $C$. In case that the Swap instruction of line 7 in $A$ returns $\perp$, denote by $C_f$ the first configuration just after the execution of line 7; in case that there is at least one configuration $C'$ in $A$ such that either $node_i(C') \rightarrow wait = false$ and $node_i(C') \rightarrow completed = false$, denote by $C_f$ the first of these configurations. We say that $p_i$ is a *combiner* at $C$ if $C_f$ is well defined and $C$ follows $C_f$; $p_i$ is a combiner from $C_f$ until it executes either line 24 or line 27 in $A$ (we show below that a combiner always returns either on line 24 or line 27).

We say that an instance $A$ of DSM-Synch visits a node $nd$, if $A$ executes line 13 or 21 and sets its $tmpNode$ variable to point to $nd$; if $A$ is executed by thread $p_i$, we

[†] We remark that $m$ may be $\infty$ if $\alpha$ is an infinite execution.

sometimes say that $p_i$ visits $nd$ (if $A$ visits $nd$). If $A$ visits a node $nd$, then there is an execution fragment starting from the configuration at which $A$ executes line 13 (or 21) to set $tmpNode$ to point to $nd$ until the configuration that $A$ executes line 21 for the next time (or until $A$ executes line 26 if this was the last time that line 21 was executed by $A$ or if line 21 was not executed by $A$).

The following observation is an immediate consequence of the pseudocode (lines 7,8 and 9).

**Observation 6.1.** *Let $A_j$ be any instance of* CC-Synch *executed by some thread $p_j$ and $S_l, l > 1$ be the* `Swap` *instruction executed by $A_j$. Let $node_j$ be the value written by $A_j$ on some base object $node_x \rightarrow next$ while executing line 9 of the pseudocode. $S_{l-1}$ is executed by some thread $p_{j'}$ that writes $node_x$ to $Tail$.*

The pseudocode (lines 1, 2 and 7) implies the following observation.

**Observation 6.2.** *Let $A_j$ and $A'_j$ be two consecutive instances executed by some thread $p_j$; let $node_j$ and $node'_j$ be the values written while line 7 is executed by $A_j$ and $A'_j$ respectively. It holds that $node_j \neq node'_j$.*

**Lemma 6.6.** *In each configuration $C$,*

1. *exactly one of the following conditions (i or ii) holds:*

   **(i)** *$Tail(C) = \bot$, there is no combiner at $C$ and there is no thread poised to execute any of lines 13-23 and 25-26.*

   **(ii)** *$Tail(C) \neq \bot$ and there is exactly one combiner at $C$.*

2. *if there is a combiner $p_i$ at $C$, the following claims hold:*

   **(i)** *$p_i$ is poised to execute one of the lines 8-23 or one of the lines 25-26 at $C$ and it is not poised to execute line 12 at $C$.*

   **(ii)** *No thread other than $p_i$ executes lines 13-23 and lines 25-26 at $C$.*

   **(iii)** *Suppose that $p_i$ is poised to execute one of the lines 13-26 at $C$, let $k$ be the number of nodes that have been visited by $p_i$ until $C$, denote by $nd'_l$, $1 \leq l \leq k$, the $l$th such node, and let $\beta_l$ be the execution fragment at which $p_i$ is visiting $nd'_l$; then, for each $l$, $1 \leq l \leq k$ and for each configuration $C'$ in $\beta_l$, if $nd'_l \neq Tail(C')$*

*there is one active thread $p_l$ such that $node_l(C') = nd'_l$, and either $p_l = p_i$ or $p_l$ executes one of the lines 8-12.*

**(iv)** *assume that $C$ is the configuration just after $p_i$ has executed line 26 of the pseudocode; if $nd'_k \neq Tail(C^-)$, then $p_k$ is the unique combiner in the system at $C$ otherwise there is no combiner in the system at $C$.*

3. *if lines 17-18 have been executed $m$ times in total until $C$, then for each $l$, $1 \leq l \leq m$, lines 16-18 for the $l$th time were executed by a combiner that had its tmpNode variable equal to $nd_l$.*

4. *let $p_j$ be the thread that executes the $S_m, m > 0$ instruction; for each $l$, $0 \leq l < m$, and for each configuration $C$ such that $C$ follows $C_l$ and $A_l$ is active at $C$, it holds that either $nd_l \rightarrow next = \perp$ or $nd_l \rightarrow next = nd_{l+1}$ at $C$; if $m$ is finite, at each configuration $C$ following $C_m$, it holds that either $nd_m \rightarrow next = \perp$ or $nd_m \rightarrow next = Tail(C)$.*

*Proof.* We prove the claim by induction on $C$.

**Base Case ($C = C_0$).** No thread is active at $C_0$ so there is no combiner at $C_0$. Moreover, $Tail = \perp$ at $C_0$. Thus, the Claim 1 holds. Claim 2 obviously holds, since there is not any combiner ar $C_0$. Furthermore, Claim 3 holds, since no thread has executed lines 16-18.

**Induction Hypothesis.** Let $C$ be any reachable configuration and assume that the claim holds in all configurations that precede $C$.

**Induction Step.** We prove that the claim holds for $C$. This is proved by a case analysis on the step $s$ that is applied from $C^-$ to get $C$, let $p_j$ be the thread that executes $s$.

1. In case that $s$ is the execution of any of $s$ is the execution any of lines 1-3 and 5-6, the claims hold by the induction hypothesis.

2. $s$ is the execution of line 4.

   It suffices to argue that $p_j$ does not become the combiner by executing $s$. Let $C_4$ be the configuration at which $p_j$ executes line 4 of the pseudocode. Assume by the way of contradiction that $node_j(C) \rightarrow wait = false$ in some configuration $C_w$ between $C_3$ and $C$. By the pseudocode (lines 3 and 4), $node_j \rightarrow wait$ does not change value between $C_3$ and $C$ by $p_j$. Since $node_j(C) \rightarrow wait = true$, there must be a thread that writes $node_j \rightarrow wait = \mathit{false}$ between $C_3$ and $C$. Let $p_m$ be a thread that does so. By the pseudocode, it follows that $p_m$ must execute either line 18 or line 26 at $C_w$. Since $p_m$ is active at $C_w$, Claim 1.i does not hold at $C_w$ and by the induction hypothesis (for $C_w$) it follows that Claim 1.ii must hold at $C_w$. Moreover, the induction hypothesis (Claim 2.ii) implies that $p_m$ must be the combiner at $C_w$. However then, the induction hypothesis (Claim 2.iii) implies that $p_j$ should be a thread executing lines 8-12 at $C_w$. This is a contradiction since $p_j$ executes any of line 4 at $C_w$. We conclude that $p_j$ is not a combiner at $C$, which is a contradiction.

3. $s$ is the execution of line 7.

   We first prove that Claim 1 holds. By the induction hypothesis (Claim 1), one of the following conditions hold at $C^-$: (Claim 1.i) $Tail(C) = \perp$, there is no combiner at $C^-$ and there is no thread poised to execute any of lines 13-23 and 25-26 or (Claim 1.ii) $Tail(C^-)$ points to a node $nd \neq \perp$ and there is exactly one combiner at $C^-$.

   - Assume first that (1.i) is *true* at $C^-$. Since $Tail(C^-) = \perp$, the `Swap` instruction of line 7 returns a value equal to $\perp$ and thus, it follows that $p_j$ is a combiner at $C$. Since condition (1.i) guarantees that there was no combiner at $C^-$, $p_j$ is the only combiner in the system at $C$.
   - Assume now that condition (1.ii) is *true* at $C^-$. Since $Tail(C^-) \neq \perp$ and $node_j(C) = Tail(C^-)$, it follows that $p_j$ is not a combiner at $C$; notice that $p_j$ could not be a combiner at $C^-$ since the combiner is poised to execute some of the lines 8-26 at $C^-$ (whereas $p_j$ is poised to execute line 7 at $C^-$).

   The rest of the claims in each case, hold by the induction hypothesis.

4. If $s$ is the execution of any of the lines 8 or 10, the claim holds by induction hypothesis.

5. $s$ is the execution of line 9.

   It suffices to prove that Claim 4 holds. Let $node_x$ be the value of $MyPred$ of $p_j$ at $C$. Observation 6.1 implies that there is some thread $p_{j'}$ that executes the $S_{l-1}$ instruction and uses $node_x$ as parameter in $S_{l-1}$. Assume that $p_{j'}$ executes its $S_{l-1}$ instruction in some instance $A_x$ of CC-Synch. The pseudocode (line 9) implies that

$p_j$ writes a value equal to $node_j$ at $node_x \rightarrow next$ at $C$. Let $C_5$ be the configuration after the execution of line 5 by $p_{j'}$. We first prove that $p_{j'}$ executes at most one Swap instruction between $S_{l-1}$ and $C$. Assume by the way of contradiction that $p_{j'}$ executes two or more Swap instructions between $S_{l-1}$ and $C$. Let $S_{w1}$ ($S_{w2}$) be the first (second) of these Swap instructions. Assume that $S_{w1}$ is executed by some $A_{w1}$ instance of CC-Synch. By the definition of $A_{w1}$, it follows that finishes its operation before the execution of $S_{w2}$ an thus before configuration $C^-$. Furthermore, the definitions of $A_{w1}$, $A_{w2}$ and $A_j$ imply that $A_j$ executes its Swap instruction before $S_{w1}$ and a combiner executes lines 16-18 for servicing $A_j$ after the execution of lines 16-18 for servicing $A_{w1}$. This contradicts the induction hypothesis (Claim 3). Thus, at most one Swap instruction is executed between $S_{l-1}$ and $C$ by $p_{j'}$.

In case that $node_x(C^-) \rightarrow next = \perp$, the claim holds trivially since $p_j$ writes $node_j = nd_l$ to $node_x \rightarrow next = nd_{l-1} \rightarrow next$. The claim also trivially holds in case that $node_x \rightarrow next = node_j$ at $C^-$. We now prove that at configuration $C^-$, it holds that either $node_x \rightarrow next = \perp$ or $node_x \rightarrow next = node_j$. Assume, by the way of contradiction, that at configuration $C^-$, it holds that $node_x \rightarrow next = node_w$, where $node_w \neq node_j$ and $node_w \neq \perp$. Let $p_w$ be the thread that writes $node_w$ into $node_x \rightarrow next$ at some configuration $C_w$ between $C_5$ and $C^-$. Let $S_h$ be the Swap instruction of line 7 executed by $p_w$. Observation 6.1 implies that $p_{j'}$ executes the $S_{h-1}$ instruction that writes $node_x$ at $Tail$. Since, $p_w \neq p_j$, it follows that $S_{h-1} \neq S_{j-1}$. Observation 6.2 and the fact that at most one Swap instruction is executed between $C_{l-1}$ and $C^-$ by $p_{j'}$, imply that $S_{h-1}$ precedes $S_{j-1}$. Therefore, $S_h$ is executed between $S_{h-1}$ and $S_{j-1}$.

6. If $s$ is the execution of line 11, we distinguish the following two cases.

   - Assume first that $p_j$ is the combiner at $C^-$. It suffices to argue that $p_j$ is not poised to execute line 12 at $C$ (as needed by Claim 2.i). By definition, there is some configuration $C''$ that precedes $C$ (and occurs during the course of the execution of the current instance of DSM-Synch by $p_j$) at which either $node_j(C'') \rightarrow wait = false$ and $node_j(C'') \rightarrow completed = false$ or the Swap instruction (notice that $p_j$ is not the combiner due to a returned $\perp$ of line 7, since in a such a case the line 11 could not be executed). By the pseudocode, the *wait* or *completed* field of $node_j$ can change if a thread $p_m$ executes one of the lines 3, 4, 17, 18, or 26 after $C''$. Pseudocode (lines 1-3) implies that the *wait* or *completed* field of $node_j$ cannot change when threads other than $p_j$ execute any of the lines 3 or 4. Moreover, the induction hypothesis (Claims 1 and 2.ii) implies that no thread other than $p_j$ can change the *wait* or *completed* fields of $node_j$ by executing lines 17, 18, or 26. Thus, no thread other than $p_j$ can change these fields of $node_j$ from $C''$ to $C^-$. It follows that $p_j$ evaluates the condition of line 11 to *false* and therefore, it is not poised to execute line 12 at $C$ (as needed by 2.i).

- Assume now that $p_j$ is not a combiner at $C^-$. It suffices to argue that $p_j$ is poised to execute line 12 at $C$ (as needed by Claim 2.i). Assume by the way of contradiction that $p_j$ is not poised to execute line 12. Since $p_j$ executes line 11 at $C^-$, it must be that there is some configuration $C_{10}$ preceding $C^-$ at which $p_j$ evaluates the condition of the `while` statement of line 10 to *true*. Since $p_j$ is not the combiner, it follows that $node_i() \rightarrow completed = true$ at $C_{10}$. Since $p_j$ evaluates the if condition of line 11 to *false*, it follows that there is a configuration, let it be $C_w$, between $C_{10}$ and $C$ at which $node_i(C_w) \rightarrow complete = false$. The pseudocode (lines 4 and 17) implies that only $p_j$ writes a value equal to *false* on $node_i \rightarrow completed$ by executing line 4 of the pseudocode. The pseudocode (lines 10-11) implies that $p_j$ does not perform any `Write` operation at $C_w$. Thus, $p_j$ evaluates the if condition of line 11 to *true*, which is a contradiction.

The rest of the claims in each case, hold by the induction hypothesis.

7. $s$ is the execution of line 13.

By induction hypothesis (Claims 1 and 2), it follows that only the combiner $p_i$ can be poised to execute line 13 at $C^-$; thus, $p_j = p_i$, and $p_j$ has not executed any iteration of the while loop (lines 14-21) yet. By the pseudocode (line 13) it follows that the node assigned to *nd* is equal to $node_j$. Thus, Claim 2.iii holds. The rest of the claims hold by the induction hypothesis.

8. In case that $s$ is the execution of any of the lines 14-16, the claims hold by induction hypothesis.

9. $s$ is the execution of line 17. Suppose that this is the $k$th time, $k \geq 1$, that $p_i$ executes line 17 during $A_i$ and assume that line 17 has been executed $m$ times in total until $C$. By the induction hypothesis (Claim 2.iii) there is a thread $p_k$ such that $node_k(C^-) = nd_k$, and either $p_k = p_i$ or $p_k$ is executing one of the lines 6-12 at $C^-$.

   - Assume first that $k = 1$. Let $C_7$ be the configuration resulted when $p_i$ executes line 7. By the induction hypothesis (Claim 2.i), $p_i$ cannot be a combiner before $C_7$.

     – Assume first that $p_i$ is not the combiner at $C_7$.
     Since $p_i$ is active at all configurations between $C_7$ and $C$, by the induction hypothesis (Claim 1), it follows that there is always some combiner in the system between $C_7^-$ and $C$. Assume that $p_i$ became the combiner at some configuration $C'$ preceding $C$, and let $p_j$ be the thread that was the combiner at $C'^-$; let $A_j$ be the instance of DSM-Synch executed by $p_j$ at $C'^-$. Since $p_i$ becomes the combiner after $p_j$ and $p_i$ is executing line 17 for the last time at $C$, it follows that the $(m-1)$st time that lines 16-18 were executed was the last time that $p_j$ executed those lines during $A_j$. Suppose that $p_j$ visits $k', k' \geq 1$ nodes. Let $nd'_{k'}$ be the last node visited by $A_j$. The induction hypothesis (Claim 2.iv) implies that $nd'_{k'} = node_i(C')$ (since $p_i$ is the unique combiner in the system right after $p_j$). The pseudocode (lines 13-21) imply that $nd'_1 = node_j(C'^-) \neq node_i(C'^-)$; thus, $k' > 1$. By the pseudocode, $p_i$ becomes a combiner when $p_j$ executes line 26 (i.e. at configuration $C'$). By the pseudocode, $node_i(C') = nd'_{k'} = nd'_{k'-1} \rightarrow next$. We distinguish the following two cases. In case that $p_{k'-1} = p_j$, the induction hypothesis (Claim 3), implies that $nd'_{k'-1} = nd_{m-1}$. Pseudocode (lines 21 and 26) implies that $nd'_{k'} = nd'_{k'-1} \rightarrow next = nd_{m-1} \rightarrow next$ and $nd'_{k'} = node_i(C) \neq \perp$, it follows that $nd_{m-1} \rightarrow next \neq \perp$. Thus, Claim 4 implies that $node_i(C) = nd'_1 = nd_{m-1} \rightarrow next = nd_m$, as needed (by Claim 3). In case that $p_{k'-1} \neq p_j$, by the induction hypothesis (Claims

2.iii), there is some thread $p_{k'-1}$ such that at each configuration $\tilde{C}$ in $\beta_{k'-1}$, $p_{k'-1}$ was executing one of the lines 8-12 at $\tilde{C}$ and $node_{k'-1}(\tilde{C}) = nd'_{k'-1}$; specifically, $p_{k'-1}$ was active at configuration $C_{21}$ at which line 21 was executed by $p_j$ during $\beta_{k'-1}$. Also, by the induction hypothesis (Claim 3), it follows that $nd'_{k'-1} = nd_{m-1}$. Since $nd'_{k'} = nd'_{k'-1} \rightarrow next = nd_{m-1} \rightarrow next$ and $nd'_{k'} = node_i \neq \perp$, it follows that $nd_{m-1} \rightarrow next \neq \perp$. Thus, Claim 4 implies that $node_i(C) = nd'_1 = nd_{m-1} \rightarrow next = nd_m$, as needed (by Claim 3).

– Assume now that $p_i$ is a combiner at $C_7$. It follows that the `Swap` instruction of line 7 returns $\perp$ to $p_i$. By the induction hypothesis (Claim 1), it follows that there is no combiner at $C_7^-$ and no active thread is executing any of the lines 8-27 at $C_7^-$. Let $C'$ be the last configuration preceding $C_7^-$ at which there was a combiner $p_c$ in the system; let $A_c$ be the instance of DSM-Synch executed by $p_c$ at $C'$. Notice that $p_c$ has executed line 26 in $A_c$ just before configuration $C'$. By the induction hypothesis (Claim 2.iv), it follows that there is no combiner in the system at $C'$. By definition of $p_c$, there is no combiner in the system between $C'$ and $C_7^-$. By the induction hypothesis (Claim 2.iv), it follows that there is no combiner in the system at $C'$. By definition of $p_c$, there is no combiner in the system between $C'$ and $C_7^-$. We first prove that there no `Swap` operation is executed between $C'$ and $C_7^-$. Assume by the way of contradiction that at least one `Swap` is executed between $C'$ and $C_7^-$. Let $s$ be the first such `Swap` and let $C''$ be the configuration just after the execution of $s$. Assume that $s$ is executed by some thread $p_s$. Since there is no combiner between $C'$ and $C_7^-$, induction hypothesis (Claim 1.i) implies that $Tail(C'') \neq \perp$. Therefore, there is a combiner at $C''$. This contradicts our assumption that there is no combiner between $C'$ and $C_7^-$. Thus, the `Swap` instruction executed by $p_i$ at $C_7^-$ is the first `Swap` instruction executed between $C'$ and $C_7^-$. Assume that $p_c$ visited $k'$ nodes in $A_c$; and denote by $nd'_{k'}$ the last node visited by $p_c$ in $A_c$. By the pseudocode (lines 19-20), it follows that $nd_{k'} \neq \perp$; Obviously, the $(m-1)$th time that lines 17-18 were executed was for node $nd_{k'}$. By the induction hypothesis (Claim 3), $nd_{k'} = nd_{m-1}$, as needed.

- Assume now that $k > 1$. By the pseudocode, $nd_k$ (in $A_i$ instance) cannot be equal to $\perp$ since otherwise the $k$th iteration of the `while` loop would not be executed by $p_i$ (see lines 19-20 of pseudocode). By the induction hypothesis (Claims 2.iii), either $p_{k-1} = p_i$ or there is some thread $p_{k-1} \neq p_i$ such that at each configuration $C'$ in $\beta_{k-1}$, $p_{k-1}$ was executing one of the lines 6-12 at $C'$. By the induction hypothesis (Claim 3), it follows that $nd'_{k-1} = nd_{m-1}$. By the pseudocode, it follows that $nd'_k = nd'_{k-1} \rightarrow next = nd_{m-1} \rightarrow next$. Since $nd'_k \neq \perp$, it follows that $nd_{m-1} \rightarrow next \neq \perp$. Thus, Claim 4 implies that $nd'_k = nd_{m-1} \rightarrow next = nd_m$, as needed (by Claim 3).

10. $s$ is the execution either of line 18.

    Assume that either line 18 is executed for the $k$th time, $k \geq 1$. Remind that it is only the combiner that can be poised to execute this line of code at $C^-$. By the induction hypothesis (Claim 2.iii) there is a thread $p_k$ such that $node_k(C^-) = nd_k$, and either $p_k = p_i$ or $p_k$ is executing one of the lines 8-12 at $C^-$. Since $s$ changes either the *completed* or the *wait* field of $node_k(C^-)$, Claim 1 holds by induction hypothesis. The rest of the claims hold by induction hypothesis.

11. In case that $s$ is the execution either of line 19 or line 20, the claims hold trivially.

12. $s$ is the execution of line 21.

    It is enough to argue that Claim 2.iii holds after the execution of $s$. The rest of claims hold by induction hypothesis. Assume that the execution of $s$ by $p_i$ identifies the $k$th node visited by $p_i$, $k > 1$ (notice that the first node visited by $p_i$ is identified by executing line 13 and not line 21; thus, $k > 1$). If $nd'_k = \perp$, Claim 2.iii holds (by induction hypothesis for each $l < k$). Thus, assume that $nd'_k \neq \perp$. Denote by $S_1$ the `Swap` operation executed by $p_i$ in *req* and denote by $S_k$ the `Swap` operation executed by $p_k$. We first prove that $S_1$ precedes $S_k$. By the pseudocode, $nd'_{k-1} \rightarrow next = nd'_k$ cannot be equal to $\perp$ since otherwise the $(k-1)$th iteration of the `while`, where the $k$th node to be visited by $p_i$ is identified (this occurs when $p_i$ executes line 21 of that iteration), would not be executed. By the induction hypothesis (Claim 2.iii), either $p_{k-1} = p_i$ or there is some thread $p_{k-1}$ such that at each configuration $C'$ in $\beta_{k-1}$, $p_{k-1}$ was executing one of the lines 6-12 at $C'$ and $node_{k-1}(C') = nd_{k-1}$. In case that

$p_{k-1} = p_i$, the pseudocode implies that $nd'_1 \rightarrow next = nd'_k$. Claim 4 implies that the `Swap` operation $S_1$ executed by $p_i$ precedes the execution of the `Swap` operation $S_k$ by $p_k$. In case that $p_{k-1} \neq p_i$, (Claim 2.iii) implies that $p_{k-1}$ was active at configuration $C_{21}$ at which line 21 was executed by $p_i$ during $\beta_{k-1}$. The pseudocode implies that $nd'_{k-1} \rightarrow next = node_{k-1}(C_{21}) \rightarrow next$. Since we have assumed that $nd'_k \neq \perp$, it follows that $node_{k-1}(C_{21}) \rightarrow next \neq \perp$. Thus, Claim 4 implies that there is a thread $p_k$ that has executed line 7 before $C_{21}$ such that $node_k(C) = nd'_k$; moreover, Claim 4 implies that the `Swap` operation $S_{k-1}$ executed by $p_{k-1}$ precedes the execution of the `Swap` operation $S_k$ by $p_k$. By the pseudocode, $p_i$ executes lines 16-18 for itself during the first iteration of the while loop of lines 14-21, i.e. before executing it for $p_{k-1}$. Since $p_i$ executes lines 16-18 later on for $p_{k-1}$, the induction hypothesis (Claims 2.iii and 3) implies that $S_1$ has occurred before $S_{k-1}$. It follows that $S_1$ has occurred before $S_k$.

We continue to prove that $p_k$ is active at $C^-$. Assume, by the way of contradiction, that the instance $A_k$ of CC-Synch executed by $p_k$ is not active at $C^-$. Recall that $p_k$ executed $S_k$ after $S_1$. Thus, $p_i$ was active while executing one of the lines 6-21 when $S_k$ was executed; let $C_k$ be the configuration just after the execution of $S_k$. Apparently, $C_k$ precedes $C$. Since $p_i$ is active between $C_1$ and $C$, by the induction hypothesis (Claim 1), it follows that there is always some combiner in the system in all configurations between $C_1$ (which results by applying $S_1$) and $C$. Thus, there is some combiner in all configurations between $C_k^-$ and $C$. By the pseudocode, the *completed* field of $node_k$ must be equal to *true* when $p_k$ terminates. By the pseudocode, this can happen only if there is some thread $p_h$ that executes lines 16-18 with $tmpNode = node_k$ at some configuration before $C$. Suppose that lines 16-18 have been executed $h'$ times in total until the configuration that $p_h$ executed line 18 with $tmpNode = node_k$. Since $S_1$ is performed before $S_k$, the induction hypothesis (Claim 3) implies that lines 18-26 have been executed for $p_i$ before being executed for $p_k$ (assume that this has happened the $h''$th time that lines 16-18 were executed). By the induction hypothesis (Claim 3), $node_i = nd_{h''}$. However, by the pseudocode, it follows that lines 16-18 are executed for $node_i$ by $p_i$ during the execution of the first iteration of the while loop of lines 14-21; let this be the $h$th time that lines 16-18 are executed. Since $p_i$ has not visited $node_k$ before $C$, it follows that $h > h''$, which contradicts the induction hypothesis (Claim 3).

Thus, $p_k$ is still active at $C^-$. By the induction hypothesis (Claim 2.ii), no thread other than the combiner $p_i$ executes lines 13-27 at $C^-$. Thus, $p_i$ executes one of the lines 6-12 at $C^-$. Since it is thread $p_i$ that executes $s$, $p_k$ executes one of the lines 6-12 at $C$. This completes the proof of Claim 2.iii.

13. In case that $s$ is the execution of line 22 claims hold trivially.

14. $s$ is the execution of line 23.

    In case that the CAS instruction of line 23 fails, $Tail(C) \neq \perp$ and line 24 is not executed by $p_i$. Thus, Claim 1.i holds. In case that the CAS instruction of line 23 succeeds, a value equal to $\perp$ is set and $p_i$, which is the unique combiner, is poised to execute line 24. Thus, Claim 1 holds.

15. In case that $s$ is the execution of line 25 claims hold trivially.

16. $s$ is the execution of line 26. Recall that it is only the combiner that can be poised to execute this line of code at $C^-$. Assume that $p_i$ visits the $(k-1)$th node at the execution of line 22. The pseudocode (lines 22-25) implies that $nd'_k = nd'_{k-1} \rightarrow next \neq \perp$ at $C$. Induction hypothesis (Claim 2.iii) implies that $nd'_k = node_k(C^-)$ for some thread $p_k$ that is poised to execute one of the lines 8-12 at $C^-$. By the induction hypothesis (Claim 3), $node_k$ is visited for the first time since $A_k$ was initialized. This implies that $node_k(C^-) \rightarrow completed = false$. Since $s$ changes $node_k(C^-) \rightarrow wait$ to *false* it follows that $p_k$ is a combiner at $C$. Since $p_i$ was the unique combiner in the system at $C^-$ and it is not a combiner anymore after the execution of line 26, it follows that the unique combiner in the system at $C$ is $p_k$, as needed by Claim 2.iv. Moreover, induction hypothesis implies that $Tail(C^-) \neq \perp$, as needed by Claim 1. The rest of the claims in each case hold by the induction hypothesis. ∎

Let $nd_i$ be the node of the list that is assigned to $p_i$ for $req_i$. Thread $p_i$ completes the execution of DSM-Synch for $req_i$ in any of lines 12, 24 and line 27. Assume first, that $p_i$ returns on line 12. Lemma 6.6 (Claim 2.iii) implies that a combiner thread $p_j$ has served $req_i$ before the execution of line 12 by $p_i$. Therefore, $p_j$ has executed line 16 for $nd_i$ at some iteration $l > 1$ of its while loop (lines 14-21). Request $req_i$ is linearized just before the execution of this instance of line 16 by $p_j$. Assume now that $p_i$ returns either on line 27 or on line 24. Lemma 6.6 (Claim 2.iii) implies that $p_i$ serves its request on its own when it executes line 16 at the first iteration of its while loop (lines 14-21). In this case, $req_i$ is linearized just before the execution of line 16 of the first iteration of $p_i$'As while loop. Obviously, in both cases the linearization point of $req_i$ is within its execution interval. Consistency is immediately implied by Claims 2.iii and 3 of Lemma 6.6. Thus, the following theorem holds.

**Theorem 6.2.** DSM-Synch *is a linearizable synchronization algorithm.*

## 6.4 Performance evaluation of CC-Synch, DSM-Synch and H-Synch

We evaluated CC-Synch and DSM-Synch in two different multiprocessor machine architectures. The first is a 32-core machine consisting of four AMD Opteron 6134 processors (Magny Cours). Each processor consists of 2 dies and each die contains 4 processing cores. Communication among the cores of the same die is achieved with a fast L3 cache. Dies communicate with Hyper-Transport links which create a complex topology that resembles a hypercube [21]. The second machine is a 128-way Sun consisting of 2 UltraSPARC-T2 processors (Niagara 2). Each processor consists of 8 processing cores, each of which is able to handle 8 threads. Communication among the cores of the same processor is achieved with a fast L2 cache. All experiments on the Magny Cours machine were performed using the gcc 4.3.4 compiler, while experiments on the Niagara 2 machine were performed using gcc 4.5.1. In order to avoid bottlenecks in memory allocation, the Hoard memory allocator [18] was used. The operating system running on the Magny Cours machine was Linux with kernel 2.6.18, while the operating system running on the Niagara 2 machine was Solaris 10.

Thread binding is employed for the following reason. Assume that the number of threads is smaller than the number of cores and suppose that two threads are running. The scheduler may decide to run them either on different processors (chips) or within the same chip. In the first case the communication cost is an order of magnitude more

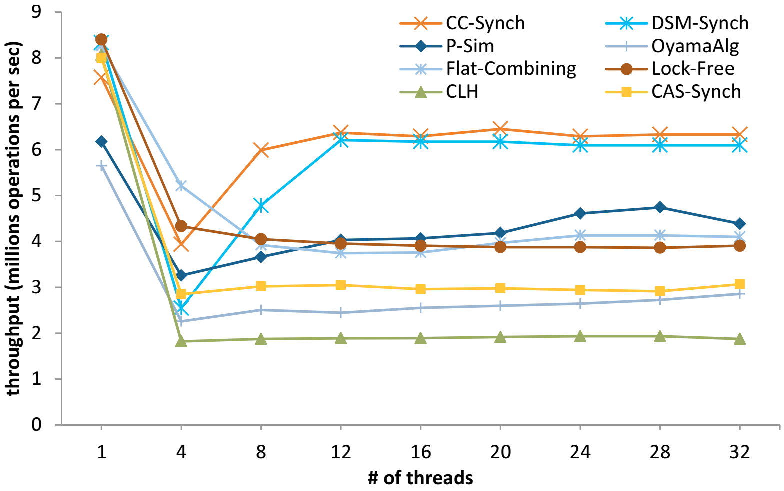


Figure 6.1: Average throughput of CC-Synch and DSM-Synch on the Magny Cours machine while simulating a Fetch&Multiply object.

than in the second. Thus, if thread binding is not used, a significant uncertainty factor is introduced which may lead to an unreliable experiment. We observed that this was a usual phenomenon. So, on the Magny Cours machine, the $i$-th thread was bound to the $i$-th core of the machine; we first exploited multi-core, then multi-chip and then multi-socket configuration. On the Niagara 2 machine, we follow a slightly different scheduling similar to that used in [24] in order to better explore the performance properties of hierarchical algorithms. More specifically, we split threads into two groups, one for each socket.

In order to evaluate CC-Synch and DSM-Synch, we compare their performance with that of state-of-the-art synchronization algorithms. Specifically, they are compared with P-Sim (the wait-free universal construction presented in [28]), flat-combining [33, 34], the CLH spin-lock [23, 47][‡], OyamaAlg [52], and a simple lock-free implementation. The lock-free implementation was implemented using a CAS object. Specifically, whenever a thread wants to apply a Fetch&Multiply, it repeatedly executes CAS until it succeeds; a backoff scheme is employed to increase the scalability of this implementation. Since the Niagara 2 machine does not support Add which is employed by P-Sim and is necessary, as shown in [28], for its good performance, no experiment was performed for P-Sim on the Niagara 2 machine. We also evaluated a variation of CC-Synch (called CAS-Synch), in which Swap is simulated with a CAS object in a lock-free manner. This allows us to explore the performance advantages of Swap over CAS.

[‡] As expected for cache-coherent NUMA architectures, we experimentally saw that MCS [49] spin locks have slightly worse performance than CLH locks in both machines, so we present experimental results for CLH locks.

On the Niagara 2 machine, H-Synch, the hierarchical NUMA lock (called FCMCS below) presented in [24] , and the hierarchical lock called C-BO-MCS presented in [25] were also evaluated. On the Magny Cours machine, experiments show no performance benefit when using any of the hierarchical algorithms. This is rational to the fact that the machine consists of many but very small clusters of cores. Thus, we have not included any performance measurements for the hierarchical algorithms on the Magny Cours machine. All algorithms were carefully optimized and for those that use backoff schemes, we performed a large number of experiments in order to choose the best backoff parameters. We used the flat-combining implementation that was provided by its inventors [33, 34] and we choose its parameters very carefully in order to achieve the best performance. We further optimized the code of flat-combining to run faster than its original version [33, 34] on the Magny Cours machine. We used the latest version of P-Sim code (version 0.8) [45, 28]. The source code of our implementations is provided at `http://code.google.com/p/sim-universal-construction/`.

The first experiment we performed is a synthetic benchmark (Figures 6.1, 6.2), where a simple `Fetch&Multiply` object is simulated. We measure the average throughput (`Fetch&Multiply` per second) that each synchronization algorithm achieves when it executes $10^7$ `Fetch&Multiply` operations (i.e. always the same amount of work), for different values of $n$; each thread executes $10^7/n$ `Fetch&Multiply`. Specifically, the horizontal axis of Figures 6.1, 6.2 represents the number of threads $n$, while the vertical axis displays the throughput (in millions of operations per second) that each synchronization algorithm has performed. For each value of n, the experiment has been performed 10 times and averages have been calculated. A random number of dummy loop iterations (up to 64) have been inserted between the execution of two `Fetch&Multiply` by the same thread; specifically, in each iteration a volatile counter is increased. In this way, we simulate a random work load large enough to avoid unrealistically low cache miss ratios and long runs (but not too big to reduce contention). Figure 6.6 studies the performance behavior of our algorithms for different values of the random work.

In the experiments performed on the Magny Cours machine (Figure 6.1), CC-Synch outperforms all other synchronization algorithms. Specifically, CC-Synch achieves up to 1.54 higher throughput than flat-combining and outperforms P-Sim by a factor of up to 1.52. The lock free implementation of `Fetch&Multiply` is slightly slower than P-Sim and

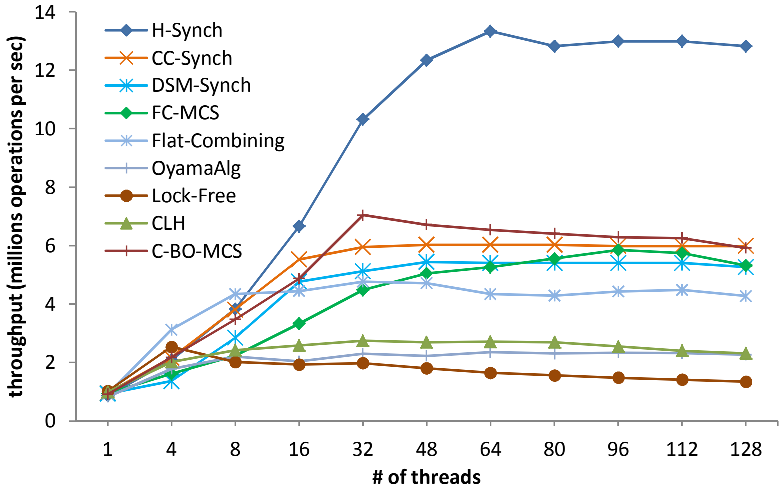


Figure 6.2: Average throughput of CC-Synch, DSM-Synch and H-Synch on the Niagara 2 machine while simulating a Fetch&Multiply object.

flat-combining. Also, CC-Synch is up to 2.7 times faster than OyamaAlg [52]. DSM-Synch performs also very well; its performance is close to that of CC-Synch, despite the fact that it is designed for machines following the DSM model. Figure 6.1 also shows that simulating Swap using CAS (in a lock-free way) induces a serious performance penalty; specifically, CAS-Synch is two times slower than CC-Synch.

Similarly to the experiments performed on the Magny Cours machine, CC-Synch outperforms all algorithms other than H-Synch and C-BO-MCS on the Niagara 2 machine (Figure 6.2). More specifically, CC-Synch outperforms flat-combining by a factor of up to 1.4. It is noticeable that even CC-Synch itself (not its hierarchical version) outperforms FCMCS [24] by a factor of up to 1.65, despite the fact that FCMCS exploits the hierarchical characteristics of communication in the machine. In contrast to FCMCS, C-BO-MCS slightly outperforms CC-Synch in case of $32 - 112$ threads, exploiting its hierarchical characteristics in the Niagara 2 machine. However, C-BO-MCS is vastly outperformed by H-Synch. The relatively small performance gap between flat-combing and FCMCS may seem surprising at first; however, this result is rational to the fact that FCMCS causes more cache misses when accessing the shared data. Specifically, in FCMCS, combining is not used in applying the requests, so each request may be applied by a different thread; thus, each time a thread accesses the shared data cache misses may occur. This is avoided when combining is employed in serving the requests, as done by the other studied combining-based synchronization algorithms. DSM-Synch exposes almost the same performance to CC-Synch on the Niagara 2 machine. H-Synch which is the hierarchical version of CC-

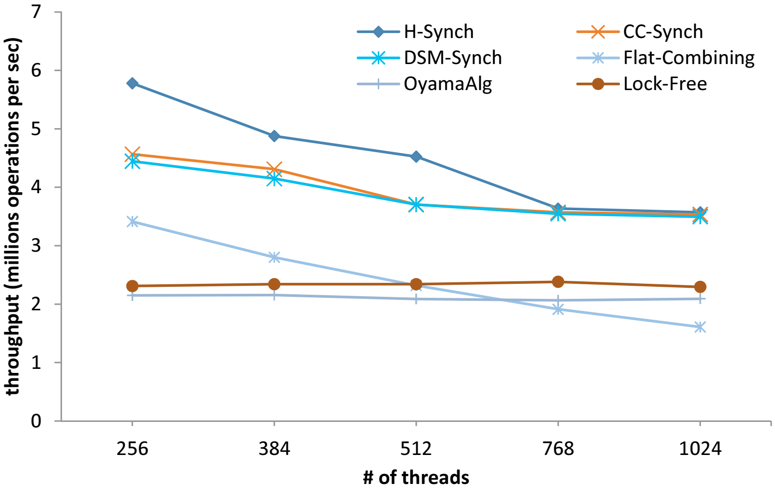


Figure 6.3: Average throughput of CC-Synch, DSM-Synch and H-Synch on the Niagara 2 machine for $n > 128$ (over-subscribing) while simulating a `Fetch&Multiply` object.

Synch outperforms FCMCS by a factor of up to 2.65 and flat-combining itself by a factor of up to 3.0. CC-Synch is up to 2.55 times faster than OyamaAlg [52], while H-Synch is more than 6 times faster. The performance of CAS-Synch is not illustrated in Figure 6.2 since CAS-Synch results in very poor performance on the Niagara 2 machine.

As shown in Figure 6.1, on the Magny Cours machine, all algorithms perform faster in case $n = 1$ than for larger values of $n$. On the contrary, Figure 6.2 shows that, on the Niagara 2 machine, the performance of all algorithms is always better for larger values of $n$. The Magny Cours machine implements atomic instructions (`CAS`, `Swap`, etc.) in the private L1 cache which is very fast. This and the fact that the local workload is small (up to 64) are the reasons for the very high performance that the Magny Cours machine achieves in case of $n = 1$. In contrast, a Niagara 2 processor implements atomic instructions in the shared L2 cache which is slower (Niagara 2 processor is optimized for contented workloads, i.e. in case of $n > 1$).

In Figure 6.3, we study the performance of each implementation on the Niagara 2 machine for $n > 128$, i.e. when $n$ is larger than the number of threads that the machine is able to handle simultaneously; thus, the machine is over-subscribed. We do not include any measurement from FCMCS and CLH locks since in this experiment they do not achieve good performance. As illustrated in Figure 6.3, H-Synch, CC-Synch and DSM-Synch achieve better performance than any other synchronization algorithm for any value of $n$.

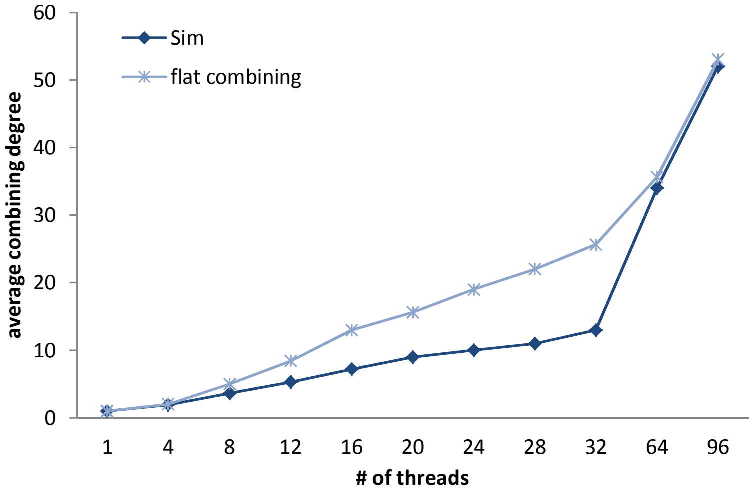


Figure 6.4: Average degree of combining of CC-Synch, DSM-Synch and H-Synch while simulating a Fetch&Multiply object.

Figures 6.4 - 6.5 aim at investigating the reasons for the good performance of CC-Synch and DSM-Synch. More specifically, from Figure 6.5, it follows that on the Magny Cours machine, P-Sim and flat-combining execute slightly more (up to 10% more) atomic instructions than CC-Synch and OyamaAlg [52]. The experiments showed that to achieve the best performance for the lock-free algorithm, the back-off should not be too high. By appropriately choosing the back-off to get the best performance, it turned out that the average number of CAS performed by each instance of the algorithm is two which is bigger than the average number of atomic instructions executed by each instance of CC-Synch and DSM-Synch. Thus, the lock-free algorithm has a performance disadvantage compared to these algorithms.

Figure 6.4 displays the average degree of combining, i.e. the average number of requests that are executed by a combiner. It shows that CC-Synch and DSM-Synch achieve better degree of combining which is almost 3 times more than that of P-Sim and flat-combining[§]. Our efforts to increase the average degree of combining for flat-combining by carefully tuning its parameters (i.e. by increasing the combining rounds or by changing the polling level), revealed that when the combining degree was increased the average throughput was decreased. On the contrary, P-Sim operates in a way that it can help as much threads as the system's point contention (i.e. as the maximum number of threads that can be simultaneously active at any point in time which might be equal to $n$).

[§] In Figure 6.4, we have not included results for OyamaAlg since the variance of the combining degree in this algorithm was too high to get a realistic view. This is due to the fact that a combiner thread in this algorithm may be enforced to help an unbounded number of operations.

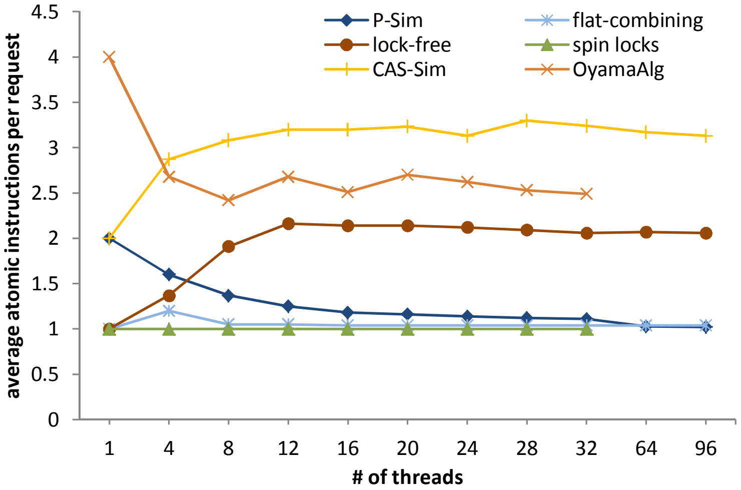


Figure 6.5: Average number of atomic instructions (`CAS`, `Swap` and `Add`) that CC-Synch, DSM-Synch and H-Synch execute on the Niagara 2 machine while simulating a `Fetch&Multiply` object.

In the experiment of Figure 6.6, we studied the behavior of the competing algorithms for different amounts of random work. This experiment was executed on the Magny Cours machine; the number of threads was fixed to 32. Figure 6.6 shows that for a wide range of values for random work ($64 - 2048$), most algorithms have a small difference in the exhibited throughput. This shows that the communication cost is the dominant factor, whereas the time invested to execute the local random work does not play any significant role. An exception is the lock-free algorithm, which, in case the random work is equal to zero, has unrealistically high performance. This is due to the fact that, in this case, a thread could uninterruptedly execute thousands of `Fetch&Multiply` before some other thread starts its execution. This phenomenon (called long runs) has been also discussed in [28, 50] as an unrealistic workload. Figure 6.6 shows that by slightly increasing the random work, the performance of the lock-free algorithm rapidly decreases. The same phenomenon, although in a smaller scale, was also noticed for flat-combining. Similarly to the lock-free implementation, flat-combining has high throughput for very small amounts of random work, although its performance vastly decreases when the random work is slightly increased. In cases that the number of iterations is 2048 or more, the time needed to execute this loop becomes the dominant performance factor, i.e. executing the loop becomes more expensive than executing the algorithm for applying a FetchAndMultiply instruction. Thus, the performance of all algorithms starts to decrease

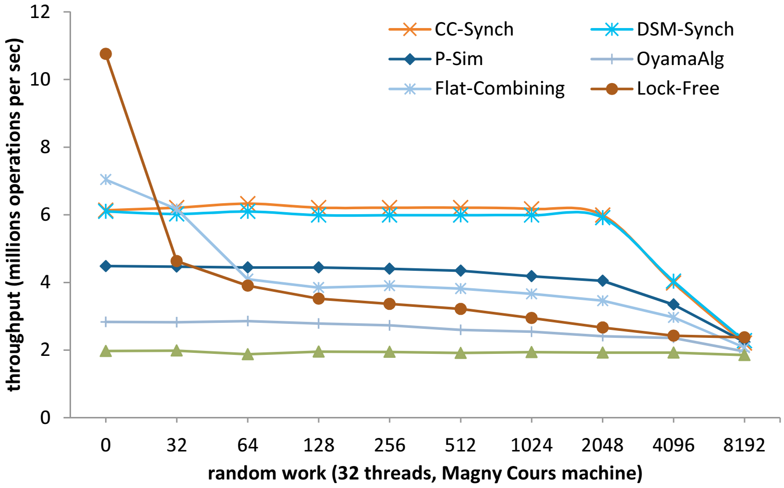


Figure 6.6: Average throughput of CC-Synch, DSM-Synch and H-Synch for different values of random work.

| Algorithm | cache misses | cpu cycles spent in memory stalls |
|---|---|---|
| CC-Synch | 4.1 | 2747 |
| Sim | 4.9 | 6328 |
| flat-combining | 5.8 | 6501 |

Table 6.3: Cache misses and memory stalls per operation for $n = 16$ of CC-Synch, P-Sim and flat-combining.

and the performance gap between them becomes insignificant. We remark that the scale of the horizontal axis of Figure 6.6 is logarithmic.

Table 6.3 shows some measurements from performance counters. We observed that the extra cache misses incurred by P-Sim and flat-combining was caused due to the number of failed CAS instructions; this number becomes worse if these algorithms are not properly tuned. Since failed CAS instructions cause cache misses and branch mispredictions, we conclude that a combining algorithm that avoids them has a serious performance advantage.

## 6.5 Highly-efficient blocking data structures

We further investigate the performance of CC-Synch, DSM-Synch and H-Synch by implementing common concurrent data structures (i.e. shared stacks and queues). We compare the performance of these implementations with state-of-the-art shared stack and queue implementations. Specifically, the shared stack implementation based on CC-Synch, called

CC-Stack, was evaluated against SimStack [28], the lock-free stack implementation presented by Treiber in [58], a stack implementation based on CLH spin locks [23, 47], and a linked stack implementation based on flat-combining [33, 34] (called FCStack). Both CC-Stack and FCStack eliminate pairs of push and pop whenever possible; the performance of elimination [35] has been studied in [28] and [34] where experiments show that elimination is outperformed by SimStack and FCStack. We also implemented a shared stack based on H-Synch and FCMCS [24] and evaluated their performance on the Niagara 2 machine.

The experiment we perform is similar to that performed by Michael and Scott for queues in [50]. We measure the average throughput (operations per second) that each algorithm achieves (every thread executes $10^7/n$ pairs of Enqueue and Dequeue operations) for different values of $n$. Again, the experiments have been performed several times and averages have been taken; we have simulated a random workload by executing a random number of iterations (up to 64) of a dummy loop after each operation.

As it is shown in Figure 6.7, on the Magny Cours machine, CC-Stack performs up to 1.68 times faster than FCStack, and up to 1.59 times faster than SimStack. The stack implementation based on the CLH spin lock had much lower performance. The stack implementation based on DSM-Synch, called DSM-Stack, performs slightly worse than CC-Synch but it is much better than all other algorithms. On the Niagara 2 machine, the shared stack based on CC-Synch performs 1.4 times faster than FCStack (Figure 6.8). The stack implementation based on DSM-Synch performs worse than CC-Stack but it is again better than all other algorithms. It is noticeable that the stack implementations based

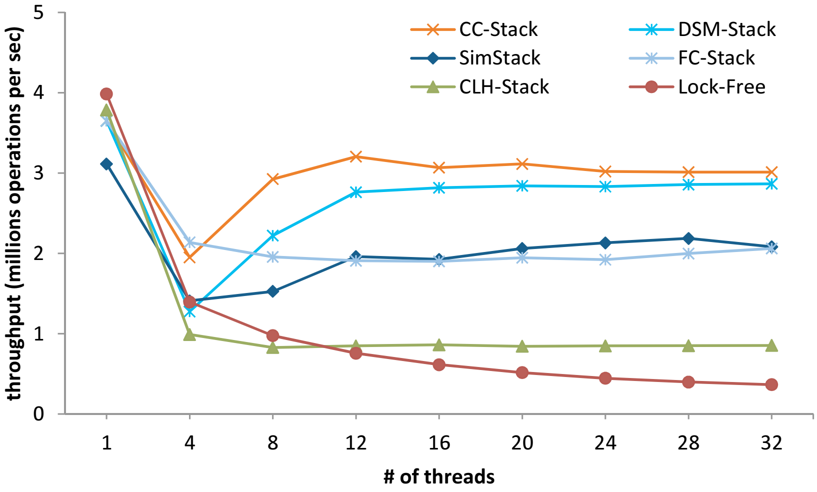


Figure 6.7: Average throughput of CC-Stack and DSM-Stack on the Magny Cours machine.

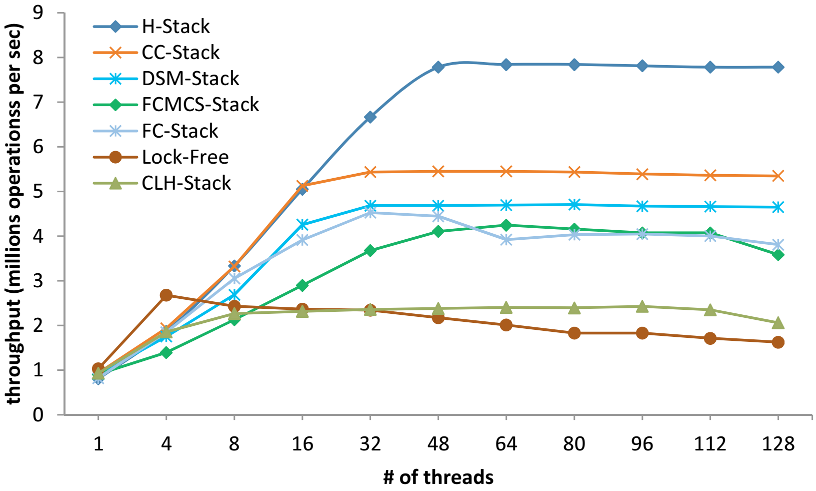


Figure 6.8: Average throughput of CC-Stack, DSM-Stack and H-Stack the Niagara 2 machine.

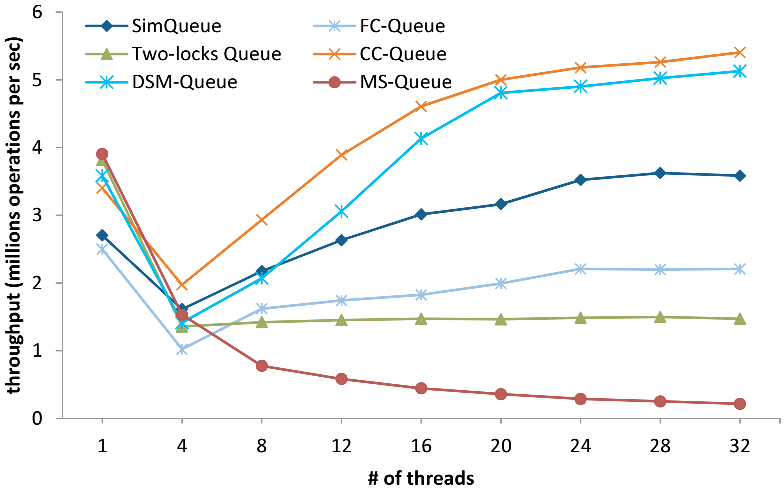


Figure 6.9: Average throughput of CC-Queue and DSM-Queue on the Magny Cours machine.

on CC-Synch and DSM-Synch outperform by a factor of up to 1.49 the shared stack based on FCMCS of [24]. The stack implementation based on H-Synch significantly outperforms all other implementations, being up to 2.0 times faster than FCStack and up to 2.1 times faster than the stack based on FCMCS [24].

We also implement and experimentally analyze shared queues based on CC-Synch, DSM-Synch and H-Synch, which are called CC-Queue, DSM-Queue and H-Queue, respectively. Specifically, the two-locks queue implementation presented in [50] is enhanced by replacing the ordinary locks with instances of CC-Synch, DSM-Synch and H-Synch, respectively. These implementations are compared (Figures 6.9-6.10) with SimQueue [28], the lock free queue implementation and the two-locks implementation presented in [50], and the queue implementation based on flat-combining [33, 34] (called FCQueue). On the

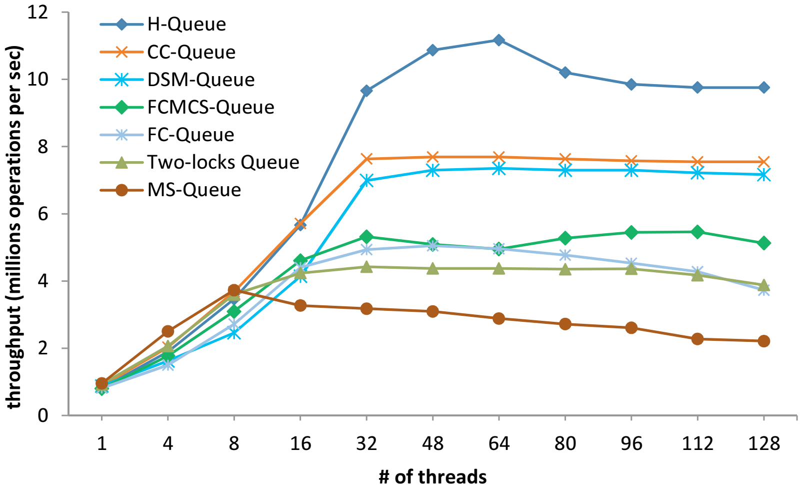


Figure 6.10: Average throughput of CC-Queue, DSM-Queue and H-Queue on the Niagara 2 machine.

Niagara 2 machine, we additionally implemented and evaluated a two-locks queue variant using FCMCS [24]. The queue experiment was similar to that for stacks.

As illustrated in Figure 6.9, on the Magny Cours machine, SimQueue exhibits better performance than any algorithm other than CC-Queue and DSM-Queue, as expected based on results in [28]. CC-Queue performs up to 2.53 times faster than FCQueue (Figure 6.9) and 2.1 times faster than SimQueue. DSM-Queue performs slightly worse than CC-Queue but better than all other algorithms. On the Niagara 2 machine (Figure 6.10), FCQueue performs better than all algorithms other than CC-Queue, DSM-Queue and H-Queue (recall that SimQueue has not been implemented in this machine since Add instructions are not included in its instruction set). CC-Queue performs up to 1.8 times faster than FCQueue and up to 1.55 times faster than the queue based on [24]. The queue implementation based on H-Synch greatly outperforms all other candidates by being up to 2.25 times faster than the queue implementation based on FCMCS [24]. It is also noticeable that the performance gap between FCQueue and the two-locks queue is smaller on the Niagara 2 machine. This is due to the fact that the CLH locks perform very well in this machine and the parallel use of two different locks (one for enqueues and one for dequeues) gives a performance boost in the two-locks algorithm. Again, DSM-Queue performs slightly worse than CC-Queue but better than all other algorithms except from H-Queue on the Niagara 2 machine.

# Chapter 7

# Conclusions and Future Work

In this dissertation the RedBlue, Sim and Synch families of synchronization algorithms are presented.

The RedBlue algorithms are adaptive synchronization algorithms using Read/Write and LL/SC base objects. The RedBlue synchronization algorithms achieve better step complexity than all previously presented algorithms. F-RedBlue which is the first algorithm of the RedBlue synchronization algorithms, matches the $\Omega(\log n)$ lower bound presented by Jaynati in [41].

The Sim synchronization algorithms achieve better step complexity by using other base objects than LL/SC or CAS. More specifically, Sim achieves constant step complexity by using Add additionally to LL/SC and Read/Write base objects. It is noticeable that P-Sim, which is a practical version of Sim outperforms the state-of-the-art synchronization algorithms and ensures stronger progress guarantees. P-Sim also shows that the architectures that support Add base objects have significant performance benefits.

The Synch synchronization algorithms are blocking implementations of the combining technique. Synch synchronization algorithms achieve much better performance than all other synchronization algorithms, while having nice complexity features. H-Synch is an hierarchical version of CC-Synch that provides improved performance in hierarchical systems.

The universal synchronization algorithm presented by Chuong, Ellen and Ramachandran in [20], is transaction friendly. Making a transaction-friendly version of L-Sim is left as future work; however, we believe that this can be easily achieved by applying similar techniques to those in [20]. The experimental analysis of L-Sim is also left as future work.

Since one of the goals of Sim is wait-freedom, each active thread executes all pending requests; this might be inefficient in terms of energy consumption. In contrast, in CC-Synch and in flat-combining, threads perform spinning until their requests have been executed (which seems to be less expensive in terms of resource usage). Measuring energy consumption is an interesting but not an easy task since several parameters (e.g., the time required to perform the computation, the resource usage, the way the thread library is implemented, etc.) should be considered. So, we leave this as future work.

In [8], Agathos, Kallimanis and Dimakopoulos present a highly efficient implementation of OpenMP tasks [51] for the OMPi OpenMP/C compiler [26]. This OpenMP tasking environment uses a work-stealing queue implementation based on CC-Synch. In synthetic benchmarks, OMPi achieves up to a 5 times better performance than other OpenMP task implementations, while for task-based real-world applications it achieves up to 87% better performance comparing to other OpenMP compilers [8]. We expect that the synchronization algorithms of the Synch and Sim families will be used in other practical applications as well.

Similarly to flat-combining [34], Sim and Synch synchronization algorithms have the same applicability limitations. Efficient implementations of data structures like search trees, where $m$ lookups can be executed in parallel by performing just a logarithmic number of shared memory accesses each, are expected to outperform Sim/Synch(since these synchronization algorithms perform each request sequentially like most previous universal constructions [20, 27, 34, 36, 37]). This limitation could be overcome by using multiple instances of them, as it is done in our queue implementation of Section 5.6 and the queue implementation of Section 6.5. For more complicated data structures, this will be part of our future work.

# Bibliography

[1] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, June 2010.

[2] Yehuda Afek, Hagit Attiya, Arie Fouren, Gideon Stupp, and Dan Touitou. Long-lived renaming made adaptive. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, pages 91–103, 1990.

[3] Yehuda Afek, Pazi Boxer, and Dan Touitou. Bounds on the shared memory requirements for long-lived & adaptive objects. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, pages 81–89, 2000.

[4] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proceedings of the 27th ACM Symposium on Theory of Computing*, pages 538–547, 1995.

[5] Yehuda Afek, Michael Merritt, and Gadi Taubenfeld. The power of multi-objects. *Information and Computation*, 153:213–222, 1999.

[6] Yehuda Afek, Michael Merritt, Gadi Taubenfeld, and Dan Touitou. Disentangling Multi-object Operations. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, pages 262–272, 1997.

[7] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived Adaptive Collect with Applications. In *Proceedings of the 40th Symposium on Foundations of Computer Science*, pages 262–272, 1999.

[8] Spiros N. Agathos, Nikolaos D. Kallimanis, and Vassilios V. Dimakopoulos. Speeding up OpenMP tasking. In *Euro-Par 2012 Parallel Processing*, pages 650–661. Springer, 2012.

[9] Gene Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings*, pages 483–485, 1967.

[10] James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 184–193, 1995.

[11] James H. Anderson and Mark Moir. Universal Constructions for Large Objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317–1332, dec 1999.

[12] James Aspnes and Maurice Herlihy. Fast, Randomized Consensus Using Shared Memory. 11(2):441–461, September 1990.

[13] Hagit Attiya and Arie Fouren. Adaptive and Efficient Wait-Free Algorithms for Lattice Agreement and Renaming. *SIAM Journal on Computing*, 31(2):642–664, 2001.

[14] Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *Journal of the ACM (JACM)*, 50:444–468, July 2003.

[15] Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. Partial snapshot objects. In *Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 336–343, 2008.

[16] Hagit Attiya, Nancy Lynch, and Nir Shavit. Are Wait-free Algorithms Fast? *Journal of the ACM (JACM)*, 41(4):725–763, July 1994.

[17] Greg Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.

[18] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, 2000.

[19] Tushar Deepak Chandra, Prasad Jayanti, and King Tan. A polylog time wait-free construction for closed objects. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing*, pages 287–296, 1998.

[20] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 335–344, 2010.

[21] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Blade Computing with the AMD Opteron Processor (Magny-Cours). *Hot chips 21*, August 2009.

[22] Intel Corporation. *Intel(R) 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part1*, January 2011.

[23] Travis S. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Department of Computer Science, University of Washington, February 1993.

[24] Dave Dice, Virendra J. Marathe, and Nir Shavit. Flat-Combining NUMA Locks. In *Proceedings of the 23nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 65–74, 2011.

[25] Dave Dice, Virendra J. Marathe, and Nir Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2012.

[26] Vassilios V Dimakopoulos, Elias Leontiadis, and George Tzoumas. A portable c compiler for openmp v. 2.0. In *Proceedings of the European Workshop on OpenMP (EWOMP'A03), Aachen, Germany*, 2003.

[27] Panagiota Fatourou and Nikolaos D. Kallimanis. The RedBlue Adaptive Universal Constractions. In *Proceedings of the 23rd International Symposium on Distributed Computing*, pages 127–141, 2009.

[28] Panagiota Fatourou and Nikolaos D. Kallimanis. A Highly-Efficient Wait-Free Universal Construction. In *Proceedings of the 23nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 325–334, 2011.

[29] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2012.

[30] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 64–75, April 1989.

[31] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. *IEEE Trans. Computers*, 32(2):175–189, 1983.

[32] Rajiv Gupta and Charles R. Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 54–63, 1989.

[33] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. The Code for Flat-Combining. http://github.com/mit-carbon/Flat-Combining.

[34] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 355–364, 2010.

[35] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the 16th ACM Symposium on Parallel Algorithms and Architectures*, pages 206–215, 2004.

[36] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13:124–149, jan 1991.

[37] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, nov 1993.

[38] Maurice Herlihy, Victor Luchangco, and Mark Moir. Space and Time Adaptive Non-blocking Algorithms. *Electronic Notes in Theoretical Computer Science*, 78, 2003.

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

[40] Damien Imbs and Michel Raynal. Help When Needed, But No More: Efficient Read/Write Partial Snapshot. In *Proceedings of the 23rd International Symposium on Distributed Computing*, pages 142–156. Springer, 2009.

[41] Prasad Jayanti. A lower bound on the local time complexity of universal constructions. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing*, pages 183–192, 1998.

[42] Prasad Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing*, pages 201–210, 1998.

[43] Prasad Jayanti. F-arrays: implementation and applications. In *Proceedings of the 21th ACM Symposium on Principles of Distributed Computing*, pages 270–279, 2002.

[44] Prasad Jayanti and Srdjan Petrovic. Efficient Wait-Free Implementation of Multi-word LL/SC Variables. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pages 59–68, 2005.

[45] Nikolaos D. Kallimanis and Panagiota Fatourou. The Code for Sim Universal Construction. http://code.google.com/p/sim-universal-construction/.

[46] Victor Luchangco, Daniel Nussbaum, and Nir Shavit. A Hierarchical CLH Queue Lock. In *Proceedings of the 12th International Euro-Par Conference*, pages 801–810, 2006.

[47] Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 165–171, 1994.

[48] Paul E. McKenney. *Memory Barriers: a Hardware View for Software Hackers*, June 2010.

[49] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[50] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 267–275, 1996.

[51] ARB OpenMP. Openmp application program interface, v. 3.1, 2008.

[52] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA '99)*, pages 182–204, 1999.

[53] Zoran Radovic and Erik Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the 9th IEEE International Symposium on High-Performance Computer Architecture*, pages 241–252, 2003.

[54] Ori Shalev and Nir Shavit. Predictive log-synchronization. In *EuroSys*, pages 305–315, 2006.

[55] Nir Shavit and Asaph Zemach. Diffracting Trees. *ACM Transactions on Computer Systems*, 14(4):385–428, 1996.

[56] Nir Shavit and Asaph Zemach. Combining Funnels: A Dynamic Approach to Software Combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 2000.

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

[58] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

[59] David L. Weaver and Tom Germond. *The SPARC Architecture Manual, Version 9*, 1994.

[60] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, 36(4):388–395, 1987.

# Author's Publications

1. Panagiota Fatourou and Nikolaos D. Kallimanis, "A Highly-Effcient Wait-Free Implementation of a Universal Object", Theory of Computing Systems (TOCS). Special issue of SPAA 2011. **In press.**

2. Spiros N. Agathos. Nikolaos D. Kallimanis and Vassilios V. Dimakopoulos, "Speeding Up OpenMP Tasking", In proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par 2012), pp. 650-661, Rhodes, Greece, August 2012.

3. Panagiota Fatourou and Nikolaos D. Kallimanis, "Revisiting the Combining Synchronization Technique", In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2012), pp. 257-266, New Orleans, LA, USA, February, 2012.

4. Panagiota Fatourou and Nikolaos D. Kallimanis, "A Highly-Efficient Wait-Free Universal Construction", In Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2011), pp. 325-334, San Jose, California, USA, June 2011. **Invited to Theory of Computing Systems as special issue of SPAA 2011.**

5. Panagiota Fatourou and Nikolaos D. Kallimanis, "The RedBlue Adaptive Universal Constructions", In Proceedings of the 23rd International Symposium on Distributed Computing (DISC 2009), pp. 127-141, Elche/Elx, Spain, September 2009.

6. Panagiota Fatourou and Nikolaos D. Kallimanis, "Time-Optimal, Space-Efficient Single-Scanner Snapshots & Efficient Multi-Scanner Snapshots using CAS", In Proceedings of the 26th Annual ACM SIGACT-SIGOPS Symposium on Principles of

Distributed Computing (PODC 2007), pp. 33-42, Portland, Oregon, USA, August 2007.

7. Panagiota Fatourou and Nikolaos D. Kallimanis, Single-Scanner Multi-Writer Snapshot Implementations are Fast!", In Proceedings of the 25th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2006), pp. 228-237, Denver, Colorado, USA, July 2006.

# Short Vita

Nikolaos Kallimanis was born in Amalias, Greece in 1983. He received his PhD in 2013 from the Department of Computer Science at the University of Ioannina. He entered with the 3rd best grade (after participating in the national exams) among the students that entered at the Department of Computer Science of University of Ioannina in 2001 and he obtained his Degree in 2005. He also obtained his MSc in 2007 from the same department. During his undergraduate and postgraduate studies he received scholarships and awards. Specifically, he received a scholarship and the best student award during the 3rd year of his undergraduate studies from Greek State Scholarship Foundation (IKY). His PhD studies were supported by a scholarship from Empirikion Foundation.

His research interests focus on theoretical and practical aspects on parallel and distributed computing, with emphasis in the design and analysis of concurrent data structures. He has published papers in top tier conferences in the domain of distributed and parallel computing, such as ACM PODC, ACM SPAA, ACM PPoPP and DISC.