Thesis for the degree of Doctor of Philosophy

Algorithms for synchronization and consistency in concurrent system services

Anders Gidenstam

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, 2006

Algorithms for synchronization and consistency in concurrent system services
Anders Gidenstam

ISBN 91-7291-832-2

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie Nr 2514
ISSN 0346-718X

Technical Report 20D
ISSN 1653-1787
Department of Computer Science and Engineering
Research group: Distributed Computing and Systems

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden

Telephone + 46 (0)31- 772 1000

Printed at Chalmers Reproservice
Göteborg, Sweden, 2006

Abstract

Synchronization, consistency and scalability are important issues in the design of concurrent computer system services. In this thesis we study the application of optimistic and scalable methods in concurrent system services. In a distributed setting we study scalable tracking of the causal relations between events, lightweight information dissemination in optimistic causal order in distributed systems and fault-tolerant and dynamic resource sharing. Further, we study scalable memory allocation, memory reclamation, threading, thread synchronization and data structures in shared memory systems. For each of the services we study we give the design of algorithms using optimistic methods, assess the correctness and analyze the behaviour of the algorithm, and in most cases describe implementations and perform experimental studies comparing the proposed algorithms to "traditional" approaches.

We present a study of the accuracy of plausible timestamps for scalable event tracking in large systems. We analyze how these clocks may relate causally independent event pairs and based on the analysis we propose two new clock algorithms to satisfy the analysis criteria. We propose an information dissemination service providing optimistic causal order called lightweight causal cluster consistency. It offers scalable behaviour, low message size overhead and high-probability reliability guarantees for e.g. multi-peer collaborative applications. A key component in the dissemination service is a dynamic and fault-tolerant cluster management algorithm, which manages a set of tickets/resources such that each ticket has at most one owner at a time. In the dissemination service this algorithm manages senders and enables the use of small fixed size vector clocks. We present a lock-free concurrent memory allocator, NBmalloc, designed to enhance performance and scalability on multiprocessors which also shows in our experimental evaluation. We present a lock-free memory reclamation algorithm for use in the implementation of lock-free data structures. Our algorithm is the first practical one that has all the following features: (i) guarantees the safety of local as well as global references, (ii) provides an upper bound of deleted but not yet reclaimed nodes, (iii) is compatible with standard memory allocation schemes. We also present LFthreads, a user-level thread library that is implemented entirely using lock-free methods aiming for increased scalability and efficiency over traditional thread libraries.

Keywords: synchronization, logical clocks, plausible clocks, time stamping system, event ordering, group communication, optimistic causal order, non-blocking, lock-free, memory management, memory reclamation, threading, atomic registers.

List of Included Papers and Reports

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

1. Anders Gidenstam and Marina Papatriantafilou, Adaptive Plausible Clocks, *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, pages 86 – 93, IEEE Press, 2004.
2. Andreas Larsson, Anders Gidenstam, Phuong H Ha, Marina Papatriantafilou and Philippas Tsigas, Multi-word Atomic Read/Write Registers on Multiprocessor Systems, *Proceedings of the 12th Annual European Symposium on Algorithms (ESA 2004)*, Lecture Notes in Computer Science Vol. 3221, pages 736 – 748, Springer-Verlag, 2004.
3. Anders Gidenstam, Boris Koldehofe, Marina Papatriantafilou and Philippas Tsigas, Lightweight Causal Cluster Consistency, *Proceedings of the Conference on Innovative Internet Community Systems (IICS 2005)*, Lecture Notes in Computer Science Vol. 3908, pages 17 – 28, Springer Verlag, 2006.
4. Anders Gidenstam, Boris Koldehofe, Marina Papatriantafilou and Philippas Tsigas, Dynamic and fault-tolerant cluster management, *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing*, pages 237 – 244, IEEE Press, 2005.
5. Anders Gidenstam, Marina Papatriantafilou and Philippas Tsigas, Allocating memory in a lock-free manner, *Proceedings of the 13th Annual European Symposium on Algorithms (ESA 2005)*, Lecture Notes in Computer Science Vol. 3669, pages 329 – 242, Springer-Verlag, 2005.
6. Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell and Philippas Tsigas, Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting, *Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN 2005)*, pages 202 – 207, IEEE Computer Society, 2005.
7. Anders Gidenstam and Marina Papatriantafilou, LFthreads: A lock-free thread library or “Blocking without locking”, *Technical Report 2005:20*, Computer Science and Engineering, Chalmers University of Technology, 2005.

Acknowledgements

First of all, I want to thank my supervisor Marina Papatriantafilou, for her constant support and encouragement during these years. Without her and Philippas Tsigas I would probably not have embarked on this journey into research in the first place, and when I did, their encouragement, knowledge and advice are and have been invaluable to me.

I want to thank my friends and colleagues in the distributing computing and systems group: Niklas Elmqvist, Phuong Ha, Boris Koldehofe, Andreas Larsson, Elad Schiller, Håkan Sundell and Yi Zhang. You have made working here much more enjoyable and your helpful comments, advice and our discussions and collaboration have helped making the research work interesting and fruitful. Further, I would like to thank everyone at Computing Science for providing such a friendly and creative working environment.

I would also like to thank my faculty opponent Prof. Evangelos Kranakis from Carleton University, my examiner Prof. David Sands, the members of my grading committee, Prof. Gerth Stølting Brodal, Prof. Shlomi Dolev, Prof. Jan Jonsson and Prof. Peter Damaschke. Many thanks also to Prof. Graham Kemp and Prof. Björn von Sydow in my advisory committee for helpful discussions and comments during these years.

Further, I want to thank my good friends for their support, for making my life more fun and enjoyable and for keeping me away from computers and research for at least a couple of weeks a year with, among other activities, alpine skiing and visits to remote archaeological sites. Many thanks to all of you!

Finally, I want to thank my parents Karin and Ingemar and my brother Hans for their love, support and encouragement throughout my life.

Anders Gidenstam

Göteborg, August 2006.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction and background

In this thesis we study the issue of synchronization and consistency in concurrent computer system services. Any tasks that are undertaken in cooperation by a number of entities, let them be people or computers, require some form of synchronization. As we all experience in everyday life, synchronized cooperation often works best when communication between the cooperators is extensive and instantaneous, as, for example, when the cooperators are at the same location. When communication becomes more limited, synchronized cooperation becomes more difficult.

Synchronization is also closely tied to consistency, which is how well the participant's views of the task correspond to each other and to the global state of the system. Both synchronization and consistency influence the *scalability*, that is, how much faster the cooperative task can be solved when the number of collaborators is increased. Ideally, twice as many cooperators should be able to perform a task twice as fast, but this is not necessarily true for all problems and there is often an upper limit where additional collaborators do not decrease the time required to perform the task at all.

The main topic of this thesis is to study the effect of optimistic and fine-grained synchronization on several key system services. In a distributed message passing setting we study scalable tracking of the causal relations between events using logical clocks; in the same communication model we also study lightweight information dissemination in optimistic causal order and fault-tolerant and dynamic distributed cluster management. Further, in shared memory systems we study scalable and lock-free memory allocation, memory reclamation for use in lock-free data structures, multithreading and thread synchronization as well as lock-free data structures useful in the above services. For each of the services studied we give the design of algorithms using optimistic methods and fine-grained synchronization, assess the correctness and analyze the behaviour of the algorithm, and in most cases describe implementations and experimental studies comparing the proposed algorithms to “traditional” approaches.

This chapter is structured as follows: Sections 1.1 and 1.2 introduce the basics of concurrent computation and concurrent system models; Section 1.3 describes logical time and logical clocks; Section 1.4 describes group communication and message orderings; Section 1.5 introduces non-blocking synchronization and data structures; Section 1.6 describes system services and synchronization; and, finally, Section 1.7 summarizes the contributions of this thesis.

1.1 Concurrent computation

Concurrent computer systems can distinguished into two categories depending on the communication paradigm used, namely (i) *message passing systems* and (ii) *shared memory systems*, both of which will be discussed below. From an abstract point of view, a concurrent system consists of a number of sequential processes, we denote them $p_1, p_2, \ldots, p_n$, which each executes a sequence of steps, called *events*. We call the sequence of events executed by process p_i in an execution of the system the *local history* for that process and denote it $H_i = e_i^1 e_i^2 \ldots$. The order in which the events occur in the local history of a process p is called the *program order* of p. The whole execution of the system is called the *global history*, denoted H, and is a partially ordered set of events formed by the union of the local histories of the processes.

The events of an execution can be ordered according to the *causal precedence relation*, introduced by Lamport in [Lam78], and usually denoted by $\rightarrow$. It is also known as the *happened-before* or *could have influenced* relation. The causal precedence relation orders two events that occurred in the same process according to the program order and orders two events that occurred in different processes if there is some way the first of them could have influenced the second. That is, if some form of communication took place between the involved processes; for example in a message-passing system a message from the process of the first event sent after that event could be received by the other process before (or at) the second event; in a shared memory system the first process might write to a memory location that is later read by the second process. Events that are not related by the causal precedence relation are *concurrent*, which is denoted $a \| b$ for two events a and b.

The global history of an execution together with the causal precedence relation form a partially ordered set, representing the distributed computation and is often presented in a *time-space diagram*, as shown in Figure 1.1. The arrows between some event pairs in the figure represent the causal precedences introduced by these events.

The processes in a concurrent system are commonly assumed to be *asynchronous*, that is, there is no bound on how fast or slow each processor executes its steps. If, on the other hand, there is some bound on the processors’ absolute or relative speeds the system is said to be *synchronous*.

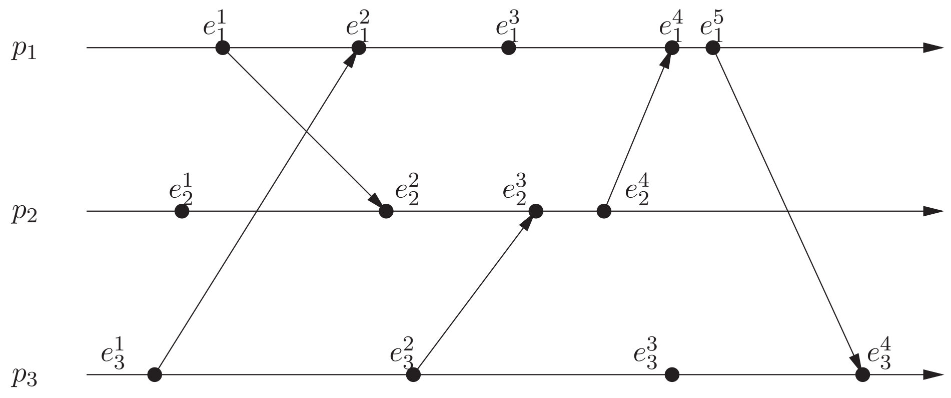


Figure 1.1: A time-space diagram of a distributed computation.

1.1.1 Scalability in concurrent systems

One important property of distributed systems and algorithms is *scalability*, that is, how well the system can handle an increased number of users and/or resources. Ideally, twice as many collaborators should be able to perform a task at least twice as fast, but this is not always true and there is often an upper limit where additional collaborators do not decrease the time required to solve a particular task at all. Consider, for example, the task of counting the number of cards in a standard deck of cards: going from one to two counters is likely to be about twice as fast, while going from 10 to 20 counters will most likely not be faster since the overhead of splitting and distributing the deck before the counting can start and the final summation of the counts will be high. However, if the deck contained 520 cards instead of 52 then it is likely that 20 counters would be faster than 10.

In shared memory systems a typical example would be that we have a concurrent program for solving some problem and want to solve large problem instances with it. The program is said to be scalable if the time needed to solve a problem instance is halved when we double the number of processors available to the program. In message passing systems scalability is often considered in terms of how the overhead, i.e. the number of messages needed and/or the message size, changes as the number of participating processes is increased.

Scalability is one of the major concerns in the development of concurrent and distributed algorithms as many systems and applications have a tendency to grow significantly over time, for example peer-to-peer systems and the Internet itself.

1.2 Concurrent system types

As mentioned above, concurrent computer systems can be distinguished based on the communication paradigm used.

A *message passing system* consists of a set of sequential processes (sometimes also called nodes) and a network or some other communication media that allows the processes to send messages to each other. This type of systems is often also physically distributed, that is, the processes are located at different locations.

A *shared memory system* consists of a set of sequential processes and a shared memory that all processes can read from and write to, thereby allowing them to communicate. While a distributed message passing system often is a group of separate computers working together, a shared memory system usually is one computer with several processors sharing the same memory. The key feature of such a computer, called a *shared memory multiprocessor*, is that processors share a single memory address space which they can read/write to [PH98].

We can view an execution on a multiprocessor system in almost the same formalism that we use for a message passing system by viewing a "send" event as a "write" to a shared memory location and a "receive" event as a "read" from the same location.

While most shared memory systems have special hardware for handling memory transactions between the interconnected processors and the memory, it is also possible to implement a shared memory abstraction in software on a pure message passing system, such as a set of workstations. This is called *distributed shared memory* (DSM) and there exist many different algorithms for it. Some examples are the algorithms by Li and Hudak [LH89] and Ahamad et al. [AHJ91]. The latency to access shared memory in a software DSM implementation running across an ordinary network may be orders of magnitude larger than the latency of memory accesses across the dedicated interconnect of a shared memory multiprocessor machine.

1.2.1 Timing in concurrent systems

Concurrent systems can be further classified into a large number of categories based on the properties of the processes and the communication media. The most important of these properties are timeliness properties and failure-models for processes and communication. In terms of its timeliness properties a concurrent system can be either *asynchronous* or partially to fully *synchronous*. A brief overview, from Turek and Shasha in [TS92], of common timeliness properties for processes is:

- **Synchronous processes**. The ratios between the processes' speeds are bounded, that is, they all make progress at approximately the same rate.

- **Asynchronous processes**. There are no bounds on the processes' absolute speed nor on the ratio between them.

A message passing system has *bounded message delay* if there is an upper bound on the time a message can be in transit, otherwise the system has *unbounded message delay.* Further, message delivery is *ordered* if messages transmitted on a communication channel are guaranteed to be received in the same order they were sent, or *unordered* otherwise. The communication system is *point-to-point* if each communication channel connects a pair of processes, or *broadcast* if a message sent on a channel can be received by many processes.

In shared memory systems the processes are commonly modeled as asynchronous, despite the fact that most real microprocessors are synchronous. One reason is that most general purpose operating systems use multiprogramming, which is when several processes share the same processor through time-sharing. The scheduling of the processes access to a processor is often preemptive, which together with other events that might delay a process, such as page-faults and blocking I/O, makes the progress rate of any particular process very difficult to predict. In some shared memory systems, for instance real-time systems, the processes are considered synchronous, but in these cases the whole system has to be designed and analyzed with great care so that the occurrence of events, such as those mentioned above are eliminated or predictable.

Memory access in shared memory systems

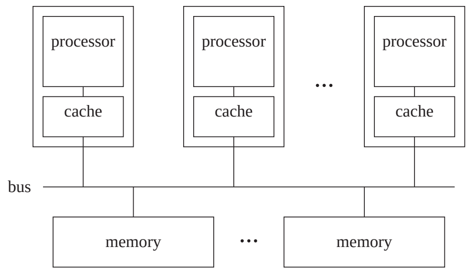
There are two main models for memory access latencies of shared memory multiprocessors, which originally stem from their organization at the hardware level. The first one is *uniform memory access* (UMA) and the other one is *non-uniform memory access* (NUMA).

In a *uniform memory access* machine (c.f. Figure 1.2(a)) all processors can access all parts of the memory with the same latency. Such machines were often organized much like single processor machines, but with several processors connected to a single memory bus instead of just one as a single processor machine has. Larger UMA machines may use an interconnect network to connect processors to the memory instead of a shared bus since a bus-based design usually does not scale well as the number of processors increases.

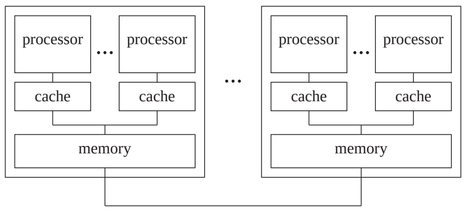
In a *non-uniform memory access* machine (c.f. Figure 1.2(b)) the processors experience different latencies when accessing different parts of the memory. In these machines the main memory and processors are often clustered in nodes, which are then in turn connected together by an interconnection network. In such a machine a processor will experience much higher memory latency when accessing memory located in another node compared to the latency of accessing the memory in its own node. Internally a NUMA (or UMA) system may closely resemble or even be a message passing system — its classification as a shared memory system is based on the programmer's view of the system.

Memory consistency in shared memory systems

Normally, when reasoning about concurrent programs we consider the shared memory to be *sequentially consistent* [Lam79], that is, the effects of all mem-



(a) The structure of a Uniform Memory Access computer.



(b) The structure of a Non-Uniform Memory Access computer.

Figure 1.2: The structure of uniform and non-uniform memory access systems.

ory accesses by the processes can be arranged into one global totally ordered sequence that is consistent with the program order of all processes.

Most actual multiprocessor machines, however, only give much weaker guarantees on the memory consistency, that is, the memory accesses from different processors can be interleaved in ways not consistent with sequential consistency. The reason for this is performance — by relaxing the ordering requirement it is possible to make memory access faster. Relaxing the ordering works well since usually the vast majority of all memory accesses done by a processor do not have to become visible to the other processors in some very specific order for the program to work as expected. For the memory accesses where the order they become visible to the other processors is important, these systems provide one or more *memory barrier* instructions which must be inserted into the program around these critical memory accesses.

Two good surveys of memory consistency models are given by Adve and Gharachorloo [AG96] and by Mosberger [Mos93].

1.2.2 Failure models in concurrent systems

The most common failure models for processes are:

- **No failures**. Processes cannot fail.
- **Stop-failures**. A faulty process ceases to execute steps and will remain stopped forever.
- **Byzantine failures**. A faulty process exhibits arbitrary behaviour, i.e. it can execute any program instead of the intended one.

For communication we have a similar set of failure models:

- **No failures**. The communication is reliable.
- **Omission failures**. Messages might disappear.
- **Byzantine failures**. The message content might be garbled and spurious messages might be created by the communication channel.

1.2.3 Hardware support for synchronization

A well-studied case of coordination which is a key to the synchronization in concurrent systems, both in shared memory and message passing ones, is the *Consensus Problem* [FLP85, LAA87, TS92]. In the consensus problem a set of processes each propose a value, usually 0 or 1, and then they all try to decide on the same one of the proposed values. A correct solution to the consensus problem needs to fulfill the following three conditions:

- **Consistency**. All the processes must agree on the same value and all decisions are final.
- **Validity**. The agreed-upon value must have been proposed by some process.
- **Termination**. Each process has to decide on a value within a finite number of steps.

The consensus problem is interesting because it is a basic building block for solutions to many concurrent problems. It has been shown [FLP85] that it cannot be solved in an asynchronous message passing or in a read/write shared memory system model [LAA87, Her91] even when only one process might fail. As a consequence it is often needed to make some assumptions other than total asynchrony to solve this and other problems that involve this type of decision.

To be able to solve the consensus problem in a distributed message passing system we need to add either some timeliness properties or stronger communication possibilities. There are three minimal cases where consensus is possible as identified by Dolev et al. in [DDS87]:

1. The processes are synchronous and the message delay is bounded.

2. The communication is done by totally ordered broadcast.
3. The processes are synchronous and point-to-point messages are ordered.

In shared memory systems it is common to assume the availability of atomic primitives with stronger synchronization capabilities than just atomic reads and writes to shared memory and which therefore can solve the consensus problem by themselves, instead of limiting the asynchrony of the system model.

The inability to solve the consensus problem in an asynchronous system with shared memory that supports only reads and writes severely limits the possibilities to write useful concurrent programs. Therefore most multiprocessor systems, in addition to reads and writes, also support one or more stronger hardware synchronization primitives. Such primitives allow a number of processes to reach *agreement*, that is, to solve the aforementioned *consensus problem* even when the system is viewed as being fully asynchronous. However, as shown by Herlihy in [Her91], the strength of these synchronization primitives, called their *consensus number*, which is the maximum number of processes they can solve the consensus problem for when processes are allowed to fail, differs. Synchronization primitives that have unbounded consensus number are called *universal*.

A selection of the more common synchronization primitives is presented in Figure 1.3. They are:

- *Test_And_Set* (TAS), which in one atomic step sets the value of a memory location to one and returns the previous value of that location;
- *Fetch_And_Add* (FAA), which atomically increments the value of a memory location and return the previous value;
- *Compare_And_Swap* (CAS), which in one atomic step replaces the current value of a memory location with a new value if and only if that current value is equal to the expected old value; and
- *Load_Linked* / *Store_Conditional* (LL/SC), which work in the following way: *Store_Conditional* conditionally writes a new value to a word in memory. The write will succeed if and only if this process previously read the location using *Load_Linked* and no other write to this location has occurred since the *Load_Linked*.

Test_And_Set and *Fetch_And_Add* have consensus number two, while *Compare_And_Swap* and *Load_Linked*/*Store_Conditional* are universal, that is, their consensus number is unbounded. By using a hardware synchronization primitive it is possible to implement other synchronization primitives with the same or lower consensus number in software, see for example Jayanti [Jay98] and Moir [Moi97].

While *Compare_And_Swap* and the other primitives usually only work on one memory location at a time, the designer of non-blocking algorithms often finds her/himself wanting to update several memory locations in one atomic step.

A few now outdated systems, like those based on the Motorola MC68020 and above processors, supported a *double-word* *Compare_And_Swap* (DCAS) primitive that worked on two separate memory locations, but unfortunately no current system supports similar primitives. It is, however, possible to implement a *multi-word* *Compare_And_Swap* primitive in software at the expense of more overhead. Such algorithms have been proposed by Harris in [HFP02] and Ha and Tsigas in [HT03].

Another way to support atomic updates of more than one memory location at a time is *Transactional Memory* introduced by Herlihy and Moss in [HM93]. Transactional memory allows processes to prepare transactions, i.e. sets of memory reads and writes that will take effect atomically (with respect to other transactions) when the transaction is committed (or not at all if the transaction is aborted). Access to such transactions could simplify the implementation of lock-free data structures significantly. However, while transactional memory can be implemented in hardware most current processors only support software implementations, which implies considerable overhead. Several algorithms implementing software transactional memory have been proposed, for example the lock-free ones by Shavit and Touitou [ST95] and by Fraser [Fra04].

A related issue to that of updating several memory words at once is the size of the memory words the primitives work on. In most systems the atomic synchronization primitives supported by the hardware work on memory words of the systems native size, usually either 32 or 64 bits. Some 32 bit systems, for example Intel IA32 and Motorola PowerPC, however, also support atomic primitives on long words (64 bit), which gives the algorithm designer some extra possibilities on these systems.

The *Compare_And_Swap* primitives mentioned above have an inherent weakness called the *ABA problem* which complicates their use. The problem is that the *Compare_And_Swap* primitive cannot detect if the target variable has been changed from the expected old value A to some other value B and then back to A again by concurrent processes, so the *Compare_And_Swap* will succeed even though this is likely to be undesirable as the "new" value it commits might be obsolete.

1.3 Logical time and logical clocks

In an asynchronous system there is no notion of a common global time available to the processes, so if one needs some ordering to tell how events in the system are related to each other, one needs to use some form of *logical time*. The concept of logical time was introduced by Lamport in [Lam78], where he also presented an algorithm for establishing a total order of all events in the system.

We call that kind of distributed algorithm a *Logical Clock Algorithm* or a *Time-Stamping System*. A logical clock algorithm P consists of the following parts:

1. *Clocks.* Each process has a local data structure called a *clock* which has the operations local-update, receive-message-update and read defined.

```
function Test_And_Set (address : pointer to word)
    return word
begin atomic
    ret := *address;
    if tmp = 0 then
        *address := 1;
    return tmp;
end Test_And_Set;

function Fetch_And_Add (address : pointer to integer;
            increment : integer) return integer
begin atomic
    ret := *address;
    *address := ret + increment;
    return ret;
end Fetch_And_Add;

function Compare_And_Swap (address : pointer to word;
            oldvalue : word; newvalue : word) return boolean
begin atomic
    if *address = oldvalue then
        *address := newvalue;
        return true;
    else return false;
end Compare_And_Swap;
```

Figure 1.3: The synchronization primitives *Test_And_Set*, *Fetch_And_Add* and *Compare_And_Swap*.

2. *Timestamps.* A timestamp is a value read from a clock. Every event has a timestamp associated with it when the event occurs. Moreover, for a "send" event, the timestamp is also attached to the sent message.

3. An ordering relation $\xrightarrow{P}$ for timestamps which is transitive and irreflexive and defines a strict partial order over all timestamps.

A time-stamping system is *plausible*, as defined by Torres-Rojas and Ahamad in [TRA99], if its ordering of the timestamps of the events in an execution is *compatible* with the causal order of the events themselves. More formally we have $\forall a, b \in H$: (i) $P(a) \overset{P}{=} P(b)$ iff $a \equiv b$; and (ii) if $a \rightarrow b$ then $P(a) \xrightarrow{P} P(b)$, where a and b are events in an execution H and $P(x)$ denotes the timestamp assigned to the event x by the algorithm. A time-stamping system *characterizes causality* if $\forall a, b \in H$: (i) $P(a) \overset{P}{=} P(b)$ iff $a \equiv b$; (ii) $a \rightarrow b$ iff $P(a) \xrightarrow{P} P(b)$; (iii) $a \| b$ iff $P(a) \overset{P}{\|} P(b)$.

1.3.1 Lamport Clock

The *Lamport Clock*, introduced by Lamport in [Lam78], is a plausible time-stamping system that produces a total order that is compatible with the causal order over all events in an execution.

The clocks in the algorithm consist of one local integer counter L_i for each process i. Initially, all L_is are set to 0. Each process i increases its local counter L_i when an event occurs at the process. A process i attaches the current value of L_i to each message it sends as a timestamp. When a process j receives a message from process i it updates its clock L_j to one plus the maximum of its previous value and the timestamp attached to the message. The ordering between events can be determined by comparing their timestamps, that is, the integer value of the local clock when they occurred. If two timestamps have the same integer value, the process ids of the source processes are used to break ties.

The ordering produced by the Lamport Clock is a total order compatible with the causal order. That is, events that are causally ordered will be ordered in the same way by the Lamport Clock. However, in addition the Lamport Clock will also order all concurrent events.

1.3.2 Vector Clock

The *Vector Clock*, which was discovered independently by Mattern [Mat89] and Fidge [Fid88, Fid91], is a plausible time-stamping system that *characterizes* causality. That is, by using vector clock timestamps the exact causal relation between any two events in the system can be determined.

The clocks and timestamps are both integer vectors with one entry for each process in the system (c.f. Figure 1.4(a)). This means that the timestamps become rather large as the number of processes in the system increases. This is a performance problem since a timestamp has to be attached to every sent message. However, Charron-Bost proved in [CB91] that any timestamping system that characterizes causality must use timestamps whose size is linear in the number of processes in the system.

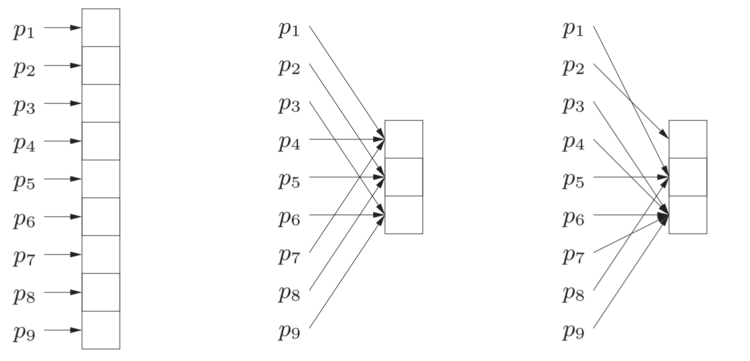
In practice it is sometimes possible to reduce the amount of information that needs to be attached to each message in order to maintain the vector clocks. For example, if the communication channels guarantee ordered (FIFO) message delivery then the compression scheme by Singhal and Kshemkalyani [SK92] can be used. It reduces the size of timestamps by sending only the clock values that have changed since the last message to the recipient. Another compression method that works even when the communication channels are unordered was introduced by Hélary et al. in [HRMB03]. The basic idea of this method is to avoid sending the clock entries whose value we know to be less than or equal to the value of the corresponding entry in the recipients clock. This can be achieved by using an auxiliary data structure in each node to keep track of which entries that would be new to what process. Common to both these compression schemes is that the reduction of the timestamp sizes depends on the execution history of the system and that there are worst-case scenarios where the timestamp sizes cannot be reduced.

One application area for vector clocks is in *causal broadcast protocols* (Birman et al. [BSS91]), that is, protocols that provide an any-to-all message service that guarantee that causally related messages are always delivered in the right order to a group of processes. However, also for these protocols their scalability is decreased by the growing size of the vector clock timestamps. One way to improve scalability, which is proposed in Chapter 3 of this thesis, is to use a more restricted protocol that allows only a subset (which may change over time) of the processes to create broadcast messages concurrently. This makes it possible to perform causal broadcast using a smaller vector clock whose size only depends on the maximum allowed number of concurrently sending processes and not on the total number of processes in the system.

1.3.3 Plausible Clocks

For many applications it is not necessary to have a timestamping system that characterizes causality. Instead one that is plausible, that is produces an ordering compatible with the causal order, is enough. This opens up the possibility to use timestamping systems that use much smaller timestamps than Vector Clocks. This is important in large systems where the overhead of using Vector Clocks would be high. Applications where such scalable plausible clocks are an attractive option are, for example, causal consistency protocols for distributed objects, such as those described by Torres-Rojas et al. in [TRAR98] and [TRA99], where plausible clocks are used to maintain a local cache of causally consistent object values at each process, and causal distributed memory protocols, such as the one by Ahamad et al. in [AHJ91].

The Lamport clock mentioned above is one such timestamping system with small timestamps. However, as it produces a total order, it does not work well in applications that benefit performance-wise by being able to detect also when



(a) Vector clock. (b) R-Entries vector clock. (c) NUREV clock.

Figure 1.4: Examples of the timestamps from different time-stamping systems.

a pair of events is concurrent.

As mentioned above, Torres-Rojas and Ahamad introduced the name *plausible clocks* for this class of timestamping systems in [TRA99], where they also proposed a number of plausible clock algorithms which have timestamps of constant size. The most significant of the clock algorithms they proposed is the *R-Entries Vector Clock*, which is a similar to a Vector Clock but whose clock vector has a fixed number (R) of entries. Each entry in the clock vector is shared by the processes whose ids are congruent modulo R (c.f. Figure 1.4(b)). The R-entry Vector Clock shows good ordering accuracy even for small timestamp sizes.

In Chapter 2 we extend the concept of the R-entry Vector Clock to introduce a new class of fixed size Vector Clocks, called *Non-Uniformly Mapped R-Entries Vector (NUREV) Clocks*, which allow the association between process and clock vector entry to change during the execution (c.f. Figure 1.4(c)). This makes it possible to construct clock algorithms that achieve very good ordering accuracy even at small timestamp sizes. Two such algorithms are also presented.

1.4 Group communication

In distributed applications it is not uncommon to have groups of processes that need to communicate and share information with each other. Such communication can be of the *unicast* type, i.e. messages are sent point-to-point from sender to receiver, or *multicast*, where each message may have several destinations.

Group communication is a form of multicast communication for a group of processes where every process may send messages and each message should reach all processes in the group. This type of communication is particularly useful in distributed applications where the processes cooperate to maintain some shared state, for example, replicated shared objects. This has various types of applications, examples thereof are services implemented by a set of replicas to enhance reliability and/or availability and distributed cooperative applications, such as virtual environments or games, where a set of geographically distributed users want to interact through the shared state of the application.

Protocols for group communication are sometimes called *broadcast protocols*, because if one only considers the processes inside the group, then each message should reach all processes.

Broadcast protocols can have a variety of properties with respect to guarantees, resource demands and assumptions on the properties provided by the underlying system, for example bounded message delays. It is common to distinguish the guarantees provided by a broadcast protocol into *reliability*, *ordering* and *timeliness* guarantees. Reliability guarantees, which concern message delivery, can range from best effort to the strong guarantees of *Reliable Broadcast* and *Uniform Reliable Broadcast*.

A Reliable Broadcast satisfies these three properties [HT93]:

- **Validity.** If a correct process broadcasts a message m, then all correct processes eventually deliver m.
- **Agreement.** If a correct process delivers a message m, then all correct processes eventually deliver m.
- **Integrity.** For any message m, every correct process delivers m at most once, and only if some process broadcasts m.

Under the Reliable Broadcast properties a faulty process could do bad things, like delivering a message twice or delivering a phony message that was never broadcast. Uniform Reliable Broadcast avoids these problems by strengthening the Agreement and Integrity properties to also include faulty processes:

- **Uniform Agreement.** If a process (correct or faulty) delivers a message m, then all correct processes eventually deliver m.
- **Uniform Integrity.** For any message m, every process (correct or faulty) delivers m at most once, and only if some process broadcast m.

The Reliable Broadcast properties do not specify in what order the processes deliver the messages, so each process is free to deliver in any order. This is not a desirable behaviour in some applications, e.g. distributed replication where it would be much more convenient if all processes delivered the messages in the same order. Consequently, a number of different message delivery ordering properties have been defined for broadcast protocols. The more common are [HT93]:

- **FIFO order.** If a process broadcasts a message m before it broadcasts a message m', then no correct process delivers m' unless it has previously delivered m.

- **Causal order.** If the broadcast of a message m causally precedes the broadcast of a message m', then no correct process delivers m' unless it has previously delivered m.
- **Total order.** If correct processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

The total order property ensures that all processes deliver the same set of messages in the same order. Such broadcast protocols are also known as *atomic* broadcasts. This property, in itself, does not guarantee that the protocol provides FIFO or causal order, but a broadcast protocol may provide a combination, such as causal atomic broadcast.

In [BJ87] Birman et. al. presented a set of protocols for reliable broadcast called *virtual synchrony*, which formed the basis for the successful group communication systems ISIS and Ensemble. In the latter system the protocols are implemented as layers and the application can choose and assemble layers into protocol stacks to achieve the desired properties, e.g. causal atomic broadcast. Protocols implementing reliable causal broadcast have been proposed in e.g. [BSS91, RST91, KS98].

1.4.1 Gossiping

The strong reliability guarantees in the presence of faults of traditional reliable broadcast protocols are not for free — they come at a considerable overhead in terms of limited scalability and potential message delivery latency.

Recent approaches for information dissemination focus on large scale systems with *peer-to-peer (P2P) communication.* These systems are usually considered to have both a very large number of participants/processes and also an almost continuous stream of processes joining and leaving the system. Both of these factors make the use of traditional reliable broadcast in such systems problematic. Instead, lightweight probabilistic group communication protocols, which allow groups to scale to many processes by providing reliability expressed with high probability, have been proposed for use in these systems.

One such lightweight information dissemination technique is *gossiping*, which is based on inspiration from nature, namely, of how a rumour spreads among a group of people or how a disease spreads in a susceptible population. Gossiping uses a flat unstructured communication model where each process knows a potentially dynamic subset of the group members, called its *view.*

Gossiping does not provide the strong guarantees of reliable broadcast (as discussed above) but aim for *predictable reliability* which guarantees that a message reaches all non-faulty processes with *high probability*, that is, with probability at least $1 - O(n^{-c})$ for some constant c.

Gossiping as a distributed systems concept was introduced by Demers et al. in [DGH$^+$87], where it was used for data replication. Birman et al. in [BHO$^+$99] applied gossiping as the basis of the probabilistic broadcast protocols *pbcast* and *bimodal multicast.* Since then many more gossip-based protocols have emerged, such as *lpbcast* [EGH$^+$01], *SCAMP* [GKM01] and [BEG04]. In [HKMP95]

Hromkovic et al. surveys results from deterministic analysis of gossiping and other broadcast algorithms in a set of different graph topologies.

1.4.2 Optimistic message ordering in group communication

As we have seen above, the reliability and ordering properties of traditional reliable broadcast are very strong. This might have the drawback of slowing down the whole system even if only a small subset of the processes is slow. In particular, given the definition of causal order broadcast above, even a single lost or late message prevents a process from delivering any of the causally succeeding messages that it might receive until the missing message has been recovered. For the sake of efficiency and/or resource constraints it may make sense for some applications to skip a missing message rather than to let it delay or prevent the delivery of newer messages as they arrive. Some example application domains include distributed audio and/or video, distributed monitoring of systems where newer operations overwrite the preceding one. This type of message ordering is called *optimistic causal order* and is defined as follows:

- **Optimistic causal order.** If the broadcast of a message m causally precedes the broadcast of a message m', then a correct process delivers m if it has not already delivered m'.

A protocol that provides optimistic causal order should preferably also provide predictable reliability, that is, that a message is delivered to all non-faulty processes with high probability.

Protocols that provide optimistic causal order for broadcast messages have been presented by Baldoni et al. in [BPRS98] and broadcast as well as point-to-point messages by Rodrigues et al. in [RBAR00]. Their protocols assign a real-time deadline to each message, which specifies how long a process will wait for any missing causally preceding messages to show up before delivering the current message anyway. To track the causal order among messages the protocols attach dependency information to each message, for [BPRS98] the worst-case size of this information is linear in the number of processes in the group, while for [RBAR00] it is quadratic (due to the protocol's support for point-to-point messages).

In Chapter 3 we present a protocol that implements optimistic causal order for broadcast messages and that reduces the dependency information in each message to a constant size by means of an upper bound on the number of processes that may send a broadcast message concurrently.

1.5 Concurrent data access in shared memory systems

In a shared memory system the processes[1] have access to a set of shared memory locations which they may use to communicate. A process can *read* data from and *write* data to each shared memory location. The number of processes can be much larger than the number processors due to multiprogramming, which may interleave the execution of several processes on the same processor. The processes are often considered to be asynchronous, that is, their rate of execution might vary arbitrarily, because of the interleaving. This has certain implications for the possibilities for the synchronization and coordination of processes which we will discuss below.

1.5.1 Linearizability

We want the semantics of all operations on a shared data object to be the same as for the same operation on the corresponding sequential object. The consistency model that captures this is called *linearizability* and was introduced by Herlihy and Wing in [HW90]. Linearizability requires that for each operation, in a concurrent execution of operations on the shared data object, there is an atomic time instant that lies within its duration where the operation takes effect, in a way such that the outcome of the operation agrees with the object's sequential specification.

1.5.2 Lock-based synchronization

The traditional way to synchronize processes/threads accessing a shared data object in a concurrent program is to use *mutual exclusion*. Mutual exclusion is normally implemented using a lock, which is a shared variable together with routines to atomically *acquire* and *release* the lock. The atomicity of *acquire* and *release* guarantees that only one process can acquire and hold the lock at a time. The most common approach when synchronizing using locks is to use the lock to implement *critical sections*, that is, some pieces of code that can only be run by one process at the time. For a shared data object, it is common that the operations it supports are implemented as mutually exclusive critical sections.

The use of locks and the sequential nature of critical sections cause a number of drawbacks, namely:

- **Deadlock prone.** With locks it is not hard to create circular lock dependencies that cause two (or more processes) to get blocked by both trying to acquire a lock that is held by the other. Furthermore, a process that crashes while holding some lock(s) is also likely to block the progress of other processes.

[1]We will use the term process and thread interchangeably in the context of general shared memory synchronization. If we talk about threads and processes in the operating system sense it will be made clear from context.

- **Blocking.** The process that has acquired the lock will delay all other processes that also need that lock until it has finished executing inside the critical section. To make matters worse the process inside the critical section may itself be delayed by being preempted by the scheduler, suffer a page-fault, try to acquire another lock or wait for IO inside the critical section.

- **Priority inversion.** This is a pathological case that can occur when using a strict priority based scheduler, where a medium priority process can delay a high priority process, potentially indefinitely on a single processor system, by preempting a low priority process that has acquired a lock needed by the high priority process. This problem can be avoided by employing *priority inheritance protocols* as proposed by Sha et al. [SRL90].

1.5.3 Non-blocking synchronization

Non-blocking synchronization techniques avoid the use of locks by using cunning algorithms, which often but not always use hardware synchronization primitives, to create shared data objects that can be accessed simultaneously by several processes. By avoiding locks non-blocking synchronization does not exhibit the problems of deadlocks, blocking and priority inversion, which lock-based synchronization suffers from. Non-blocking shared data objects also have a higher degree of fault-tolerance than lock-based ones since they can tolerate any number of processes experiencing stop-failures.

There are two kinds of non-blocking synchronization, *lock-free* synchronization and the stronger *wait-free* synchronization.

Lock-free synchronization

A *lock-free* algorithm guarantees that regardless of the contention caused by concurrent operations and the interleaving of their steps, at each point in time there is at least one operation which is able to make progress. However, as there is no fairness guarantee, some operation could be starved and take unbounded time to finish.

The lack of fairness guarantee significantly simplifies the construction of lock-free algorithms compared to wait-free ones and leads to algorithms that are fast when there are no conflicts but cause slow down for all except one process involved in a conflict. Hence, lock-free synchronization is also known as *optimistic synchronization* [Rin99].

In [Her93] Herlihy described a general method for transforming any sequential data object implementation to a lock-free shared data object implementation. In short, the methodology is like this: The state of the shared data object is represented by a pointer to the current version; an operation on the shared data object first makes a new private copy of the current version, applies the sequential version of the desired operation on the private copy and thus creates a new prospective state of the shared object. Then it tries to install this prospective state as the new version of the shared object using an atomic synchronization primitive that will only succeed if the current version of the shared object is still the same as the one the new state was computed from. If the operation fails to install its new state, some other operation(s) have managed to install their new versions and this operation has to retry from the beginning.

This general methodology is often not very efficient because (i) the entire object is copied for each update (this can be optimized though) and (ii) the resulting lock-free shared object is not *disjoint-access parallel*, that is, all concurrent operations on it cause conflicts even when the operations only access disjoint parts of the shared object.

For the above reasons, a significant research effort is being spent on the task of designing and developing efficient lock-free implementations of various data structures.

The use of lock-free instead of lock-based synchronization can give significant performance gains in parallel applications, as shown by Tsigas and Zhang in [TZ01a, TZ02], as well as in operating systems, for example as suggested by Greenwald and Cheriton in [GC96].

Wait-free synchronization

A *wait-free* algorithm is both lock-free and *fair*, it guarantees that every operation finishes in a bounded number of its own steps, regardless of the actions of other operations. This is a very strong property, as it decouples the processes using the same shared data object from each other. This makes wait-free shared data objects attractive to use, for example, in hard *real-time systems* where the worst-case execution time has to be known for every operation and where lock-based critical sections limit the schedulability of the system and complicate the schedulability analysis. A drawback, however, is that algorithms that are wait-free, are often also quite complex, in particular for non-trivial shared objects.

A common approach in implementing wait-free algorithms is the use of *helping schemes* [Her91]. In a helping scheme each operation first announces information about what it wants to do with the shared object in some global data structure, then it checks in the announce-structure to see if there are other operations that it needs to help before proceeding with its own.

Barnes presented a method similar to helping in [Bar93]. In his method each operation on the shared data object is divided into a sequence of *virtually atomic suboperations*, where each suboperation is constructed so that once it has begun, it is guaranteed to be performed fully, either by the initiating process or by being helped by another process.

In [Her91] Herlihy presented a universal method for constructing a wait-free algorithm for any shared data object. However, as for the general methodology for construction of lock-free algorithms, the universal construction for wait-free algorithms is not practical in all cases and therefore significant research efforts are being spent on developing efficient wait-free algorithms.

1.5.4 Non-blocking data structures

There is a plethora of lock-free and wait-free shared data object implementations in the literature. In this section we will briefly mention a few of them and some of the issues the designers of non-blocking algorithms face.

Atomic Registers

A fundamental problem for concurrent shared memory systems that has received a significant amount of research effort is the *readers/writers problem*. In this problem a number of concurrent processes are interested in reading from and/or writing to a shared data object also called a *register*. All read or write operations should take effect atomically and they return or update the entire state of the shared data object. For small shared data objects that fit in a single memory word (of the word size supported by the multiprocessor system at hand) the hardware read/write instructions and, if needed, the memory barrier instructions described in Section 1.2.1 above provide the properties required. If, on the other hand, the shared data object is larger than a single word (of the word size supported by the multiprocessor system at hand) a software algorithm is needed to solve the readers/writers problem.

The classical solution is to use mutual exclusion to enforce that either (i) no read or write operations overlap each other; or (ii) no write operations overlap each other or any read operation. These methods, normally implemented using a *mutual exclusion lock* or a *readers-writers lock*, respectively, both suffer from the drawbacks of mutual exclusion which are further discussed in Section 1.5.2.

In [Lam77] Lamport introduced a solution to the readers/writers problem with one writer which did not use mutual exclusion. Lamport's algorithm allows the writer unimpeded access to the register regardless of what the readers do, while the readers will never interfere which each other but can be forced to retry if the writer interferes with them. This can force a slow reader to retry indefinitely. Lamport's algorithm marked the start of long running research efforts to construct solutions to the readers/writers problem where neither readers nor writer could be delayed indefinitely by interference from other readers or writers.

This has made this problem, also known as the problem of multi-word wait-free read/write registers, one of the well-studied problems in the area of non-blocking synchronization, with numerous results for the construction of e.g.:
(i) single-writer single-reader registers [Lam86, Sim90, CB97];
(ii) single-writer n-reader registers [Pet83, BP87, KKV87, NW87, KR93, SAG94, HV95, LGH$^+$04];
(iii) 2-writer n-reader registers [Blo88]; and
(iv) m-writer n-reader registers [VA86, PB87, IS92, LV92, LTV96, HV96].

The main goal of most of the algorithms in these results is to construct wait-free multi-word read/write registers from single-word read/write registers and not from other synchronization primitives which may be provided by the hardware in a system. This has been very significant, providing fundamental results in the area of wait-free synchronization, in particular considering the nowadays well-known and well-studied hierarchy of shared data objects and their synchronization power [Her91]. Many of these solutions also involve elegant and symmetric ideas and have formed the basis for further results in the area of non-blocking synchronization.

In Chapter 8 we present a simple, efficient wait-free algorithm for implementing multi-word n-reader/single writer registers of arbitrary word length utilizing synchronization primitives available on current multiprocessor systems.

Other non-blocking data structures

The FIFO queue is one of the fundamental data structures for which several researchers have proposed lock-free implementations, for example Valois [Val94], Michael and Scott [MS96], Michael [Mic02b] and Tsigas and Zhang [TZ01b].

The singly linked list is another data structure for which lock-free implementations have been proposed by Valois [Val95b] and Harris [Har01] among others. A lock-free implementation of doubly linked lists using *single-word* *Compare_And_Swap* has recently been proposed by Sundell and Tsigas [ST04, Sun04a]. The earlier implementations, for example Greenwald [Gre99] used *double-word* *Compare_And_Swap*.

There are a number of lock-free implementations of deques (that is, double-ended queues), two recent ones by Sundell and Tsigas [ST04] and Michael [Mic03] that use only *single-word* *Compare_And_Swap*, and a number of earlier implementations using *double-word* *Compare_And_Swap*, for example Greenwald [Gre99], Agesen et al. [ADF$^+$02] and Martin et al. [MMS02].

There are at least two implementations of lock-free priority queues, one by Sundell and Tsigas [ST03] and one by Israeli and Rappoport [IR93]. The latter is actually wait-free but requires the multi-word *Compare_And_Swap* synchronization primitive, which is not available in hardware on current processors and therefore has to be implemented in software which leads to significant overhead.

The software library *NOBLE* by Sundell and Tsigas [ST02] is a much welcome initiative as it makes a collection of implementations of lock-free algorithms easily accessible and usable for application programmers, something that was missing before.

There are a number of issues that complicate the implementation of non-blocking algorithms. We will discuss two of these below, namely management of dynamic memory and composition of non-blocking data structures.

Management of dynamic memory

Due to the concurrent nature of non-blocking algorithms the management and reclamation of dynamically allocated memory become difficult. In particular, after a thread has read a pointer to a dynamically allocated node from a shared variable the thread might be delayed for an arbitrary amount of time before it actually dereferences the pointer and accesses the node. This implies that some form of delayed *memory reclamation* or *garbage collection* scheme is necessary, since, if some other thread decides to delete the node in question, the node must not be reused until it is certain that no thread still have any local pointer to the node. Several such memory reclamation schemes have been proposed and will be discussed further in Section 1.5.5.

While the above schemes make it possible to handle dynamically allocated memory and to know when such memory can be freed safely, a non-blocking memory allocator is also needed to have a completely non-blocking memory management system. Memory allocators, including lock-free ones, will be discussed further in Section 1.6.2.

Composition of non-blocking data structures

Composition of non-blocking shared data objects is tricky since the operations on the combined object do not trivially have the desired linearizability properties. Consider for example some shared container object that internally has objects stored in different non-blocking subcontainers. In this case it is, for example, not obvious if it is possible to implement a lock-free membership test. If we naively move a stored object from one subcontainer to another by removing it from the first and then inserting it into the other there is a time interval when that object is only known to the process performing the move and therefore not a visible member of the container during that time.

1.5.5 Dynamic memory in non-blocking data structures

To manage dynamically allocated memory in non-blocking algorithms is difficult due to overlapping operations that might read, change or *dereference* (i.e. follow) references to dynamically allocated blocks of memory concurrently. One of the most problematic cases is when a slow process dereferences a pointer value that it previously read from a shared variable. This dereference of the pointer value could occur an arbitrarily long time after the shared pointer holding that value was overwritten and the memory designated by the pointer removed from the shared data structure. Consequently it is impossible to safely free or reuse the block of memory designated by this pointer value until we are sure that there are no such slow processes with pointers to that block.

This implies that some form of garbage reclamation scheme is necessary. There are several such schemes in literature with a wide and varying range of properties. Here we will use a subset of these properties to classify and present an overview of some of the memory reclamation schemes. The properties are:

I Safety of local references.
For local references, which are stored in private variables accessible only by one thread, to be safe the memory reclamation scheme must guarantee that a dynamically allocated node is never reclaimed while there still are local references pointing to it (cf. Figure 1.5). This is a fundamental property that all memory reclamation schemes discussed here have.

II Safety of shared references.
Additionally, a memory reclamation scheme could also guarantee that it is always safe for a thread to dereference any shared references located within a dynamic node the thread has a local reference to. Property I alone does not guarantee this, since for a node that has been deleted but cannot be reclaimed yet any shared references within it could reference nodes that have been deleted and reclaimed since the node was removed from the data structure, as is described in Figure 1.6.

III Automatic or explicit deletion.
A dynamically allocated node could either be reclaimed automatically when it is no longer accessible through any local or shared reference, that is, the scheme provides *automatic garbage collection*, or the user algorithm or data structure could be required to explicitly tell the memory reclamation scheme when a node is removed from the active data structure and should be reclaimed as soon as it has become safe. While automatic garbage collection is convenient for the user, explicit deletion by the user gives the reclamation scheme more information to work with. This could enhance the efficiency and/or enable the reclamation scheme to provide stronger guarantees, e.g. bounds on the amount of deleted but yet unreclaimed memory.

IV Requirements on the underlying memory allocator.
Some memory reclamation schemes require special properties from the underlying memory allocator, like, for example, that each allocatable node has a permanent (i.e. for the rest of the system's lifetime) reference counter associated with it. Other schemes are compatible with the well-known and simple *allocate*/*free* allocator interface where the node has ceased to exist after the call to *free*.

V Required synchronization primitives.
Some memory reclamation schemes are defined using synchronization primitives that few if any current processor architectures provide in hardware, such as for example *double word Compare_And_Swap*, which then have to be implemented in software often adding considerable overhead. Other schemes make do with *single word Compare_And_Swap*, *single word Load_Linked*/*Store_Conditional* or even just reads and writes alone.

The properties of the memory reclamation schemes discussed here are summarized in Table 1.1. One of the most important properties is Property II, that is whether the memory reclamation scheme guarantees that references contained within a referenced node can be dereferenced. Many lock-free algorithms and data structures need this property. Among the memory reclamation schemes that guarantee Property II we have the following schemes which are all based on reference counting: Valois et al. [Val95a, MS95], Detlefs et al. [DMMS01], Herlihy et al. [HLMM02] and Gidenstam et al. [GPST05] (also presented in Chapter 5) and the potentially blocking epoch-based scheme by Fraser [Fra04] (see also [Har05]).

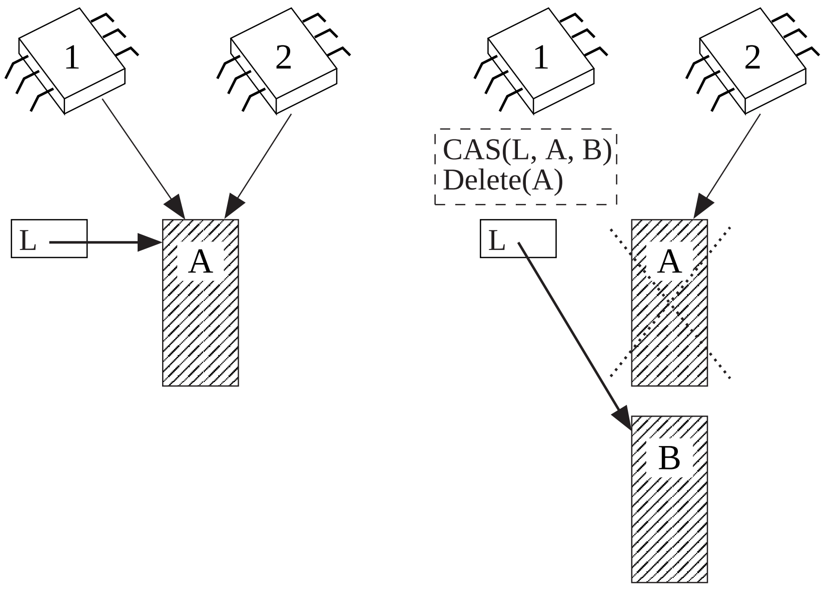


Figure 1.5: Example of Property I for reclamation of dynamic memory in non-blocking algorithms. Thread 1 and Thread 2 both have a local reference to the node A (created by reading L). Thread 1 changes the shared reference L to point to the node B, thereby deleting A from the shared structure. However, past that point Thread 2 might still access A through its local reference.

On the other hand, for data structures that do not need Property II, like for example stacks, the use of a reclamation scheme that does not provide this property has significant potential to offer reduced overhead compared to the stronger schemes. Among these memory reclamation schemes we have the non-blocking ones by Michael [Mic02b, Mic04c] and Herlihy et al. [HLM02].

1.6 System services and synchronization

Computer programs as we know them in current computer systems are usually run in a process environment provided by the *operating system*. Most current operating systems support multiple threads of control in each process, which consequently can be viewed as a virtual shared memory multiprocessor. Non-blocking shared data objects, such as the previously mentioned ones, are usually employed at the user-level by concurrent applications to provide synchronization between threads without involving the operating system kernel.

While it is possible to employ non-blocking method throughout an operating system kernel, as done, for example, by Massalin in [Mas92], this is, to my knowledge, not done in any commonly used operating system at this time.

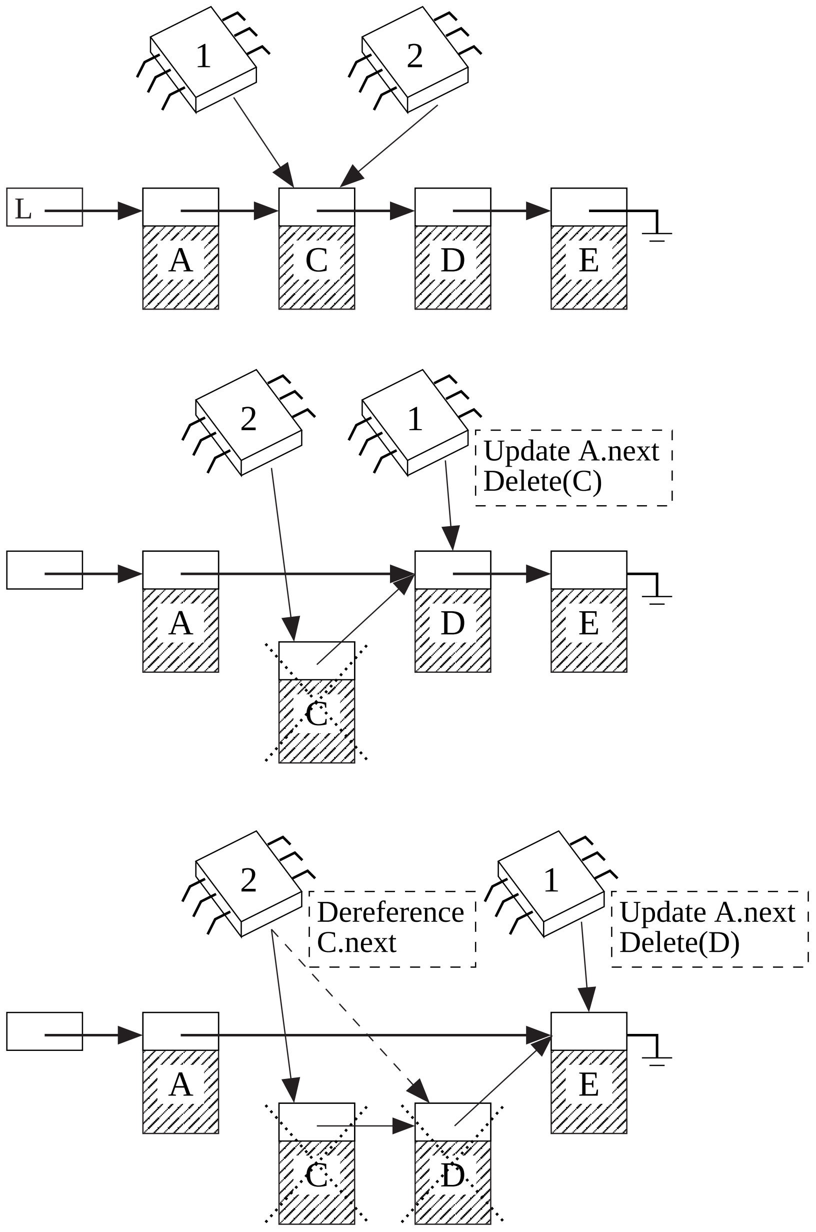


Figure 1.6: Example of Property II for reclamation of dynamic memory in non-blocking algorithms. Thread 1 and Thread 2 both have a local reference to the node C. Thread 1 removes the nodes C and D from the active structure. Thread 2, having a local reference to node C, might still dereference $C.next$ to access node D, e.g. to continue traversing the linked structure.

	Property II	Property III	Property IV	Property V
Michael [Mic02b, Mic04c]	No	Explicit	Yes	Yes
Herlihy et al. [HLM02]	No	Explicit	Yes	No
Valois et al. [Val95a, MS95]	Yes	Automatic	No	Yes
Detlefs et al. [DMMS01]	Yes	Automatic	Yes	No
Herlihy et al. [HLMM02]	Yes	Automatic	Yes	No
Gidenstam et al. [GPST05]	Yes	Explicit	Yes	Yes
Fraser [Fra04]	Yes	Explicit	Yes	Yes

Table 1.1: Properties of different approaches to non-blocking memory reclamation.

However, some system services might also benefit from non-blocking methods or could be adapted to simplify the implementation of such methods at the user-level. A major candidate among the system services for application of non-blocking methods is the memory management, which is, to a large extent, handled at the user-level within the process and usually only involves the kernel when additional pages have to be added to the heap area.

1.6.1 Information dissemination and consistency services

In distributed systems, applications often have communication and synchronization needs that are not satisfied by the operating systems on the participating computers alone. In such cases the required services could be implemented as part of the application itself or as *middleware*, that is, additional independent software components running on top of the operating systems.

In some distributed applications, such as *collaborative virtual environments* (CVEs) where remotely located participants interact in a shared virtual world and other distributed collaborative applications, there are (i) timing constraints, since the participants interact in real time, (ii) consistency requirements, since the participants need some degree of consistency in their views of the shared state to cooperate efficiently and (iii) scalability requirements since they might need to support a large number of participants. For these applications suitable middleware components could provide group communication protocols for information dissemination, consistency management and consistency tracking, such as logical clocks, and protocols for distributed resource management.

1.6.2 Memory allocation

Almost all computer systems have some form of general purpose memory management and the properties and design of *memory allocators* for this purpose have been researched for many years (see Wilson et al. [WJNB95] for an overview). However, memory management on a shared memory multiprocessor and/or in combination with non-blocking algorithms raises additional issues, among which the synchronization is an important one [Ber02].

On conventional general purpose memory allocators the application can request (allocate) arbitrarily-sized blocks of memory and free them in any order. Essentially the memory allocator is an online algorithm that manages some pool of memory (heap), for example a contiguous range of addresses or a set of such ranges, keeping track of which parts of that memory that is currently given to the application and which parts that are unused and can be used to meet future allocation requests from the application. The memory allocator is not allowed to move or otherwise disturb pieces of memory that are currently owned/allocated by the application. This, together with the fact that the request (allocation and deallocation) sequence is arbitrary, make it impossible to construct a memory allocator algorithm that always ensures efficient memory usage — it will always be possible for some application to produce a request sequence that causes severe fragmentation [Rob71, WJNB95].

Fortunately, most applications are well behaved in some sense with respect to memory allocation behaviour and exhibit regularities in their request sequences that a good memory allocator can exploit to reduce fragmentation.

Concurrent Memory Allocators

Multi-threaded programs add a number of extra complications to the memory allocator. Obviously some kind of synchronization has to be added to protect the heap during concurrent requests. There are also other issues, which may have significant impact on application performance when the application is run on a shared memory multiprocessor [Ber02].

The first of these is *false sharing* which is when different parts of the same cache-line end up being used by threads running on different processors. This will put a potentially large and completely unnecessary load on the cache-coherence mechanism. This can never be avoided completely since application threads may pass allocated memory between themselves but a memory allocator can avoid to *actively induce* false sharing by making sure not to satisfy requests from different processors with memory from the same cache-line.

The second issue is *heap blowup* which is an (potentially unbounded) over-consumption of memory (compared to what the application actually requires) that may occur if the memory allocator fails to make memory freed by one processor available for allocation by other processors (for example as the result of a coarse policy for avoiding false sharing). A typical application that might trigger this behaviour is an application that has producer and consumer threads, where the producer allocates memory and passes it to the consumer which in turn frees the memory. If the memory freed by the consumers is never made available to the producers then the resulting heap blowup can be unbounded.

The last issues are *scalability* and *speed*. For a memory allocator to be scalable, its performance has to scale linearly with the number of processors in the system. In terms of speed, the concurrent memory allocator should be about as fast as a good sequential one in order to ensure good performance even when a multi-threaded program is executed on a single processor.

A brief overview of some concurrent memory allocator designs:

- **Single serial heap.** A normal sequential memory allocator protected by a global lock. This type of memory allocator will scale poorly on multiprocessors, since only one thread can access the heap at a time, and it may in fact in some instances also perform bad on a single processor since a thread holding the heap lock might easily be delayed inside the memory allocator code (for example by a page fault). It is also prone to induce false sharing, but does not suffer from heap blowup and should be fast on a single processor in most cases (but note the lock problem mentioned above). Examples include the standard memory allocators on Solaris and Windows 2000.

- **Concurrent single heap.** A memory allocator with a single heap with fine-grained synchronization so that several threads may operate on it concurrently. This kind of memory allocator avoids heap blowup and might be scalable but is prone to induce false sharing and it is not easy to make it fast due to the synchronization overhead.

- **Pure private heaps.** The memory allocator maintains a separate heap for each processor, where threads running on that processor allocate memory. When a thread frees some memory it is added to the heap of the processor running that thread. This causes pure private heaps to suffer from potentially unbounded heap blowup. On the other hand this type of memory allocator is scalable, fast and is less prone to induce false sharing (if designed correctly). Examples include the STL allocator and the allocator in Cilk.

- **Private heaps with ownership.** These memory allocators are similar to pure private heaps but freed memory is always returned to the heap it was allocated from. This bounds the worst case heap blowup to $O(P)$, where P is the number of processors. Examples include MTmalloc (Solaris) and Ptmalloc (glibc).

- **Private heaps with thresholds.** To avoid the $O(P)$ heap blowup that private heaps with ownership suffer from, private heaps with thresholds use a global heap in addition to the per-processor heaps. When there is too much free memory in a per-processor heap some of it is transfered to the global heap from where it can be transfered to another per-processor heap and reused. Examples include Hoard [BMBW00], LFmalloc [DG02], the memory allocator in [Mic04c] and the memory allocator presented in Chapter 6 of this thesis.

On multiprocessor systems the Hoard memory allocator by Berger et al. [BMBW00] is currently one of the fastest and most scalable memory allocators. There is significant interest in developing lock-free memory allocators based on the Hoard architecture, examples are Michael's allocator [Mic04c], NBmalloc presented in Chapter 6 of this thesis and the almost lock-free allocator by Dice et al. [DG02]. The main reasons for this interest are i) to have a general purpose memory allocator that can be used by lock-free algorithms without destroying the lock-freedom; and ii) improve the scalability even further, in particular in cases when there are more threads than processors.

1.6.3 Scheduling

Processor scheduling of processes and threads is one of the fundamental tasks of a multiprogrammed operating system and how it is performed also influences how synchronization among threads and processes behaves. There are three basic approaches on how to handle the scheduling of processes and the scheduling of threads, which are separate execution flows/threads of control within one process, that is, share the same address space, file table and other process related resources. These approaches are (these brief descriptions are based on Tanenbaum [Tan01]):

- **User-level threads**. In this approach the threads are managed and scheduled entirely at the user-level and operating system kernel is completely unaware of them. The main benefit of this approach is a low overhead for managing the thread but there are several drawbacks: (i) it is difficult to handle blocking system calls; (ii) threads from the same process cannot utilize multiple processors and (iii) a page-fault will block all threads in a process.

- **Kernel-level threads**. In this approach the threads are scheduled and managed by the operating system kernel much in the same way as processes are. One benefit with this approach is that blocking system calls and page-faults only affect the involved thread and not the other threads of the process. Furthermore, it allows threads of the same process to run with full concurrency on a multiprocessor. The drawbacks are the increased overhead as all thread management and context switches between threads now have to be done by the kernel.

- **Hybrid methods**. These approaches try to combine the good properties of both the above approaches by implementing two levels of threads: *kernel threads* which are scheduled and managed by the kernel and *user-level threads* which are managed at the user-level and multiplexed onto kernel threads in a similar way as in the first approach. This should allow both thread management with low overhead, good concurrency on multiple processors and reasonable behaviour for blocking system calls. One of the main drawbacks is the increased complexity of the implementation when thread management is present both in the kernel and at the user-level. One interesting implementation of the hybrid approach is *scheduler activations* proposed by Anderson et al. [ABLL92].

Non-blocking synchronization in combination with process and thread scheduling are interesting in two different ways. The first is to make the scheduling process itself faster and more scalable by making it non-blocking, the other is to make synchronization at the user-level easier and more efficient by utilizing a scheduler with special properties or making the scheduler aware of ongoing user-level synchronization.

The first case is more easily adaptable for user-level or hybrid thread implementations as its use for kernel level threads require modifications in the kernel. An example is the work-stealing scheduler by Blumhofe et al. in [BL94].

Examples of the second case are work on non-blocking synchronization in systems with quantum scheduling by Anderson et al. [AJO98] and [AM99]; the implementation of non-blocking critical sections by having the scheduler release the lock on preemption by Bershad [Ber93]; and a method to turn lock-free synchronization into wait-free in single processor hard real-time systems by Anderson et al. [ARJ97].

1.6.4 High-level synchronization objects

It is common for operating systems or thread libraries to provide a set of synchronization objects, like for example semaphores, condition variables and mutexes, which can be used by the applications for synchronization purposes. Most such synchronization objects have blocking semantics, that is, a thread that has to wait will be put in a waiting queue and yields the processor. The implementation of the synchronization objects themselves on the other hand could benefit from being non-blocking, in particular when running many threads on a multiprocessor system with few processors — a situation where the traditional implementations based on spin-locks is likely to perform less well.

1.7 Contributions of the thesis

This thesis makes contributions to distributed system services for managing causality and concurrency by introducing scalable logical clock algorithms for tracking causality in the system and scalable and lightweight algorithms for middleware providing optimistic causal order group communication and resource management. Further, it makes contributions to the area of optimistic and non-blocking synchronization for increased scalability in key system services for shared memory systems by proposing and evaluating algorithms for lock-free memory allocation, lock-free memory reclamation, lock-free thread scheduling and synchronization and atomic multi-word registers.

1.7.1 System services for managing causality and concurrency

Adaptive Plausible Clocks

In the area of synchronization in message passing systems this thesis presents results on plausible logical clocks for use in large systems that demand both low overhead in terms of message size and good accuracy in determining causal relations between events. We have introduced the *Non-Uniformly Mapped R-Entries Vector Clocks* (NUREV), a general class of vector-based plausible clocks that extends and includes the R-Entries Vector (REV) clocks algorithm of [TRA99] and the full Vector clocks [Mat89, Fid88, Fid91]. The NUREV class forms a framework for implementing new plausible clock algorithms that may adaptively change their mapping between processes and clock entries in order to improve accuracy and scalability. Further, we analyze the behaviour of NUREV clocks to determine when, where and why they might order causally unrelated events. Based on this analysis we propose two new adaptive NUREV plausible clock algorithms that show very competitive accuracy, in particular for small timestamp sizes. Our work on logical clocks and new scalable methods to achieve optimistic forms of consistency for, e.g., peer-to-peer applications is presented in Chapter 2.

Lightweight Causal Cluster Consistency

Within an effort for providing a layered architecture of services supporting multi-peer collaborative applications, we propose a new type of consistency management aimed for applications where a large number of processes share a large set of replicated objects. Many such applications, like peer-to-peer collaborative environments for training or entertaining purposes, platforms for distributed monitoring and tuning of networks, rely on a fast propagation of updates on objects, however they also require a notion of consistent state update. To cope with these requirements and also ensure scalability, we propose the *cluster consistency* model. We also propose a two-layered architecture for providing cluster consistency. This is a general architecture that can be applied on top of the standard Internet communication layers and offers a modular, layered set of services to the applications that need them. Further, we present a *fault-tolerant protocol* implementing causal cluster consistency with predictable reliability, running on top of decentralized probabilistic algorithms supporting group communication and we analyze its reliability and fault-tolerance properties. Our experimental study, conducted by implementing and evaluating the two-layered architecture on top of standard Internet transport services, shows that the approach scales well, imposes an even load on the system, and provides high-probability reliability guarantees. Our work on providing group communication with optimistic causal order consistency is presented in Chapter 3.

Dynamic and Fault-tolerant Cluster Management

Recent decentralized event-based systems have focused on providing event delivery which scales with increasing number of processes. While the main focus of research has been on ensuring that processes maintain only a small amount of information for membership and routing, an important factor in achieving scalability for event-based peer-to-peer dissemination system is the number of events disseminated at the same time. This thesis presents a dynamic and fault-tolerant cluster management method which can be used to coordinate concurrent access to resources in a peer-to-peer system. In the context of event-based dissemination systems the cluster management can be used to control the number of concurrently disseminated events. We present and analyze an algorithm implementing the proposed cluster management model in a fault-tolerant and decentralized way. The algorithm provides for each cluster a limited set of tickets. A process which has obtained a ticket may send events corresponding to the resources of the cluster. The algorithm guarantees that no two processes ever issue an event corresponding to the same ticket at the same time. The cluster management model on its own has interesting properties which can be useful for peer-to-peer applications. Our work on synchronization and resource management for managing clusters as needed for the cluster consistency (Chapter 3) and more peer-to-peer applications is presented in Chapter 4.

1.7.2 Optimistic synchronization in shared memory system services

Lock-free Memory Reclamation

For lock-free shared memory data structures using dynamic memory this thesis presents an efficient and practical lock-free implementation of a memory reclamation scheme based on reference counting. The memory reclamation algorithm is aimed for use with arbitrary lock-free dynamic data structures. The algorithm guarantees the safety of local as well as global references, supports arbitrary memory reuse, uses atomic primitives which are available in modern computer systems and provides an upper bound on the amount of dynamically allocated memory prevented for reuse. To the best of our knowledge, this is the first lock-free algorithm that provides all of these properties. Experimental results indicate significant performance improvements for lock-free algorithms of dynamic data structures that require strong garbage collection support. Our work on lock-free memory reclamation is presented in Chapter 5.

Lock-free Memory Allocator

For shared memory systems this thesis presents results on synchronization in one of the fundamental system services in multiprocessor systems, namely memory allocation. In particular, it presents NBmalloc, a scalable lock-free memory allocator for use in concurrent applications. Its architecture is inspired by

Hoard [BMBW00], which is a successful system for concurrent memory allocators, based on a modular, scalable design that in itself offers good scalability and helps circumvent significant obstacles present in shared-memory multiprocessor environments such as false sharing and heap blowup. Our allocator is lock-free to improve scalability even further, in particular in highly concurrent applications that use more threads than the number of available processors. Within our effort on designing appropriate lock-free algorithms for the synchronization in this system, we propose and give a non-blocking implementation of a new data structure called flat-set, which supports conventional "intra-object" operations (to insert and find elements) as well as "inter-object" operations, for moving an element from one *flat-set* to another. Our work on lock-free memory management is presented in Chapter 6.

Lock-free Thread Library

For system services in shared memory systems this thesis presents LFthreads, a thread library whose synchronization is entirely based on lock-free techniques, which means that no spin-locks or similar synchronization mechanisms are employed in the implementation of the multithreading. This implies that processors are always able to do useful work and is achieved by a new synchronization algorithm, hand-off, that does not need any special kernel support. Since lock-freedom is highly desirable in multiprocessors due to its advantages in performance, fault-tolerance, convoy- and deadlock-avoidance, there is an increased demand in lock-free methods in multiprocessor applications, hence also in multiprocessor system services. This is why the existence of a lock-free multithreading library is important. To the best of our knowledge LFthreads is the first thread library that provides a lock-free blocking synchronization primitive for application threads. Our work on the LFthreads library, including the hand-off synchronization algorithm is presented in Chapter 7.

Multiword Atomic Register

Modern multiprocessor systems offer advanced synchronization primitives, built in hardware, to support the development of efficient parallel algorithms. In this thesis we develop a simple and efficient algorithm for atomic registers (variables) of arbitrary length. The simplicity and better complexity of the algorithm is achieved via the utilization of two such common synchronization primitives. We also evaluate the performance of our algorithm and the performance of a practical previously know algorithm that is based only on read and write primitives. The evaluation is performed on three well-known parallel architectures. This evaluation clearly shows that both algorithms are practical and that as the size of the register increases our algorithm performs better, accordingly to its complexity behavior. Our work on atomic registers of arbitrary size for shared memory systems is presented in Chapter 8.

Chapter 2

Adaptive Plausible Clocks[1]

Anders Gidenstam Marina Papatriantafilou

Abstract

Having small-sized logical clocks with high causal-ordering accuracy is useful, especially where (i) the precision of the knowledge of the causal dependencies among events implies savings in time overhead and (ii) the cost of transmitting Full Vector clock timestamps —that precisely characterize the causal relation— is high. Plausible clocks can be used as timestamps to order events in a distributed system in a way that is consistent with the causal order as long as the events are causally dependent. In this work we study the accuracy of plausible clocks, which is measured by the number of causally independent event pairs that they relate. We introduce the Non-Uniformly Mapped R-Entries Vector (NUREV) clocks, a general class of plausible clocks, which allow the use of clock vectors with a small number of entries and which also allow each process in the system to use a different mapping between process-ids and clock-entry indices, the idea being that dynamic mappings allow self-tuning and adaptation to improve the accuracy of the clocks. Furthermore, we analyze the ways that these clocks may relate causally independent event pairs. Our analysis resulted in a set of conclusions and the formulation of new, adaptive plausible clocks algorithms, with improved accuracy, even when the number of clock entries is very small, which is important in peer-to-peer communication systems.

Keywords: Timestamping systems, logical clocks, plausible clocks, event ordering, consistency.

[1]This is an extended version of the paper that appeared in the Proceedings of ICDCS 2004, Tokyo, Japan, 24-26 March, 2004.

2.1 Introduction

To observe consistent states in a distributed execution, it is desirable to be able to establish some order among processes' local states or among events of the execution. Certain consistency requirements demand the ability to produce total orderings. Examples include sequentially consistent distributed shared memory, distributed FIFO queues, among others. Where total orderings are not a must, as for several modern peer-to-peer applications [SJZ$^+$98, Mau00], relaxed consistency guarantees such as *causal consistency* can imply higher flexibility and lower overhead [AHJ91, TRA99, FJC00]. This is also the case in several protocols for *optimistic replication* [SS05], where performance may be improved by a good ability to distinguishing whether pairs of events have a cause and effect relation or not.

To be able to determine the causal relation between events (i.e. whether they have a cause-effect relation or are independent) we can *timestamp* the events using some clock value. For asynchronous systems without physical clocks, *logical clocks* can be used for timestamping events, in a way so that event orderings based on incremental timestamp values are consistent with causal precedence. *Vector clock* timestamps [Mat89, Fid91, Sch88] can capture the exact causal relation at the cost of $O(N)$ clock entries per timestamp for a system of N processes. Since any logical clock that can determine the exact causal relation between events in the system directly from the timestamps requires that a timestamp of size $O(N)$ is included in every message [CB91], it is important to investigate the *accuracy* of logical clocks that *approximate* the causal relation and use timestamps with fixed, small size, thus enhancing the scalability of the system.

Having logical clocks with high causal-ordering accuracy is useful where the precision of the knowledge of the causal relation among events implies savings in time overhead, e.g. in resource allocation and consistency maintenance [TRA99]. This becomes all the more important in larger systems, where the cost of transmitting Full Vector clock timestamps is high.

Related Work

Torres-Rojas and Ahamad in [TRA99] formalized the notion of *plausible clocks.* A plausible clock can *always determine the order of causally related events correctly* but *may order events which are actually concurrent.* Lamport's logical clocks [Lam78] are examples of plausible clocks. The *inaccuracy* of a plausible clock algorithm is measured by the number of ordering errors it makes in an execution, i.e. the number of causally independent event pairs that it relates. In [TRA99] a couple of plausible clock algorithms were introduced, namely the R-Entries Vector (REV) clocks and the k-Lamport clocks. A method for combining plausible clocks was also introduced: the combined clock is also plausible and has at least the same or potentially improved accuracy, at the cost of having size equal to the sum of the sizes of its components. The experimental evaluation of the algorithms in [TRA99] has shown that they have improved accuracy compared to Lamport clocks, with the REV clock showing consistently better behaviour. As pointed out in the same paper, the accuracy of a plausible clock may depend on a number of factors, such as the size of the system and the execution, the communication patterns among the processes and possibly more. The analysis and evaluation of these dependencies, as well as their implications in the design of plausible clock algorithms were left as issues for future research.

The issue of reducing the size of Vector Clocks as has received considerable interest in literature.

In [PS97] Prakash and Singhal introduced methods for more space efficient alternatives to Vector Clocks in a two level system model where mobile nodes always communicate via base stations.

Building on that work Khotimsky and Zhuklinets in [KZ99] introduced a plausible clock algorithm, Hierarchical Vector Clocks, which also targets hierarchical networks, but which allows the hierarchy to have any number of levels.

In [MS05] Moore and Sivilotti introduced an algorithm for plausible clocks with bounded inaccuracy. In their algorithm the user specifies an upper bound on the acceptable precision loss for any timestamp. The size of each timestamp is then chosen such that the precision bound is not violated.

In some systems it is enough to track only a subset of all events, *the relevant events.* For such systems that also have access to shared memory Agarwal and Garg in [AG06] introduced a class of logical clocks, called *Chain Clocks*, and two algorithms in this class: the *Dynamic Chain Clock* and the *Antichain-based Chain Clock.*

There is another *trade-off* that can be made to gain fixed-size timestamps, namely *time and functionality*: in [BM03] Baldoni and Melideo introduced a timestamping system that characterizes causality and has fixed-size timestamps. However, in this algorithm only a dedicated checker process, which needs to be notified about all events in the system, can determine the causal relation between events —and sometimes it has to delay its decision until related notifications have arrived.

The results of our work make significant steps in the investigation of these issues. More specifically, we extend the notion of reduced size vector clocks to include clocks where each process chooses its own mapping between process-ID's and clock entries and furthermore we allow these mappings to change dynamically during the lifetime of the system. We call this class of clocks (which includes the aforementioned R-Entries Vector clock and the Full Vector clock algorithms) *Non-Uniformly Mapped R-Entries Vector* (NUREV) clocks and show that all NUREV clocks are plausible. The idea behind NUREV clocks is that *dynamic mappings* allow *self-tuning* and *adaptation* of the clocks in order to improve their accuracy. We analyze the ways that plausible clocks may relate causally independent event pairs and show that it is both the communication patterns, as well as the actual clock values — in particular the value-differences among related entries — that influence the accuracy of the clock. These conclusions implied a set of criteria on which to base decisions, which, in turn, resulted in new adaptive mapping strategies, MinDiff and ROV-MRS that offer high causal-ordering accuracy.

The experimental evaluation of the performance of our proposed methods agrees well with the conclusions of the analysis part and also shows promising results from the applicability point of view, namely that adaptive plausible clocks with very small number of entries can give very good event-ordering accuracy. In our evaluation we include both *peer-to-peer* communication systems, which are the main target applications for such algorithms, as well as *client-server* communication systems.

After describing our system model, we present our results, in the aforementioned order. We conclude by a discussion of the results and future research issues in this direction.

2.2 Model and definitions

We follow the standard model used in the related literature. For most of the notational conventions, we adopted the notation used in [TRA99].

The system consists of N sequential processes which are identified by distinct identity numbers, denoted by $p_1, \ldots, p_N$. The processes communicate with each other by messages. The communication is point-to-point and fault-free. There is no physical clock accessible to processes. We make no assumptions about the relative speeds of the processes or the communication channels. Each system execution is a set H of *events*. An event can be the sending of a message, the reception of a message or a local step by some process. In each system execution, each process p_i executes a sequence of events, which is called its *local history* for that execution and is denoted by $H_i = e_i^1, e_i^2, e_i^3, \ldots$.

The *happened-before* a.k.a. *causal precedence* relation $\rightarrow$ on the set H of all the events of a system execution has been defined by Lamport [Lam78]: an event a is said to precede an event b, denoted as $a \rightarrow b$, iff:

(i) for some i, $a = e_i^k, b = e_i^l$ and $k < l$ (i.e. p_i executed a before b); or

(ii) a is the sending of a message by some p_i and b is the receipt of the same message by some p_j; or

(iii) there exists an event $c \in H$, s.t. $a \rightarrow c$ and $c \rightarrow b$.

If a does not precede b and b does not precede a then they are *concurrent*. *Plausible Time-Stamping Systems* are mechanisms for timing events so that event orderings based on incremental clock values are consistent with causal precedence. Following are the formal definitions paraphrased from [TRA99]:

For a global history H of a distributed system, a *Time-Stamping System* (*TSS*) P is a pair $\left(\langle S, \overset{P}{\rightarrow}\rangle, P.\textbf{stamp}\right)$, where:

- S is a set of timestamp values (whose details are left open by this definition);
- $\overset{P}{\rightarrow}$ is an irreflexive and transitive relation defined on the elements of S such that $\langle S, \overset{P}{\rightarrow}\rangle$ is a strict partial order;

- P.**stamp** is the timestamping function that maps H to S. We write P.**stamp**(a), or shorter $P(a)$, to represent the timestamp of a.

For $u, v \in S$ define:

(i) $v \overset{P}{=} u$ iff $v = u$; and

(ii) $v \overset{P}{\parallel} u$ iff $\neg(v \overset{P}{=} u) \wedge \neg(v \overset{P}{\rightarrow} u) \wedge \neg(u \overset{P}{\rightarrow} v)$.

Further, we write:

- $a \overset{P}{=} b$ iff $P(a) \overset{P}{=} P(b)$, i.e. when P believes that a, b are the same event;
- $a \overset{P}{\rightarrow} b$ iff $P(a) \overset{P}{\rightarrow} P(b)$, i.e. when P believes that a precedes b; and
- $a \overset{P}{\parallel} b$ iff $P(a) \overset{P}{\parallel} P(b)$, i.e. when P believes that a and b are concurrent.

A time-stamping system P is *plausible* if $\forall a, b \in H$:

(i) $a = b$ iff $a \overset{P}{=} b$; and

(ii) if $a \rightarrow b$ then $a \overset{P}{\rightarrow} b$.

The second condition is also known as the *weak clock condition*.

A time-stamping system *characterizes causality* if $\forall a, b \in H$:

(i) $a = b$ iff $a \overset{P}{=} b$;

(ii) $a \rightarrow b$ iff $a \overset{P}{\rightarrow} b$; and

(iii) $a \| b$ iff $a \overset{P}{\parallel} b$.

This set of requirements is also known as the *strong clock condition*.

A reader familiar with the related literature, has probably observed at this point that Lamport's *logical clocks* [Lam78] are plausible, while *vector clocks* (cf. Fidge [Fid91] and Mattern [Mat89]) characterize causality –but, as mentioned also elsewhere in this paper, at the cost of $O(N)$ vector entries, which has been shown to be necessary for any logical clock that characterizes causality [CB91].

To measure the amount of inaccuracy of a plausible time-stamping system we use the proportion of all pairs of concurrent events in a history that are believed to be causally related by the time-stamping system; i.e.:

$$error(P, H) = \frac{|(a, b) \in H \times H : (a \| b) \wedge (a \overset{P}{\rightarrow} b)|}{|(a, b) \in H \times H : a \| b|}.$$

Note that [TRA99] uses the ratio against *all* event pairs. We claim that the ratio against the number of *concurrent* event pairs is a more precise measure since a plausible clock never orders ordered event pairs wrongly.

2.3 Non-uniformly mapped vector clocks

The *R-Entries Vector Clock* (REV) introduced in [TRA99] is similar to a vector clock but has only R ($\leq N$) entries whereas a full vector clock has N entries. Each process is associated with one entry in the clock vector by a mapping function, $f(\cdot)$, which in the case of the REV clock is *process-ID* mod R.

To address the question, pointed out in [TRA99], of how the choice of mapping function affects the performance of a fixed size vector clock, we introduce the class of *Non-Uniformly Mapped R-Entries Vector* clocks (*NUREV*), which generalize the REV clock by allowing each process to use its own mapping between process-IDs and clock-entry indices and to change this mapping over time. These two generalizations make it possible to define a class of R-entry vector clocks where each clock may automatically tune itself to perform better for the current communication patterns in the system. Formally a NUREV time-stamping system is a tuple $\left(\langle S, \overset{NUREV}{\rightarrow}\rangle, NUREV.\mathbf{stamp}\right)$ where:

- S is a set of tuples of the form $\langle p_i, V_i, f_i\rangle$ where p_i is an integer that identifies each process in the system, V_i is a $1-$dimensional vector of R integers and f_i is a function from process-ID to entry index $\{1, \ldots, R\}$ (f_i may be different in different tuples).

- $NUREV$.**stamp** is defined by the rules
  NUREV0) Initial value:
  $$\begin{aligned} p_i &= \text{unique process-ID} \in \{1, \ldots, N\} ; \\ f_i &= \text{some mapping function} ; \\ V_i[r] &= 0 \ \forall r \in 1, \ldots, R. \end{aligned}$$
  NUREV1) When a send or local event with timestamp $\langle p_i, V_i^+, f_i^+\rangle$ is generated:
  $$\begin{aligned} f_i^+ &= \text{updated } f_i ; \\ V_i^+[r] &= \max\left\{V_i[f_i(j)] : \forall j. f_i^+(j) = r\right\} + \\ & \quad own(r), \text{ where} \\ own(r) &= \begin{cases} 1 & \text{if } f_i^+(p_i) = r \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$
  NUREV2) When a message with timestamp $\langle s, V_s, f_s\rangle$ is received:
  $$\begin{aligned} f_i^+ &= \text{updated } f_i ; \\ V_i[r] &= \max\left\{\max(V_i[f_i(j)], V_s[f_s(j)]) : \right. \\ & \quad \left. \forall j. f_i^+(j) = r\right\} + own(r). \end{aligned}$$

- Let $\langle p_i, V_i, f_i\rangle, \langle p_j, V_j, f_j\rangle \in S$ then:
  $\langle p_i, V_i, f_i\rangle \overset{NUREV}{\rightarrow} \langle p_j, V_j, f_j\rangle \Leftrightarrow$
  $((p_i = p_j \wedge V_i[f_i(p_j)] < V_j[f_j(p_j)]) \vee (p_i \neq p_j \wedge$
  $(\forall k. V_i[f_i(k)] \leq V_j[f_j(k)]) \wedge (V_i[f_i(p_j)] < V_j[f_j(p_j)])))$

Intuitively, the NUREV TSS applies the ordinary Vector Clock update and comparison rules on (implicitly) expanded versions of the R-entry vectors by using the mapping functions to access the clock values. Note that the NUREV rules do not impose any restrictions on how the mapping function f_i looks nor on how it is updated. Observe also that the ordinary Vector Clock is a NUREV clock using the identity function on N entries and the REV clock of [TRA99] is a NUREV clock using the fixed mapping function *process-ID* mod R.

An important property of the NUREV logical clocks is that they all are plausible clocks.

Theorem 2.3.1 *NUREV clocks are plausible Time-Stamping Systems.*

Proof:
First we need to prove that NUREV is a Time-Stamping System and then that NUREV is plausible. The former is proved by showing that the relation $\overset{NUREV}{\rightarrow}$ is irreflexive and transitive. The later is proved by showing that NUREV satisfies the definition of a plausible clock.

Let $\langle p_i, V_i, f_i\rangle \in S$ and assume towards a contradiction that $\langle p_i, V_i, f_i\rangle \overset{NUREV}{\rightarrow} \langle p_i, V_i, f_i\rangle$. Then we have

$$\langle p_i, V_i, f_i\rangle \overset{NUREV}{\rightarrow} \langle p_i, V_i, f_i\rangle \quad \Leftrightarrow \quad (p_i = p_i \wedge V_i[f_i(p_i)] < V_i[f_i(p_i)])$$

which is a contradiction. So $\overset{NUREV}{\rightarrow}$ is irreflexive.

To prove that *NUREV* is transitive let $\langle p_i, V_i, f_i\rangle, \langle p_j, V_j, f_j\rangle, \langle p_k, V_k, f_k\rangle \in S$ and assume that $\langle p_i, V_i, f_i\rangle, \overset{NUREV}{\rightarrow} \langle p_j, V_j, f_j\rangle$ and $\langle p_j, V_j, f_j\rangle \overset{NUREV}{\rightarrow} \langle p_k, V_k, f_k\rangle$. The proof is done by case analysis on the sources of the events and follows below.

$$\begin{aligned}
(p_i = p_j = p_k) \quad &\Rightarrow \quad (p_i = p_j \wedge V_i[f_i(p_j)] < V_j[f_j(p_j)]) \wedge \\
& \quad (p_j = p_k \wedge V_j[f_j(p_k)] < V_k[f_k(p_k)]) \\
&\Rightarrow \quad (p_i = p_k \wedge V_i[f_i(p_k)] < V_k[f_k(p_k)]) \\
&\Rightarrow \quad \langle p_i, V_i, f_i\rangle \overset{NUREV}{\rightarrow} \langle p_k, V_k, f_k\rangle
\end{aligned}$$

$$\begin{aligned}
(p_i \neq p_j = p_k) \quad &\Rightarrow \quad (p_i \neq p_j \wedge \forall l. V_i[f_i(l)] \leq V_j[f_j(l)] \wedge \\
& \quad V_i[f_i(p_j)] < V_j[f_j(p_j)]) \wedge \\
& \quad (p_j = p_k \wedge V_j[f_j(p_k)] < V_k[f_k(p_k)]) \\
&\Rightarrow \quad (p_i \neq p_k \wedge \forall l. V_i[f_i(l)] \leq V_k[f_k(l)] \wedge \\
& \quad V_i[f_i(p_k)] < V_k[f_k(p_k)]) \\
&\Rightarrow \quad \langle p_i, V_i, f_i\rangle \overset{NUREV}{\rightarrow} \langle p_k, V_k, f_k\rangle
\end{aligned}$$

$$\begin{aligned}
(p_j \neq p_i = p_k) \quad &\Rightarrow \quad (p_i \neq p_j \wedge \forall l. V_i[f_i(l)] \leq V_j[f_j(l)] \wedge \\
& \quad V_i[f_i(p_j)] < V_j[f_j(p_j)]) \wedge \\
& \quad (p_j \neq p_k \wedge \forall l. V_j[f_j(l)] \leq V_k[f_k(l)] \wedge \\
& \quad V_j[f_j(p_k)] < V_k[f_k(p_k)]) \\
&\Rightarrow \quad (p_i = p_k \wedge V_i[f_i(p_k)] < V_k[f_k(p_k)]) \\
&\Rightarrow \quad \langle p_i, V_i, f_i\rangle \overset{NUREV}{\rightarrow} \langle p_k, V_k, f_k\rangle
\end{aligned}$$

$$\begin{aligned}
(p_k \neq p_i = p_j) \quad &\Rightarrow \quad (p_i = p_j \wedge V_i[f_i(p_j)] < V_j[f_j(p_j)]) \wedge \\
&\quad\ \ (p_j \neq p_k \wedge \forall l.V_j[f_j(l)] \leq V_k[f_k(l)] \wedge \\
&\quad\ \ V_j[f_j(p_k)] < V_k[f_k(p_k)]) \\
&\Rightarrow \quad (p_i \neq p_k \wedge \forall l.V_i[f_i(l)] \leq V_k[f_k(l)] \wedge \\
&\quad\ \ V_i[f_i(p_k)] < V_k[f_k(p_k)]) \\
&\Rightarrow \quad \langle p_i, V_i, f_i\rangle \overset{NUREV}{\rightarrow} \langle p_k, V_k, f_k\rangle
\end{aligned}$$

$$\begin{aligned}
(p_i \neq p_j \neq p_k) \quad &\Rightarrow \quad (p_i \neq p_j \wedge \forall l.V_i[f_i(l)] \leq V_j[f_j(l)] \wedge \\
&\quad\ \ V_i[f_i(p_j)] < V_j[f_j(p_j)]) \wedge \\
&\quad\ \ (p_j \neq p_k \wedge \forall l.V_j[f_j(l)] \leq V_k[f_k(l)] \wedge \\
&\quad\ \ V_j[f_j(p_k)] < V_k[f_k(p_k)]) \\
&\Rightarrow \quad (p_i \neq p_k \wedge \forall l.V_i[f_i(l)] \leq V_k[f_k(l)] \wedge \\
&\quad\ \ V_i[f_i(p_k)] < V_k[f_k(p_k)]) \\
&\Rightarrow \quad \langle p_i, V_i, f_i\rangle \overset{NUREV}{\rightarrow} \langle p_k, V_k, f_k\rangle
\end{aligned}$$

Thus the NUREV clock is a Time-Stamping System since $\overset{NUREV}{\rightarrow}$ is both irreflexive and transitive.

We need now to proceed by proving that the NUREV TSS is plausible.

Let $a, b \in H$ be two arbitrary events such that $NUREV(a) = \langle p_i, V_i, f_i\rangle$ and $NUREV(b) = \langle p_j, V_j, f_j\rangle$.

It is easy to see that

$$\forall a, b \in H : a = b \Leftrightarrow a \overset{NUREV}{=} b.$$

If a and b occurred at the same site then $\overset{P}{\rightarrow}$ will establish their causal relation correctly by comparing $V_i[f_i(p_j)]$ and $V_j[f_j(p_j)]$.

Consider the case when a and b occurred at different sites. If $a \rightarrow b$ then from the definition of $NUREV$.**stamp** we have that $\forall l.V_i[f_i(l)] \leq V_j[f_j(l)]$ and $V_i[f_i(p_j)] < V_j[f_j(p_j)]$ must hold. This is also what $\overset{NUREV}{\rightarrow}$ requires so

$$a \rightarrow b \quad \Rightarrow \quad a \overset{NUREV}{\rightarrow} b$$

and thereby proving the NUREV TSS to be plausible.

□

2.4 Analysis of event orderings using plausible clocks

To come up with a gnomon on which to base decisions regarding which processes should share vector entries, we should identify the cases where such a clock may incorrectly order a pair of concurrent events and draw conclusions about what a mapping function should be aiming at, to recognize as many concurrent events as possible. The results in this section are formalized for NUREV clocks. They can be adapted to any plausible clock in which a notion of mapping between processes and clock entries can be defined.

We assume a global time model, where $t(e_i)$ denotes the time when event e_i happened. This assumption is not for the algorithms, but only for analyzing the clocks' behaviour. For any event e_i with timestamp $\langle p_i, V_i, f_i\rangle$, we introduce (again, for the analysis' sake) a corresponding N-entry vector $xp(V_i)$, where $xp(V_i)[j] = V_i[f_i(j)]$. We call $xp(V_i)$ the *expanded form of* V_i. For two vectors xp_1, xp_2 we say that $xp_1 > xp_2$ iff $xp_1[i] \geq xp_2[i]$ for all i and there is at least one entry j such that $xp_1[j] > xp_2[j]$. For the following, consider e_i, e_j be two arbitrary events of processes p_i, p_j ($p_i \neq p_j$) in a system execution and let $\langle p_i, V_i, f_i\rangle, \langle p_j, V_j, f_j\rangle$ be their corresponding NUREV timestamps.

Lemma 2.4.1 *If* $V_i[f_i(p_i)] > V_j[f_j(p_i)]$ *then* $e_i \overset{NUREV}{\not\to} e_j$ *and* $e_i \not\to e_j$.

Proof: Assume towards a contradiction that $e_i \overset{NUREV}{\to} e_j$. Then, according to the definition of $\overset{NUREV}{\to}$ (cf. section 2.3) we either have that:

(i) $p_i = p_j$ and $V_i[f_i(p_j)] < V_j[f_j(p_j)]$ which is a contradiction; or

(ii) that $p_i \neq p_j$ and $\forall k.V_i[f_i(k)] \leq V_j[f_j(k)]$, that is, we have $V_i[f_i(p_i)] \leq V_j[f_j(p_i)]$ and a contradiction again.

Now, assume towards a contradiction that $e_i \to e_j$. We have two cases:

(i) e_i and e_j are events of the same process. Then there has to be at least one invocation of the **NUREV1** or **NUREV2** update rules between them. Both of these increase the own entry of the process by at least 1, so we have that $V_i[f_i(p_i)] < V_j[f_j(p_i)]$ which is a contradiction.

(ii) e_i and e_j are events of different processes, then there has to at least one invocation of the **NUREV2** rule that merges the timestamp of e_i or an event that is causally preceded by e_j with the clock of the process of e_j (this either precedes e_j or is the event e_j). Then be the definition of **NUREV2** we have $V_j[f_j(p_i)] \geq V_i[f_i(p_i)]$ which is a contradiction.

□

Lemma 2.4.2 *If* $(V_i[f_i(p_i)] > V_j[f_j(p_i)]) \wedge (V_i[f_i(p_j)] < V_j[f_j(p_j)])$ *then* $e_i \overset{NUREV}{\|} e_j$ *and* $e_i \| e_j$.

Proof: By applying the previous lemma (2.4.1) in both directions we get $e_i \overset{NUREV}{\not\to} e_j$, $e_i \not\to e_j$ and $e_j \overset{NUREV}{\not\to} e_i$, $e_j \not\to e_i$. Then, from the definitions of $\overset{NUREV}{\|}$ and $\|$ (cf. section 2.2), it follows that $e_i \overset{NUREV}{\|} e_j$ and $e_i \| e_j$. □

Notice that for Full Vector clocks and if the timestamp of each event includes the identity of the process that executed it, the condition in the latter lemma is both necessary and sufficient for the two events being concurrent [BM93].

In the following we analyze the conditions for NUREV to incorrectly relate a pair of concurrent events. We first define some terms that will be useful in the rest of this section. Given an event e_i and its corresponding NUREV timestamp $\langle p_i, V_i, f_i\rangle$, we call:

- $V_i[f_i(p_i)]$: e_i's *own key.*
- $V_i[f_i(p_j)]$ $(p_i \neq p_j)$: p_i's *presumption of the own key of* p_j's *latest event preceding* e_i. (For the limit case when there is no event by p_j preceding e_i, consider the *initializing event* by each process, which sets all the vector entries to 0, to precede the first events in the executions of all processes).

Further, we call a sequence of events $e_i, e_{m_1} \dots e_{m_n}, e_j$ the *value-propagating sequence for the presumed key of* p_k *in* e_j if the value of $V_j[f_j(p_k)]$ originates from $V_i[f_i(p_i)]$, that is, the sequence of events is such that $e_i \rightarrow e_{m_1} \rightarrow \dots \rightarrow e_{m_n} \rightarrow e_j$, the events have timestamps with mapping functions such that $f_{m_1}(p_i) = f_{m_1}(\overline{p_1})$, $f_{m_2}(\overline{p_1}) = f_{m_2}(\overline{p_2}), \dots f_{m_n}(\overline{p_{n-1}}) = f_{m_n}(\overline{p_n})$ and $f_j(\overline{p_n}) = f_j(p_k)$ for some set of process-IDs $\{\overline{p_1} \dots \overline{p_n}\}$.

Lemma 2.4.3 *If for two events e_i and e_j $V_i[f_i(p_i)] \leq V_j[f_j(p_i)]$ then either $e_i \rightarrow e_j$ or $e_i \| e_j$ and there exists an event timestamped $\langle p_k, V_k, f_k\rangle$ by a process p_k $(p_k \neq p_i)$ such that $e_k = e_j$ or $e_k \rightarrow e_j$ and $V_i[f_i(p_i)] \leq V_k[f_k(p_k)]$ and there is a* value-propagating sequence *of events for the presumed key of p_i in e_j from e_k to e_j.*

Proof: From lemma 2.4.1 it follows that $e_j \not\rightarrow e_i$. Then either $e_i \rightarrow e_j$ or $e_i \| e_j$. In the latter case, consider the possible origins of e_j's presumed key of p_i (i.e. the value of $V_j[f_j(p_i)]$). There are four cases:

(i) $V_j[f_j(p_i)]$ is shared with the own key of p_j in e_j, that is, $f_j(p_i) = f_j(p_j)$. Then e_j is the e_k in the lemma.

(ii) The value of $V_j[f_j(p_i)]$ was last increased by a previous event e'_j at p_j and $V'_j[f'_j(p_i)]$ was shared with the own key of p_j, that is, $f'_j(p_i) = f'_j(p_j)$. Then e'_j is the e_k in the lemma.

(iii) The value of $V_j[f_j(p_i)]$ was last increased when p_j's presumption of p_i's key became shared with the (larger) presumed key of another process p_m either in the timestamp e_j or in the timestamp of a preceding event e'_j of p_j. Then we can apply the lemma recursively to find the origin of the value of $V_j[f_j(p_m)]$ or $V'_j[f'_j(p_m)]$, respectively.

(iv) The value of $V_j[f_j(p_i)]$ was received from another process p_m in the timestamp of a send event and message e_m to p_j $(e_m \rightarrow e_j)$. Then we can apply this lemma recursively to e_m to find the origin of the value of the presumed key of p_i in e_m and e_j.

To see that the recursions in case (iii) and case (iv) are well founded, note first that it follows from the NUREV update rules that all values that occur in clock entries have to originate from the own key of some process, because a new value can only be introduced by the addition in the update of the own key of a process. When two or more other keys are to share one clock entry, the value of that entry becomes the largest value from a set of already present values.

Now, in case (iii) the recursion is done on the largest presumed key of p_m that is mapped to the same clock entry as the presumed key of p_i. There may be several such keys for this entry but regardless of which one we choose, its origin will be traceable to a preceding event.

In case (iv) the recursion is done on an event e_m preceding e_j, so each recursion brings us closer to the start of the execution (and the *initializing event*, which, however, can only be reached if the inflated value of e_j presumed key of p_i is 0, which is impossible since it should be inflated).

□

Intuitively, clocks advance because of precedence or merged entries, i.e. using NUREV clocks, e_j's presumption of the own key of p_i's latest event preceding e_j can be an *inflated value*, since p_i might have shared its entry in p_j's clock with another process in the meanwhile. Note that when using Full Vector clocks, e_j's presumption of the own key of p_i's latest event e_i preceding e_j is exactly equal to e_i's own key.

Lemma 2.4.4 *If $e_i \rightarrow e_j$ and $V_i[f_i(p_i)] = a$ and $V_j[f_j(p_i)] = a + B$ (for some $B > 0$) and there is no event e_i' by p_i such that $e_i \rightarrow e_i' \rightarrow e_j$ then for all e_i^l ($1 \le l \le x, x \le B$) such that $e_i^l \| e_j$ and that $e_i \rightarrow e_i^1 \rightarrow \ldots \rightarrow e_i^l \rightarrow \ldots \rightarrow e_i^x$, it is possible that $e_i^l \overset{NUREV}{\rightarrow} e_j$.*

Proof: Since $e_i \rightarrow e_j$, $xp(V_i) < xp(V_j)$. Process p_i must increase its vector by modifying at least its own entry ($f_i(p_i)$) by at least one in each such e_i^l, thus causing their timestamp vectors to compare similarly with V_j. Since $V_j[f_j(p_i)] = a + B$ this can cause NUREV timestamps to order e_j with up to B consecutive events of p_i which follow e_i and which are concurrent with e_j (cf. Figure 2.1, part A). □

This means that if e_j's presumption of the own key of p_i's last preceding event is *inflated* by B, this may result in e_j to be NUREV-ordered with a maximum number of B events of p_i which are in fact concurrent with e_j. Since the length of such a sequence of events e_i^l as described in the lemma is related to B, let us call it a B_{dep}-*error-prone sequence* of p_i caused by the event pair (e_i, e_j) (cf. Figure 2.1, part A). Moreover, let us define *time-to-hear-back*(e_i, p_j): if $e_i^1, \ldots, e_i^h (h \ge 1)$ is the minimal sequence of p_i's events between (i) event e_i by p_i that precedes an e_j of p_j such that there is no e_i' by p_i which $e_i \rightarrow e_i' \rightarrow e_j$ and (ii) an event e_i^{h+1} ($h \ge 0$) such that $e_j \rightarrow e_i^{h+1}$, then *time-to-hear-back*$(e_i, p_j) = h$. If there is no such e_i^{h+1}, consider instead the last event of p_i in the execution. In other words, after an event e_i that is directly preceding an event (e_j) of p_j,

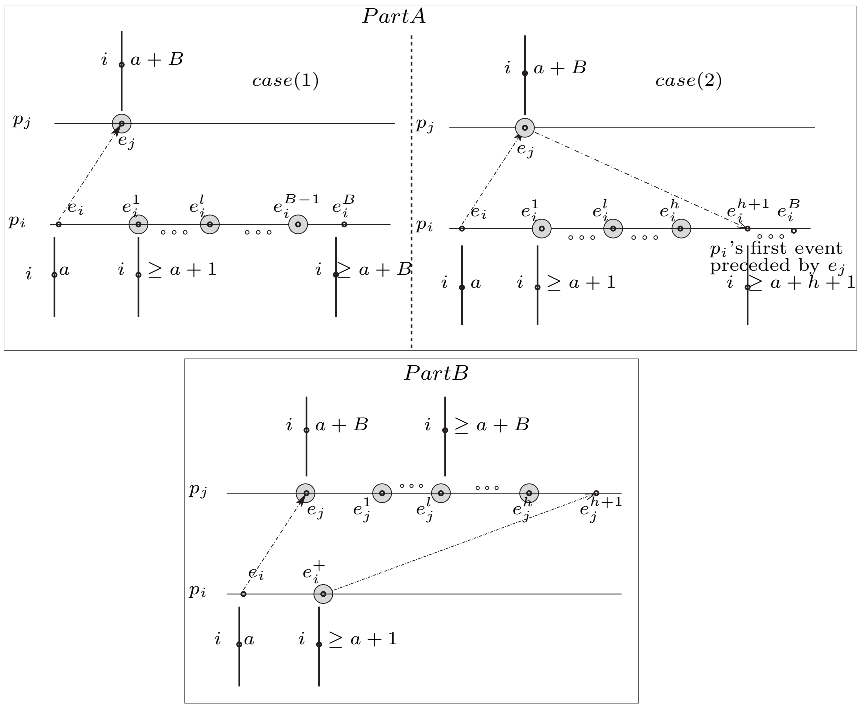


Figure 2.1: Possible errors by e_j's inflated presumption of e_i's own key (p_i's last event preceding e_j). Vertical lines represent events' timestamps. Circles indicate events that are pairwise concurrent but may be NUREV-ordered. **Part A:** The B_{dep}-*error-prone sequence* is bounded by B (case 1) and *time-to-hear-back*(e_i, p_j) (case 2). **Part B:** The B_{indep}-*error-prone sequence* is bounded by *time-to-hear-again*(p_i, e_j).

the *time-to-hear-back*(e_i, p_j) is the time (number of events) for p_i to hear back from p_j, i.e. to reach an event e_i^{h+1} that is directly preceded by e_j.

Lemma 2.4.5 *Given the precondition in lemma 2.4.4, the length of the corresponding* B_{dep}-error-prone sequence *is bounded from above by* time-to-hear-back(e_i, p_j).

Proof: The timestamp of e_j (or of a subsequent event that causes the precedence between e_j and e_i^{h+1}) informs p_i about the inflation of e_i's own value by p_j. Moreover, this precedence will actually establish order among the events of the processes (cf. Figure 2.1, part A). $\square$

The last lemma and its corollary imply that the length of a B_{dep}-*error-prone sequence* of some p_i caused by some (e_i, e_j) is bounded from above by (i) B, the difference between e_i's own key and e_j's presumption of it and (ii) the *time-to-*

hear-back(e_i, p_j).

Lemma 2.4.6 *Consider the case that* $e_i \rightarrow e_j$ *and there is no event* e'_i *by* p_i *such that* $e_i \rightarrow e'_i \rightarrow e_j$ *and* $V_i[f_i(p_i)] = a$ *and* $V_j[f_j(p_i)] = a + B$ *(for some* $B > 0$*). If there exists* e_i^+ *by* p_i *such that* $e_i \rightarrow e_i^+$ *and* $e_i^+ \| e_j^1$ *by* p_j *and* $e_i^+ \overset{NUREV}{\rightarrow} e_j^1$*, where* $e_j^1 = e_j$ *or* $e_j \rightarrow e_j^1$*, then* $e_i^+ \overset{NUREV}{\rightarrow} e_j^k$ *for each* e_j^k *such that* $e_j^1 \rightarrow e_j^k$*, and, specifically, there may exist a subsequence* $\{e_j^l\}$ *of the* e_j^k*'s by* p_j *where* $e_j^l \| e_i^+$*.*

Proof: Let V_e denote the vector of the timestamp of an event e. If $e_i^+ \overset{NUREV}{\rightarrow} e_j^1$ then from the definition of $\overset{NUREV}{\rightarrow}$ we have that $xp(V_{e_i^+}) < xp(V_{e_j^l})$ and from $e_j \rightarrow e_j^l$ and the NUREV update rules we have $xp(V_{e_j^1}) < xp(V_{e_j^l})$. This implies that $e_i^+ \overset{NUREV}{\rightarrow} e_j^l$. □

This means that if e_j's presumption of the own key of p_i's last preceding event (e_i) is *inflated* by B, this may result in some event(s) e_i^+ of p_i subsequent to e_i to be NUREV-ordered with a sequence of events of p_j subsequent to e_j but actually concurrent with e_i^+ (cf. Figure 2.1, part B). Since the length of such a sequence of e_j^l events as described in the lemma is not related to B, let us call it a B_{indep}-*error-prone sequence* of p_j caused by (e_i, e_j). Moreover, let us define *time-to-hear-again*(p_i, e_j) to be the length h of the sequence of p_j's events between e_j and the event e_j^{h+1} ($h \geq 0$) which is the first event by p_j that is preceded both by e_i and e_i^+ of p_i, where: e_i is the event by p_i directly preceding e_j, i.e. there is no e'_i by p_i such that $e_i \rightarrow e'_i \rightarrow e_j$; and e_i^+ is the event by p_i that is directly preceded by e_i. If there is no such e_j^{h+1}, consider instead the last event of p_j in the execution. Note that for infinite executions this implies that *time-to-hear-again*(p_i, e_j) can have an unbounded value. In other words, after an event (e_i) of p_i that is directly preceding an event e_j (of p_j), the *time-to-hear-again*(p_i, e_j) is the time, i.e. number of events, to hear again from p_i, i.e. to have the first event e_j^{h+1} at p_j which is directly preceded by an event of p_i subsequent to e_i.

Lemma 2.4.7 *Given the precondition in lemma 2.4.6 and the corresponding* B_{indep}-error-prone sequence*, the length of the sequence is bounded from above by* time-to-hear-again(p_i, e_j)*.*

Proof: We know from lemma 2.4.6 that given the preconditions there we have that $e_i^+ \overset{NUREV}{\rightarrow} e_j^k$ for each event e_j^k of p_j such that $e_j^1 \rightarrow e_j^k$. The B_{indep}-*error-prone sequence* starting at e_j^1 consists of e_j^1 and those e_j^k that are concurrent to e_i^+. Since the event e_j^{h+1}, which ends the *time-to-hear-again*(p_i, e_j) sequence, is preceded by e_i^+, the sequence of events e_j^k that are concurrent to e_i^+ has to end before e_j^{h+1}. So the length of the B_{indep}-*error-prone sequence* is bounded from above by *time-to-hear-again*(p_i, e_j). □

Combining lemmas 2.4.4, 2.4.5, 2.4.6 and 2.4.7, we get:

Corollary 2.4.1 *If during the clock update of an event e_j the presumed value of p_i's last preceding event e_i is inflated by $B > 0$, then the corresponding B_{dep}-*error-prone sequence *of p_i caused by (e_i, e_j) and the potential B_{indep}*-error-prone sequence *p_j caused by (e_i, e_j), together imply a total number of possibly NUREV-ordered concurrent events which may be as high as the product of the length of the two sequences. The total number of errors implied by the inflated value is bounded from above by:*

$$\begin{aligned} &min\{\text{time-to-hear-back}(e_i, p_j), B\} \times \text{time-to-hear-again}(p_i, e_j) \\ &\qquad\qquad \leq B \times \text{time-to-hear-again}(p_i, e_j) \end{aligned}$$

Hence, considering a process p_j deciding the new mapping when executing an event e_j, the goals of its adaptive mapping function to minimize the number of NUREV-ordered concurrent event pairs should be:

Inflation control

In the timestamp of e_j the sharing of entries must be arranged so that the inflation of the resulting timestamp's presumed values for each process' latest preceding event is kept minimal.

Next-Contact influence

In the timestamp of e_j the sharing of entries must be such that: the longer the time intervals (length of event sequences) until p_j and p_i communicate again after e_j (directly, or indirectly, via another process in the system), the smaller inflation is permitted in e_j's presumption about p_i's last preceding event.

The first conclusion implies that values with large differences should not be assigned to share the same entry, since the lower one must be equalized to the higher one to maintain plausibility. The second conclusion implies that an optimal adaptive mapping may need information about the *future* in order to decide how processes should share entries in a timestamp value. Such information cannot be assumed in general in distributed executions.

In the next sections we define algorithmic goals for satisfying the conclusions and we describe the design of algorithms for adaptive mapping functions to achieve them.

2.5 MinDiff NUREV clock

Armed with the conclusions above one can see that errors are introduced when a process' presumption of the own key of another process is inflated, so a good plausible clock should try to minimize this inflation.

There are two operations on a NUREV clock where inflation can occur. The first operation is the local step or the send operation where inflation can occur only if the local process shares its clock entry with another process. This can be avoided by always assigning a clock entry for the local process' own exclusive use. The second case is the receive operation, where it is impossible to avoid inflation in all cases since a clock with R entries cannot hold more than R different values.

We now introduce the MinDiff NUREV clock that aims at minimizing the inflation at each receive operation (it avoids inflation at local and send operations by always using an exclusive entry for its own key). How the new MinDiff clock value and mapping are calculated at a receive operation is described below and visualized by the example in Figure 2.2.

Let $\langle p_i, V_i, f_i\rangle$ be the current clock of process p_i when a timestamp $\langle p_j, V_j, f_j\rangle$ is received from process p_j. First the clock and the timestamp are merged into an N-entry vector W_i, where each entry is marked with its corresponding process:

$$\begin{aligned} W_i[k] &= \max(V_i[f_i(k)], V_j[f_j(k)]) \ \forall k \in \{1, \ldots, N\}; \\ W_i[k].id &= k \ \forall k \in \{1, \ldots, N\}. \end{aligned}$$

Note that there can be at most $2R$ different values in W_i as V_i and V_j can hold only R different values each.

In order to keep the inflation down, the MinDiff clock tries to minimize the sum of the inflation suffered by all process entries. To do this we need to select which processes should share clock entries in the new clock vector. Let $\hat{W}_i$ be all entries of W_i except $W_i[i]$, sorted in increasing order and $(C_{k,l})$ be a (lower triangular) cost matrix defined as follows:

$$\begin{aligned} C_{k,k} &= 0, \ k \in \{1, \ldots, N\}; \\ C_{k,l} &= C_{k,(l+1)} + \hat{W}_i[k] - \hat{W}_i[l] \text{ for } l < k. \end{aligned}$$

An entry $C_{k,l}$ is the cost (inflation) incurred by letting the set

$$\{p_n | n = \hat{W}_i[r].id \text{ for } k \leq r \leq l\}$$

of processes share the same entry in the updated clock vector. (The cost of computing $\hat{W}_i$ and C is $O(N)$ and $O(R^2)$, respectively.) A minimal-cost-assignment of processes to clock-entries corresponds to a partitioning of the sorted sequence $\hat{W}_i$ into $R-1$ blocks (one clock entry is reserved for the process i itself, to avoid recomputing the mapping function upon send and local events).

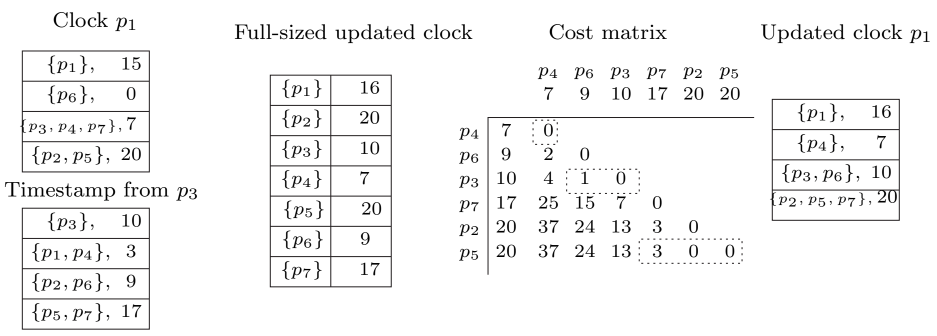
In Algorithm 2.1 we present a new algorithm that finds a good K-partitioning of a L-element sequence in $O(K \cdot L)$ steps. Initially the algorithm places the $K-1$ partition boundaries in at the first $K-1$ of the $L-1$ places where a partition boundary can be placed. Let b_k be the position of the k-th partition boundary and let b_0 and b_K denote position 0 and L, respectively. Define the cost of the partitioning $b_1, \cdots, b_{K-1}$ as

$$cost(b_1, \cdots, b_{K-1}) = \textstyle\sum_{k=1}^{K} C_{b_k, b_{k-1}+1}.$$

The partitioning proceeds by selecting the rightmost partition boundary and moving it to the right until it reaches a local cost minimum. Then it selects

Algorithm 2.1 The K-partitioning algorithm used by MinDiff.

```
$b_0 \leftarrow 0$; $b_K \leftarrow L$; $b_k \leftarrow k$ for $k = 1, \ldots, K-1$
repeat
  for $k = 1, \ldots, K-1$ do
    while $cost(b_1, \cdots, b_k, \cdots, b_{K-1}) >$
      $cost(b_1, \cdots, b_k + 1, \cdots, b_{K-1})$
      and $b_k + 1 < b_{k+1}$ do $b_k \leftarrow b_k + 1$
until no $b_k$ changed.
```



Clock p_1

$\{p_1\}$,	15
$\{p_6\}$,	0
$\{p_3, p_4, p_7\}$,	7
$\{p_2, p_5\}$,	20

Timestamp from p_3

$\{p_3\}$,	10
$\{p_1, p_4\}$,	3
$\{p_2, p_6\}$,	9
$\{p_5, p_7\}$,	17

Full-sized updated clock

$\{p_1\}$	16
$\{p_2\}$	20
$\{p_3\}$	10
$\{p_4\}$	7
$\{p_5\}$	20
$\{p_6\}$	9
$\{p_7\}$	17

Cost matrix

$$\begin{array}{c|cccccc} & p_4 & p_6 & p_3 & p_7 & p_2 & p_5 \\ & 7 & 9 & 10 & 17 & 20 & 20 \\ \hline p_4 & 7 & 0 & & & & \\ p_6 & 9 & 2 & 0 & & & \\ p_3 & 10 & 4 & 1 & 0 & & \\ p_7 & 17 & 25 & 15 & 7 & 0 & \\ p_2 & 20 & 37 & 24 & 13 & 3 & 0 \\ p_5 & 20 & 37 & 24 & 13 & 3 & 0 & 0 \end{array}$$

Updated clock p_1

$\{p_1\}$,	16
$\{p_4\}$,	7
$\{p_3, p_6\}$,	10
$\{p_2, p_5, p_7\}$,	20

Figure 2.2: Example of MinDiff clock update at message reception. (See section 2.5 for the details.)

the next partition boundary to the left and moves it to the right until it either finds a local minimum or reaches the first boundary. This procedure is repeated for all boundaries until no changes occur. Note that since the boundaries only move to the right, the algorithm needs at most $O(K \cdot L)$ steps to terminate. (In our case K is $R - 1$ and L is at most $2R$.)

Once the partitioning of the processes into $R - 1$ blocks is decided, the new mapping function is determined by letting the processes in each block share an entry. The new clock entry values can be computed according to the NUREV rules (in fact, the new value for each entry r is simply $\hat{W}_i[b_r]$).

To store any arbitrary mapping that comes as a result of the grouping of entries according to the MinDiff clock in the timestamp requires N values of $\log_2 R$ bits each, so the total space requirement of a MinDiff timestamp would be R sizeof(*clock entry*) $+ N \log_2 R$ bits. This is significantly smaller than the N clock entries required by the full-size vector clock — cf. figure 2.3 for a comparison of the growth of the corresponding timestamp values as the system size grows. Still, it is challenging to study how to follow the conclusions of our analysis for mappings with constant-size representation.

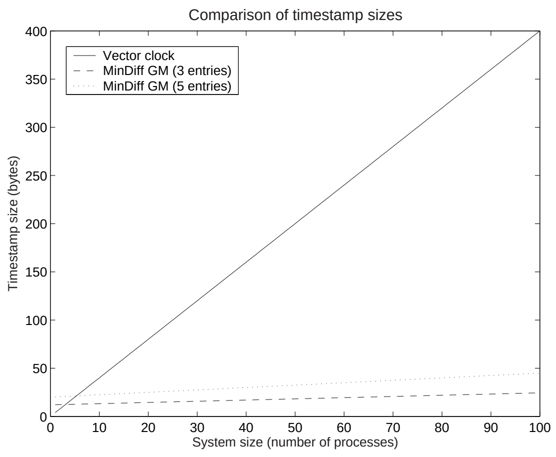


Figure 2.3: The growth of timestamp size as a function of the number of processes. The size of a clock entry is 4 bytes.

2.6 ROV-MRS NUREV clock

Consider a NUREV clock which, at each process, maps $R-1$ of the other processes to exclusive clock entries and all other processes to the remaining clock entry, called the "others entry". The owner of a clock and the source process of a timestamp are always assigned an exclusive clock entry, for the same reason as in the MinDiff clock. We call this clock the *R-others vector clock* (ROV). The formal description of the ROV clock algorithm is given in figure 2.4.

This mapping-class has constant-size representation, but the issue of which processes should be allocated to exclusive clock entries in each process' clock remains to be solved. Following the conclusions from section 2.4, we propose the following strategy:
Most Recent Senders (ROV-MRS) mapping: In this policy the $R-2$ last processes the process received messages from are mapped to exclusive entries in the clock. If there are less than $R-2$ such processes, that is, there are some unused exclusive entries, those entries are allocated to processes that had exclusive entries in the most recently received timestamps. All other processes (apart from the process itself) are mapped to the others-entry. Since each process assigns exclusive entries to its most recent senders, it will thus add *no inflation* to their entries. Since these processes are likely to take longer than others to send something again (*time-to-hear-again* may be longer than the other processes), such a mapping would agree well with the [Next-Contact]-conclusion of the analysis.

It is worth pointing out that considering the symmetric strategy, namely a Least Recent Senders (ROV-LRS) mapping, the results from section 2.4 argue against it: a process which is a non-sender for long time might send a message

Let $ROV = \left(\langle S, \overset{ROV}{\rightarrow}\rangle, ROV.\mathbf{stamp}\right)$ where

- S is a set of tuples of the form $\langle p_i, V_i, f_i\rangle$ where p_i is an integer that identifies each process in the system, V_i is a $1-$dimensional vector of R integers and f_i is a function from process-IDs to $\{1, \ldots, R\}$ such that p_i and at most $R-2$ other process-IDs are bijectively mapped to $\{1, \ldots, R-1\}$ and all other process-IDs are mapped to R

- $ROV.\mathbf{stamp}$ is defined by the rules
  ROV0) Initial value:
  $$\begin{aligned} p_i &= \text{unique process-ID} \in \{1, \ldots, N\} \ ; \\ f_i &= \{p_i \mapsto 1, \forall j \neq p_i . j \mapsto R\} \ ; \\ V_i[f_i(j)] &= 0 \ \forall j \in 1, \ldots, N \end{aligned}$$
  ROV1) Before a send or local event with timestamp $\langle p_i, V_i^+, f_i\rangle$ is generated:
  $$\begin{aligned} f_i & \quad \text{is not changed} \ ; \\ V_i^+[f_i(p_i)] &= V_i[f_i(p_i)] + 1. \end{aligned}$$
  ROV2) When a message with time-stamp $\langle p_s, V_s, f_s\rangle$ is received:
  $$\begin{aligned} f_i^+ &= \text{updated } f_i \ ; \\ V_i[r] &= \max\big\{\max(V_i[f_i(j)], V_s[f_s(j)]) : \\ & \qquad \forall j . f_i^+(j) = r\big\} + own(r). \end{aligned}$$

- Let $\langle p_i, V_i, f_i\rangle, \langle p_j, V_j, f_j\rangle \in S$ then:
  $\langle p_i, V_i, f_i\rangle \overset{ROV}{\rightarrow} \langle p_j, V_j, f_j\rangle \Leftrightarrow$
  $(p_i = p_j \wedge V_i[f_i(p_j)] < V_j[f_j(p_j)]) \vee (p_i \neq p_j \wedge$
  $(\forall k . V_i[f_i(k)] \leq V_j[f_j(k)]) \wedge (V_i[f_i(p_j)] < V_j[f_j(p_j)]))$

Figure 2.4: Definition of the R-others vector clock. Note that the set of allowed mapping functions is restricted to those where only entry R is shared by more than one process. The operation updating the mapping should only produce mappings from that set.

soon (i.e. *time-to-hear-again* can be short) and hence establish order among events, so it may not need an exclusive entry; that entry could be used to prevent other errors instead.

2.7 Experimental evaluation

We first study *peer-to-peer* communication systems the main target application domains for such algorithms. Peer-to-peer systems can have a wide variety of communication patterns, here we study such systems with uniform random as well as more structured clustered random communication patterns. As shown in section 2.4, the communication patterns play a special role in the accuracy of the time-stamping systems. We study the *client-server* communication systems separately, as they have very different communication patterns. In particular, considering that direct communication will only take place between the clients and the server(s), the servers have a key-role, since they are likely the ones to cause and propagate a large portion of the errors. A parallel consideration is that the servers may also play key-role in the *actual ordering* of the events (requests). However, for the purpose of enhancing the understanding of plausible clocks, we wish to discuss the algorithms' accuracy in such communication patters, as well. The conclusions of section 2.4 are compared with the outcome of the evaluation. The plausible clocks we focus on are: the *R-Entries Vector Clock* (REV) [TRA99], our *R-Others Vector Clock* with the *Most Recent Senders* (ROV-MRS) dynamic mapping and our MinDiff clock.

The k-Lamport clocks [TRA99] were not included as their behaviour is such that their accuracy for small k is rather low, while after the first few values of k, adding more entries in the vector, the accuracy does not improve. This is explained by our analysis, since these clocks do not keep per-process information in the clock. Similar is the effect of combining k-Lamport clocks with other plausible clocks using the methodology in [TRA99]. Our experiments confirmed these conclusions. An experimental study on how the performance of the combination of REV and k-Lamport clock depends on different system parameters, such as system size, communication pattern and local history length, is presented in [TR01].

Experiments

The experiments were conducted by creating system histories of simulated distributed systems and annotating the events with timestamps from a number of different plausible clocks and also a full vector clock to measure the accuracy of the plausible clocks. The system history is generated by letting each (client/peer) process randomly select whether to send or receive a message or to perform a local step. The destination of each message is selected at random in the peer-to-peer and clustered peer-to-peer communication cases. In the client-server case the server processes requests immediately and in FIFO order. Figure 2.5, Figure 2.6 and Figure 2.7 shows results from the peer-to-peer, clustered peer-to-peer and client-server executions, which will be discussed further below.

2.7.1 Peer-to-peer communication

Experiment description

The system history is generated by letting each process randomly select whether to send or receive a message or to perform a local step. The destination of each message is selected uniformly at random.

Figure 2.5 presents results from peer-to-peer experiments. In the experiments there are 80 processes. The probability of sending or receiving a message is 0.15.

Discussion of results

The MinDiffclock shows very good accuracy even for very small number of clock entries (i.e. small R), as expected from the analysis and its design to follow the [Inflation]-conclusion. However, the uniform random communication leaves few communication patterns that could be exploited and as the length of the execution grows and the relative significance of the start up and close down phases diminish MinDiff's advantage to, e.g., the *REV* clock, which uses an arbitrary but uniform assignment of processes to clock entries and copes with the randomness slightly better.

With the good accuracy of MinDiff at small numbers of entries combined with Figure 2.3, which compares the MinDiff's and Full Vector Clock's timestamp sizes, the results look promising from the applicability point of view, especially when considering the desire for scalability in peer-to-peer systems. From the intellectual-challenge point-of-view, constant-size representation of mapping functions may deserve more investigation for proving bounds in their relative performance.

The *R-Others Vector Clock* with the *Most Recent Senders* dynamic mapping (ROV-MRS) shows comparable or better accuracy than *REV* clock for small numbers of clock entries (i.e. small R).

2.7.2 Clustered peer-to-peer communication

Experiment description

The system history is generated by randomly assigning each process to one of the clusters, each process will then randomly select whether to send message to a process within the cluster, send a message to a process in another cluster, receive a message or perform a local step. The destination of each message, within or outside the cluster, is selected at random.

Figure 2.6 presents results from the clustered peer-to-peer experiments. In the experiments there are 80 processes divided into 7 clusters. The probability of

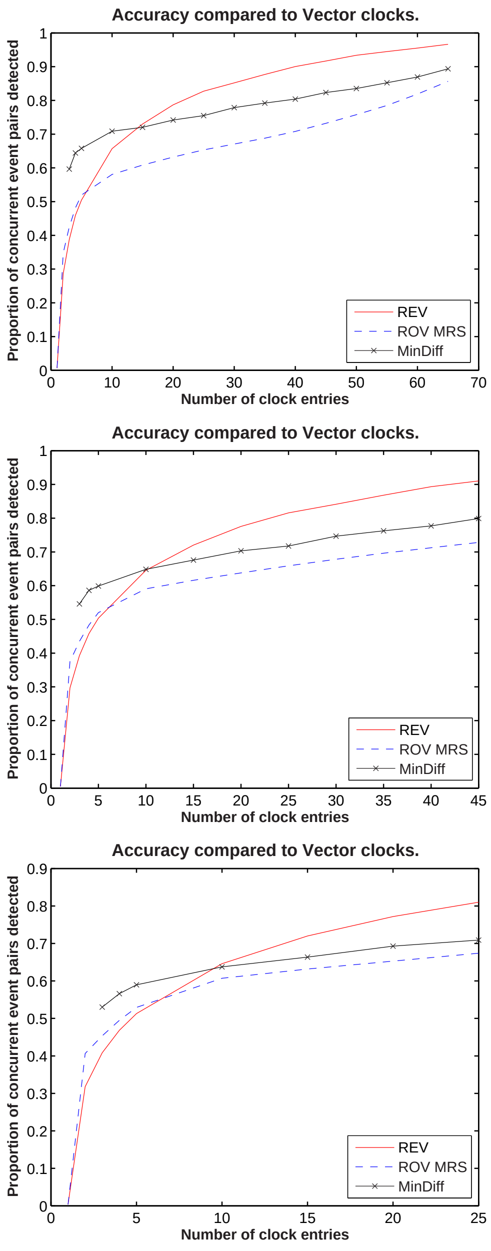


Figure 2.5: 80-process peer-to-peer systems with local history length of 250, 500 and 1000 events. The number of concurrent/total event pairs is 115 762 669/198 732 016, 318 673 228/798 700 528 and 813 886 635/3 177 599 340, respectively.

sending a message to a destination within the cluster is 0.15 while the probability to send to a destination outside the cluster is 0.01.

Discussion of results

The communication patterns in the clustered peer-to-peer executions are more structured and have a higher degree of concurrency since there are few causal dependencies between the clusters. This is highly beneficial for MinDiff and the *R-Others Vector Clock* with the *Most Recent Senders* policy as can be seen in Figure 2.6.

Due to the clustered nature of the system it is likely that the own entries of processes in the same cluster advance at approximately the same rate, while the processes' presumptions about the own entries of processes in other clusters stay constant most of the time since they are only updated when rare cross-cluster communication occur. In this type of execution the MinDiff clock can avoid inflation by grouping processes from the other clusters to a small number of entries (e.g. one entry per cluster) and processes in the same cluster to one or more entries. This agrees well with the [Inflation]-conclusion.

The *R-Others Vector Clock* with the *Most Recent Senders* will most often use single entries for (some of) the processes in the same cluster while processes in other clusters will all use the others-entry.

2.7.3 Client-server communication

Experiment description

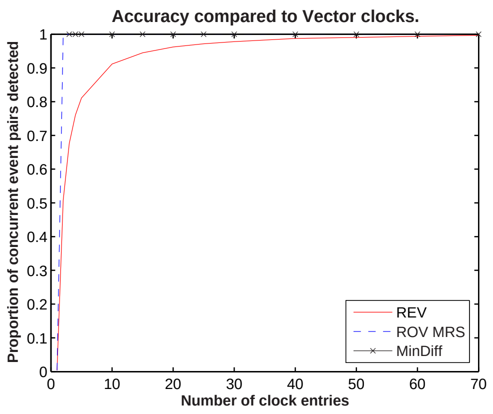
The system history is generated by letting each client randomly decide whether to send a request, receive a response or do a local step. The server processes requests immediately and in FIFO order. Figure 2.7 shows results from client-server executions. The probability for a client to send a request to or receive a response from the server in a time step is 0.15.

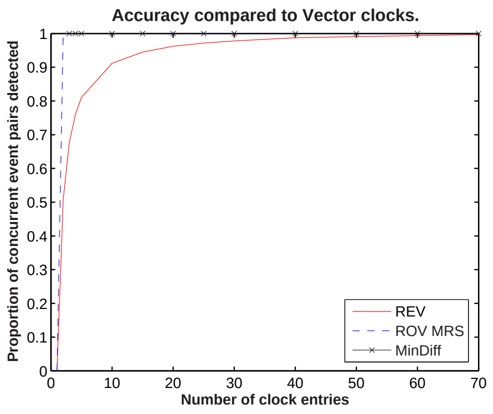
Discussion of results

Consider the servers' and the clients' perspectives separately:

When a *server* receives a message from a client:

- The *time-to-hear-back* (from the server) for that client is likely to be very small, since in most cases the client simply waits for the server's reply. Minimizing the inflation is the key to accuracy here, justifying the MinDiff clock.
- The *time-to-hear-again* from that client might be large, hence it may be better to have unique entries for each of the recently requesting clients following conclusion [Next-Contact] of section 2.4, justifying the ROV-MRS policy. However, later in the execution, that client may be displaced in the server's clock by new, more recently requesting clients (if there are more than $R-2$ of them). In that case it will have to share the others-entry with all other processes, resulting in inflation of its value or the other processes'





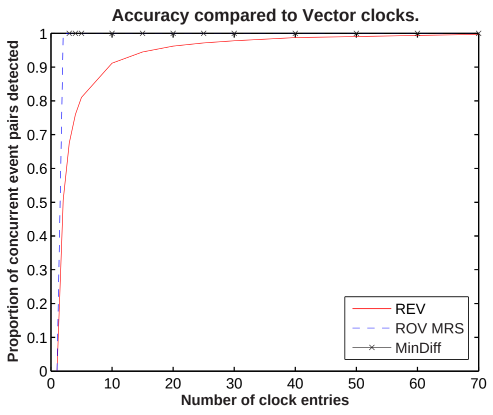
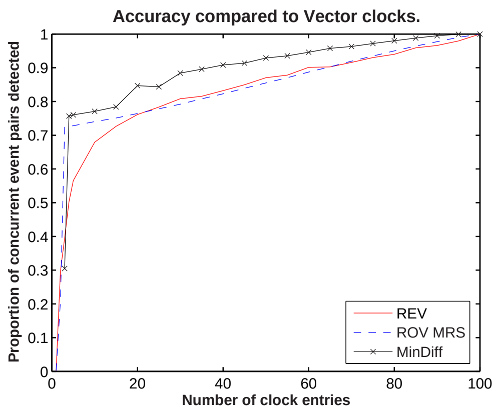


Figure 2.6: 80-process clustered peer-to-peer systems with 7 clusters and local history length of 250, 500 and 1000 events. The number of concurrent/total event pairs is 31307142/31700703, 196688513/199170861, 789520315/799500078 and 3 135 312 726/3 174 969 141, respectively.



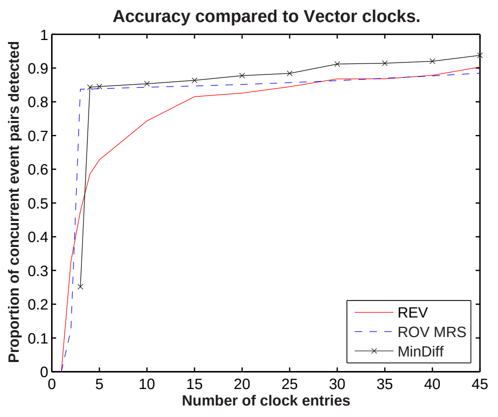


Figure 2.7: 1-server-99-client systems with local history length of 100 and 500 events. The number of concurrent/total event pairs is 28 597 743/80 334 150 and 292 840 959/2 063 356 680, respectively.

values. (In the many servers case and if the nodes form clusters (disjoint or with small overlaps), the method should be appropriate, especially if the cluster size is smaller than or close to R.)

- Since the *time-to-hear-again* (from the same client) can be *arbitrarily large* any (even small) inflation can cause *arbitrarily many errors* in any of the algorithms (cf. lemma 2.4.6). MinDiff (which uses the minimization of inflation as a tool), tends to show better performance than REV, (which uses the arbitrary mapping as a tool).

The latter argument, from the perspective of a *client* shows that:

- When a client inflates the value of another client, both the *time-to-hear-again* and *time-to-hear-back* between the two clients depend on when the next requests from *both* the clients will be issued (they communicate indirectly, via the server). These values can be arbitrarily large, even unbounded if any of the two clients stops issuing requests.
- When a client inflates the value of the server, the client may have some estimation on the *time-to-hear-back* (server from client) and *time-to-hear-again* (client from server) values, e.g. if it knows when it will send its next request.

A general conclusion is that the particular communication patterns play a very significant role in the client-server communication case. An algorithm that could give guarantees would need information about the future (e.g. knowledge or estimation of the request frequency), as also concluded in section 2.4. If such information is available or predictable, it may be possible to have even better performance from within applications, by adopting next-contact-aware conditions in the update of the mapping functions.

2.8 Discussion

Logical clocks have been studied extensively in the distributed computing literature. Still there are aspects of them which need to be discovered to enhance performance and scalability towards satisfying needs of future distributed systems, e.g. in the context of scalable multicast and collaborative applications in peer-to-peer systems. This paper makes steps for an in-depth study on the accuracy of vector timestamps with fixed and small number of entries, aiming at scalable solutions for large systems. The work builds on the work of Torres-Rojas and Ahamad [TRA99], where the notion of plausible clocks and some plausible clock algorithms were introduced.

In particular, our contributions are the following: (i) We introduce the Non-Uniformly Mapped R-Entries Vector (NUREV) clocks, a general class of clocks that extends and includes the R-Entries Vector (REV) clocks algorithm of [TRA99] and the Full Vector clocks. With NUREV clocks each process in the system can use a different mapping between process-IDs and clock-entry indices, the idea being that dynamic mappings may allow self-tuning and adaptation to improve the accuracy of the clocks. (ii) We prove that NUREV clocks are plausible. This makes it easier to design new adaptive plausible clock algorithms. (iii) Furthermore, we analyze the ways that these clocks may relate causally independent event pairs. Our analysis resulted in a set of criteria to concentrate on in order to improve performance. (iv) These, in turn, resulted in new adaptive mapping strategies, MinDiff and ROV-MRS, which show very competitive performance for small clock/timestamp sizes, that is, where it matters in practice. The experimental evaluation of the performance of our proposed methods agrees with the conclusions of the analysis part and also shows promising results from the applicability point of view.

2.9 Future Work

Our work points to new issues that need investigation. One of them is the issue of constant-size representation of mapping functions, for example while our MinDiff clock provides outstanding performance for small timestamps, it is, strictly speaking, not a fixed-size clock, even though its timestamp size grows very slowly with N. Although from the applicability point of view, as is shown in the paper, this is no obstacle to the performance and scalability of the system, from the intellectual point of view, it can be a challenging issue. Possible directions to investigate this include the use of appropriate approximated mappings (e.g. low-pass filters or polynomials) or the use of smaller sets of mapping functions that can be represented in constant space. Other important and challenging research issues that follow from this research are (i) to study the performance and accuracy of plausible clocks from an information-theory point of view, (ii) to bound the size of the vector entries and (iii) to consider dynamic group sizes and other varying parameters.

Acknowledgements

We wish to thank Philippas Tsigas for the fruitful discussions; Niklas Elmqvist, Ha Hoai Phuong and Håkan Sundell for their feedback on a draft of this paper; the anonymous reviewers of ICDCS'04 for their constructive comments.

Chapter 3

Lightweight Causal Cluster Consistency[1]

Anders Gidenstam Boris Koldehofe

Marina Papatriantafilou Philippas Tsigas

Abstract

Within an effort for providing a layered architecture of services supporting multi-peer collaborative applications, this paper proposes a new type of consistency management aimed for applications where a large number of processes share a large set of replicated objects. Many such applications, like peer-to-peer collaborative environments for training or entertaining purposes, platforms for distributed monitoring and tuning of networks, rely on a fast propagation of updates on objects, however they also require a notion of consistent state update. To cope with these requirements and also ensure scalability, we propose the *cluster consistency* model. We also propose a two-layered architecture for providing cluster consistency. This is a general architecture that can be applied on top of the standard Internet communication layers and offers a modular, layered set of services to the applications that need them. Further, we present a *fault-tolerant protocol* implementing causal cluster consistency with predictable reliability, running on top of decentralized probabilistic protocols supporting group communication. Our experimental study, conducted by implementing and evaluating the two-layered architecture on top of standard Internet transport services, shows that the approach scales well, imposes an even load on the system, and provides high-probability reliability guarantees.

Keywords: large scale group communication, consistency, collaborative environments, peer-to-peer systems, Internet application layer services.

[1]This is an extended version of the paper that appeared in the Proceedings of IICS 2005, Paris, France, June 20-22, 2005 .

3.1 Introduction

Many applications like collaborative environments (e.g. [MT95, GB97, CH93]) allow a possibly large set of concurrently joining and leaving processes to share and interact on a set of common replicated objects. State changes on the objects are distributed among the processes by update messages (a.k.a. *events*). Providing the infrastructure to support such applications and systems places demands for multi-peer communication, with guarantees on reliability, latency, consistency and scalability, even in the presence of failures and variable connectivity of the peers in the system. Applications building on such systems would also benefit from an event delivery service that satisfies the causal order relation, i.e. satisfies the "happened before" relation as described in [Lam78].

The main focus of earlier research in distributed computing dealing with these issues has its emphasis in proving feasible, robust solutions for achieving reliable causal delivery in the occurrence of faults [BJ87, BSS91, RST91, KS98], rather than considering the aforementioned variations in needs and behaviour. Further, since the causal order semantics require that an event is delivered only after all causally preceding events have been delivered, the need to always recover lost messages can lead to long latencies for events, while applications often need short delivery latencies. Moreover, the latency in large groups can also become large because a causal reliable delivery service needs to add timestamp information, whose size grows with the size of the group, to every event.

To improve the latency, *optimistic causal order* [BPRS98, RBAR00] can be suitable for systems where events are associated with deadlines. In contrast to the strict causal order semantics, optimistic causal order only ensures that no events that causally precede an already delivered event are delivered. Events that have become obsolete do not need to be delivered and may be dropped. Nevertheless, optimistic causal order algorithms aim at minimizing the number of lost events. In order to determine the precise causal relation between pairs of events in the system, processes can use *vector clocks* [Mat89], which also allow detection of missing events and their origin. However, since the size of the vector timestamps grows linearly with the number of processes in the system, one may need to introduce some bound on the growing parameter to ensure scalability.

Recent approaches for information dissemination use lightweight probabilistic group communication protocols [BHO$^+$99, EGH$^+$01, GKM01, Kol03, PRMK03, BEG04]. These protocols allow groups to change over time and to scale to many processes by providing reliability expressed with high probability. In [PRMK03] it is shown that probabilistic group communication protocols can perform well also in the context of collaborative environments. However, per se these approaches do not provide any ordering guarantees.

In this paper we propose a consistency management method called *causal cluster consistency*, providing optimistic causal delivery of update messages to a large set of processes. Causal cluster consistency takes into account that for many applications the number of processes which are interested in performing updates can be low compared to the overall number of processes which are interested in receiving updates and maintaining replicas of the respective objects.

Therefore, the number of processes that are entitled to perform updates at the same time is restricted to n, which also corresponds to the maximum size of the vector clocks used. However, the set of processes entitled to perform updates is not fixed and may change dynamically.

Our proposed approach is in line with and inspired from recent approaches in multipeer information dissemination [BHO$^+$99, EGH$^+$01, GKM01], where the aim is at what is called *predictable reliability*, guaranteeing that each event is delivered to all non-faulty destinations with a high-probability guarantee. We present a two-layer architecture implementing cluster consistency that can make use of lightweight communication algorithms which can in turn run using standard Internet (or other) transport services. Our method is also designed to tolerate a bounded number of process failures, by using a combined push-and-pull (recovery) method. We also present an implementation and experimental evaluation of the proposed method and its potential with respect to reliability and scalability, by building on recently evolved large-scale and lightweight probabilistic group communication protocols. Our implementation and evaluation have been carried out in a real network, and also in competition with concurrent network traffic by other users.

Also of relevance and inspiration to this work is the recent research on peer-to-peer systems and in particular the methods of such structures to share information in the system (cf. e.g. [SMK$^+$01, AGBH03, RFH$^+$01, RD01, ZHS$^+$04]), as well as a recent position paper for atomic data access on CAN-based data management [LMR02].

The paper is structured as follows: in Section 3.2 notation and definitions are given, in Section 3.3 we introduce a layered architecture for achieving causal delivery and the two-layered protocol implementing it. Section 3.4 discusses the implementation and experimental evaluation of the proposed protocol running on top of standard Internet transport services. The paper concludes with a discussion of the contribution and future work.

3.2 Notation and problem statement

Let $G = \{p_1, p_2, \ldots\}$ denote a group of processes, which may dynamically join and leave, and a set of replicated objects $B = \{b_1, b_2, \ldots\}$. Processes maintain replicas of objects they are interested in. Let B be partitioned into disjoint clusters $C_1, C_2, \ldots$ with $\cup_i C_i \subseteq B$. Further, let C denote a cluster and p a process in G, then we write also $p \in C$ if p is interested in objects of C. *Causal cluster consistency* allows any processes in C to maintain the state of replicated objects in C by applying updates in optimistic causal order. However, at most n processes (n is assumed to be known to all processes in C) may propose updates to objects in C at the same time. Processes which may propose updates are called *coordinators* of C. Let $Core_C$ denote the set of coordinators of C. The set of coordinators can change dynamically over time. Throughout the paper we will use the term *events* when referring to update messages sent or received by processes in a cluster.

The propagation of events is done by multicast communication. It is not assumed that all processes of a cluster will receive an event which was multicast, nor does the multicast need to provide any ordering by itself. Any lightweight probabilistic group communication protocol as appears in the literature [EGH$^+$01, GKM01, Kol03] would be suitable. We refer to such protocols as *PrCast*. PrCast is assumed to provide the following properties: (i) an event is delivered to all destinations with high probability; and, (ii) decentralized and lightweight group membership, i.e. a process can join and leave a multicast group in a decentralized way and processes do not need to know all members of the group.

Within each cluster we apply vector timestamps of the type used in [ANB$^+$95]. Let the coordinator processes in $Core_C$ be assigned to unique identifiers in $\{1, \ldots, n\}$ (a process which is assigned to an identifier is also said to *own* this identifier). Then, a time stamp t is a vector whose entry $t[j]$ corresponds to the $t[j]$th event send by a process that *owns* index j or a process that owned index j before (this is because processes may leave and new processes may join $Core_C$). A vector time stamp t_1 is said to be smaller than vector time stamp t_2 if $\forall i \in \{1, \ldots, n\}$ $t_1[i] \leq t_2[i]$ and $\exists i \in \{1, \ldots, n\}$ such that $t_1[i] < t_2[i]$. In this case we write $t_1 < t_2$.

For any multicast event e, we write t_e for the corresponding timestamp of e. Let e_1 and e_2 denote two multicast events in C, then e_1 causally precedes e_2 if $t_{e_1} < t_{e_2}$, while e_1 and e_2 are said to be concurrent if neither $t_{e_1} < t_{e_2}$ nor $t_{e_2} < t_{e_1}$. Further we denote the index owned by the creator of event e as $index(e)$ and the event id of event e as $\langle index(e), t_e[index(e)]\rangle$.

Throughout the paper it is assumed that each process p maintains for each cluster C a *cluster-consistency-tailored logical vector clock* (for brevity also referred to as a CCT-*vector clock*) denoted by $clock_p^C$. A CCT-vector clock is defined to consist of a vector time stamp and a sequence number. We write T_p^C when referring to the timestamp and seq_p^C when referring to sequence number of $clock_p^C$. T_p^C is the timestamp of the latest delivered event while seq_p^C is the sequence number of the last multicast event performed by p. In Section 3.3 when describing the implementation of causal cluster consistency, we explain how these values are used. Note, whenever we look at a single cluster C at a time, we write for simplicity $clock_p$, T_p, and seq_p instead of $clock_p^C$, T_p^C, and seq_p^C respectively.

3.3 Layered protocol for optimistic causal delivery

This section proposes a layered protocol for achieving optimistic causal delivery. Here we assume that coordinators of a cluster are assigned to vector entries and that the coordinators of a cluster know each other. To satisfy these requirements we choose a decentralized and fault-tolerant cluster-management protocol [GKPT05a] which can map a process to a unique identifier in the CCT-vector clock in a decentralized way and can inform all processes in $Core_C$ about this mapping.

3.3.1 Protocol description

The first of the two layers uses *PrCast* in order to multicast events inside the cluster (cf. pseudo-code description Algorithm 3.1). The second layer, the causality layer, implements the optimistic causal delivery service. The causal delivery protocol is inspired by the protocol by Ahamad et. al. [ANB$^+$95] and is adapted and enhanced to provide the optimistic delivery service of the cluster consistency model and the recovery procedure for events that may be missed due to *PrCast*.

Each process in a cluster interested in observing events in optimistic causal order (which is always true for a coordinator), maintains a queue of events denoted by H_p^C. For any arriving event e one can determine from T_p^C and the event's timestamp t_e whether there exist any events which (i) causally precede e, (ii) have not been delivered, and (iii) could still be deliverable according to the optimistic causal order property. More precisely we define this set of not yet delivered deliverable events as

$$\mathit{to\_deliver\_before}(e) = \{e' \mid t_{e'} < t_e \wedge \neg(t_{e'} < T_p^C)\}$$

and their event ids, which can be used for recovery, can be calculated as follows

$$\begin{aligned}\mathit{to\_deliver\_before\_ids}(e) = \{\langle i, j\rangle \quad | \quad & (\forall i \neq \mathit{index}(e)\ .\ T_p^C[i] < j \leq t_e[i]) \\ & \vee\ (i = \mathit{index}(e) \wedge T_p^C[i] < j < t_e[i])\}.\end{aligned}$$

If there exist any such events, e will be enqueued in H_p^C until those events have been delivered or e is about to become obsolete at which point e will be delivered. (prior to that process p may "pull" missing events —see below). Otherwise, p delivers e to the application.

When a process p delivers an event e referring to cluster C, the CCT-vector clock clock_p^C is updated by setting $\forall i\ T_p^C[i] = \max(t_e[i], T_p^C[i])$. Process p also checks whether any events in H_p or recovered events now can be dequeued and delivered. Before a coordinator p in Core_C, owning the jth vector entry, multicasts an event it updates clock_p^C by incrementing seq_p^C by one. The event is then stamped with a vector timestamp t such that $t[i] = T_C^p[i]$ for $i \neq j$ and $t[j] = \mathit{seq}_p^C$.

Since PrCast delivers events with high probability, a process may need to recover some events. The recovery procedure, which is invoked when an event e in H_p is close to become obsolete, sends recovery messages for the missing events that precede e. The time before e becomes obsolete depends the amount of time since the start of the dissemination of e, and is assumed to be larger than the duration of a PrCast (which is estimated by the number of hops that an event needs to reach all destinations with w.h.p.) and the time it takes to send a recovery message and receive an acknowledgement. At the time $e \in H_p$ becomes obsolete, p delivers all recovered events and events in H_p that causally precede e and e in their causal order. A simple recovery method is to contact the sender of the missing event. For this purpose the sender has a *recovery buffer* which stores events until no more recovery messages are expected (this is e.g. the case if $\forall i \; t_e[i] < T_p^C[i]$). Below we will present and analyze a another recovery method that enhances the throughput and the fault-tolerance.

3.3.2 Properties of the protocol

The PrCast protocol provides a delivery service that guarantees that an event will reach all its destinations with high probability, i.e. PrCast can achieve high message stability. When an event needs recovery, the number of processes that did not receive the event is expected to be low. Thus a process multicasting an event is expected to receive a low number of recovery messages. If there are no process, link or timing failures, reliable point to point communication succeeds in recovering all missing events, and thus provides causal order without any message loss. The following lemma is straightforward, following the analysis in [ANB$^+$95].

Lemma 3.3.1 *An execution of the two-layer protocol guarantees causal delivery of all events disseminated to a cluster if neither processes nor links are slow or fail.*

3.3.3 Event recovery, fault-tolerance and throughput

The throughput and fault-tolerance of the protocol can be increased by introducing redundancy in the recovery protocol. All processes could be required to keep a history of some of the observed events, so that a process only needs to contact a fixed number of other processes to recover an event. Further, such redundancy could help the recovery of a failed process. As it is desirable to bound the size of this buffer we analyze the recovery buffer size and number of processes to contact such that the recovery succeeds with high probability.

Following [Kol03], we describe a model suitable to determine the probability for availability of events that are deliverable and may need recovery in an arbitrary system consisting of a cluster C of n processes that communicate using the Two-Layer protocol. Let $\mathcal{C}$ denote this system and T denote the time determined by the number of rounds an event stays at most in $\mathcal{C}$. Note the similarity of the buffer system to a single-server queueing system, where new events are admitted to the queue as a random process. However, unlike common queueing systems, the service time (time needed for all processes in C to get the event using the layered protocol) in this model depends on the arrival times of events. The service time is such that every event stays at least as long in the queue as it needs to stay in the buffer of $\mathcal{C}$ in order to guarantee delivery/recovery (i.e. whether the queue is stable is not an issue here). Below we estimate the probability that the length of the queue exceeds the choice of the length for the recovery buffer of $\mathcal{C}$.

Algorithm 3.1 Two-Layer protocol for causal cluster consistency.

```
VAR
  $H_p$:          set of received events that can not be delivered yet
  $R$:            set of recovered events that can not be delivered yet
  $B$:            fixed size recovery buffer with FIFO replacement.
  $seq_p^C$:      sequence number of the last event created by a process owning
                  the identifier $p$ in $Core_C$.
  $T_p^C$:        vector timestamp indicating the causal present for $p$.
On process $p$ in $Core_C$ creates the event $e$
  $seq_p^C := seq_p^C + 1$; $t_e := T_p^C$; $t_e[p] := seq_p^C$ /* Create timestamp $t_e$ */
  PrCast($\langle e, t_e \rangle$)
  Insert $e$ into recovery buffer $B$
On process $p$ receives $\langle e, t_e \rangle$
  Insert $e$ into recovery buffer $B$
  if $e$ can be delivered then
    deliver($e$)
    for all $e' \in H_p \cup R$ that can be delivered
       deliver($e'$)
  else
    if $e$ is not delivered or obsolete then
      delay($e$, time_to_terminate)
On timeout($e$, time_to_terminate)
  for all $eid \in to\_deliver\_before\_ids(e)$ not in $H_p \cup R$ and $eid$ not already under recovery
     send($\langle RECOVER, \text{eid} \rangle$) to source($eid$) or to $k$ arbitrary processes in cluster
  delay($e$, time_to_recover)
On timeout($e$, time_to_recover)
  for all $e' \in to\_deliver\_before(e) \cap (H_p \cup R)$ that can be delivered
     deliver($e'$)
  deliver($e$)
  for all $e' \in H_p$ that can be delivered
     deliver($e'$)
On process $p$ receives $\langle RECOVER, \text{eid} \rangle$ from process $q$
  if $p$ has $e$ with identifier eid in its buffer then
    respond( $\langle ACKRECOVER, e, t_e \rangle$ ) to process $q$
On process $p$ receives $\langle ACKRECOVER, e, t_e \rangle$
  Insert $e$ into recovery buffer
  if $e$ can be delivered then
    deliver($e$)
    for all $e' \in R \cup H_p$ that can be delivered
       deliver($e'$)
  else
    if $e$ is not delivered or obsolete then
      $R := R \cup \{e\}$
On deliver($e$)
  $\forall i$ $T_p^C[i] := \max(t_e[i], T_p^C[i])$ /* Update $T_p^C$ */
  Remove $e$ from $R$ and $H_p$
  Deliver $e$ to the application
```

If a_i denotes the arrival time of an event e_i, the "server" processes each event at time $s_i = a_i + T$. Observe that if the length of the buffer in $\mathcal{C}$ is greater than the maximum length of the queue within the time interval $[a_i, s_i]$ then $\mathcal{C}$ can safely deliver e_i.

Consider $[t_a, t_s]$ denoting an interval of length T and the random variable $X_{i,j}$ denoting the event that at time $t_a + i$ process j inserts a new event in the system. Further, assume that all $X_{i,j}$ occur independently, and that $\mathbf{Pr}[X_{i,j} = 1] = p$ and $\mathbf{Pr}[X_{i,j} = 0] = 1 - p$. The number of admitted events in the system can be represented by the random variable $X := \sum_{j=1}^{n} \sum_{i=1}^{T} X_{i,j}$, hence the random process describing the arrival rate of new events is a binomial distribution and the expected number of events in the queue in an arbitrary time interval $[t_a, t_s]$ equals

$$\mathbf{E}[X] = npT.$$

Clearly, the length of the recovery buffer must be at least as large as $\mathbf{E}[X]$, or we are expected to encounter a large number of events that cannot be recovered.

Now, using the Chernoff bound [Kol03, MR95], we bound the buffer size so that the probability of an event that needs recovery not to be present in the recovery buffer of any arbitrary process becomes low.

Theorem 3.3.1 *Let e be an event admitted to a system $\mathcal{C}$ executing the two-layered protocol, where each event is required to stay in $\mathcal{C}$ for T rounds. Each of the n processes in the system admits a new event to $\mathcal{C}$ in a round with probability p. Then $\mathcal{C}$ can guarantee the availability of e in the recovery buffer of an arbitrary process with probability strictly greater than $1 - \left(\frac{e}{4}\right)^{npT}$ if the size of the buffer is chosen greater than or equal to $2npT$.*

Proof: Following the Chernoff bound for binomial distributions, for any $\delta > 0$ it is the case that $\mathbf{Pr}[X > (1+\delta)npT] < \left(\frac{e^{\delta}}{(1+\delta)^{\delta+1}}\right)^{npT}$. By choosing $\delta = 1$, the result follows. □

To estimate T, we can use the estimated duration of a PrCast, e.g. as in [Kol03]. Let PrCastTime denote this time. An event e is likely to be needed in $\mathcal{C}$ for (i) PrCastTime rounds (to be delivered to all processes with high probability); (ii) plus PrCastTime rounds, if missed, to be detected as missing by the reception of a causally related event (note that this is relevant under high load, since in low loads PrCast algorithms are even more reliable); (iii) plus the time $time\_to\_terminate + time\_to\_recover$ spent before and after requesting recovery.

Further, since processes may fail, a process that needs to recover some event(s) should contact a number of other processes to guarantee recovery with high probability. Assume that processes fail independently with probability p_f and let X_f be the random variable denoting the number of faulty processes in the system. Then

$$\mathbf{E}[X_f] = p_f n.$$

By applying the Chernoff bound as in Theorem 3.3.1 we get:

Lemma 3.3.2 *If, in a system of n processes where each one may fail independently with probability p_f, we consider an arbitrary process subset of size greater than or equal to $2np_f$, with probability strictly greater than $1 - \left(\frac{e}{4}\right)^{np_f}$ there will be at least one non-failed process in the subset.*

Proof: Following the Chernoff bound for binomial distributions, for any $\delta > 0$ it is the case that $\mathbf{Pr}[X > (1+\delta)np_f] < \left(\frac{e^\delta}{(1+\delta)^{\delta+1}}\right)^{np_f}$. By choosing $\delta = 1$, the result follows. □

This implies that if a process requests recovery from $R = 2p_f n$ processes then w.h.p. there will be at least one non-faulty to reply.

Theorem 3.3.2 *In a system of n processes where each one may fail independently with probability $p_f \leq k/(2n)$ for fixed k, an arbitrary process that needs to recover events according to the Two-Layer protocol, will get a reply with high probability if it requests recovery from k processes.*

Proof: From Lemma 3.3.2 above we know that if a process requests recovery from at least $2p_f n$ processes then w.h.p. there will be at least one non-faulty process among them, which can answer. So we choose $k \geq 2p_f n$, thereby for a given k high probability guarantee holds for $p_f \leq k/(2n)$. □

Note that requesting recovery only once and not propagating the recovery messages is good because in cases of high loss due to networking problems we do not flood the network with recovery messages. Compared to recovery by asking the originator of an event, this method may need k times more recovery messages. However, the advantages are tolerance of failures and process departures, as well as distributing the load of the recovery in the system.

Regarding replacement of events in the recovery buffer, the simplest option is FIFO replacement. Another option is an aging scheme, e.g. based on the number of hops the event has made. As shown in [Kol03], an aging scheme may improve performance from the reliability point of view. However, to employ such a scheme here we need to sacrifice the separation between the consistency layer and the underlying dissemination layer to access this information. Instead, note that using a dissemination algorithm such as the *Estimated-Time-To-Terminate-Balls-and-Bins*(ETTB)-gossip algorithm [Kol03] that uses an aging method to remove events from process buffers and guarantees very good message stability, implies that the reliability is improved since fewer processes may need to recover events.

3.4 Experimental evaluation

In this section we investigate the scalability of causal cluster consistency and the reliability and throughput effects of the optimistic causality layer in the Two-Layer protocol. We refer to a message/event as lost if it was not received or could not be delivered without violating optimistic causal order.

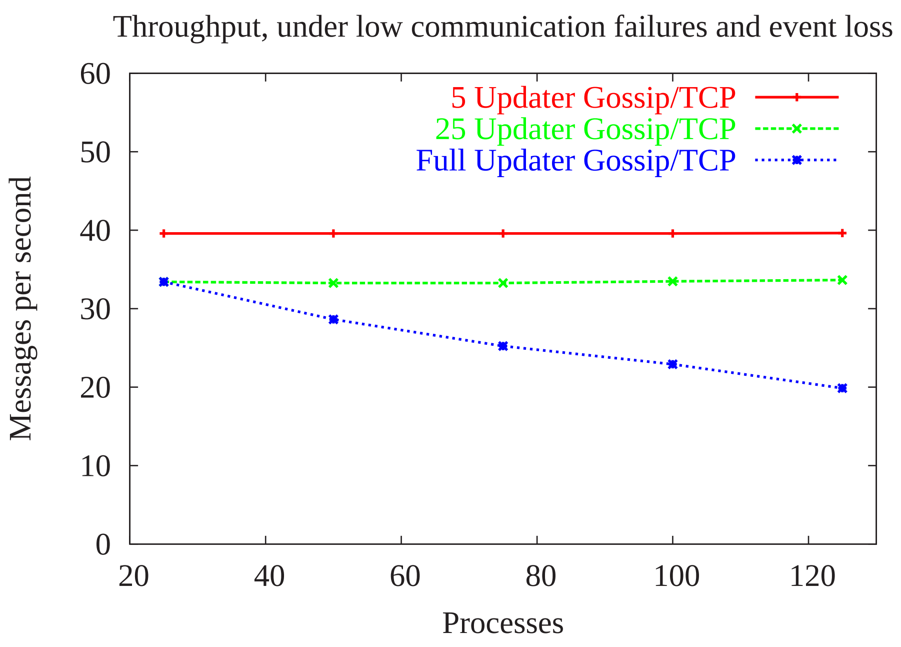


Figure 3.1: Event throughput with increasing number of cluster members.

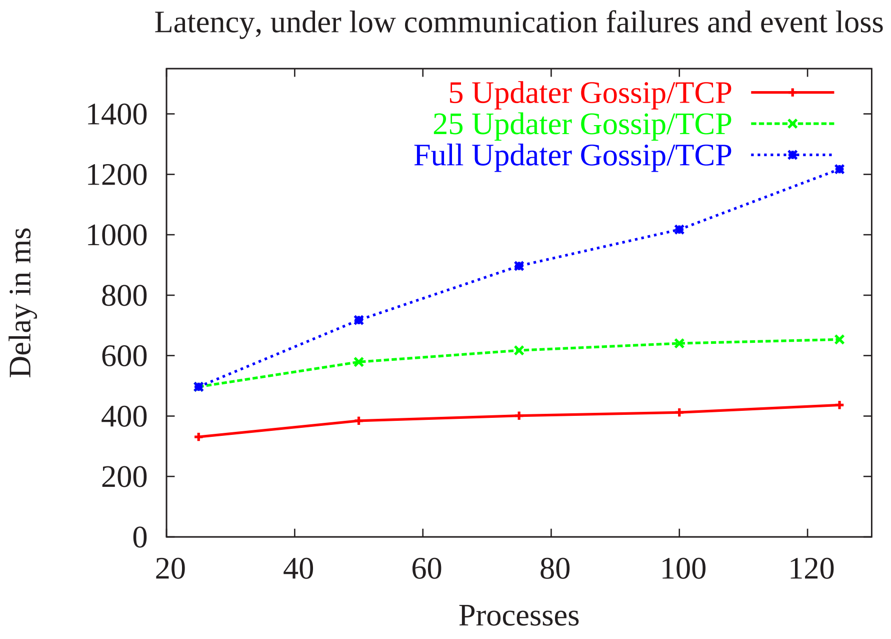


Figure 3.2: Event latency with increasing number of cluster members.

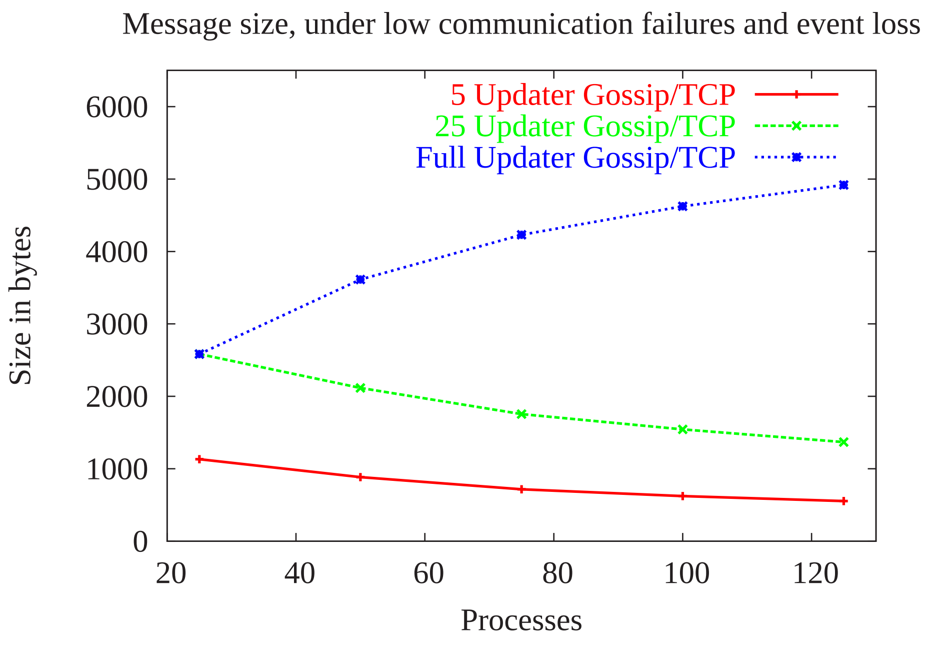


Figure 3.3: Message size with increasing number of cluster members.

System and implementation

The evaluation of the Two-Layer protocol was done on 125 networked computers at Chalmers University of Technology. The computers were Sun Ultra 10 and Blade workstations running Solaris 9 and PC's running Linux distributed over a few different subnetworks of the university network. The average round-trip-time for a 4KB IP-ping message was between 1ms and 5ms. As we did not have exclusive access to the computers and the network, other users might potentially have made intensive use of the network concurrently with the experiments.

The Two-Layer protocol is implemented in an object oriented, modular manner in C++. The implementation of the causality layer follows the description in Section 3.3.1 and can be used with several group communication objects within our framework. Our PrCast is the ETTB-dissemination algorithm described in [Kol03] together with the membership algorithm of lpbcast [EGH$^+$01]. TCP was used as message transport (UDP is also supported). Multi-threading allows a process to send its gossip messages in parallel and a timeout ensures that the communication round has approximately the same duration for all processes.

Scalability results

Our first set of experiments evaluate how the number of coordinators affects throughput, latency and message size. In our test application a process acts

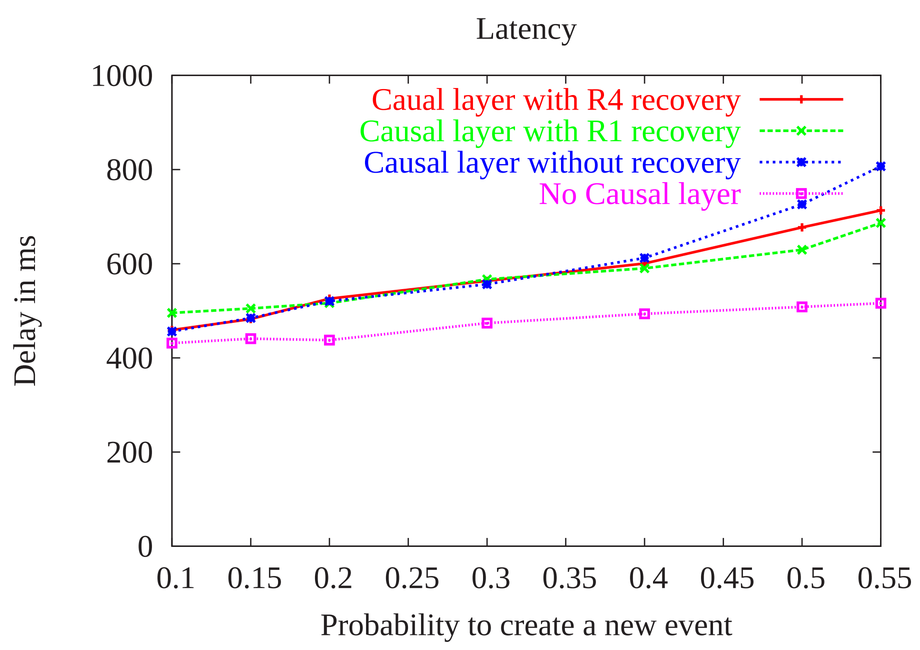


Figure 3.4: Event latency under varying load with and without the causality layer.

either as a coordinator, which produces a new event with probability p in each PrCast round, or as an ordinary cluster member. The product of the number of coordinators and p was kept constant (at 6).

To focus on the performance of the causality layer the PrCast was configured to satisfy the goal of each event reaching 250 processes w.h.p. (the fan-out was 4 and the event termination time was 5 hops). PrCast was allowed to know all members to avoid side effects of the membership scheme. The maximum number of events transported in each gossip message was 20. The size of the history buffer was 40 events, which according to [Kol03] is high enough to prevent w.h.p. PrCast from delivering the same event twice. The duration of each PrCast round was tuned so that all experiments had approximately the same rate of TCP connection failures (namely 0.2%).

The Figures 3.1, 3.2 and 3.3 compares the throughput, latency and message size of three instances of the Two-Layer protocol: the full-updater instance where all processes act as coordinators, the 5-updater and the 25-updater instances with 5 and 25 coordinators, respectively.

The causality layer used the first recovery method, described in Section 3.3.1. The results show the impact of the size of the vector clock on the overall message size and throughput. For the protocols using a constant number of coordinators message sizes even decreased slightly with growing group size since the dissemi-

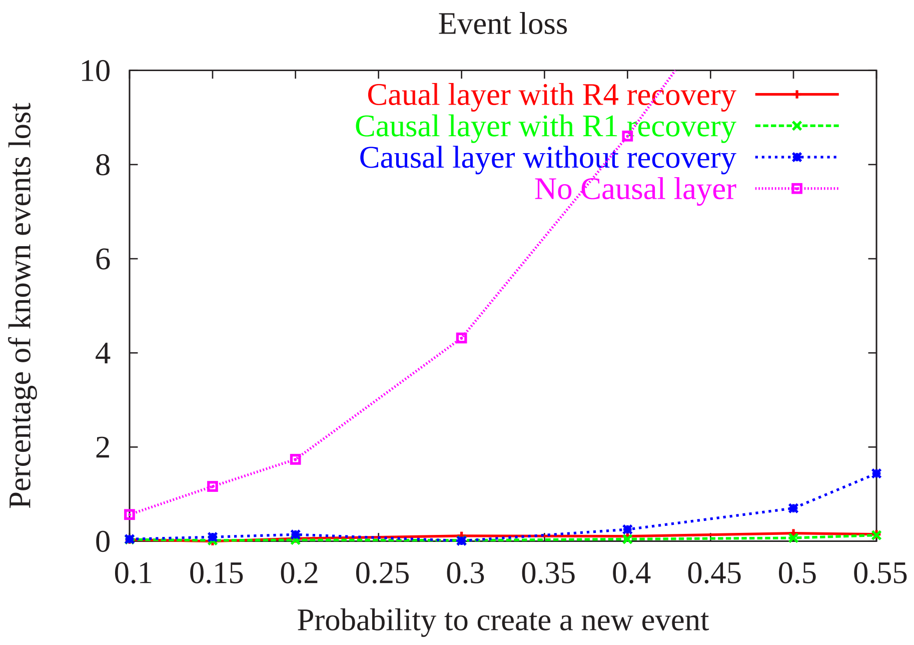


Figure 3.5: Event loss under varying load with and without the causality layer.

nation distributes the load of forwarding events better then, i.e. for large groups a smaller percentage of processes performs work on an event during the initial gossip rounds. However, for the full updater protocol messages grow larger with the number of coordinators which influences the observed latency and throughput. For growing group size the protocols with a fixed number of coordinators experience only a logarithmic increase in message delay and throughput remains constant while for the full-updater protocol latency increases linearly and throughput decreases.

Comparison of recovery schemes

Our second set of experiments study the effects of the causality layer and the recovery schemes in the Two-Layer protocol. The Figures 3.4 and 3.5 compares the gossip protocol and the Two-Layer protocol with and without recovery. The recovery is done in two ways, both described in Section 3.3.3: (i) from the originator (marked “R1 recovery”) and (ii) from k arbitrary processes (marked “R4 recovery” as the recovery fan-out k was 4). The recovery buffer size follows the analysis in Section 3.3.3, with the timeout-periods set to the number of rounds of the PrCast. Unlike the first experiment, the number of coordinators and processes was fixed to 25; instead varying values of p were used, to study the behaviour of the causality layer under varying load. Larger p values imply increased load in the system; at the right edge of the diagrams approx-

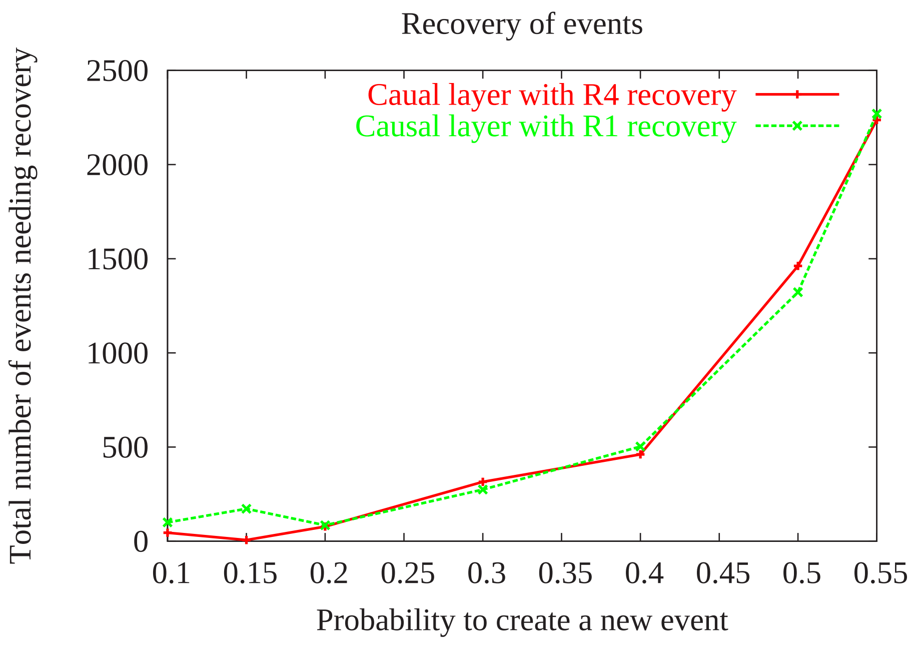


Figure 3.6: Total number of attempts to recover an event under varying load.

imately $n/2$ new events are multicast in each round. As the load increases, more events are reordered by the dissemination layer and message losses begin to occur due to buffer overflows, thus putting the causality layer protocols under stress. The results in Figure 3.5 show that the causality layer significantly reduces the amount of lost (ordered) events, in particular when the number of events disseminated in the system is high. With the recovery schemes almost all events could be delivered in optimistic causal order. With increasing load latency grows only slowly (cf. Figure 3.4), thus manifesting scalability. The causality layer adds a small overhead by delaying events in order to respect the causal order. The recovery schemes do not add much overhead with respect to latency, while they significantly reduce the number of lost events. At higher loads the recovery schemes even improve latency since by recovering missing events causally subsequent events in H_p can be delivered before they time out. Figure 3.6 shows the total number of attempts to recover missing events in the system and Figure 3.7 shows the success rate for the recovery attempts. The number of recovery attempts increase as the load in the system increases, when the load is low very few events need recovery (cf. the event loss without the causality layer in Figure 3.5). There are three likely causes for a recovery to fail: (i) the reply arrives too late; (ii) the process(es) asked did not have the event; and (iii) the reply or request(s) messages were lost. The unexpectedly low success rate during low load for the R4 method could be because a PrCast may

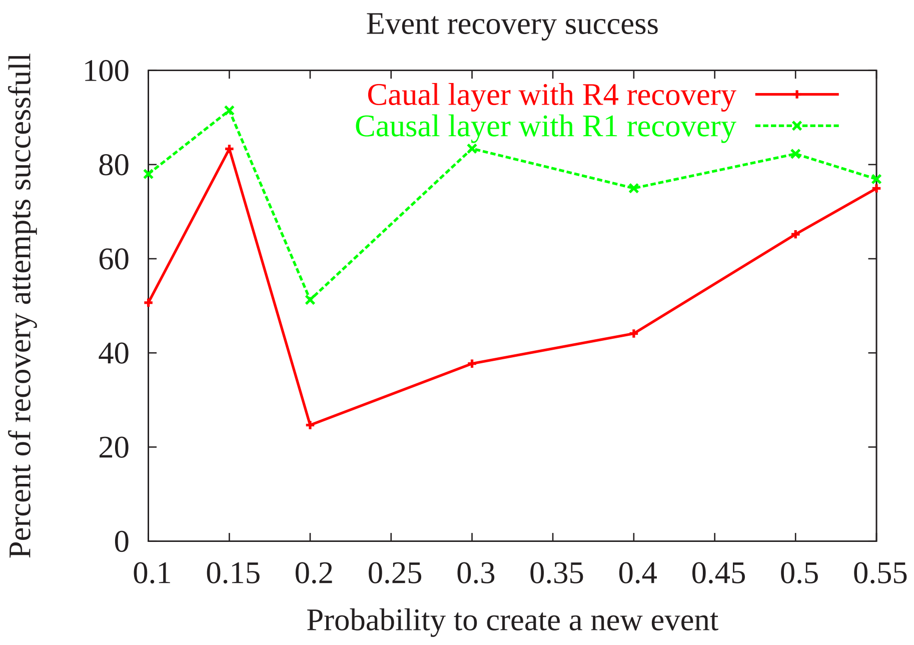


Figure 3.7: Percentage of successful recovery attempts under varying load.

reach very few processes when a gossip message is lost early in the propagation of an event. Also note that as the load is low the number of missing events and recovery attempts is very small. However, as load and the number of recovery attempts increase the success rate converges towards the predicted outcome.

3.5 Discussion and future work

We have proposed lightweight causal cluster consistency, a hierarchical layer-based structure for multi-peer collaborative applications. This is a general architecture, which can be applied on top of the standard Internet transport-layer services, and offers a layered set of services to the applications that need them.

We also presented a two-layer protocol for causal cluster consistency running on top of decentralized probabilistic protocols supporting group communication. Our experimental study, conducted by implementing and evaluating the proposed architecture as a two-layered protocol that uses standard Internet transport communication, shows that the approach scales well, imposes an even load on the system, and provides high-probability reliability guarantees.

Future work includes complementing this service architecture with other consistency models such as total order delivery with respect to objects. Object ownership and caching are other topics that are worth studying.

Chapter 4

Dynamic and Fault-Tolerant Cluster Management for Controlling Concurrency[1]

Anders Gidenstam   Boris Koldehofe

Marina Papatriantafilou   Philippas Tsigas

Abstract

Recent decentralized event-based systems have focused on providing event delivery which scales with increasing number of processes. While the main focus of research has been on ensuring that processes maintain only a small amount of information on membership and routing, an important factor in achieving scalability for event-based peer-to-peer dissemination system is the number of events disseminated at the same time. This work presents a dynamic and fault-tolerant cluster management method which can be used to coordinate concurrent access to resources in a peer-to-peer system. In the context of event-based dissemination systems the cluster management can be used to control the number of concurrently disseminated events. We present and analyze an algorithm implementing the proposed cluster management model in a fault-tolerant and decentralized way. The algorithm provides for each cluster a limited set of tickets. A process which has obtained a ticket may send events corresponding to the resources of the cluster. The algorithm guarantees that no two processes ever issue an event corresponding to the same ticket at the same time. The cluster management model on its own has interesting properties which can be useful for many peer-to-peer

[1]This is an extended version of the paper that appeared in the Proceedings of P2P 2005, Konstanz, Germany, August 31 - September 2, 2005.

applications.

Keywords: peer-to-peer communication, large scale group communication, middleware.

4.1 Introduction

Many applications like collaborative applications rely on an event-based dissemination service, for instance to exchange information on the state of shared replicated objects. For some applications there may be a large number of processes involved. Peer-to-peer dissemination algorithms for structured and unstructured networks have been studied to provide scalable event dissemination for a large number of processes. A lot of work has focused on providing delivery guarantees in the occurrence of dynamically joining and leaving processes by maintaining a low amount of resources locally at each process.

Current peer-to-peer dissemination systems rely on a good behaviour of each peer such that the overall number of events disseminated at the same time remains sufficiently small. A common assumption is that the rate of all incoming events remains constant. The reason is that there exists a limit on the amount information which can be stored locally, and also that the amount of information which one can send in a message per time unit is bounded by the physical constraints of computer networks. Since often the dissemination of an event is triggered by local decisions it is a difficult problem to control the amount of events which are disseminated at the same time. Once this rate exceeded the assumptions made by the dissemination system, the dissemination system cannot provide the original guarantees.

Here we address this problem by proposing a distributed cluster management. A cluster represents a region of interest in a peer-to-peer system, for example it may consist of a set of resources or objects which processes would like to access. To coordinate access to the resources, a cluster issues a finite set of enumerated tickets. Processes which received a ticket from the cluster receive the right to perform some action, for instance to disseminate an event corresponding to a resource, or to use a particular entry of a vector clock to issue causally ordered events [GKPT05b]. In order to prevent conflicts the cluster management needs to ensure in a decentralized fashion that, in spite of continuously joining and leaving as well as failing processes, never two processes perform an action corresponding to the same ticket at the same time. Moreover, one needs to ensure liveness by providing the possibility to reclaim tickets from processes that have crashed.

In this work we present an algorithm which can manage the cluster in the described way. Besides proving the correctness of the algorithm, we also present an analysis of availability of tickets depending on the failure rate and the amount of tickets maintained by non-faulty processes.

Structure of the paper. In Section 4.2 we describe the problem and introduce notation and definitions. Then we present two algorithms implementing a dynamic cluster management. The protocol of Section 4.3 works in the absence of failures and illustrates the basic idea, while Section 4.4 describes and proves a fault-tolerant membership protocol. In Section 4.5 we discuss related work on resource management in peer-to-peer applications and in the subsequent section we conclude with a discussion of the presented results and future work.

4.2 Notation and problem statement

Consider a peer-to-peer system supporting a large number processes (each process is considered to be a peer) to join and leave the system dynamically. The processes are said to form a group denoted by $G = \{p_1, p_2, \ldots\}$. Processes in G maintain a set of resources $R = \{r_1, \ldots r_l\}$. We assume the set of resources is partitioned into several disjoint clusters $C_1, C_2, \ldots$ with $\cup_i C_i \subseteq R$. Processes which are interested in certain resources need to join the respective cluster and will be informed afterwards about events corresponding to all resources maintained inside the cluster. A process which wishes to create events corresponding to a resource inside a cluster need to obtain a ticket of the cluster. For a cluster C there exists a maximum of n tickets where n is known to the processes which joined C.

Processes which own a ticket are called *coordinators* of C. Let $Core_C$ denote the set of coordinators of C. The set of coordinators can change dynamically over time. Throughout the paper we will use the term *events* when referring to messages which were sent with respect to a ticket of the cluster.

An algorithm implementing the dynamic cluster management needs to implement the following operations:

- *Ordinary joining/leaving a cluster.*
- *Coordinator joining/leaving the core of a cluster.*

Any ordinary process in G can perform a join or leave operation on C corresponding to the ordinary join and leave operation of the underlying multicast primitive. With respect to cluster management we will also call these operation *join* and *leave.* An ordinarily joined process will be able to observe events related to resources of a cluster.

In order to become a coordinator in a cluster C, i.e. to become member of $Core_C$ and be able to send events, a process performs an operation called *cjoin.* If process p performs a *cjoin* operation, p becomes assigned to be the owner of a unique ticket of C. When p performs a *cleave* operation it will release its ticket and cannot send events related to resources of the cluster after that. The tickets released by p may then be reused by any other process performing a *cjoin* operation.

For correct cluster management it is essential that there are never two or more coordinators that own the same tickets within the cluster at the same time.

The ticket of a process that performed a *cleave* or has failed should eventually be reusable for other processes. Moreover, the cluster management should perform well even if a large number of processes concurrently perform *cjoin* operations.

Using a single process for cluster management is the simplest solution. However, if the cluster manager fails, then no processes can perform *cjoin* or *cleave*. Finding a new coordinator reduces to the agreement problem.

The propagation of events is done by multicast communication. It is not assumed that all processes of a cluster will receive an event which was multicast, nor does the multicast need to provide any ordering by itself. Any lightweight probabilistic group communication protocol as appears in the literature [EGH$^+$01, GKM01, Kol03] would be suitable. We refer to such protocols as *PrCast*. PrCast is assumed to provide (i) that an event is delivered to all destinations with high probability, (ii) decentralized and lightweight group membership, i.e. a process can join and leave a multicast group in a decentralized way and processes do not need to know all members of the group.

4.3 Dynamic cluster management

In the following we present a method that allows interleaved *cjoin* and *cleave* operations. The main idea of our approach is to make every process in the core of the cluster the coordinator of a subset of the tickets $\{0 \ldots n-1\}$. We will ensure that there are never two processes that simultaneously own or coordinate the same ticket. In order to illustrate the basic idea we assume in this Section that communication is reliable and processes do not fail. In Section 4.4 we show how to extend the presented ideas under a realistic failure model.

We assume that tickets form a cyclic relation according to their number, i.e. the succeeding ticket to ticket i is ticket $i-1 \mod n$, while the preceding ticket to ticket i is ticket $i+1 \mod n$. Each process which becomes coordinator of the cluster will own one ticket. Let i be the ticket owned by process p, also denoted as ticket$(p) = i$. The successor of p is the closest process which can be reached by following the chain of succeeding tickets to i. Accordingly, the predecessor of p is the closest process which can be reached by following the chain of preceding tickets. Moreover, we denote q the dth closest successor (predecessor) of p, if the process q is reachable in d steps from p by following the chain of successors (predecessors) starting at p.

In order to manage free tickets, the processes which own tickets also become coordinator of a subset of all tickets maintained in a cluster. We define the set of tickets which is coordinated by a process in terms of successor and predecessor. Let p and q denote two processes owning tickets i and j respectively and let q be the successor of p. Process p coordinates its own ticket i and all tickets succeeding its own ticket and preceding ticket j. Let the coordinated set, S_p, denote the set of tickets coordinated by p. Formally, we write

$$\begin{aligned} S_p = \{\, l \quad | \quad & l = i - k \mod n,\ 0 \le k < \\ & \min\{m \mid j = i - m \mod n,\ m > 0\}\}. \end{aligned}$$

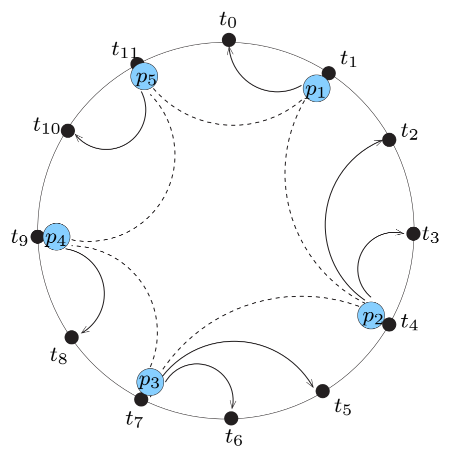


Figure 4.1: Illustration on how processes maintain and coordinate tickets of a cluster. An arrow from process p_i to a ticket indicates that p_i is the respective coordinator.

Figure 4.1 gives an example of how processes maintain and coordinate tickets, e.g. p_2 owns ticket 4 and coordinates the tickets $\{2, 3\}$.

Lemma 4.3.1 *Let C denote a cluster with* $\text{Core}_C \neq \emptyset$ *and no two processes own the same tickets. Then,*

1. *for $p, q \in \text{Core}_C$ and $p \neq q \Rightarrow S_p \cap S_q = \emptyset$,*
2. $\cup_{p \in \text{Core}_C} S_p = \{0, \ldots, n-1\}$.

Proof: The lemma follows immediately from the definition of the coordinated set of a process. □

Algorithm 4.1, continued in Algorithm 4.2, presents a decentralized solution which can coordinate the tickets of a cluster if no failures occur. The algorithm ensures that no two processes coordinate the same tickets at the same time; the key to achieve this is by preserving the successor/predecessor relation between coordinators. A process p which wishes to become coordinator in the cluster selects an arbitrary coordinator. To enforce a good load balance of requests to coordinators the selection by p could take the coordinator of a ticket chosen uniformly at random from the set of available tickets (this can be known by contacting any coordinator in the cluster). Let q be the selected coordinator then p sends a *cjoin* message to q. Before responding to p's request, q will first serve all previous *cjoin* and *cleave* operations it received earlier by other processes. In this way interleaving *cjoin* and *cleave* requests with respect to the same coordinator become serialized. If q has decided to perform a *cleave* operation or does not have any available tickets it will reply negatively to p.

Algorithm 4.1 Cluster management in the absence of failures, part I.

```
VAR
  Cview_p:       vector of processes
  ImmedSucc_p:   immediate successor of process p
  ImmedPred_p:   immediate predecessor of process p
  state_p:       state variable
  ticket_p:      the ticket owned by process p
Message types:
  CJOIN, CLEAVE, ACKJOIN, ACKSUCC, ACKCLEAVE, REJECT
Init_p:
  $ticket_p := \perp$
  $state_p :=$ joining
  Send $\langle$CJOIN, $p\rangle$ to a known coordinator in $Core_C$.
Initialization of variables when cjoin accepted
On process p receives $\langle ACKCJOIN, i, j, Cview\rangle$ from process q
  $Cview_p :=$ Cview
  $p$ becomes the coordinator for all tickets from $i$ down to $j+1$
  $ticket_p := i$
  $ImmedSucc_p :=$ Cview[$j$]
  $ImmedPred_p := q$
  Send $\langle$NEWSUCC$\rangle$ to Cview[$j$]
Successor acknowledged
On process p receives $\langle ACKSUCC\rangle$ from process q
  $state_p :=$ coordinator
  $ImmedSucc_p := q$
Receiving a cjoin request
On process p being coordinator of tickets i down to $j+1$ receives $\langle CJOIN\rangle$ from process q
  if $state_p \neq$ coordinator then
    Send $\langle$REJECT$\rangle$ to q
  else
    Process all previously received CJOIN and CLEAVE requests
    if $|S_p| > 1$ then
      Select ticket $t \in S_p \setminus \{i\}$.
      Cview[$t$] := $q$
      $ImmedSucc_p := q$
      Send $\langle$ACKCJOIN, $t, j$, Cview$\rangle$ to $q$
    else
      Send $\langle$REJECT$\rangle$ to $q$
A new predecessor
On process p being coordinator of tickets i down to $j+1$ receives $\langle NEWSUCC\rangle$ from q
  if $state_p =$ leaving then
    Send $\langle$CLEAVE, Cview[$j$]$\rangle$ to $q$
  else
    Send $\langle$ACKSUCC$\rangle$ to $q$
    $ImmedPred_p := q$
```

Algorithm 4.2 Cluster management in the absence of failures, part II.

```
Leaving the cluster
On process p being coordinator of tickets i down to $j+1$ decides to leave the cluster
  $\text{state}_p$ := leaving
  Serve all previously received cjoin and cleave requests
  Send $\langle \text{CLEAVE}, \text{Cview}[j] \rangle$ to $\text{ImmedPred}_p$
Receiving a cleave request
On process p being coordinator of tickets i down to $j+1$ receives $\langle CLEAVE, r \rangle$ from q
  if $q = \text{ImmedSucc}_p$ and p is not serving any cjoin and $\text{state}_p \neq$ leaving then
    Send $\langle \text{ACKCLEAVE}, q \rangle$
    Send $\langle \text{NEWSUCC}, r \rangle$
    $\text{ImmedSucc}_p := r$
Receiving a cleave acknowledgment
On process p being coordinator of tickets i down to $j+1$ receives $\langle ACKLEAVE, p \rangle$ from q
  if $\text{state}_p$ = leaving then
    $\text{state}_p$ := not_a_coordinator
    $\text{ticket}_p := \perp$
```

If q is ready to serve the *cjoin* request by p, it will assign a ticket $t \in S_q$ to p (possibly reflecting the random choice when determining q as a suitable coordinator). Let r be q's successor. Process q will send a message *ACKCJOIN* to p with information about t and r to p and will select p as its new successor.

When p receives the message *ACKCJOIN*, p will select q as its predecessor and r as its new successor. In order to allow process r to leave the cluster and maintain its predecessor information correctly, p must, before being able to perform as a coordinator, send a message *NEWSUCC* to process r. If r is not intending to leave the cluster, it will reply by sending an acknowledgement *ACKSUCC* to p and update its predecessor to be p. Process p can then perform as a coordinator of the cluster.

In the case a process r intends to leave the cluster it first processes all previously received cjoin and cleave requests and sends afterwards a *CLEAVE* message including information of the successor of r, say s, to its predecessor, say q. If r receives afterwards from another process p a message *NEWSUCC* it will again sent a message *CLEAVE* to p. Process r only leaves the cluster after it has received a message *ACKCLEAVE*.

A process p serves a *cleave* message by r only if r is the current successor of p. In this case p will sent a message *ACKCLEAVE* to r. Thereafter p sets s as its new successor and sends a message *NEWSUCC* to s. Note that p may have to subsequently serve *CLEAVE* messages from its new successor until finally receiving a message *ACKSUCC* from a successor. However, after each *ACKCLEAVE* a process coordinates a larger amount of tickets and hence the number of subsequent *NEWSUCC* messages before a process can perform as a coordinator is bounded.

Once a process may perform as a coordinator it also PrCasts that it became a coordinator in $Core_C$ and that it owns ticket t. Note that the PrCast operation is only of relevance to inform other processes about p being a coordinator, but it is not necessary to prevent any pair of distinct processes from maintaining the same ticket.

In order to verify correctness of the protocol as stated in Theorem 4.3.1, recall that according to Lemma 4.3.1 correctly preserving the relation among successors and predecessors, suffices to guarantee unique assignment of processes to tickets. This is shown in Lemma 4.3.2.

Lemma 4.3.2 *Let q be a coordinator in* Core_C *with successor r, serving a* cjoin *operation of p. Then*

1. *any interleaving* cjoin *operation will take effect earliest after processes p and q successfully updated their successor and predecessor,*

2. *an interleaving* cleave *operation of r will successfully be managed at p and therefore preserve the predecessor successor relation of* Core_C *correctly.*

Theorem 4.3.1 *Let $\Sigma := \sigma_1, \ldots, \sigma_m$ denote a sequence of potentially interleaved operations on a cluster C where σ_i corresponds to a* cleave *or* cjoin *operation. If Σ maintains* Core_C *to include at least one process the algorithm guarantees for any $p, q \in \mathrm{Core}_C$*

1. *unless $p = q$, $S_p \cap S_q = \emptyset$;*

2. *unless $p = q$, p and q maintain different tickets.*

4.4 Supporting link and process failures

In the following we present an algorithm which extends the previous framework of Section 4.3 to deal with link and process failures. It is assumed that processes fail by stopping, we do not consider Byzantine faults. Links may be slow or failing. Communication between pairs of processes is connection oriented. Let δ denote the maximum tolerated message delay and let p and q denote processes. Connection oriented communication guarantees: if p sends a message, M, to q, p expects to receive a status about M not later than time δ. If status of M is *OK* then q has received M not later than time δ. Otherwise p has no knowledge whether q received the message or not; we say then that p *weakly detects q as faulty.* Since the algorithm works in rounds, we also assume that processes have clocks which maintain approximately the same speed. Let T denote a time period larger than the maximum tolerated message delay. If m processes periodically with period T send messages to p , then p will receive $m - \epsilon < m' < m + \epsilon$ messages during any time interval of length T which starts after p has received the messages sent in the previous period by the m sources, when none of the m processes failed.

The algorithm performs in rounds, where the time between two consecutive rounds is assumed to be long enough to host a PrCast, i.e. to inform members of the cluster C about a successful *cjoin* operation (if any has happened). The fault-tolerance of the algorithm is controlled by the parameter k. In a round

Algorithm 4.3 Decentralized and fault-tolerant cluster management, part I.

```
VAR
  $L_p$:            set consisting of $2k+1$ predecessors $p$ received from its immediate prede-
  cessor
  $R_p$:            set consisting of $p$ and $2k$ predecessors successfully sent to its immediate
  successor
  ALIVE$_p$:        set of processes which $p$ received an *ALIVE* message from during a round
  Cview$_p$:        vector of processes
  ImmedSucc$_p$:    immediate successor of $p$
  ImmedPred$_p$:    immediate predecessor of $p$
  TempRounds$_p$:   indicates the number of rounds for which a process is not sending *UPDATE*
  messages
  state$_p$:        state variable
  ticket$_p$:       the ticket owned by process $p$
  $P_{\text{exclude}}$:  probability to start exclusion algorithm after weakly detecting a faulty
  successor
Message types:
  CJOIN, ALIVE, UPDATE, ACKJOIN, EXCLUDE, REQCOORD, ACKEXCLUDE
Init$_p$:
  ticket$_p$ := $\perp$
  state$_p$ := joining
  Send $\langle$CJOIN, $p\rangle$ to a known coordinator in $Core_C$.
Main loop of the coordinator algorithm
Do in every round (duration longer than PrCast) while ticket$_p$ = coordinator
  if $|\text{ALIVE}_p \cap L_p| < k+1$ then
    state$_p$ := disconnected
    exit loop
  Send $\langle$ALIVE, $p\rangle$ to $2k+1$ closest successors in *Cview*.
  if TempRounds$_p = 0$ then
    $R := \{r \in L_p \mid r$ is among the $2k$ closest predecessors of $p\} \cup p$
    STATUS := Send $\langle$UPDATE, $R\rangle$ to ImmedSucc$_p$
    if STATUS is OK then
      $R_p := R$
    else
      Run exclusion algorithm with probability $P_{\text{exclude}}$
  else
    TempRounds$_p$ := TempRounds$_p - 1$
```

Algorithm 4.4 Decentralized and fault-tolerant cluster management, part II.

```
Initialisation of variables when cjoin succeeds
On process $p$ receives $\langle ACKCJOIN, L, i, j, Cview\rangle$ from $q$
  $\text{Cview}_p := \text{Cview}$
  $L_p := L$
  $R_p := \emptyset$
  process $p$ becomes the coordinator for all tickets $i$ down to $j+1$
  $\text{ticket}_p := i$
  $\text{ImmedSucc}_p := \text{Cview}[j]$
  $\text{ImmedPred}_p := q$
  $\text{TempRounds}_p := 0$
  Send $\langle \text{ALIVE}, p\rangle$ to $2k+1$ closest successors in $\text{Cview}_p$.
Handling of UPDATE messages
On process $p$ receiving $\langle \text{UPDATE}, R\rangle$
  $L_p := R$
Receiving a cjoin request
On process $p$ being coordinator of tickets $i$ down to $j+1$ receives $\langle CJOIN\rangle$ from $q$
  if $(|S_p| > 1) \wedge (\text{TempRounds}_p = 0)$ then
    Select ticket $t \in S_p$.
    $\text{Cview}[t] := q$
    $\text{ImmedSucc}_p := q$
    $R := \{r \in L_p \mid r \text{ is among the } 2k \text{ closest predecessors of } p\} \cup p$
    STATUS := Send $\langle \text{ACKCJOIN}, R, t, j, \text{Cview}\rangle$
    if STATUS is OK then
      $R_p := R$
    else
      Run exclusion algorithm with probability $P_{\text{exclude}}$
  else
    Send $\langle \text{REJECT}\rangle$ to $q$
```

Algorithm 4.5 Decentralized and fault-tolerant cluster management, the exclusion algorithm.

```
Do
  STATUS := FALSE
  while ($p \neq$ succ(ImmedSucc)) $\wedge$ (STATUS is FALSE) do
    ImmedSucc := succ(ImmedSucc) {Finds the next possible successor from Cview}
    STATUS := Send$\langle$EXCLUDE, $p\rangle$ to ImmedSucc
  if (STATUS is True) $\wedge$ ($p$ receives $\langle$ACKEXCLUDE, $L_q\rangle$ from $q$) then
    $E_{pq}$ := all tickets succeeding ticket($p$) and preceding $ticket(q)$
    Send$\langle$REQCOORD, $E_{pq}\rangle$ to all processes in $L_q \cap R_p$
    Wait for time $2\delta$ for replies of type $ACKCOORD$
    if p receives $\geq k+1$ replies of type $ACKCOORD$ then
      {Do not send $UPDATE$ messages while some excluded processes may still be alive}
      TempRounds$_p$ := $dist(p,q) - 1$
    else
      state$_p$ := disconnected
      exit loop
  else
    state$_p$ := disconnected
    exit loop

On $q$ receives $\langle$EXCLUDE, $p\rangle$
  Reply$\langle$ACKEXCLUDE, $L_q\rangle$

On $r$ receives $<$ REQCOORD, $E_{pq}$ $>$
  if $r \notin E_{pq}$ then
    Send $\langle$ACKCOORD$\rangle$ to $p$
    Remove processes in $E_{pq}$ from $Cview$
```

of the algorithm, a process can tolerate in its $2k+1$ neighbourhood up to k process or communication failures. The algorithm is described in pseudocode (cf. Algorithm 4.3, Algorithm 4.4, and Algorithm 4.5), and below we present the ideas informally. During a round the algorithm maintains the following two invariants:

1. Any non-faulty process p in $Core_C$ which does not perform a *cleave* operation remains in $Core_C$ as long as p knows that at least $k+1$ of its $2k+1$ closest predecessors have not experienced any process or link failures.

2. Failed processes will eventually be excluded from $Core_C$ and processes which perform *cjoin* subsequently may reuse the respective tickets.

The first invariant is achieved by the processes in $Core_C$ sending *ALIVE* messages to their $2k+1$ closest successors in each round. A process that receives less than $k+1$ *ALIVE* messages during a round thinks that it is considered as failed and immediately stops being a coordinator and leaves $Core_C$ (cf. Algorithm 4.3 and Figure 4.2(I)-(II)).

In order to manage the exclusion scheme (cf. Algorithm 4.4, 4.5 and Figure 4.2(III)), a process p maintains two sets denoted by L_p and R_p. The set L_p is used to store p's "knowledge" of its $2k+1$ predecessors (this information is received from its immediate predecessor), while R_p contains the information in p's last successful *UPDATE* message to p's immediate successor containing the $2k$ closest predecessors of p and p itself. Both sets are needed to determine whether a range of coordinators can be excluded. When p joins $Core_C$, L_p is initialized by the coordinator performing the *cjoin* operation for p. The set R_p is initially empty. Each process also maintains an array denoted by $Cview_p$ which is p's local view on the set of coordinators in $Core_C$, i.e. if $Cview_p[i] = q$ holds, then p assumes q to be the coordinator owning ticket i .

In each round p proceeds if it has received, during the last round, at least $k+1$ *ALIVE* messages from processes in L_p, otherwise p thinks that it is considered as failed (cf. below for this case). If p also successfully received an *UPDATE* message from its direct predecessor proposing a new set L'_p, which includes $2k+1$ predecessors of p, then p sets $L_p = L'_p$.

If p may proceed, it creates $2k+1$ *ALIVE* messages and sends them to the $2k+1$ closest successors known from $Cview_p$. Moreover, it sends an *UPDATE* message to its direct successor containing a set denoted R'_p. The set R'_p contains the $2k$ closest predecessors in L_p and p itself. If p's transmission of the message *UPDATE*(R'_p) to its direct successor is successful, then p will set $R_p = R'_p$.

Assume a process weakly detects its successor r to be faulty, for instance because it could not establish a connection to r for some time. In order to release the tickets owned and coordinated by r, which is potentially faulty, p will try to contact the next closest successor in $Cview_p$ that is reachable, i.e. not detected as weakly faulty. Let q be the next closest successor reachable by p then q will reply by sending L_q. Process p will request from all processes in $R_p \cap L_q$ to be the new coordinator of all tickets succeeding p and preceding q denoted by E_{pq}. Only if p receives $k+1$ acknowledgement messages from

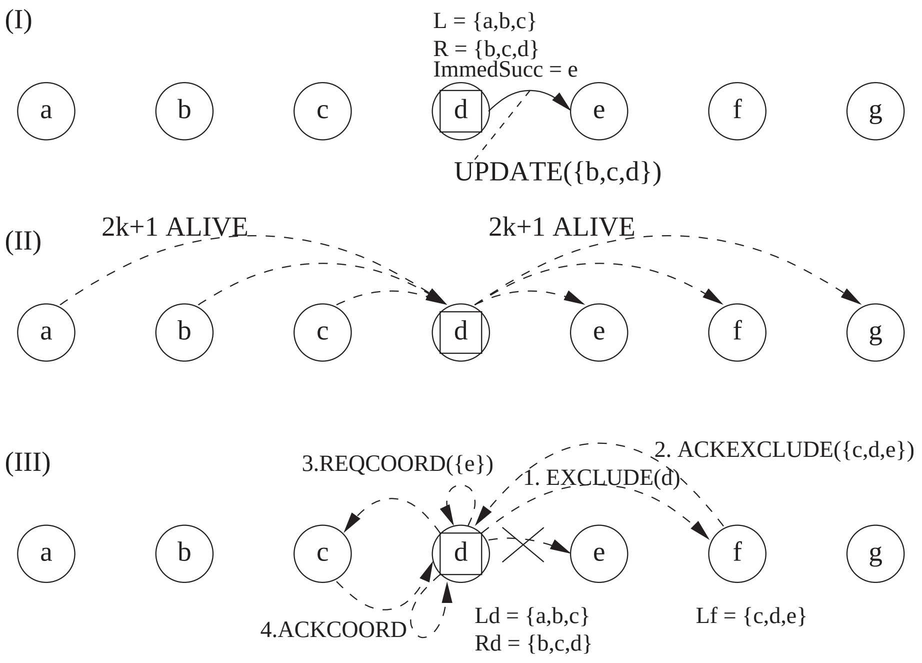


Figure 4.2: Example of the fault-tolerant cluster management algorithm with $k = 1$ focusing on the process d. (I) and (II) UPDATE messages and ALIVE messages under normal operation. (III) Message exchanges during an exclusion of d's immediate successor e.

destinations in $R_p \cap L_q$, p becomes the *temporary coordinator* of the tickets, otherwise p thinks it is considered as failed.

While being temporary coordinator, p behaves like an ordinary coordinator, however it does not attempt to change L_q by sending an *UPDATE* message and it does not serve *cjoin* requests. All processes in E_{pq} which neither have failed nor think they are considered to have failed are said to be *alive.* Once, there does not exist any alive processes in E_{pq}, p behaves like an ordinary coordinator again. Note that the time for a process remaining a temporary coordinator is bounded by at most the distance between p's and q's tickets since in every round the closest alive process in E_{pq} is guaranteed to think it is considered to have failed at the end of the round.

Processes which are requested to acknowledge an exclusion interval E_{pq} only acknowledge if their ticket is not contained in E_{pq}. Processes which acknowledged the exclusion of a process will remove processes in E_{pq} from *Cview* and prevent any updates of tickets corresponding to E_{pq} for $\text{dist}(p, q)$ rounds.

4.4.1 Correctness

In order to prove correctness of the membership algorithm of Section 4.4, we need to show that even in the occurrence of failures (i) two processes will never create conflicting events and (ii) the algorithm invariants are maintained.

In Lemma 4.4.1 we first consider the behaviour of the algorithm when no failures occur.

Lemma 4.4.1 *Let neither process failures, link failures, or slow links occur and processes always receive sufficiently many* ALIVE *messages. For any sequence of interleaving* cjoin *operations the membership scheme is equivalent to the membership protocol of Section 4.3.*

Proof: Both algorithms show only different behaviour if p executing Algorithm 4.3 weakly detects its immediate successor r to be faulty. Since neither processes, nor links do fail p must have detected r as faulty because r thinks it has been considered to be failed. This implies that r did not receive sufficiently many *ALIVE* messages or decided to leave the cluster, which is a contradiction to our assumption. $\square$

The critical case to analyze is after process p initiated the exclusion of E_{pq}. Lemma 4.4.2 states that during a round the closest successor in E_{pq} will fail.

Lemma 4.4.2 *Let E_{pq} denote the set of processes to be excluded where p coordinates the exclusion and q is the new successor of p. Further, let A denote the set of processes which received sufficiently many* ALIVE *messages in the current round. Let r denote the closest process in E_{pq} which is still alive. Then*

$$A \cap (L_r - E_{pq} - R_p) = \emptyset.$$

Proof: We can associate the passing of an *UPDATE* message with a token. We say process q received a token from p if there is a chain of consecutive *UPDATE* messages originating in p and ending in q. We define a relation $\prec$ where $p \prec q$ if q has received a token from p when it was created (i.e. the time it performed the *cjoin* operation), while $p \not\prec q$ if q did not receive a token from p at the time it was created.

Consider case $p \prec r$: In this case $L_r - E_{pq} - R_p$ is either empty or it contains destinations which where in a previous *Cview* of p. However, when p successfully updated R_p, the respective destinations were guaranteed to be excluded by the predecessors of p. Hence, this case yields $A \cap (L_r - E_{pq} - R_p) = \emptyset$

Let $p \not\prec r$: Any token originated by p and received by q must have been received by r. In particular if *Cview* of q was influenced by p, also r must have received influence by p. Then we can reason the same as before.

The difficult case remains where q did not receive any influence from p. We define for two processes p' and q', p' to be the parent of q' if p' coordinated q' to enter the cluster. Further, we define ancestor by the transitive closure of the parent relation. If q did not receive any token from p, but share a common influence, then q must have received a token from an ancestor of p. Let s denote the ancestor of p which succeeded last in sending a token to q.

Case r received the respective token: If r received the respective token, then it shares the same influence as q. Every consecutive token which origins from set E_{pq}, has no impact on $A \cap (L_r - E_{pq} - R_p)$. However, every token originating outside E_{pq} by transitivity will affect L_p once p has joined the cluster. Hence, no vertices in $L_r - E_{pq} - R_p$ are alive after p determined its set R_p.

Case r did not receive the respective token: There must be an ancestor which received the respective token. If there was not we would conclude $E_{pq} = \emptyset$. Then again p on its creation would share all influence by s on the ancestor of r and by transitivity to r itself. Hence, again all tokens which did not influence p originate from the set E_{pq}. Therefore no processes in $L_r - E_{pq} - R_p$ are alive, once p has updated R_p. □

Lemma 4.4.2 immediately implies Corollary 4.4.1 which states how long a process p needs to be temporary coordinator until at least i alive processes in E_{pq} have failed.

Corollary 4.4.1 *The ith successor of p in E_{pq} will fail latest i rounds after p was acknowledged.*

Proof: The immediate successor r of p clearly fails because all *ALIVE* messages r can expect according to Lemma 4.4.2 are from processes inside R_p (suppose p maintains a copy of send L_p) and at most k processes did not acknowledge p. Assume now that until round $i-1$ the closest $i-1$ successors have failed. Then in round i the only candidates for sending *ALIVE* messages are in L_i. However, there are at most k candidates which did not acknowledge the exclusion of the ith successor. □

Theorem 4.4.1 *Algorithm 4.3 guarantees that two processes never have common tickets they either own or coordinate.*

Proof: Lemma 4.4.1 shows that only exclusion could cause any such conflicts. Assume that during an execution two alive processes r and s, are two processes coordinating common tickets. This implies that one process, say r, failed to be excluded, while s was inserted. Let p be the process which failed to exclude r and inserted s.

After p initiated the exclusion of E_{pq} with $r, s \in E_{pq}$, p switches state to become temporary coordinator for $\text{dist}(p, q)$ rounds. During this time p could not have inserted s. However, when p switches state to become active coordinator and inserts s, Corollary 4.4.1 guarantees that r by that time thinks it is considered to have failed, contradicting that both r, and s were active. □

4.4.2 Performance and liveness properties

Message overhead. Note that the duration of a round is assumed to be longer than the time of a PrCast. PrCast is used to inform all processes which joined a cluster about an event regarding the resources of the cluster. The overhead which is induced by the membership protocol corresponds to the number of sent *ALIVE* messages. In each round a process sends and receives at most $2k + 1$ messages. Hence, the cluster management protocol can be considered as lightweight, i.e. it only adds a low number of additional messages while performing in combination with an application using the cluster management protocol. In addition every successful ticket acquisition is followed by a PrCast which involves all processes which joined the cluster.

Availability. An interesting performance measure is how well the algorithm manages to grant new processes access to tickets in the occurrence of failures and dependent on the amount of tickets maintained by non-faulty processes. Let α denote the fraction of tickets taken by non-faulty processes. Moreover, let p_f denote the probability for a process to fail in a round. Whenever a process q fails, the predecessor, say p, is trying to reclaim the tickets maintained by q. While running the exclusion algorithm p performs as a temporary coordinator and does not release any tickets.

Observe that $Core_C$ consists of the processes which have not been excluded and processes which perform correctly, i.e. we know $|Core_C| \geq \alpha n$. Since there exists at most n tickets the expected number of tickets maintained by each coordinator of $Core_C$ is smaller or equal to $1/\alpha$. Hence, the time to reclaim tickets from a failing process is expected to take time less or equal to $1/\alpha$.

Assume that (i) α remains constant, and (ii) the exclusion algorithm needs $1/\alpha$ rounds. Then the expected number of failing processes which needs to be excluded is $p_f n$ because in each round $\alpha p_f n$ processes are expected to fail. By applying the Chernoff bound [MR95], one can bound the probability that in a round of the algorithm's execution there exist more than $2p_f n$ faulty processes. The probability is strictly smaller than $(e/4)^{2p_f n}$. That means a process which attempts to acquire a ticket succeeds w.h.p. if $p_f < \frac{1}{2}(1 - \alpha)$.

4.5 Related Work

Many distributed applications like collaborative environments (e.g. [MT95, GB97, CH93]) use event-based dissemination to interact on a distributed shared state. In order to perform well for many processes, such systems rely on a middleware which provides scalable group communication, supports maintenance of membership information according to processes interest as well as fast dissemination of events in the system.

Recent approaches for information dissemination use lightweight probabilistic group communication protocols [BHO$^+$99, EGH$^+$01, GKM01, Kol03, PRMK03, BEG04]. These protocols allow groups to scale to many processes by providing reliability expressed with high probability. In [PRMK03] it is shown that probabilistic group communication protocols can perform well also in the context of collaborative environments. However, to guarantee a delivery with high probability one needs a control mechanism for the number of concurrently disseminated events as achieved by the cluster management protocol. In [GKPT05b] we show how the cluster management protocol can be used to implement such a control mechanism which also provides causal delivery of events disseminated in the cluster.

Alternatively, recently proposed dissemination systems implement the publish/subscribe paradigm in combination with structured peer-to-peer systems [RKCD01, ZZJ+01]. For each region of interest the protocols construct an application level multicast tree. Also these protocols assume a maximum number of concurrently disseminated events. Otherwise the dissemination system may overload the source of a multicast-tree and perform unstable thereafter.

The way structured peer-to-peer systems share information in the system (cf. e.g. [SMK+01, AGBH03, RFH+01, RD01, ZHS+04]) has been of relevance and inspiration to this work. Note, however, that uniform hashing, as used in many peer-to-peer systems, is not suitable to solve the cluster management problem since the number of processes is expected to be larger than the number of available tickets in a cluster. Even in the situation of network partitioning the cluster management needs to ensure that no two processes will create an event with respect to the same ticket.

One may notice some similarity between the problem in this paper and the l-exclusion problem [ADHK97, ADG+94]. However, to the best of our knowledge, the solutions to the l-exclusion problem do not satisfy the cluster management problem requirements. Nevertheless, the solution to the cluster management problem proposed here could also serve as solution basis to the l-exclusion problem.

4.6 Discussion and future work

This paper presented and analyzed a solution for a dynamic and fault-tolerant cluster management for event-based peer-to-peer dissemination systems. Since the protocol guarantees that never two processes perform some action corresponding to the same ticket of a cluster, the protocol is suitable for several coordination tasks, such as resource management, controlling the number of concurrently disseminated events, as well as consistency management for replicated distributed objects. The cost of combining the presented solution with an application is low since the duration of a round is longer than the time of a multicast and in each round only a low number of messages are sent. Moreover we have shown how the protocol guarantees access to tickets in spite of failing processes.

Current and future work deals with integrating the cluster management with existing peer-to-peer dissemination algorithms in order to increase reliability as well as to achieve decentralized ordering of messages by maintaining small distributed vector timestamps.

Chapter 5

Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting[1]

Anders Gidenstam Marina Papatriantafilou
Håkan Sundell Philippas Tsigas

Abstract

We present an efficient and practical lock-free implementation of a memory reclamation scheme based on reference counting, aimed for use with arbitrary lock-free dynamic data structures. The algorithm guarantees the safety of local as well as global references, supports arbitrary memory reuse, uses atomic primitives that are available in modern computer systems, and provides an upper bound on the memory prevented for reuse. To the best of our knowledge, this is the first lock-free algorithm that provides all of these properties. Experimental results indicate significant performance improvements for lock-free algorithms of dynamic data structures that require strong garbage collection support.

Keywords: reference counting, memory reclamation, garbage collection, lock-free, shared memory.

[1]This is an extended version of the paper that appeared in the Proceedings of I-SPAN 2005, Las Vegas, USA, December 7-9, 2005.

5.1 Introduction

Memory management is essential for building dynamic concurrent data structures. Concurrent algorithms for data structures and related memory management are commonly based on mutual exclusion. However, mutual exclusion causes blocking and can consequently incur serious problems as deadlocks, priority inversion or starvation. Researchers have addressed these problems by introducing non-blocking synchronization algorithms, which are not based on mutual exclusion. Lock-free algorithms are non-blocking, and guarantee that always at least one operation can progress, independently of the actions taken by the concurrent operations. Wait-free [Her91] algorithms are lock-free, and guarantee that all operations can finish in a finite number of their own steps, regardless of the actions taken by the concurrent operations. It is important in non-blocking algorithms that the effects of the concurrent operations can be observed by the involved processes in a consistent manner. The common consistency requirement is called linearizability [HW90].

In this paper we are focusing on practical and efficient memory management in the context of lock-free dynamic data structures. For an operation of an algorithm to be lock-free, all sub-operations must be at least lock-free. Consequently, lock-free dynamic data structures typically require lock-free memory management. The memory management problem is normally divided into the sub-problems of *dynamic memory allocation* and *garbage collection*.

Valois as well as Michael and Scott [Val95a, MS95] presented a memory allocation scheme for fixed-sized memory segments; this scheme has to be used in combination with the corresponding garbage collection scheme. Lock-free memory allocation schemes for general use have been presented by Michael [Mic04c] and Gidenstam et al. [GPT05].

Various lock-free garbage collection schemes have been presented in the literature.

Michael [Mic02b, Mic04a] proposed the hazard pointer algorithm that focuses on local references. A similar scheme has been proposed by Herlihy et al. [HLM02]; this scheme uses unbounded tags and is based on the double-width CAS atomic primitive, a compare-and-swap operation that can atomically update two adjacent memory words. This is available in some 32-bit architectures, but only in very few of the current 64-bit architectures.

As the aforementioned schemes only guarantee the safety of local pointers from the threads and not the safety of pointers inside dynamically allocated nodes, they cannot support arbitrary lock-free algorithms that might require to always being able to trust global references (i.e. pointers from within the data structure) to objects. This constraint can be strong and restrictive, and may force the data structure algorithms to retry traversals in the possibly large data structures, with resulting large performance penalties that increase with the level of concurrency.

Garbage collection schemes that are based on reference counting can guarantee the safety of global as well as local references to objects. Valois et al. [Val95a, MS95] presented a lock-free reference counting scheme that can be implemented using available atomic primitives, though it is limited to be used only with the corresponding algorithm for memory allocation. Detlefs et al. [DMMS01] presented a scheme that allows also arbitrary reuse of reclaimed memory, but it is based on double-word CAS, which is a compare-and-swap operation that can atomically update two arbitrary memory words. This instruction is not available in any common modern processor architecture.

Herlihy et al. [HLMM02, MLH03] presented a modification of the previous scheme such that it only uses single-word CAS (compare-and-swap) for the reference counting part. However, this scheme relies on another scheme that itself requires double-width CAS, an instruction which, as mentioned above, can atomically update two adjacent memory words and is available only in very few of the current 64-bit architectures.

A problem with reference counting techniques, which was identified in [MS95] is that reference counting techniques can potentially cause a local reference from a slow thread to block (due to the ability of creating recursive references) arbitrarily number of nodes from being reclaimed. Consider for example a chain of nodes that has been removed from a singly linked list in order from the front to back where the slow thread holds a reference to the first deleted node. This node cannot (currently) be reclaimed and would still contain a reference to the next (subsequently) deleted node, preventing it, too, from being reclaimed and so on.

In the context of wait-free memory management, a wait-free extension of Valois' reference counting scheme and memory allocator has been presented by Sundell [Sun05, Sun04b]. Hesselink and Groote [HG98, HG01] have presented a wait-free memory management scheme that is restricted to the specific problem of sharing tokens.

This paper combines the strength of reference counting with the efficiency of hazard pointers, into a general lock-free reference counting scheme, with the aim of keeping only the advantages of the involved techniques while avoiding the respective drawbacks. Our new garbage collection/memory reclamation algorithm is lock-free and linearizable, is compatible with arbitrary schemes for memory allocation, can be implemented using commonly available atomic primitives and guarantee the safety of local as well as global references. We also show how to bound the amount of memory that can be temporarily withheld from reclamation by any thread.

The rest of the paper is organized as follows. In Section 5.2 we describe the type of systems that our implementation is aiming for. Section 5.3 describes the specifics of the problem of garbage collection we are focusing on. The actual algorithm is described in Section 5.4. In Section 5.5 we define the precise semantics of the operations on our implementation, and show the correctness of our algorithm by proving the lock-free and linearizability properties as well as proving an upper bound of the memory that can be temporarily held for reclaiming by our algorithm. Section 5.6 presents an experimental evaluation of the new algorithm in the context of a lock-free data structure. We conclude the paper with Section 5.7.

	Guarantees the safety of shared references (Property 5)	Bounded number of unreclaimed deleted nodes (Property 2)	Compatible with standard memory allocators (Property 4)	Suffices with single-word compare-and-swap
New algorithm	Yes	Yes	Yes	Yes
Detlefs et al. [DMMS01]	Yes	No [e]	Yes	No [a]
Herlihy et al. [HLM02]	No	Yes	Yes	No [b]
Herlihy et al. [HLMM02, MLH03]	Yes	No [e]	Yes	No [c]
Michael [Mic02b, Mic04a]	No	Yes	Yes	Yes [d]
Valois et al. [Val95a, MS95]	Yes	No [e]	No	Yes

[a]The LFRC algorithm uses the double-word compare-and-swap (DCAS) atomic primitive.

[b]The pass-the-buck (PTB) algorithm uses the double-width compare-and-swap atomic primitive.

[c]The SLFRC algorithm is based on the pass-the-buck (PTB) algorithm, and thus uses double-width compare-and-swap.

[d]The hazard pointer algorithm uses only atomic reads and writes.

[e]These reference count-based schemes allow arbitrary long chains of deleted nodes that recursively reference each other to be created. In addition, deleted nodes that cyclically reference each other (i.e. cyclic garbage) will be not be reclaimed ever.

Table 5.1: Properties of different approaches to non-blocking memory management.

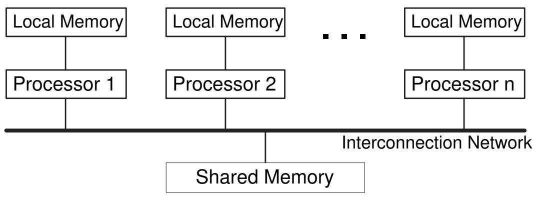


Figure 5.1: Shared memory multiprocessor system structure.

```
procedure FAA(address:pointer to word, number:integer)
    atomic do
        *address := *address + number;

function CAS(address:pointer to word, oldvalue:word,
  newvalue:word):boolean
    atomic do
        if *address = oldvalue then
            *address := newvalue;
            return true;
        else return false;
```

Figure 5.2: The Fetch-And-Add (FAA) and Compare-And-Swap (CAS) atomic primitives.

5.2 System description

A typical abstraction of a shared memory multi-processor system configuration is depicted in Figure 5.1. Each node of the system contains a processor together with its local memory. All nodes are connected to the shared memory via an interconnection network. A set of co-operating tasks is running on the system performing their respective operations. Each task is sequentially executed on one of the processors, while each processor can serve (run) many tasks via multi-programming. The co-operating tasks, possibly running on different processors, use shared data objects built in the shared memory to co-ordinate and communicate. Tasks synchronize their operations on the shared data objects through sub-operations on top of a cache-coherent shared memory. The shared memory may not though be uniformly accessible for all nodes in the system; processors can have different access times on different parts of the memory.

The shared memory system should support atomic read and write operations of single memory words, as well as stronger atomic primitives for synchronization. In this paper we use the Fetch-And-Add (FAA) and the Compare-And-Swap (CAS) atomic primitives; see Figure 5.2 for a description. These read-modify-write style operations are available on most common architectures or can be easily derived from other synchronization primitives with the same or higher consensus number [Moi97, Jay98].

5.3 Problem description

In this paper we are aiming at solving the garbage collection/memory reclamation problem in the context of dynamic lock-free data structures. Lock-free data structures typically consist of a set of memory segments, called *nodes* that each contain arbitrary data. These nodes are interconnected by referencing each other in an arbitrary pattern. The references are typically implemented by using *pointers* that can identify each individual node by the means of memory addresses. Each node may contain an arbitrary number of pointers, called *links*, which reference other nodes. The operation to follow the referenced node through a link is called *dereferencing*. Some nodes are typically always part of the data structure, all others nodes are part of the data structure when they are referenced by a node that itself is a part of the data structure. In a dynamic and concurrent data structure, arbitrary nodes can continuously and concurrently be added or removed from the data structure. As systems have limited amount of memory, the occupied memory of these nodes needs to be dynamically allocated and reclaimed from/to the system.

In a sequential implementation of a data structure, the memory of a node is typically explicitly reclaimed to the system when the last reference to it has been removed, i.e. when the node has been *deleted*. In a concurrent environment this should also include possible local references to a node that any thread might have, as the possible access to the memory of a reclaimed node might be fatal to the correctness of the data structure and/or the whole system. The logical unit that correctly decides about reclaiming is called the *garbage collector* and should thus have the following property:

Property 5.3.1 *The garbage collector should only reclaim possible garbage that is not part of the data structure and for which future access by any thread is not possible.*

It should also always be possible to predict the maximum amount of memory that is used by the data structure, thus adding this requirement to the garbage collector:

Property 5.3.2 *At any time, there should exist an upper bound on the number of nodes that is not part of the data structure, but not yet reclaimed to the system.*

In real implementations of a garbage collector (GC) these properties can be very hard to achieve, as local references to nodes might not be accessible globally (e.g. they might be stored in processor registers). Therefore implementations of GC's typically need to interact with the involved threads and put restrictions on the access to the nodes, e.g. by providing special operations for dereferencing links and demanding that the data structure implementation explicitly calls the garbage collector when a node has been deleted.

Moreover, as the underlying data structures of interest are lock-free and typically also linearizable, the garbage collector also has to guarantee these features:

Algorithm 5.1 The Node structure used by the memory reclamation algorithm.

```
structure Node
    mm_ref: integer /* Initially 0 */
    mm_trace: boolean /* Initially false */
    mm_del: boolean /* Initially false */
    ... /* Arbitrary user data and links follows */
    link[NR_LINKS_NODE]: pointer to Node /* initially NULL */
```

Property 5.3.3 *All operations of the garbage collector for communication with the underlying data structure implementation should be lock-free and linearizable.*

In order to minimize the whole system's total amount of occupied memory for the various data structures, we sometime would like to fulfill the following property:

Property 5.3.4 *The memory that is reclaimed by the garbage collector should be accessible for any arbitrary future reuse; i.e. the garbage collector should be compatible with the system's default memory allocator.*

In a concurrent environment it might frequently occur that a thread is holding a local reference to a node that has been deleted (i.e. removed from the data structure) by some other thread. In these cases it may be very useful for the first thread to be able to use the deleted node's links, e.g. in search procedures in large data structures:

Property 5.3.5 *A thread that has a local reference to a node, should also be able to dereference all of the links that are contained in that node.*

The new algorithm in this paper fulfills all of these properties in addition to the property of only using atomic primitives that are commonly available in modern systems. Table 5.1 shows a comparison of the fulfilled properties with previously presented lock-free garbage collection schemes. All of the schemes fulfill properties 1 and 3, whereas only a subset of the other properties is met by the previously presented schemes.

5.4 The new lock-free algorithm

In order to fulfill all of the requested properties in Section 5.3 as well as to provide an efficient and practical method, our aim is to devise a reference counting method which can also employ the *hazard pointer* (HP) scheme of Michael [Mic02b, Mic04a]. Roughly speaking, hazard pointers are used for guaranteeing the safety of local references and reference counts for guaranteeing the safety of internal links in the data structure. Thus, the reference count of each node should indicate the number of globally accessible links that reference that node. Algorithm 5.1 describes the node structure as it is used in our algorithm. As in the HP scheme, each thread maintains a list of nodes that are deleted but not yet reclaimed, and this list is scanned for possible reclamation when its length has reached a certain threshold (i.e. THRESHOLD_2). Some of the deleted nodes might be prevented from reclamation because of a fixed number of hazard pointers, while some deleted nodes might be prevented because of a positive reference count adherent to links. Thus, it is important to keep the number of references to deleted nodes from links to a minimum. Before we continue with the techniques for bounding the size of the deletion lists, we introduce an assumption about what could be required by the lock-free data structure algorithm:

Assumption 5.4.1 *For each of the links in a deleted node that reference a deleted node, it should be possible to replace it with a reference to an active node, with retained semantics for any of the involved threads.*

The intuition behind this assumption lays behind an observation why links of a deleted node should be useful to dereference by a thread that has a local reference to it. The thread with a local reference to a deleted node surely wants to find an appropriate active node and therefore takes advantage of the links. If the corresponding reference also adheres to a deleted node, the previous step is repeated. From the point of view of the thread of interest, it would not make any difference if some other thread helped with the procedure and already made sure that the links of the deleted node all references active node. The procedure of replacing the links of a deleted node with references to active nodes is called *clean-up*.

As described earlier, besides hazard pointers, nodes in the deletion lists are possibly prevented from reclamation by links of other deleted nodes. These nodes might be in the same deletion list or in some other thread's deletion list. For this reason, all threads' deletion lists are accessible for reading by any thread. When the length of the deletion list reaches a certain threshold (THRESHOLD_1) the thread performs a clean-up of all the nodes in its deletion list. If all of the nodes are still prevented from reclamation, this must be due to nodes in some other thread's deletion list, and thus the thread tries to perform a clean-up of all of the other threads' deletion lists as well. As this procedure is repeated until the length of the deletion list is below the threshold, the amount of deleted nodes that are not yet reclaimed is bounded. The actual calculation of THRESHOLD_1 is described in 5.5.2. The threshold THRESHOLD_2 is set according to the HP scheme or less or equal than THRESHOLD_1.

5.4.1 Application programming interface

The following functions are defined for safe handling of the reference counted nodes:

```
function DeRefLink(link:pointer to pointer to Node): pointer to Node
procedure ReleaseRef(node:pointer to Node)
function CompareAndSwapRef(link:pointer to pointer to Node, old:pointer
to Node, node:pointer to Node): boolean
```

Algorithm 5.2 Reference counting algorithm: global and local variables.

```
/* Global variables */
HP[NR_THREADS][NR_INDICES]: pointer to Node;
DL_Nodes[NR_THREADS][THRESHOLD_1]: pointer to Node;
DL_Claims[NR_THREADS][THRESHOLD_1]: integer;
DL_Done[NR_THREADS][THRESHOLD_1]: boolean;
/* the above matrixes should be initialized to the values of
NULL, NULL, 0 respective false */

/* Local static variables */
threadId: integer; /* Unique and fixed number for each thread
  between 0 and NR_THREADS-1 */
dlist: integer; /* Initially ⊥ */
dcount: integer; /* Initially 0 */
DL_Nexts[THRESHOLD_1]: integer;

/* Local temporary variables */
node, node1, node2, old: pointer to Node;
thread, index, new_dlist, new_dcount: integer;
plist: array of pointer to Node;
```

```
    procedure StoreRef(link:pointer to pointer to Node, node:pointer to Node)
    function NewNode:pointer to Node
    procedure DeleteNode(node:pointer to Node)
```

The function *DeRefLink* safely de-references a given link, and sets a hazard pointer to the de-referenced node, thus guaranteeing the future safety to access the returned node. The procedure *ReleaseRef* should be called when a given node will not be accessed by the current thread anymore. It will clear the corresponding hazard pointer.

To update a link for which there might be concurrent updates to the link, the function *CompareAndSwapRef* should be used, which gives result whether the update was successful or not. The procedure will make sure that any thread that calls *DeRefLink* on the link can safely do so, if the thread has a hazard pointer reference to the node that contains the link. The requirements are that the calling thread of *CompareAndSwapRef* should have a hazard pointer to the given node that should be stored.

To update a link for which there cannot be any concurrent updates the procedure *StoreRef* should be called. The procedure will make sure that any thread that calls *DeRefLink* on the link can safely to so, if the thread has a hazard pointer reference to the node that contains the link. The requirements are that the calling thread of *StoreRef* should have a hazard pointer to the given node that should be stored, and that no other thread will possibly write concurrently to the link (otherwise *CompareAndSwapRef* should be invoked instead).

The function *NewNode* allocates a new node, sets a free hazard pointer to it for guaranteeing the future safety for access, and then returns it. The procedure *DeleteNode* should be called when a node is removed from the data structure and which memory should be possible to reclaim for reuse. The user operation

Algorithm 5.3 Reference counting functions, part I.

```
function DeRefLink(link:pointer to pointer to Node):
  pointer to Node
D1   Choose index such that HP[threadId][index]=NULL
D2   while true do
D3       node := *link;
D4       HP[threadId][index] := node;
D5       if *link = node then
D6           return node;

procedure ReleaseRef(node:pointer to Node)
R1   Choose index such that HP[threadId][index]=node
R2   HP[threadId][index]:= NULL;

function CompareAndSwapRef(link:pointer to pointer to Node,
  old: pointer to Node, node: pointer to Node): boolean
C1   if CAS(link,old,node) then
C2       if node ≠ NULL then
C3           FAA(&node.mm_ref,1);
C4           node.mm_trace:=false;
C5       if old ≠ NULL then FAA(&old.mm_ref,-1);
C6       return true;
C7   return false;

procedure StoreRef(link:pointer to pointer to Node,
  node: pointer to Node)
S1   old := *link;
S2   *link := node;
S3   if node ≠ NULL then
S4       FAA(&node.mm_ref,1);
S5       node.mm_trace:=false;
S6   if old ≠ NULL then FAA(&old.mm_ref,-1);

function NewNode : pointer to Node
NN1  node := Allocate the memory of node (e.g. using malloc)
NN2  node.mm_ref := 0;
NN3  node.mm_del := false;
NN4  Choose index such that HP[threadId][index]=NULL
NN5  HP[threadId][index] := node;
NN6  return node;
```

Algorithm 5.4 Reference counting functions, part II.

```
procedure DeleteNode(node:pointer to Node)
DN1  ReleaseRef(node);
DN2  node.mm_del := true; node.mm_trace := false;
DN3  Choose index such that DL_Nodes[threadId][index]=NULL
DN4  DL_Done[threadId][index]:=false;
DN5  DL_Nodes[threadId][index]:=node;
DN6  DL_Nexts[index]:=dlist;
DN7  dlist := index; dcount := dcount + 1;
DN8  while true do
DN9       if dcount = THRESHOLD_1 then CleanUpLocal();
DN10      if dcount ≥ THRESHOLD_2 then Scan();
DN11      if dcount = THRESHOLD_1 then CleanUpAll();
DN12      else break;
```

that called *DeleteNode* is responsible for removing all references to the deleted node from the active nodes in the data-structure. This is similar to what is required when using a memory allocator in a sequential data-structure. The memory manager will not reclaim the deleted node until it is safe to do so.

In Section 5.4.4 we give an example of how these functions can be used in the context of a lock-free queue algorithm based on linked lists.

Callbacks

The following functions are callbacks that have to be defined by the designer of each specific data structure:

procedure CleanUpNode(node:**pointer to** Node)
procedure TerminateNode(node:**pointer to** Node, concurrent:**boolean**)

The procedure *TerminateNode* will make sure that none of the links in the given node will have any claim on any other node. *TerminateNode* is called on a deleted node when there are no claims from any other node or thread to the node.[2]

The procedure *CleanUpNode* will make sure that all claimed references from the links of the given node will only point to active nodes, thus removing redundant passages through an arbitrary number of deleted nodes.

5.4.2 Auxiliary procedures

Auxiliary functions that are defined for internal use by the reference counting algorithm:

procedure Scan()
procedure CleanUpLocal()
procedure CleanUpAll()

[2]In principle this procedure could be provided by the memory manager but in practice it is more convenient to let the user decide the memory layout of the node records. All node records would still be required to start with the `mm_ref`, `mm_trace` and `mm_del` fields.

Algorithm 5.5 Callback functions.

```
procedure TerminateNode(node:pointer to Node,concurrent:boolean)
TN1 if not concurrent then
TN2     for all x where link[x] of node is reference-counted do
TN3         StoreRef(node.link[x],NULL);
TN4 else
TN5     for all x where link[x] of node is reference-counted do
TN6         repeat node1 := node.link[x];
TN7         until CompareAndSwapRef(&node.link[x],node1,NULL);

procedure CleanUpNode(node:pointer to Node)
CN1 for all x where link[x] of node is reference-counted do
    retry:
CN2     node1:=DeRefLink(&node.link[x]);
CN3     if node1 $\neq$ NULL and node1.mm_del then
CN4         node2:=DeRefLink(&node1.link[x]);
CN5         CompareAndSwapRef(&node.link[x],node1,node2);
CN6         ReleaseRef(node2);
CN7         ReleaseRef(node1);
CN8         goto retry;
CN9     ReleaseRef(node1);
```

Algorithm 5.6 Internal functions, part I.

```
procedure CleanUpLocal()
CL1 index := dlist;
CL2 while index $\neq \perp$ do
CL3     node:=DL_Nodes[threadId][index];
CL4     CleanUpNode(node);
CL5     index := DL_Nexts[index];

procedure CleanUpAll()
CA1 for thread := 0 to NR_THREADS-1 do
CA2     for index := 0 to THRESHOLD_1-1 do
CA3     node:=DL_Nodes[thread][index];
CA4     if node $\neq$ NULL and not DL_Done[thread][index] then
CA5         FAA(&DL_Claims[thread][index],1);
CA6         if node = DL_Nodes[thread][index] then
CA7             CleanUpNode(node);
CA8         FAA(&DL_Claims[thread][index],-1);
```

Algorithm 5.7 Internal functions, part II.

```
procedure Scan()
SC1  index := dlist;
SC2  while index $\neq \perp$ do
SC3      node:=DL_Nodes[threadId][index];
SC4      if node.mm_ref = 0 then
SC5          node.mm_trace := true;
SC6          if node.mm_ref $\neq$ 0 then node.mm_trace := false;
SC7      index := DL_Nexts[index];
SC8  plist := $\emptyset$; new_dlist:=$\perp$; new_dcount:=0;
SC9  for thread := 0 to NR_THREADS-1 do
SC10     for index := 0 to NR_INDICES-1 do
SC11         node := HP[thread][index];
SC12         if node $\neq$ NULL then
SC13             plist := plist + node;
SC14 Sort and remove duplicates in array plist
SC15 while dlist $\neq \perp$ do
SC16     index := dlist;
SC17     node:=DL_Nodes[threadId][index];
SC18     dlist := DL_Nexts[index];
SC19     if node.mm_ref = 0 and node.mm_trace and node $\notin$ plist then
SC20         DL_Nodes[threadId][index]:=NULL;
SC21         if DL_Claims[threadId][index] = 0 then
SC22             TerminateNode(node,false);
SC23             Free the memory of node
SC24             continue;
SC25         TerminateNode(node,true);
SC26         DL_Done[threadId][index]:=true;
SC27         DL_Nodes[threadId][index]:=node;
SC28     DL_Nexts[index]:=new_dlist;
SC29     new_dlist := index;
SC30     new_dcount := new_dcount + 1;
SC31 dlist := new_dlist;
SC32 dcount := new_dcount;
```

The procedure *Scan* will search through all not yet reclaimed nodes deleted by this thread and reclaim only those that does not have any matching hazard pointer and do not have any counted references from any links inside of nodes. The procedure *CleanUpLocal* will try to remove redundant claimed references from links in deleted nodes that has been deleted by this thread. The procedure *CleanUpAll* will try to remove redundant claimed references from links in deleted nodes that has been deleted by any thread.

5.4.3 Detailed algorithm description

DeRefLink (Algorithm 5.3), first reads the pointer to a node stored in `*link` at line D3. Then at line D4 it sets one of the thread's hazard pointers to point to the node. At line D5 it verifies that the link still points to the same node as before. If `*link` still points to the node, it knows that the node is still not yet reclaimed and that it cannot not be reclaimed until the hazard pointer now pointing to it is released. If `*link` has changed since the last read, it retries.

ReleaseRef (Algorithm 5.3), removes this thread's hazard pointer pointing to `node`. Note that if the node `node` is deleted, has a reference count of zero and no other hazard pointers are pointing to it, the node can now be reclaimed by *Scan* (invoked by the thread that called *DeleteNode* on the node).

CompareAndSwapRef (Algorithm 5.3), performs a common CAS on the link and updates the reference counts of the respective nodes accordingly. Line C4 notifies any concurrent *Scan* that the reference count of `node` has been increased. Notice that the node `node` is safe to access during *CompareAndSwapRef* since the thread calling *CompareAndSwapRef* is required to have a hazard pointer pointing to it. At line C5 the reference count of `old` is decreased. The previous reference count must have been greater than zero since `*link` referenced the node `old`.

StoreRef (Algorithm 5.3), is valid to use only when there are no concurrent updates of `*link`. After updating `*link` at line S2 *StoreRef* increases the reference count of `node` at line S4, which is safe since the thread calling *StoreRef* is required to have a hazard pointer to the node `node`. Line S5 notifies any concurrent *Scan* that the reference count of `node` is non-zero. At line S6 the reference count of `old` is decreased. The previous reference count must have been greater than zero since `*link` referenced the node `old`.

NewNode (Algorithm 5.3), allocates memory for the new node from the underlying memory allocator and initializes the header fields each node should have. It also sets a hazard pointer to the node.

DeleteNode (Algorithm 5.4), marks the node `node` as logically deleted at line DN2. Then at the lines DN3 to DN7 the node is inserted into this thread's set of deleted but not yet reclaimed nodes. By clearing `DL_Done` at line DN4 before writing the pointer at line DN5 concurrent *CleanUpAll* operations can access the node and tidy up its references.

If the number of deleted nodes in this thread's set of deleted but not yet reclaimed nodes is larger than or equal to THRESHOLD_2 a *Scan* is performed which will reclaim all nodes in the set that are not referenced by other nodes or threads.

If the thread's set of deleted but not yet reclaimed nodes is now full, that is, it contains THRESHOLD_1 nodes, the thread will first run *CleanUpLocal* at line DN9 to make sure that all of its deleted nodes only points to nodes that were alive when *CleanUpLocal* started. Then it runs *Scan* at line DN10. If *Scan* is unable to reclaim any node at all then the thread will run *CleanUpAll*, which cleans up the sets of deleted nodes of all threads.

Callbacks

TerminateNode (Algorithm 5.5), should clear all links in the node `node` by writing NULL to the links in `node`. This is done by using either *CompareAndSwapRef* or *StoreRef* depending on whether there might be concurrent updates of these links or not.

CleanUpNode (Algorithm 5.5), should make sure that none of the links of the node `node` points to nodes that were deleted before this invocation of *CleanUpNode* started.

Auxiliary procedures

Scan (Algorithm 5.7), reclaims all nodes deleted by the current thread that are not referenced by any other node or any hazard pointer. To determine which of the deleted nodes that can safely be reclaimed *Scan* first sets the `mm_trace` bit of all deleted nodes that have reference count zero (lines SC1 to SC7). The check at line SC6 ensures that the reference count was indeed zero when the `mm_trace` bit was set.

Then *Scan* records all active hazard pointers of all threads in plist (lines SC8 to SC14). In the lines SC15 to SC30 *Scan* traverses all not yet reclaimed nodes deleted by this thread. For each of these nodes the tests at line SC19 determine if (i) the reference count is zero, (ii) the reference count has consistently been zero since before the hazard pointers were read (indicated by the `mm_trace` bit being set) and (iii) the node is not referenced by any hazard pointer. If all three of these conditions are true, the node is not referenced and *Scan* checks if there may be concurrent *CleanUpAll* operations working on the node at line SC21. If there are no such *CleanUpAll* operations *Scan* uses *TerminateNode* to release all references the node might contain and then reclaim the node (lines SC22 and SC23). In case there might be concurrent *CleanUpAll* operations accessing the node *Scan* uses the concurrent version of *TerminateNode* to set all of the node's links to NULL. By setting the `DL_Done` flag at line SC26 before the node is reinserted into the set of unreclaimed nodes at line SC27 later *CleanUpAll* operations cannot prevent this node from being reclaimed by a subsequent *Scan*.

CleanUpLocal (Algorithm 5.6), traverses the thread's list of deleted but unreclaimed nodes and calls *CleanUpNode* on each of them to make sure that their links do not reference any nodes that were already deleted when *CleanUpLocal* started.

CleanUpAll (Algorithm 5.6), traverses the `DL_Nodes` arrays of all threads and try to make sure that none of the nodes it finds contains links to nodes that were already deleted when *CleanUpAll* started. The tests at line CA4 prevent *CleanUpAll* from needlessly interfere with *Scan* for nodes that have no references left. The test at line CA6 prevents *CleanUpAll* from accessing a node that *Scan* has already reclaimed. If the node is still present in `DL_Nodes[thread][index]` at line CA6 then a concurrent *Scan* accessing this node must be before line SC20 or be after line SC27 without having reclaimed the node.

5.4.4 Example application

The application of the new algorithm for memory management to lock-free algorithms for dynamic data structures can be done straight forward in a similar manner to previously presented memory management schemes. Algorithm 5.8 shows the lock-free queue algorithm by Valois et al. [Val94, MS95] as it would be integrated with the new algorithm for memory management.

5.4.5 Algorithm extensions

For simplicity reasons, the algorithm in this paper is described with a fixed number of threads. However, the algorithm can easily be extended for a dynamic number of threads in a similar way as described for the HP scheme in [Mic04a]. The global matrix of hazard pointers (*HP*) can be turned into a linked list of arrays. The deletion lists can also be linked into a global chain, and as the size of the deletion lists changes, old redundant deletion lists can be safely reclaimed by using an additional HP scheme for memory management.

5.5 Correctness proof

In this section we present the correctness proof of our algorithm. The outcome of the correctness proof is the following theorems that state the most important properties of our algorithm.

Theorem 5.5.1 *The algorithm implements a lock-free and linearizable algorithm for garbage collection.*

Theorem 5.5.2 *The number of deleted but not yet reclaimed nodes in the system is bounded from above by*

$$N^2 \cdot (k + l_{max} + \alpha + 1),$$

where N is the number of threads in the system, k is the number of hazard pointers per thread, l_{max} is the maximum number of links a node can contain and α is the maximum number of links in live nodes that may transiently point to a deleted node.

Algorithm 5.8 Example of a queue algorithm using the new memory management scheme.

```
structure QNode
    mm_ref: integer
    mm_trace: boolean
    mm_del: boolean
    next: pointer to QNode
    value: pointer to Value

/* Global variables */
head, tail:pointer to QNode

procedure InitQueue()
IQ1  node := NewNode();
IQ2  node.next := NULL;
IQ3  head := NULL; tail:= NULL;
IQ3  StoreRef(&head,node);
IQ4  StoreRef(&tail,node);

function Dequeue():pointer to Value
DQ1  while true do
DQ2      node1 := DeRefLink(&head);
DQ3      next := DeRefLink(&node2.next);
DQ4      if next = NULL return NULL;
DQ5      if CompareAndSwapRef(&head,node1,next) then break;
DQ6      ReleaseRef(node1); ReleaseRef(next);
DQ7  DeleteNode(node1);
DQ8  value := next.value; ReleaseRef(next);
DQ9  return value;

procedure Enqueue(value:pointer to Value)
EQ1  node := NewNode();
EQ2  node.next := NULL; node.value := value;
EQ3  old := DeRefLink(&tail); prev := old;
EQ4  repeat
EQ5      while prev.next ≠ NULL do
EQ6          prev2 := DeRefLink(&prev.next);
EQ7          if old ≠ prev then ReleaseRef(prev);
EQ8          prev := prev2;
EQ9  until CompareAndSwapRef(&prev.next,NULL,node);
EQ10 CompareAndSwapRef(&tail,old,node);
EQ11 if old ≠ prev then ReleaseRef(prev);
EQ12 ReleaseRef(old); ReleaseRef(node);
```

The above theorems are proved below using a series of lemmas. We first prove that our algorithm does not reclaim memory that could still be accessed, then we prove an upper bound on the amount of such deleted but unreclaimed garbage there can be and last we prove that the algorithm is a linearizable and lock-free one [HW90]. A set of definitions that will help us to structure and shorten the proof is first described in this section. We start by defining the sequential semantics of our operations.

Definition 5.5.1 *Let F_t denote the state of the pool of free nodes at time t. We interpret $n \in F_t$ to be true when n has been freed as per line SC23 in* Scan*. Any (preferably lock-free) memory allocator can be used to manage the free pool.*

Let $n \in HP_t(p)$ denote that thread p has a verified hazard pointer set to point to node n at time t. A verified hazard pointer is one that has been or will be returned by a successful DeRefLink *operation. The array of hazard pointers in the implementation, the array* HP*, may also contain pointers temporarily set by unsuccessful* DeRefLink *operations, but these are not considered as part of the $HP_t(p)$ sets.*

Let $n \in DL_t(p)$ denote that node n is deleted and is awaiting reclamation in the dlist *of thread p at time t.*

Let $Del_t(n)$ denote that the node n is marked as logically deleted at time t. The deletion mark is not removed until the node is returned to the free pool.

Let $Links(n)$ denote the set of shared links (pointers) present in node n.

Let $l_x \mapsto_t n_x$ denote that the shared link l_x points to node n_x at time t.

Let $Ref_t(n)$ denote a set containing the shared links that point to the node n at time t. A shared link is either a global shared variable visible to the application or a pointer variable residing inside a node. Specifically, the elements in the per thread arrays of hazard pointers, HP*, and the per thread arrays of deleted nodes,* DL_Nodes*, are* not *considered as shared links, since these are internal to the memory management.*

The operations that are of interest for linearizability are DeRefLink *(DRL),* ReleaseRef *(RR),* NewNode *(NN),* DeleteNode *(DN) and* CompareAndSwapRef *(CASR). For the safety and correctness of the memory management the following additional internal operations are also of interest:* TerminateNode *(TN),* Scan *(SCAN),* CleanUpNode *(CUN),* CleanUpLocal *(CUL),* CleanUpAll *(CUA).*

In the following expressions which define the sequential semantics of our operations, the syntax is $S_1 : O_1, S2$, where S_1 is the conditional state before the operation O_1 and S_2 is the resulting state after the operation has been performed.

DeRefLink

$$\exists n_1 . l_1 \mapsto_{t_1} n_1 : \mathbf{DRL(l_1)} = n_1, n_1 \in HP_{t_2}(p_{curr}) \tag{5.1}$$

$$l_1 \mapsto_{t_1} \perp : \mathbf{DRL(l_1)} = \perp, \tag{5.2}$$

ReleaseRef

$$n \in HP_{t_1}(p_{curr}) : \mathbf{RR(n_1)}, n \notin HP_{t_2}(p_{curr}) \tag{5.3}$$

NewNode

$$\begin{array}{l}\exists n_1 . n_1 \in F_{t_1} : \\ \quad \mathbf{NN}() = \mathbf{n_1}, \\ \qquad n_1 \notin F_{t_2} \wedge Ref_{t_2}(n_1) = 0 \wedge \neg Del_{t_2}(n_1) \\ \qquad \wedge n_1 \in HP_{t_2}(p_{curr})\end{array} \tag{5.4}$$

DeleteNode

$$\begin{array}{l}n_1 \in HP_{t_1}(p_{curr}) : \\ \quad \mathbf{DN(n_1)}, \\ \qquad Del_{t_2}(n_1) \wedge n_1 \in DL_{t_2}(p_{curr}) \\ \qquad \wedge n_1 \notin HP_{t_2}(p_{curr})\end{array} \tag{5.5}$$

CompareAndSwapRef

$$\begin{array}{l}l_1 \mapsto_{t_1} \perp \wedge n_2 \in HP_{t_1}(p_{curr}) : \\ \quad \mathbf{CASR(l_1, \perp, n_2) = True}, \\ \qquad l_1 \mapsto_{t_2} n_2 \wedge l_1 \in Ref_{t_2}(n_2) \wedge \\ \qquad n_2 \in HP_{t_2}(p_{curr})\end{array} \tag{5.6}$$

$$\begin{array}{l}\exists n_1 \ . \ l_1 \mapsto_{t_1} n_1 \wedge n_2 = n_1 : \\ \quad \mathbf{CASR(l_1, n_2, \perp) = True}, \\ \qquad l_1 \mapsto_{t_2} \perp \wedge l_1 \notin Ref_{t_2}(n_2)\end{array} \tag{5.7}$$

$$\begin{array}{l}\exists n_1 \ . \ l_1 \mapsto_{t_1} n_1 \wedge n_2 = n_1 \wedge \\ n_3 \in HP_{t_1}(p_{curr}) \wedge l_1 \in Ref_{t_1}(n_2) : \\ \quad \mathbf{CASR(l_1, n_2, n_3) = True}, \\ \qquad l_1 \mapsto_{t_2} n_3 \wedge l_1 \notin Ref_{t_2}(n_2) \wedge \\ \qquad l_1 \in Ref_{t_2}(n_3) \wedge n_3 \in HP_{t_2}(p_{curr})\end{array} \tag{5.8}$$

$$\begin{array}{l}\exists n_1 \ . \ l_1 \mapsto_{t_1} n_1 \wedge n_1 \neq n_2 \wedge n_3 \in HP_{t_1}(p_{curr}) : \\ \quad \mathbf{CASR(l_1, n_2, n_3) = False}, \\ \qquad l_1 \mapsto_{t_2} n_1 \wedge n_3 \in HP_{t_2}(p_{curr})\end{array} \tag{5.9}$$

Scan

$$\begin{array}{l}: \\ \quad \mathbf{Scan}(), \\ \qquad \forall n_i \in DL_{t_1}(p_{curr}).(n_i \in F_{t_2} \wedge \\ \qquad (\forall n_x \text{ s.t. } l_x \mapsto_{t_1} n_x \wedge l_x \in Links(n_i). \\ \qquad l_x \notin Ref_{t_2}(n_x)) \vee (\exists p_j . n_i \in HP_{t_1}(p_j)) \vee \\ \qquad (\exists n_j . n_j \notin F_{t_1} \wedge \exists l_x \in Links(n_j). l_x \mapsto_{t_1} n_i)\end{array} \tag{5.10}$$

TerminateNode (Implemented by the application programmer).

$$\begin{array}{l}n_1 \in DL_{t_1}(p_{curr}) : \\ \quad \mathbf{TerminateNode(n_1, c)}, \\ \qquad \forall l_x \in Links(n_1).(l_x \mapsto_{t_2} \perp \wedge \\ \qquad \forall n_x \text{ s.t. } l_x \mapsto_{t_1} n_x \ . \ l_x \notin Ref_{t_2}(n_x)\end{array} \tag{5.11}$$

CleanUpNode (Implemented by the application programmer).

$$\begin{array}{l} \exists p_i . n_1 \in DL_{t_1}(p_i) \wedge Del_{t_1}(n_1) : \\ \quad \mathbf{CleanUpNode(n_1)}, \\ \quad\quad \forall l_x \in Links(n_1).(l_x \mapsto_{t_2} \bot \vee \\ \quad\quad (\exists n_x . l_x \mapsto_{t_2} n_x \wedge \neg Del_{t_1}(n_x))) \end{array} \tag{5.12}$$

CleanUpLocal

$$\begin{array}{l} : \\ \quad \mathbf{CleanUpLocal}(), \\ \quad\quad \forall n_i \in DL_{t_1}(p_{curr}).(\forall l_x \in Links(n_i). \\ \quad\quad l_x \mapsto_{t_2} \bot \vee (\exists n_x . l_x \mapsto_{t_2} n_x \wedge \neg Del_{t_1}(n_x))) \end{array} \tag{5.13}$$

CleanUpAll

$$\begin{array}{l} : \\ \quad \mathbf{CleanUpAll}(), \\ \quad\quad \forall p_i .(\forall n_j \in DL_{t_1}(p_i).(\forall l_x \in Links(n_j). \\ \quad\quad l_x \mapsto_{t_2} \bot \vee (\exists n_x . l_x \mapsto_{t_2} n_x \wedge \neg Del_{t_1}(n_x)))) \end{array} \tag{5.14}$$

StoreRef (Can only be used to update links in nodes that are inaccessible to all other threads.)

$$\begin{array}{l} l_1 \mapsto_{t_1} \bot \wedge n_2 \in HP_{t_1}(p_{curr}) : \\ \quad \mathbf{SR(l_1, n_2)}, \\ \quad\quad l_1 \mapsto_{t_2} n_2 \wedge l_1 \in Ref_{t_2}(n_2) \wedge n_2 \in HP_{t_2}(p_{curr}) \end{array} \tag{5.15}$$

$$\begin{array}{l} \exists n_1 . l_1 \mapsto_{t_1} n_1 \wedge n_2 \in HP_{t_1}(p_{curr}) : \\ \quad \mathbf{SR(l_1, n_2)}, \\ \quad\quad l_1 \mapsto_{t_2} n_2 \wedge l_1 \notin Ref_{t_2}(n_1) \wedge \\ \quad\quad l_1 \in Ref_{t_2}(n_2) \wedge n_2 \in HP_{t_2}(p_{curr}) \end{array} \tag{5.16}$$

Definition 5.5.2 *A node n is said to be* reclaimable *at time t iff $Ref_t(n) = \emptyset$ and $\forall p . n \notin HP_t(p)$ and $Del_t(n)$.*

5.5.1 Safety

Lemma 5.5.1 *If a node n is* reclaimable *at time t_1 then $Ref_t(n) = \emptyset$ for all $t \geq t_1$.*

Proof: Assume towards a contradiction that a node n was reclaimable at time t_1 and that later at time t_2 $Ref(n)_{t_2} \neq \emptyset$. From the definition of reclaimable follows that at time t_1 there were no shared links pointing to n and no process had a hazard pointer to n.

Then, clearly, n has to have been stored to some shared link l after time t_1 and before t_2. There are only two operations that can update a shared link: *StoreRef* (SR) and *CompareAndSwapRef* (CASR). However both of these

operations require that process issuing the operation has a hazard pointer to n at that time. There are two cases:
(i) The process has had a hazard pointer set to n since already before time t_1. This is impossible since there were no hazard pointers to n at time t_1.
(ii) The process set the hazard pointer to n at a time later than t_1. This is impossible as the only way to set a hazard pointer to an existing node is the *DeRefLink* operation and it requires that there is a shared link pointing to n to dereference and at time t_1 there are no such links. (See also the linearizability proof for *DeRefLink*)

So since there cannot be any processes capable of using *StoreRef* or *CompareAndSwapRef* to update a link to point to n at time t_1 or later we have that $Ref_t(n) = \emptyset$ for all $t \geq t_1$. $\square$

Lemma 5.5.2 *A node n returned by a DeRefLink(l_1) operation performed by a process p is not* reclaimable *and cannot become* reclaimable *before a corresponding ReleaseRef(n) operation is performed by the same process.*

Proof: The node n is not reclaimable when *DeRefLink* returns because p has set a hazard pointer to point to n at line D4. Furthermore, line D5 verifies that n is referenced by l_1 also after the hazard pointer was set which guarantees that n cannot have become reclaimable between line D3 and D4 since n is still referenced by a shared link.[3][4] $\square$

Lemma 5.5.3 *The mm_ref field together with the hazard pointers provides a safe approximation of the $Ref_t(n)$ set.*

Proof: The reference count field, mm_ref, in each node approximates the set of links referencing n $Ref(n)$. As such the mm_ref field of a node n is only guaranteed to be accurate when there are no ongoing[5] operations concerning n. The only operations that may change the mm_ref field of a node n are *CompareAndSwapRef* and *StoreRef*.

For the memory management scheme the critical aspect of the $Ref(n)$ set is to know whether it is empty or non-empty to determine if the node is reclaimable or not. In particular, the important case is when the mm_ref field is to be increased, since delaying a decrease of the reference count will not compromise the safety of the memory management scheme.

Thus, although the mm_ref field of a node n to be stored in a shared link by a *CompareAndSwapRef* or *StoreRef* operation is not increased in the same atomic time instant as the operation takes effect, it *does not matter* for the safety of the memory management scheme since the node is clearly not reclaimable

[3]Note 1: Between D3 and D5 n might have been moved away from l_1 and then moved back again.

[4]Note 2: Between D3 and D4 the "original" n could actually have been removed and reclaimed and then the same memory could be reused for a new node n which is stored in l_1 before D5. This is no problem as the "new" n is what the DeRefLink really returns.

[5]Consider any crashed operations as ongoing.

during the duration of the operation anyway, since the process performing the operation is required to have a hazard pointer set to n. □

Lemma 5.5.4 *The operation Scan will never reclaim a node n that is not reclaimable.*

Proof: *Scan* is said to reclaim a node n when it is returned to the pool of free memory, which takes place at line SC23.

Assume that *Scan* reclaimed a node n at time t_3 and let time t_1 and t_2 denote the time *Scan* executed line SC5 and line SC19 for the node n, respectively.

First, note that there exists no process p such that $n \in HP(p)$ during the whole interval between t_1 and t_2 since such a hazard pointer would be detected by *Scan* (lines SC9 - SC13). Consequently, any process that is able to access n after time t_3 must have dereferenced (with *DeRefLink*) a shared link pointing n after time t_1.

Second, since *Scan* is reclaiming the node, we know that the mm_trace field of n, which were set to **true** at line SC5, and the mm_ref field, which was verified to be zero at line SC6, still had those values when line SC19 was reached. This implies that:
(i) There were no *StoreRef* or *CompareAndSwapRef* operations to store n in a shared link that started before t_1 and had not finished before t_2, since the hazard pointers to n these operations require would have been detected when *Scan* searched the hazard pointers at lines SC9 - SC13.
(ii) There were no *StoreRef* or *CompareAndSwapRef* operations to store n in a shared link that finished between t_1 and t_2, as a such operation would have cleared the mm_trace field and thereby caused the comparison at SC19 to fail.
Therefore, there were no ongoing *StoreRef* or *CompareAndSwapRef* operation to store n in a shared link at the time *Scan* executed line SC5 and, consequently, as these operations are the only ones that can increase the mm_ref field, we have $Ref_{t_1}(n) = \emptyset$. Further, because of (ii) there cannot have been any *StoreRef* or *CompareAndSwapRef* operation to store n in a shared link that started and finished between t_1 and t_2.

Since $Ref_{t_1}(n) = \emptyset$ no *DeRefLink* operation can finish by successfully dereference n after time t_1 unless n is stored to a shared link after time t_1. However, as we have seen above such a store operation must begins after time t_1 and finish after time t_2 and the process performing it must therefore, by our first observation that no single process could have held a hazard pointer to n during the whole interval between t_1 and t_2, have dereferenced n after t_1. This is a clearly a contradicition and therefore it is impossible for any process to successfully dereference n after time t_1.

From the above we have that $Ref_t(n) = \emptyset$ for $t \geq t_1$ and $\forall p \,.\, n \notin HP_t(p)$ for $t \geq t_2$ and therefore, since $t_3 > t_2 > t_1$, n is reclaimable at time t_3 . □

Lemma 5.5.5 *The operation Scan will never reclaim a node n that is accessed by a concurrent CleanUpAll operation.*

Proof: Before reclaiming the node n stored at position i in its DL_Nodes array *Scan* writes NULL into DL_Nodes[i] (line SC20) and then checks that DL_Claims[i] is zero (line SC21).

Before accessing the node n the *CleanUpAll* operation reads n from DL_Nodes[i] (line CA3), then increases DL_Claims[i] (line CA5) and then verifies that DL_Nodes[i] still contains n (line CA6).

Now, for a concurrent *CleanUpAll* operation to also access n it has to do both reads of DL_Nodes[i] (line CA3 and line CA5) before *Scan* performs line SC20, but then DL_Claims[i] has been increased (line CA3) and *Scan* will detect this at line SC21 and will not reclaim the node.

If, on the other hand, *Scan* reads a claim count of 0 at line SC21, then the concurrent *CleanUpAll* operation will read NULL from DL_Nodes[i] at line CA6 and will not access the node. □

5.5.2 Bounding the number of deleted but unreclaimed nodes

Theorem 5.5.3 *For each thread p_i the maximum number of deleted but not reclaimed nodes in $DL(p_i)$ is at most $N \cdot (k + l_{max} + \alpha + 1)$, where N is the number of threads in the system, k is the number of hazard pointers per thread, l_{max} is the maximum number of links a node can contain and α is the maximum number of links in live nodes that may transiently point to a deleted node.*[6]

Proof: The only operation that increases the size of $DL(p_i)$ is *DeleteNode* and when $|DL(p_i)|$ reaches THRESHOLD_1 it runs *CleanUpAll* before attempting to reclaim nodes.

First consider the case where there are no concurrent *DeleteNode* operations by other threads. Then, after *CleanUpAll*, there cannot be any deleted nodes that point to nodes in $DL(p_i)$ left. So, what may prevent p_i from reclaiming one particular node in $DL(p_i)$? The node might have: (i) a hazard pointer pointing to it, or (ii) there might be some live nodes still pointing to it. The number of links in live nodes, α, that might point to a deleted node depends on the application data-structure using the memory manager. We require that each application operation that deletes a node must also remove all references to that node from the live nodes of the data-structure before it is completed, which ensures that there at all times are at most $N \cdot \alpha$ links in live nodes that point to deleted nodes. So, in the absence of concurrent *DeleteNode* operations the maximum number of nodes in $DL(p_i)$ that a *Scan* is unable to reclaim is $N \cdot (k + \alpha)$.

In the case where there are concurrent *DeleteNode* operations three more things of interest may occur: (i) Additional nodes that might hold pointers to the

[6]Note that the numbers l_{max} and α depend only on the application.

nodes in $DL(p_i)$ might be deleted after the start of *CleanUpAll* and prevent *Scan* from reclaiming any node. However, in that case p_i is free to retry *CleanUpAll* and *Scan* again since some concurrent operation has made progress. (ii) Some concurrent *DeleteNode* operation may get delayed or crash, either between line DN2 and DN5 or between SC21 and SC22 in its call to *Scan* which will "hide" one deleted node that might contain links that point to nodes in $DL(p_i)$ from p_i's *CleanUpAll*. In this way each other thread can prevent p_i from reclaiming up to l_{max} nodes in $DL(p_i)$. (iii) Finally, concurrent *CleanUpAll* operations might prevent p_i from reclaiming reclaimable nodes by claiming them for performing *CleanUpNode* operations on them. However, if such a node is encountered p_i's *Scan* will use *TerminateNode* to set the links of the node to NULL and set the DL_Done flag for the node, which prevents future *CleanUpAll* operations from preventing the node from being reclaimed. If p_i needs to retry the *Scan*, it can only be prevented from reclaiming at most N of the reclaimable nodes it failed to reclaim due to concurrent *CleanUpAll*s during the previous *Scan*.

So, the maximum number of nodes in $DL(p_i)$ that p_i cannot reclaim is less than $N(k + l_{max} + \alpha + 1)$. □

Corollary 5.5.1 *The cleanup threshold, THRESHOLD_1, used by the algorithm should be set to* $N(k + l_{max} + \alpha + 1)$.

Corollary 5.5.2 *The number of deleted but not yet reclaimed nodes in the system is bounded from above by*

$$N^2 \cdot (k + l_{max} + \alpha + 1),$$

where N is the number of threads in the system, k is the number of hazard pointers per thread, l_{max} is the maximum number of links a node can contain and α is the maximum number of links in live nodes that may transiently point to a deleted node.

5.5.3 Linearizability

Lemma 5.5.6 *The DeRefLink ($DRL(l_1) = n_1$) operation is atomic.*

Proof: A *DeRefLink*(DRL) operation has direct interactions with *CompareAndSwapRef*(CASR) that targets the same link and the memory reclamation in *Scan*. A CASR operation takes effect before the DRL operation iff the CAS instruction at line C1 is executed before `*link` is read at line D5 in DRL.

A *Scan* that reads the hazard pointer set by DRL at line D4 after it was set will not free the node dereferenced by DRL (the test at line SC19 in *Scan* prevents this. If a concurrent *Scan* read the hazard pointer in question after it was set by DRL then it will not free the node. If *Scan* read the hazard pointer in question before it was set by DRL then *Scan* will detect that the reference count of the node is non-zero or has been non-zero during the execution of the *Scan* operation. □

Lemma 5.5.7 *The ReleaseRef ($RR(n_1)$) operation is atomic.*

Proof: The operation *ReleaseRef* takes effect at line R2 when the hazard pointer to the node is set to NULL. □

Lemma 5.5.8 *The NewNode ($NN() = n_1$) operation is atomic if the memory allocator used to manage the pool of free memory itself is linearizable.*

Proof: The operation *NewNode* takes effect when the memory for the new node is removed from the pool of free memory. For a linearizable memory allocator this will take place at a well-defined time instant. □

Lemma 5.5.9 *The DeleteNode ($DN(n_1)$) operation is atomic.*

Proof: The operation *DeleteNode* takes effect when the node is marked as deleted at line DN2. □

Lemma 5.5.10 *The CompareAndSwapRef ($CASR(l_1, n_1, n_2)$) operation is atomic.*

Proof: The *CompareAndSwapRef* ($CASR(n_1)$) operation has direct interactions with other *CompareAndSwapRef* (CASR) and *DeRefLink* operations that targets the same link and with the memory reclamation in *Scan*.

A CASR operation takes effect when the CAS instruction at line C1 is executed. □

Lemma 5.5.11 *The reclamation of a deleted node n by Scan is atomic.*

Proof: *Scan* is said to reclaim a node n when it is returned to the pool of free memory, which takes place at line SC23. The node is safe to reclaim because the tests at line SC19 guarantees that (i) *Scan* found no hazard pointers pointing to the node and, (ii) the reference count of the node has been consistently 0 since before the hazard pointers were scanned. That (ii) holds is ensured by the n.mm_trace bit, which detects if a node, n, had $Ref(n) = 0$ when *Scan* executed line SC4, but a pointer to it was later stored in a shared link variable so that there were no hazard pointer to it when *Scan* read them at line SC9 to SC13. Then, this invocation of *Scan* will not attempt to reclaim the node even if the shared link to the node is removed again, since the n.mm_trace was cleared when the pointer to the node was stored in a shared variable (either by a *CompareAndSwapRef* or a *StoreRef*).

The test at line SC21 guarantees that there are no other threads that might access the node as part of the *CleanUpAll* operation. A *CleanUpAll* operation verifies that the node is still there at line CA6 after increasing the claim counter at line CA5 before attempting to access the node. If *Scan* reads a claim count of 0 at line SC21 then there are no concurrent *CleanUpAll* operations that might access the node, because they will read NULL at line CA6. □

Theorem 5.5.4 *The algorithm correctly implements a linearizable garbage collector.*

Proof: This follows from Lemmas 5.5.6, 5.5.7, 5.5.8, 5.5.9 and 5.5.10. □

5.5.4 Proof of the Lock-free Property

Lemma 5.5.12 *With respect to the retries caused by synchronization, one operation will always do progress regardless of the actions by the other concurrent operations.*

Proof: We now examine the possible execution paths of our implementation. The operations *ReleaseRef*, *NewNode* and *CompareAndSwapRef* do not contain any loops and will thus always do progress regardless of the actions by the other concurrent operations. In the remaining concurrent operations there are several potentially unbounded loops that can delay the termination of the operations. We call these loops retry-loops. If we omit the conditions that are because of the operations semantics (i.e. searching for the correct criteria etc.), the loop retries when sub-operations detect that a shared variable has changed value. This is detected either by a subsequent read sub-operation or by a failed CAS. These shared variables are only changed concurrently by other CAS sub-operations. The read operation in line D5 will possibly fail because of a successful CAS operation in lines C1, TN7 or CN5. Likewise, the CAS operations in lines C1, TN7 or CN5 will possibly fail if one of the other CAS operations has been successful. According to the definition of CAS, for any number of concurrent CAS sub-operations, exactly one will succeed. This means that for any subsequent retry, there must be one CAS that succeeded. As this succeeding CAS will cause its retry loop to exit, and our implementation does not contain any cyclic dependencies between retry-loops that exit with CAS, this means that the corresponding *DeRefLink* operation or *TerminateNode* suboperation will progress.

In the operation *DeleteNode* there are calls to three suboperations, *CleanUpLocal*, *Scan* and *CleanUpAll* which contains loops, inside an unbounded loop. The loop in the suboperation *CleanUpNode*, used by *CleanUpLocal* and *CleanUpAll*, is bounded in the absence of concurrent *DeleteNode* operations because of Assumption 5.4.1. If there are concurrent *DeleteNode* operations *CleanUpNode* only their progress, i.e. that they set the mm_del bit on additional nodes, might force the loop in *CleanUpNode* to continue. So *CleanUpNode* is lock-free. The loops in *CleanUpAll* are all bounded and the loop in *CleanUpLocal* and in *Scan* are bounded since the size of the DL_Nodes list is bounded by Theorem 5.5.3. The loop in *DeleteNode* is also bounded by the bound in Theorem 5.5.3.

Consequently, independent of any number of concurrent operations, one operation will always progress. □

Theorem 5.5.5 *The algorithm implements a lock-free garbage collector.*

Proof: The theorem follows from Lemma 5.5.12. □

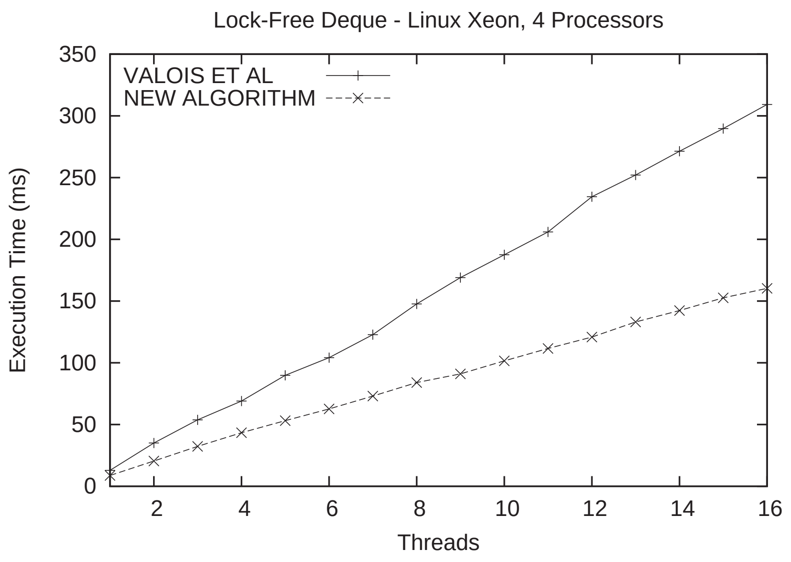
5.6 Experimental evaluation

We have performed experiments in our effort to estimate the average overhead of using the new lock-free memory management algorithm in comparison to previous lock-free memory management algorithms supporting reference counting. For this purpose we have chosen the lock-free algorithm of a deque (double-ended queue) data structure by Sundell and Tsigas [ST04, Sun04a]. As presented, the implementation of this algorithm uses the lock-free memory management with reference counting by Valois et al. [Val95a, MS95]. In order to fit better with the new memory management algorithm, the recursion calls in the deque algorithm were unrolled.

In our experiments, each concurrent thread performed 10000 randomly chosen sequential operations on a shared deque, with an equal distribution among the *PushRight*, *PushLeft*, *PopRight* and *PopLeft* operations. Each experiment was repeated 50 times, and an average execution time for each experiment was estimated. Exactly the same sequence of operations was performed for all different implementations compared.

The experiments were performed using different number of threads, varying from 1 to 16 with increasing steps. In our experiments we compare two implementations of the lock-free deque; (i) using the lock-free memory management by Valois et al., and (ii) using the new lock-free memory management (including support for dynamic number of threads) with 6 hazard pointers per thread. These are the only memory management algorithms which (i) satisfy the demands of the lock-free deque algorithm (as well as other common lock-free algorithms that need to traverse through nodes which may concurrently be deleted, such as the Queue algorithm used for the example in Algorithm 5.8) and (ii) work with available atomic primitives. Both implementations use a shared fixed-size memory pool (i.e. freelist) for memory allocation and freeing. Two different platforms were used, with varying number of processors and level of shared memory distribution. Firstly, we performed our experiments on a 4-processor Xeon PC running Linux. In order to evaluate our algorithm with higher concurrency we also used a 8-processor SGI Origin 2000 system running Irix 6.5. A clean-cache operation was performed just before each subexperiment. All implementations are written in C and compiled with the highest optimization level. The atomic primitives are written in assembly. The results from the experiments are shown in Figure 5.3. The average execution time is drawn as a function of the number of threads.

Our results show that the new lock-free memory management algorithm outperforms the corresponding algorithm by Valois et al. for any number of threads. The advantages of using the new algorithm appear to be even more significant for systems with non-uniform memory architecture.



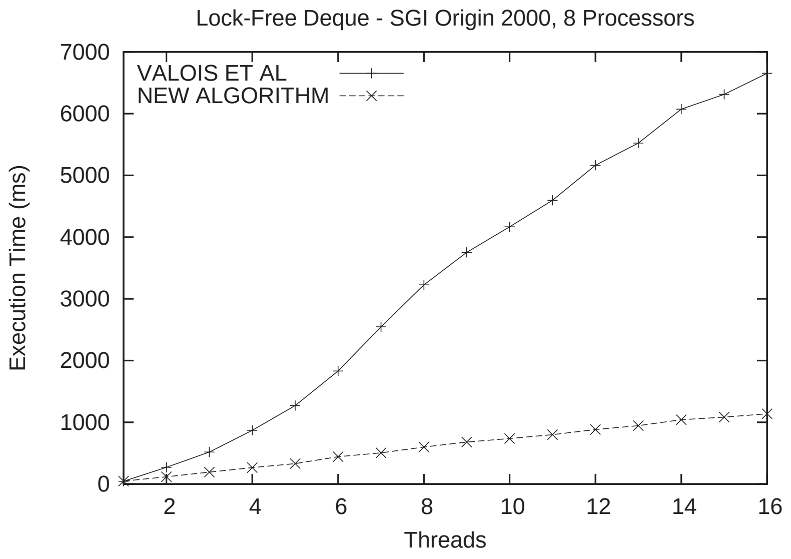


Figure 5.3: Experiment with lock-free deques and various memory reclamation algorithms.

5.7 Conclusions

To the best of our knowledge, we have presented the first lock-free algorithm for lock-free garbage collection based on reference counting that has all the following features: (i) guarantees the safety of local as well as global references, (ii) provides an upper bound of deleted but not yet reclaimed nodes, (iii) is compatible with arbitrary memory allocation schemes, and iv) uses atomic primitives which are available in modern architectures.

Experimental results indicate that our new lock-free garbage collection can significantly improve the performance and reliability of implementations of lock-free dynamic data structures that require the safety of global references. We believe that our implementation is of highly practical interest for multi-processor applications. We are currently incorporating it into the NOBLE [ST02] library.

Chapter 6

NBmalloc: Allocating memory in a lock-free manner[1]

Anders Gidenstam Marina Papatriantafilou

Philippas Tsigas

Abstract

Efficient, scalable memory allocation for multithreaded applications on multiprocessors is a significant goal of recent research. In parallel, and not only recently, in the distributed computing literature it has been emphasized that lock-based concurrency-control may limit the parallelism in multiprocessor systems. Thus, system services that employ such methods can be a hinder in realizing the potential in these systems. A natural research question is the plausibility and the impact of lock-free concurrency control in key services for multiprocessors, such as in the memory allocation service, which is the theme of this work. We show the design and implementation of NBmalloc, a lock-free memory allocator designed to enhance the parallelism in the system. The architecture of NBmalloc is inspired by Hoard, a successful concurrent memory allocator, with modular, scalable design that preserves scalability and helps avoiding false-sharing and heap blowup. Within our effort on designing appropriate lock-free algorithms for this system, we propose and show a lock-free implementation of a new data structure, flat-set, supporting conventional "internal" operations as well as "inter-object" operations, for moving items between flat-sets. The design of NBmalloc also involved a series of other algorithmic problems, which are discussed in the paper,

[1]This work was supported by computational resources provided by the Swedish National Supercomputer Centre (NSC). This chapter contains an extended version of the paper that appeared in the proceedings of ESA 2005, Palma, Spain, 3-7 October, 2005.

along with the solutions designed and adopted in this work. Further, we present the implementation of NBmalloc in a set of multiprocessor systems and the study of its behaviour. The results show that the good properties of Hoard w.r.t. false-sharing and heap-blowup are preserved, while the scalability properties are enhanced even further with the help of lock-free synchronization.

6.1 Introduction

Non-blocking implementations of shared data objects are an alternative to the traditional solution for maintaining the consistency of a shared data object (i.e. for ensuring *linearizability* [HW90]) by enforcing mutual exclusion. Non-blocking synchronization allows multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion [Bar93, GC96, Her91, Rin99, HPT02]. Regarding efficiency and scalability, it is known that the use of locks in synchronization is a limiting factor, especially in multiprocessor systems, since it reduces parallelism. Thus, constructions which guarantee that concurrent access to shared objects is free from locking are of particular interest, as they help to increase the amount of parallelism and to provide fault-tolerance. This type of synchronization is called lock-/wait-free, non-blocking or optimistic synchronization [Bar93, GC96, Her91, Rin99]. Non-blocking algorithms have been shown to have significant impact in applications [TZ01a, TZ02], and there is also a library, NOBLE [ST02], containing many implementations of non-blocking data structures. Besides, the potential of this type of synchronization in the performance of system-services and data structures has also been pointed out in [DG02, GC96, MP91].

The present article studies the impact of lock-free synchronization on the memory-allocation system service. Some form of dynamic memory management is used in most computer programs for multiprogrammed computers. It comes in a variety of flavors, from the traditional manual general purpose allocate/free type memory allocator to advanced automatic garbage collectors. Here we focus on conventional, general-purpose memory allocators (such as the "libc" malloc) where the application can request (allocate) arbitrarily-sized blocks of memory and free them in any order. Essentially a memory allocator is an online algorithm that manages a pool of memory (heap), e.g. a contiguous range of addresses or a set of such ranges, keeping track of which parts of that memory are currently given to the application and which parts are unused and can be used to meet future allocation requests from the application. The memory allocator is not allowed to move or otherwise disturb memory blocks that are currently owned by the application.

An important optimization goal of a good allocator is to minimize *fragmentation*, i.e. minimize the amount of free memory that cannot be used (allocated) by the application. Fragmentation is distinguished in internal and external. *Internal fragmentation* results in cases that free memory is wasted when the application is given a larger memory block than it requested. *External fragmentation* results in cases when free memory has been split into non-contiguous blocks that are too small to be useful to satisfy the requests from the application.

Moreover, multi-threaded programs add some more complications to the memory allocator. Obviously some kind of synchronization has to be added to protect the heap during concurrent requests. There are also other issues which have significant impact on application performance when the application is run on a multiprocessor [Ber02]. Namely, a good concurrent memory allocator should (i) avoid *false sharing*, which is when different parts of the same cache-line end up being used by threads running on different processors; (ii) avoid *heap blowup*, which is an overconsumption of memory that may occur if the memory allocator fails to make memory deallocated by threads running on one processor available to threads running on other processors; (iii) ensure *efficiency* and *scalability*, i.e. the concurrent memory allocator should be as fast as a good sequential one when executed on a single processor and its performance should scale with the load in the system.

The Hoard [BMBW00] concurrent memory allocator is designed to meet the above goals. The allocation is done on the basis of per-processor heaps, which avoids false sharing and reduces the synchronization overhead in many cases, improving both performance and scalability. Memory requests are mapped to the closest matching size in a fixed set of size-classes, which bounds internal fragmentation. The heaps are sets of superblocks, where each superblock handles blocks of one size class, which helps in coping with external fragmentation. To avoid heap blowup, freed blocks are returned to the heap they were allocated from and empty superblocks may be reused in other heaps.

The present paper proposes a new memory allocator based on lock-free, fine-grained synchronization, to enhance parallelism, fault-tolerance and scalability. The architecture of our allocation system is inspired by Hoard, due to its well-justified design decisions, which we roughly outlined above. Further, in the process of designing appropriate data structures and lock-free synchronization algorithms for our system, we introduced a new data structure, which we call *flat-set*. The operations supported by the sets include operations of common sets, as well as "inter-object" operations, for moving an item from one flat-set to another in a lock-free manner. The lock-free algorithms we introduce make use of standard synchronization primitives provided by multiprocessor systems, namely single-word *Compare-And-Swap*, or its equivalent *Load-Linked/Store-Conditional*. The design of the proposed memory allocator also involved a set of other interesting algorithmic issues, which are discussed and analyzed here, along with the solutions designed and adopted in NBmalloc.

We have implemented and evaluated the allocator proposed here on common multiprocessor platforms, namely an UMA Sun Fire 880 running Solaris 9, a NUMA Origin 3800 running IRIX 6.5 and an Intel Xeon PC running Linux 2.9.6. We compare our allocator with the standard "libc" allocator of each platform and with Hoard (on the Sun system, where we had the original Hoard allocator available) using standard benchmark applications to test the efficiency, scalability, cache behaviour and memory consumption behaviour. The results show that our system preserves the good properties of Hoard, while it offers a higher scalability potential, as justified by its lock-free nature.

Recently Michael presented a lock-free allocator [Mic04c] that, like our contribution, is loosely based on the Hoard architecture and uses *Compare-And-Swap*. Despite both having started from the Hoard architecture, we have used two different approaches to achieve lock-freedom. Another allocator that tries to reduce the use of locks is the LFMalloc [DG02]. To be able to relate these contributions with the one presented here some more detail is needed and for this reason we describe this relation in section 6.8.

Also of relevance to lock-free memory allocators are algorithms for lock-free memory management and garbage collection. Such schemes need to be used in lock-free dynamic data-structures to provide safe reclamation of dynamically allocated shared memory blocks, i.e. to make sure that a deleted memory block is not reused until it is certain that it cannot be accessed by any concurrent or future operation anymore. Some schemes focus on the safety of local references to objects only, such as the efficient hazard pointer algorithm by Michael [Mic02b, Mic04a] and the algorithm by Herlihy et al. [HLM02], but these cannot be used with all data-structures. Other schemes based on reference counting can guarantee the safety of local as well as global references to objects. In this category we have the work of Valois et al. [Val95a, MS95], Detlefs et al. [DMMS01], Herlihy et al. [HLMM02] and Gidenstam et al. [GPST05].

Earlier related work in similar direction is the work on non-blocking operating systems by Massalin and Pu [MP91, Mas92] and Greenwald and Cheriton [GC96, Gre99]. The respective algorithms, however, made extensive use of the *2-Word-Compare-And-Swap* (*2CAS*) primitive, which can update two arbitrary memory locations in one atomic step, while this primitive is not available in current systems and is expensive to simulate in software.

The next section provides some more technical background on concurrent memory allocation and lock- and wait-free synchronization. (Throughout the paper, we use the terms non-blocking and lock-free interchangeably). Earlier and recent related work is discussed in more detail in section 6.8, after the presentation of the more technical background and of our method and implementation, as some insight in these is needed to relate the contributions. Sections 6.3, 6.4 and 6.5 describe our memory allocator, including the lock-free implementation of the flat-sets data structure designed for this purpose. Section 6.7 describes details on the implementation done for the experimental evaluation of the system, including the platforms on which it was implemented and benchmark information. The results of our experimental evaluation are also presented there. We conclude with a section discussing the achieved results and describing some future work.

6.2 Background

6.2.1 Concurrent Memory Allocators

As mentioned in the introduction, there is a set of extra complications introduced in the memory allocation issues from the perspective of multiprocessor systems and multithreaded applications. We discuss them here, giving also some more technical details.

The first of these is *false sharing* which is when different parts of the same cache-line end up being used by threads running on different processors. This will put a potentially large and unnecessary load on the cache-coherence mechanism. This can never be avoided completely since application threads may pass allocated memory between themselves but a memory allocator can *actively induce* false sharing by directly satisfying memory requests from different processors with memory from the same cache-line.

The second issue is *heap blowup* which is a potentially unbounded overconsumption of memory that might occur if the memory allocator fails to make memory freed by one processor available to others, e.g. as the result of a coarse policy for avoiding false sharing. A typical application that triggers this is an application that has producer and consumer threads, where the producer allocates memory and passes it to the consumer which in turn frees the memory. If the memory freed by the consumers is never made available to the producers then the resulting heap blowup can be unbounded.

The last issues are *scalability* and *speed*. For a memory allocator to be scalable, its performance has to scale well with the number of processors and the load in the system. In terms of speed, the concurrent memory allocator should be about as fast as a good sequential one in order to ensure good performance even when a multithreaded program is executed on a single processor.

Below follows a brief overview of some concurrent memory allocator designs based on the taxonomy presented in [Ber02]:

Single serial heap. A normal sequential memory allocator is protected by a global lock. This type of memory allocator scales poorly on multiprocessors, since only one thread can access the heap at a time. Moreover, in some instances it also performs bad on a single processor since the thread holding the lock protecting the heap might be delayed inside the memory allocator code, by, for example, a preemption or page fault. It is also prone to induce false sharing, but does not suffer from heap blowup and should be fast on a single processor in most cases (the potential for threads being delayed while holding the lock, as mentioned above, is an exception). Examples of this kind of allocator is the standard memory allocators on Solaris, Irix and Windows 2000.

Concurrent single heap. A memory allocator that uses a single heap implemented using fine-grained synchronization so that several threads may operate on it concurrently. This kind of memory allocator avoids heap blowup and has the potential to be scalable. However, it is prone to induce false sharing and is not easy to make fast due to the potential for very high levels of contention and large synchronization overhead.

Pure private heaps. The memory allocator maintains a separate heap for each processor, where threads running on that processor allocate memory. When a thread frees some memory it is added to the heap of the processor running that thread. This causes pure private heaps to suffer from potentially unbounded heap blowup. On the other hand this type of memory allocator is scalable, fast and less prone to induce false sharing (if designed correctly). Examples: STL allocator [SGI03], Cilk [BL94].

Private heaps with ownership. These memory allocators are similar to pure private heaps but freed memory is always returned to the heap it was allocated from. This bounds the worst case heap blowup to $O(P)$, where P is the number of processors. Examples: MTmalloc (Solaris), Ptmalloc (glibc) [Glo03].

Private heaps with thresholds. Private heaps with thresholds use a global heap in addition to the per-processor heaps in order to avoid the $O(P)$ heap blowup that private heaps with ownership suffer from. This is done by transferring part of the free memory in a per-processor heap to the global heap when the amount of free memory in the per-processor heap becomes too large. The memory in the global heap can be transferred to other per-processor heaps for reuse as needed. Examples: Hoard[BMBW00], LFmalloc[DG02](mostly) and NBMALLOC, the memory allocator presented in this paper.

Thread-local allocations. Since most of the dynamic memory requests in many applications concern memory that will only be used by one thread, it might be advantageous to distinguish between these thread-local allocations and allocations of memory that is to be shared between threads. In particular the thread-local memory could be handled without any synchronization. The distinction between thread-local allocations and shared memory allocations could be made explicitly by the programmer (using an extension of the traditional malloc interface) or automatically by compile time analysis, as for example in [Ste00].

6.2.2 Non-blocking Synchronization

The most commonly required consistency guarantee for shared data objects is *atomicity*, a.k.a. *linearizability* [HW90]. A shared object (its implementation) is *atomic* or *linearizable* if it guarantees that even when operations overlap in time, each of them appears to take effect in an atomic time instant that lies in its respective time duration, in a way that the effect of each operation is in agreement with the object's sequential specification. The latter means that if we speak of e.g. read/write objects, the value returned by each read equals the value written by the most recent write according to the sequence of "shrunk" operations in the time axis.

Compared to the traditional solution for maintaining the consistency of a shared data object (i.e. for ensuring linearizability) by enforcing mutual exclusion, non-blocking implementations of shared data objects are an alternative approach. Non-blocking mechanisms allow multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion [Bar93, GC96, Her91, Rin99]. Non-blocking (a.k.a. optimistic) synchronization can be lock-free or wait-free. *Lock-free* algorithms guarantee that regardless of the contention caused by concurrent operations and the interleaving of their steps, at each point in time there is at least one operation that is able to make progress. However, the progress of other operations might cause one specific operation to take unbounded time to finish. In a *wait-free* algorithm, every operation is guaranteed to finish in a bounded number of its own steps, regardless of the actions of concurrent operations.

Lock-free algorithms typically involve fine-grained synchronization, with attempts to commit updates using certain synchronization primitives (like *CAS*, *LL*/*SC*; see below) or to verify non-interfered access to small amounts of shared data. If such an attempt fails, then that process/thread needs to *retry*. The above may happen due to preemption or due to interleaving by threads running in parallel. *Helping* is a method proposed to alleviate the problem: an operation that detects that it has preempted or has otherwise interleaved steps with another operation, helps the latter operation to progress before proceeding with its own steps, to reduce the fail-retry overhead.

It has been shown that there exist *universal synchronization primitives*, that can implement, in a wait-free manner –hence, also in a lock-free manner–, any object with a sequential specification using those primitives [Her91]. There exist also *universal constructions* that can implement any object using these universal synchronization primitives [Her91]. These constructions also use helping. However, being generic, they are are introduced for showing feasibility and not performance.

Some of the aforementioned universal primitives are common instructions available in most processors, e.g. the *Compare-And-Swap* instruction (also denoted *CAS*), which atomically executes the steps described in Fig. 6.1. If *CAS* cannot assign the new value to the location for which it is invoked, we say that it *fails*, otherwise, it *succeeds*. *CAS* is available in e.g. SPARC processors. Another primitive that is equivalent with *CAS* in synchronization power is the *Load-Linked/Store-Conditional* (also denoted *LL*/*SC*) pair of instructions, available in, e.g. MIPS processors. *LL*/*SC* is used as follows: (i) *LL* loads a word from memory. (ii) A short sequence of instructions may modify the value read. (iii) *SC* stores the new value into the memory word, unless the word has been modified by other process(es) after *LL* was invoked. In the latter case the *SC* *fails*, otherwise the *SC* *succeeds*. Another useful primitive is *Fetch-And-Add* (also denoted *FAA*), described in Fig. 6.1. *FAA* can be simulated in software using *CAS* or *LL*/*SC* when it is not available in hardware.

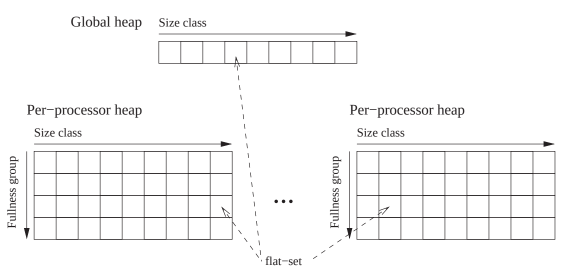
An issue that sometimes arises in connection with the use of *CAS*, is the so-called *ABA problem*. It can happen if a thread reads a value A from a shared variable, and then invokes a *CAS* operation to try to modify it. The *CAS* will (undesirably) succeed if between the read and the *CAS* other threads have changed the value of the shared variable from A to B and back to A. A common way to cope with the problem is to use version numbers of b bits as part of the shared variables [Val95b]. Then, for the same problem to occur, it would be

```
atomic CAS(mem : pointer to integer;
        new, old : integer) return integer
begin
  tmp := *mem;
  if tmp == old then
    *mem := new; /* CAS succeeded */
  return tmp;
end CAS;

atomic CAS(mem : pointer to integer;
        new, old : integer) return boolean
begin
  tmp := *mem;
  if tmp == old then
    *mem := new; /* CAS succeeded */
  return tmp == old;
end CAS;

atomic FAA(mem : pointer to integer;
        increment : integer) return integer
begin
  tmp := *mem;
  *mem := tmp + increment;
  return tmp;
end FAA;
```

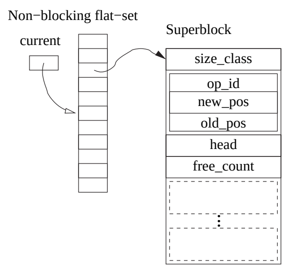
Figure 6.1: The synchronization primitives Compare-And-Swap (denoted *CAS*) and Fetch-And-Add (denoted *FAA*).

necessary to have a sequence of 2^b successful *CAS* operations between the read A and its corresponding *CAS*, with the last such operation storing value A to the shared variable. By choosing b appropriately, this is made extremely unlikely. An alternative method to cope with the ABA problem is to introduce special NULL values, which was proposed and used in a lock-free queue implementation in [TZ01b]. An appropriate garbage-reclamation mechanism, such as [Mic02b], can also solve the problem. This method can also be used to implement lock-free and linearizable *Load-Linked/Store-Conditional* operations for arbitrarily large objects in software [Mic04b].

There is a plethora of research articles that focus on wait-free and lock-free synchronization (for a few examples cf. [Bar93, Har01, Her91, Her93, HPT02, Mic02b, Moi97, Rin99, PT95, PT97, ST03, TZ01a, Val94, Val95b]). Non-blocking algorithms have been shown to also have significant impact in applications [TZ01a, TZ02], and recently NOBLE, which is a library of shared data structures and includes blocking and non-blocking implementations, has been presented [ST02].



(a) The organization of the global and per-processor heaps.



(b) A non-blocking flat-set and a superblock.

Figure 6.2: The architecture of the memory allocator.

6.3 The NBmalloc: Architecture

The architecture of our lock-free memory allocator is inspired by Hoard [BMBW00], which is a well-known and practical concurrent memory allocator for multiprocessors.

The memory allocator provides allocatable memory of a fixed set of sizes, called *size-classes*. The size of memory requests from the application are rounded upwards to the closest size-class. To reduce false-sharing and contention, the memory allocator distributes the memory into *per-processor heaps*. The managed memory is handled internally in units called *superblocks*. Each superblock contains allocatable blocks of one size-class. Initially all superblocks belong to the *global heap*. During an execution superblocks are moved to per-processor heaps as needed. When a superblock in a per-processor heap becomes almost empty (i.e. few of its blocks are allocated) it is moved back to the global heap. The superblocks in a per-processor heap are stored and handled separately, based on their size-class. Within each size-class the superblocks are kept sorted into bins based on *fullness*(cf. Figure 6.2(a)). As the fullness of a particular superblock changes it is moved between the groups. A memory request (*malloc* call) first searches for a superblock with a free block among the superblocks in the "almost full" fullness-group of the requested size-class in the appropriate per-processor heap. If no suitable superblock is found there, it will proceed to search in the lower fullness-groups, and, if that, too, is unsuccessful, it will request a new superblock from the global heap. Searching the almost full superblocks first reduces external fragmentation. When freed (by a call to *free*) an allocated block is returned to the superblock it was allocated from and, if the new fullness requires so, the superblock is moved to another fullness-group.

6.4 Managing superblocks: The bounded non-blocking flat-sets

Since the number of superblocks in each fullness-group varies over time, a suitable collection-type data structure is needed to implement a fullness-group. Hoard, which uses mutual-exclusion on the level of per-processor heaps, uses linked-lists of superblocks for this purpose, but this issue becomes very different in a lock-free allocator. While there exist several lock-free linked-list implementations, e.g. [Har01, Mic02a, Val95b], we cannot apply those here, because not only do we want the operations on the list to be lock-free, but we also need to be able to move a superblock from one set to another without making it inaccessible to other threads during the move. To address this, we propose a new data structure we call a *bounded non-blocking flat-set*, supporting conventional "internal" operations (*Get_Any* and *Insert* item) as well as "inter-object" operations, for moving an item from one flat-set to another.

To support "inter"-flat-set operations it is crucial to be able to move superblocks from one set to another in a lock-free fashion. The requirements that make this difficult are:

1. the superblock should be reachable for other threads even while it is being moved between flat-sets, i.e. a non-atomic *first-remove-then-insert* sequence is not acceptable;
2. the number of shared references to the superblock should be the same after a move (or set of concurrent move operations) finish as it was before the move (or the set of moves) started.

Below we present the operations of the lock-free flat-set data structure and the lock-free algorithm, *Move*, which implements the "inter-object" operation for moving a reference to a superblock from one shared variable (pointer) to another, while satisfying the above requirements.

6.4.1 Operations on bounded non-blocking flat-sets

A bounded non-blocking flat-set provides the following operations:

1. *Get_Any*, which returns any item in the flat-set; and
2. *Insert*, which inserts an item into the flat-set.

Note that an important property that needs to be satisfied is that an item can only reside inside one flat-set at the time; when an item is inserted into a flat-set it is also removed from its old location.

The flat-set data structure consists of an array of M shared locations set.set[i], each capable of holding a reference to a superblock, and a shared index variable set.current. The data structure and operations are shown in Algorithm 6.2 and are explained in the following paragraphs.

To speed up flat-set operations there is an index variable set.current that is used as a *marker*. It contains a bit used as an *empty flag* for the flat-set and a

Algorithm 6.1 The flat-set and superblock data structures in NBmalloc.

```
type flat-set_t is record
    size : constant integer
    current : flat-set_info
    set[size] : array of superblock_ref_t

type flat-set_info is atomic record
    /* fits in one machine word */
    index : integer_15
    empty : boolean
    version : integer_16

type superblock_ref_t is atomic record
    /* fits in one machine word */
    ptr : integer_16
    version : integer_16

/* superblock_ref_t utility functions. */
function pointer(ref : superblock_ref_t) return pointer to superblock_t
function version(ref : superblock_ref_t) return integer_16
function make_sb_ref(sb : pointer to superblock_t, op_id : integer_16)
    return superblock_ref_t

type superblock_t is record
    mv_info : move_info_t
    freelist_head : block_ref_t
    free_block_cnt : integer
    ...

type move_info_t is record
    op_id : integer_16
    new_pos : pointer to superblock_ref_t
    cur_pos : pointer to superblock_ref_t

type block_ref_t is atomic record
    /* fits in one machine word */
    offset : integer_16
    version : integer_16

function pointer(ref : block_ref_t) return pointer to block_t
function version(ref : block_ref_t) return integer_16
function make_ref(sb : pointer to block_t, version : integer_16) return block_ref_t

type block_t is record
    /* block header */
    owner : pointer to superblock_t
    next : pointer to block_t
```

index field that is used as the starting point for searches, both for items and for free slots.

- The *empty flag* is set by a *Get_Any* operation that discovers that the flat-set is empty, so that subsequent *Get_Any* operations know this; the *Insert* operation and successful *Get_Any* operations clear the flag. Due to asynchrony, the *empty flag* might not always be set when (i.e. at each time instant that) the flat-set is empty, as superblocks can be moved away from the flat-set at any time, but it is always cleared when the flat-set is nonempty.

- The *Insert* operation scans the array set.set[i] forward from the position marked by set.current until it finds an empty slot. It will then attempt to move the superblock reference to be inserted into this slot using the *Move* operation described in detail in the next subsection.

- The *Get_Any* operation first reads set.current to check the *empty flag*. If the *empty flag* is set, *Get_Any* returns immediately, otherwise it starts to scan the array set.set[i] backwards from the position marked by set.current, until it finds a location that contains a superblock reference. Note that *Get_Any* uses the operation *Strict_Dereference* (c.f Algorithm 6.3) to read a superblock reference from set.set[i]. *Strict_Dereference* will also help any ongoing *Move* operation concerning that superblock to finish before returning the reference —or null if the superblock was moved away (cf. next subsection for an explanation of such helping actions). This takes care that the operations are properly linearizable.

  If a *Get_Any* operation has scanned the whole set.set[i] array without finding a reference it will try to set the *empty flag* for the flat-set. This is done at line G13 using *CAS* and will succeed if and only if set.current has not been changed since it was read at line G2. This indicates that the flat-set is empty so *Get_Any* sets the empty flag and returns failure. If, on the other hand, set.current has changed between line G2 and G13, then either an *Insert* is in progress or has finished during the scan (line I6 and I9) or some other *Get_Any* has successfully found a superblock during this time (line G10), so *Get_Any* should redo the scan. To facilitate moving of superblocks between flat-sets via *Insert* *Get_Any* returns both a superblock reference and a reference to the shared location containing it.

6.4.2 How to move a shared reference: Moving items between flat-sets

The algorithm supporting the operation *Move* moves a superblock reference sb from a shared location from to a shared location to. The target location (i.e. to) is known via the Insert operation. The algorithm requires the superblock to contain an auxiliary variable mv_info with the fields op_id, new_pos and cur_pos and all superblock references to have a version field (cf. Fig 6.1). Optionally, the mv_info structure could also contain additional information. This is the case in NBMALLOC where two additional fields, cur_owner and new_owner, are used to keep track of which flat-set the superblock currently is located in.

A *Move* operation may *succeed* by returning SB_MOVED_OK or *fail* (abort) by returning SB_MOVED (if the block was moved away by another overlapping *Move*) or SB_NOT_MOVED (if the to location is occupied). It succeeds if it is completed successfully by the thread that initiated it or by a helping thread. It fails if it detects that another overlapping *Move* of the same superblock has already removed the reference to the superblock from the from location or if it detects that a preceding or overlapping *Move* of another superblock has occupied the to location.

To ensure the lock-free property, the move operation is divided into four steps of a number of atomic suboperations. The first step (1), to *register* the *Move* operation, is done in the *Move(sb, from, to)* function in Algorithm 6.3, while the other three steps, (2) to update location to, (3) to clear location from and (4) to update sb.mv_info, are done in the *Move_Help* function in Algorithm 6.3.

A *Move* operation that encounters an unfinished *Move* of the same superblock will *help* the old operation to finish before it attempts to perform its own move. The helping procedure is performed by *Move_Help* and is therefore identical to steps 2 - 4 of the move operation as described below. Since all information required to finish an ongoing operation is stored in the mv_info, any thread that encounters the superblock can continue the operation and finish it.

1. A *Move*(sb, from, to) is initiated by atomically *registering* the operation. This is done by first reading the current value of sb.mv_info using *Load_Linked* (line M2 in *Move*); then, if and only if the read value of sb.mv_info.from was null (line M6) –which indicates that there are no ongoing move of this superblock– sb.mv_info is set to (version(sb), to, from) using *Store_Conditional* (line M8). If the read value of sb.mv_info.op_id was not null, then there is an ongoing *Move* operation that needs to be helped before this one can proceed (line M11). If the reference to the superblock disappears from location from before this move has been registered, this move operation is abandoned and returns SB_MOVED (lines M4 and M5). When a *Move* operation has been registered, its remaining steps are performed by the *Move_Help* operation (line M12).
2. The second step of a *Move* operation (and the first in the *Move_Help* operation) attempts to update the to location. The current value of *to is read (line H1), a check ensures that the current operation is still active (lines H2 to H4) and a new value for the to location is prepared (line H5). Then *CAS* is used to update to with the new value if and only if the current value is null and still the same as it was read at line H1. Otherwise, if the contents of to is not null and not already pointing to the superblock (line H7) the destination is occupied and this *Move* is abandoned. The information about the *Move* is removed from the superblock (lines H8 - H11) and SB_NOT_MOVED is returned.
3. The from location is set to null using *CAS* (line H13). The *CAS* succeeds if

and only if from still contains the expected superblock reference (including the right version). If it does not then someone else has already helped this move to complete this step.

4. The last step is to remove the move information from the superblock. The current value of sb.mv_info is read using *Load_Linked* (line H14) and if the value belongs to this operation (line H15), *Store_Conditional* is used to replace it with the no op value. The *Move* operation is now finished and returns SB_MOVED_OK (line H18).

In the presentation above and in the pseudo-code in Algorithm 6.3 we use the atomic primitive *CAS* to update shared variables that fit in a single memory word, but other atomic synchronization primitives, such as *Load-Linked/Store-Conditional* could be used as well. The auxiliary mv_info variable in a superblock might need to be larger than one word. To handle that we use the lock-free software implementation of *Load-Linked/Store-Conditional* for large words by Michael [Mic04b], which can be implemented efficiently from the common single-word *CAS*. Some hardware platforms provide a *CAS* primitive for words twice as wide as the standard word size, which could also be used for this.

Dereferencing a superblock reference.

The operation *Strict_Dereference*, last in Algorithm 6.3, is used to read a superblock reference variable (i.e. a value of type superblock_ref_t) in shared memory when we desire to always get a reference to a *stable* superblock, i.e. one that currently is not involved in any ongoing *Move* operation. *Strict_Dereference* achieves this by helping any ongoing *Move* operation concerning the referenced superblock to finish before it it returns the reference.

6.4.3 Correctness

The Move operation

Lemma 6.4.1 (Linearizability) *All executions of a set of (possibly concurrent) Move operations $m_1, m_2, \ldots, m_n$ on the same superblock sb are linearizable.*

Proof: The linearization point of each *Move* operation is the *Store-Conditional* on line M8 of step 1: A new move operation m will successfully register when the operation information for m is successfully written into the superblock (see step 1 above). This can only happen when there is no other ongoing *Move*:s concerning that superblock. Once the operation m has been registered (i.e. it has completed step 1 successfully) no other move operation m' concerning this superblock will be able to register until m has finished (either by succeeding or failing) by itself or by having been helped by such an m'. Therefore, such an operation m' will be linearized after m. By recursively repeating the above argument, we get the lemma. $\square$

Algorithm 6.2 The flat-set operations *Get_Any* and *Insert*.

```
function Get_Any(set : in out flat-set_t, sb : in out superblock_ref_t,
      loc : in out pointer to superblock_ref_t)
return status_t
         i, j : integer;   old_current : flat-set_info_t;
begin
G1   loop
G2       old_current := set.current;
G3       if old_current.empty then
G4          return FAILURE;
G5       i := old_current.index;
G6       for j := 1 .. set.size do
G7          sb := Strict_Dereference(&set.set[i]);
G8          if pointer(sb) /= null then
G9             loc := &set.set[i];
G10            set.current := (i, false);// Clear empty flag
G11            return SUCCESS;
G12         if i == 0 then i := set.size - 1; else i–;
G13      if CAS(&set.current, old_current,
                (old_current.index, true)) then
G14         return FAILURE;
end Get_Any;

function Insert(set : in out flat-set_t, sb : in superblock_ref_t,
      loc : in out pointer to superblock_ref_t)
return status
         i, j : integer;
begin
I1   loop
I2       i := (set.current.index + 1) mod set.size;
I3       for j := 1 .. set.size do
I4          while pointer(set.set[i]) == null do
I6             set.current := (i, false);
I7             case Move(sb, loc, &set.set[i]) is
I8             when SB_MOVED_OK:
I9                set.current := (i, false);
I10               loc := &set.set[i];
I11               return SB_MOVED_OK;
I12            when SB_MOVED:
I13               return SB_MOVED;
I14            when others:
I15            end case;
I16         i := (i + 1) mod set.size;
I17      if set.set not changed since prev. iter. then
I18         return FAILURE; /* The flat-set is full. */
end Insert;
```

Algorithm 6.3 The superblock *Move* operation.

```
function Move(sb : in superblock_ref_t,
      from : in pointer to superblock_ref_t,
      to : in pointer to superblock_ref_t)
return status
          new_op, old_op : move_info_t;
          cur_from : superblock_ref_t;
begin
      /* Step 1: Initiate move. */
M1  loop
M2      old_op := Load_Linked(&sb.mv_info);
M3      cur_from := *from;
M4      if pointer(cur_from) /= pointer(sb) then
M5          return SB_MOVED;
M6      if old_op.new_pos == null then /* No current operation. */
M7          new_op := (version(cur_from), to, from);
M8          if Store_Conditional(&sb.mv_info, new_op)
M9          then break;
M10     else
M11         Move_Help(make_sb_ref(pointer(sb), old_op.op_id),
                    old_op.cur_pos, old_op.new_pos);
M12 return Move_Help(cur_from, from, to);
end Move;
```

Lemma 6.4.2 (Reachability) *Consider a superblock sb, a set of shared locations $s_1, s_2, \ldots, s_m$ and a set of* Move *operations M_{sb} on sb, where, initially, exactly one shared location s_1 contains a reference to the superblock. Then at least one of the shared locations holds a reference to the superblock at all times during the execution of the move operations.*

Proof: The shared location from is not cleared (line H13 in *Move_Help* until the shared location to has already been successfully updated with a reference to the superblock being moved (line H6). Furthermore, the only way to remove the superblock reference from to is to successfully use another move operation to move the superblock reference from to to another shared location. □

Lemma 6.4.3 (No pointer multiplication) *Consider a superblock sb, a set of shared locations $s_1, s_2, \ldots, s_m$ and a set of* Move *operations M_{sb} on sb, where, initially, exactly one shared location s_1 contains a reference to the superblock, then: (i) after all moves of sb have finished there is exactly one shared reference to sb; and (ii) at any given time there are no more than two shared references to sb.*

Proof: Since no new *Move* can begin before all previous ones concerning this superblock have finished (by themselves or through helping) there will only be

Algorithm 6.4 The superblock *Move_Help* operation.

```
function Move_Help(sb : in superblock_ref_t,
      from : in pointer to superblock_ref_t,
      to : in pointer to superblock_ref_t)
return status
          old, new, res : superblock_ref_t; mi : move_info_t;
begin
      /* Step 2: Update "TO". */
H1    old := *to;
H2    mi := Load_Linked(&sb.mv_info);
H3    if mi /= (version(sb), to, from) then
          /* This operation has been helped to completion. */
H4        return SB_MOVED;
H5    new := make_sb_ref(sb, version(old) + 1);
H6    res := CAS(to, make_sb_ref(null, version(old)), new)
H7    if res /= make_sb_ref(null, version(old)) and pointer(res) /= pointer(sb) then
          /* To is occupied, abandon this operation. */
H8        mi := Load_Linked(&sb.mv_info);
H9        if mi == (version(sb), to, from) then
H10           mi := (0, null, from);
H11           Store_Conditional(&sb.mv_info, mi);
H12       return SB_NOT_MOVED;
      /* Step 3: Clear "FROM". */
H13   CAS(from, sb, make_sb_ref(null, version(sb) + 1));
      /* Step 4: Remove operation information.*/
H14   mi := Load_Linked(&sb.mv_info);
H15   if mi == (version(sb), to, from) then
H16       mi := (0, null, to);
H17       Store_Conditional(&sb.mv_info, mi);
H18   return SB_MOVED_OK;
end Move_Help;
```

Algorithm 6.5 The superblock *Strict_Dereference* operation.

```
function Strict_Dereference(loc : in pointer to superblock_ref_t)
return superblock_ref_t
          sb : superblock_ref_t; mi : move_info_t;
begin
SD1   loop
SD2       sb := *loc;
SD3       if pointer(sb) == null then return sb;
SD4       mi := Load_Linked(&sb.mv_info);
SD5       if mi.new_pos == null then
SD6           return sb;
SD8       else
              /* Help ongoing move operation to finish. */
SD9           Move_Help(sb, mi.cur_pos, mi.new_pos);
end Strict_Dereference;
```

Algorithm 6.6 The superblock operations *Get_block* and *Put_Block*.

```
function Get_Block(sb : in pointer to superblock_t) return pointer to block_t
        nb : block_ref_t;
begin
GB1 loop
GB2     nb := sb.freelist_head;
GB3     if pointer(nb) /= null then
GB4         if CAS(&sb.freelist_head, nb, make_ref(nb.next, version(nb) + 1)) then
GB5             FAA(&sb.free_block_cnt, -1);
GB6             return pointer(nb);
GB7     else
GB8         return null;
end Get_Block;

procedure Put_Block(sb : in pointer to superblock_t, bl : in pointer to block_t)
        oh : block_ref_t;
begin
PB1 loop
PB2     oh := sb.freelist_head;
PB3     bl.next := pointer(oh);
PB4     if CAS(&sb.freelist_head, oh, make_ref(bl, version(oh) + 1)) then
PB5         FAA(&sb.free_block_cnt, 1);
PB6         return;
end Put_Block;
```

one shared reference when a move begins and the move itself creates at most one more during its execution. □

Lemma 6.4.4 (Strict_Dereference I) *The operation Strict_Dereference always returns a superblock that was* stable *(i.e. there was no move operation concerning that superblock) at some time instant during the operation's duration.*

Proof: *Strict_Dereference* will only return a superblock reference that was stable at line SD4 when its mv_info was read by construction. If the reference was unstable at that point *Strict_Dereference* attempts to help the ongoing *Move* (line SD9) and continues for another iteration of the loop. □

Lemma 6.4.5 (Strict_Dereference II) *The operation Strict_Dereference is lock-free.*

Proof: There is only one loop in *Strict_Dereference* and none in *Move_Help*. For a *Strict_Dereference* to remain in the loop it needs to read a non-null superblock reference from loc (line SD2) and, further, find that the referenced superblock is unstable (line SD5). If the referenced superblock is unstable *Strict_Dereference* will make it stable by the *Move_Help* operation that finishes an ongoing *Move* operation. So, in the next iteration loc is either: (i) null, if the move operation was moving the superblock from loc; or (ii) references the now stable superblock; in both these cases will *Strict_Dereference* terminate. However, loc could also contain a reference to a different (unstable or stable) superblock that could force *Strict_Dereference* to do another iteration. But in that case, clearly, some other concurrent operation (i.e. the one that wrote the new value into loc) has made progress, so it is acceptable for *Strict_Dereference* not to make progress. □

The flat-set Insert and Get_Any operations

Lemma 6.4.6 (Flat-set I) *A successful Get_Any(s, sb, loc) returns a superblock that was* stable *in the flat-set at some time instant during the duration of the Get_Any operation.*

Proof: *Get_Any* uses *Strict_Dereference* to read the reference it returns (line G7). The lemma then follows from Lemma 6.4.4. □

Lemma 6.4.7 (Flat-set II) *A superblock is* stable *in at most one flat-set at any time instant.*

Proof: Since superblocks are always moved by applying the *Move* operation the Lemma follows from Lemma 6.4.1, Lemma 6.4.3 and the definition of stable.
□

Lemma 6.4.8 (Flat-set III) *When a superblock sb is being moved from one flat-set A to another B by a Move operation no search operation (i.e. Get_Any) on A can find, by executing Strict_Dereference, sb location in A empty before sb can be found* stable *in B, also by a Strict_Dereference.*

Proof: Consider the *Strict_Dereference* on *sb*'s location in A, (denote that location l_A and the new location in B l_B). According to the Lemma is must return null. There are two cases:
(i) *Strict_Dereference* read null from l_A (line SD2). But *Move* updates l_B (step 2) before clearing l_A (step 3), so a *Strict_Dereference* on to l_B at this time will return *sb* (ii) *Strict_Dereference* read the reference to the unstable *sb* from l_A. In this case *Strict_Dereference* will itself execute *Move_Help* on *sb*, thus making *sb* stable at the new location l_B before returning anything. □

6.5 Managing the blocks within a superblock

The allocatable memory blocks within each superblock are kept in a lock-free IBM free-list [IBM83]. The IBM free-list is essentially a lock-free stack implemented from a single-linked-list where the push and pop operations are done by a *CAS* operation on the head-pointer. To avoid ABA-problems the head-pointer contains a version field. Each block has a header containing a pointer to the superblock it belongs to and a next pointer for the free-list. The two free-list operations *Get_Block* and *Put_Block* are shown in Algorithm 6.6. The free blocks counter, sb.free_block_cnt, is used to estimate the fullness of a superblock.

6.6 Interacting with NBMALLOC

The user application interacts with the memory allocator via the two operations: *malloc*, to make a memory request, and *free*, to release previously allocated memory.

These two operations together with the main global data structures of the memory allocator are shown in Algorithm. 6.7. The Global_Heap structure consists of an array of flat-set, one for each size-class, which contains all empty or nearly empty superblocks in the system (cf. also Figure 6.2(a)). A per-processor heap is a two dimensional array of flat-sets indexed by size-class and superblock fullness group.

A memory request *malloc* call for a memory block of a certain size-class sc will first search the flat-sets for the required size-class in the appropriate per-processor heap for a superblock with a free block. The search begins in the flat-set for the "almost full" fullness-group. If no suitable superblock is found there, the search proceeds to search in the lower fullness-groups (lines A1 to A7 in Algorithm 6.7). Searching in the almost full superblocks set first is a strategy to reduce external fragmentation, since it allows less full superblocks in the per-processor to get fewer allocation requests and in time become empty enough to be moved to the global heap. If no superblock with a free block is found in the per-processor heap, *malloc* will attempt to get a new superblock from the Global_Heap (line A9) and move it to the per-processor heap (line A10). If such a superblock is found then *malloc* tries to allocates a block from it. If no such superblock is found the system is out of memory for this size-class and *malloc* will return null.

The operation *free* is used by the application to return a no longer needed block of memory to the memory allocator. The operation uses the owner field of the block header to find the superblock it belongs to. When the block is returned to the superblock (line F1) the superblock might need to be moved to a different fullness-group or, if it has become almost empty, to the global heap. To be able to move the superblock, its current location is needed. The location is read from the mv_info field in the superblock (line F4) and the superblock is then moved at line F9 or F11. The test at line F7 makes sure that it is the right superblock that is going to be moved – the superblock in question could have been removed from sbr between line F4 and F6 by some concurrent operation.

6.7 Performance evaluation

6.7.1 Systems

There are two major families of cache-coherent multiprocessor architectures – UMA (Uniform Memory Architecture) and NUMA (Non-Uniform Memory Architecture). In a UMA system all processors have the same latency to the memory. In a NUMA system, this is not the case, since access to memory on another node can be significantly slower.

The performance of the new lock-free allocator has been measured on a three multiprocessor systems, both on UMA and NUMA memory architectures. The three systems are

- (i) an UMA Sun Sun-Fire 880 with 6 900MHz UltraSPARC III+ (4MB L2 cache) processors running Solaris 9;
- (ii) a ccNUMA SGI Origin 2000 with 30 250Mhz MIPS R10000 (4MB L2 cache) processors running IRIX 6.5;
- (iii) a PC with 2 2.80GHz Intel Xeon (512KB L2 cache) processors running Linux 2.6.9-22 SMP.

6.7.2 Benchmarks

We used three common benchmarks to evaluate our memory allocator:

Algorithm 6.7 The *malloc* and *free* operations.

```
Global_Heap : global shared array [SIZE_CLASSES] of flat-set_t;
type per-processor_heap_t is array
     [SIZE_CLASSES] [MIN_FULLNESS .. MAX_FULLNESS] of flat-set_t;

function malloc(sc : in size_class) return pointer to block_t
        heap : pointer to per-processor_heap_t := select_heap(thread_id);
        sb : superblock_ref_t; sbr : pointer to superblock_ref_t;
        bl : block_ref_t:= null;
begin
A1  for fg := MAX_FULLNESS - 1 .. MIN_FULLNESS do
A2     while Get_Any(heap[sc][fg], sb, sbr) = SUCCESS do
A3        bl := Get_Block(pointer(sb));
A4        if bl /= null then
A5           exit for loop;
A6        else
             /* Move the full superblock out of the way. */
A7           Insert(heap[sc][MAX_FULLNESS], sb, sbr);
A8  while bl == null loop
       /* Move a superblock from the global heap to the per-processor heap. */
A9     if Get_Any(Global_Heap[sc], sb, sbr) then
A10       if SB_MOVED_OK == Insert(heap[sc][MIN_FULLNESS], sb, sbr) then
A11          bl := Get_Block(pointer(sb));
A12    else
A13       return null; /* Out of memory. */
A14 if fullness(sb) /= fg then  /* Move the superblock to the right fullness group. */
A15    Insert(heap[sc][fullness(sb)], sb, sbr);
A16 return bl;
end malloc;

procedure free(bl : in pointer to block_t)
        sbp : pointer to superblock_t := bl.owner;
        heap : pointer to per-processor_heap_t := sbp.owner;
        newfg, oldfg : fullness_group := fullness(sbp);
        mv_info : move_info_t; sb : superblock_ref_t; sbr : pointer to superblock_ref_t;
begin
F1  Put_Block(sbp, bl);
F2  newfg := fullness(sbp);
F3  if newfg ≠ oldfg then
F4     mv_info := Load_Linked(&sbp.mv_info);
F5     sbr := mv_info.cur_pos;
F6     sb := *sbr;
F7     if pointer(sb) == sbp then
F8        if newfg == empty or almost empty then
F9           Insert(Global_Heap[sc], sb, sbr);
F10       else
F11          Insert(heap[sc][newfg], sb, sbr);
end free;
```

- The **Larson** [BMBW00, Ber02, LK98] benchmark simulates a multi-threaded server application that makes heavy use of dynamic memory. Each thread allocates and deallocates objects of random sizes (between 5 to 500 bytes) and also transfers some of the objects to other threads to be deallocated there. The benchmark result is throughput in terms of the number of allocations and deallocations per second, which reflects the allocator's behaviour with respect to false-sharing and scalability, and the resulting memory footprint of the process which should reflect any tendencies for heap blowup. We measured the throughput during 60 second runs for each set of threads.
- The **Active-false** and **passive-false** [BMBW00, Ber02] benchmarks measure how the allocator handles active (i.e. directly caused by the allocator) respective passive (i.e. caused by application behaviour) false-sharing. In the benchmarks each thread repeatedly allocates an object of a certain size (1 byte) and read and write to that object a large number of times (1000) before deallocating it again. If the allocator does not take care to avoid false-sharing, several threads might get objects located in the same cache-line and this will slow down the reads and writes to the objects considerably. In the *passive-false* benchmark all initial objects are allocated by one thread and then transfered to the others to introduce the risk of passive false-sharing when those objects are later freed for reuse by the threads. The benchmark result is the total wall-clock time for performing a fixed number (10^6) of allocate-read/write-deallocate cycles among all threads.

6.7.3 Implementation

In our memory allocator[2] we use the *CAS* primitive (implemented from the hardware synchronization instructions available on the respective system) for our lock-free operations. To avoid ABA problems we use the version number solution ([Val95b], cf. section 6.2.2). We use 16-bit version numbers for the superblock references in the flat-sets. The reason is that for a bad event (i.e. that a *CAS* of a superblock reference succeeds when it should not) to happen not only must the version numbers be equal but also that same superblock must have been moved back to the same location in the flat-set, which contains thousands of locations. We use superblocks of 64KB and this leaves enough space for version numbers in superblock pointers. We also use size-classes that are powers of two, starting from 8 bytes. This is not a decision forced by the algorithm; a more tightly spaced set of size-classes can also be used; this would impose some extra fixed space overhead due to the preallocated flat-sets for each size-class, but it would also further reduce internal fragmentation. Blocks larger than 32KB are allocated directly from the operating system instead of being handled in superblocks. Our implementation uses four fullness-groups and a fullness-change-threshold of $\frac{1}{4}$, i.e. a superblock is not moved to a new

[2]Our implementation is available at http://www.cs.chalmers.se/∼dcs/nbmalloc.html .

group until its fullness is more than $\frac{1}{4}$ outside its current group. This prevents superblocks from rapidly oscillating between fullness-groups. Further, we set the maximum size for the flat-sets used in the global heap and for those in per-processor heaps to 4093 superblocks each (these values can be adjusted separately).

6.7.4 Results

In the evaluation we compare our allocator with the standard "libc" allocator of the respective platform using the above standard benchmark applications. On the Sun platform, for which we had the original Hoard allocator available, we also compare with Hoard (version 3.4.0). To the best of our knowledge, Hoard is not available for ccNUMA SGI IRIX platforms. Note that on the PC/Linux platform, the default "libc" malloc is in fact Ptmalloc, a lock-based concurrent memory allocator with private per-processor heaps by Gloger [Glo03].

The benchmarks are intended to test scalability, fragmentation and false-sharing behaviour, which are the evaluation criteria of a good concurrent allocator, as explained in the introduction. When performing these experiments our main goal was not to optimize the performance of the lock-free allocator, but rather to examine the benefits of the lock-free design itself. There is plenty of room for optimization of the implementation.

The results from the two false-sharing benchmarks, shown in Figures 6.3 - 6.5 and Figures 6.6 - 6.8, respectively, show that our memory allocator and Hoard, induce very little false-sharing. The standard "libc" allocator, on the other hand, suffers significantly from false-sharing as shown by its longer and irregular runtimes. For "libc" false-sharing causes the largest slowdown when there are few but fully concurrent threads, as they are the most likely to get objects in the same cache-line and also access them concurrently. When the number of threads gets larger, objects are more likely to be in different cache-lines and also, due to time-sharing, not all threads execute at the same time.

An important observation throughout the experiments is that NBmalloc shows consistent behaviour as the number of processors and memory architectures change.

The throughput results from the Larson benchmark, shown in Figures 6.9 - 6.11, show that our lock-free memory allocator has good scalability, not only in the case of full concurrency (where Hoard also shows excellent scalability), but also when the number of threads increases beyond the number of processors. In that region, Hoard's performance quickly drops from its peak at full concurrency on the Sun (cf. Figure 6.10) and slowly on the PC (cf. Figure 6.9).

We can actually observe more clearly the scalability properties of the lock-free allocator in the performance diagrams on the SGI Origin 2000 platform (Figure 6.11). We can observe a linear-style of throughput increase when the number of processors increases (when studying the diagram recall we have 30 processors available in the Origin 2000). Furthermore, when the load on each processor increases beyond 1 thread, the throughput of the lock-free allocator

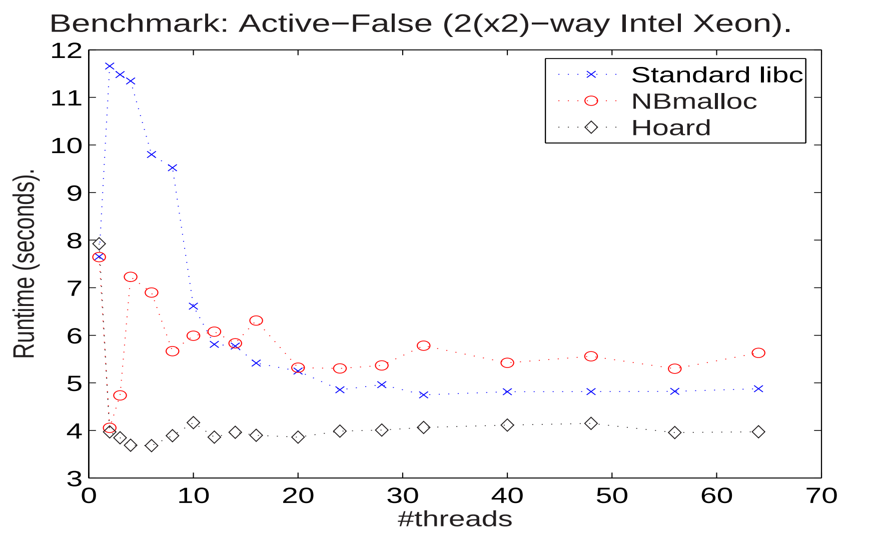


Figure 6.3: The Active-False benchmark: PC with 2 Intel Xeon CPUs.

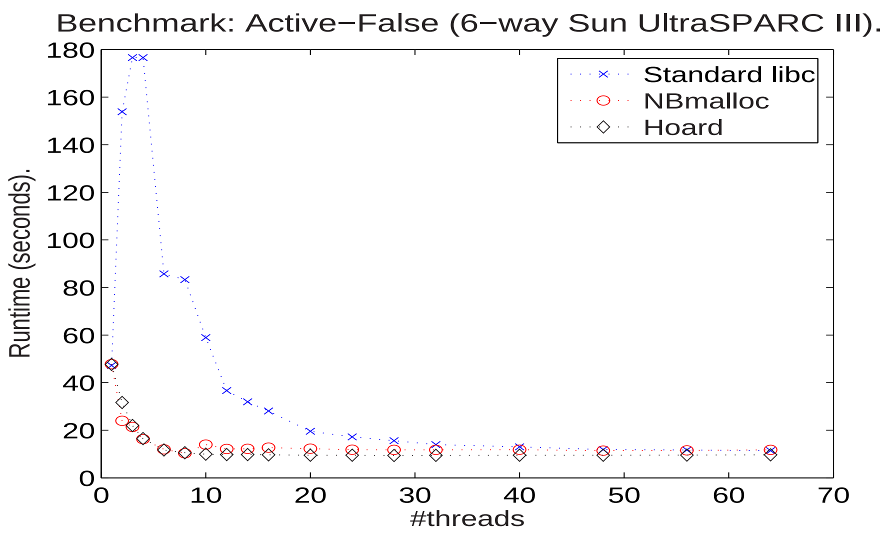


Figure 6.4: The Active-False benchmark: Sun with 6 UltraSPARC III+ CPUs.

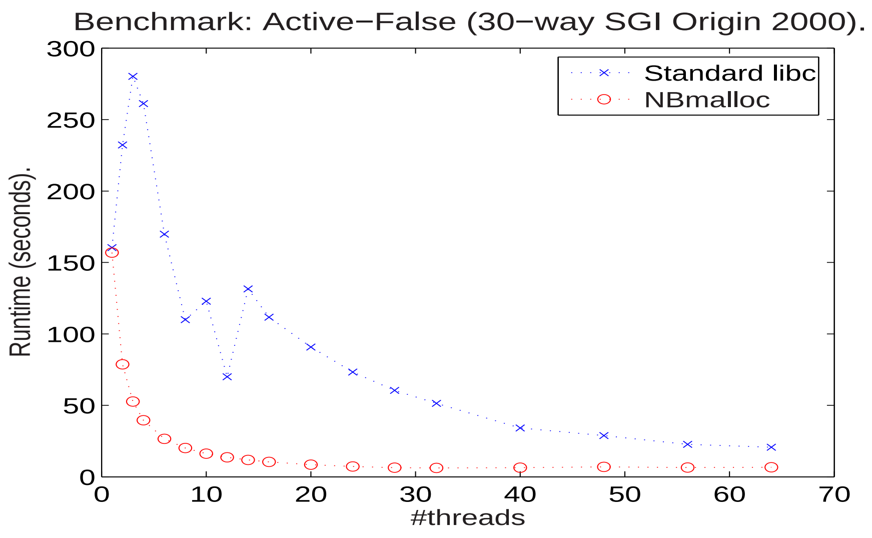


Figure 6.5: The Active-False benchmark: SGI Origin 2000 with 30 MIPS 10k CPUs.

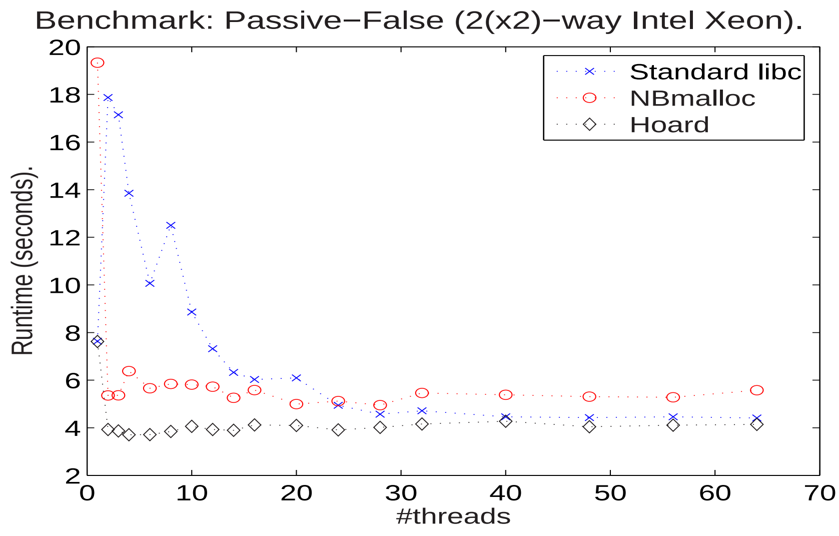


Figure 6.6: The Passive-False benchmark: PC with 2 Intel Xeon CPUs.

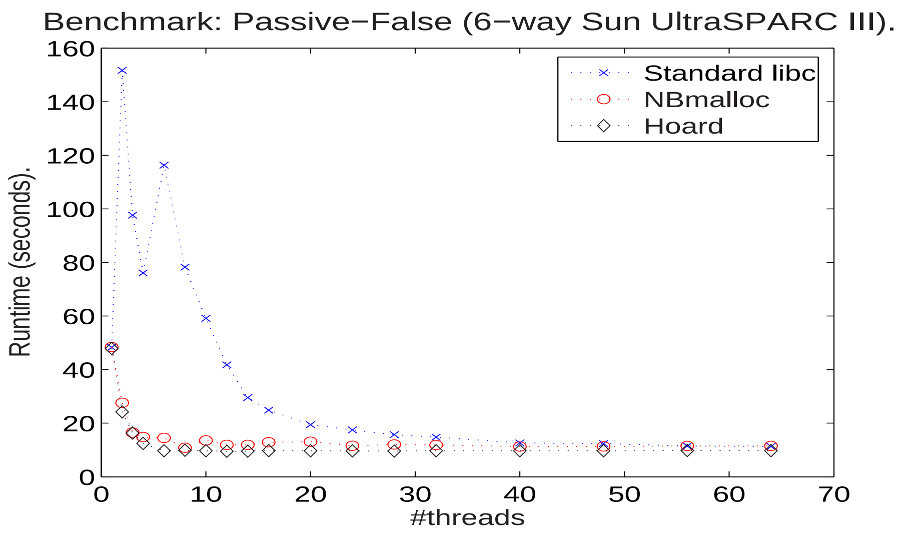


Figure 6.7: The Passive-False benchmark: Sun with 6 UltraSPARC III+ CPUs.

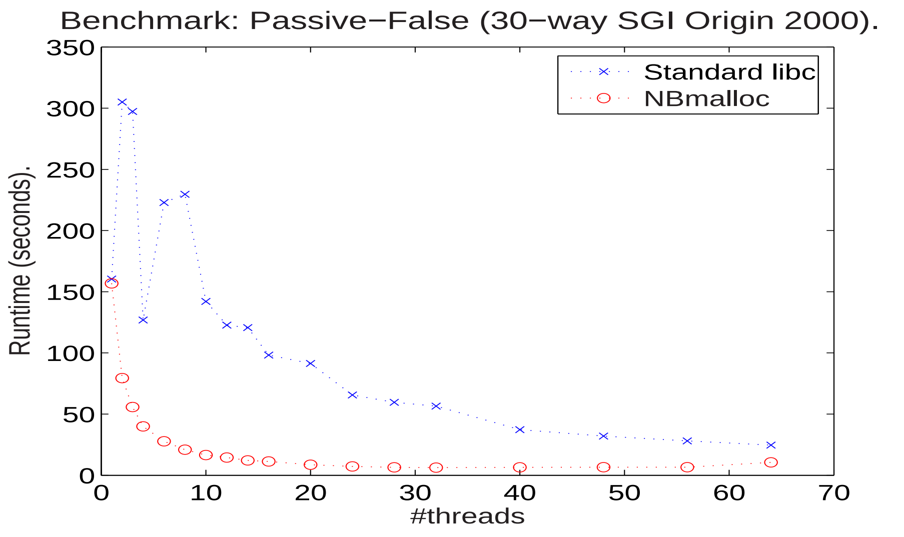


Figure 6.8: The Passive-False benchmark: SGI Origin 2000 with 30 MIPS 10k CPUs.

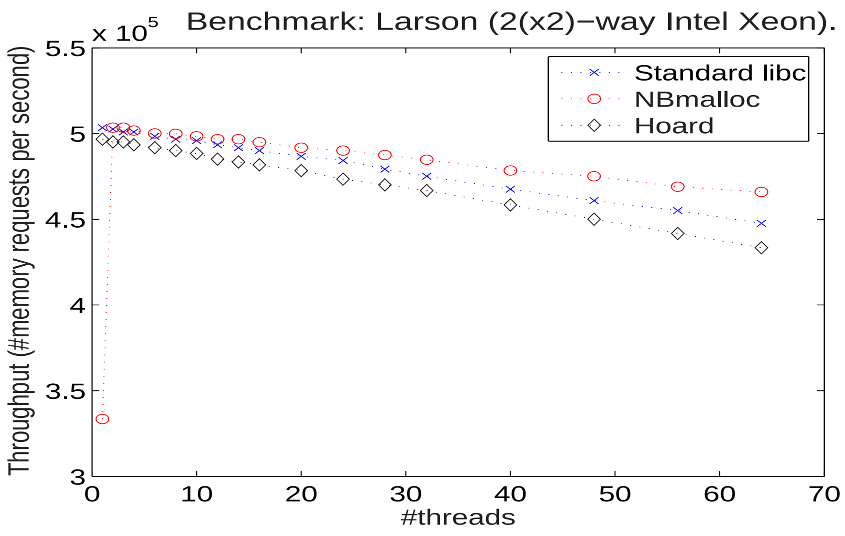


Figure 6.9: The Larson benchmark: Throughput on a PC with 2 Intel Xeon CPUs.

stays high, as is desirable for scalability. In terms of absolute throughput, Hoard is superior to NBmalloc on the Sun platform where we had the possibility to compare them. This is not surprising, considering that it is very well designed and has been around enough time to be well tuned. It is interesting to note that on the PC/Linux platform the situation is reversed and except for the sequential case NBmalloc has higher throughput than both Hoard and Ptmalloc/glibc.

An interesting conclusion is that the scalability of Hoard's architecture is further enhanced by lock-free synchronization.

The results with respect to memory consumption, in Figures 6.12 - 6.14, show that for the Larson benchmark the memory usage (and thus fragmentation) of the non-blocking allocator stays at a similar or better level than Hoard (cf. Figure 6.12 and 6.13). Moreover, the use of per-processor heaps with thresholds, while having a larger overhead than the "libc" allocator, still have almost as good scalability with respect to memory utilization as a single heap allocator.

Moreover, that our lock-free allocator shows a very similar behaviour in throughput on both the UMA and the ccNUMA systems is an indication that there are few contention hot-spots, as these tend to cause much larger performance penalties on NUMA than on UMA architectures.

6.8 Other related work

Recently Michael presented a lock-free allocator [Mic04c] that, like our contribution, is loosely based on the Hoard architecture. Our work and Michael's

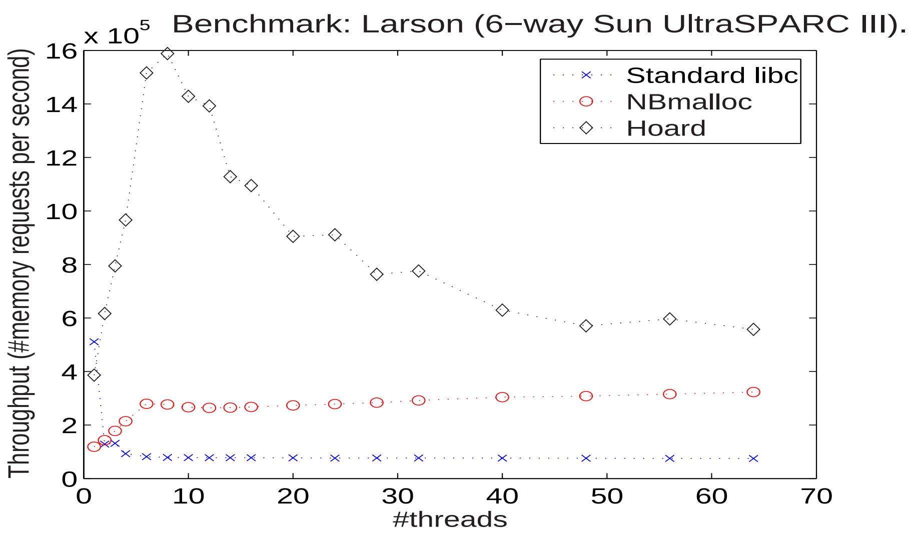


Figure 6.10: The Larson benchmark: Throughput on a Sun with 6 UltraSPARC III+ CPUs.

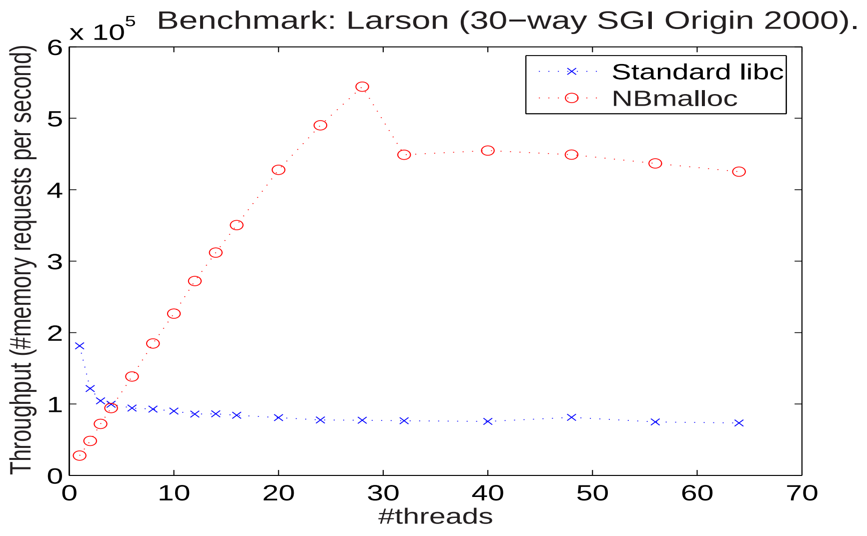


Figure 6.11: The Larson benchmark: Throughput on a SGI Origin 2000 with 30 MIPS 10k CPUs.

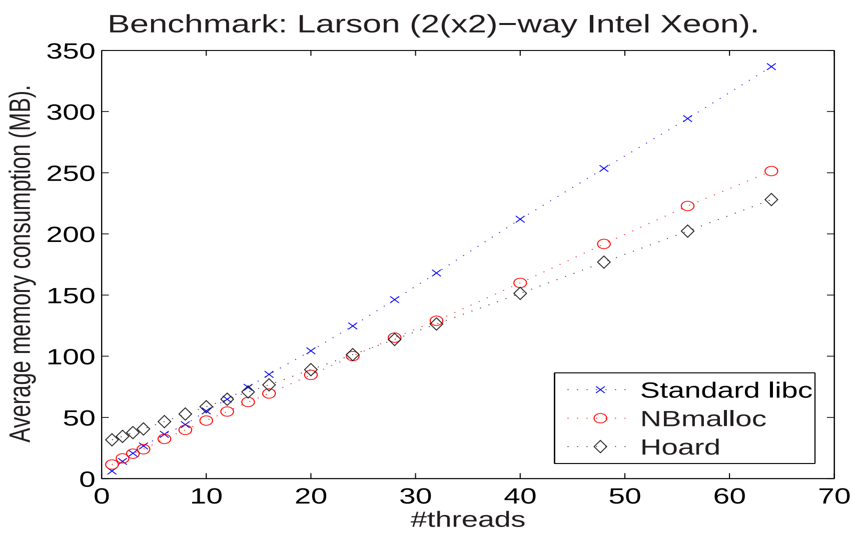


Figure 6.12: The Larson benchmark: Average memory consumption on a PC with 2 Intel Xeon CPUs.

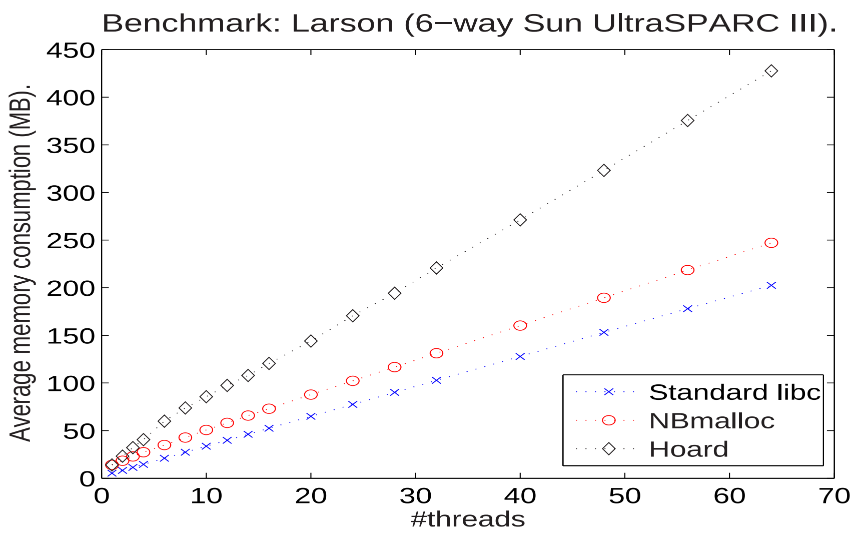


Figure 6.13: The Larson benchmark: Average memory consumption on a Sun with 6 UltraSPARC III+ CPUs.

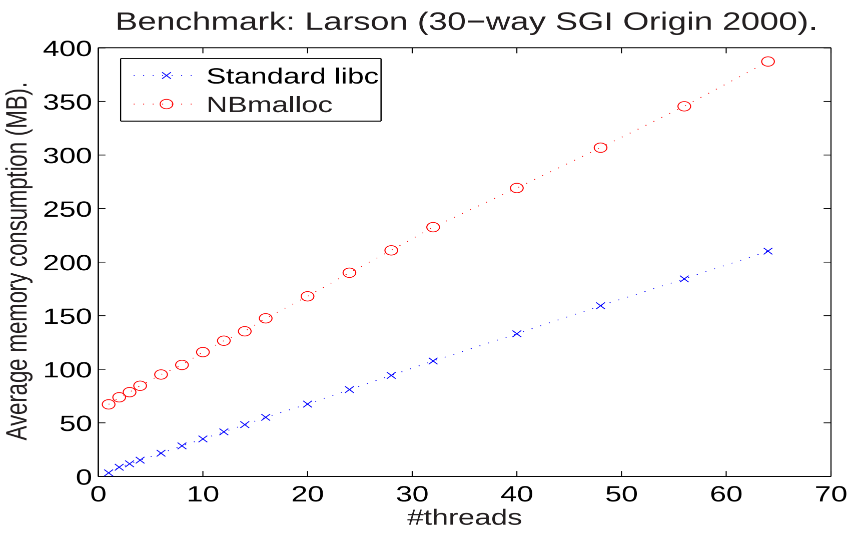


Figure 6.14: The Larson benchmark: Average memory consumption on a SGI Origin 2000 with 30 MIPS 10k CPUs.

have been done concurrently and independently, an early version of our work is in the technical report [GPT04]. Despite both having started from the Hoard architecture, we have used two different approaches to achieve lock-freedom. In Michael's allocator each per-processor heap contains one active (i.e. used by memory requests) and at most one inactive partially filled superblock per size-class, plus an unlimited number of full superblocks. All other partially filled superblocks are stored globally in per-size-class FIFO queues. It is an elegant algorithmic construction, and from the scalability and throughput performance point of view it performs excellently, as is shown in [Mic04c], in the experiments carried out on a 16-way POWER3 platform. By further studying the allocators, it is relevant to note that: Our allocator and Hoard keep all partially filled superblocks in their respective per-processor heap while the allocator in [Mic04c] does not and this may increase the potential for inducing false-sharing. Our allocator and Hoard also keep the partially filled superblocks sorted by fullness and not doing so, like the allocator in [Mic04c] does, may imply some increased risk of external fragmentation since the fullness order is used to direct allocation requests to the more full superblocks which makes it more likely that less full ones becomes empty and thus eligible for reuse. The allocator in [Mic04c], unlike ours, uses the *first-remove-then-insert* approach to move superblocks around, which in a concurrent environment could affect the fault-tolerance of the allocator and cause unnecessary allocation of superblocks since a superblock is invisible to other threads while it is being moved.

As this is work that has been carried out concurrently and independently with our contribution, we do not have any measurements of the impact of the above differences, however this is interesting to do as part of future work, towards further optimization of these allocators.

Another allocator that reduces the use of locks is LFMalloc [DG02]. It uses a method for almost lock-free synchronization, whose implementation requires the ability to efficiently manage CPU-data and closely interact with the operating system's scheduler. To the best of our knowledge, this possibility is not directly available on all systems. LFMalloc is also based on the Hoard design, with the difference in that it limits each per-processor heap to at most one superblock of each size-class; when this block is full, further memory requests are redirected to the global heap where blocking synchronization is used and false-sharing is likely to occur. However, a comparative study with that approach can be worthwhile, when it becomes available for experimentation.

Earlier related work is the work on non-blocking operating systems by Massalin and Pu [MP91, Mas92] and Greenwald and Cheriton [GC96, Gre99]. They, however, made extensive use of the *2-Word-Compare-And-Swap* primitive in their algorithms. This primitive can update two arbitrary memory locations in one atomic step but is not available in current systems and expensive to do in software.

6.9 Discussion and future work

The lock-free memory allocator proposed in this paper confirms our expectation that fine-grain, lock-free synchronization is useful for scalability under increasing load in the system. To the best of our knowledge, this, together with the allocator that was independently presented in [Mic04c] are also the first lock-free general allocators (based on single-word CAS) in the literature. We expect that this contribution will have an interesting impact in the domain of memory allocators.

In the future we plan to generalize of the method for "inter-object" operations, which is illustrated here in the move operation. A general methodology in this direction would enable combinations of known lock-free data structures (e.g. list-structures) into larger, interconnected ones, to be integrated in systems such as the one studied here.

Acknowledgements

We would like to thank Håkan Sundell for interesting discussions on non-blocking methods and Maged Michael for his helpful comments on an earlier version of this paper.

Chapter 7

LFTHREADS: A lock-free thread library or "Blocking without locking"[1]

Anders Gidenstam Marina Papatriantafilou

Abstract

This paper presents LFTHREADS, a thread library whose synchronization is entirely based on lock-free techniques, which means that no spin-locks or similar synchronization mechanisms are employed in the implementation of the multithreading. This implies that processors are always able to do useful work. It is achieved by a synchronization mechanism that does not need any special kernel support. Since lock-freedom is highly desirable in multiprocessors due to its advantages in performance, fault-tolerance, convoy- and deadlock-avoidance, there is an increased demand in lock-free methods in multiprocessor applications, hence also in multiprocessor system services. This is why the existence of a lock-free multithreading library is important. To the best of our knowledge LFTHREADS is the first thread library that provides a lock-free implementation of a blocking synchronization primitive for application threads.

Keywords: lock-free, multithreading, synchronization, shared memory.

7.1 Introduction

Multiprogramming and threading is a fundamental part of modern operating systems. It allows the processor to be shared by several applications efficiently.

[1]This is chapter is an extended version of the Technical Report 2005:20, Computer Science and Engineering, Chalmers University of Technology, 2005

In this paper we study the problem of implementing multithreading on a multiprocessor or multi-core system from the point of view that a processor should *never* be forced to wait for some action that should be performed by another processor. This means that we cannot use spin-locks or any other locking synchronization mechanism in the implementation of the multithreading. The rationale is that a processor should always be able to do useful work.

Synchronization mechanisms that do not use spin-locks or other locking methods are distinguished into lock-free and wait-free. *Lock-free* synchronization guarantees that in a set of concurrent operations at least one of them makes progress each time and thus eventually completes, while *wait-free* synchronization guarantees that every operation finishes in a finite number of its own steps regardless of the actions of concurrent operations. The correctness condition for lock-free and wait-free constructions is *atomicity*, a.k.a. *linearizability* [HW90]. An execution is *atomic* or *linearizable* if it guarantees that even when operations overlap in time, each of them appears to take effect at an atomic time instant that lies within its respective time duration, such that the effect of each operation is consistent with its corresponding operation in a sequential execution in which the operations appear in the same order.

Lock-free/wait-free synchronization is attractive in multiprocessor and multi-core systems as it offers significant advantages over lock-based synchronization schemes, because: (i) it does not give rise to priority inversion; (ii) it avoids lock convoys; (iii) it provides higher fault tolerance (processor stop failures will not corrupt shared data objects); and (iv) it eliminates deadlock scenarios involving two or more processes both waiting for locks held by the other. Due to these facts there is extended research literature on lock-free/wait-free synchronization (c.f. [Sun04a] for an overview) as well as on *universal methods* to transform lock-based constructions into lock-free/wait-free ones (e.g. [Bar93, Her93, TSP92]). Besides ensuring the above qualitative properties, it has also been shown, using well-known parallel applications, that lock-free methods imply at least as good performance as lock-based ones, and in many cases significantly better [TZ01a, TZ02].

For all the above, lock-free synchronization is desirable in multiprocessors applications, hence also at the operating system level and runtime system level. There has been work at the operating system kernel level [MP91, Mas92, GC96, Gre99], where basic kernel data structures have been replaced with lock-free ones with both performance and quality benefits. Besides, the NOBLE library project [ST02] provides implementation of a large range of data structures using lock-free methods, while there is an increasing interest on projects that aim at providing support for non-blocking programming, e.g. the Software Transactional Memory package for C#[Her06]. There is also extensive interest and results on lock-free methods for memory management (garbage collection, memory allocation, e.g. [Val95b, MS95, DMMS01, Mic04c, GPST05, GPT05]).

In the research literature on multithreading, a special kernel-level mechanism, called *scheduler activations*, has been proposed [ABLL92], to enable user-level threads to offer the functionality of kernel-level threads and also leave no processor idle in the presence of ready threads. In a further study [FCL93] it was shown that application-controlled blocking and interprocess communication can be resolved at user-level without modifications to the kernel to achieve the same goals as above, but multiprogramming demands and blocking, such as for page-faults, need scheduler activations to do the same. Further, there is extensive research literature on efficiently utilizing computational resources in multithreading systems. To mention several relevant and motivating examples: [BL94] focuses on scheduling with work-stealing, as a method to have enough threads active to keep processors busy, meeting also memory constraints; [ZLE91] studied methods of scheduling to reduce the amount of spinning in multithreading; [ZS97] focuses on demands in real-time and embedded systems and studies methods for efficient, low-overhead semaphores. Moreover, [ON01] aims at introducing lock-freedom in multithreading and proposes a scheduling mechanism that does not require locking, that is used in their multithreading library called Lesser Bear.

Here we present a lock-free thread library, LFTHREADS. In LFTHREADS there are no spin-locks or similar synchronization mechanisms employed in the implementation of the multithreading. The lock-freedom of the thread library implementation implies that a processor is always able to continue doing useful work, despite other processors suffering stop failures or delays (e.g. from interrupts caused by page-faults or I/O devices). To do this, we introduce a new synchronization method, which we call *hand-off*, which may also be useful in other lock-free synchronization constructions. Further, the method uses a lock-free queue data structure and standard kernel operations to manage thread execution contexts, without need for scheduler activations. Since lock-freedom is highly desirable in multiprocessors due to its advantages in performance, fault-tolerance, convoy- and deadlock-avoidance, there is an increased demand in lock-free methods in multiprocessor applications, hence also in multiprocessor system services, to retain the lock-free properties even at system level. This is why the existence of a lock-free multithreading library is important. To the best of our knowledge LFTHREADS is the first library that provides a lock-free implementation of a blocking synchronization primitive for application threads. Even a lock-free thread library needs to provide mutual exclusion objects, e.g. for legacy applications and for other applications where threads might need to be blocked, e.g. to interact with some external device. Our hand-off synchronization method in LFTHREADS allows the mutual exclusion object to block application threads without enforcing mutual exclusion among the processors executing the threads.[2]

In the paper we first present the system model (section 7.2), then the application programming interface of LFTHREADS (section 7.3), the detailed description including the algorithmic design of the synchronization in LFTHREADS (section 7.4), the correctness of the above (section 7.5), and finally the implementation and an experimental study of the behaviour of the library (section 7.6). We conclude in section 7.7.

[2]Note that an application that uses the mutual exclusion objects still needs to take care to avoid deadlocks among its threads.

```
function CAS(address : pointer to word;
        oldvalue : word; newvalue : word) : boolean
    atomic do
        if *address = oldvalue then
            *address := newvalue;
            return true;
        else return false;

function FAA(address: pointer to integer;
        increment: integer): integer
    atomic do
        ret := *address;
        *address := ret + increment;
        return ret;
```

Figure 7.1: The Compare-And-Swap (CAS) and Fetch-And-Add (FAA) atomic primitives.

7.2 System model

We consider shared memory multiprocessor systems. The system consists of a set of processors, each having its own local memory as well as being connected to a shared memory through an interconnect network. Each processor executes instructions sequentially at an arbitrary rate. The shared memory might not be uniform, that is, for each processor the latency to access some part of the memory is not necessarily the same as the latency for any other processor to access that part of the shared memory. The shared memory supports atomic read and write operations of any single memory word, and also stronger single-word synchronization primitives, such as Compare-And-Swap (CAS) and Fetch-And-Add (FAA) (see Figure 7.1) which are used in the algorithms in this paper. These synchronization primitives are either available or can easily be derived from other available synchronization primitives [Jay98, Moi97] on most (more advanced) contemporary microprocessor architectures.

The processors in the system cooperate to run a set of application threads, where each thread consists of a sequence of operations and communication is accomplished via shared-memory operations.

7.3 LFTHREADS's application programming interface

The lock-free thread library defines the following functions for thread handling:

procedure create(thread : **out** thread_t; main : **in pointer to procedure**);
procedure exit();
procedure yield();

Procedure *create* creates a new thread which will start in the procedure main. Procedure *exit* terminates the calling thread and if this was the last thread of the application/process it is terminated as well. Procedure *yield* causes the calling thread to be put on the ready queue and the (virtual) processor that it was running on to pick a new thread to run from the ready queue.

For applications that need lock-based synchronization between threads the thread library provides a mutex object with the following operations:

```
procedure lock(mutex : in out mutex_t)
procedure unlock(mutex : in out mutex_t)
function trylock(mutex : in out mutex_t): boolean
```

Procedure *lock* attempts to lock the mutex. If the mutex is locked already the calling thread is blocked and enqueued on the waiting queue of the mutex. Function *trylock* attempts to lock the mutex. If it succeeds in locking the mutex true is returned, otherwise false. Procedure *unlock* unlocks a mutex if there are no threads waiting in the mutex's waiting queue, otherwise the first of the waiting threads are removed from the waiting queue and made runnable. That thread is now the owner of the mutex. Only the thread that has locked the mutex may call *unlock*.

7.4 Detailed description of the LFTHREADS library

7.4.1 Data structures and global variables

In Algorithm 7.1 the data structures used to implement the thread library are presented. We assume that we have a data type, context_t, where the CPU context of an execution (i.e. thread) can be stored and some operations to manipulate such contexts. These operations, which can be supported by most common operating systems[3], are:
(i) *save* which stores the state of the current CPU context in the supplied variable and switches processor to a new special stack. There is one such stack available for each processor. The return value from *save* is true when the context is stored and false when the context is restored.
(ii) *restore* which loads the supplied stored CPU context onto the processor. The restored context continues execution in the (old) call to *save*, returning false. The CPU context that made the call to *restore* is lost (unless it was saved before the call to *restore*).
(iii) *make_context* which creates a new CPU context in the supplied variable. The new context will start in a call to the supplied procedure main when it is loaded onto a processor with *restore*.

Each thread in the system will be represented by an instance of the thread control block data type, thread_t, which contains a context_t variable that stores

[3]For example in systems conforming to the Single Unix Specification v2 (*SUSv2*) they can be implemented from `getcontext(2)`, `setcontext(2)` and `makecontext(3)`, while in other Unix systems `setjump(3)` and `longjmp(3)` or similar could be used.

Algorithm 7.1 The thread context type, the thread control block type and the lock-free queue type and their operations used in LFTHREADS.

```
type context_t is record ⟨implementation defined⟩;
function save(context : out context_t): boolean;
/* Saves the current CPU context and switches to the
 * a separate stack. The call to save returns true when
 * the context is saved and false when it is restored. */
procedure restore(context : in context_t);
/* Replaces the current CPU context with a
 * previously stored CPU context.
 * The current context is destroyed. */
procedure make_context(context : out context_t;
        main : in pointer to procedure);
/* Creates a new CPU context which will wakeup
 * in a call to the procedure main when restored. */

type thread_t is record
    uc : context_t;

type lf_queue_t is record ⟨implementation defined⟩;
procedure enqueue(queue : in out lf_queue_t;
        thread : in pointer to thread_t);
/* Places the thread control block thread last in the queue. */
function dequeue(queue : in out lf_queue_t;
        thread : out pointer to thread_t): boolean;
/* If the queue is not empty the first thread_t pointer in the queue
 * is dequeued and true is returned. Returns false if the queue is empty. */
function is_empty(queue : in out lf_queue_t): boolean;
/* Returns true if the queue is empty, and false otherwise. */
```

the thread's state when it is not being executed on one of the processors.

Further, we also assume that we have a lock-free queue data structure (like e.g. [TZ01b]) for pointers to thread control blocks; the queue supports three lock-free and linearizable operations: *enqueue*, *dequeue* and *is_empty* (each with its intuitive semantics). The lock-free queue data structure is used as a building block in the implementation of LFTHREADS. However, as we will see in detail below, additional synchronization methods are needed to make operations involving more than one queue instance lock-free and linearizable.

The global and local variables needed are shown in Algorithm 7.2. The persistent global and per-processor variables consist of the global Ready_Queue, which contains all runnable threads that are not currently being executed by any processor, and the per-processor persistent variable Current, which contains a pointer to the thread control block of the thread that is currently being executed on that particular processor.

Algorithm 7.2 The variables used in LFTHREADS.

```
/* Global shared variables. */
Ready_Queue : lf_queue_t;

/* Private per-processor persistent variables. */
Current$_p$ : pointer to thread_t;

/* Local temporary variables. */
next : pointer to thread_t;
old_count : integer;
```

Algorithm 7.3 The basic thread operations in LFTHREADS.

```
procedure create(thread : out thread_t;
              main : in pointer to procedure)
C1   make_context(thread.uc, main);
C2   enqueue(Ready_Queue, thread);

procedure yield()
Y1   if not is_empty(Ready_Queue) then
Y2       if save(Current$_p$.uc) then
Y3           enqueue(Ready_Queue, Current$_p$);
Y4           cpu_schedule();

procedure exit()
E1   cpu_schedule();

procedure cpu_schedule()
CI1  loop
CI2      if dequeue(Ready_Queue, Current$_p$) then
CI3          restore(Current$_p$.uc);
```

Algorithm 7.4 The lock-free mutex data type in LFTHREADS.

```
type mutex_t is record
    waiting : lf_queue_t;
    count : integer := 0;  /* UNLOCKED=0 unless hand-off is set. */
    hand-off : boolean := false;
```

7.4.2 The thread operations in LFTHREADS

The general thread operations of LFTHREADS are shown in Algorithm 7.3, where it is possible to see their interaction with the data structures and variables described in the previous section. The thread handling operations, whose required functionality was introduced in section 7.3, work as follows in LFTHREADS:
(i) The operation *create* creates a new thread control block, initializes the context stored in the block and enqueues the new thread on the ready queue.
(ii) The operation *exit* terminates the thread currently being executed by this processor, which then picks another thread to run from the ready queue.
(iii) The operation *yield* saves the context of the thread currently being executed by this processor, enqueues this thread on the ready queue and then picks another thread to run from the ready queue.

In addition to the public operations, there is an internal operation in LFTHREADS, namely *cpu_schedule*, which is used to select the next thread to load onto the processor. If there are no threads waiting for execution in the Ready_Queue, the processor is idle and will spin waiting for a runnable thread to appear.

7.4.3 The mutex operations in LFTHREADS

To facilitate blocking mutual exclusion-based synchronization among application threads, LFTHREADS provides a mutex primitive, mutex_t. While the operations on a mutex, *lock*, *trylock* and *unlock*, have their usual semantics for the application threads, they are lock-free with respect to the processors in the system. This implies a higher degree of fault-tolerance against stop and timing faults in the system compared to a traditional spin-lock-based mutex implementation, since even if a processor is stopped or delayed in the middle of a mutex operation all other processors are still able to continue performing operations on the same mutex.

The mutex_t structure, which is shown in Algorithm 7.4, consists of three fields:

(i) an integer counter, which counts the number of threads that are in or want to enter the critical section protected by the mutex;

(ii) a lock-free queue, where the thread control blocks of threads that want to lock the mutex when it is already locked can be stored; and

(iii) a boolean hand-off flag, whose role and use will be described in detail in this section.

Algorithm 7.5 The lock-free mutex protocol in LFTHREADS.

```
procedure lock(mutex : in out mutex_t)
L1  old_count := FAA(&mutex.count, 1);
L2  if old_count $\neq$ 0 then
       /* The mutex was locked. Help or run another thread. */
L3     if save(Current$_p$.uc) then
L4        enqueue(mutex.waiting, Current$_p$);
L5        if not CAS(&mutex.hand-off, true, false) then
             /* The mutex is still busy. */
L6           cpu_schedule();  /* We are done. */
L7        else  /* We now own the mutex. Find a waiting thread to run. */
L8           loop
L9              if dequeue(mutex.waiting, Current$_p$) then
L10                restore(Current$_p$);  /* Done. */
L11             else  /* The waiting thread not ready. Initiate hand-off! */
L12                mutex.hand-off := true;
L13                if is_empty(mutex.waiting) then  /* It is safe to leave. */
L14                   cpu_schedule();  /* Done. */
L15                if not CAS(&mutex.hand-off, true, false) then
L16                   cpu_schedule();  /* Done. */

function trylock(mutex : in out mutex_t): boolean
TL1 if CAS(&mutex.count, 0, 1) then return true;
TL2 else if CAS(&mutex.hand-off, true, false) then
TL3    FAA(&mutex.count, 1);
TL4    return true;
TL5 return false;

procedure unlock(mutex : in out mutex_t)
U1  old_count := FAA(&mutex.count, $-1$);
U2  if old_count $\neq$ 1 then  /* There is at least one waiting thread. */
U3     loop
U4        if dequeue(mutex.waiting, next) then
U5           enqueue(Ready_Queue, next);
U6           return;
U7        else  /* The waiting thread is not ready yet! Initiate hand-off! */
U8           mutex.hand-off := true;
U9           if is_empty(mutex.waiting) then
                /* Some concurrent operation will see/or has seen the hand-off. */
U10             return;
U11          if not CAS(&mutex.hand-off, true, false) then
U12             return;  /* Some concurrent operation acquired the mutex. */
```

The operations on the mutex_t structure are shown in Algorithm 7.5. In rough terms, the *lock* operation locks the mutex and makes the calling thread its owner. If the mutex is already locked the calling thread is blocked and the processor switches to another thread. The blocked thread's context will be activated and executed later. In the ordinary case a blocked thread's context is activated by the thread releasing the mutex by invoking *unlock*, but due to fine-grained synchronization, it may also happen in other ways. In particular, notice that checking whether the mutex is locked and entering the mutex waiting queue are distinct[4] atomic operations. Therefore, the interleaving of thread-steps can cause such situations that e.g. a thread A finds the mutex locked, but by the time it has entered the mutex queue the mutex has been released, hence A should not remain blocked and wait in the queue. The synchronization required for such cases to be resolved is managed with the *hand-off* method. In particular, the thread(/processor) that is releasing the mutex is able, using appropriate fine-grained synchronization steps, to detect whether such a situation has occurred and, in response, "hand-off" the ownership (or responsibility) for the mutex to the other thread(/processor). By performing the *hand-off*, the processor executing the *unlock* operation can finish this operation and continue executing threads without needing to wait for the concurrent *lock* operation to finish (and vice versa). As a result the mutex primitive in LFTHREADS can tolerate arbitrary delays and even stop failures inside mutex operations without affecting the other processors ability to do useful work (including performing operations on the same mutex). However, do note that individual application threads that use a mutex still have to wait if some other application thread has locked it and that a faulty application still are able to dead-lock its threads. It is the responsibility of the application developer to prevent such situations from happening.

The details of the *hand-off* method are given in the description of the operations, below:

The lock operation: First, at line L1, the count of threads that want to access the mutex is increased atomically using Fetch-And-Add. If the old value was 0 the mutex was free and is now locked by the thread. If the old value was not 0 then the mutex is likely to be locked and the current thread has to block. At line L3 the context of the current thread is stored in its thread control block and on line L4 the thread control block is enqueued in the mutex's waiting queue. From this point this invocation of *lock* is not associated with any thread.

However, the processor cannot just leave and do something else yet, because in the meanwhile (since line L1), the thread that owned the mutex might have unlocked it; this is checked by line L5. If the CAS at line L5 succeeds an *unlock* operation has tried to unlock the mutex but found (line U2) that there is some thread waiting to lock the mutex that has not yet appeared in the waiting

[4]The "traditional" way to avoid this problem is to ensure that only one processor at a time modifies the mutex state, i.e. by enforcing mutual exclusion among the processors, e.g. by using a spin-lock.

queue (line U4) and therefore has set the hand-off flag (line U8). Since the *lock* operation successfully acquired the hand-off flag it is now responsible for the mutex and must find the thread that wants to lock the mutex. If that thread (it might or might not be the one this *lock* operation enqueued) has now appeared in the waiting queue its thread control block is dequeued and this processor will proceed to execute it (line L9 - L10). If the thread has not yet appeared in the waiting queue, the *lock* operation tries to get rid of its responsibility by initiating a *hand-off* (line L12):

- The hand-off is successful if: (i) the waiting queue is still empty at line L13; in that case either the offending thread has not yet been enqueued there (in which case it has not yet checked for hand-offs) or it has in fact already been dequeued (in which case some other processor took responsibility for the mutex); or if (ii) the attempt to retake the hand-off flag at line L15 fails, in which case some other processor has taken responsibility for the mutex. After a successful hand-off the processor leaves the *lock* operation and proceeds to execute a thread fetched from the Ready_Queue.
- If the hand-off is unsuccessful, i.e. the CAS at line L15 succeeds, then this processor is still responsible for the mutex and needs to retry. Note that if the hand-off is unsuccessful, then some other concurrent *lock* operation made progress, namely by completing an enqueue on the waiting queue (otherwise this *lock* would have completed at line L13 and L14).

The trylock operation: It will lock the mutex and return true if the mutex was unlocked otherwise it does nothing and returns false. The operation first tries to lock the mutex by increasing the waiting count on line TL1. This will only succeed if the mutex was unlocked and there were no ongoing *lock* operations. If there are ongoing *lock* operations or some thread has locked the mutex, *trylock* will attempt to acquire the hand-off flag. This might succeed if the thread owning the mutex is trying to unlock it and did not find any thread in the waiting queue. If the *trylock* operation succeeds in acquiring the hand-off flag then it becomes the owner of the mutex and increases the waiting count at line TL3 before returning true. Otherwise *trylock* returns false.

The unlock operation: It unlocks the mutex if there are no waiting threads. If there are waiting threads, one of them is made owner of the mutex and is activated by being enqueued on the Ready_Queue. The operation begins by decreasing the waiting count at line U1, which was increased when this thread called *lock* or *trylock* to lock the mutex. If the count becomes 0, then there are no waiting threads and the *unlock* operation is done. Otherwise, there is at least one thread wanting to acquire the mutex. If that thread has been enqueued in the waiting queue, it is dequeued (line U4) and moved to the Ready_Queue (line U5) which completes the *unlock* operation. It the waiting queue is empty, the *unlock* operation initiates a hand-off to get rid of the responsibility for the mutex (line U8). The hand-off is handled in the same way as in the *lock* operation, the only difference is that the *unlock* operation is completed by a `return`-statement instead of by calling *cpu_schedule*.

7.5 Correctness of synchronization in LFTHREADS

To prove the correctness of the thread library we will first show that the mutex operations are lock-free and linearizable with respect to the processors and then that the lock-free mutex implementation satisfies the conditions for mutual exclusion with respect to the behaviour of the application threads.

We start with (i) defining some notation that will facilitate the presentation of the arguments and (ii) establishing some lemmas that will be used later to prove the safety, liveness, fairness and atomicity properties of the algorithm.

Definition 7.5.1 *A thread's call to a blocking operation Op is said to be* completed *when the processor executing the call leaves the blocked thread and goes on to do something else (like executing another thread). The call is said to have* returned *when the thread (after becoming unblocked) continues its execution from the point of the call to Op.*

A mutex m *is* locked *when calls to* lock *on it block the calling thread and calls to* trylock *return $False$.*

When a thread τ's call to lock *on a mutex* m *returns we say that thread τ has* locked *or* acquired *the mutex* m*. Similarly, we say that thread τ has* locked *or* acquired *the mutex* m *when the thread's call to* trylock *on the mutex* m *returns $True$.*

Further, when a thread τ has acquired a mutex m *by a* lock *or successful* trylock *operation and not yet released it by calling* unlock *we say that the thread τ is the* owner *of the mutex* m *(or that τ* owns m*).*

Lemma 7.5.1 *A mutex* m *is* locked *if and only if* m.count $\neq 0$ *and* m.hand-off $= False$.

Proof: If m.count $= 0$ then, clearly, the mutex is not locked as a call to *lock* returns after line L2 and a call to *trylock* returns $True$ at line TL1.

Assume towards a contradiction that the mutex is locked and m.count $\neq 0$ and m.hand-off $= True$. Now, consider a call to *trylock*. It would return $True$ because the CAS at line TL2 succeeds so the mutex is not locked and we have a contradiction. Similarly, A call to *lock* would save the thread's context and enqueue it on the waiting queue and then succeed with the CAS at line L5 and go on to (try to) dequeue and run this waiting thread, thus letting it become the owner of the mutex.

If, on the other hand, m.count $\neq 0$ and m.hand-off $= False$ then *trylock* would return $False$ at line TL5 since both CAS fails and *lock* would save and enqueue the thread's context on the m.waiting queue (lines L3-L4) and then let the processor continue with other work (line L6), thus leaving the calling thread blocked. □

Lemma 7.5.2 *The value of the* m.count *variable is always greater than zero when a thread* owns *the mutex* m.

Proof: Note that m.count is increased by one for each *lock* (line L1) and each successful *trylock* (line TL1 or TL3) operation and it is decreased by one for each *unlock* operation (line U1). Therefore, m.count cannot be zero unless the number of calls to *unlock* is the same as the number of calls to *lock* and successful calls to *trylock* together. In a correct application all threads must have a matching *unlock* call after each *lock* (or *trylock*) call and no *unlock* calls without a matching *lock* (or *trylock*). For a thread τ to own the mutex it must have called *lock* (or *trylock*) but not (yet) called *unlock* after that, so m.count must be positive. □

7.5.1 lock-freedom

The lock-free property of the thread library operations will be established with respect to the processors. The lock-freedom with respect to application threads of the operations *trylock* and *unlock* follows trivially from their lock-freedom with respect to processors, since there are no context switches in them. For the operation *lock* it is obvious that it cannot be lock-free with respect to application threads, since threads calling *lock* when the mutex is busy should be blocked.

Theorem 7.5.1 *The mutex operations* lock, trylock *and* unlock *are all lock-free.*

Proof: The only instances of non-sequential code are the hand-off loops in the operations *lock* and *unlock*. Consider the loop in *lock*. The conditions that must hold for the processor to stay in the loop are:

(i) the m.waiting queue is empty at line L9; and

(ii) the m.waiting queue is non-empty at line L13; and

(iii) the processor successfully captures the m.hand-off flag at line L15.

For both (i) and (ii) to hold at least one other processor must have completed an enqueue operation on the m.waiting queue between the execution of line L9 and L13 and thus have made progress.[5] The same argument holds for the loop in *unlock*. □

7.5.2 Mutual exclusion properties

The mutual exclusion properties of the new mutex protocol are established with respect to application threads and mutexes.

[5] Note that condition (iii) is irrelevant for the proof of lock-freedom.

Theorem 7.5.2 (Safety) *For any mutex* m *and at any time* t *there is at most one thread* τ *such that* τ *is the owner of* m *at time* t*.*

Proof: A thread τ becomes owner of m because:

(i) it has seen that m.count was zero before τ incremented it (*lock*, *trylock*); since the m.count increments and decrements are atomic, the lemma is satisfied in this case;

(ii) it has succeeded in taking over the ownership (*trylock*) from a thread τ' after τ' detected that there are waiting threads (m.count is non-zero), τ' executed *unlock*, τ' found no thread waiting in m.waiting and τ' handed over the ownership of m by setting m.hand-off to $True$. Since the hand-off is done by τ performing CAS on m.hand-off, and due to the way that m.count is modified, the lemma is satisfied in this case;

(iii) it has been activated by a thread τ'' (performing an *unlock*) that succeeded in the hand-off procedure as in case 2 above. Again, since this is done by τ'' successfully performing CAS on m.hand-off, when m.count is non-zero, the lemma is satisfied here too;

(iv) it has been activated and is being executed by a processor P that succeeded in the hand-off procedure (in *lock* line L8 and below). Since P successfully performed CAS on m.hand-off and m.count is non-zero, the lemma is satisfied here, as well.

□

Lemma 7.5.3 *No thread is left blocked in the waiting queue of an unlocked mutex* m *when all concurrent operations concerning* m *have completed.*

Proof: Recall from Lemma 7.5.1 that a mutex m is locked if and only if m.count $\neq 0$ and m.hand-off $= False$, so the mutex is unlocked if m.count $= 0$ or m.hand-off $= True$.

If m.count $= 0$ then there clearly cannot be any waiting threads since the first action of a thread trying to acquire a mutex using *lock* is to increase m.count.

Assume towards a contradiction that there are no uncompleted operations and there is a thread τ left in the m.waiting queue, m.count $\neq 0$ and m.hand-off $= True$. Consider the *lock* operation by the thread τ. It cannot have been the last operation to complete, because if m.hand-off was $True$, the CAS at L5 would have succeeded and *lock* would enter the hand-off loop with thread τ available in the m.waiting queue.

But if m.hand-off was $False$ when τ's *lock* operation completed, then there must have been some other uncompleted operation active inside a hand-off loop after that point, since that is the only place m.hand-off is set to $True$ (at line L12 or U8). However, for that operation to leave its hand-off loop and complete, it must find the m.waiting queue empty after setting m.hand-off to $True$ (line

L13 or U9). This contradicts our assumption that τ's *lock* operation completed before m.hand-off was set to $True$.

Thus it is impossible that all operations on m completed leaving m unlocked and the thread τ in the m.waiting queue. □

Lemma 7.5.4 *A mutex is locked if and only if it is* owned *by a thread.*

Proof: Recall from Lemma 7.5.1 that a mutex m is locked if and only if m.count $\neq 0$ and m.hand-off $= False$.

Further, by Lemma 7.5.2 and because m.count is increased by *lock* and *trylock* and only decreased by *unlock* the mutex m can only be locked when there are threads that have executed a *lock* or a successful *trylock* but not yet the matching *unlock*. One of these threads owns the mutex. □

Lemma 7.5.5 *A thread τ waiting to acquire a mutex* m *in a call to* lock *will at most have to wait for the thread currently owning* m *and all threads that have called* lock *on* m *before τ's call to* lock *enqueued τ on the* m.waiting *queue.*

Proof: Once the thread τ has been enqueued on the m.waiting queue (line L4) it only needs to wait for the threads ahead of it in the queue in addition to any current owner of the mutex before it will acquire the mutex. This is ensured by the *unlock* protocol that will activate the first thread in the m.waiting queue (lines U4-U5). A *trylock* operation cannot bypass the waiting threads since m.count is nonzero and *unlock* only sets the m.hand-off if it finds the waiting queue to be empty. □

Theorem 7.5.3 (Liveness I) *A thread τ waiting to acquire a mutex* m *will eventually acquire the mutex once its* lock *operation has enqueued τ on the* m.waiting *queue.*

Proof: The theorem follows from Lemma 7.5.1, Lemma 7.5.3, Lemma 7.5.4 and Lemma 7.5.5. □

Theorem 7.5.4 (Liveness II) *A thread τ wanting to acquire a mutex* m *can only be starved if there is an unbounded number of* lock *operations on* m *performed by threads on other processors.*

Proof: The theorem follows from the lock-free nature of the m.waiting queue and Theorem 7.5.3.

We know from Theorem 7.5.3 that once the thread has enqueued itself on the m.waiting queue it will not starve, so to starve it must not succeed to enter the m.waiting queue, that is, its enqueue operation must never complete.

Each time two or more operations on the lock-free queue interfere with each other, at least one of them make progress, so for one processor to never complete its operation, it will have to be interfered with by a concurrent successful operation every time it tries to progress. □

Theorem 7.5.5 (Fairness) *A thread τ wanting to acquire a mutex m will only have to wait for the other threads whose lock operation enqueued them on the $\mathsf{m.waiting}$ queue before τ did so.*

Proof: The theorem follows from Lemma 7.5.5. □

7.5.3 Linearizability

Theorem 7.5.6 *The mutex operations lock, $\mathsf{trylock}$ and unlock are atomic.*

Proof: A *lock* or *trylock* operation takes effect when the calling thread becomes the owner of the mutex. An *unlock* operation takes effect when the next waiting thread is activated or, if there is no waiting thread, when the mutex becomes unlocked. From Lemma 7.5.2 we know that no more than one thread can own the mutex at a time, and therefore any set of concurrent mutex operations will take effect one by one in a sequence. □

7.6 Experiments

The primary purpose of this work is to enhance qualitative properties of thread library implementations, such as the tolerance to delays and processor failures. However, since these properties may also provide a performance advantage with increasing number of processors, we also wanted to evaluate the performance impact of the lock-free mutex implementation and the lock-free thread library. We made an implementation on the GNU/Linux operating system. The implementation is written in the C programming language and was done entirely at the user-level using "cloned"[6] processes as *virtual processors* for running the threads. The implementation uses the lock-free queue by Tsigas and Zhang [TZ01b] for the Ready_Queue and the waiting queue in the mutex.

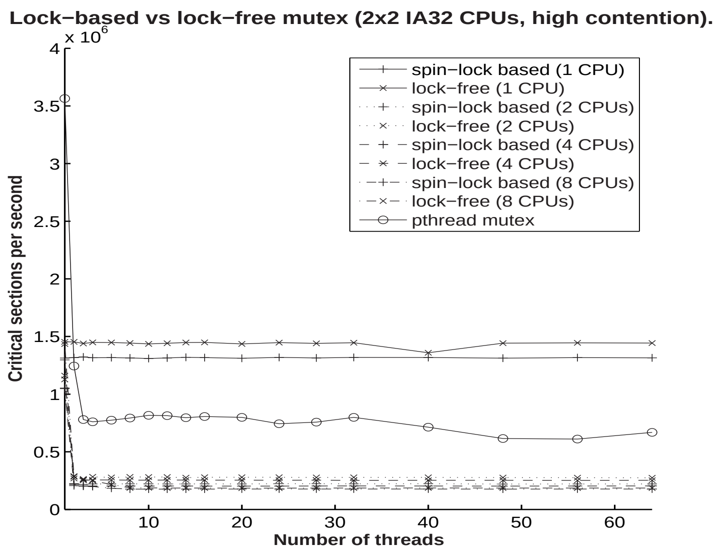
The experiments were run on a PC with two Intel Xeon 2.80GHz processors (acting as 4 due to hyper-threading) using the GNU/Linux operating system with kernel version 2.6.9.

The benchmark used for the experimental evaluation consists of a single critical section protected by a mutex and a set of threads that each try to enter the critical section a fixed number of times. To change the contention level on the mutex the amount of work done by each thread between accesses to the critical section can be changed.

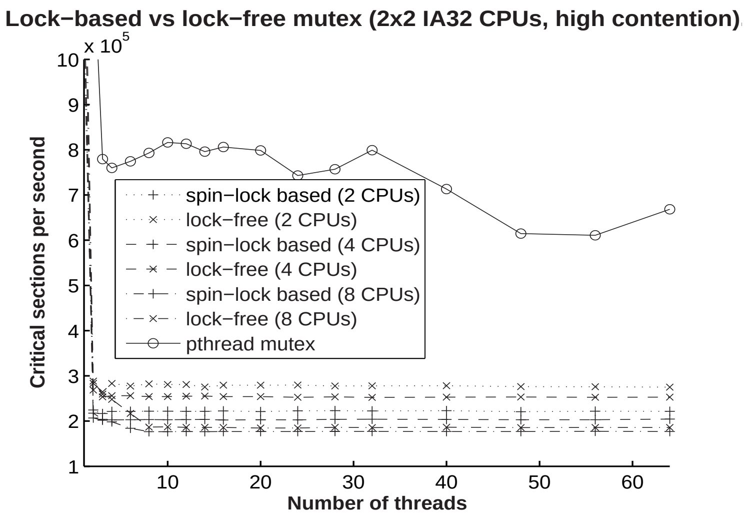
In the experimental evaluation we tested the following thread library configurations:

- The non-blocking thread library and a lock-free mutex using the protocol presented in this paper. This configuration was tested using 1, 2, 4 and 8 virtual processors to run the threads.

[6] "Cloned" processes share the same address space, file descriptor table and signal handlers etc and are also the basis of Linux's native pthread library implementation.

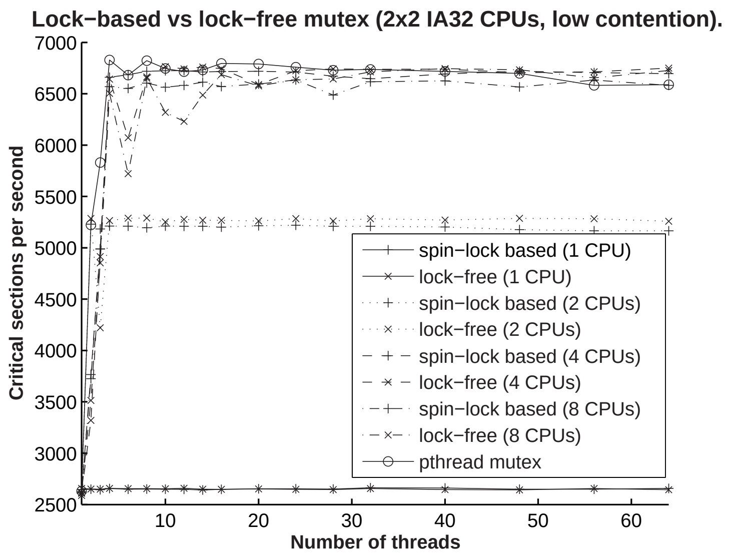


(a) High contention result.

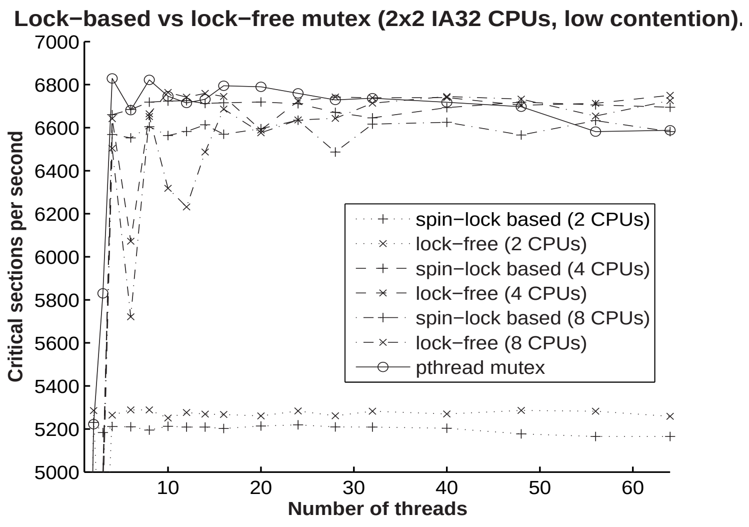


(b) Magnified view of the high contention result.

Figure 7.2: Mutex performance in LFthreads and pthreads at high contention.



(a) Low contention result.



(b) Magnified view of the low contention result.

Figure 7.3: Mutex performance in LFthreads and pthreads at low contention.

- The non-blocking thread library and a spin-lock based mutex. This configuration was tested using 1, 2, 4 and 8 virtual processors to run the threads.
- The platform's standard pthreads library and a standard pthread mutex. On GNU/Linux the pthreads library's threads are kernel-level "cloned" processes, which will be scheduled on all available processors, that is at the same level as the virtual processors in LFTHREADS. This makes it difficult to interpret the pthreads results with respect to the others and they should be considered to be primarily for reference.

Each test configuration was run 10 times. The diagrams present the mean of these 10 runs.

High contention In Figure 7.2 we show the results from a benchmark where all work is done inside the critical section, that is, the contention on the mutex will be high. In this case the desired result would be that the number of critical sections executed per second for an implementation stays the same regardless of the number of threads, as this should imply that the synchronization cost does not increase with the number of threads.

The number of processors, however, affect the synchronization overhead. In particular, going from a single processor to more than one processor for our thread library implies a cost since with more than one processor the thread contexts will have to be stored and restored much more often. (Note that threads currently use non-preemptive scheduling in our implementation so with only one virtual processor the threads will run to completion one after the other without any extra blocking.) The results show that with the same number of virtual processors the lock-free mutex has less overhead than the lock-based one (besides offering the qualitative advantages discussed earlier in the paper).

Low contention In Figure 7.3 we show the results from a benchmark where the threads perform 1000 times more work outside the critical section than inside, thus making the contention on the mutex low. With the majority of the work outside the critical section the expected behaviour is that the throughput increases linearly with the number of threads when there are fewer threads than (physical) processors and stays constant when the processors stay saturated with threads running outside the critical section. The results in Figure 7.3 are in agreement with the expected behaviour; we can see that going from one to two virtual processors doubles the throughput for both the lock-free and spin-lock based cases. (The fact that going to 4 virtual processors does not double the throughput yet again is due to the hyper-threading — there are not 4 physical processors available. This can also be seen from the behaviour of the pthread-based case.) Further, the lock-free mutex shows higher throughput than the spin-lock-based one using the same number of virtual processors; it also shows comparable and even better performance than pthread-based case when the number of threads are large and there are "enough" virtual processors (i.e. more than the number of physical processors).

Our experimental results show that for the LFTHREADS library and with the same number of virtual processors our lock-free mutex protocol has less overhead than a spin-lock-based one both in high and low contention scenarios. This is a very promising result as the lock-free protocol also offers better fault-tolerance properties.

7.7 Conclusions

We have presented the first lock-free thread library that provides a lock-free blocking synchronization primitive. The library is implemented entirely at the user-level without any need for modifications to the operating system kernel, using processes as virtual processors. However, the principles behind the library could also be applied together with scheduler activations, which we believe would be a very good match, as well as directly at the hardware-level, where they could form the basis for a fully lock-free implementation of a multiprogrammed multiprocessor kernel.

We have implemented the library on a PC multiprocessor platform with two Intel Xeon processors running the GNU/Linux operating system. This implementation constitutes a proof-of-concept of the lock-free blocking primitive introduced in the paper and serves as basis for an experimental study of its performance. The experimental study performed here, using an intensive mutex-benchmark, shows very positive and promising performance figures. Moreover, this implementation can also serve as basis for further development, for porting the library to other multiprocessors such as SGI IRIX/MIPS (e.g. for running tests on a large SGI NUMA multiprocessor machine) and experimenting with parallel applications such as the Spark98 matrix kernels or applications from the SPLASH-2 suite.

Chapter 8

Multi-word Atomic Read/Write Registers on Multiprocessor Systems[1]

Andreas Larsson Anders Gidenstam Phuong H. Ha
Marina Papatriantafilou Philippas Tsigas

Abstract

Modern multiprocessor systems offer advanced synchronization primitives, built in hardware, to support the development of efficient parallel algorithms. In this paper we develop a simple and efficient algorithm for atomic registers (variables) of arbitrary length. The simplicity and better complexity of the algorithm is achieved via the utilization of two such common synchronization primitives. In this paper we also evaluate the performance of our algorithm and the performance of a practical previously know algorithm that is based only on read and write primitives. The evaluation is performed on three well-known parallel architectures. This evaluation clearly shows that both algorithms are practical and that as the size of the register increases our algorithm performs better, accordingly to its complexity behavior.

Keywords: atomic register, synchronization, wait-free.

[1]This work was supported by computational resources provided by the Swedish National Supercomputer Centre (NSC) and is an extended version of the paper that appeared in the Proceedings of ESA 2004, Bergen, Norway, 14-17 September, 2004.

8.1 Introduction

In multiprocessor and multiprocessing systems cooperating processes may share data via shared data objects. In this paper we are interested in designing and evaluating the performance of shared data objects for cooperative tasks in multiprocessor systems. More specifically we are interested in designing a practical wait-free algorithm for implementing registers (or memory words) of arbitrary length that could be read and written atomically. (Typical modern multiprocessor systems support words of 64-bit size.)

The most commonly required consistency guarantee for shared data objects is *atomicity*, also known as *linearizability* [HW90]. An implementation of a shared object is *atomic* or *linearizable* if it guarantees that even when operations overlap in time, each of them appears to take effect at an atomic time instant that lies in its respective time duration, in a way that the effect of each operation is in agreement with the object's sequential specification. The latter means that if we speak of e.g. read/write objects, the value returned by each read equals the value written by the most recent write according to the sequence of "shrunk" operations in the time axis.

The classical, well-known and simplest solution for maintaining consistency of shared data objects enforces mutual exclusion. Mutual exclusion protects the consistency of the shared data by allowing only one process at time to access it. Mutual exclusion causes large performance degradation especially in multiprocessor systems [SGG05, Sun04a] and suffers from potential priority inversion in which a high priority task can be blocked for an unbounded time by a lower priority task [SRL90]. Several synchronization protocols have been introduced to solve the priority inversion problem for uniprocessor [SRL90] and multiprocessor [Raj90] systems. The solution presented in [SRL90] solves the problem for the uniprocessor case with the cost of limiting the schedulability of task sets and also making the scheduling analysis of real-time systems hard, but the situation is much worse in a multiprocessor real-time system, where a task may be blocked by another task running on a different processor [Raj90].

Non-blocking implementation of shared data objects is an alternative approach for the problem of inter-task communication. Non-blocking mechanisms allow multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion to accomplish this. They offer significant advantages over lock-based schemes because (i) they do not suffer from priority inversion; (ii) they avoid lock convoys; (iii) they provide high tolerance to process failures (processes or processor stop failures will never corrupt shared data objects); and (iv) they eliminate deadlock scenarios involving two or more tasks both waiting for locks held by the other. On the other hand non-blocking protocols have to use more delicate strategies to guarantee data consistency than the simple enforcement of mutual exclusion between the readers and the writers of the data object.

Non-blocking algorithms can be lock-free or wait-free. *Lock-free* implementations guarantee that regardless of the contention and the interleaving of concurrent operations, at least one operation will always make progress. However, there is a risk that the progress of other operations might cause one specific operation to take unbounded time to finish. In a *wait-free* [Her91] algorithm, every operation is guaranteed to finish in a limited number of steps, regardless of the actions of the concurrent operations. Non-blocking algorithms have been shown to be of big practical importance [TZ01a, TZ02], and recently NOBLE, which is a non-blocking inter-process communication library, has been introduced [ST02].

From a historic perspective research on non-blocking algorithms stems from the *readers/writers problem.* In this problem a number of concurrent processes are interested in reading from or writing to a shared data object (here also called a *register*). A read operation as well as a write operation should take effect atomically and return or update the entire state of the shared data object. When the shared data object is larger than a single word (of the word size supported by the multiprocessor system at hand) a software algorithm is needed to solve the readers/writers problem. The classical solution is to use mutual exclusion to enforce that either (i) no read or write operations overlap each other; or (ii) no write operations overlap each other or any read operation. These methods, normally implemented using a *mutual exclusion lock* or a *readers-writers lock*, respectively, suffer from the drawbacks of mutual exclusion mentioned above. In [Lam77] Lamport introduced a lock-free solution to the readers/writers problem with one writer. Lamport's algorithm is actually wait-free for the writer but lock-free for the readers since the writer can force a slow reader to retry indefinitely. This algorithm followed by the first wait-free algorithm by Peterson [Pet83] marked the start of long running research efforts to construct wait-free solutions to the readers/writers problem.

This problem, also known as the problem of multi-word wait-free read/write registers, has become one of the well-studied problems in the area of non-blocking synchronization, with numerous results for the construction of e.g.: (i) single-writer single-reader registers [Lam86, Sim90, CB97]; (ii) single-writer n-reader registers [Pet83, BP87, KKV87, NW87, KR93, SAG94, HV95, LGH$^+$04]; (iii) 2-writer n-reader registers [Blo88]; and (iv) m-writer n-reader registers [VA86, PB87, IS92, LV92, LTV96, HV96].

The main goal of the algorithms in the above results is to construct wait-free multi-word read/write registers using single-word read/write registers and not other synchronization primitives that may be provided by the hardware in a system. This has been very significant, providing fundamental results in the area of wait-free synchronization, especially when we consider the nowadays well-known and well-studied hierarchy of shared data objects and their synchronization power [Her91]. Many of these solutions involve elegant and symmetric ideas. Moreover, they have formed the basis for further results in the area of non-blocking synchronization.

Our motivation for further studying this problem is as follows: As the aforementioned solutions were using only read/write registers as components, they necessarily have each write operation on the multi-word register write the new value in several copies (roughly speaking, as many copies as we have readers in the system), which may be costly. However, modern architectures provide

```
int swap(int *mem, int new)
{
        tmp := *mem;
        *mem := new;
        return tmp;
}
```

```
int fetch_and_or(int *mem,
                 int arg)
{
        tmp := *mem;
        *mem := tmp | arg;
        return tmp;
}
```

Figure 8.1: The specifications of the *swap* and *fetch_and_or* atomic suboperations.

hardware synchronization primitives stronger than atomic read/write registers, some of them even accessible at a constant cost-factor away from read/write accesses. We consider it a useful task to investigate how to use this power, to the benefit of designing economical solutions for the same problem, which can lead to structures that are more suitable in practice. Moreover, to the best of our knowledge, none of the previous solutions has been implemented and evaluated on real systems.

In this paper we present a simple, efficient wait-free algorithm for implementing multi-word n-reader/single-writer registers of arbitrary word length. In the new algorithm each multi-word write operation only needs to write the new value in one copy, thus having significantly less overhead. The new algorithm uses synchronization primitives called *fetch_and_or* and *swap* (c.f. Figure 8.1 and [SGG05]), which are available in several modern processor architectures, to synchronize n readers and a writer accessing the register concurrently. Since the new algorithm is wait-free, it provides high parallelism for the accesses to the multi-word register and thus significantly improves performance. We compare the new algorithm with the wait-free one in [Pet83], which is also practical and simple, and two lock-based algorithms, one using a single spin-lock and one using a readers-writer spin-lock. We design benchmarks to test them on three different architectures: UMA Sun-Fire-880 with 6 processors, ccNUMA SGI Origin 2000 with 29 processors and ccNUMA SGI Origin 3800 with 128 processors.

The rest of this paper is organized as follows. In Section 8.2 we describe the formal requirements of the problem and the related algorithms that we are using in the evaluation study. Section 8.3 presents our protocol. In Section 8.4, we give the proof of correctness and complexity of the new protocol. Section 8.5 is devoted to the experimental study comparing our non-blocking protocol with previous work, both non-blocking and lock-based. The paper concludes with Section 8.6, with a discussion on the contributed results and further research issues.

8.2 Background

System and Problem Model

A *shared register* of arbitrary length [Pet83, Her93] is an abstract data structure that is shared by a number of concurrent processes, which perform read or write operations on the shared register. In this paper we make no assumption about the relative speed of the processes, i.e. the processes are asynchronous. One of the processes, the *writer*, executes write operations and all other processes, the *readers*, execute read operations on the shared register. Operations performed by the same process are assumed to execute sequentially.

An implementation of a register consists of: (i) *protocols* for executing the operations (read and write); (ii) a data structure consisting of shared *subregisters* and (iii) a set of initial values for these. The protocols for the operations consist of a sequence of operations on the subregisters, called *suboperations*. These suboperations are reads, writes or other atomic operations, such as *fetch_and_or* or *swap* (cf. Figure 8.1), which are either available directly on modern multiprocessor systems or can be implemented from other available synchronization primitives, such as *compare_and_swap* or *load_linked/store_conditional* [Her91]. Furthermore, matching the capabilities of modern multiprocessor systems, the subregisters are assumed to be atomic and to support multiple processes.

A register implementation is *wait-free* [Her91] if it guarantees that any process will complete each operation in a finite number of steps (suboperations) regardless of the execution speeds of the other processes.

For each operation O there exists a time interval $[s_O, f_O]$ called its *duration*, where s_O and f_O are the starting and ending times, respectively. There is a precedence relation on the operations that form a strict partial order (denoted '$\rightarrow$'). For two operations a and b, $a \rightarrow b$ means that operation a ended before operation b started. If two operations are incomparable under $\rightarrow$, they are said to *overlap*.

A *reading function* π for a register is a function that assigns a high-level write operation w to each high-level read operation r such that the value returned by r is the value that was written by w (i.e. $\pi(r)$ is the write operation that wrote the value that the read operation r read and returned).

Criterion 8.2.1 *A shared register is* atomic *iff the following three conditions hold for all possible executions:*

1. **No-irrelevant**. *There exists no read r such that $r \rightarrow \pi(r)$.*

2. **No-past**. *There exists no read r and write w such that $\pi(r) \rightarrow w \rightarrow r$.*

3. **No N-O inversion**. *There exist no reads r_1 and r_2 such that $r_1 \rightarrow r_2$ and $\pi(r_2) \rightarrow \pi(r_1)$.*

Variable	**Type**	**Description**
WFLAG	Boolean	Indicates that the writer is writing in BUF1.
SWITCH	Boolean	To check if the writer has written in BUF1.
READING	Array of n Boolean	Used together with WRITING to handle reader-writer handshaking.
WRITING	Array of n Boolean	Used together with READING to handle reader-writer handshaking.
BUF1	Buffer	Main buffer.
BUF2	Buffer	Backup buffer.
COPYBUF	Array of n buffers	An individual backup buffer for each reader.

Figure 8.2: The shared variables used by Peterson's algorithm. The number of readers is n. BUF1 holds the initial register value. All other variables are initialized to 0 or false.

Algorithm 8.1 Peterson's algorithm. Lower-case variables are local variables.

```
Read operation by reader r:
PR1   READING[r] := !WRITING[r];
PR2   flag1 := WFLAG;
PR3   sw1 := SWITCH;
PR4   read BUF1;
PR5   flag2 := WFLAG;
PR6   sw2 := SWITCH;
PR7   read BUF2;
PR8   if (READING[r] == WRITING[r])
PR9       return the value in COPYBUF[r];
PR10  else if ((sw1 != sw2) || flag1 || flag2)
PR11      return the value read from BUF2;
PR12  else
PR13      return the value read from BUF1;

Write operation:
PW1   WFLAG := true;
PW2   write to BUF1;
PW3   SWITCH := !switch;
PW4   WFLAG := false;
PW5   for (each reader r)
PW6       if (READING[r] != WRITING[r])
PW7           write to COPYBUF[r];
PW8           WRITING[r] := READING[r];
PW9   write to BUF2;
```

Peterson's Shared Multi-Word Register

In [Pet83] Peterson describes an implementation of an atomic shared multi-word register for one writer and many readers. The protocol does not use any other atomic suboperations than reads and writes and is described below.

The idea is to use $n+2$ shared buffers, each of which can hold a value of the register, together with a set of shared handshake variables to make sure that the writer does not overwrite all buffers that is being read by a reader and that each reader chooses a stable but up-to-date buffer to read from. The shared variables are shown in Figure 8.2 and the protocols for the read and write operations are shown in Algorithm 8.1.

In the algorithm each reader r have three choices of where to get the current register value, namely BUF1, BUF2 and COPYBUF[r]. The primary alternative for a reader is BUF1 which it reads at line PR4. However, if there is a concurrent write, the writer might write to BUF1 (line PW2) while the reader is reading the same buffer. The flags WFLAG and SWITCH are used to detect most of these conflicts. A reader reads WFLAG and SWITCH both before (lines PR2 and PR3) and after (lines PR5 and PR6) it reads BUF1. The writer sets WFLAG to true (line PW1) before it begins to write to BUF1 (line PW2) and flips SWITCH (line PW3) and resets WFLAG to false (line PW4) after it has updated BUF1. Through this handshake mechanism a reader can detect if (i) the writer was writing to BUF1 when it started to read it, (ii) the writer was writing to BUF1 when it finished reading that buffer, or (iii) the writer did a complete write of BUF1 (i.e. lines PW1 to PW4) while the reader was reading that buffer (in this case SWITCH has been flipped). In all these cases the reader cannot assume it managed to read BUF1 correctly.

However, by using WFLAG and SWITCH alone, a reader cannot detect the case where the writer managed to complete an even number of writes to BUF1, since in those cases SWITCH will be back to its original value. The per-reader flags READING[r] and WRITING[r] are used to handle these cases. A reader r makes READING[r] equal to the writer's corresponding flag WRITING[r] at the beginning of each read operation (line PR1). This is a signal to the writer that it should write the register value also to COPYBUF[r] (lines PW6 and PW7). When the writer has written COPYBUF[r] it sets WRITING[r] equal to READING[r] (line PW8) to signal the reader that COPYBUF[r] now contains a valid register value. If the reader detects this signal (line PR8), i.e. when the writer performed lines PW6 to PW8 after the reader performed line PR1 but before it reached line PR8, then the reader can safely return the value in COPYBUF[r] as the current register value.

If the reader didn't detect this signal, then it can examine (line PR10) the values it read from WFLAG and SWITCH to determine whether it read a correct value from BUF1 or not. This test (line PR10) is now safe, since to fool it the writer must have flipped SWITCH (line PW3) at least twice since the reader was at line PR3 and in that case the writer should have written to COPYBUF[r] and signaled the reader (at line PR8) as described above.

If the test at line PR10 indicates that the reader did not read BUF1 correctly

Algorithm 8.2 A spin-lock algorithm with exponential back-off.

```
spin_lock(int *lock)
    backoff := 1;
    while (swap(lock, 1))
        backoff := 2 * backoff;
        spin for backoff iterations;

spin_unlock(int *lock)
    *lock := 0;
```

then the reader can safely return the value it read from BUF2 (line PR7) instead. The value read from BUF2 is safe because for the writer to interfere with both the reading of BUF1 (line PR4) and the reading of BUF2 (line PR7) the writer must execute both its write of BUF1 (line PW1) and its write of BUF2 (line PW9) before the reader reached line PR8, and in that case it should have written to COPYBUF[r] and signaled the reader (at line PR8) as described above.

Peterson's implementation is simple and efficient in most cases, however, a high-level write operation potentially has to write $n+2$ copies of the new value and all high-level reads read at least two copies of the value, which can be quite expensive when the register is large. While it is unlikely that one can do better using only read/write subregisters, most modern systems support additional atomic suboperations, such as *swap*. Our new register implementation uses these additional suboperations to implement high-level read and write operations that only need to read or write one copy of the register value.

We have decided to compare our method with this algorithm because: (i) they are both designed for the 1-writer n-reader shared register problem; (ii) compared to other more general solutions based on weaker subregisters (which are much weaker than what common multiprocessor machines provide) this one involves the least communication overhead among the processes, without requiring unbounded timestamps or methods to bound the unbounded version (cf. [HV02, IL93] for examples of such methods).

Mutual-Exclusion Based Solutions

For comparison we also evaluate the performance of two mutual-exclusion-based register implementations, one that uses a single *spin-lock* with exponential back-off (see Algorithm 8.2) and another that uses a *readers-writers spin-lock* (see Algorithm 8.3 and [SGG05]) with exponential back-off to protect the shared register. The readers-writers spin-lock is similar to the spin-lock but allows readers to access the register concurrently with other readers.

8.3 The New Algorithm

The idea of the new algorithm is to remove the need for reading and writing several buffers during read and write operations by utilizing the atomic

Algorithm 8.3 A readers-writers spin-lock algorithm with exponential back-off.

```
struct rwlock_t
    int lock
    int rlock
    int readers

writer_lock(rwlock_t *lock)
    spin_lock(&lock−>lock);

writer_unlock(rwlock_t *lock)
    spin_unlock(&lock−>lock);

reader_lock(rwlock_t *lock)
    spin_lock(&lock−>rlock);
    lock−>readers := lock−>readers + 1;
    if (lock−>readers == 1)
        spin_lock(&lock−>lock);
    spin_unlock(&lock−>rlock);

reader_unlock(rwlock_t *lock)
    spin_lock(&lock−>rlock);
    lock−>readers := lock−>readers - 1
    if (lock−>readers == 0)
        spin_unlock(&lock−>lock);
    spin_unlock(&lock−>rlock);
```

synchronization primitives available on modern multiprocessor systems. These primitives are used for the communication between the readers and the writer. The new algorithm uses $n + 2$ shared buffers that each can hold a value of the register. The number of buffers is the same as for Peterson's algorithm, which matches the lower bound on the required number of buffers. Informally the argument is that the number of buffers cannot be less for any wait-free implementation because each of the n readers may be reading from one buffer concurrently with a write, and the write should not overwrite the last written value (since one of the readers might start to read again before the new value is completely written).

The shared variables used by the algorithm are presented in Figure 8.3. The shared buffers are in the $(n+2)$-element array BUF. The atomic variable SYNC is used to synchronize the readers and the writer. This variable consists of two fields: (i) the *pointer field*, which contains the index of the buffer in BUF that contains the most recent value written to the register and (ii) the *reading-bit field*, which holds a *handshake* bit for each reader, each such bit is set when the corresponding reader has read the value presently contained in the pointer field.

A reader (Algorithm 8.4) uses *fetch_and_or* to atomically read the value of SYNC and set its reading-bit. Then it reads the value from the buffer pointed to by the pointer field.

The writer (Algorithm 8.4) needs to keep track of the buffers that are available for use. To do this it stores the index of the buffer where it last saw each reader, in a n-element array trace, in persistent local memory. At the beginning of each write the writer selects a buffer index to write to. This buffer should be different from the last one it used and with no reader intending to use it. The writer writes the new value to that buffer and then uses the suboperation *swap* to atomically read SYNC and update it with the new buffer index and cleared reading-bits. The old value read from SYNC is then used to update the trace

Constants	**Description**
PTRFIELDLEN	The number of bits used for the pointer field indexing the most recently update buffer.
PTRFIELD	A bitmask containing 1's in the PTRFIELDLEN least significant bits.

Variable	**Type**	**Description**
BUF$[n+2]$	Array of $n+2$ buffers	The buffers for the register value.
SYNC	Unsigned word	Consists of two fields.
bit 0 .. PTRFIELDLEN-1 of SYNC	Integer	Index of the buffer with the most recent register value.
bit PTRFIELDLEN $+ r$ of SYNC	Bit	The reading bit for reader r.

Figure 8.3: The constants and shared variables used by the new algorithm. The number of readers is n. Initially BUF[0] holds the register value and SYNC points to this buffer while all reader-bits are 0.

array for those readers whose reading-bit was set.

The maximum number of readers is limited by the size of the words that the two atomic primitives used can handle. If we are limited to 64-bit words we can support 58 readers as 6 bits are needed for the pointer field to be able to distinguish between 58+2 buffers.

8.4 Analysis

We first prove that the new algorithm satisfies the conditions in Lamport's criterion [Lam86] (cf. Criterion 8.2.1 in section 8.2), which guarantee atomicity.

Lemma 8.4.1 *The new algorithm satisfies condition* "No-irrelevant"

Proof:
A read r reads the value that is written by the write $\pi(r)$. Therefore r's $read(BUF[j])$ operation (line R4 in Fig. 8.4) starts after $\pi(r)$'s $write(BUF[j])$ operation (line W2 in Fig. 8.4) starts. On the other hand, the starting time-point of the $read(BUF[j])$ operation is before the ending time-point of r and the starting time-point of the $write(BUF[j])$ operation is after the starting time-point of $\pi(r)$, so the ending time-point of r must be after the starting time-point of $\pi(r)$, or $r \not\to \pi(r)$. □

Lemma 8.4.2 *The new algorithm satisfies condition* "No-past"

Proof: We prove the lemma by contradiction. Assume there are a read r and a write w such that $\pi(r) \to w \to r$, which means (i) write $\pi(r)$ ends before

Algorithm 8.4 The read and write operations of the new algorithm. The trace-array and oldwptr are static, i.e. stay intact between write operations. They are both initialized to zero.

```
Read operation by reader r:
R1    readerbit := 1 << (r + PTRFIELDLEN);
R2    rsync := fetch_and_or(&SYNC, readerbit);
R3    rptr := rsync & PTRFIELD;
R4    read BUF[rptr]

Write operation:
W1    choose newwptr such that newwptr != oldwptr and
          newwptr != trace[r] for all r;
W2    write BUF[newwptr];
W3    wsync := swap(&SYNC, 0 | newwptr); /* Clears all reading bits */
W4    oldwptr := wsync & PTRFIELD;
W5    for each reader r
W6        if (wsync & (1 << (r + PTRFIELDLEN)))
W7            trace[r] := oldwptr;
```

write w starts and write w ends before read r starts and (ii) r reads the value written by $\pi(r)$. Because $\pi(r) \rightarrow w \rightarrow r$, the value of $SYNC$ that r reads (line R2 in Fig. 8.4) is written by a write w' using the *swap* primitive (line W3 in Fig. 8.4), where $w' = w$ or $w \rightarrow w' \rightarrow r$, i.e. $w' \neq \pi(r)$ because $\pi(r) \rightarrow w$. On the other hand, because r reads the buffer that is pointed by $SYNC$ (lines R2-R4), r would read the buffer that has been written completely by $w' \neq \pi(r)$. That means r does not read the value written by $\pi(r)$, a contradiction. $\square$

Lemma 8.4.3 *The new algorithm satisfies condition* "No N-O inversion"

Proof: We prove the lemma by contradiction. Assume there are reads r_1 and r_2 such that $r_1 \rightarrow r_2$ and $\pi(r_2) \rightarrow \pi(r_1)$. Because (i) $\pi(r_1)$ always keeps track of which buffers are used by readers in the array *trace*[] (lines W4-W7 in Fig. 8.4) and (ii) the buffer $\pi(r_1)$ chooses to write to is different from those recorded in *trace*[] as well as the last buffer the writer has written, *wptr* (lines W1-W2), r_1 reads the value written by $\pi(r_1)$ only if r_1 has read the correct value of $SYNC$ (line R2 in Fig. 8.4) that has been written by $\pi(r_1)$ (line W3 in Fig. 8.4).

On the other hand, because $r_1 \rightarrow r_2$, the value of $SYNC$ that r_2 reads must be written by a write w_k where $w_k = \pi(r_1)$ or $\pi(r_1) \rightarrow w_k$, i.e. $w_k \neq \pi(r_2)$ because $\pi(r_2) \rightarrow \pi(r_1)$. Moreover, because r_2 reads the buffer that is pointed to by $SYNC$ (lines R2-R4), r_2 would read the buffer that has been written completely by $w_k \neq \pi(r_2)$ (lines W2-W3). That means r_2 reads the value that has not been written by $\pi(r_2)$, a contradiction. $\square$

With the correctness of the proposed wait-free algorithm established, let us focus on its complexity, also in comparison with Peterson's wait-free algorithm.

Complexity

The complexity of a read operation is of order $O(m)$, where m is the size of the register, for both the new algorithm and Peterson's algorithm [Pet83]. However, in Peterson's algorithm the reader may have to read the value up to 3 times, while in our algorithm the reader will only read the value once but has to use the *fetch_and_or* suboperation once. A write operation in the new algorithm writes one value of size m and then traces the n readers. The complexity of the write operation is therefore of order $O(n+m)$. For Peterson's algorithm however, the writer must, in the worst case, write to $n+2$ buffers of size m, thus its complexity is of order $O(n \cdot m)$. As the size of registers and the number of threads increase, the new algorithm is expected to perform significantly better than Peterson's algorithm with respect to the writer. With respect to the readers the handshake mechanism used in the new algorithm can be more expensive compared to the one used by Peterson's, but on the other hand the new algorithm only needs to read one m-word buffer, whereas Peterson's need to read at least two and sometimes three buffers.

Using the above, we have the following theorem:

Theorem 8.4.1 *A multi-reader, single-writer, m-word sized register can be constructed using $n+2$ buffers of size m each. The complexity of a read operation is $O(m)$. The complexity of a write operation is $O(n+m)$.*

8.5 Performance Evaluation

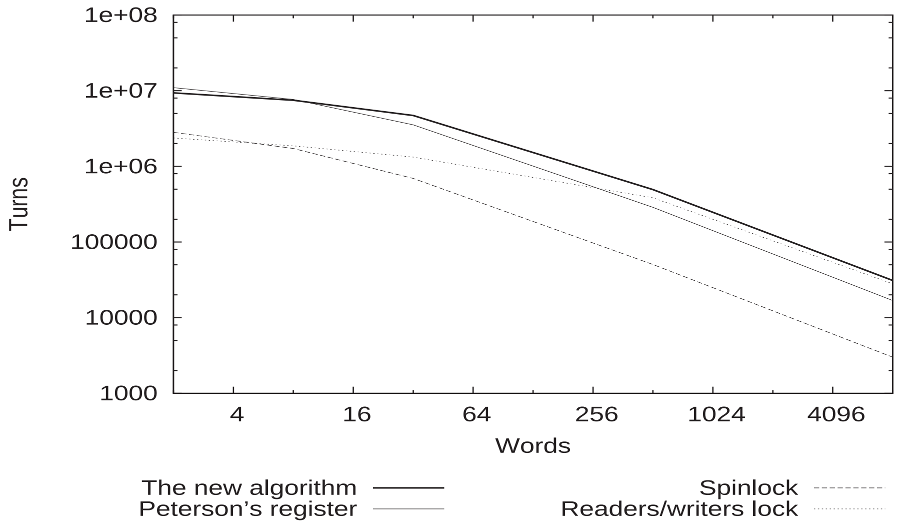
The performance of the proposed algorithm was tested against: (i) Peterson's algorithm [Pet83], (ii) a spinlock-based implementation with exponential backoff and (iii) a readers-writers spinlock with an exponential backoff (cf. Section 8.2 for descriptions of the respective algorithms).

Method

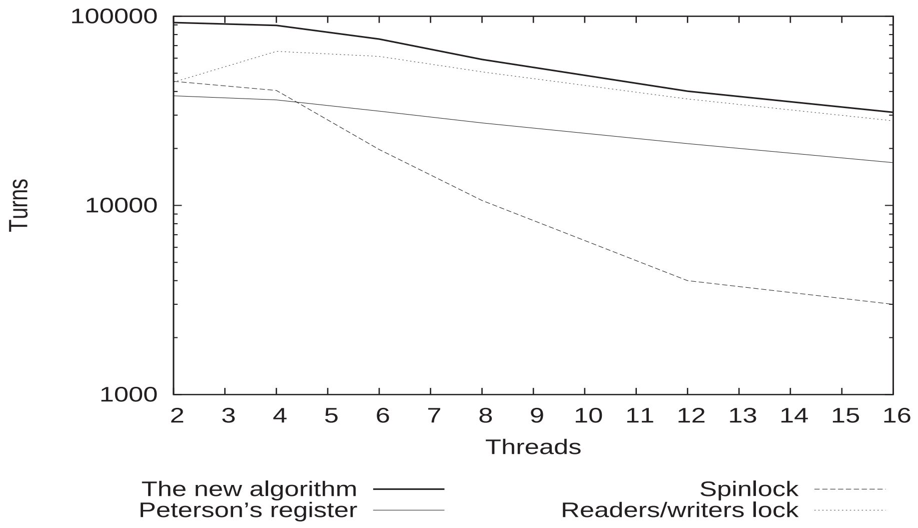
We measured the number of successful read and write operations during a fixed period of time. The higher this number the better the performance. In each test one thread is the writer and the rest of the threads are the readers. Two sets of tests have been done: (i) one set with *low contention* and (ii) one set with *high contention*. During the high-contention tests each thread reads or writes continuously with no delay between successive accesses to the multi-word register. During the low-contention tests each thread waits for a time-interval between successive accesses to the multi-word register. This time interval is much longer than the time used by one write or read. Tests have been performed for different number of threads and for different sizes of the register.

Systems

The performance of the new algorithm has been measured on both UMA (Uniform Memory Architecture) and NUMA (Non Uniform Memory Architecture)

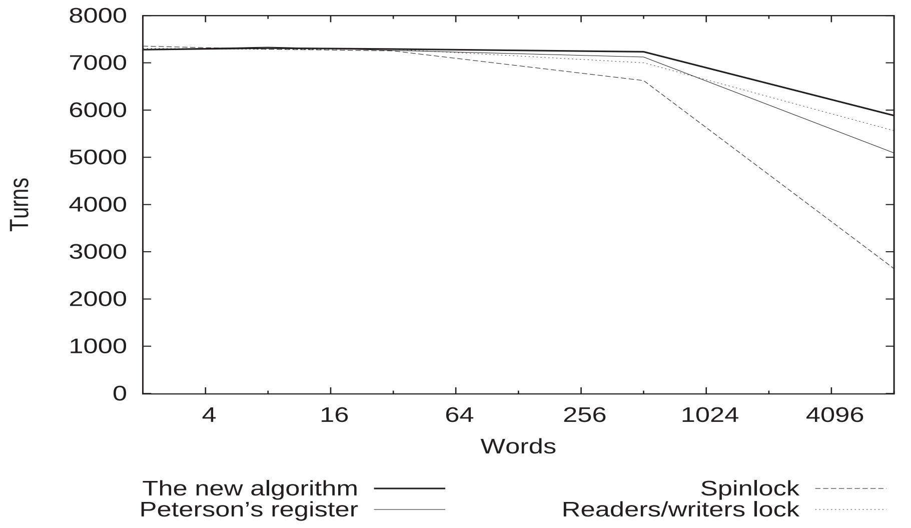


(a) 16 threads, high contention.

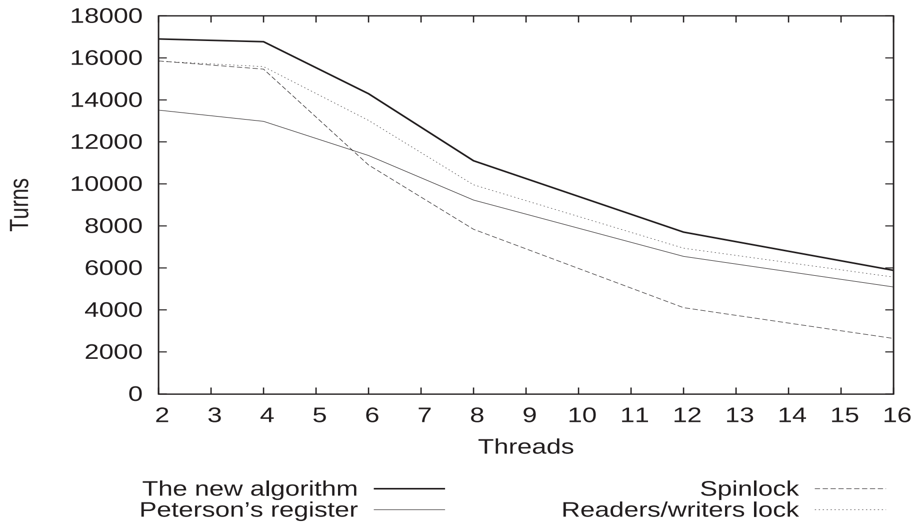


(b) 8192 word register, high contention.

Figure 8.4: Average number of reads or writes per thread on the UMA SunFire 880 at high contention.

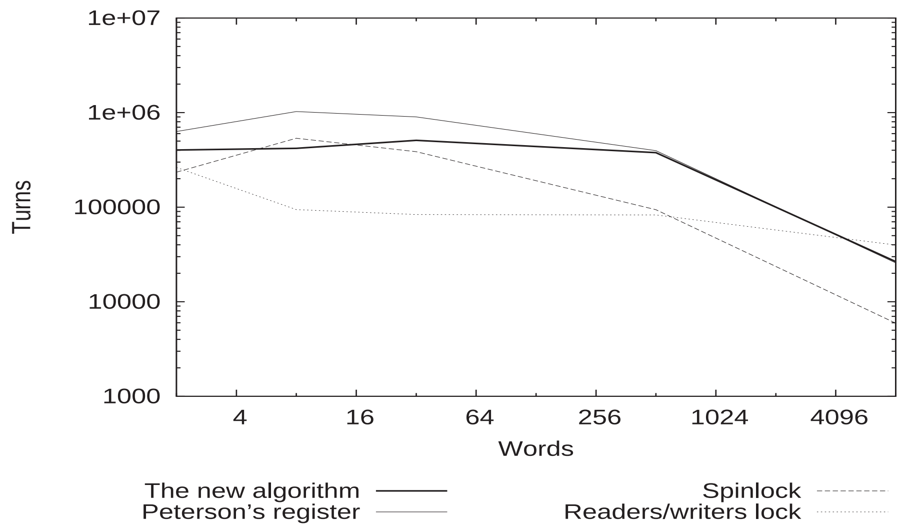


(a) 16 threads, low contention.

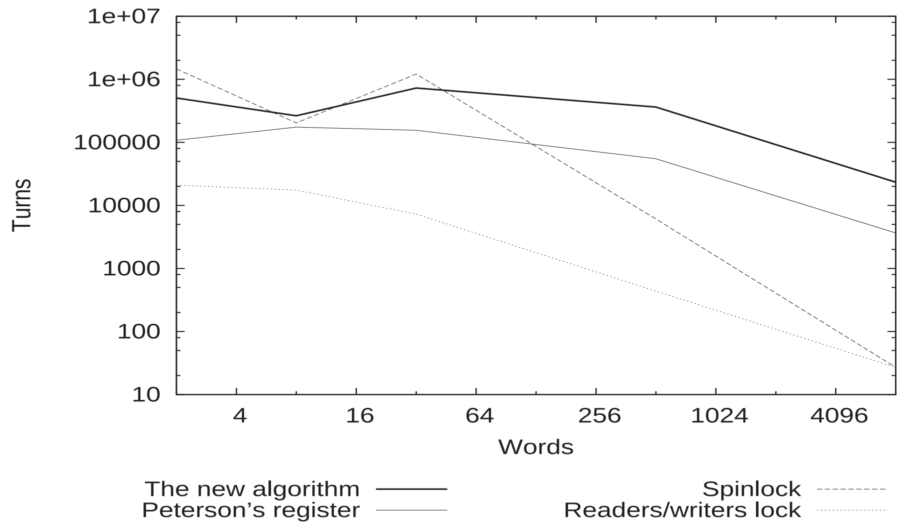


(b) 8192 word register, low contention.

Figure 8.5: Average number of reads or writes per thread on the UMA SunFire 880 at low contention.

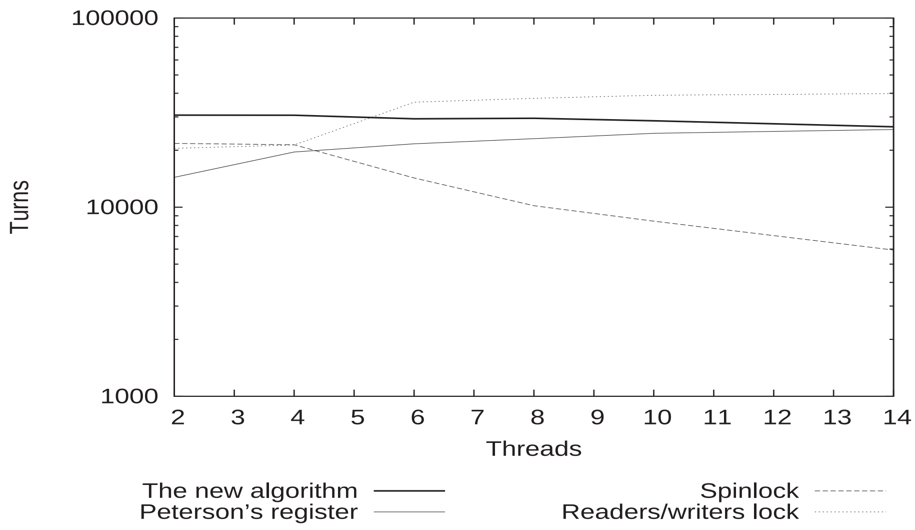


(a) All operations (readers and writer).

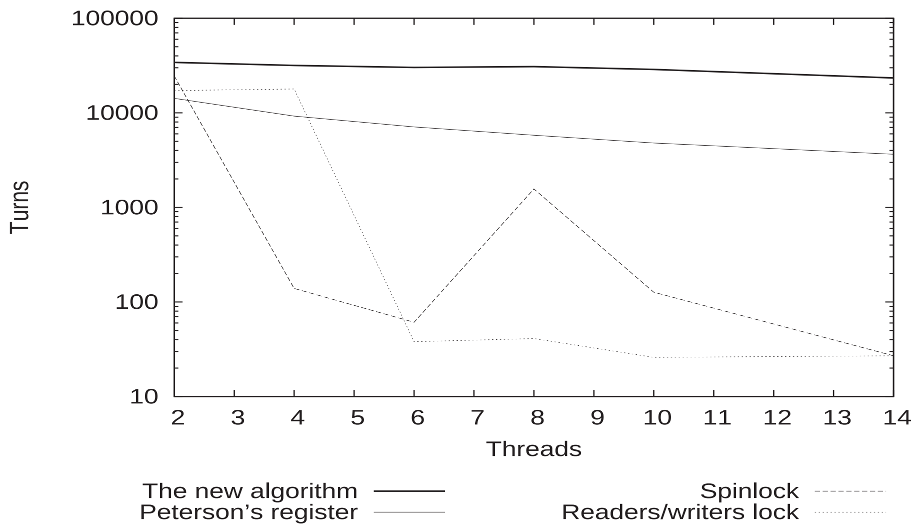


(b) Operations by the writer.

Figure 8.6: Average number of operations per thread with 14 threads and high contention on NUMA Origin 2000.

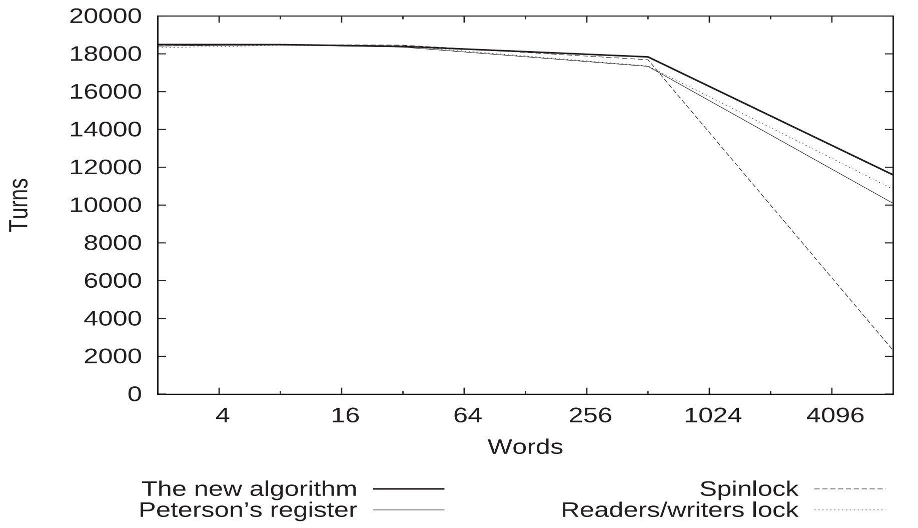


(a) All operations (readers and writer).

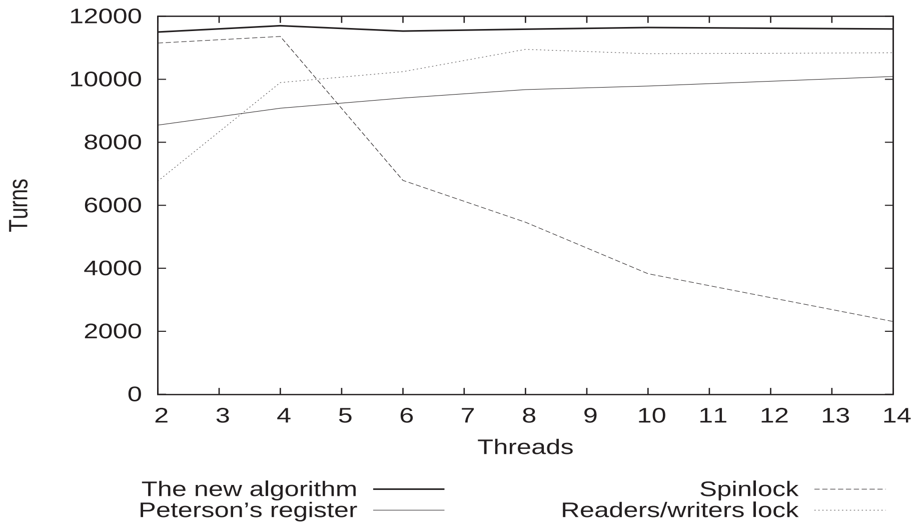


(b) Operations by the writer.

Figure 8.7: Average number of operations per thread with a register size of 8192 words and high contention on NUMA Origin 2000.

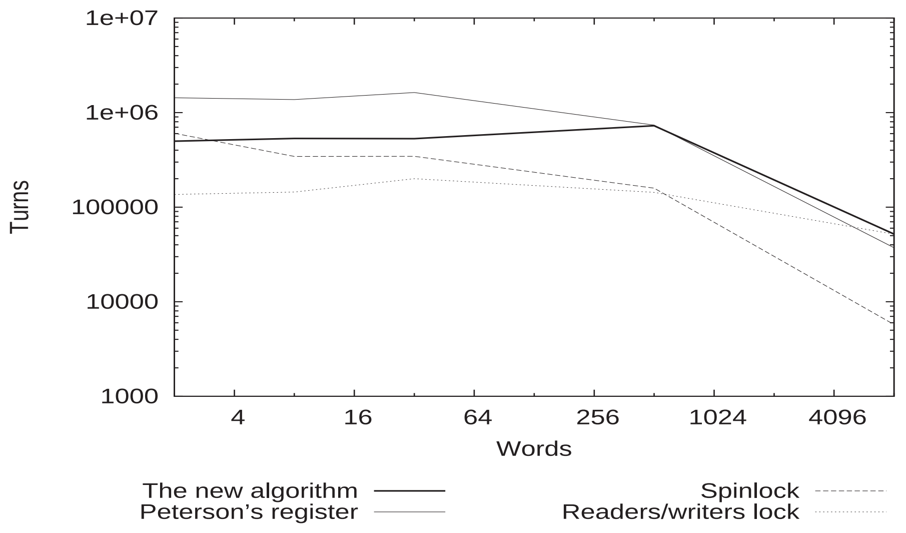


(a) Run with 14 threads

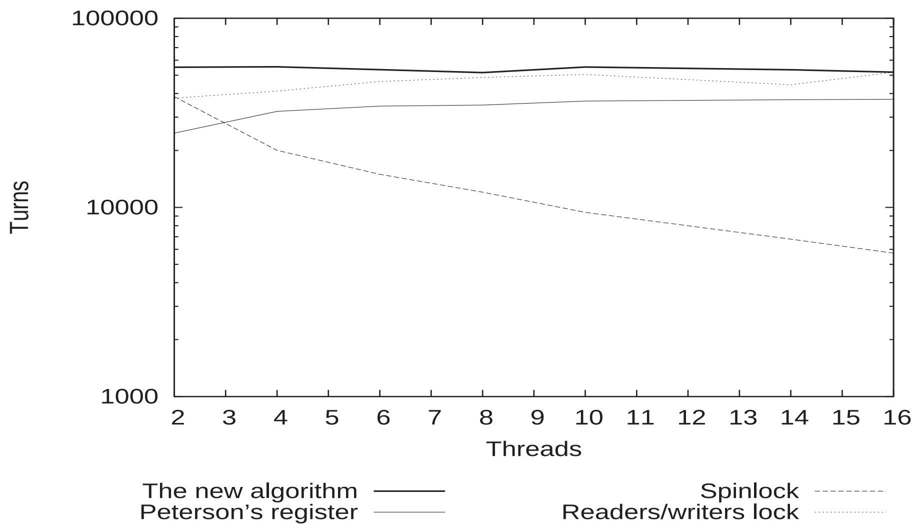


(b) Run with 8192 word register

Figure 8.8: Average number of operations per thread with low contention on NUMA Origin 2000.

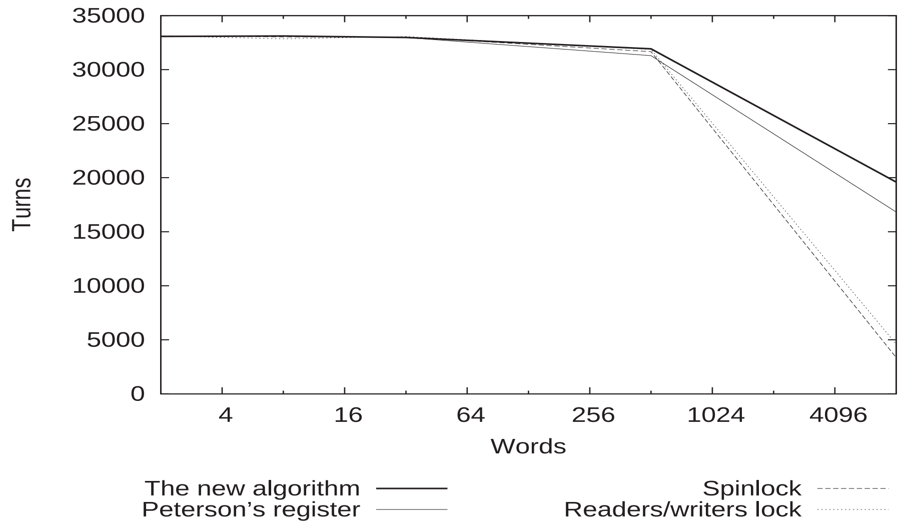


(a) 16 threads, high contention.

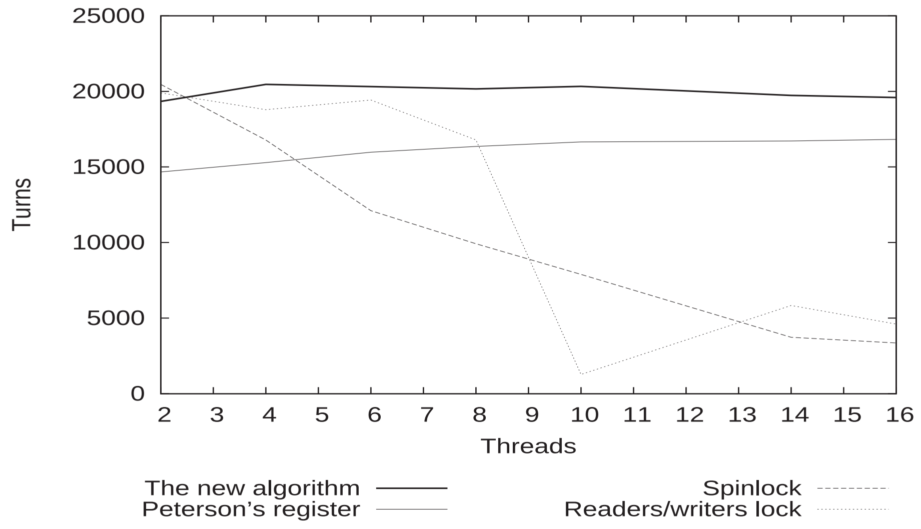


(b) 8192 word register, high contention.

Figure 8.9: Average number of reads or writes per thread on NUMA Origin 3800 at high contention.

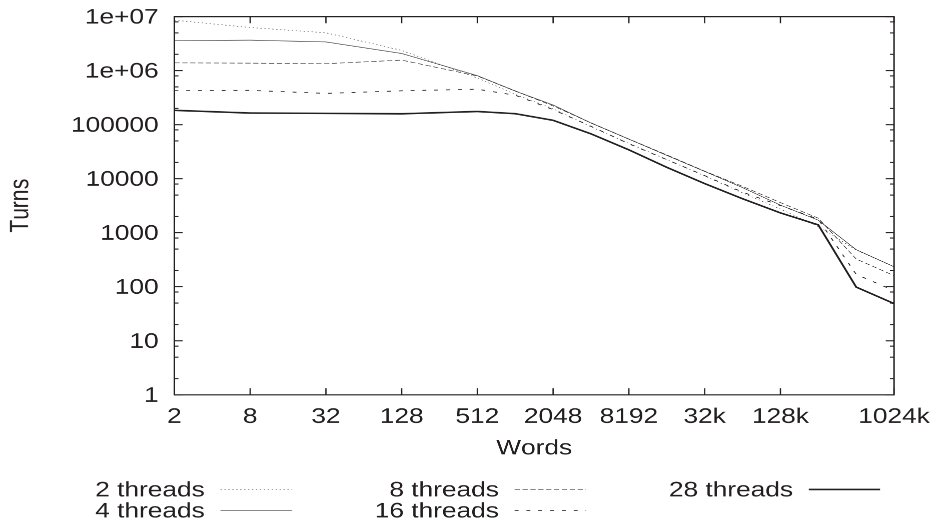


(a) 16 threads, low contention.

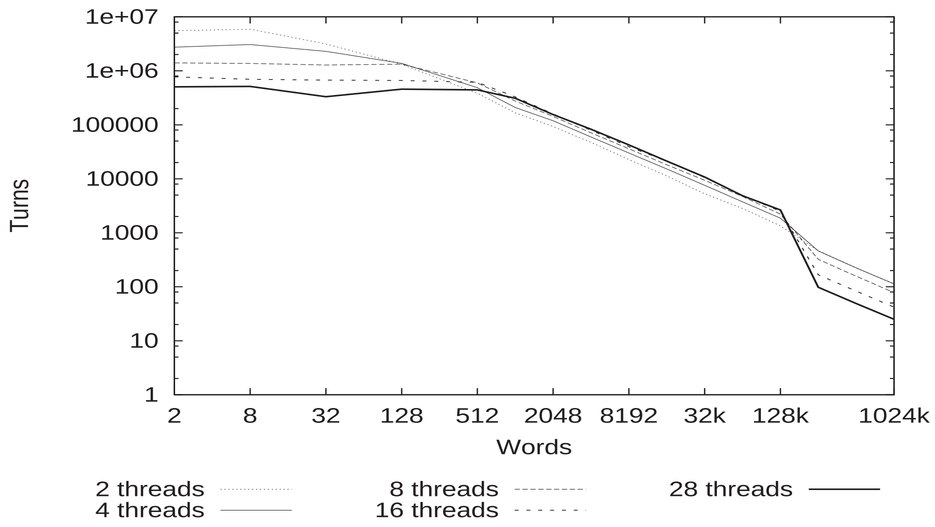


(b) 8192 word register, low contention.

Figure 8.10: Average number of reads or writes per thread on NUMA Origin 3800 at low contention.

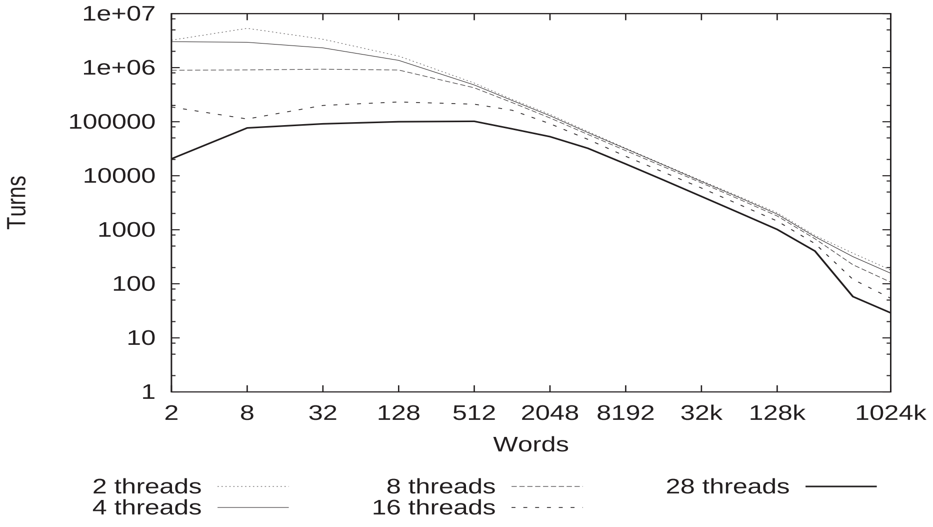


(a) New algorithm, all operations

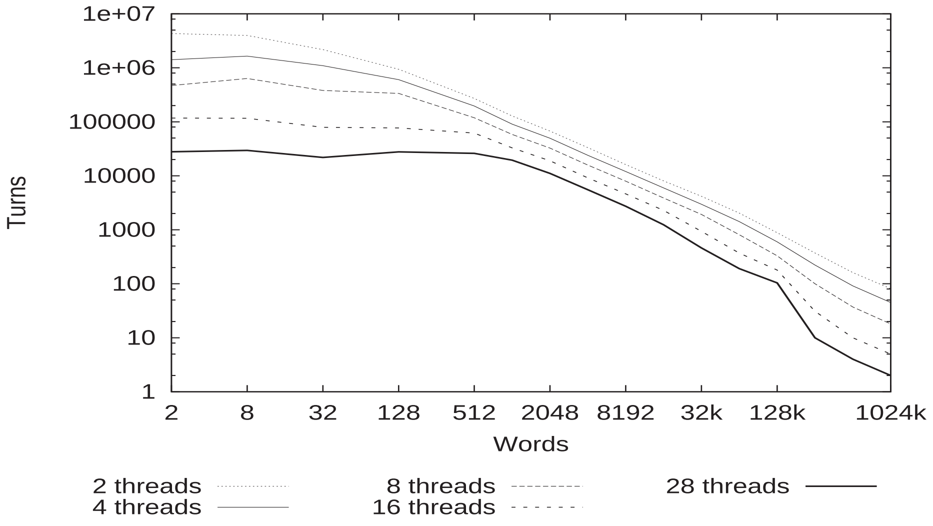


(b) Peterson's algorithm, all operations

Figure 8.11: The average number of reads or writes per thread for the new algorithm and Peterson's algorithm compared with themselves for different number of threads under high contention. Run on NUMA Origin 2000.

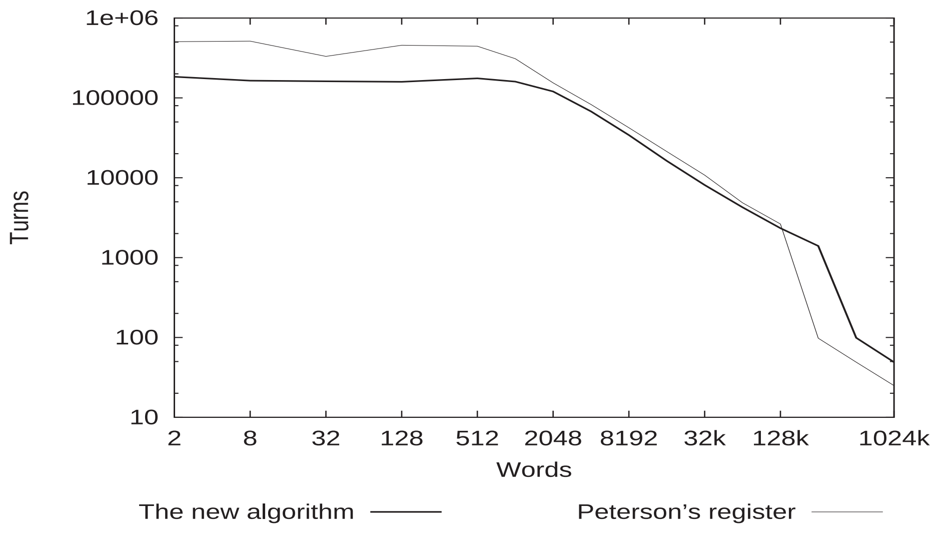


(a) New algorithm, number of writes

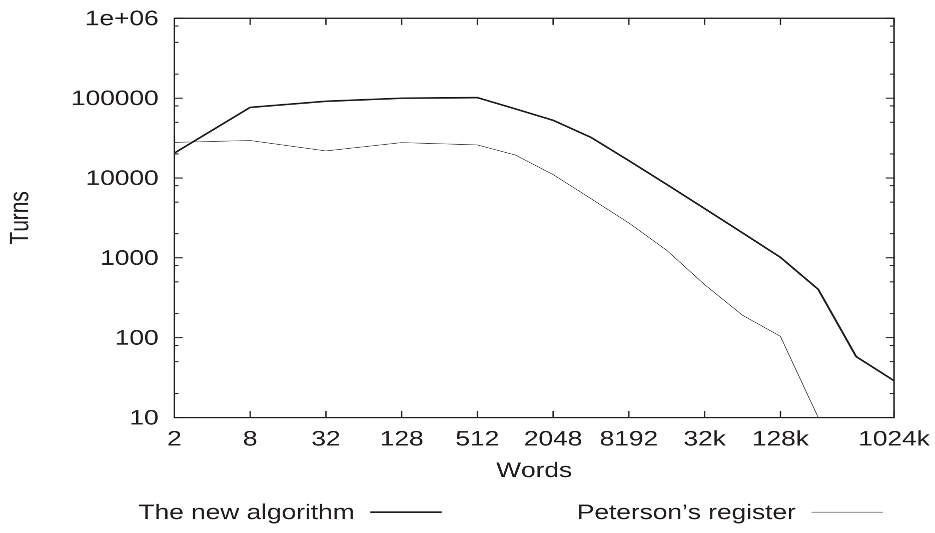


(b) Peterson's algorithm, number of writes

Figure 8.12: The number of writes per thread for the new algorithm and Peterson's algorithm compared with themselves for different number of threads under high contention. Run on NUMA Origin 2000.



(a) All operations.



(b) Write operations.

Figure 8.13: Average number of operations per thread at high contention and 28 threads on NUMA Origin 2000.

multiprocessor systems [Tan01]. The difference between UMA and NUMA is how the memory is organized. In a UMA system all processors have the same latency and bandwidth to the memory. In a NUMA system, processors are placed in nodes and each node has some of the memory directly attached to it. The processors of one node have fast access to the memory attached to that node, but accesses to memory on another node have to be made over the interconnect network and are therefore significantly slower.

The three different systems we used are:

- An UMA Sun SunFire 880 with 6 900MHz UltraSPARC III+ (8MB L2 cache) processors running Solaris 9.
- A ccNUMA SGI Origin 2000 with 29 250MHz MIPS R10000 (4MB L2 cache) processors running IRIX 6.5.
- A ccNUMA SGI Origin 3800 with 128 500MHz MIPS R14000 (8MB L2 cache) processors running IRIX 6.5.

The systems were used non-exclusively, but for the SGI systems the batch-system guarantees that the required number of CPUs was available. The *swap* and *fetch_and_or* suboperations were implemented by the `swap` hardware instruction [WG00] and a lock-free subroutine using the `compare_and_swap` hardware instruction on the SunFire machine. On the SGI Origin machines *swap* and *fetch_and_or* were implemented by the system provided [Cor04] synchronization primitives `__lock_test_and_set` and `__fetch_and_or`, respectively.

Results

Following the analysis and the diagrams presenting the experiments' outcome, it is clear that the performance of the lock-based solutions is not even near the figures of the wait-free algorithms unless the number of threads is minimal (2) *and* the size of the register is small. Moreover, as expected following the analysis, the algorithm proposed in this paper performs at least as well and in the large-size register cases better than Peterson's wait-free solution.

More specifically, on the UMA SunFire the new algorithm outperforms the others for large registers under both low and high contention (cf. Figure 8.4-8.5). The worst performer under high contention is the spinlock, which is particularly vulnerable to threads being preempted inside the critical section, something that is increasingly likely as the number of threads exceed the number of processors. Under low contention the differences are, as expected, much less pronounced. Note that the high contention results are presented on an exponential scale while the low contention results use a linear scale.

On the NUMA Origin 2000 platform (cf. Figures 8.6-8.8) and on the NUMA Origin 3800 platform (cf. Figures 8.9-8.10), we observe the effect of the particular architecture, namely that the possibility to cause high contention on a synchronization variable significantly affects the performance of the solutions. By observing the performance diagrams for this case, we still see the writer in the new algorithm performs significantly better than the writer in Peterson's algorithm in both the low and high-contention scenarios. Recall that the writer following Peterson's algorithm may have to write to more buffers as the number of readers grow. The writer of the algorithm proposed here has no such problems. The latter phenomenon, though, has a seemingly positive side-effect in Peterson's algorithm: namely, as the writer becomes slower, the chances that the readers have to read their individual buffers apart from the regular two buffers, become smaller. Hence, the difference in the readers' performance for the two wait-free algorithms under high contention becomes smaller. (cf. Figure 8.6(a) and 8.7(a). The difference in behaviour between readers and writers under high contention and a varying number of threads can be studied in detail in Figure 8.7(a) and Figure 8.7(b). Some observations based on the results are: (i) the readers/writers lock does well in terms of read operations while the writer suffers significantly since a write can be locked out by preceding and concurrent overlapping reads; (ii) the new algorithm's advantage over Peterson's decreases when the number of threads increase for the readers, while it increases for the writer.

Under low contention the two NUMA Origin platforms (cf. Figure 8.8 and Figure 8.10) show similar behaviour. In the experiments with varying register size (cf. Figure 8.8(a) and Figure 8.10(a)) there is a break in the number of operations per thread at a certain register size which is probably caused by the increased amount of interconnect traffic needed at the larger sizes. The two wait-free algorithms show good scalability behaviour across different number of threads (cf. Figure 8.8(b) and Figure 8.10(b)) with the new algorithm having an advantage over Peterson's. The average number of operations per thread for the readers/writers lock is also good, although as discussed above the writer is likely to be locked out most of the time. The large drop at 10 threads for the readers/writers lock in Figure 8.10(b) is probably caused by the NUMA architecture in conjunction with competing workloads (the NUMA Origin 3800platform is shared among many concurrent users) causing a high load on the interconnect and possibly also forcing our benchmark to use widely distributed processors and memory in the machine.

In Figure 8.11 and Figure 8.12 we can see how the performance of the new algorithm and Peterson's algorithm changes both with the number of words in the register and the number of threads. The new algorithm is more sensitive to the addition of threads than Peterson's algorithm on this platform, most likely due to increased contention. For larger words the difference becomes smaller and smaller and eventually the new algorithm outperforms Peterson's. On the writer's side we see that Peterson's algorithm loses performance to a higher degree with increasing number of threads than the new algorithm does. The similar and characteristic shape of the performance curves for both algorithms are due to the properties of the particular multiprocessor hardware, e.g. cache-line and page size and the interconnect bandwidth. Figure 8.13 directly compares the performance of the new algorithm and Peterson's algorithm for 28 threads from the experiments on the NUMA Origin 2000 system above. Here we can see that while Peterson's algorithm maintains an advantage in the number of read operations until a large register size, the new algorithm has an increasing advantage in the number of write operations.

8.6 Conclusions

This paper presents a simple and efficient algorithm for atomic registers (memory words) of arbitrary size. The simplicity and the good time complexity of the algorithm are achieved via the use of two common synchronization primitives. The paper also presents a performance evaluation of (i) the new algorithm; (ii) a previously known practical algorithm that is based only on read and write operations; and (iii) two mutual-exclusion-based registers. The evaluation is performed on three different well-known multiprocessor systems.

Since shared objects are commonly used in parallel/multithreaded applications, such results and further research along this line, on shared objects implementations, is significant towards providing better support for efficient synchronization and communication for these applications.

Chapter 9

Conclusions and future work

In this thesis we have studied optimistic methods for synchronization in key system services. We have proposed new algorithms, and by analyzing their properties, implementing them and evaluating them in practical experiments we have shown that these optimistic methods offer benefits for concurrent system services in terms of overhead, throughput and scalability.

We proposed lightweight causal cluster consistency, an information dissemination service providing optimistic causal order, e.g. for multi-peer collaborative applications. Our algorithm runs on top of decentralized probabilistic protocols for group communication and its design aims to scale well, impose an even load on the system and provide high-probability reliability guarantees. Our analysis and experimental study indicate that our algorithm meets its goals.

Together with the lightweight causal cluster consistency algorithm we also developed a dynamic and fault-tolerant cluster management algorithm for managing the amount of concurrency in event-based peer-to-peer dissemination systems. The algorithm manages a set of tickets/shared resources such that each ticket has only one owner and can also recover tickets from crashed processes.

We also studied the accuracy of plausible timestamps with fixed and small number of entries, aiming at scalable solutions for large systems. Within this effort we analyzed how these clocks may relate causally independent event pairs. Based on the criteria derived from the analysis we designed two logical clock algorithms, MinDiff and ROV-MRS that both show very competitive performance for small clock/timestamp sizes.

Future work in this area is, among other things, to examine applications where plausible clocks with good ordering accuracy may give significant benefits, such as maintenance of causally consistent replicas of objects and causally consistent information dissemination in large peer-to-peer systems. Further, the cluster consistency model could be extended to other types of consistencies besides causal; e.g. when the interest is on the ordering of overlapping or commuting operations on shared data.

For shared memory systems we proposed two lock-free implementations of system services together with new algorithms and data structures needed in the designs (flat-sets and hand-off) and the analysis of these. In particular, we presented NBmalloc, a lock-free concurrent memory allocator designed to enhance performance and scalability on multiprocessors and LFthreads, a lock-free user-level thread library. We evaluated our implementations experimentally using benchmark applications and compared them to the "traditional" service implementations. Our evaluation shows promising results with respect to scalability and throughput in addition to the qualitative properties gained from a lock-free design, such as tolerance against delayed or stopped processes. Future work along this track include generalizing the method for lock-free "inter-object" operations from the flat-set data-structure in NBmalloc. A general methodology in this direction would enable combinations of known lock-free data structures (e.g. list-structures) into larger, interconnected ones, for use in lock-free applications and system services.

We presented a lock-free memory reclamation algorithm intended for use in lock-free data structures needing dynamic memory. Our algorithm is based on reference counting, and is to our knowledge the first lock-free solution that has all the following properties: (i) guarantees the safety of local as well as global references, (ii) provides an upper bound of deleted but not yet reclaimed nodes, (iii) is compatible with arbitrary memory allocation schemes, and (iv) uses only atomic primitives that are available in modern architectures. It also showed improved performance compared to one of the most practical previous solution.

We also presented a simple and efficient algorithm for atomic registers of arbitrary size. The algorithm has improved time complexity and also improved performance compared to the most practical previously known algorithm.

Projects of interest to further continuing and building on this work include the assembling of lock-free data structures as well as support components for implementing lock-free data structures, such as memory reclamation algorithms, into a library. The availability of such libraries is crucial for the adoption of lock-free methods in real applications, since to successfully implement lock-free algorithms from scratch often requires considerable effort and expertise. Another research issue related to libraries of lock-free components is to investigate how to design clean, easy to use and to understand interfaces to these components.

Bibliography

[ABLL92] Thomas Anderson, Brian Bershad, Edward Lazowska, and Henry Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *ACM Transactions on Computer Systems*, pages 53–79, February 1992.

[ADF$^+$02] Ole Agesen, David L. Detlefs, Christine H. Flood, Alex Garthwaite, Paul A. Martin, Mark Moir, Nir Shavit, and Guy L. Steele. DCAS-based concurrent deques. *MST: Mathematical Systems Theory*, 35, 2002.

[ADG$^+$94] Yehuda Afek, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. A bounded first-in, first-enabled solution to the ℓ-exclusion problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):939–953, May 1994.

[ADHK97] Uri Abraham, Shlomi Dolev, Ted Herman, and Irit Koll. Self-stabilizing l-exclusion. In *Proceedings of the 3rd Workshop on Self-Stabilizing Systems*, pages 48–63. Carleton University Press, 1997.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[AG06] Anurag Agarwal and Vijay K. Garg. Efficient dependency tracking for relevant events in concurrent systems. *Distributed Computing*, pages 1 – 21, 2006.

[AGBH03] Luc Onana Alima, Ali Ghodsi, Per Brand, and Seif Haridi. Multicast in DKS(N; k; f) overlay networks. In *Proceedings of the 7th International Conference on Principles of Distributed Systems*, volume 3144 of *Lecture Notes in Computer Science*, pages 83–95. Springer Verlag, 2003.

[AHJ91] Mustaque Ahamad, Phillip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 274–281. IEEE Computer Society, May 1991.

[AJO98] James H. Anderson, Rohit Jain, and David Ott. Wait-free synchronization in quantum-based multiprogrammed systems. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC '98)*, volume 1499 of *Lecture Notes in Computer Science*, pages 34–48. Springer Verlag, September 1998.

[AM99] James H. Anderson and Mark Moir. Wait-free synchronization in multiprogrammed systems: Integrating priority-based and quantum-based scheduling (extended abstract). In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC '99)*, pages 123–132, May 1999.

[ANB$^+$95] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[ARJ97] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems*, 15(1):134–165, February 1997.

[Bar93] Greg Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, June 1993.

[BEG04] Sbastien Baehni, Patrick Th. Eugster, and Rachid Guerraoui. Data-aware multicast. In *Proceedings of the 5th IEEE International Conference on Dependable Systems and Networks*, pages 233–242, 2004.

[Ber93] Brian N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 264–274. IEEE Computer Society Press, May 1993.

[Ber02] Emery D. Berger. *Memory Management for High-Performance Applications*. PhD thesis, The University of Texas at Austin, August 2002.

[BHO$^+$99] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failure. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BL94] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing,. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 356–368, November 1994.

[Blo88] Bard Bloom. Constructing two-writer atomic registers. *IEEE Transactions on Computers*, 37(12):1506–1514, December 1988.

[BM93] Özalp Babaoğlu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In Sape Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.

[BM03] Roberto Baldoni and Giovanna Melideo. k-dependency vectors: A scalable causality-tracking protocol. In *Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 219–226, 2003.

[BMBW00] Emery Berger, Kathryn McKinley, Robert Blumofe, and Paul Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, November 2000.

[BP87] James E. Burns and Gary L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 222–231. ACM Press, August 1987.

[BPRS98] Roberto Baldoni, Ravi Prakash, Michel Raynal, and Mukesh Singhal. Efficient Δ-causal broadcasting. *International Journal of Computer Systems Science and Engineering*, 13(5):263–269, 1998.

[BSS91] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[CB91] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, July 1991.

[CB97] J. Chen and A. Burns. A fully asynchronous reader/writer mechanism for multiprocessor real-time systems. Technical Report YCS-288, Department of Computer Science, University of York, May 1997.

[CH93] Christer Carlsson and Olof Hagsand. DIVE - a multi-user virtual reality system. In *Proceedings of the IEEE Annual International Symposium*, pages 394–400. IEEE, September 1993.

[Cor04] David Cortesi. *Topics in IRIX Programming.* Silicon Graphics, Inc., 2004. (doc #:007-2478-009).

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronization needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[DG02] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In *ISMM '02 Proceedings of the 3rd International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 163–174. ACM Press, June 2002.

[DGH$^+$87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12. ACM Press, 1987.

[DMMS01] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele. Lock-free reference counting. In *Proceedings of the 20th annual ACM symposium on Principles of distributed computing*, pages 190–199. ACM Press, August 2001.

[EGH$^+$01] Patrick Th. Eugster, Rachid Guerraoui, Sidath B. Handurukande, Anne-Marie Kermarrec, and Petr Kouznetsov. Lightweight probabilistic broadcast. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '01)*, pages 443–452. IEEE Computer Society, July 2001.

[FCL93] Michael J. Feeley, Jeffrey S. Chase, and Edward D. Lazowska. User-level threads and interprocess communication. Technical Report TR-93-02-03, University of Washington, Department of Computer Science and Engineering, February 1993.

[Fid88] Colin Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1), February 1988.

[Fid91] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, August 1991.

[FJC00] Antonio Fernández, Ernesto Jiménez, and Vicente Cholvi. On the interconnection of causal memory systems. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC-00)*, pages 163–170. ACM Press, July 2000.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibilty of distributed consensus with one faulty. *Journal of the ACM*, 32(2):374–382, April 1985.

[Fra04] Keir A. Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, February 2004.

[GB97] Chris Greenhalgh and Steven Benford. A multicast network architecture for large scale collaborative virtual environments. In *Proceedings of the 2nd European Conference on Multimedia Applications, Services and Techniques*, volume 1242 of *Lecture Notes in Computer Science*, pages 113–128. Springer Verlag, 1997.

[GC96] Michael Greenwald and David R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Operating Systems Design and Implementation*, pages 123–136, 1996.

[GKM01] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. SCAMP: Peer-to-peer lightweight membership service for large-scale group communication. In *Proceedings of the 3rd International COST264 Workshop*, volume 2233 of *Lecture Notes in Computer Science*, pages 44–55. Springer Verlag, 2001.

[GKPT05a] Anders Gidenstam, Boris Koldehofe, Marina Papatriantafilou, and Philippas Tsigas. Dynamic and fault-tolerant cluster management. In *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing*, pages 237–244. IEEE, August 2005.

[GKPT05b] Anders Gidenstam, Boris Koldehofe, Marina Papatriantafilou, and Philippas Tsigas. Lightweight causal cluster consistency. In *Proceedings of the Conference on Innovative Internet Community Systems (I^2CS '05)*, volume 3908 of *Lecture Notes in Computer Science*, pages 17–28. Springer Verlag, June 2005.

[Glo03] Wolfram Gloger. Wolfram Gloger's malloc homepage, 2003. http://www.malloc.de/en/.

[GPST05] Anders Gidenstam, Marina Papatriantafilou, Hkan Sundell, and Philippas Tsigas. Practical and efficient lock-free garbage collection based on reference counting. In *Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN)*, pages 202–207. IEEE Computer Society, December 2005.

[GPT04] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. Allocating memory in a lock-free manner. Technical Report 2004-04, Computing Science, Chalmers University of technology, May 2004.

[GPT05] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. Allocating memory in a lock-free manner. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 329–242. Springer Verlag, October 2005.

[Gre99] Michael B. Greenwald. *Non-blocking synchronization and system design.* PhD thesis, Stanford University, 1999.

[Har01] Timothy L. Harris. A pragmatic implementation of non-blocking linked lists. In *Distributed Computing, 15th International Conference*, volume 2180 of *Lecture Notes in Computer Science*, pages 300–314. Springer Verlag, October 2001.

[Har05] Thomas E. Hart. Comparative performance of memory reclamation strategies for lock-free and concurrently-readable data structures. Master's thesis, University of Toronto, 2005.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transaction on Programming and Systems*, 11(1):124–149, January 1991.

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[Her06] Maurice Herlihy. Software transactional memory package for C#, 2006. http://www.cs.brown.edu/people/mph/home.html.

[HFP02] Timothy L. Harris, Keir Fraser, and Ian A Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC '02)*, volume 2508 of *Lecture Notes in Computer Science.* Springer Verlag, October 2002.

[HG98] Wim H. Hesselink and Jan Friso Groote. Waitfree distributed memory management by create and read until deletion (CRUD). Technical Report SEN-R9811, CWI, Amsterdam, 1998.

[HG01] Wim H. Hesselink and Jan Friso Groote. Wait-free concurrent memory management by create and read until deletion (CaRuD). *Distributed Computing*, 14(1):31–39, January 2001.

[HKMP95] J. Hromkovic, R. Klasing, B. Monien, and R. Peine. Information in interconnection networks (broadcasting and gossiping). In F. Hsu and D.-A. Du, editors, *Combinatorial Network Theory*, pages 125–212. Kluwer, 1995.

[HLM02] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structure. In *Proceedings of 16th International Symposium on Distributed Computing (DISC 2002)*, volume 2508 of *Lecture Notes in Computer Science*, pages 339–353. Springer Verlag, October 2002.

[HLMM02] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Dynamic-sized lock-free data structures. In *Proceedings of the 21st annual symposium on Principles of distributed computing*, pages 131–131. ACM Press, 2002.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture (ISCA 1993)*, pages 289–300. ACM Press, 1993.

[HPT02] Jaap-Henk Hoepman, Marina Papatriantafilou, and Philippas Tsigas. Self-stabilization of wait-free shared memory objects. *Journal of Parallel and Distributed Computing*, 62(5):766–791, 2002.

[HRMB03] Jean-Michel Helary, Michel Raynal, Giovanna Melideo, and Roberto Baldoni. Efficient causality-tracking timestamping. *IEEE Transactions on Knowledge and Data Engineering*, 15, 2003.

[HT93] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, pages 353–383. Addison-Wesley, 1993.

[HT03] Phuong Hoai Ha and Philippas Tsigas. Reactive multi-word synchronization for multiprocessors. In *Proceedings of the 12th IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*, pages 184–193. IEEE Computer Society Press, September 2003.

[HV95] S. Haldar and K. Vidyasankar. Constructing 1-writer multireader multivalued atomic variable from regular variables. *Journal of the ACM*, 42(1):186–203, January 1995.

[HV96] S. Haldar and K. Vidyasankar. Simple extensions of 1-writer atomic variable constructions to multiwriter ones. *Acta Informatica*, 33(2):177–202, 1996.

[HV02] Sibsankar Haldar and Paul Vitányi. Bounded concurrent timestamp systems using vector clocks. *Journal of the ACM*, 49(1):101–126, January 2002.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[IBM83] IBM. *IBM System/370 Extended Architecture, Principles of Operation*, 1983. Publication No. SA22-7085.

[IL93] Amos Israeli and Ming Li. Bounded time-stamps. *Distributed Computing*, 6(4):205–209, 1993.

[IR93] Amos Israeli and Lihu Rappoport. Efficient wait-free implementation of a concurrent priority queue. In *Proceedings of the 7th International Workshop on Distributed Algorithms (WDAG '93)*, volume 725 of *Lecture Notes in Computer Science*, pages 1–17. Springer Verlag, September 1993.

[IS92] Amos Israeli and Amnon Shaham. Optimal multi-writer multi-reader atomic register. In *Proceedings of the 11th Annual Symposium on Principles of Distributed Computing*, pages 71–82. ACM Press, August 1992.

[Jay98] Prasad Jayanti. A complete and constant time wait-free implementation of CAS from LL/SC and vice versa. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC '98)*, volume 1499 of *Lecture Notes in Computer Science*, pages 216–230. Springer Verlag, September 1998.

[KKV87] Lefteris M. Kirousis, Evangelos Kranakis, and Paul M. B. Vitányi. Atomic multireader register. In *Distributed Algorithms, 2nd International Workshop*, volume 312 of *Lecture Notes in Computer Science*, pages 278–296. Springer Verlag, July 1987.

[Kol03] Boris Koldehofe. Buffer management in probabilistic peer-to-peer communication protocols. In *Proceedings of the 22nd Symposium on Reliable Distributed Systems*, pages 76–85. IEEE, October 2003.

[KR93] Hermann Kopetz and Johannes Reisinge. The non-blocking write protocol NBW: A solution to a real-time synchronisation problem. In *Proceedings of the Real-Time Systems Symposium*, pages 131–137. IEEE Computer Society Press, December 1993.

[KS98] Ajay D. Kshemkalyani and Mukesh Singhal. Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. *Distributed Computing*, 11:91–111, 1998.

[KZ99] Denis A. Khotimsky and Igor A. Zhuklinets. Hierarchical Vector Clock: Scalable plausible clock for detecting causality in large distributed systems. In *Proceedings of 1999 Second International Conference on ATM (ICATM'99)*, pages 156–163, June 1999.

[LAA87] Loui and Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4, 1987.

[Lam77] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*, volume 7 of *21*, pages 558–565, July 1978.

[Lam79] Leslie Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[Lam86] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–101, 1986.

[LGH$^+$04] Andreas Larsson, Anders Gidenstam, Phuong H Ha, Marina Papatriantafilou, and Philippas Tsigas. Multi-word atomic read/write registers on multiprocessor systems. In *Proceedings of the 12th Annual European Symposium on Algorithms (ESA '04)*, volume 3221 of *Lecture Notes in Computer Science*, pages 736–748. Springer Verlag, September 2004.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LK98] Paul Per-Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. In *ISMM'98 Proceedings of the 1st International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 176–185. ACM Press, October 1998.

[LMR02] Nancy Lynch, Dahlia Malkhi, and David Ratajczak. Atomic data access in distributed hash tables. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 295–305. Springer Verlag, 2002.

[LTV96] Ming Li, John Tromp, and Paul M. B. Vitányi. How to share concurrent wait-free variables. *Journal of the ACM*, 43(4):723–746, July 1996.

[LV92] Ming Li and Paul M. B. Vitányi. Optimality of wait-free atomic multiwriter variables. *Information Processing Letters*, 43(2):107–112, August 1992.

[Mas92] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* PhD thesis, Columbia University, 1992.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier, 1989.

[Mau00] Martin Mauve. Consistency in replicated continuous interactive media. In *Proceedings of ACM CSCW'00 Conference on Computer-Supported Cooperative Work*, pages 181–190, 2000.

[Mic02a] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA-02)*, pages 73–82. ACM Press, August 2002.

[Mic02b] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, pages 21–30. ACM Press, 2002.

[Mic03] Maged Michael. CAS-based lock-free algorithm for shared deques. In *Proceedings of the 9th International Euro-Par Conference*, Lecture Notes in Computer Science. Springer Verlag, August 2003.

[Mic04a] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(8), August 2004.

[Mic04b] Maged M. Michael. Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In *Proceedings of the 18th International Conference on Distributed Computing (DISC '04)*, October 2004.

[Mic04c] Maged M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of SIGPLAN 2004 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices. ACM Press, June 2004.

[MLH03] Mark Moir, Victor Luchangco, and Maurice Herlihy. Lock-free implementation of dynamic-sized shared data structure. US Patent WO 2003/060705 A3, January 2003.

[MMS02] Paul A. Martin, Mark Moir, and Guy L. Steele. DCAS-based concurrent deques supporting bulk allocation. Technical Report TR-2002-111, Sun Microsystems, 2002.

[Moi97] Mark Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, 1997.

[Mos93] David Mosberger. Memory consistency models. Technical Report TR 93/11, Department of Computer Science, The University of Arizona, Tucson, 1993.

[MP91] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, 1991.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, June 1995.

[MS95] Maged M. Michael and Michael L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, Computer Science Department, December 1995.

[MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM Press, 1996.

[MS05] Brad T. Moore and Paolo A. G. Sivilotti. Plausible clocks with bounded inaccuracy. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)*, volume 3724 of *Lecture Notes in Computer Science*, pages 214–228. Springer Verlag, 2005.

[MT95] Duncan C. Miller and Jack A. Thorpe. SIMNET:the advent of simulator networking. In *Proceedings of the IEEE*, volume 8 of *83*, pages 1114–1123, August 1995.

[NW87] Richard Newman-Wolfe. A protocol for wait-free, atomic, multireader shared variables. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 232–248. ACM Press, August 1987.

[ON01] Hisashi Oguma and Yasuichi Nakayama. A scheduling mechanism for lock-free operation of a lightweight process library for SMP computers. In *Proceedings of the 8th International Conference on Parallel and Distributed Systems (ICPADS 2001)*, pages 235–242, 2001.

[PB87] Gary L. Peterson and James E. Burns. Concurrent reading while writing II: The multi-writer case. In *28th Annual Symposium on Foundations of Computer Science*, pages 383–392. IEEE, October 1987.

[Pet83] Gary L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[PH98] David A. Patterson and John L. Hennessy. *Computer Organization: The Hardware/Software Interface.* Morgan Kaufmann Publishers, second edition, 1998.

[PRMK03] J. Pereira, L. Rodrigues, M.J. Monteiro, and A.-M. Kermarrec. NEEM: Network-friendly epidemic multicast. In *Procedings of the*

22nd Symposium on Reliable Distributed Systems, pages 15–24. IEEE, October 2003.

[PS97] Ravi Prakash and Mukesh Singhal. Dependency sequences and hierarchical clocks: Efficient alternatives to vector clocks for mobile computing systems. *Wireless Networks*, 3:349–360, 1997.

[PT95] Marina Papatriantafilou and Philippas Tsigas. Wait-free consensus in “in-phase” multiprocessor systems. In *Symposium on Parallel and Distributed Processing (SPDP ’95)*, pages 312–319. IEEE Computer Society Press, October 1995.

[PT97] Marina Papatriantafilou and Philippas Tsigas. On self-stabilizing wait-free clock synchronization. *Parallel Processing Letters*, 7(3):321–328, September 1997.

[Raj90] Ragunathan Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *10th International Conference on Distributed Computing Systems (ICDCS ’90)*, pages 116–123. IEEE Computer Society Press, May 1990.

[RBAR00] Luis Rodrigues, Roberto Baldoni, Emmanuelle Anceaume, and Michel Raynal. Deadline-constrained causal order. In *Proceedings of the 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing*, March 2000.

[RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, volume 2218 of *Lecture Notes in Computer Science*. Springer Verlag, November 2001.

[RFH$^+$01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *ACM SIGCOMM Computer Communication Review*, volume 31, pages 161–172, 2001.

[Rin99] Martin C. Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Transactions on Computer Systems*, 17(4):337–371, 1999.

[RKCD01] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proceedings of the 3rd International COST264 Workshop*, volume 2233 of *Lecture Notes in Computer Science*, pages 30–43. Springer Verlag, 2001.

[Rob71] J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *Journal of the ACM*, 18(3):416–423, July 1971.

[RST91] Michel Raynal, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, September 1991.

[SAG94] Ambuj K. Singh, James H. Anderson, and Mohamed G. Gouda. The elusive atomic register. *Journal of the ACM*, 41(2):311–339, March 1994.

[Sch88] Frank B. Schmuck. The use of efficient broadcast protocols in asynchronous distributed systems. Technical Report TR88-928, Cornell University, Computer Science Department, August 1988.

[SGG05] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating Systems Concepts*. Wiley, 2005.

[SGI03] SGI. The standard template library for C++, 2003. http://www.sgi.com/tech/stl/Allocators.html.

[Sim90] H. R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings, Computers and Digital Techniques*, 137(1):17–30, January 1990.

[SJZ+98] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, 1998.

[SK92] Mukesh Singhal and Ajay Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43(1):47–52, August 1992.

[SMK+01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM 2001 Conference*, pages 149–160. ACM Press, August 2001.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing (PODC 1995)*, pages 204–213. ACM Press, August 1995.

[ST02] Håkan Sundell and Philippas Tsigas. NOBLE: A non-blocking inter-process communication library. In *Proceedings of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*, Lecture Notes in Computer Science. Springer Verlag, 2002.

[ST03] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proceedings of the 17th IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS 03)*. IEEE Press, 2003.

[ST04] Håkan Sundell and Philippas Tsigas. Lock-free and practical deques using single-word compare-and-swap. In *Proceedings of the 8th International Conference on Principles of Distributed Systems (OPODIS '04)*, volume 3544 of *Lecture Notes in Computer Science*. Springer Verlag, December 2004.

[Ste00] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*. ACM Press, October 2000.

[Sun04a] Håkan Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Chalmers University of Technology, November 2004.

[Sun04b] Håkan Sundell. Wait-free reference counting and memory management. Technical Report 2004-10, Computing Science, Chalmers University of Technology, November 2004.

[Sun05] Håkan Sundell. Wait-free reference counting and memory management. In *Proceedings of the 19th International Parallel & Distributed Processing Symposium*. IEEE, April 2005.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, second edition, 2001.

[TR01] Francisco J. Torres-Rojas. Performance evaluation of plausible clocks. In *Proceedings of the 7th International Euro-Par Conference (Euro-Par '01)*, volume 2150 of *Lecture Notes in Computer Science*, pages 476–481. Springer Verlag, August 2001.

[TRA99] Francisco J. Torres-Rojas and Mustaque Ahamad. Plausible clocks: Constant size logical clocks for distributed systems. *Distributed Computing*, 12:179–195, 1999.

[TRAR98] Francisco J. Torres-Rojas, Mustaque Ahamad, and Michel Raynal. Lifetime based consistency protocols for distributed objects. In *Proceedings of the 12th International Symposium on Distributed*

Computing (DISC '98), volume 1499 of *Lecture Notes in Computer Science*, pages 378–392. Springer Verlag, September 1998.

[TS92] John Turek and Dennis Shasha. The many faces of consensus in distributed systems. *IEEE Computer*, 25(6):8–17, June 1992.

[TSP92] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 212–222. ACM Press, 1992.

[TZ01a] Philippas Tsigas and Yi Zhang. Evaluating the performance of non-blocking synchronisation on shared-memory multiprocessors. In *Proceedings of the ACM SIGMETRICS 2001/Performance 2001*, pages 320–321. ACM Press, June 2001.

[TZ01b] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the 13th annual ACM Symposium on Parallel Algorithms and Architectures*, pages 134–143. ACM Press, 2001.

[TZ02] Philippas Tsigas and Yi Zhang. Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies. In *Proceedings of the 3rd ACM Workshop on Software and Performance (WOSP '02)*, pages 55–67. ACM Press, July 2002.

[VA86] Paul M. B. Vitányi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware. In *27th Annual Symposium on Foundations of Computer Science*, pages 233–243. IEEE, October 1986.

[Val94] John D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, October 1994.

[Val95a] John D. Valois. *Lock-Free Data Structures.* PhD thesis, Rensselaer Polytechnic Institute, 1995.

[Val95b] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*, pages 214–222. ACM Press, August 1995.

[WG00] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual.* Pretice Hall, 2000. Version 9.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management (IWMM '95)*, volume 986 of *Lecture Notes in Computer Science*, pages 1–78, September 1995.

[ZHS$^+$04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, and Anthony D. Joseph. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53, 2004.

[ZLE91] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel processors. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, April 1991.

[ZS97] Khawar M. Zuberi and Kang G. Shin. An efficient semaphore implementation scheme for small-memory embedded systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, pages 25–37. IEEE, June 1997.

[ZZJ$^+$01] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 11–20. ACM Press, 2001.