Thesis for the Degree of Doctor of Philosophy

Non-blocking Synchronization: Algorithms and Performance Evaluation

Yi Zhang

CHALMERS | GÖTEBORG UNIVERSITY

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

Göteborg, 2003

Non-blocking Synchronization: Algorithms and Performance Evaluation
Yi Zhang
ISBN 91-7291-299-5

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 1981
ISSN 0346-718X

Technical report 14D
ISSN 1651-4971
School of Computer Science and Engineering

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg,
Sweden
Telephone + 46 (0)31-772 1000

Chalmers Reproservice
Göteborg, Sweden, 2003

Abstract

The thesis investigates non-blocking synchronization in shared memory systems, in particular in high performance shared memory multiprocessors and real-time shared memory systems. We explore the performance impact of non-blocking synchronization in high performance shared memory multiprocessors and the applicability of non-blocking synchronization in real-time systems.

The performance advantage of non-blocking synchronization over mutual exclusion in shared memory multiprocessors has been advocated by the theory community for a long time. In this work, we try to make non-blocking synchronization appreciated by application designers and programmers through a sequence of results. First, we develop a non-blocking FIFO queue algorithm which is simple and can be used as a building block for applications and libraries. The algorithm is fast and scales very well in both symmetric and non-symmetric shared memory multiprocessors. Second, we implement a fine-grain parallel Quicksort using non-blocking synchronization. Although fine-grain parallelism has been thought to be inefficient for computations like sorting due to synchronization overhead, we show that efficiency can be achieved by incorporating non-blocking techniques for sharing data and computation tasks in the design and implementation of the algorithm. Finally, we investigate how performance and speedup of applications would be affected by using non-blocking rather than blocking synchronization in parallel systems. We show that for many applications, non-blocking synchronization leads to significant speedup for a fairly large number of processors, while they never slow the applications down.

Predictability is the dominant factor in performance matrices of real-time systems and a necessary requirement for non-blocking synchronization in real-time multiprocessors. In this thesis, we propose two non-blocking data structures with predictable behavior and present an inter-process coordination protocol that bounds the execution time of lock-free shared data object operations in real-time shared memory multiprocessors. The first data structure is a non-blocking buffer for real-time multiprocessors. The buffer gives a way to concurrent real-time tasks to read and write shared data and allows multiple write operations and multiple read operations to be executed concurrently and has a predictable behavior. Another data structure is a special wait-free queue for real-time systems. We present efficient algorithmic implementations for the queue. These queue implementations can be used to enable communication between real-time tasks and non-real-time tasks in systems. The inter-process protocol presented is a general protocol which gives predictable behavior to any lock-free data structure in real-time multiprocessors. The protocol works for the lock-free implementations in real-time multiprocessor systems in the same way as the multiprocessor priority ceiling protocol (MPCP) works for mutual exclusion in real-time multiprocessors. With the new protocol, the worst case execution time of accessing a lock-free shared data object can be bounded.

List of Included Papers and Reports

This thesis is based on the following papers.

1. Philippas Tsigas, Yi Zhang, "Non-blocking Data Sharing in Multiprocessor Real-Time Systems", in the Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA '99), pages 247–254. IEEE Computer Society Press, December 1999.

2. Philippas Tsigas, Yi Zhang, "Evaluating The Performance of Non-Blocking Synchronisation on Shared-Memory Multiprocessors", in the Proceedings of the ACM SIGMETRICS 2001/Performance 2001, pages 320–321. ACM press, June 2001.

3. Philippas Tsigas, Yi Zhang, "A Simple, Fast and Scalable Non-Blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems", in the Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '01), pages 134–143. ACM press, July 2001.

4. Philippas Tsigas, Yi Zhang, "Integrating Non-blocking Synchronisation in Parallel Applications: Performance Advantages and Methothologies", in the Proceedings of the 3rd ACM Workshop on Software and Performance (WOSP'02), pages 55–67. ACM press, July 2002.

5. Philippas Tsigas, Yi Zhang, "A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000", In the Proceedings of the 11th Euromicro Conference on Parallel Distributed and Network based Processing, pages 372–381. IEEE Computer Society Press, February 2003.

6. Philippas Tsigas, Yi Zhang, "Efficient Wait-Free Queue Algorithms for Real-Time Synchronization", Technical Report 2002-05, Department of Computing Science, Chalmers University of Technology, 2002.

7. Yi Zhang, Philippas Tsigas, "Lock-free Object-Sharing for Shared Memory Multiprocessors", Technical Report 2003-03, Department of Computing Science, Chalmers University of Technology, 2003.

ACKNOWLEDGMENTS

I would like to thank my advisor Philippas Tsigas who always lets me pursue my ideas and gives me stimulating advice. He has supported, inspired, and encouraged me in many different ways through different stages. I feel so fortunate to have been one of his students.

I also want to thank Marina Papatriantafilou for many insightful comments and suggestions in the past five years and for her helpful comments on preliminary drafts of this thesis. Thanks are also due to Sven-Arne Andrèasson for serving on my committee and for helpful suggestions.

I am grateful to the present and former members of the Distributed Computing and System group. David Rutter, my previous office mate, helped me to correct my English in several papers. Hkan Sundell drove me to several summer schools organized by ARTES and had many helpful discussions together with me. Niklas Elmqvist, Anders Gidenstam, Boris Koldehofe, Ha Hoai Phuong, gave me many comments and feedback on my work.

It is wonderful to work at the Department of Computing Science of Chalmers, where people are always friendly and helpful. I would like to thank the staff of the Department of Computing Science for their administrative and technical help.

I am thankful to the system administrators of UNICC, Unix Numeric Intensive Calculations at Chalmers, for providing me the valuable access to ORIGIN 2000 and SUN Enterprise 10000 for the experiments.

This work is partially funded by ARTES, a national Swedish strategic research initiative in Real-Time Systems. I would like to especially thank Hans Hansson, the program director of ARTES.

Last but not least, my thanks go to my family. I am eternally grateful for the love, support, understanding, and encouragement of my parents and my sister. They have seen me through all my years and have shared in all my hardship and success. I also thank my parents-in-law for their love and support. I am very grateful to Fang, my wife, for her love, supporting, helping and caring. Without them, I would never have gotten this far.

Contents

Chapter 1

Introduction

The focus of the thesis is on studying non-blocking synchronization in shared memory multiprocessors, in particular in high performance shared memory multiprocessors and real-time shared memory multiprocessors.

1.1 High Performance Shared Memory Multiprocessors

A shared memory multiprocessor system consists of multiple processors, provides a single address space for programming, and supports communication between processors through operations on shared memory. Applications running on such systems can use more than one processor at the same time. Programs can increase their execution speed by exploiting the parallelism available on such systems. Single address space shared memory provides an easy programming model to programmers. Shared memory operations can be implemented in hardware or software.

To programmers, programming for shared memory multiprocessors is similar to traditional sequential programming for uniprocessor systems. Communication between processors in shared memory multiprocessors is implicit and transparent via conventional memory access instructions, such as Read/Write, that are also used in sequential programming. Therefore, programmers do not have to consider details of low-level communication between processors and can focus mainly on the applications themselves. When an application is running on shared memory multiprocessors, all processes of the application share the same address space; traditional sequential programming also treats memory as a single address space. Such similarity in programming between shared memory multiprocessors and uniprocessors makes shared memory multiprocessors attractive.

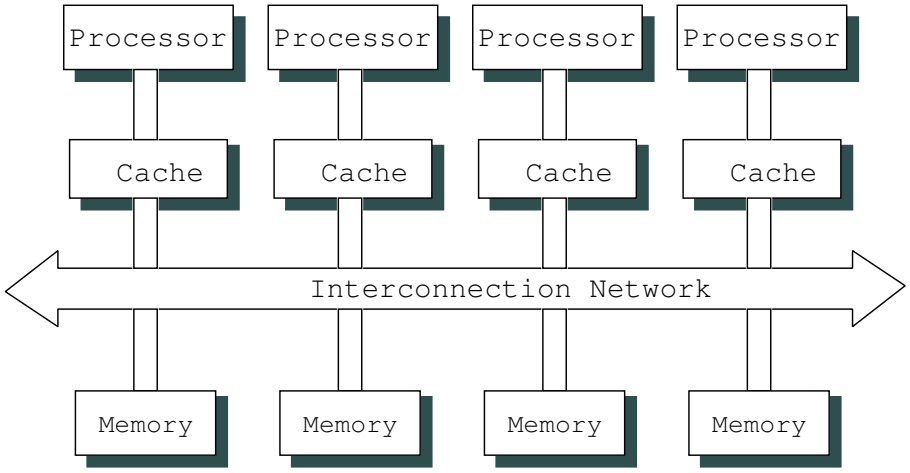


Figure 1.1: A generic architecture for shared memory multiprocessors

Shared memory multiprocessors are ideal for high performance computing. The shared memory communication mechanism supported by hardware provides low latency and high bandwidth for communication between processors. Fast communication enables programmers to explore fine grain parallelism in programs. As processors communicate with each other by conventional memory operations on shared memory, it is easy to transfer sequential programs into parallel ones for shared memory multiprocessors.

A generic architecture for shared memory multiprocessors is illustrated in Figure 1.1. A collection of processors and memory modules are connected through an interconnect communication network. In such systems, all processors run at the same speed and each processor executes its own instructions independently of the others. Processors communicate with each other through shared memory. If the access times are equal between any processor and any memory module, the machine is called UMA, *uniform memory access* shared memory multiprocessor system. An example of such a system is the SUN Enterprise family of multiprocessors. Otherwise it is called NUMA, *non-uniform memory access* shared memory multiprocessor. An example of such a system is the SGI Origin family of multiprocessors. Local cache memories [10, 19] between processors and memory modules are almost always introduced to reduce contention. When cache is introduced, a cache coherency scheme is implemented within the interconnect communication network to ensure that data held in memory is consistent on a system-wide basis.

Synchronization in high performance shared memory multiprocessors is discussed in the next section. Then the structure of real-time shared memory multiprocessors and the comparison between real-time multiprocessors and high performance multiprocessors will be discussed. Finally, we discuss issues related to synchronization

in real-time systems.

1.2 Synchronization

Programming for shared memory multiprocessors introduces synchronization problems that sequential programming does not need to address. Processes in shared memory multiprocessors communicate and coordinate with each other through reading from and writing to shared memory locations. Such Read/Write operations on memory can be executed simultaneously on several processors. The final results of these operations depend on their interleaving. To maintain consistency, synchronization is used to guarantee that only desired interleaving of operations can happen. There are two ways to do synchronization in shared memory: mutual exclusion and non-blocking synchronization.

1.2.1 Mutual Exclusion

Mutual exclusion ensures that certain sections of code will not be executed by more than one process simultaneously. The standard solution to mutual exclusion at kernel level in uniprocessor systems is to momentarily disable interrupts to guarantee that the operation of a shared memory object will not be preempted before it completes. This solution is not feasible for uniprocessor systems at user level, where users do not have the privilege to disable interrupts. In multiprocessor systems, where processes execute on several processors, disabling interrupts at kernel level is too costly. In such cases, locks are used to guarantee that only one process can access a shared memory object: before a process accesses a shared memory object, it must get a lock associate with the object; after accessing the object, it will release the lock. Only one lock protects an object. The part of code that the process executes to access the object is called code in "critical section". If a process cannot get the lock of an object, then another process owns the lock and is working on the object in the critical section.

There are many implementations of locks. The simplest and most popular implementation is the spinlock, which is described with pseudo-code in Figure 1.2. The `lock` function in the spinlock implementation repeatedly checks the content of the shared variable "owned" and changes the content of "owned" to $TRUE$ if it is $FALSE$ in one atomic step with the hardware primitive Test-and-Set. If the old value is $FALSE$, the lock is not owned by any other process. Then, the process puts $TRUE$ in it and owns the lock. If the old value is not $FALSE$, the process just tries again. After a process finishes its critical section, it calls the `unlock` function to release the lock by setting the content of "owned" to $FALSE$.

```
shared boolean owned = FALSE;
void lock()
{
      do
      {
      /* if owned is FALSE, test&set will return
         succeed and change owned to TRUE */
      rtn = Test-and-Set(owned);
      } while(rtn != succeed);
}

void unlock() {
   owned = FALSE;
}
```

Figure 1.2: The simple `spinlock`

For a detailed description of the mutual exclusion problem and recent research on mutual exclusion algorithms, the readers are referred to [2].

1.2.2 Non-blocking Synchronization

Non-blocking synchronization is an alternative to mutual exclusion for implementing shared data objects. Shared data objects with non-blocking synchronization do not use mutual exclusion and do not require any communication with the kernel. Rather, they rely on hardware atomic primitives such as Compare-and-Swap or the pair Load-Link and Store-Conditional.

An implementation of a shared data object is called non-blocking if first it supports concurrency: several processes can perform operations on the shared data object concurrently; and moreover if it ensures that at any point of time *some/all* of the non-fault concurrent processes will complete their operations on the object in a bounded time regardless of the speed or status of other processes. If an implementation guarantees progress of some non-fault processes, it is called lock-free; if it guarantees progress of all non-fault processes, it is called wait-free. This requirement rules out the use of locks for non-blocking synchronization: if a process crashes while holding a lock, no process waiting for the lock can make any progress.

Compared with mutual exclusion, non-blocking synchronization has the following significant advantages:

1. it avoids lock convoying effects [13]: if a process holding a lock is preempted or delayed, any other process waiting for the lock is unable to perform any

useful work until the process holding the locks has finished its access to the shared object.

2. it provides high fault tolerance. By the definition of non-blocking synchronization, failures of processes should never corrupt the shared data objects. When using mutual exclusion, a process which dies during modifying a shared object in its critical section might leave the shared object in an invalid state. Some kind of fault recovery technique must be used to recover the object then.

3. it eliminates deadlock scenarios, where two or more tasks are waiting for locks held by the other.

4. it does not give priority inversion scenarios. A description of this problem is present in section 1.3 later.

1.2.3 Performance of Synchronization

The performance of synchronization is one of the bottlenecks for the performance of applications running on shared memory multiprocessors. The performance of applications depends on the parallelism of applications and the efficiency of communication between processes. In shared memory multiprocessors, processes communicate through synchronized operations on shared memory. Thus the performance of synchronization effects the efficiency of communication and determines consequently the performance of parallel applications.

The performance of synchronization on high performance shared memory multiprocessors is effected mainly by two factors: contention in the communication network and preemption.

For shared memory multiprocessors, contention in the communication network is an inherent consequence of sharing and becomes a critical architectural bottleneck. Contention happens when processors access the same memory location. Because the bandwidth of the interconnect network is limited, the access time on a memory location will increase dramatically when contention increases. Contention has been considered as one of the main factors affecting the performance of shared memory multiprocessors [9].

Results of many researchers [5, 11, 15] have shown that the performance of spinlocks degrades dramatically in the presence of contention on shared memory multiprocessors. The spin operation on a memory location is CPU intensive, performs no useful work and moreover generates an overwhelming amount of network traffic on shared memory multiprocessors.

A lot of efforts have been done to alleviate the detrimental effects of the spin operation. Segall and Rudolph [23] proposed a test-and-test-and-set algorithm to

reduce network traffic on cache-coherent shared memory multiprocessors. Anderson [5] and Agarwal and Cherian [1] independently proposed exponential backoff as a way of reducing contention of synchronization. The basic idea is to have each waiting process delay for some time between lock accesses in the spinlock algorithm.

Queuing the processes asking for the lock is another way to avoid network contention generated by lock operations. Anderson [5], Graunke and Thakkar [11] and Mellor-Crummey and Scott [17] independently proposed several queue lock algorithms. Two queue-based lock algorithms are specially developed by Magnusson, Landin and Hagersten [16] for cache coherent multiprocessor systems and have better performance than previous ones on such systems. Comparing with spinlocks, queue locks reduce the contention in interconnection networks significantly.

However, contention is not the only factor for the performance of synchronization in shared memory multiprocessors; preemption also effects the performance. In shared memory multiprocessors for high performance computing, parallel programs are expected to be run in a multiprogramming environment. Processes are scheduled with a UNIX time-sharing scheduling [7]. In a time-sharing scheduling, the process priority, assigned by users, is used to determined how many time-slices the process is able to run without relinquishing the processor [25, 26]. The scheduler in a system with time-sharing scheduling will adjust priorities of all processes dynamically to balance the response time of all the processes and improve the processor utilization. Therefore, in a time-sharing scheduling, a process may relinquish the processor after its time quota runs out even if its job is not finished. So a process which is assigned a high priority by some user can be preempted and relinquishes the processor to a process assigned a low priority. In such scheduling, processes with the same priority will have the same time quotas and run usually in round-robin manner. Besides running out of time quota, a process may be preempted at any point for other reasons: some kernel processes run for periods of time, even if the machine is used exclusively; background daemons run from time to time; or page faults and I/O interrupts take place.

For mutual exclusion, preemptions are responsible for the lock convoying problems [13]: if a process holding a lock is preempted or delayed, any other process waiting for the lock is unable to perform any useful work until the process holding the locks has finished its access to the shared object. With lock-based synchronization, a parallel program as a whole slows down when one process is slowed because of the convoying effect. The performance of both spinlock and queue lock implementations degrades significantly [12].

To avoid the performance problem with preemption, people advocate preemption-safe and scheduler-conscious locks [12, 18] and non-blocking synchronization. Preemption-safe and scheduler-conscious locks require cooperation between processes and operating system kernels to avoid the problem of lock convoying. The differ-

ence between preemption-safe locks and scheduler-conscious locks is that processes involved in preemption-safe locks interact with the scheduler of operating system with their own states; processes involved in scheduler-conscious locks can get the information of the state of both their own and others and modify them. Non-blocking synchronization does not need such information available at the kernel level. Instead of kernel support, they rely on advanced hardware atomic primitives. For computation with non-blocking synchronization, if a process is preempted, it will not block any other process and will not slow down any other process.

1.3 Real-time Shared Memory Multiprocessors

Although performance is desired, real-time systems require mostly predictable behavior. The correctness of real-time systems depends not only on the correct computational results but also on the timing of the computational results [14]. The structure of real-time shared memory multiprocessors is the same as the one of general high performance ones. The extra architectural requirement of real-time systems is that they should provide predictable performance for computation, I/O bus throughput and memory bandwidth.

The main differences between shared memory systems for high performance computing and ones for real-time computing are the scheduling policies that they use and the characteristics of the task sets run on them. High performance systems mostly use time-sharing scheduling; real-time systems use priority-based scheduling. In a priority-based scheduling, a task will only be preempted by high priority tasks and will only relinquish the processor after it finishes its computation. The characteristics are different between tasks running on high performance shared memory systems and real-time systems. Most tasks on high performance systems are aperiodic; most tasks on real-time systems are periodic. In high performance systems, the users are usually interested in the average execution times of tasks run on them; in real-time systems, the worst case execution times of tasks are of interest to them. In high performance systems, tasks do not have release times and deadlines; in real-time systems, tasks do have deadlines and may have release times. All these characteristics of tasks in real-time systems are required by the systems to deliver predictable behavior.

Predictable behavior is also expected for synchronization in real-time shared memory systems. However, without special consideration, synchronization can introduce unbounded or unpredictable worst case behavior into real-time systems.

Priority inversion problem with mutual exclusion can introduce unpredictable behavior into real-time systems. As an example, two tasks might share a resource which is protected with a lock : one has a high priority and one low priority. When the high priority task wants to get the lock and access the shared resource, it finds

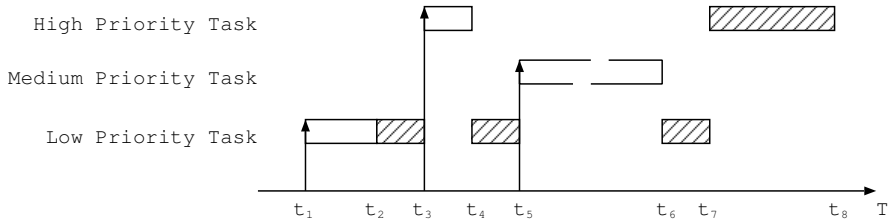


Figure 1.3: A scenario of priority inversion

that the lock is held by the low priority task. Then, the high priority task has to relinquish the processor to the low priority task to finish its operation on the shared resource. If at this time another medium priority task who does not need to access the shared resource to finish its job comes, the low priority must relinquish the processor to it because of priority scheduling policy. This is a priority inversion problem scenario: the high priority task has to wait for a medium priority task, even if they do not share any resources except the processor. The situation is depicted in Figure 1.3.

At time t_1, the low priority task is released and begins to execute. It enters critical section at time t_2. At time t_3, the high priority task is released and preempts the low priority task. At time t_4, the high priority task tries to access the critical section and finds that the lock is hold by the low priority task. The high priority task relinquishes the processor to let the low priority task finish its operation inside the critical section. At time t_5, the medium priority task is released. As the current running task is the low priority task who has a lower priority than the medium priority task, by priority scheduling policy, the medium priority preempts the low priority task and in turn preempts the high priority task. The time that the high priority task begin to access the critical section is determined by not only the critical access time of the low priority task but also the number, the periods and the execution times of all tasks who might preempt the low priority task.

A lot of research work has been performed in order to minimize the effect of priority inversion. The priority inheritance protocol (PIP) and priority ceiling protocol (PCP) [8, 20, 24] are proposed for uniprocessor real-time systems. In [6], a stack-based resource allocation policy for real-time systems is described by Baker. For real-time multiprocessor systems, Rajkumar et al. have proposed the distributed priority ceiling protocol (DPCP) [21] (for message passing systems) and the multiprocessor priority ceiling protocol (MPCP) [20] (for shared memory systems).

Non-blocking synchronization does not suffer from the priority inversion problem. There are other problems associated with non-blocking synchronization when we apply them in real-time systems, for example: the “enabled late-write” problem [22] and unbounded retry loops of lock-free synchronization. The “enabled late-write”

problem can happen when using only Read/Write memory operations to implement non-blocking synchronization in priority based real-time systems. The problem was defined by Ramamurthy, Moir and Anderson in [22]. The "enabled late-write" problem arises when a low priority task A is preempted while it is about to write to a memory position, and is preempted by other tasks that access and modify the same memory position. When task A resumes running, it overwrites the previous "fresh" value with an "old" one. To solve the "enabled late-write" problem, Anderson et al. propose a majority voting scheme in [22]. Beside the "enabled late-write" problem, we face other problems when applying lock-free synchronization in real-time systems. Lock-free synchronization usually uses a retry loop of read-compute-modify cycle. The number of loops for a process to finish its access on a shared object with lock-free synchronization is potentially unbounded. The unbounded retry loops lead to infinite execution times in the worst case, which make tasks infeasible for scheduling. In [3, 4], Anderson, Ramamurthy and Jeffay addressed how to apply lock-free synchronization into real-time uniprocessor systems.

1.4 Contributions

In this thesis, we explore the performance impact of non-blocking synchronization in high performance shared memory multiprocessors and the applicability of non-blocking synchronization in real-time systems.

For *high performance systems*, throughput and scalability are the main factors in their performance matrices. We investigate how to design high performance non-blocking share data structures and how to use non-blocking synchronization to gain performance.

In chapter 2, we present a non-blocking concurrent FIFO queue algorithm for shared memory multiprocessors. We introduce an algorithmic mechanism to restrict contention on key variables generated by concurrent enqueue and/or dequeue operations; we also give a new solution to the pointer recycling problem. Experimental results show that our algorithm considerably outperforms the previous best-known non-blocking algorithms and lock-based algorithm in both UMA and ccNUMA shared memory machines with respect to both dedicated and multiprogramming workloads.

In chapter 3, we study the performance impact of different synchronization mechanisms on parallel applications running on modern cache-coherent shared memory multiprocessors. We investigate how performance and speedup in high performance shared memory applications would be affected by using non-blocking rather than blocking synchronization and how to efficiently transform lock-based synchronization mechanisms into non-blocking ones. Our results show that for certain applications non-blocking synchronization has significant performance advantage over mu-

tual exclusions; non-blocking synchronization never slows down applications; there exist simple transformations between frequently used lock-based synchronization mechanisms to non-blocking ones.

In chapter 4, we implement a parallel Quicksort algorithm with techniques that improves the cache behavior of our implementation and at the same time supports non-blocking synchronization. We believe that as the gap between the computation speed of microprocessors and the access speed of memory becomes larger and larger, we have to consider carefully the memory access and synchronization in algorithm design for cache-coherent shared memory multiprocessors. Our experimental results agree with our claim and show that parallel Quicksort can outperform Sample Sort on cache-coherent shared memory multiprocessors, where Sample Sort has been long thought to be the best, general parallel sorting algorithms.

In *real-time systems*, predictability is the dominant factor in performance matrices. We investigate how to achieve predictability of non-blocking synchronization in real-time systems.

We present a non-blocking data sharing scheme for multiprocessor real-time systems in chapter 5. The scheme extends previous single-writer multiple-reader schemes by allowing multiple writers. A simple efficient memory management scheme is embedded to limit the number of retries of reader operations and has efficient space utilization.

In chapter 6, we present algorithmic implementations of the wait-free queue classes of the Real-time Specification for Java. These implementations are designed to have the unidirectional nature of these queues in mind and they are more efficient, with respect to space, compared to previous wait-free implementations, without losing in time complexity. We also give a new solution to the "enabled late-write" problem mentioned in Section 1.3. The wait-free queue classes proposed in the Real-time Specification for Java are of general interest to any real-time synchronization system where hard real-time tasks have to synchronize with soft or even non real-time tasks.

In chapter 7, we address on the problem of retry loops of lock-free synchronization, mentioned in Section 1.3. We analyze the cause of interference for lock-free synchronization in real-time shared memory multiprocessors. We propose a retry-level based protocol to bounds the execution time of operations on lock-free shared data objects in real-time shared memory *multiprocessors*. We analyze the worst case behaviors of the proposed methods. To the best of our knowledge, we are the first to show how to apply lock-free synchronization in real-time shared memory multiprocessors.

Bibliography

[1] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 396–406. IEEE Computer Society Press, June 1989.

[2] J. H. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing, special issue celebrating the 20th anniversary of PODC*, page to appear.

[3] J. H. Anderson and S. Ramamurthy. Using lock-free objects in hard real-time applications. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*, pages 272–272. ACM, Aug. 1995.

[4] J. H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems*, 15(1):134–165, Feb. 1997.

[5] T. Anderson. The performance implications of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1:6–16, Jan. 1990.

[6] T. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, 3(1), Mar. 1991.

[7] J. M. Barton and N. Bitar. A scalable multi-discipline, multiple-processor scheduling framework for IRIX. In *IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 45–69. Springer-Verlag, 1995. Lecture Notes in Computer Science. vol. 949.

[8] M.-I. Chen and K.-J. Lin. Dynamic priority ceilings: a concurrency control protocol for real-time systems. *Real-Time Systems*, 2(4):325–346, 1990.

[9] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, Nov. 1997.

[10] A. Gottlieb, R. Grishman, C. P. Krukal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer – designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–190, Feb. 1983.

[11] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23(6):60–69, June 1990.

[12] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, Feb. 1997.

[13] H. Kopetz and J. 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, Raleigh-Durham, NC, Dec. 1993. IEEE Computer Society Press.

[14] C. M. Krishna and K. G. Shin. *Real-Time Systems*. McGraw-Hill, 1997.

[15] B.-H. Lim and A. Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 25–35. ACM press, Oct. 1994.

[16] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 165–171, Los Alamitos, CA, USA, Apr. 1994. IEEE Computer Society Press.

[17] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, Feb. 1991.

[18] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 25 May 1998.

[19] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing, ICPP'85*, pages 764–771, University Park, Pennsylvania, Aug. 1985. IEEE. IBM T. J. Watson Research Center, Yorktown Heights, NY.

[20] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 116–123, Paris (France), May–June 1990. IEEE, IEEE Computer Society Press.

[21] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the 1988 IEEE Real-Time Systems Symposium*, pages 259–269, 1988.

[22] S. Ramamurthy, M. Moir, and J. H. Anderson. Real-time object sharing with minimal system support. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 233–242. ACM, May 1996.

[23] Z. Segall and L. Rudolph. Dynamic decentralized cache schemes for mimd parallel processors. In *Proceeding of 11th Annual International Symposium on Computer Architecture*, pages 340–347. ACM Press, June 1984.

[24] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept. 1990.

[25] Silicon Graphics, Inc. *System Interface Guide*. Silicon Graphics, Inc., 2000.

[26] Sun Microsystems, Inc. *System Interface Guide*. SUN Microsystems, Inc., 2000.

Chapter 2

A Simple, Fast and Scalable Non-Blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems

This paper is an extended version of the paper appeared in the *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '01)*.

A Simple, Fast and Scalable Non-Blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems*

Philippas Tsigas and Yi Zhang

Department of Computing Science,
Chalmers University of Technology,
SE-412 60, Gothenburg, Sweden

Abstract

A non-blocking FIFO queue algorithm for multiprocessor shared memory systems is presented in this paper. The algorithm is very simple, fast and scales very well in both symmetric and non-symmetric multiprocessor shared memory systems. Experiments on a 64-node SUN Enterprise 10000 – a symmetric multiprocessor system – and on a 64-node SGI Origin 2000 – a cache coherent non-uniform memory access multiprocessor system – indicate that our algorithm considerably outperforms the best of the known alternatives in both multiprocessors in any level of multiprogramming. This work introduces two new, simple algorithmic mechanisms. The first lowers the contention to key variables used by the concurrent enqueue and/or dequeue operations which consequently results in the good performance of the algorithm; the second deals with the pointer recycling problem, an inconsistency problem that all non-blocking algorithms based on the Compare-and-Swap synchronisation primitive have to address. In our construction we selected to use Compare-and-Swap since Compare-and-Swap is an atomic primitive that scales well under contention and either is supported by modern multiprocessors or can be implemented efficiently on them.

1 Introduction

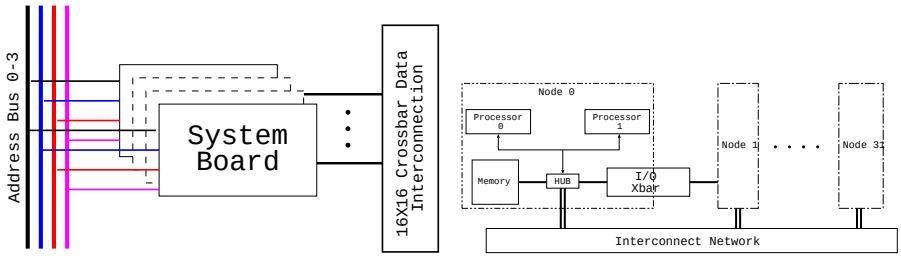
Concurrent FIFO queue data structures are fundamental data structures used in many applications, algorithms and operating systems for multiprocessor systems. To

* This work is partially supported by: i) the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se) supported by the Swedish Foundation for Strategic Research and ii) the Swedish Research Council for Engineering Sciences.

protect the integrity of the shared queue, concurrent operations that have been created either by a parallel application or by the operating system have to be synchronised. Typically, algorithms for concurrent data structures, including FIFO queues, use some form of mutual exclusions (locking) to synchronise concurrent operations. Mutual exclusions protect the consistency of the concurrent data structure by allowing only one process (the holder of the lock of the data structure) at a time to access the data structure and by blocking all the other processes that try to access the concurrent data structure at the same time. Mutual exclusions and, in general, other solutions that introduce blocking are penalised by locking that introduces priority inversion, deadlock scenarios and performance bottlenecks. The time that a process spends blocked while waiting to get access to the critical section can form a substantial part of the algorithm execution time [5, 9, 10, 14]. There are two main reasons that locking is so expensive. The first reason is the lock convoying effect that blocking synchronisation suffers from: if a process holding the lock is preempted, any other process waiting for the lock is unable to perform any useful work until that the process holding the locks is scheduled. When we taking into account that the multiprocessor system running our program is used in a multiprogramming environment, convoying effects can become even serious. The second is that locking tends to produce a large amount of memory and interconnection network contention, locks become hot memory spots. Researchers in the field first designed different lock implementations that lower the contention when the system is in a high congestion situation, and they give different execution times under different contention instances. But on the other hand the overhead due to blocking remains. To address the problems that arise from blocking researchers have proposed non-blocking implementations of shared data structures. Non-blocking implementation of shared data objects is a new alternative approach to the problem of designing scalable shared data objects for multiprocessor systems. Non-blocking implementations allow multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion to accomplish this. Since in non-blocking implementations of shared data structures one process is not allowed to block another process, non-blocking shared data structures have the following significant advantages over lock-based ones:

(1) they avoid lock convoys and contention points (locks).
(2) they provide high fault tolerance (processor failures will never corrupt shared data objects) and eliminates deadlock scenarios, where two or more tasks are waiting for locks held by the other.
(3) they do not give priority inversion scenarios.

Among all the innovative architectures for multiprocessor systems that have been proposed the last forty years shared memory multiprocessor architectures are gaining a central place in high performance computing. Over the last decade many shared memory multiprocessors have been built and almost all major computer vendors develop and offer shared memory multiprocessor systems nowadays. There are two main



(a) The architecture of the SUN Enterprise 10000

(b) The architecture of the Origin 2000

Figure 1: Architectures

classes of shared memory multiprocessors: the Cache-Coherent Nonuniform Memory Access multiprocessors (ccNUMA) and the symmetric or Uniform Memory Access (UMA) multiprocessors. Their differences come from the architectural philosophy they are based on. In symmetric shared memory multiprocessors every processor has its own cache and all the processors and memory modules attach to the same interconnect, which is a shared bus. ccNUMA is a relatively new system topology that is the foundation for next-generation shared memory multiprocessor systems. As in UMA systems, ccNUMA systems maintain a unified, global coherent memory and all resources are managed by a single copy of the operating system. A hardware-based cache coherency scheme ensures that data held in memory is consistent on a system-wide basis. In contrast to symmetric shared memory multiprocessor systems in which all memory accesses are equal in latency, in ccNUMA systems, memory latencies are not all equal, or uniform (hence, the name - Non-Uniform Memory Access). Accesses to memory addresses located on "far" modules take longer than those made to "local" memory.

This paper addresses the problem of designing scalable, practical FIFO queues for shared memory multiprocessor systems. First we present a non-blocking FIFO queue algorithm. The algorithm is very simple, it algorithmically implements the FIFO queue as a circular array and introduces two new algorithmic mechanisms that we believe can be of general use in the design of efficient non-blocking algorithms for multiprocessor systems. The first mechanism restricts contention to key variables generated by concurrent enqueue and/or dequeue operations in low levels; contention to shared variables degrades performance not only in memory banks where the variables are located but also in the processor-memory interconnection network. The second algorithmic mechanism that this paper introduces is a mechanism that deals with the pointer recycling (also known as ABA) problem, a problem that all non-blocking algorithms based on the Compare-and-Swap primitive have to address. The performance improvements are due to these two mechanisms and to its simplicity that comes from the simplicity and richness of the structure of circular arrays. We have

selected to use the Compare-and-Swap primitive since it scales well under contention and either is supported by modern multiprocessors or can be implemented efficiently on them. Last, we evaluate the performance of our algorithm on a 64-node SUN Enterprise 10000 multiprocessor and a 64-node SGI Origin 2000. The SUN system is a Uniform Memory Access (UMA) multiprocessor system while the SGI system is a Cache-Coherent Nonuniform Memory Access (ccNUMA) one; SUN Enterprise 10000 supports the Compare-and-Swap while SGI Origin 2000 does not. The experiments clearly indicate that our algorithm considerably outperforms the best of the known alternatives in both UMA and ccNUMA machines with respect to both dedicated and multiprogramming workloads. Second, the experimental results also give a better insight into the performance and scalability of non-blocking algorithms in both UMA and ccNUMA large scale multiprocessors with respect to dedicated and multiprogramming workloads, and they confirm that non-blocking algorithms can perform better than blocking on both UMA and ccNUMA large scale multiprocessors, and that their performance and scalability increases as multiprogramming increases.

Concurrent FIFO queue data structures are fundamental data structures used in many multiprocessor programs and algorithms and, as can be expected, many researchers have proposed non-blocking implementations for them. Lamport [6] introduced a wait-free queue that does not allow more than one enqueue operation or dequeue operation at a time. Herlihy and Wing in [4] presented an algorithm for a non-blocking linearisable FIFO queue which requires an infinite array. Prakash, Lee and Johnson in [11] presented a non-blocking and linearisable queue algorithm based on a singly-linked list. Stone describes a non-blocking algorithm based on a circular queue. Massalin and Pu [8] present a non-blocking array-based queue which requires the Double-Word-Compare-and-Swap atomic primitive that is available only on some members of the Motorola 68000 family of processors. Valois in [12] presents a non-blocking queue algorithm together with several other non-blocking data structures, his queue is an array-based one. Michael and Scott in [10] presented a nonblocking queue based on a singly-link list, which is the most efficient and scalable non-blocking algorithm compared with the other algorithms mentioned above.

The remainder of the paper is organised as follows. In Section 2 we give a brief introduction to shared memory multiprocessors. Section 3 presents our algorithm together with a proof sketch. In Section 4, the performance evaluation of our algorithm is presented. The paper concludes with Section 5.

2 Shared Memory Multiprocessors: Architecture and Synchronization

There are two main classes of shared memory multiprocessors: the Cache-Coherent Nonuniform Memory Access (ccNUMA) multiprocessors and the symmetric multiprocessors. The most familiar design for shared memory multiprocessor systems is the "fixed bus" or shared-bus multiprocessor system. The bus is a path, shared by all processors, but usable only by one at a time to handle transfers from CPU to/from memory. By communicating on the bus, all CPUs share all memory requests, and can synchronise their local cache memories. Such systems include the Silicon Graphics Challenge/Onyx systems, OCTANE, Sun's Enterprise (300-6000), Digital's 8400, and many others - most server vendors offer such systems.

Central Crossbar Mainframes and supercomputers have often used a crossbar "switch" to build shared multiprocessor systems with higher bandwidth than feasible with busses, where the switch supports multiple concurrent paths to be active at once. Such systems include most mainframes, the CRAY T90, and Sun's new Enterprise 10000. Figure 1(a) graphically describes the architecture of the new SUN Enterprise 10000. Shared-bus and central crossbar systems are usually called UMAs, or Uniform Memory Access systems, that is, any CPU is equally distant in time from all memory locations. Uniform memory access shared memory multiprocessors dominate the server market and are becoming more common on the desktop. The price of these systems rise quite fast as the number of processors increases.

```
Boolean Compare-and-Swap(WORD *mem, register WORD old, new)
{
    WORD temp;
    temp = *mem;
    if (temp == old){
        *mem = new;
        return TRUE;
    }else
        return FALSE;
}
```

Figure 2: The Compare-and-Swap primitive

ccNUMA is a relatively new system topology that is the foundation for many next-generation shared memory multiprocessor systems. Based on "commodity" processing modules and a distributed, but unified, coherent memory system, ccNUMA extends the power and performance of shared memory multiprocessor systems while preserving the shared memory programming model. As in UMA systems, ccNUMA systems maintain a unified, global coherent memory and all resources are managed by a single copy of the operating system. A hardware-based cache coherency scheme ensures

```
LL(p_i , O)
{
    Pset(O) := Pset(O) ∪ {p_i}
    return value(O)
}
```

```
SC( p_i, v, O )
{
  if p_i ∈ Pset(O)
     value(O):= v
     Pset(O) := ∅
     return TRUE
  else
     return FALSE
}
```

Figure 3: The Load-Linked/Store-Conditional primitives

that data held in memory is consistent on a system-wide basis. I/O and memory scale linearly as processing modules are added, and there is no single backplane bus. The nodes are connected by an interconnect, whose speed and nature varies widely. Normally, the memory "near" a CPU can be accessed faster than memory locations that are "further away". This attribute leads to the "Non" in Non-Uniform. ccNUMA systems include the Convex Exemplar, Sequent NUMA-Q, Silicon Graphics/CRAY S2MP (Origin and Onyx2). In the Silicon Graphics Origin 2000 system a dual-processor node is connected to a router. The routers are connected with a fat hypercube interconnect, Figure 1(b) graphically describes the architecture.

ccNUMA systems are expected to become the dominant systems on large high performance systems over the next few years. The reasons are: i) they scale up to as many processors as needed. ii) they support the cache-coherent globally addressable memory model. iii) their entry level and incremental costs are relatively low.

A widely available hardware synchronisation primitive that can be found on many common architectures is Compare-and-Swap, (CAS). The Compare-and-Swap primitive takes as arguments the pointer to a memory location, and old and new values. As it can be seen from Figure 2 that describes the specification of the Compare-and-Swap primitive, it automatically checks the contents of the memory location that the pointer points to, and if it is equal to the old value, updates the pointer to the new value. In either case, it returns a boolean value that indicates whether it has succeeded or not. The IBM System 370 was the first computer system that introduced Compare-and-Swap. SUN Enterprise 10000 is one of the systems that support this hardware primitive. Some newer architectures, SGI Origin 2000 included, introduce the Load-Linked/Store-Conditional instruction which can be implemented by the Compare-and-Swap primitive. The Load-Linked/Store-Conditional is comprised by two simpler operations, the Load-Linked and the Store-Conditional one. The Load-Linked, (LL) loads a word from the memory to a register. The matching Store-Conditional, (SC) stores back possibly a new value into the memory word, unless the value at the memory word has been modified in the meantime by another process. If the word has not been modified, the store succeeds and a TRUE is returned. Otherwise the, Store-Conditional fails, the memory is not modified, and a FALSE is returned. The

specification of this operation is shown in Figure 3. For more information on the SGI Origin 2000 and the SUN ENTERPRISE the reader is referred to [2, 7] and [1], respectively.

The Compare-and-Swap primitive though gives rise to the pointer recycling (also known as ABA) problem. The ABA problem arises when a process p reads the value A from a shared memory location, computes a new value based on A, and uses Compare-and-Swap updates the same memory location after checking that the value in this memory location is still A and mistakenly concluding that there was no operation that changed the value to this memory location in the meantime. But between the read and the Compare-and-Swap operation, other processes may have changed the context of the memory location from A to B and then back to A again. In this scenario the Compare-and-Swap primitive fails to detect the existence of operations that changed the value of the memory location; in many non-blocking implementations of shared data structures this is something that we would like to be able to detect without having to use the `Read-Modify-Write` operation that has very high latency and creates high contention. A common solution to the ABA problem is to split the shared memory location into two parts: a part for a modification counter and a part for the data. In this way when a process updates the memory location, it also increments the counter in the same atomic operation. There are several drawbacks of such a solution. The first is that the real word-length decreases as the counter now occupies part of the word. The second is that when the counter rounds there is a possibility for the ABA scenario to occur, especially in systems with many, and with fast processors such as the systems that we are studying. In this paper we present a new, very simple efficient technique to overcome the ABA problem; the technique is described in the next section together with the algorithm.

```
Boolean Compare-and-Swap(WORD *mem, register WORD old, new)
{
    WORD temp;
    do
    {
     temp = LL(mem);
     if  (temp != old)
         return FALSE;
    }while(!SC(mem,new));
    return TRUE;
}
```

Figure 4: Emulating Compare-and-Swap from Load-Linked/Store-Conditional

3 The Algorithm

3.1 Contention, consistency and non-blocking synchronisation

During the design phase of any efficient non-blocking data structure, a large effort is spent on guaranteeing the consistency of the data structure without generating many interconnection transactions. The reason for this is that the performance of any synchronisation protocol for multiprocessor systems heavily depends on the interconnection transactions that they generate. A high number of transactions causes a degradation in the performance of memory banks and the processor/memory interconnection network.

As a first step, when designing the algorithm presented here, we tried to use simple synchronisation instructions (primitives), with low latency, that do not generate a lot of coherent traffic but are still powerful enough to support the high-level synchronisation needed for the non-blocking implementation of a FIFO queue. In the construction described in this paper, we have selected to use the Compare-and-Swap atomic primitive since it meets the three important goals that we were looking for. First, it is a quite powerful primitive and when used together with simple read and write registers is sufficient for building any non-blocking implementation of any "interesting" shared data-structure [3]. Second, it is either supported by modern multiprocessors or can be implemented efficiently on them. Finally, it does not generate a lot of coherent traffic. The only problem with the Compare-and-Swap primitive is that, it gives rise to the pointer recycling (also known as ABA) problem. As a second step, we have tried when designing the algorithm presented here to use the Compare-and-Swap operation as little as possible. The Compare-and-Swap operation is an efficient synchronisation operation and its latency increases linearly with the number of processors that use it concurrently, but still it is a transactional one that generates coherent traffic. On the other hand Read or Write operations require a single message in the interconnection network and do not generate much coherent traffic. As a third step, we propose a simple new solution that overcomes the ABA problem that does not generate a lot of coherent traffic and does not restrict the size of the queue.

Figure 6 and Figure 7 present commented pseudo-code for the new non-blocking queue algorithm. The algorithm is simple and practical, and we were surprised not to find it in the literature. The non-blocking queue is algorithmically implemented as a circular array with a *head* and a *tail* pointer A ghost copy of `NULL` has been introduced in order to help us to avoid the ABA problem as we are going to see at the end of this section. During the design phase of the algorithm we realised that: i) we could use the structural properties of a circular array to reduce the number of Compare-and-Swap operations that our algorithm uses as well as to overcome more efficiently the ABA problem and ii) all previous non-blocking implementations were

trying to guarantee that the *tail* and the *head* pointers always show the real head and tail of the queue but by allowing the *tail* and *head* pointers to lag behind we could even further reduce the number of Compare-and-Swap asymptotically close to optimal. We assume that enqueue operations inserts data at the tail of the queue and dequeue operations remove data from the head of the queue if the queue is not empty. In the algorithm presented here we allow the head and the tail pointers to lag at most m behind the actual head and tail of the queue, in this way only one every m operations has to consistently adjust the tail or head pointer by performing a Compare-and-Swap operation. Since we implement the queue as a circular array, each queue operation that successfully enqueues or dequeues data knows the index of the array where the data have been placed, or have been taken from, respectively; if this index can be divided by m, then the operation will try to update the head/tail of the queue, otherwise it will skip the step of updating the head/tail and let the head/tail lag behind the actual head/tail. In this way, the amortised number of Compare-and-Swap operations for an enqueue or dequeue operation is only $1 + 1/m$ compare-and-swap operation per enqueue/dequeue operation is necessary. The drawback that such a technique introduces is that each operation on average will need $m/2$ more read operations to find the actual head or tail of the queue; but if we fix m so that the latency of $(m-1)/m$ Compare-and-Swap operations is larger than the latency of $m/2$ read operations, there will be a performance gain from the algorithm, and these performance gains will increase as the number of processes increases.

It is definitely true that array-based queues are inferior to link-based queues, because they require inflexible maximum queue size. But, on the other hand, they do not require memory management schemes that link-based queue implementations need and they benefit from spatial locality significantly more than link-based queues. Taking these into account and having a a simple, fast and practical implementation in mind we decided to use a cyclical-array in our construction.

We have used the Compare-and-Swap primitive to atomically swing the head and tail pointers from their current value to a new one. For the SGI Origin 2000 system we had to emulate the Compare-and-Swap atomic primitive with the Load-linked/Store-conditional instruction; this implementation is shown in Figure 4. However, using Compare-and-Swap in this manner is susceptible to the ABA problem. In the past researchers have proposed to attach a counter to each pointer, reducing in this way the size of the memory that these pointers can point at efficiently. In this paper we observe that the circular array itself works like a counter mod l where l is the length of the cyclical array, and we can fix l to be arbitrary large. In this way by designing the queue as a circular array we overcome the ABA problem on the head and tail of the queue the same way the counters do but without having to attach expensive counters to the pointers, that restrict the pointer size. From now on, when an enqueue operation takes place, the tail changes in one direction and goes back to zero when it reaches the end of the array. Henceforth, the tail will change back to the

same old value after the shared object finishes l enqueue operations and not after two successive operations (exactly as when using a counter mod l). The same also holds for the dequeue operations.

```
# MAXNUM is l, the length of the cyclical
structure Queue
{
   head: unsigned integer,
   nodes: array[0..MAXNUM+1] of pointer,
   tail: unsigned integer,
}

newQueue(): pointer to Queue
  Queue *temp;
  temp = (Queue *) malloc( sizeof(Queue));
  temp->head = 0;
  temp->tail = 1;
  for (i=0;i<=MAXNUM;i++)
      #NULL(0) and NULL(1) mean empty
      temp->nodes[i]=NULL(0);
  temp->nodes[0] = NULL(1);
  return temp;
```

Figure 5: Initialisation

The atomic operations on the array are other potential places where the ABA problem can take place giving rise to the following scenario:

When the array is (almost) empty,

(1) The array location x is the actual tail of the queue and its content is $Null$ [1]
(2) Processes a and b want to enqueue data and process c want to do a dequeue operation. Processes a and b find the actual tail.
(3) Process a enqueues data and updates the content of location x with Compare-and-Swap. Since the content of x is $Null$, a succeeds
(4) Process c dequeues data and updates the content of location x to $Null$, changing also the pointer head
(5) Process b enqueues data and updates the contents of location x with Compare-and-Swap. Since the content of x is $Null$, b incorrectly succeeds to enqueue a non-active cell in the queue.

Or when the array is (almost) full,

(1) The array location x is the actual head of the queue and its content is C.

[1] the cell is empty.

(2) Processes a and b want to do dequeue operations and process c want to enqueue data. Processes a and b find the actual head and read the content C of location x out.
(3) Process a dequeues data and updates the content of location x to $Null$ with the use of Compare-and-Swap. Since the contents of x is C, a succeeds.
(4) Process c comes and enqueues data C and updates the content of location x to C, changing also the pointer tail.
(5) Process b dequeues data and updates the contents of location x to $Null$ with the use of Compare-and-Swap. Since the content of x is C, b succeeds to dequeue a data not in a FIFO order.

In order to overcome these specific ABA instances instead of using a counter with all the negative side-effects, we introduce a new simple mechanism that we were surprised not to find in the literature. The idea is very simple: we use one bit of each word as a tag. When the tag bit is 0, the value is called normal copy. When the tag bit is 1, we call the value is a ghost copy of original one. When using CAS, if the old value is a normal copy of $NULL$, we will swap in a ghost copy of the new data; otherwise a normal copy of the new data. If the old value is a normal copy of a non-$NULL$ value, a normal copy of $NULL$ will be swap in; otherwise a ghost copy. Now let us take the $NULL$ as an example to demonstrate how to deal with the ABA problem. Instead of using one value to describe that an entry in the array is empty, now we have two, $NULL(0)$, the normal copy, and $NULL(1)$, the ghost copy. When a processor dequeues an item, it will swap into the cell one of the two $NULLs$ in such a way that two consecutive dequeue operations on the same cell give different $NULL$ values to the cell.

Returning to the first ABA scenario described above, the scenario would now look like this:

(1) Array location x is the actual tail and it's content is $NULL(0)$
(2) Processes a and b find the actual tail, ie. location x and the old value of the location, $NULL(0)$.
(3) Process a enqueues data and updates the content of location x with a Compare-and-Swap operation. Since $x's$ content is $NULL(0)$, a succeeds. The value swap in by a is a ghost copy.
(4) Process c dequeues data and updates the content of location x to $NULL(1)$, because the old value is a ghost copy.
(5) process b enqueues data and updates the content of location x with Compare-and-Swap. As the content is $NULL(1)$, b fails in this turn.

With this mechanism the ABA scenario that was taking place before, when a process was preempted by only one other process, now changes to an $ABA'B'A$ scenario. The $ABA'B'A$ scenario is still a pointer recycling problem, but in order to take place l

```
 Enqueue(t: pointer to Queue, newnode: pointer to data type):Boolean
2loop
     te = t->tail;          #read the tail
4    #assume the tail points the actual tail
     ate = te;
6    tt = t->nodes[ate];
     #the next slot of the tail
8    temp = (ate + 1) & MAXNUM;
     #Now we want to find the actual tail
10   while (tt != NULL(0) AND tt != NULL(1)) do
         #check tail's consistency
12       if (te != t->tail) break;
         #if tail meet head, the queue is probably full.
14       if (temp == t->head) break;
         #now check the next cell
16       tt = t->nodes[temp];
         ate = temp;
18       temp = (ate + 1) & MAXNUM;
     end while
20   #check the tail's consistency
     if (te != t->tail) continue;
22   #check whether Queue is full
     if (temp == t->head)
24       ate = (temp + 1) & MAXNUM;
         tt = t->nodes[ate];
26       #the cell after head is OCCUPIED
         if (tt != NULL(0) AND tt != NULL(1))
28           return FAILURE;       #Queue Full
         #help the dequeue to update head
30       cas(&t->head,temp,ate);
         #try enqueue again
32       continue;
     end if
34   if (tt == NULL(1))
        tnew = newnode | 0x80000000;
36   else
        tnew = newnode;
38   #check the tail consistency
     if (te != t->tail) continue;
40   #get the actual tail and try enqueue data
     if (cas(&(t->nodes[ate]),tt,newnode))
42      if (temp%2==0)    #enqueue has succed
           cas(&(t->tail),te,temp);
44      return SUCCESS;
     end if
46endloop
```

Figure 6: The enqueue operation

dequeue operations are needed to take the system from A to B and subsequently to A', after that l more dequeue operations are needed in order to take the system from A' to B' and then to A. Moreover, all these operations have to take place while the process that will experience the pointer recycling is preempted. Taking into account that l is an arbitrary large number, the probability that the above $ABA'B'A$ scenario can happen can go as close to 0 as we want [2].

The above sketches a proof of the following lemma:

Lemma 1 *The algorithm does not give rise to the pointer recycling problem, if an enque or dequeue operation can not be preempted by more than l operations, l is an arbitrary large number.*

For the rest of this paper we assume that we have selected l to be large enough not to give rise to the pointer recycling problem in our system.

Our queue is a concurrent non-blocking FIFO implementation. `Enqueue` operations can overlap with other `enqueue` and `dequeue` operation of the same queue. We use linearisability [4] as correctness criterion. The accessing of the shared object is modelled by a history h. A history h is a finite (or not) sequence of operation invocation and response events. Any response event is preceded by the corresponding invocation event. For our case there are two different operations that can be invoked, an Enqueue operation or a Dequeue operation. An operation is called complete if there is a response event in the same history h; otherwise, it is said to be pending. A history is called complete if all its operations are complete. In a global time model each operation q "occupies" a time interval $[s_q, f_q]$ on one linear time axis ($s_q < f_q$); we can think of s_q and f_q as the starting and finishing time instants of q. During this time interval the operation is said to be *pending*. There exists a precedence relation on operations in history denoted by $<_h$, which is a strict partial order: $q_1 <_h q_2$ means that q_1 ends before q_2 starts; Operations incomparable under $<_h$ are called *overlapping*. A complete history h is linearisable if the partial order $<_h$ on its operations can be extended to a total order $\rightarrow_{\mathrm{h}}$that respects the specification of the object [4].

We will prove that every execution of our implementation is linearisable and it is a FIFO queue. In order to do so, we will show that any possible history ($<_h$), produced by our implementation, can be extended to a total order ($\rightarrow_{\mathrm{h}}$) by using a "linearization point" for each operation. The "linearization point" of an operation is an atomic point on its execution, during which the operation takes effect. The "linearization point" for

[2] We should point out that the technique of using 2 different $NULL$ values can be extended to k different values, with more tag bits, requiring more than $k * l$ dequeue operations to preempt an operation in order to cause the pointer recycling problem. We think that the scheme with $2 * NULL$ values is simple enough and sufficient for the systems that we are looking at.

the enqueue operation is when statement 41 execute successfully and the "linearization point" for the dequeue operation is when statement 32 execute successfully. With the above "linearization points", we can map any partial order of enqueue and dequeue events into total order events. Furthermore any possible history, produced by our implementation, can be mapped to a history where operations use an auxiliary array that is not bounded on the right side. In order to simplify the proof we will use this new auxiliary array.

Lemma 2 *The proposed queue algorithm guarantees that enqueue operations enqueue data at consecutive array entries from left to right on this array.*

Proof: Assume that enqueue operations skip one entry of the array. There are two cases: no enqueue operation find this empty entry or no enqueue operation updates this entry successfully. Because the `tail` variable is always lag behind of the actual tail of the queue and all enqueue operations look for an empty entry from the entry pointed by the `tail` variable. So, the first case can not happen. Assume that enqueue operations find the empty entry at the tail of the queue, but no one updates it successfully. This is against the definition of CAS: at least one of the processes will succeed. □

The same argument can be given us the following lemma.

Lemma 3 *The proposed queue algorithm guarantees that dequeue operations dequeue items from left to right consecutively.*

To proof our queue is a FIFO one and satisfies the linearizability requirement, we also need the following two lemma for our queue. The lemma below proves that our queue implementation is a FIFO one and no element enqueued gets lost.

Lemma 4 *In a complete history H, $\forall Enqueue(x), Enqueue(y) \in H$ such that $Enqueue(x) \rightarrow_h Enqueue(y)$, if $Dequeue(y) \in H$, then $Dequeue(x) \in H$ and $Dequeue(x) \rightarrow_h Dequeue(y)$.*

Proof: First we proof $Dequeue(x) \in H$. Let's assume $Dequeue(x) \notin H$. It means no process has execute `CAS` successfully. The content x is still in the array and the location of x should not be empty. Because $Enqueue(x) \rightarrow_{\mathrm{h}} Enqueue(y)$, the location of x should between the real `head` and the location of y before y is removed from the queue. As point out before, the real `head` of the queue is lag behind the *head* pointer of the queue. If a process A dequeue y, it must conclude that all locations in the array between the *head* pointer and the location of y is empty. It is contradict with our assumption.

With the same argument above, we can proof that $Dequeue(x) \rightarrow_{\mathrm{h}} Dequeue(y)$. □

The lemma below proves that dequeue operations dequeue items that have really been enqueued.

Lemma 5 *In a complete history,* $\forall Dequeue(x) \in H$, $\exists Enqueue(x) \in H$ *and* $Enqueue(x) \rightarrow_h Dequeue(x)$.

Proof: The location in the array only changed by `CAS` with *enqueue* and *dequeue* operations. If x is dequeue, it means the location contains x. As the array is initialized to be `NULL`. There must exists a `CAS` operation which change the content of the location from `NULL` to x before the *dequeue* operation. □

The above lemmas give us two properties of our queue implementation. According to [4], we can conclude the following theorem.

Theorem 1 *Our queue algorithm is a linearizable FIFO concurrent queue.*

3.2 Description of the algorithm

Figure 6 and figure 7 present commented pseudo-code for the new non-blocking queue algorithm. The queue is implement as a circular array. If the value of one of the cells of the array is $NULL(0)$ or $NULL(1)$, the cell is empty. All entries of the array are initialised to be empty. In our algorithm, head and tail are used to indicate that the actual head is in the scope of $[head, ..., head + m]$ and $[tail, ..., tail + m]$. In this way, we can reduce the number of atomic operations $(m - 1)/m$ times and reduce the average number of Compare-and-Swap for one queue operation from 2 to $1 + 1/m$.

There is a tradeoff when choosing m. Large $m's$ will cause low bus contention but require more time to find the actual position of head or tail. Now most shared memory system use cache coherence protocol. Usually, one cache line contains 64 or more bytes; that means for a 64 bit machine, each line will contain 8 words (machine word). When the processor fetches one word from the memory to cache, it also fetch other 7 words or more together into the cache. This feature make the checking for the real location of header or tail to operate in cache most times if $m \ll 8$. As we reduce the bus contention in our algorithm, the performance for queue operation become more scalable than other queue algorithms.

```
Dequeue(t: pointer to Queue, oldnode:
        pointer of pointer to data type)
loop
  th = t->head;      #read the head
  #here is the one we want to dequeue
  temp = (th + 1) & MAXNUM;
  tt = t->nodes[temp];
  # find the actual head after this loop
  while (tt == NULL(0) OR tt == NULL(1)) do
      #check the head's consistency
      if (th != t->head) break;
      #two consecutive NULL means EMPTY return
      if (temp == t->tail) return 1;
      temp = (temp + 1) & MAXNUM;  #next cell
      tt = t->nodes[temp];
  end while
  #check the head's consistenicy
  if (th != t->head) continue;
  #check whether the Queue is empty
  if (temp == t->tail)
    #help the enqueue to update end
    cas(&t->tail,temp,(temp+1) & MAXNUM);
    continue;      #try dequeue again
  end if
  if (tt & 0x80000000)
     tnull = NULL(1);
  else
     tnull = NULL(0);
  #check the head's consistency
  if (th != t->head) continue;
  #Get the actual head, null value means empty
  if (cas(&(t->nodes[temp]),tt,tnull))
     if ((temp%2)==0) cas(&(t->head),th,temp);
     *oldnode = tt & 0x7fffffff;       #return the value
     return 0;
  end if
endloop
```

Figure 7: The dequeue operation

4 Performance Evaluation

We implemented our algorithm and conducted our experiments on a SUN Enterprise 10000 with 64 250MHz UltraSPARC processors and on a SGI Origin 2000 with 64 195MHz MIPS R10000 processors. The SUN multiprocessor is a symmetric multiprocessor while the SGI multiprocessor is a ccNUMA one. To ensure accuracy of the results, we had exclusive access to the multiprocessors while conducting the experiments. For the tests we compared the performance of our algorithm (new) with the

performance of the algorithm by Michael and Scott (MS) [10] because their algorithm appears to be the best non-blocking FIFO queue algorithm. In our experiments, we also included a solution based on locks (ordinary lock) to demonstrate the superiority of non-blocking solutions over blocking ones.

4.1 Experiments on SUN Enterprise 10000

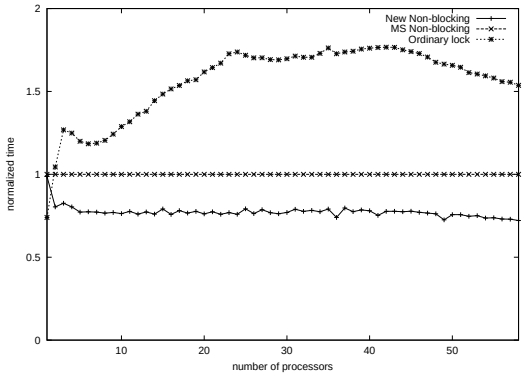
We have conducted 3 experiments on the SUN multiprocessor, in all of them we had exclusive use. In the first experiment we measured the average time taken by all processors to perform one million pairs of enqueue/dequeue operations. In this experiment (Figure 8a) a process enqueues an item and then dequeues an item and then it repeats. In the second experiment (Figure 8b) processes stay idle for some random time between each two consecutive queue operations. In the third experiment we used parallel quick-sort, that uses a queue data structure, to evaluate the performance of the three queue implementations. Parallel quick-sort had to sort 10 million randomly generated keys. The results of this experiment are shown in Figure 8c. The horizontal axis in the figures represent the number of processors, while the vertical one represents execution time normalised to that of Michael and Scott algorithm.

The first two experiments (on 58 processors), show that the new algorithm outperforms the MS algorithm by more than 30% and the spin-lock algorithm by more than 50%. The third experiment shows that the new queue implementation offers 40% better response time to the sorting algorithm.

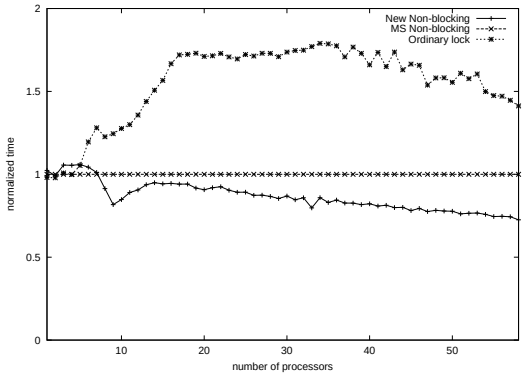
4.2 Experiments on the SGI multiprocessor

On the SGI machine, the first three experiments were basically the same experiments that we performed on the SUN multiprocessor. The only difference is that on the SGI machine we could select to use the system as a dedicated system (multiprogramming level one) or as a multiprogrammed system with two and three processes per processor (multiprogramming level two and three respectively). For the SUN multiprocessor this was not possible. Figures 9, 10 and 11a show graphically the performance results. What is very interesting is that our algorithm gives almost the same performance improvements on both machines.

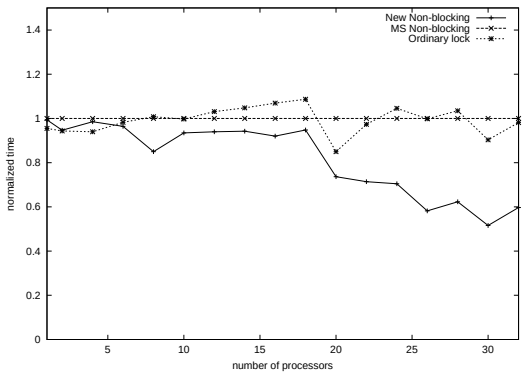
On the SGI multiprocessor, it was possible to use the radiosity from SPLASH-2 shared-address-space parallel applications [13]. Figure 11b shows the performance improvement compared with the original SPLASH-2 implementation. The vertical axis represents execution time normalised to that of the SPLASH-2 implementation.



(a) on the SUN Starfire with full contention

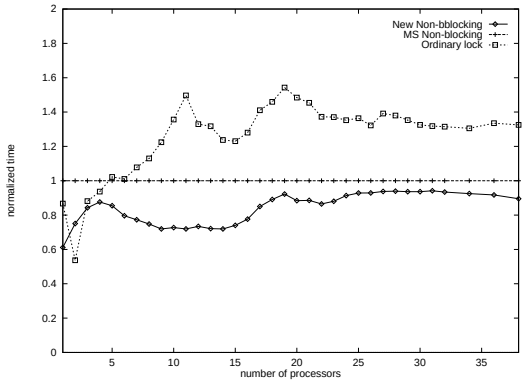


(b) on the SUN Starfire with random waiting contention

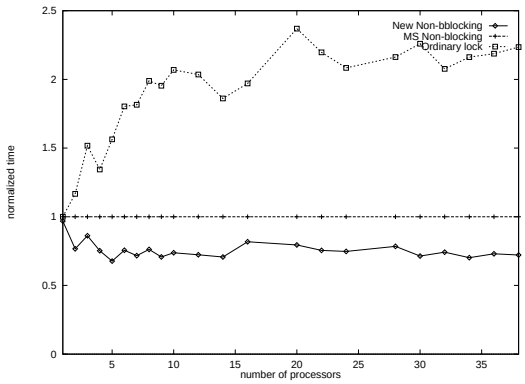


(c) Quick-sort on SUN Starfire

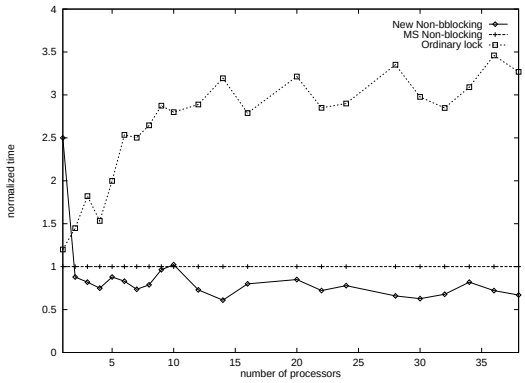
Figure 8: Results on the SUN multiprocessor



(a) Level one

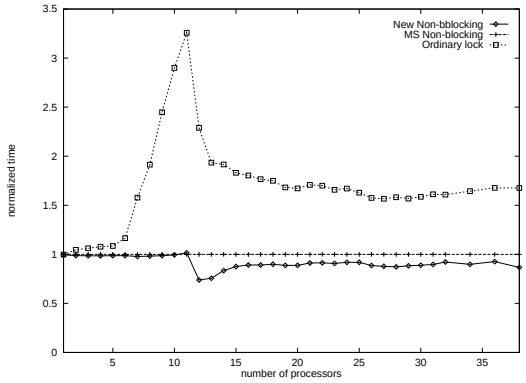


(b) Level 2

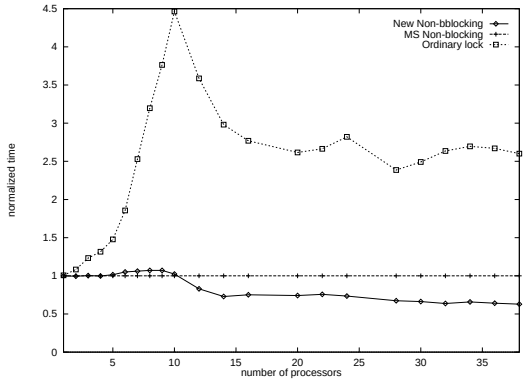


(c) Level 3

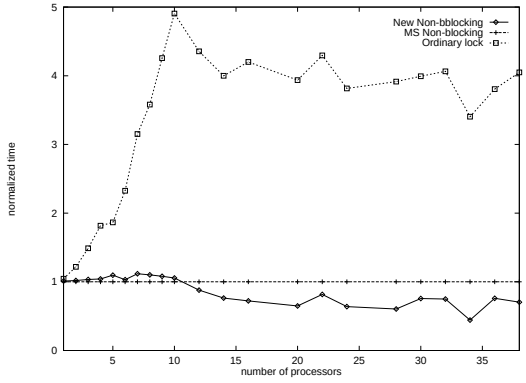
Figure 9: Results on the SGI multiprocessor with different multiprogramming levels under full contention



(a) Level 1

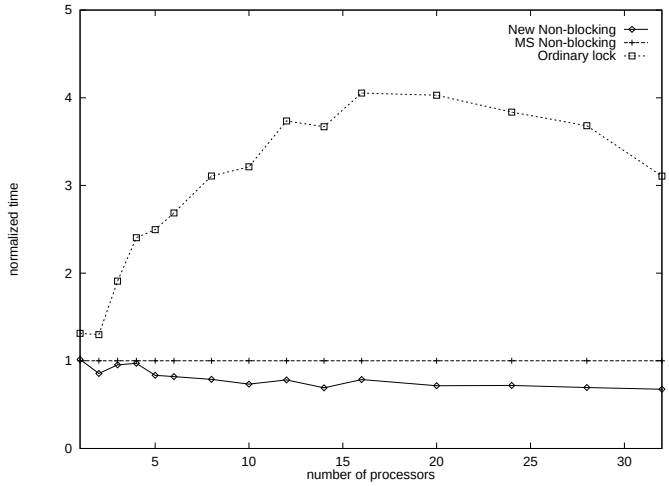


(b) Level 2

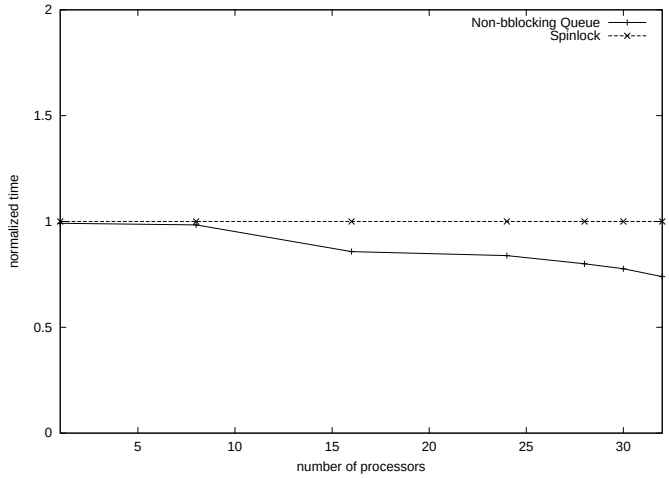


(c) Level 3

Figure 10: Results on the SGI multiprocessor with different multiprogramming levels under random waiting contention



(a) Quick-sort



(b) Radiosity

Figure 11: Applications on SGI

5 Conclusions

In this paper we presented a new bounded non-blocking concurrent FIFO queue algorithm for shared memory multiprocessor systems. The algorithm is simple and introduces two new simple algorithmic mechanisms that can be of general use in the design of efficient non-blocking algorithms. The experiments clearly indicate that our algorithm considerably outperforms the best of the known alternatives in both UMA and ccNUMA machines with respect to both dedicated and multiprogramming workloads. The experimental results also give a better insight into the performance

and scalability of non-blocking algorithms in both UMA and ccNUMA large scale multiprocessors with respect to dedicated and multiprogramming workloads, and they confirm that non-blocking algorithms can perform better than blocking on both UMA and ccNUMA large scale multiprocessors.

Acknowledgements

We would like to thank David Rutter for his great help during the writing phase of this paper. We are grateful to Carl Hallen, Andy Polyakov and Paul Waserbrot, they made the impossible possible and at the end we could have exclusive access to our heavily (thanks to our physics department) loaded parallel machines.

References

[1] A. Charlesworth. Starfire — extending the SMP envelope. *IEEE Micro*, 18(1):39–49, 1998.

[2] D. Cortesi. Origin 2000 and onyx2 performance tuning and optimization guide. http://techpubs.sgi.com/library/, SGI Inc., 1998.

[3] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, Jan. 1991.

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

[5] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles Operating Systems Review (13th SOSP 1991)*, pages 41–55, Pacific Grove, CA, Oct. 1991.

[6] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, Apr. 1983.

[7] J. Laudon and D. Lenoski. The SGI origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 241–251, New YOrk, June 2–4 1997. ACM Press.

[8] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, 1991.

[9] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, Feb. 1991.

[10] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 25 May 1998.

[11] S. Prakash, Y. Lee, and T. Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43:548–559, May 1994.

[12] J. D. Valois. *Lock-Free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, Department of Computer Science, 1995.

[13] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characteriation and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, New York, June 22–24 1995. ACM Press.

[14] J. Zahorjan, E. D. Lazowska, and D. L. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel processors. *IEEE Transactions on Parallel and Distributed Systems*, PDS-2(2):180–198, Apr. 1991.

Chapter 3

Integrating Non-blocking Synchronisation in Parallel Applications: Performance Advantages and Methothologies

This paper is the full version of the paper appeared in the *Proceedings of the 3rd ACM Workshop on Software and Performance (WOSP'02)*.

Integrating Non-blocking Synchronisation in Parallel Applications: Performance Advantages and Methodologies *

Philippas Tsigas and Yi Zhang

Department of Computing Science,
Chalmers University of Technology,
SE-412 60, Gothenburg, Sweden

Abstract

In this paper we investigate how performance and speedup of applications would be affected by using non-blocking rather than blocking synchronisation in parallel systems. The results obtained show that for many applications, non-blocking synchronisation leads to significant speedups for a fairly large number of processors, while it never slows the applications down. As part of this investigation this paper also provides a set of efficient and simple translations that show how typical blocking operations found in parallel applications, such as simple locks, queues and lock trees can be translated into non-blocking equivalents that use hardware primitives common in modern multiprocessor systems. With these translations, this paper clearly demonstrates that it is easy for the application designer/programmer to replace the blocking operations commonly found on with non-blocking equivalents ones. For the empirical results a set of representative applications running on a large-scale ccNUMA machine were used.

1 Introduction

Parallel programs running on shared memory multiprocessors coordinate via shared data objects. To ensure the consistency of the shared data objects, programs typically

* This paper is based on preliminary work that appeared: i) in the proceedings of the ACM SIGMETRICS 2001/Performance 2001 Joint International Conference on Measurement and Modeling of Computer Systems, Cambridge, Massachusetts, USA, June 2001 and ii) in the proceedings of the ACM SIGSOFT, SIGMETRICS Workshop on Software and Performance, Rome, Italy, July 2002.

rely on some forms of software synchronisations. Typical software synchronisation mechanisms are based on blocking that unfortunately results in poor performance because it produces high levels of memory and interconnection network contention and, more significantly, because it causes convoy effects: if one process holding a lock is preempted, other processes on different processors waiting for the lock will not be able to proceed. Researchers have introduced non-blocking synchronisation to address the above problems. However, its performance implications on modern systems or on real applications are not well understood. In this paper we study the impact of non-blocking synchronisation on representative applications running on a large-scale ccNUMA machine: a 64-processor SGI Origin 2000. Our results show that non-blocking synchronisation leads to significant speedups for a fairly large number of processors, while, more interestingly, it never slows the applications down. We used several SPLASH-2 applications [24] with different communication requirements and some Spark98 kernels [17]. The SPLASH-2 suite of parallel applications has been developed to facilitate the study of centralised and distributed and shared-address-space multiprocessor systems and has been used extensively by the parallel architecture community. The set of Spark98 kernels is a collection of sparse matrix kernels for shared memory and message passing systems. Spark98 kernels have been developed to facilitate system builders with a set of example sparse matrix codes that are simple, realistic, and portable. The identification of the key synchronisation schemes that are used in multiprocessor applications and the efficient transformation to non-blocking ones with the use hardware primitives that are commonly found in multiprocessor systems are integral parts of the study presented here.

Cache-coherent non-uniform memory access (ccNUMA) shared memory multiprocessor systems have attracted considerable research and commercial interest in the last years. Unfortunately, synchronisation is still an intrusive source of bottlenecks in many parallel programs running on shared memory multiprocessors. Synchronisation in these systems is explicit via high-level synchronisation operations like locks, barriers, semaphores, etc. The systems typically provide a set of hardware primitives in order to support the software implementation of these high-level synchronisation operations. There has been a considerable debate about how much hardware support and which hardware primitives should be provided by the systems to support software synchronisation primitives that the user can build.

1.1 Blocking vs. Non-Blocking Synchronisation

Software implementations of synchronisation constructs are usually included in system libraries. Good synchronisation library design can be challenging and as it is expected many efficient implementations for for the basic synchronisation constructs (locks, barriers and semaphores) have been proposed in the literature. Many such

implementations have been designed with the aim to lower the contention when the system is in a high congestion situation. These implementations give different execution times under different contention instances. But still the time spend by the processes on the synchronisation can form a substantial part of the program execution time [7, 13, 14, 16, 27]. The reason for this is that typical synchronisation is based on blocking that introduces performance bottlenecks because of busy-waiting and convoying. Busy-waiting tends to produce a large amount of memory and interconnection network contention. The convoying effect that takes place when a process holding a lock is preempted (slows down), all other process waiting for the same lock that become unable to perform any useful work until the process that holds the locks is scheduled back. In a typical multiprocessor environment, processes run for periods of time (multiprogramming environment) or, even if the machine is used exclusively, background daemons run from time to time, processes are interrupted by page faults, I/O interrupts. These events can cause the rate at which processes make progress to vary considerably. With synchronisation that is based on blocking the parallel program as a whole slows down when one process is slowed (convoying effect). To address the problems that arise from blocking researchers have proposed non-blocking implementations of shared data objects.

Non-blocking implementation of shared data objects is a new alternative approach to the problem of designing scalable shared data objects for multiprocessor systems. Non-blocking implementations allow multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion to accomplish this. Since, in non-blocking implementations of shared data objects, one process is not allowed to block another process, non-blocking shared data objects have the following significant advantages over lock-based ones:

(1) they avoid lock convoys and contention points (locks).
(2) they provide high fault tolerance (processor failures will never corrupt shared data objects) and eliminates deadlock scenarios, where two or more tasks are waiting for locks held by the other.
(3) they do not give priority inversion scenarios.

The above features of non-blocking synchronisation makes it ideal for parallel and real-time systems.

1.2 Previous and Current Work

As it was expected, non-blocking synchronisation has attracted the attention of many researchers that developed efficient non-blocking implementations for several data objects. Some studies have focused on the developing of better software algorithms, while others have identified the properties of different atomic transaction operations

in terms of their synchronisation power [5]. Some evaluation studies have also been performed for specific data structure implementations. Most of these performance evaluations were using micro-benchmarks and were performed on small scale symmetric multiprocessors, as well as distributed memory machines [1, 6, 8, 9, 14] or simulators [8, 11]. Micro-benchmarks are useful since they enable easy isolation of performance issues, but the real goal of better synchronisation methods is to improve performance of real applications, which micro-benchmarks may not represent well. A substantial number of realistic scalable applications now exist. On the systems side, scalable, hardware coherent machines with physically distributed memory have become very popular for moderate to large scale computing. It is important to evaluate the benefits of non-blocking synchronisation in a range of interesting applications running on top of modern realizations of these systems. In [16] the authors assess the performance and scalability of several software synchronisation algorithms, as well as the interrelationship between synchronisation, multiprogramming and parallel job scheduling. In their evaluation, minor modifications are applied in the synchronisation code of small number of applications that spend a significant amount of time in synchronisation.

In the work presented here, we try to provide an indepth understanding of the performance benefits of integrating non-blocking synchronisation in general parallel applications. That is the reason that applications like volrent, from SPLASH-2, that do not spend a lot of time in communication were included. Simple methodologies that could transform all major lock-based synchronisations used in the applications had to be introduced. We tried to select representative applications well known and with characteristics that are well-understood. We selected applications from the SPLASH-2 shared-address-space parallel applications suit [24] and the Spark98 kernels [17].

More specifically, the main issues addressed in this paper include:

i) The identification of the basic locking operations that parallel programmers use in their applications.
ii) The study of the architectural support found in modern ccNUMA architectures like the SGI Origin 2000 machine that could be used to support non-blocking synchronisation mechanisms.
iii) The efficient translation of lock-based synchronisation operations found in the applications to non-blocking semantically equivalent ones.
iv) The experimental comparison of the lock-based and lock-free versions of applications selected.

The work presented here shows that although the applications used are optimised for parallel performance and usually perform synchronisation only when really needed — It is reasonable to expect versions of the same or similar applications to be produced by non-expert programmers with more synchronisation — the integration of non-

blocking synchronisation to them lead to significant speedups for a fairly large number of processors, and more surprisingly never slowed the applications down. Another integrated result that is presented in this paper is that it is easy to replace the lock-based synchronisation operations with non-blocking equivalent ones. We think that this is a strong argument for making non-blocking synchronisation a common practice. Our results can benefit parallel programmers in two ways. First, to understand the benefits of non-blocking synchronisation, and then to transform some typical lock-based synchronisation operations that are probably used in their programs to non-blocking ones by using the general translations that we provide in this paper. Although for our examination we used a set of applications on a 64 processor SGI Origin 2000 multiprocessor system, the conclusions and the methods presented in this paper have general applicability in other realizations. A preliminary and short version of this paper presented at a poster session at the ACM Conference of Sigmetrics/Performance [23].

The rest of the paper is organised as follows. Section 2 outlines the Origin 2000 architecture and its hardware support for synchronisation. Section 3 discusses the applications that we used for our evaluation. Section 4 presents the transformations that we applied in order to translate the basic blocking synchronisation operations used in these applications to non-blocking ones. In the same section we also present the experimental results. Section 5 discusses the questions addressed in this work together with the results obtained. Finally, Section 6 concludes this paper.

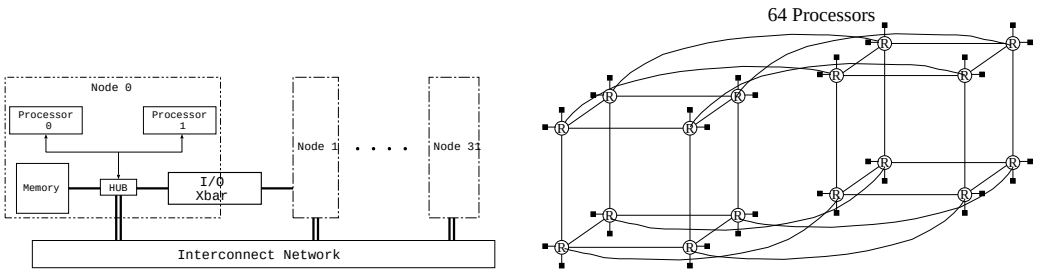


Figure 1: The SGI Origin 2000 architecture

2 Origin 2000

The SGI Origin 2000 [9] is a commercial ccNUMA machine with fast MIPS R10000 processors [26] and an aggressive scalable distributed shared memory (DSM) architecture. ccNUMA is a relatively new system topology that is the foundation for many next-generation shared memory multiprocessor systems. Based on "commodity" processing modules and a distributed, but unified, coherent memory, ccNUMA extends the power and performance of shared memory multiprocessor systems while preserv-

```
LL(p_i , O)
{
    Pset(O) := Pset(O) ∪ {p_i}
    return value(O)
}
```

```
SC( p_i, v, O )
{
  if p_i ∈ Pset(O)
     value(O):= v
     Pset(O) := ∅
     return TRUE
  else
     return FALSE
}
```

Figure 2: The Load-Linked and Store-Conditional primitives

ing the shared memory programming model. ccNUMA systems maintain a unified global coherent memory and all resources are managed by a single copy of the operating system. A hardware-based cache coherency scheme ensures that data held in memory is consistent on a system-wide basis.

2.1 The Platform

The SGI Origin 2000 [9] is a scalable shared memory multiprocessing architecture, Figure 1 describes the architecture of a 64 processor machine. It provides global address spaces not only for memory, but also for the I/O subsystem. The communication architecture is much more tightly integrated than in other recent commercial distributed shared memory (DSM) systems, with the stated goal of treating a local access as simply as an optimisation of a general DSM memory reference. The two processors within a node do not function as a snoopy share memory multiprocessor cluster, but operate separately over the single multiplexed physical bus and are governed by the same, one-level directory protocol. Less snooping keeps both absolute memory latency and the ratio of remote to local latency low [9, 10], and provides remote memory bandwidth equal to local memory bandwidth (780MB/s each) [9, 10, 12]. The two processors within a node share a hardwired coherence controller called the Hub that implements the directory based cache coherence protocol.

Two nodes (4 processors) are connected to each router, and routers are connected by CrayLinks [2]. Within a node, each processor has separate 32KB first level I and D caches, and a unified 4MB second-level cache with 2 way associativity and 128 byte block size. The machine we use has sixty-four 195MHz MIPS R10000 CPUs with 4MB L2 cache and 15.5GB main memory.

2.2 Hardware Support for Synchronisation

The SGI Origin 2000 provides two groups of transactional instructions that can be used to implement any other transactional synchronisation operations. The first one contains two simple operations, Load-Linked and Store-Conditional. The Load-Linked (LL) loads a word from the memory to a register. The matching Store-Conditional (SC) stores back possibly a new value into the memory word, unless the value at the memory word has been modified in the meantime by another process. If the word has not been modified, the store succeeds and a 1 is returned. Otherwise the, Store-Conditional fails, the memory is not modified, and a 0 is returned. The specification of this operation is shown in Figure 2.

The second hardware synchronisation mechanism is a group of `fetch_and_op` operations. The `fetch_and_op` operations are implemented at the node memory and supports at-memory atomic read-modify-write operations to special uncached memory locations. These operations are called `fetchops` and only a few atomic operations are supported on this machine. The specification of this set of operations is shown in Figure 3. The operations that are supported in Origin 2000 include `fetch_and_and`, `fetch_and_or`, `fetch_and_increment`, `fetch_and_decrement`, `fetch_and_exchange_with_zero`. The `fetch_and_and` was first introduced by the NYU Ultracomputer Project [4]. Reads and updates of `fetchop` memory blocks require a single message in the interconnection network and do not generate coherence traffic. A shortcoming of `fetchops` is the read latency experienced by a processor that spins on an uncacheable variable; spinning on `fetchop` variables may generate significant network traffic. A second drawback of `fetchops` is that they lack the synchronisation power that operations like the Compare-and-Swap, that can atomically check the and exchange the contents of a memory location, have. LL&SC or Compare-and-Swap are *universal* atomic primitives, while `fetchops` are not.

```
int fetch_and_op(int *address,int value)
{
   int temp;
   temp = *address;
   *address = op(temp,value);
   return temp;
}
```

Figure 3: The `fetch_and_op` primitive

For more information on the SGI Origin 2000 the reader is referred to [9, 19].

3 The Applications

Evaluating the impact of the synchronisation performance on applications is important for several reasons. First, micro-benchmarks can not capture every aspect of primitive performance. It is hard to predict the primitive impact on the application performance. For example, a lock or barrier that generates a lot of additional network traffic might have little impact on applications. Second, even in applications that spend significant time in synchronisation operations, the synchronisation time might be dominated by the waiting time due to load imbalance and serialisation in the application itself, which better implementations of locks and barriers may not be helpful in reducing. Third, micro-benchmarks rarely capture (generate) scenarios that occur in real applications.

We used all the applications from the SPLASH-2 [24] shared memory benchmark suite that use locks for synchronisation, and the kernels for shared memory machines from the Spark98 kernels suit [17]. We included the Spark98 kernels since they cover irregular applications based on sparse matrices. Such applications are at the core of many important scientific computations that simulate physical systems. The importance of such applications is likely to increase in the future.

Later in this section we briefly describe the applications that we have used. The actual descriptions of the applications can be found in [18, 20, 21, 24].

3.1 Problem Size

Problem size is a very important issue. Generally, the larger the problem size the lower the frequency of synchronisation relative to computation. On one hand, using large problem sizes will therefore make synchronisation operations seem less important. On the other hand, small problem sizes might result in very low speedup making them uninteresting on a machine of this scale. Because we wanted to make the evaluation on realistic problem sizes for this machines, we selected significant problem sizes that do not favour synchronisation, but still as we will show later the improvements were big in many applications. Table 1 shows the inputs that we used for each of the applications.

3.2 Application Description

Ocean simulates eddy currents in an ocean basin [25]. Both its inherent and induced (at page granularity) data referencing patterns generally involve one producer with

Application	Input
Ocean	1026
radiosity	largeroom
volrend	256x256x126
spark98	sf5.1.pack
water-spatial	1331 molecules
water-nsquared	1331 molecules

Table 1. Applications and inputs

one consumer.

Volrend renders three dimensional volume data into an image using a ray-casting method [15]. The volume data are read only. Its inherent data referencing pattern on data that are written (task queues and image data) is migratory, while its induced pattern at page granularity involves multiple producers with multiple consumers. Both the read accesses to the read only volume and the write accesses to task queues and image data are fine grained, so it suffers both fragmentation and false sharing.

Radiosity computes the equilibrium distribution of light in a scene using the iterative hierarchical diffuse radiosity method [3]. The structure of the computation and the access patterns to data objects are highly irregular.

Water-Nsquared is an improved version of the Water program in SPLASH [21]. This application evaluates forces and potentials that occur over time in a system of water molecules. A process updates a local copy of the particle accelerations as it computes them, and accumulates into the shared copy once at the end.

Water-Spatial solves the same problem as Water-Nsquared, but uses a more efficient algorithm. It imposes a uniform 3-D grid of cells on the problem domain, and uses an O(n) algorithm which is more efficient than Water-Nsquared for large numbers of molecules. The advantage of the grid of cells is that processors which own a cell need only look at neighbouring cells to find molecules that might be within the cutoff radius of molecules in the box it owns. The movement of molecules into and out of cells causes cell lists to be updated, resulting in communication.

Spark98 is a collection of sparse matrix kernels for shared memory and message passing systems. Each kernel performs a sequence of sparse matrix vector product operations using matrices that are derived from a family of three dimensional finite element earthquake applications. The multiplication of a sparse matrix by a dense vector is central to many computer applications, including scheduling applications based on linear programming and applications that simulate physical systems. These

applications are irregular applications based on sparse matrices. The running time of these applications is dominated by a sparse matrix-vector product (SMVP) operation that is repeated thousands of times, and the SMVP is the only operation besides I/O that requires the transfer of data between processors.

4 Typical Lock-Based Synchronisation Operations and Their Translations to Non-blocking Ones

In all parallel applications that we looked at, the most frequent use of a lock is to protect a single global shared variable while a process first reads the variable, then it performs the operation on the number it has just read and finally it writes the number back to the variable (the Read-Modify-Write problem). These shared variables are used in the application programs to either: i) assign consecutive values to a set of processes, or ii) to sum up values computed by processes of the system, or iii) as simple indexes of arrays.

We call this kind of locks *SimpleLocks*. It is easy to observe that such a lock can be replaced with the respective `fetch_and_op` operation to achieve the same functionality without enforcing locking, if the variable protected by the lock is of integer type. One shortcoming of the `fetch_and_op` operations is that they do not provide support for floating point numbers. In high performance scientific computing though, computations based on floating point numbers are very common. In order to overcome this shortcoming of the hardware, an efficient software implementations for the `fetch_and_op` that could supports floating point numbers was developed. For the rest of the paper we will refer to these `fetch_and_op` operation that can support floating point numbers as `double_fetch_and_op` (denoted DFAD also) operations. As a building block in our implementation we used the `load_link` and `store_conditional` primitives. The specification of the new `double_fetch_and_add` operation is given in Figure 4.

```
double double_fetch_and_add(double *address, double value)
{
    double temp;
    temp = *address;
    *address = temp + value;
    return temp;
}
```

Figure 4: The `double_fetch_and_add` primitive

Now, with the help of the FAD (`fetch_and_add`) and DFAD (`double_fetch_and_add`) operations we can remove all *SimpleLock*s in any parallel application. As it is going

to become clear in the next subsection the big majority of the locks that we found are *SimpleLock*s.

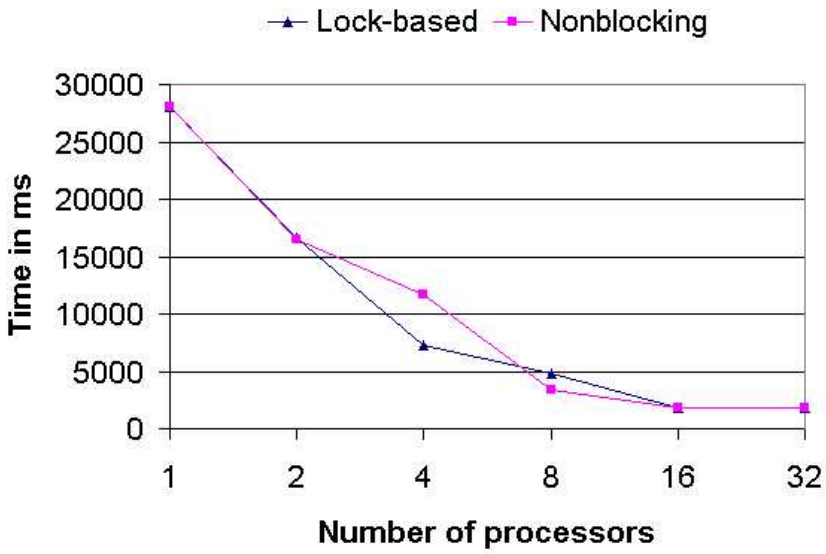


Figure 5: Performance results of Ocean

4.1 The Applications and Their Synchronisation

In this subsection, we describe the different lock-based synchronisation operations that are used in the applications that we examine, together with our transformations that transform them to non-blocking ones with the same functionality.

In the **Ocean** application 4 different locks are used:

- idlock is a *SimpleLock* that protects the global variable `index`.
- psiailock is also a *SimpleLock* that protects the global variable `psiai` that carries floating point numbers.
- psibilock is also a *SimpleLock* that protects the global variable `psibi` that carries floating point numbers.
- error_lock on the other hand is not a *SimpleLock*, and, it protects the global variable `err_multi`. The use of `err_multi` is describe below.

We replaced the first three of these locks with FAD or DFAD operations using the methods described before in this section. The fourth lock (`error_lock`) protects a global variable which is updated conditionally as follows:

```
LOCK(locks->error_lock)
if (local_err > multi->err_multi)
```

```
    {
      multi->err_multi = local_err;
    }
    UNLOCK(locks->error_lock)
```

For this lock we had to implement a non-blocking synchronisation with the same functionality to replace it, in our implementation we used the `load_link` and `store_conditional` primitives. Figure 6 describes our simple implementation.

```
rtn = TRUE;
do
{
  temp = LL(multi->err_multi);
  if (local_err > temp)
     rtn = SC(multi->err_multi, local_err);
}
while(rtn == FALSE)
```

Figure 6: Lock-free implementation of the conditional update of `error_lock`

Figure 5 shows performance results for the original version and the modified non-blocking version of the **Ocean** application. Because the **Ocean** application requires the number of processes to be power of 2, we could only do the experiments for up to 32 processors. For this particular application we do not observe any significant improvement after the modification, but, we also notice that the non-blocking synchronisation do not hamper it's performance. Ocean is a regular application with very regular communication patterns.

In the **radiosity** application, a scene is initially modelled as a number of large input polygons. Light transport interactions are computed among these polygons. and polygons are hierarchically subdivided into patches as necessary to improve accuracy. In each step, the algorithm iterates over the current interaction lists of patches, subdivides patches recursively, and modifies interaction lists as necessary. At the end of each step, the patch radiosities are combined via an up-ward pass through the quad-trees of patches to determine if the overall radiosity has converged. The main data structures represent patches, interactions, interaction lists, the quad-tree structures, and a BSP tree which facilitates efficient visibility computation between pairs of polygons. Parallelism is managed by distributed task queues, one per processor, with task stealing for load balancing.

Radiosity uses 11 different locks:

- `index_lock` is a *SimpleLock* that protects the variable `index`.
- `bsp_tree_lock` is a lock that protects the `bsp_tree` structure.
- `pbar_lock` is a lock that protects the global variable `pbar_counter`.

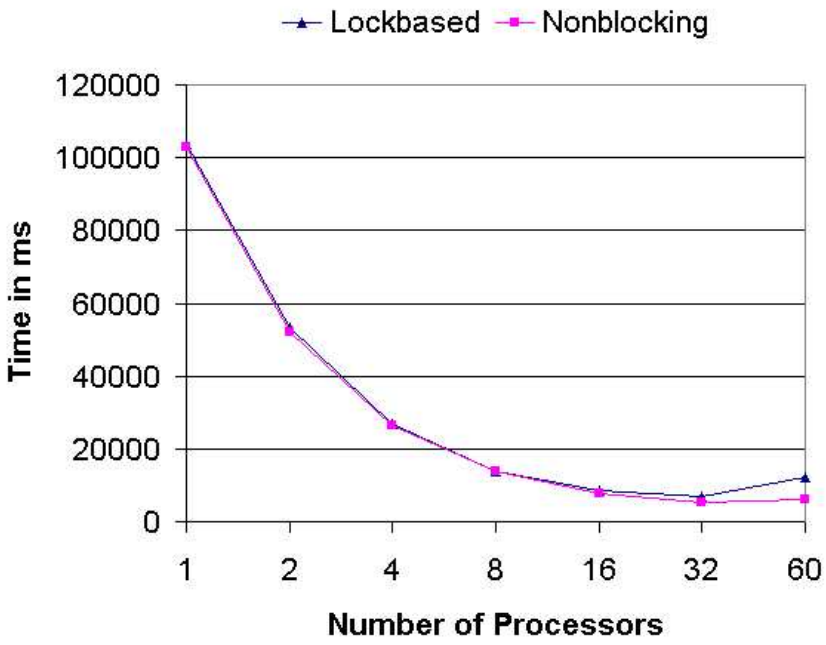


Figure 7: Performance results of **radiosity**

- `task_counter_lock` is a lock that protects the global shared variable `task_counter`.
- `free_patch_lock` is the lock that protects the global shared data object `Patch` that is implemented as a queue where free "patches" are queued.
- `free_element_lock` is the lock that protects the global shared data object `Element`. `Element` is implemented as a queue where processes queue free "elements".
- `free_interaction_lock` is the lock that protects the global shared data object `Interaction`. `Interaction` is a queue structure where "interactions" are queued.
- `free_elemvertex_lock` is the lock that protects the global shared data object `Elemvertex`. `Elemvertex` is also implemented as a queue where "free elements" are stored.
- `free_edge_lock` is the lock that protects the global shared data object `Edge`. `Edge` is also a queue structure where "free edges" are queued.
- `avg_radiosity_lock` is a lock that acts as a barrier that processes can use in order to determine when some parts of the computation should stop.
- `q_lock` protects the task queue.

The `bsp_tree` is protected by the `bsp_tree_lock` that has also a tree structure. We used the Compare-and-Swap (CAS) atomic operation to implement a non-blocking version of the `bsp_tree`. The specification of the Compare-and-Swap primitive is shown in Figure 9. For the SGI Origin 2000 system we had to emulate the Compare-and-Swap atomic primitive with the `load_linked store_conditional` instruction; this implementation is shown in Figure 10.

```
do
{
  ... ...
  traversal the tree to find the
  leaf to add the node;
  ... ...
}
while(!CAS(leaf's address, NULL, node))
```

Figure 8: Non-blocking operations on bsp_tree

In the program, nodes are only added to the bsp_tree and they are never deleted from it. Moreover, there is no operation that can change the position of a node that is already in the tree. New nodes are added as leaves. Because of these special properties of the bsp_tree, we do not face the ABA problem that most non-blocking protocols that use Compare-and-Swap have to phase. The ABA problem arises when a process p reads the value A from a shared memory location, computes a new value based on A, and using Compare-and-Swap updates the same memory location after checking that the value in this memory location is still A and mistakenly concluding that there was no operation that changed the value to this memory location in the meantime. But between the read and the Compare-and-Swap operation, other processes may have changed the context of the memory location from A to B and then back to A again. Our lock-free implementation for the bsp_tree is described in Figure 8.

```
Boolean Compare-and-Swap(WORD *mem, register WORD old, new)
{
   WORD temp;
   temp = *mem;
   if (temp == old)
   {
        *mem = new;
        return TRUE;
   }
   else
        return FALSE;
}
```

Figure 9: The Compare-and-Swap primitive

The variable pbar_counter is a counter that counts the number of working processors. It also emulates the behaviour of a barrier; when there is no processor working, the program will exit the current iteration and will check the radiosity convergence to determine whether to continue the iterations or not. We used the FAD operation to replace the locks, in this way we achieved the same functionality without using locks.

```
Boolean Compare-and-Swap(WORD *mem, register WORD old, new)
{
   WORD temp;
   do
   {
    temp = LL(mem);
    if  (temp != old)
        return FALSE;
   }while(!SC(mem,new));
   return TRUE;
}
```

Figure 10: Emulating Compare-and-Swap with `load_linked/store_conditional`

The `task_counter` is used by the processes to determine the task that enters the function `check_task_counter`. We implement this counter in a lock-free manner using the Compare-and-Swap primitive, our implementation is shown in Figure 11.

```
check_task_counter(process_id)
{
  do
  {
     tempold = global->task_counter;
     tempnew = (tempold + 1) % n_processors;
  } while(!CAS(global->task_counter, tempold, tempnew));
  flag = !tempold;
  return( flag ) ;
}
```

Figure 11: Non-blocking version of the `check_task_counter`

The remaining shared data objects that are protected by locks (`free_patch`, `free_element`, `free_interaction`, `free_elemvertex`, `free_edge`, `task_queue`) are implemented as queues. Table 2, describes some special properties of these queues.

We used the non-blocking queue implementation presented in [22], to replace the lock-based implementations for the queue based shared objects mentioned before.

Figure 7 shows the performance of our non-blocking version comparing with original one. There is no big difference between the two versions until we reach 32 processors where synchronisation becomes a significant part of the total computing time. With 32 processors, the non-blocking version is about 34% faster than the lock-based one and as the number of processors increases the improvement on the performance also increase reaching a 93% better performance when using 60 processors, the maximum number of processors that we could use exclusively for running this application. The access patterns to shared data structures in **radiosity** are highly irregular, as we

Data Object Name	Functionality
`free_patch`	no enqueue operations run in parallel
`free_element`	no enqueue operations run in parallel
`free_interaction`	enqueue and dequeue operations run in parallel
`free_elemvertex`	no enqueue operations are running in parallel
`free_edge`	no enqueue operations are running in parallel
`task_queue`	enqueue and dequeue operation are running in parallel

Table 2. Data objects in **radiosity**

mentioned in the previous section.

Volrend in contrast with radiosity does not use many locks. It uses only two *SimpleLock*s and an array lock. These locks are described below:

- `IndexLock` is a *SimpleLock* that protects the shared variable `index`.
- `CountLock` is a *SimpleLock* that protects the shared variable `Counter`.
- `QLock` is an array lock used to protect a global queue. The global queue is implemented as an array. The protection is on the index of the array. As there is only one arithmetic operation, we used a normal `fetch_and_add` to translate it into a non-blocking one.

Figure 12 shows the performance of our non-blocking version comparing with original one. The performance advantage of the non-blocking version starts to show as the number of processors becomes greater than 8. The performance of the non-blocking one is close to optimal since its speed up is very close to the theoretical limit. **Volrend's** inherent data referencing pattern on data that are written (task queues and image data) is migratory, while its induced pattern at page granularity involves multiple producers with multiple consumers.

From the **Spark98** kernel we used the shared memory applications, the *lmv* and the *rmv*. The *lmv* is a parallel shared memory program based on locks. The *rmv* is a parallel shared memory program based on a reduction of the number of locks that are used in *lmv*. Based on the naming schemes that the developers of **Spark98** have used, we named our version *nmv*. In order to create this non-blocking version we used the *lmv* version from the kernel. All locks in this program are *SimpleLock*s and they handle floating point numbers. Due to the limited time for exclusive use that we had we performed the experiments for up to 28 processors for this application. The results, graphically shown in Figure 13, clearly show the power of non-blocking synchronisation for unstructured applications like this one. The speedup of *rmv* and

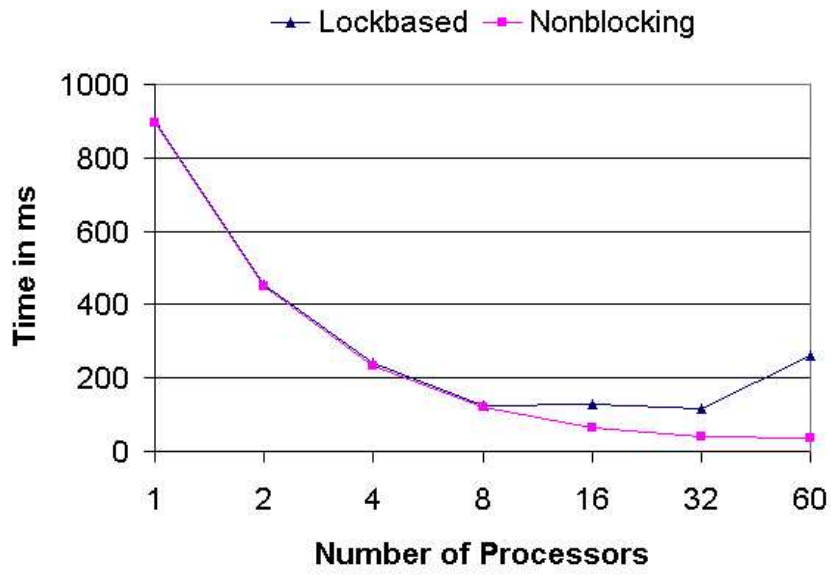


Figure 12: Performance results of VOLREND

lmv stop when we go above 16 processors while *nmv* scales uniformly. This allows us to conjecture that non-blocking will dramatically increase the performance of these applications as the number of processors increases.

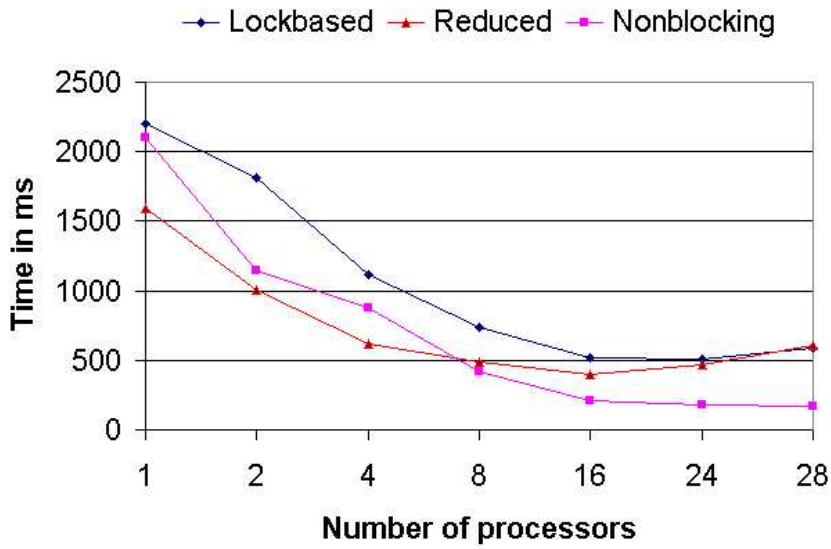


Figure 13: Performance results of Spark98

In **Water-nsquared** although 10 different locks are defined, only 7 are used. These 7 are described bellow:

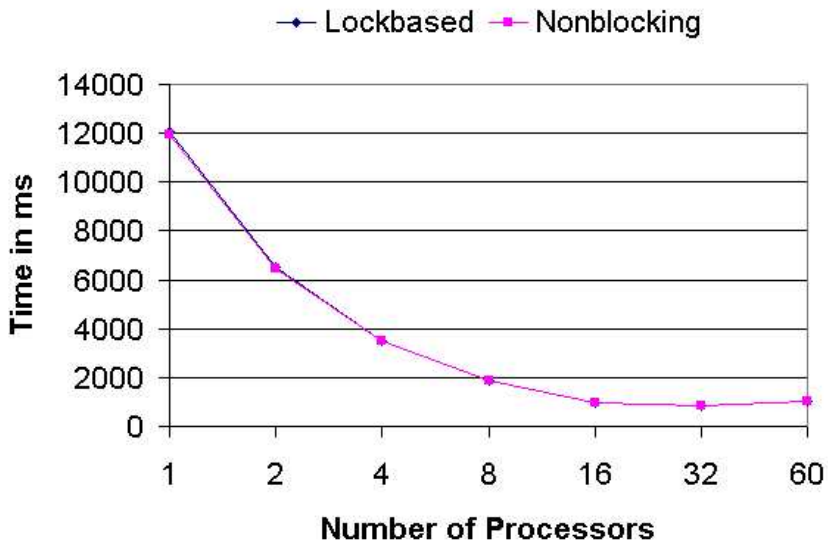


Figure 14: Performance results of WATER-NSQUARED

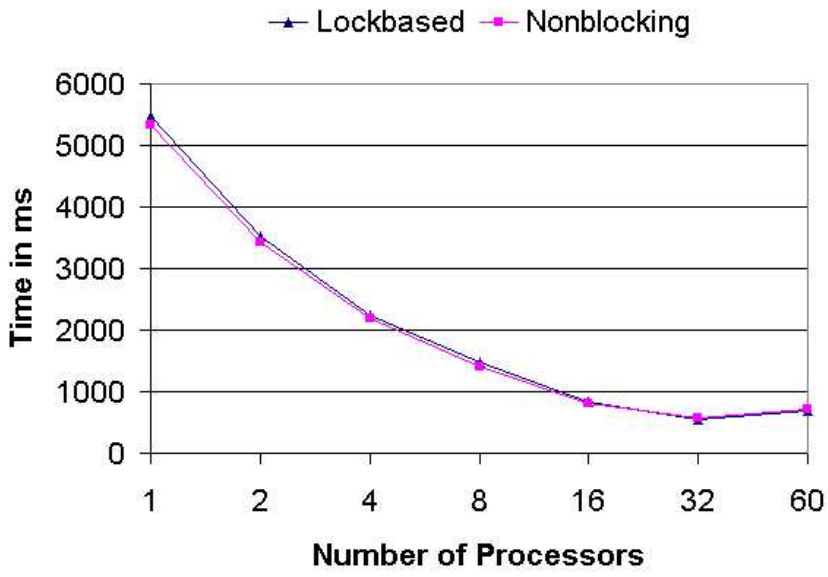


Figure 15: Performance results of WATER-SPATIAL

- `IndexLock` is a *SimpleLock* that protects the global variable `Index`
- `IntrafVirLock` is a *SimpleLock* that protects the global variable `VIR` when computing the intra-molecular force/mass acting.
- `InterfVirLock` is a *SimpleLock* that protects the variable `VIR` when computing the inter-molecular force.
- `KinetiSumLock` is a *SimpleLock* that protects the array `SUM`

- `PotengSumLock` is a *SimpleLock* that protects the variables `POTA, POTR, POTRF`.
- `MolLock`, is an array of locks, all of them are *SimpleLock*s and they are used in order to update the force on all molecular.
- `IOLock` is a special lock that is used for I/O control. We used the implementations described in the previous subsection in order to replace all *SimpleLock*s.

Water-spatial uses 7 different locks. Five of these are *SimpleLock*s, the first five *SimpleLock*s that are listed in the **Water-nsquared** above (`IndexLock, IntrafVirLock, InterfVirLock, KinetiSumLock, PotengSumLock`). We used the implementations described in the previous subsection in order to replace all *SimpleLock*s.

In **Water-nsquared** and **Water-spatial** the communication and the sharing of the data is very simple: A process updates a local copy of the particle accelerations as it computes them, and accumulates into the shared copy once at the end. This simple communication pattern does not give the opportunity to lock-free synchronisation to show it's power. On the other hand, the experiments show that lock-free synchronisation does not harm the performance of the applications. The lock-free versions of both applications perform as well as the respective lock-based ones.

Figure 16 summarises our experimental results. It graphically shows the maximum speedup of the lock-free and the respective lock-based implementation for each of our implementations.

5 Discussion

There has been much advocacy arising from the theory community for the use of non-blocking synchronization primitives, rather than blocking ones, in the design of inter-process communication mechanisms for parallel and high performance computing. This advocacy is intuitive, but has not been investigated on top of real and well-understood applications; such an investigation could also reveal the effectiveness of non-blocking synchronization on different applications. There has been a need for an investigation of how performance and speedup in parallel applications would be affected by using non-blocking rather than blocking synchronization primitives. From our interaction with practitioners we could definitely conclude that one of the main reasons why non-blocking synchronization has not become popular among them is the lack of such an investigation. One other significant reason is that many non-blocking synchronization mechanisms are quite complex. In this paper we want to address in an effective way this issue, by performing, a fair evaluation of non-blocking synchronization in the context of well-established parallel benchmark applications.

The results in this paper come to support the general belief of people working in

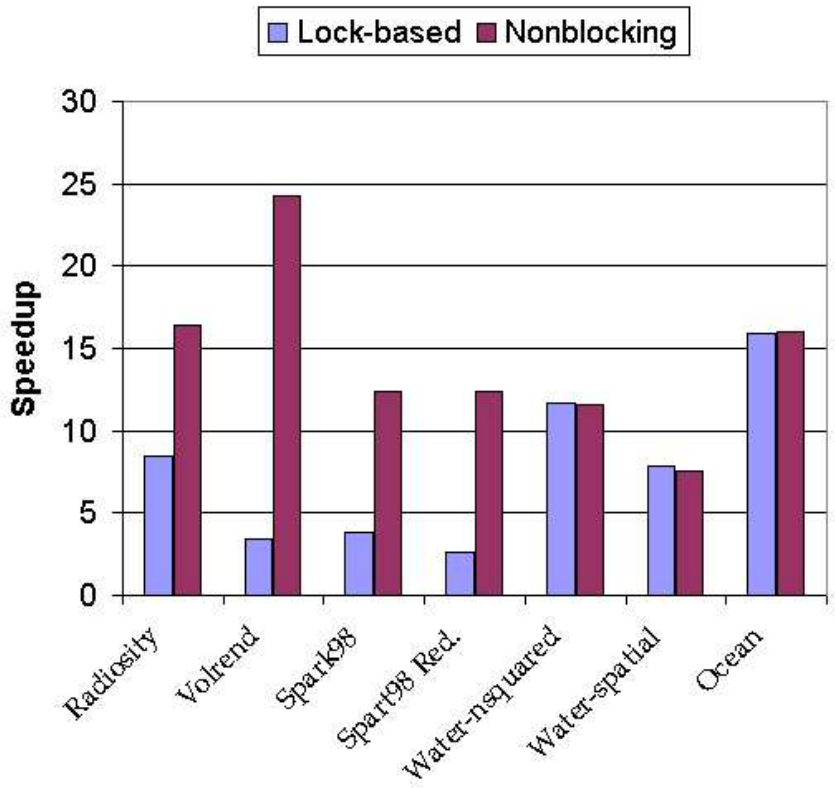


Figure 16: Speedup for the non-blocking and the original versions

the research area of non-blocking synchronization that advocated that non-blocking synchronization can lead to better performance in the context of parallel applications. They clearly show that applications that have irregular communication patterns and spend significant part of their execution time on communication can benefit a lot from non-blocking synchronization. Something that was not that clear from the beginning, was that the performance of applications with regular communication patterns that do not generate congestion are not going to be affected negatively from the introduction of non-blocking synchronization. Our intuition on this was not that clear, since, lock-based synchronization performs usually better that non-blocking synchronization when contention is very low. Since this work was aiming at clarifying the practical aspects of non-blocking synchronization, we also wanted to demonstrate that it is easy to replace the blocking operations with non-blocking equivalents and get the benefits of non-blocking synchronization, which is a strong argument for making non-blocking synchronization common practice. As part of this investigation, this paper also provides a set of efficient and simple translations that show how typical blocking operations found in parallel applications, such as simple locks, queues and lock trees can be translated into non-blocking equivalents that use hardware primitives common in modern multiprocessor systems.

We believe that our work is a first step and more experiments are needed to reveal the effectiveness of non-blocking synchronization on different applications. But we think that it is a fair evaluation of the proposed non-blocking primitives in the context of well-established parallel benchmark applications.

6 Conclusion

The main conclusions of our study are the following:

- For the fairly wide range of applications examined, non-blocking synchronisation performs as well, and often better than the respective blocking synchronisation.
- For certain applications, the use of non-blocking synchronisation yields great performance improvement. Figure 16 shows graphically the maximum speedup of the lock-free and the respective lock-based implementation for each of our implementations. With 60 processors, the non-blocking version of radiosity is about two times faster than the lock-based one; non-blocking Volrend is about 7 times faster that the lock based one.
- Irregular applications benefit the most from non-blocking synchronisation. Since the importance of such applications is likely to increase in the future, the importance of lock-free synchronisation in high-performance parallel systems is also expected to increase.
- The methods that we introduce to remove lock based synchronisations are quite simple and can be used in any parallel application.

Acknowledgements

This work was partially supported by: i) the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se) supported by the Swedish Foundation for Strategic Research and ii) the Swedish Research Council.

We are grateful to Carl Hallen, Andy Polyakov and Paul Waserbrot, they made the impossible possible and at the end we could have exclusive access to our heavily (thanks to our physics department) loaded Origin 2000. Many thanks to Marina Papatriantafilou for her always helpful comments.

References

[1] A. Eichenberger and S. Abraham, Impact of Load Imbalance on the Design of Software Barriers, in Proceedings of the 1995 International Conference on Parallel Processing, pp. 63-72, August 1995.

[2] M. Galles, Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SGI Spider Chip,in Proceedings of Hot Interconnects IV, pp. 141-146, 1996.

[3] P. Hanrahan and D. Salzman, A Rapid Hierarchical Radiosity Algorithm, in Proceeding of SIGGRAPH, pp. 197-206,1991.

[4] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, The NYU Ultracomputer - Designing a MIMD Shared-Memory Parallel Machine", *IEEE Trans. on Computers*, 32(2), p. 175, February 1983.

[5] M. Herlihy, Wait-Free Synchronization, ACM Transactions on Programming Languages and Systems, 13(1), pp. 124-149, January 1991.

[6] D. Jiang and J. Singh, A Methodology and an Evaluation of the SGI Origin2000, in Proceedings of ACM SIGMETRICS 1998, pp. 171-181.

[7] A. Karlin, K. Li, M. Manasse and S.Owicki, Empirical Studies of Competitive Spinning for a Shared-memory Multiprocessor, in Proceedings of the 13th ACM Symposium on Operating Systems Principles, pp. 41-55, October 1991.

[8] A. Kägi, D. Burger and J. Goodman, Efficient Synchronization: Let Them Eat QOLB, in Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97), pp. 170-180, ACM Press, June 2-4 1997.

[9] J. Laudon and D. Lenoski, The SGI Origin: A ccNUMA Highly Scalable Server, in Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97), Computer Architecture News, Vol. 25,2, pp. 241-251, ACM Press, June 2-4 1997.

[10] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and John Hennessy, The DASH prototype: Logic Overhead and Performance, IEEE Transactions on Parallel and Distributed Systems, 4(1), pp. 41-61, January 1993.

[11] B. Lim and A. Agarwal, Reactive Synchronization Algorithms for Multiprocessors, in Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI), pp. 25-35, October 1994.

[12] T. Lovett and R. Clapp, STiNG : A CC-NUMA Computer System for the Commercial Marketplace,in Proceedings of the 23rd Annual International Symposium on Computer Architecture, pp. 308-317, ACM Press, May 22-24 1996.

[13] J. M. Mellor-Crummey and M. L. Scott, Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM Trans. on Computer Systems, 9(1), pp. 21-65 February 1991.

[14] M. M. Michael and M. L. Scott, Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors, Journal of Parallel and Distributed Computing 51(1), pp. 1-26, 1998.

[15] J. Nieh and M. Levoy, Volume Rendering on Scalable Shared Memory MIMD Architectures, in Proceeding of the 1992 Workshop on Volume Visualization, pp 17-24, October 1992.

[16] D. S. Nikolopoulos and T. S. Papatheodorou, A Quantitative Architectural Evaluation of Synchronization Algorithms and Disciplines on ccNUMA Systems: The Case of the SGI Origin2000, in Proceedings of the 1999 Conference on Supercomputing, ACM SIGARCH, pp. 319-328, June 1999.

[17] D. R. O'Hallaron, Spark98: Sparse Matrix Kernels for Shared Memory and Message Passing Systems, Technical Report CMU-CS-97-178, October 1997.

[18] E. Rothberg, J. P. Singh and A. Gupta, Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors, in Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 14-26, IEEE Computer Society Press, May 1993.

[19] SGI, SGI TechPubs Library, http://techpubs.sgi.com/, 2000.

[20] J. P. Singh, A. Gupta and Marc Levoy, Parallel Visualization Algorithms: Performance and Architectural Implications, Computer, 27(7), pp. 45-55, July 1994.

[21] J. P. Singh, W. D. Weber and Anoop Gupta, SPLASH: Stanford Parallel Applications for Shared-Memory, Computer Architecture News, 20(1), pp. 2-12, March 1992.

[22] P. Tsigas and Y. Zhang, A Simple, Fast and Scalable Non-Blocking Concurrent FIFO queue for Shared Memory Multiprocessor Systems, in Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures, pp. 134-143, July 2001.

[23] P. Tsigas and Y. Zhang, Evaluating The Performance of Non-Blocking Synchronisation on Shared-Memory Multiprocessors (Poster Paper), in Proceedings of SIGMETRICS 2001, June 2001.

[24] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, The SPLASH-2 Programs: Characterization and Methodological Considerations, in Proceedings of the 22nd International Symposium on Computer Architectures, pp. 24-36, June 1995.

[25] S. C. Woo, J. P. Singh and J. L. Hennessy, The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors, in Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 219-229, October 4-7, 1994.

[26] K. Yeager, The MIPS R10000 superscalar microprocessor, IEEE Micro, 16(2), pp. 28-40, April 1996.

[27] J. Zahorjan, E. D. Lazowska and D. L. Eager, The effect of scheduling discipline on spin overhead in shared memory parallel systems, IEEE Transactions on Parallel and Distributed Systems, 2(2), pp. 180-198, April 1991.

Chapter 4

A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000

This paper is an extended version of the paper appeared in the *Proceedings of the 11th Euromicro Conference on Parallel Distributed and Network based Processing.*

A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000 ⋆

Philippas Tsigas and Yi Zhang

Department of Computing Science,
Chalmers University of Technology,
SE-412 60, Gothenburg, Sweden

Abstract

We have implemented sample sort and a parallel version of Quicksort on a cache-coherent shared address space multiprocessor: the SUN ENTERPRISE 10000. Our computational experiments show that parallel Quicksort outperforms sample sort. Sample sort has been long thought to be the best, general parallel sorting algorithms, especially for larger data sets.

On 32 processors of the ENTERPRISE 10000 the speedup of parallel Quicksort is more than six units higher than the speedup of sample sort, resulting in execution times that were more than 50% faster than sample sort. On one processor parallel quicksort achieved 15% percent faster execution times than sample sorting. Moreover, because of its low memory requirements, parallel Quicksort could sort data sets twice the size that sample sort could under the same system memory restrictions.

The parallel Quicksort algorithm that we implemented is a simple, fine-grain extension of Quicksort. Although fine-grain parallelism has been thought to be inefficient for computations like sorting due to the synchronization overheads, we show as part of this work that efficiency can be achieved by incorporating non-blocking techniques for sharing data and computation tasks in the design and implementation of the algorithm. Non-blocking synchronization has increased concurrency between communication and computation and gives good execution behavior on cache-coherent shared memory multiprocessor systems. Cache-coherent shared memory multiprocessors offer fruitful ground for algorithmic or programming techniques that were considered impractical before, in the context of high-performance programming, to develop and change a little the way we think about high-performance programming.

⋆ This work is partially supported by: i) the national Swedish Real-Time Systems research

1 Introduction

Sorting is an important kernel for sequential and multiprocessing computing and a core part of database systems. Donald Knuth in [10] reports that *"computer manufacturers of the 1960s estimated that more than 25 percent of the running time on their computers was spend on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time."* As it was expected, sorting is one of the most heavily studied problems in computer science. Parallel algorithms for sorting have been studied for long, with many major advances in the area coming from as early as the sixties [10]. Accordingly, a vast number of research articles dealing with parallel sorting have been published; the number is too large to allow mentioning them all, so we will restrict discussion to those that are directly related to our work. Considerable effort has been made by the theoretical community in the design of parallel algorithms with excellent and occasionally optimal asymptotic efficiency. However there has only been limited success in obtaining efficient implementations on actual parallel machines [8, 19]. Similar research effort has also been made with the practical aspects in mind, for a list of work in this area please see [3] and [14]. The latter work has given many exciting results due to interaction between the algorithms research area and the computer architectures research area. Most of the work on high-performance sorting is based on message-passing machines, vector supercomputers and clusters.

Among all the innovative architectures for multiprocessor systems that have been proposed the last forty years, a new tightly-coupled multiprocessor architecture is gaining a central place in high performance computing. This new type of architecture supports a shared address programming model with physically distributed memory and coherent replication (either in caches or main memory). A hardware-based cache coherency scheme ensures that data held in memory is consistent on a system-wide basis. These systems are commonly referred to as cache-coherent distributed shared memory (DSM) systems and are built for server and desktop computing. Over the last decade many such systems have been built and almost all major computer vendors develop and offer cache coherent shared memory multiprocessor systems nowadays. This class of systems differs a lot from the traditional message-passing machines, vector supercomputers and clusters on which high-performance sorting has been studied. These new systems offer very fast interprocess communication capabilities and give space to new programming and algorithmic techniques, which would have been impractical on vector supercomputers or clusters because of high communication costs. Shan and Singh in examined the performance of radix sorting and sample sorting (the two most efficient parallel sorting algorithms) in hardware cache-coherent shared ad-

initiative ARTES (www.artes.uu.se) supported by the Swedish Foundation for Strategic Research and ii) the Swedish Research Council for Engineering Sciences.

dress space multiprocessors under three major programming models.

This paper looks into the behavior of a simple, fine-grain parallel extension of Quicksort for cache-coherent shared address space multiprocessors. Quicksort has many nice properties: i) it is fast and general purpose; it is widely believed that Quicksort is the fastest general-purpose sorting algorithm, on average, and for a large number of elements [1, 4, 7, 16], ii) it is in-place, iii) it exhibits good cache performance and iv) it is simple to implement. The new generation of hardware-coherent, shared address space multiprocessor systems with their already dominant position on the tightly-coupled multiprocessor systems are our target systems. The implementation of the parallel Quicksort algorithm utilizes the capabilities that these new systems have to offer and uses the following algorithmic techniques:

Cache-efficient: Each processor tries to use all keys when sequentially passing through the keys of a cached-block from the key array.

Communication Overlapping Fine-grain Parallelism: It is a fine-grain parallel algorithm. Although fine-grain parallelism has been thought to be inefficient for computations like sorting due to the synchronization overheads, we achieved efficiency by incorporating non-blocking techniques for sharing data and computation tasks. No mutual locks or semaphores are used in our implementation.

Parallel Partition of Data: A parallel technique for partitioning the data similar to the one presented in [5] is used. We rediscovered this technique when parallelizing Quicksort.

We implemented the algorithm on a SUN ENTERPRISE 10000, a leading example of the tightly-coupled, hardware-coherent architecture and we compared it with sample sort, which has been previously shown to outperform other comparison based, general sorting algorithms, especially for larger data sets [1, 4, 7, 16]. On 32 processors we achieved a speedup that was more than 6 units higher than the speedup of sample sort. This speedup resulted in an execution time that was over 50% faster than sample sort. On one processor of the ENTERPRISE 10000 parallel Quicksort gave 15% percent faster execution times than the sample sort on many large sorting instances. Moreover, parallel Quicksort could sort data sets double the size that sample sort could because of the its low memory requirements. One one processor parallel Quicksort behaves as it sequential parent. The asymptotic number of all comparisons and computation and memory steps used by the algorithm is the same as in quicksort: $O(N \lg(N))$ on average and $O(N^2)$ for the worst case. When $B \ll N$, the average time complexity for the parallel algorithm is $O(\frac{N \lg(N)}{P})$ and the worst case time complexity is $O(\frac{N^2}{P})$. $O(N + P)$ is the asymptotic space complexity.

The remainder of the paper is organized as follows: In Section 2 the algorithm and its analysis are presented. We describe the experimental evaluation in Section 3. The paper concludes with Section 4.

2 The Algorithm

Quicksort [9] is a sequential sorting algorithm that is widely believed to be the fastest comparison-based, sequential sorting algorithm (on average, and for a large input sets) [2, 11, 18]. It is a recursive algorithm that uses the "Divide and Conquer" method to sort all keys. The standard Quicksort first picks a key from the list, the *pivot*, and finds its position in the list where the key should be placed. This is done by "walking" through the array of keys from one side to the other. When doing this, all other keys are swapped into two parts in the memory: i) the keys less than or equal to the *pivot* are placed to "the low side" of the *pivot* and ii) the keys larger than or equal to the *pivot* are placed to "the high side" of the *pivot*. Then the same program is recursively applied on these two parts.

2.1 Description

Assume that we have an array with N keys, indexed from 0 to $N-1$, to be sorted on a cache-coherent shared memory multiprocessor with P asynchronous processors. Each processor is assigned a unique index, *pid*, from 0 to $P-1$.

The parallel Quicksort algorithm presented here is a simple parallelization of Quicksort. It is a 3+1-phase algorithm. The first three phases constitute the divide phase and are recursively executed. The last phase is a sequential sorting algorithm that processors execute in parallel, during this phase a helping scheme is used. The four phases are: i) the Parallel Partition of the Data phase, ii) the Sequential Partition of the Data phase, iii) the Process Partition phase and iv) the Sequential Sorting in Parallel with Helping phase. The detailed explanation of the four phases is given below.

Phase One: Parallel Partition of the Data The algorithm sees the array of the data as a set of consecutive *blocks* of size B. B depends on the size of the system's first-level cache, and is selected so that two *blocks* of length B can fit in cache at the same time. In our system where the size of the first-level cache is 16KB, we selected $B = 2048$ so as to be able to fit two *blocks* of data in the cache at the same time. In order to simplify the description and without loss of generality let us consider first the case where all keys can be divided into *blocks* exactly, i.e. $N \bmod B = 0$. Later on, we will show how to extend this phase for the case where $N \bmod B \neq 0$. The whole array can be viewed as a line of $\frac{N}{B}$ data *blocks*; processors can only choose *blocks* to work on, from the two ends of the line.

The first phase starts with the processor P_0, the one with the smallest *pid*, picking a *pivot*. After that, each processor in parallel picks the *block* that it finds at the very end of the left side of the line $leftblock$ and then the *block* that it finds at the

```
SIDE neutralize (Data *leftblock, Data *rightblock, Data pivot)
{
    int i, j;
    do{
        for ( i=0; i<BlockSize; i++ )
            if (leftblock[i] > pivot)
                break;
        for( j=0; j<BlockSize; j++)
            if (rightblock[j] < pivot)
                break;
        if ((i== BlockSize) || (j == BlockSize))
            break;
        SWAP( leftblock[i], rightblock[j]);
        i++; j++;
    } while ( i < BlockSize && j < BlockSize )

    if (i == BlockSize && j == BlockSize)
        return BOTH;
    if (i == BlockSize)
        return LEFT;
    return RIGHT;
}
```

Figure 1: The *neutralize* function

very end of the right side of the line *rightblock* and uses these two *blocks* together with the *pivot* as an input to a function that is called the *neutralize* function. This function is described in pseudo-code in Figure 1. The function takes as input two blocks, $leftblock$ and *rightblock*, and the *pivot* and swaps the keys in $leftblock$ which are larger than the *pivot* with keys in *rightblock* that are smaller than the *pivot* in an increasing order, as long as this can be done. A call of the *neutralize* function will result into one of the following results: either i) all keys in $leftblock$ are going to be less than or equal to the *pivot*, in this case we say that $leftblock$ has been *neutralized* or ii) all keys in *rightblock* are going to be larger than or equal to the *pivot* and then we say that *rightblock* has been neutralized, or iii) it can also happen that both $leftblock$ and *rightblock* have been neutralized at the same time.

Each processor will then try to get a fresh *block* from the left side of the array if its $leftblock$ was *neutralized* before, or from the right side if its *rightblock* was *neutralized* before and it will then *neutralize* this *block* with the still charged *block* that it has on hand. If both *blocks* were *neutralized*, the processor gets two fresh *blocks* from both ends. Processes continue the above steps until all *blocks* are exhausted. At this moment, each processor has at most one *block* unfinished in hand and puts it on the *remainingBlocks* shared array that consequently can collect at most P blocks, and exits the *parallel partition* phase. The parallel partition phase

is described in pseudo-code in Figure 2. Processors report the number of keys contained on $leftblock$s that have been neutralized by summing the numbers into LN (Left-Neutralized). The number of keys on $rightblock$s that have been neutralized are counted into RN (Right-Neutralized).

For the general case where $N \bmod B = M \neq 0$ we can modify the end condition for the parallel phase so that a processor exits when it finds that the remaining keys are not enough to form a *block*. In this case there are going to be at most P *blocks* plus M keys left. To process the remaining keys, the processor P_0, with the smallest pid, will run the *sequential partition* phase.

```
if (pid == smallestpid)
    pivot = PivotChoose();
barrier(P);
leftblock = Get A Block From LEFT End;
rightblock = Get A Block From RIGHT End;
leftcounter = 0;
rightcounter = 0;
do{
    side = neutralize(leftblock, rightblock, pivot);
    if ((side == LEFT) || (side == BOTH))
    {
     leftblock = Get A Block From LEFT End;
     leftcounter ++;
    }
    if ((side == right) || (side == BOTH))
    {
     rightblock = Get A Block From RIGHT End;
     rightcounter ++;
    }
} while((leftblock != EMPTY) && (rightblock != EMPTY));
if (leftblock != EMPTY)
    remainingBlocks[pid] = leftblock;
else
    remainingBlocks[pid] = rightblock;
LN = LN + (leftcounter * B);
RN = RN + (rightcounter * B);
```

Figure 2: The procedure that implements the parallel partition phase.

Phase Two: Sequential Partition of the Data The purpose of **sequential partition** phase is to finish what the parallel partition has started: the placement of keys to the "correct side" of the *pivot*. When the parallel partition finishes all *neutralized blocks* that are between $[0, LN-1)$ and $[RN, N-1)$ are correctly placed with respect to the *pivot*. After the Parallel Partition phase, the remaining P *blocks* that can appear in any position on the array as shown in figure 4a need to be correctly placed and the neutralized *blocks* that are placed in $[LN-1, RN]$ have to

```
sort(remainingBlocks, ascend order);
/* p is the number of remain blocks; p ≤ P */
left = 0; right = p - 1;
/* Treat the remainingBlocks as an array and do
Sequential block Partition */
while(left<right)
{
    /*Neutralize the most left block and the most right block */
    side = neutralize( remainingBlocks[left],
                       remainingBlocks[right], pivot);
    if ((side == LEFT) || (side == BOTH))
    {
        if (remainingBlocks is between [0, LN − 1])
        {
        /* update LN and remove the block from remainingBlocks
           only if it is between [0, LN − 1] */
            LN +=B;
            remainingBlocks[left] = EMPTY;
        }
        left ++;
    }
    if ((side == right) || (side == BOTH))
    {
        if (remainingBlocks is between [N − RN, N − 1])
        {
        /* update RN and remove the block from remainingBlocks
           only if it is between [N − RN, N − 1] */
            RN +=B;
            remainingBlocks[right] = EMPTY;
        }
        right --;
    }
}
/* For those which still remain in the remainingBlocks array,
   we will swap them with blocks between [LN, N − RN) */
for ( i=0; i<p; i++)
{
    if (remainingBlocks[i] is not EMPTY)
    {
        Swap remainingBlocks[i] with neutralized
             blocks between [LN, N − RN);
    }
}
```

Figure 3: The procedure that implements the sequential partition phase.

be swapped in a correct position. During this phase processor P_0 first sorts the P *blocks* using the start indices of these *blocks*. It then uses this order to pick a *block* from the left ($leftblock$) and a *block* from right ($rightblock$) and give them as input to the *neutralize* function together with the previously selected *pivot*. It does this until all remaining *blocks* are exhausted. In this phase it is not always true that a neutralized *block* can always contribute to LN or RN, for example, a *block* that was picked out from the right end during the parallel partition phase and is neutralized as $leftblock$ now will not contribute to LN. All *blocks* between $[0, LN-1]$ were picked during the parallel partition phase by some processors from the left end. If they can be neutralized as $leftblock$s in this sequential partition phase, they can contribute to the LN as other neutralized *blocks* $[0, LN-1]$. Similar is the case for *blocks* between $[N-RN, N-1]$. The procedure of sequential partition is described in figure 3. The first step is sorting the remaining *blocks*. Then, picking *blocks* from the two ends and running the *neutralize* function on them as shown in figure 4b. Finally, some blocks are still misplaced between $[LN, N-RN-1]$. If there are m *blocks* unfinished between $[0, LN-1]$, then there should be at least m *blocks* which are *neutralized* as $leftblock$s between $[LN, N-RN-1]$. The Sequential Partition will swap them as shown in figure 4c. The same methods will be applied to the remain *blocks* between $[N-RN, N-1]$. Now, all *blocks* between $[0, LN-1]$ contain keys less than or equal to the *pivot* and all *blocks* $[N-RN, N-1]$ contain keys larger than or equal to the *pivot*. The remaining task is to partition between LN and RN as sequential quicksort.

Next we will demonstrate the behavior of the two phases presented before by a way of an example. The example is also graphically shown in in Figure 5. For input 37 random integers are selected. Our system for this example has 3 processors and the *block* size is 4. First, we need to select a *pivot*. We use the method proposed by Sedgewick in [13] to choose the *pivot*, which is the median of the first, middle and last keys in the array (in this case the *pivot* does not have to be an input key).

$$pivot = \lfloor \frac{(min(17,7,32) + max(17,7,32))}{2} \rfloor = 19$$

After choosing the *pivot*, all processors will pick two *blocks* of keys one *block* from the left end and one *block* from the right end. One possible result is the following: i) processor 0 gets *block* L2 (the second — in input order — *block* from the left end) and R1 (the first — in input order — *block* from the right end), ii) processor 1 gets L1 and R2, ii) processor 2 gets L3 and R3. Then, every processor calls the *neutralize* function. For processor 0, after the return from the *neutralize* function, all keys in *block* L2 are less than or equal to the *pivot* and all keys in block R1 are larger than or equal to the *pivot*, i.e. L1 and R1 have been neutralized. *Block* R2 and L3 are neutralized by processors 1 and 2 respectively. To continue the algorithm processor 0 needs to get two more *blocks* one from each end of the array and processor 1 needs

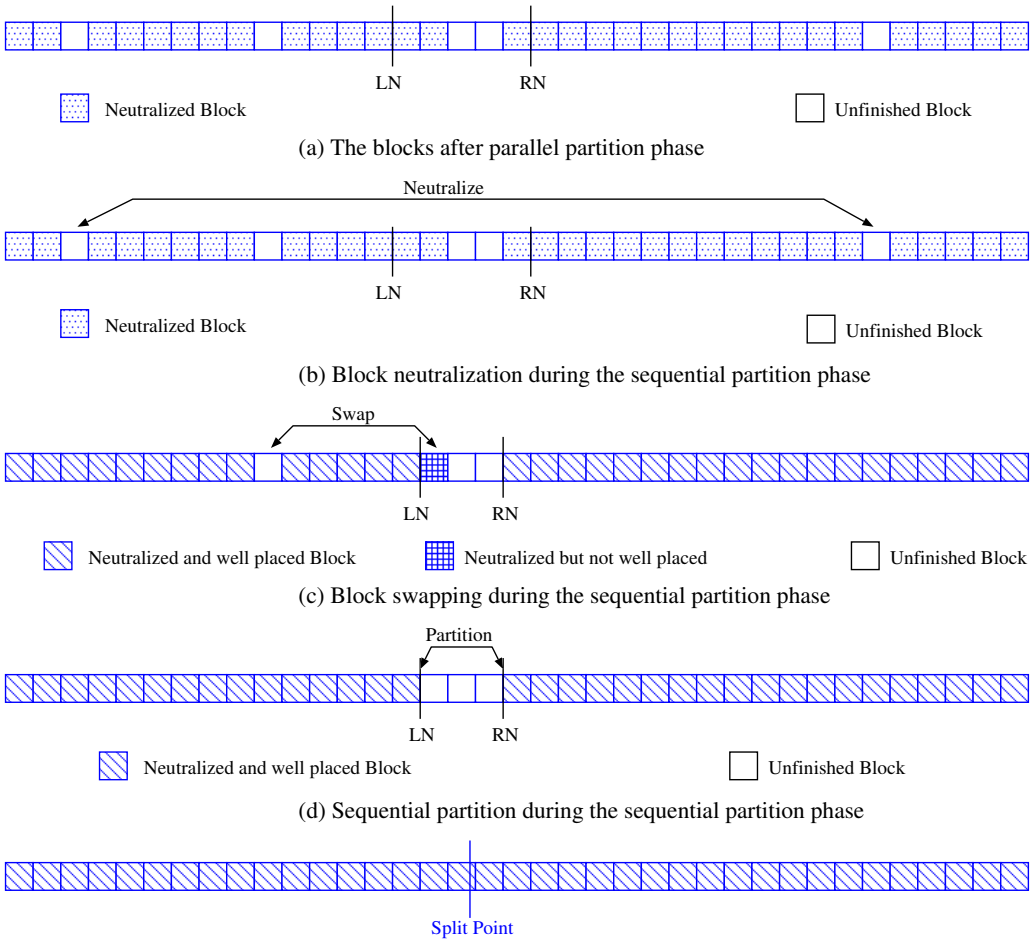


Figure 4: A graphical description of the sequential partition phase.

to get one *block* from the right end and processor 2 needs to get one from the left end. As the whole system is asynchronous, this time processor 2 gets *block* L4 and processor 0 get *block* L5 and R4. When processor 1 tries to pick a *block*, it finds out that the remaining keys are not enough to form a *block*, and consequently it exits the parallel partition phase and puts *block* L1 in the $remainingBlocks$ array. Processor 0 and processor 2 use the *neutralize* function with input L5 and R4, and L4 and R3, respectively. Processor 0 exits the parallel partition phase with L5 unfinished. Processor 2 exits with no unfinished *block*. The parallel partition phase is over with 2 *blocks* ($\leq P$) marked as unfinished and 1 key ($\leq B-1$) (key 29) left unprocessed in the middle. The algorithm will enter the sequential partition phase to process these keys, in our example there are no neutralized *blocks* in (LN,$N-RN$) to be processed, $LN = 12$ and $RN = 16$.

Processor 0, as the one with the smallest *pid*, will process first the *blocks* L1 and L5 that are in the $remainingBlocks$ array. Processor 0 calls the *neutralize* function

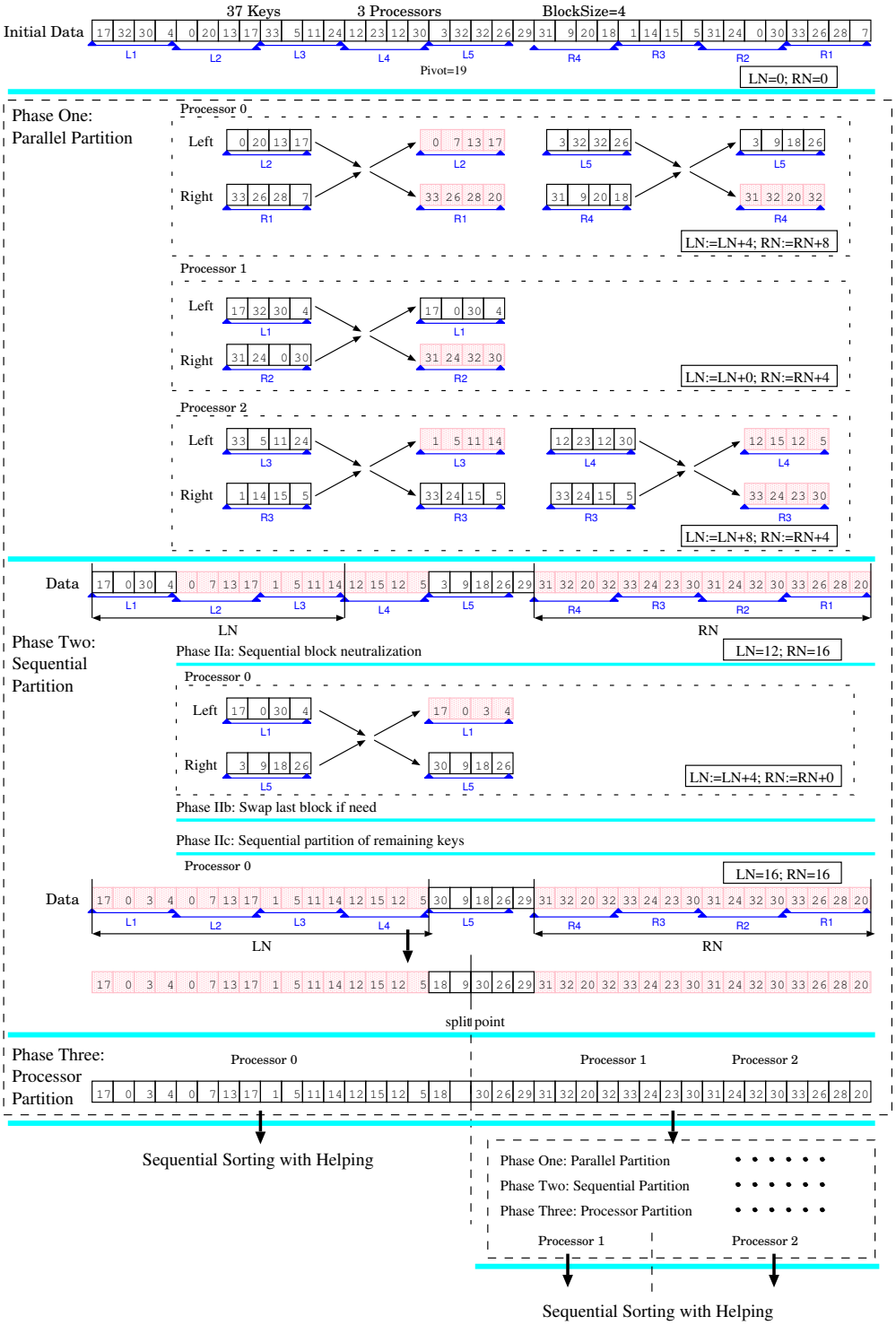


Figure 5: The algorithm by way of an example.

with L1 as the $leftblock$ and L5 as the $rightblock$ and as a result L1 is neutralized. As the start index of L1 is less than LN, LN will be increased by 4. In this example the remaining *block* L5 is between [LN,$N-RN$) and there is no need to swap it. The processor 0 splits the keys between [LN,$N-RN$)=[16, 21)=[16, 20] with the *pivot*, 19. The final split point is index 18, as shown in figure 5.

After the two partition phases, the array is split into two subarrays and all keys are placed on the "right side" of the *pivot*.

Phase Three: Process Partition During this phase, the algorithm partitions all processors into two groups. The sizes of these groups are proportional to the sizes of the respective subarrays. If the size of one group is zero, then its size is set to 1 and the other group takes the remaining processors. Each processor group will take a subarray from Phase Two and apply the same parallel partition method, Phase One to Phase Three, on it recursively. When only one processor is assigned to a subarray, the processor will exit the partition phase and enter the *Sequential Sorting with Helping* phase.

Phase Four: Sequential Sorting with Helping During the sequential sorting phase, every processor uses quicksort to sort the subarray it gets from phase three and help other processors' work after it finishes its own. For the sequential quicksort we use, the optimization introduced in [11] is applied. This optimization gives good cache behavior. Every processor uses an auxiliary stack for itself to keep track of the algorithm's state, and the recursion of Quicksort turns into a loop of $PUSH$ and POP operations. Whenever a processor encounters a small subarray which can fit in cache, it will use inserting sort to sort it without $PUSH$ing it into the stack.

In our parallel sorting algorithm, we introduce the following helping scheme to achieve good load balance. The stacks of the sequential Quicksort of all processors are implemented as lock-free (non-blocking) stacks shared among all processors. All these stacks are restricted shared concurrent stacks, because only one processor performs the $PUSH$ operations. In the algorithm presented here, we used a variant of [17] that has been optimized for our restricted class of stacks. When one processor finished its job, it will start to help other processors by *popping* out unsorted subarray (one at a time) from their stacks. In this way we can achieve load balance online.

The pseudo code of the complete algorithm is shown in Figure 6 [1] .

[1] The complete version of the code is available for non-commercial use at: http://www.cs.chalmers.se/~yzhang/PQuick

```
PQuicksort(Data *array, int size, int ProcessorNumber)
{
    if ( ProcessorNumber == 1 )
    {
    /* Parallel Sorting Phase*/
        SequentialSorting(array);
        HelpOtherProcessor();
    }
    /* Parallel Partition Phase; The code is in Figure 2*/
    parallel partition;

    /* Sequential Partition Phase */
    if ( pid == smallestpid )
    {
        /* The code is in Figure 3 */
        split = sequential partition;

        /* Processor Partition Phase */
        processorsplit = Processor Partition;
    }
    barrier(ProcessorNumber);

    /* Recursive call */
    PQuicksort(&array[0], split+1, processorsplit);
    PQuicksort(&array[split+1], size-split-1,
               ProcessorNumber-processorsplit);
}
```

Figure 6: The complete algorithm

2.2 Analysis of the Algorithm

The parallel Quicksort presented before is a simple parallelization of Quicksort. The parallel algorithm follows the same divide-and-conquer steps as Quicksort and the depth of recursion is the same between the parallel quicksort and the sequential quicksort. Therefore, the amount of comparison and swap operations of the parallel algorithm is the same with the sequential one: $O(N \lg(N))$ for the average case and $O(N^2)$ for the worst case. Now, when looking into the analysis of the speedup of the algorithm we can see that parallelism is introduced in two places: i) the partition phase and ii) the sorting phase. The time cost of the partition phase is $O(N)$. The parallel algorithm finishes the partition of the whole array in two steps. First, all processors neutralize *blocks* of keys in parallel; this will take $O(\frac{N}{P})$ time. Then one processor will process the unfinished P *blocks* and M keys; this will take $O(P * B + M)$ time. As $M \leq B$, the whole time complexity for the partition phase is $O(\frac{N}{P} + (P+1) * B)$. If $B \ll N$ then the speedup of the partition phase will be $O(\frac{N}{P})$.

The next parallelism introduced is in the sorting phase. When the whole array is partitioned into P subarrays, each processor will run Quicksort on one of these subarrays. The stacks used by all processors for Quicksort during this phase will be shared by all processors. When a processor is finished with its own subarray, it will access other processors' stacks to help them until all keys are sorted. The speedup for this phase is approximately P with some some small synchronization overhead.

From the above analysis, we can see that the time complexity of the algorithm depends on the size of *block*, B, and the number of processors, P and does not depend on the distribution of keys.

3 Experimental Results

3.1 The SUN ENTERPRISE 10000 Platform

The SUN ENTERPRISE 10000, is a scalable, hardware-supported, cache-coherent, symmetric or uniform memory access (cc-UMA) multiprocessor machine. In ENTERPRISE 10000, every processor has its own cache and all the processors and memory modules attach to the same interconnect. In symmetric shared memory multiprocessor systems, data normally needs to be moved from point to point, while addresses often must be broadcasted throughout the system. Therefore the interconnect of E10000 uses a packet switched scheme with separate address and data paths. Data is transfered with a fast crossbar interconnect, and addresses are distributed with a broadcast router. The crossbar interconnect is constructed with two levels: global and local. In the Global level, there is a 16 byte wide, 16 x 16 crossbar that steers data packets between the 16 system boards. The global data crossbar connects the 16 system boards' ports, as Local level, together. At the Local level, "many-to-one" routers are used on the system boards to gather on-board requests and direct them to one port (per board). The address routing is implemented over a separate set of four global address buses, one for each of the four memory banks that can be configured on a system board. The buses are 48 bits wide including error correcting code bits. Each bus is independent, meaning that there can be four distinct address transfers simultaneously. Figure 7 graphically describes the architecture of the new SUN ENTERPRISE 10000. Main memory is configured in multiple logical units. All memory units, processors, and I/O buses, are equidistant in the architecture and all memory modules comprise a single global shared memory space. The latter means that the machine provides not only a global address space but also the memory access to any memory location is uniform. There are four coherency interface controllers (CICs). Each CIC connects to a separate global address router through one of the four global address buses. The CICs maintain cache coherency. The machine we used had 36, 249 MHz UltraSPARC

processors, which where divided into two logical domains: one with 4 processors as frontend and another one with 32 processors. Each CPU had a 16 KB first-level data cache and a 4 MB second-level cache. Our experiments were done on the 32 processor domain.

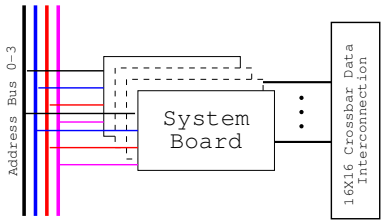


Figure 7: The architecture of the SUN Enterprise 10000

3.2 Parallel Sorting with Sample Sort

We compared parallel Quicksort with sample sort. Sample sort has been shown to be the best comparison-based and consequently general sorting algorithm for larger data sets [1, 4, 7, 16].

Sample sorting is a five-phases algorithm. First it divides the keys evenly among all P processors, and then, during the first phase each processor sorts locally its own keys. During the second phase each processor samples a fixed number of keys from its locally sorted keys. These sample keys in phase three of the algorithm are sent to one processor, that sorts them and selects $P-1$ out of them as sample splitters for the next phase. During the fourth phase each processor uses these $P-1$ splitters to partition the sorted input values and to decide locally the appropriate destinations of its partitioned keys. Finally, in the last phase of the algorithm, each processor uses mergesort locally to merge the key sequences send to it. In [12, 15] description of the algorithm and its analysis can be found together with a study on ways to decide how to perform the sampling of the keys and how to select the sample splitters and how these selection affect the load balance and the behavior of the program. Sample sort is a very efficient parallel sorting algorithm on distributed memory and message passing systems [14].

3.3 Sorting Benchmarks

The performance of sorting depends on the distribution of key values. We used the benchmark data sets that are described below. A detailed description and justification of the benchmarks can be found in [6]. In the following description, P is the total number of processors used and N is the total number of keys.

- In *Uniform or Random* the input keys are uniformly distributed. For the case where the input is comprised only by integers, it is obtained simply by calling the random number generator function *random*() to initialize each key. The function returns integers in the range from 0 to 2^{31}. For the experiments with *double* floating-point input, we divide the integer benchmark values with a prime number, 97 for our experiments.
- In *Gaussian* the input values follow the Gaussian distribution. For integer input, each key is the average of four consecutive integers returned by the *random*() function. For the experiments with *double* data type (floating-point) input, we normalized the integer inputs in the way described in the *Uniform* benchmark case.
- *Zero* is created by setting every key to a constant that is randomly selected by calling the function *random*().
- *Bucket* is obtained by setting the first $\frac{n}{P^2}$ elements assigned to each process to be random numbers between 0 and $\frac{2^{31}}{P} - 1$, the second $\frac{n}{P^2}$ elements at each process to be random numbers between $\frac{2^{31}}{P}$ to $\frac{2^{32}}{P} - 1$, and so forth. For the experiments with *double* data type (floating-point) input, we normalized the integer inputs the way described in the *Uniform* benchmark case.
- *Stagger* is obtained by setting the keys as follows: i) each processors with index i less than or equal to $\frac{p}{2}$, is assigned $\frac{n}{p}$ keys randomly chosen from the interval $[(2i+1)\frac{2^{31}}{P}(2i+2)\frac{2^{31}}{P}]$, ii) each processor with index i greater than $\frac{p}{2}$, is assigned $\frac{n}{p}$ keys randomly chosen from the interval $[(2i-P)\frac{2^{31}}{P}, (2i-p+1)\frac{2^{31}}{P}]$.

3.4 Results with Integer Input

We used five different input sizes of integers for each of the above five benchmarks: 8M, 32M, 64M, 128M and 256M. For our experiments we had exclusive access to 32 processors of a SUN ENTERPRISE 10000 machine. For parallel Quicksort, we choose the size of *block* to be 2048, with this number two *blocks* could be placed in the first level cache of our system at the same time. For the sample sort, we selected the sample size according to [12, 15] and the number of processors that we had access to. The speedup results are show in Figures 8, 9, 10, 11 and 12. The results for parallel Quicksort are labeled PQuick. For the experiments with 256M integer keys, we do not have any results for sample sort. This is because the kernel configuration of our systems does not allow single programs to allocate more than 2G bytes of memory and sample sort has higher memory needs than the parallel Quicksort presented here.

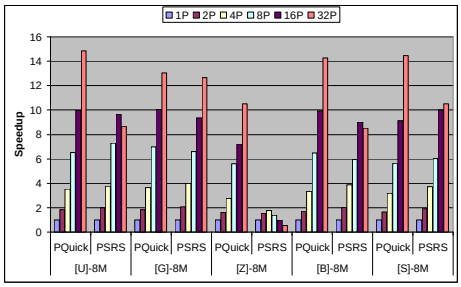


Figure 8: Experiments with 8M Integers

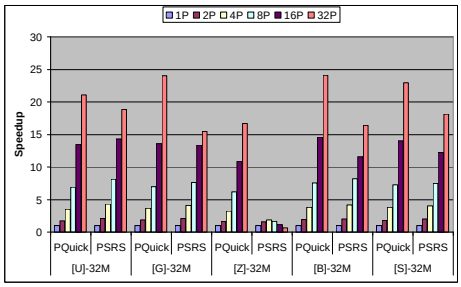


Figure 9: Experiments with 32M Integers

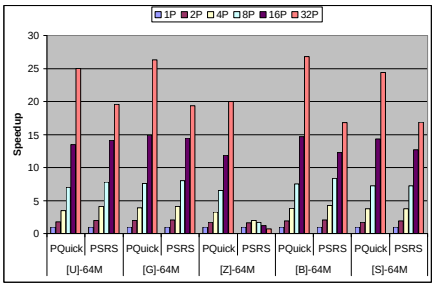


Figure 10: Experiments with 64M Integers

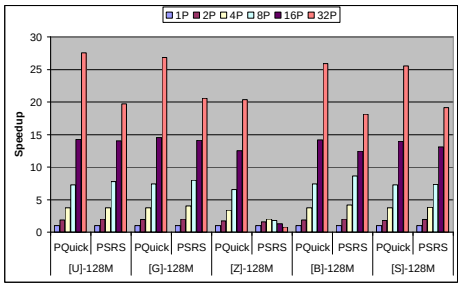


Figure 11: Experiments with 128M Integers

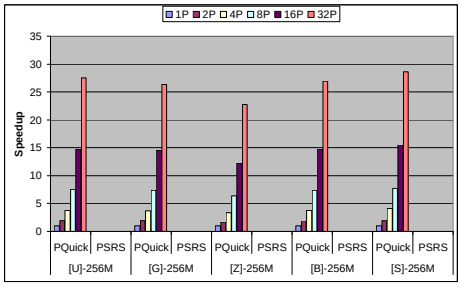


Figure 12: Experiments with 256M Integers

From the results obtained, we can see that for any number of processors and for any data size selected parallel Quicksort exhibits the following characteristics:

The execution time of parallel Quicksort is not sensitive to the distribution of the input

No. Proc.	[U]-64M		[G]-64M	
	PQuick	PSRS	PQuick	PSRS
1	139221763	157149728	147128678	160416695
2	79665604	80083084	74646110	77354959
4	39740592	38210913	37798256	38968839
8	19799215	20113143	19299378	20098710
16	10309607	11108192	9900953	11084988
32	5564001	8015688	5590313	8291338
No. Proc.	[B]-64M		[S]-64M	
	PQuick	PSRS	PQuick	PSRS
1	139882786	156416659	141337885	157289882
2	71928127	75946922	82174454	83648007
4	36426837	36768544	37653127	41555739
8	18661486	18728796	19496991	21802329
16	9518032	12768536	9878692	12357691
32	5221680	9297328	5796658	9318278

Table 1. The execution times (μs) with 64M Integers

data. Parallel Quicksort produced the same speedup for the the benchmark datasets *Uniform, Gaussian, Bucket*, and *Stagger*. For the benchmark *Zero*, the speedup that we got is always less than the speedup that we got on the other benchmarks. But, when looking at the absolute execution times, we can see that the sequential time for the *Zero* benchmark is about 30% faster than the sequential execution time for the other benchmarks while all other benchmarks have almost the same sequential execution times. The execution time of parallel Quicksort is not sensitive to the distribution of the data as mentioned in Section 2.2.

Better Execution Times and Speed-ups than Sample Sort. The absolute sequential execution times from sample sort are at least 15% slower than those of the parallel Quicksort in general, with only exception a small number of instances with processors between 2 to 8 and mostly in the benchmarks *Uniform* and *Gaussian*. All the instances in which sample sort outperforms the parallel Quicksort are shown in Table 3. The excess execution part in sample sort is due to the Mergesort that copies the data from one array to another array. From these figures, we can see that the speedups of the sample sort are better than the parallel Quicksort algorithm when the input sizes are small and the number of processors are small with the only excep-

No. Proc.	[Z]-8M		[Z]-32M	
	PQuick	PSRS	PQuick	PSRS
1	9643146	12912753	48593446	62147791
2	5928310	8383057	28895086	39120591
4	3508595	7251678	14904826	33235381
8	1723524	9285370	7848143	37933231
16	1334066	13362880	4478794	54547186
32	917237	23342432	2910912	93403783
No. Proc.	[Z]-64M		[Z]-128M	
	PQuick	PSRS	PQuick	PSRS
1	100276211	127420481	211486046	262376511
2	59113923	79046473	123347231	162749347
4	30510105	65319136	62769102	131289494
8	15400692	73537879	32329882	144721896
16	8475414	104793490	16883436	203698914
32	5020324	178521414	10375268	343280741

Table 2. The execution times (μs) of Benchmark *Zero*

tion the benchmark *Zero* (in *Zero* sample sort performs very poorly). Table 1 shows the execution times of the benchmark datasets *Uniform, Gaussian, Bucket*, and *Stagger* for parallel Quicksort and the sample sort algorithm with 64M integers (64 is in the median of 8 and 128, the smallest and the biggest input sizes for sample sort). In sample sort, the local sort phase is well load balanced. The mergesort will also be well load balanced if there are not many duplicate keys, which is the case in benchmarks *Uniform* and *Gaussian*. At the same time, when the number of processors increases, the cache capacity that is used in the algorithm increases. The increase of cache capacity would offset the parallelism overhead in the sample sort algorithm and even introduces superlinear speedup e.g. most experiments on 2 processors for all input data sizes with only exception the benchmark *Zero*. However, when the number of processors and/or the input size of the data is large enough, the cost for parallelism could not be offset by the increase of cache capacity any more. There is no superlinear speedup for sample sort for more than 16 processors and for the experiments with 128M integers only those with 2 processor show superlinear speedup. After that point, the speedup of sample sort lags behind the speedup of the parallel Quicksort. As the sequential execution time of sample sort is longer than the execution time of the parallel Quicksort, the sample sort can only beat the parallel Quicksort when a

No. Proc.	[B]-8M		[S]-8M	
	PQuick	PSRS	PQuick	PSRS
1	14269874	16414229	14325041	16452619
2	8244073	**8031141**	8614346	**8548883**
4	4252418	**4238064**	4485420	**4402723**
8	2204853	2760520	2537265	2722804
No. Proc.	[U]-32M		[U]-64M	
	PQuick	PSRS	PQuick	PSRS
1	68339788	76938437	139221763	157149728
2	38923895	**36649493**	79665604	80083084
4	19327928	**17931728**	39740592	**38210913**
8	9908154	**9465147**	19799215	20113143

Table 3. The execution times (μs) of instances (in **boldface**) where sample sort is faster than parallel Quicksort

large superlinear speedup is achieved.

On benchmark *Zero*, sample sort never performed better than the parallel Quicksort. The benchmark *Zero* is the most difficult for sample sort. The execution times of the parallel Quicksort and sample sort for benchmark *Zero* with 8M, 32M, 64M, 128M are shown in Table 2. When using the benchmark *Zero* the bottleneck is the fifth phase of mergesort. Since all keys have the same value, all keys will be send to one processor at the last phase of the algorithm to be mergesorted. When sorting inputs from *Zero*, only the first phase (local sorting) executes in parallel. On the other hand, parallel Quicksort delivers the best absolute execution times when sorting inputs from benchmark *Zero*. The reason is that the performance of Quicksort is optimal because any *pivot* will partition keys evenly; the best sequential time of parallel Quicksort for benchmark *Zero* also confirm this. Partitioning the keys also help the parallel algorithm by distributing the subarrays evenly among processors.

3.5 Results with Floating Point Inputs

In this section, we present the results of our experiments of sorting 64 bits *double type* (floating-point) input. Because of the same memory allocation limitation mentioned early, we have the results with data sizes of 8M, 32M, and 64M for both the parallel Quicksort and the sample sort and the results on 128M for parallel Quicksort only.

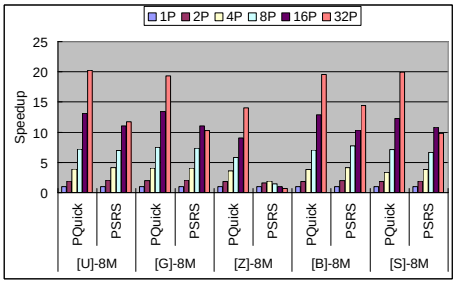


Figure 13: Experiments with 8M Double Floating-points

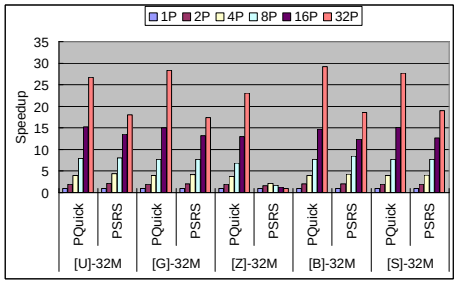


Figure 14: Experiments with 32M Double Floating-points

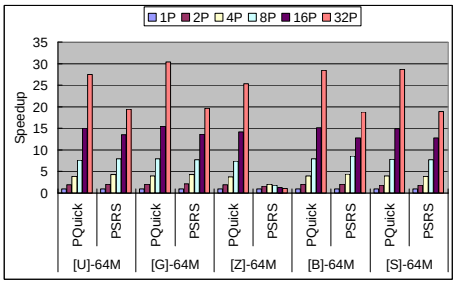


Figure 15: Experiments with 64M Double Floating-points

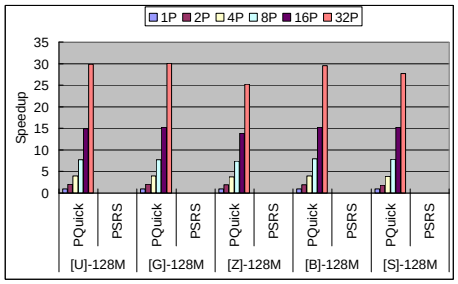


Figure 16: Experiments with 128M Double Floating-points

Figure 13, 14, 15 and 16 compare the speedup between the parallel Quicksort and sample sort algorithm with different double floating point benchmarks. We can observe the same trend: parallel Quicksort deliver almost the same speedup all the time; the speedup of the sample sort slowdown when the data size increases and when the number of processor increases.

4 Conclusion

Cache-coherent shared memory multiprocessors offer fruitful ground for algorithmic or programming techniques that were considered impractical before, in the context of high-performance programming, to develop and change a little the way we think about high-performance programming. We have implemented sample sort and a parallel version of Quicksort on a cache-coherent shared address space multiprocessors: the SUN ENTERPRISE 10000. Our computational experiments show that parallel

Quicksort outperforms sample sort. Sample sort has been long thought to be the best, general parallel sorting algorithms, especially for larger data sets. The parallel version of Quicksort is a simple fine-grain parallelization of Quicksort. Although fine-grain parallelism has long been thought to be inefficient for computations like sorting due to the synchronization overheads, efficiency was achieved by increased concurrency between communication and computation, This concurrency comes from the incorporation of non-blocking techniques especially when sharing data and subtasks. Quicksort might be a practical choice when it comes to general purpose in-place sorting both for uniprocessor and multiprocessor cc-DSM systems.

Acknowledgements

We thank Carl Hallen and Andy Polyakov from our Supercomputing Center, for their help on our inconvenient requests for exclusive use. Many thanks to Marina Papatriantafilou for her support and continuous flow of comments. Niklas Elmqvist and Anders Gidenstam provided us with many comments that made this paper much better to read.

References

[1] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the 3rd annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'91)*, pages 3–16, Hilton Head, South Carolina, July 1991.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[3] R. Dlekmann, J. Gehring, R. Lüling, B. Monien, M. Nübel, and R. Wanka. Sorting large data sets on a massively parallel system. In *Proceedings of the 6th Symposium on Parallel and Distributed Processing*, pages 2–9, Los Alamitos, CA, USA, Oct. 1994. IEEE.

[4] A. C. Dusseau, D. E. Culler, K. E. Schauser, and R. P. Martin. Fast Parallel Sorting Under LogP: Experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):791–805, Aug. 1996.

[5] P. Heidelberger, A. Norton, and J. T. Robinson. Parallel quicksort using Fetch-and-Add. *IEEE Transactions on Computers*, 39(1):133–137, Jan. 1990.

[6] D. R. Helman, D. A. Bader, and J. JáJá. A randomized parallel sorting algorithm with an experimental study. Technical Report CS-TR-3669 and UMIACS-TR-96-53,

Institute for Advanced Computer Studies, University of Maryland, College Park, MD, Aug. 1996.

[7] D. R. Helman, D. A. Bader, and J. JáJá. Parallel algorithms for personalized communication and sorting with an experimental study. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 211–222, Padua, Italy, June 1996.

[8] W. Hightower, J. Prins, and J. Reif. Implementations of Randomized Sorting on Large Parallel Machines. In *Proceedings of the 4th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'92)*, June 1992.

[9] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, Apr. 1962.

[10] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, second edition, 1998.

[11] A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 370–379, New Orleans, Louisiana, 5–7 Jan. 1997.

[12] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10):1079–1103, Oct. 1993.

[13] R. Sedgewick. Implementing Quicksort programs. *Communications of the ACM*, 21(10):847–857, Oct. 1978.

[14] H. Shan and J. P. Singh. Parallel sorting on cache coherent DSM multiprocessors. In *Proceedings of Supercomputing '99*, Portland, Oregon, USA, Nov. 1999.

[15] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[16] A. Sohn and Y. Kodama. Load balanced parallel radix sort. In *Proceedings of the International Conference on Supercomputing (ICS-98)*, pages 305–312, New York, July 1998. ACM press.

[17] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RC 5118, IBM T. J. Watson Research Center, Yorktown Heights, NY, Apr. 1986.

[18] E. W. Weisstein. Eric Weisstein's world of mathematics. Technical report, Wolfram Research, http://mathworld.wolfram.com/Quicksort.html, 1999.

[19] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings of Supercomputing'91*, pages 712–721, Albuquerque, New Mexico, Nov. 1991. IEEE.

Chapter 5

Non-blocking Data Sharing in Multiprocessor Real-Time System

This paper is an extended version of the paper appeared in the *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA '99)*.

Non-blocking Data Sharing in Multiprocessor Real-Time Systems ⋆

Philippas Tsigas and Yi Zhang

Department of Computing Science,
Chalmers University of Technology,
SE-412 60, Gothenburg, Sweden

Abstract

A non-blocking protocol that allows real-time tasks to share data in a multiprocessor system is presented in this paper. The protocol gives a way to concurrent real-time tasks to read and write shared data; the protocol allows multiple write operations and multiple read operations to be executed concurrently. Our protocol extends previous results and is optimal with respect to space requirements. Together with the protocol, its schedulability analysis and a set of schedulability tests for a set of random task sets are presented. Both the schedulability analysis and the schedulability experiments show that the algorithm presented in this paper exhibits less overhead than the lock based protocols.

1 Introduction

In any multiprocessing systems, cooperating processes share data via shared data objects. In this paper, we are interested in designing shared data objects for cooperative processes in real-time multiprocessor systems.

The challenges that have to be faced in designing of inter-process communication protocols for multiprocessor systems become more delicate when these systems have to support real-time computing. In real-time multiprocessor systems, inter-process communication protocols i) have to support sharing of data between different tasks e.g. on an operational flight program (OFP), tasks like navigation, maintaining of pilot displays, control of and communication with a variety of special purpose hardware,

⋆ Partially supported by ARTES, a national Swedish strategic research initiative in Real-Time Systems and TFR the Swedish Research Council for Engineering Sciences.

weapon delivery, and so on; ii) must meet strict time constraints, the hard real-time (HRT) deadlines; and iii) have to be efficient in time and in space since they must perform under tight time and space constraints. A nice description of fine challenges that inter-process communication protocols for real-time systems have to address can be found in [11].

The classical, well-known and most simple solution enforces mutual exclusion. Mutual exclusion protects the consistency of shared data by allowing only one process at time to access it. However, mutual exclusion i) causes large performance degradation especially in multiprocessor systems [12]; ii) leads to complex scheduling analysis since processes can be delayed because they were either preempted by other more urgent processes, or because they are blocked before a critical section by another process that can in turn be preempted by another more urgent process and so on. (this is also called as the convoy effect) [2]; and iii) leads to priority inversion in which a high priority task can be blocked for an unbounded time by a lower priority task [4]. Several synchronisation protocols have been introduced to solve the priority inversion problem for uniprocessor [4] and multiprocessor [3] systems. The solution presented in [4] solves the problem for the uniprocessor case with the cost of limiting the schedulability of task sets and also makes the scheduling analysis of real-time systems hard. 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 [3].

Non-blocking implementation of shared data objects is a new alternative approach for the problem of inter-process communication. Non-blocking mechanisms allow multiple processes to access a shared object at the same time, but without enforcing mutual exclusion to accomplish this. Non-blocking inter-process communication does not allow one process to block another process gives significant advantages over lock-based schemes because:

(1) it does not give priority inversion, avoids lock convoys that make scheduling analysis hard and delays longer.
(2) it provides high fault tolerance (processor failures will never corrupt shared data objects) and eliminates deadlock scenarios from two or more processes both waiting for locks held by the other.
(3) and more significantly it completely eliminates the interference between process schedule and synchronisation.

Non-blocking protocols on the other hand have to use more delicate strategies to guarantee data consistency than the simple enforcement of mutual exclusion between readers and writers of a data object. These new strategies on the other hand, in order to be useful for real-time systems, should be efficient in time and space in order to perform under the tight space and time constraints that real-time systems demand.

In this paper, we present an efficient non-blocking solution to the general readers/writ-

ers inter-process communication problem; our solution allows any arbitrary number of readers and writers to perform their respective operations. With a simple and efficient memory management scheme in it, our protocol needs $n + m + 1$ memory slots (buffers) for n readers and m writers and is optimal with respect to space requirements. Sorencen and Hemacher in [8] have proven that $n + m + 1$ memory slots (buffers) are necessary. Together with the protocol, its schedulability analysis and a set schedulability tests for a set of random task sets are presented. Both the schedulability analysis and the schedulability experiments show that the algorithm presented in this paper exhibits less overhead than the lock based protocol. Our protocol extends previous results by allowing any arbitrary number of tasks to perform read or write operations concurrently without trading efficiency. In previous work, Simpson [6], presented a non-blocking asynchronous protocol for task communication between 1 writer and 1 reader which needs 4 buffers. Chen and Burns [7] presented a non-blocking synchronous protocol for n readers and 1 writer that needs $n+1+1$ buffers. Kopetz and Reisinger [2] also presented a non-blocking synchronous protocol for n readers and 1 writer that contains a mechanism to configure the number of buffers to the application requirements, trading memory space for execution time. We also believe that the memory management scheme that we introduce in this paper and use in our protocol is of interest and can be used as an independent component with other non-blocking shared data object implementations.

The rest of this paper is organised as follows. In Section 2 we give a description of the basic characteristics of a multiprocessor architecture and describe the formal requirements that any solution to the synchronisation problem that we are addressing must guarantee. Section 3 presents our protocol. In Section 4, we give the proof of correctness of the protocol. Section 5 is devoted to the schedulability analysis and schedulability experiments that compare our non-blocking protocol with the lock-based one. The paper concludes with Section 6.

2 Problem Statement

2.1 Real-time Multiprocessor System Configuration

A typical abstraction of a shared memory multiprocessor real-time system configuration is depicted in Figure 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 cooperating tasks (processes) with timing constraints 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 at a time. The cooperating tasks now, possibly running on different processes,

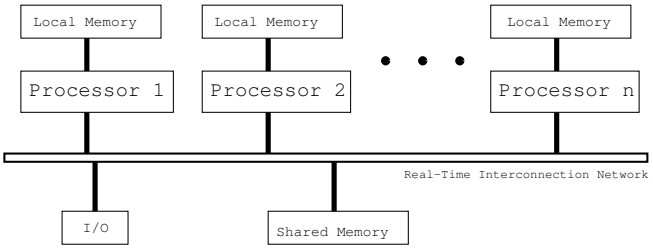


Figure 1: Shared Memory Multiprocessor System Structure

use shared data objects build in the shared memory to coordinate and communicate. Every task [1] has a worst case execution time and has to be completed by a time specified by a deadline. Tasks synchronise their operations through read/write operations to shared memory.

2.2 General Reader/Writer Problem

In this paper we are interested in the general read/write buffer problem where several reader-tasks, *(the readers)*, access a buffer maintained by several writer-tasks, *(the writers)*. There is no limit on the length of the buffer that can be increased by the writers on-line depending on the length of the data that they want to write in one atomic operation. The shared data object can be used to increase the word length of the system.

The accessing of the shared object is modelled by a history h. A history h is a finite (or not) sequence of operation invocation and response events. Any response event is preceded by the corresponding invocation event. For our case there are two different operations that can be invoked, a read operation or a write operation. An operation is called complete if there is a response event in the same history h; otherwise, it is said to be pending. A history is called complete if all its operations are complete. In a global time model each operation q "occupies" a time interval $[s_q, f_q]$ on one linear time axis ($s_q < f_q$); we can think of s_q and f_q as the starting and finishing time instants of q. During this time interval the operation is said to be *pending*. There exists a precedence relation on operations in history denoted by $<_h$, which is a strict partial order: $q_1 <_h q_2$ means that q_1 ends before q_2 starts; Operations incomparable under $<_h$ are called *overlapping*. A complete history h is linearisable if the partial order $<_h$ on its operations can be extended to a total order $\rightarrow_{\mathrm{h}}$that respects the specification of the object [1]. For our object this means that each read operation should return the value written by the write operation that directly precedes the read operation by this total order ($\rightarrow_{\mathrm{h}}$).

[1] throughout the paper the terms, *process* and *task*, are used interchangably

To sum it up as we are looking for a non-blocking solution to the general Read/Writer problem for real-time systems we are looking for a solution that satisfies that:

- Every read operation guarantees the integrity and coherence of the data it returns
- The behaviour of each read and write operation is predictable and can be calculated for use in the scheduling analysis
- Every possible history of our protocol should be linearisable.

We assume that writer-tasks have the highest priority on the host processor and no two writer-tasks execute on one host processor. Two writer-tasks may overlap but no write processes can be preempted by another process. Reader processes may have different priorities and may be scheduled with the writer process on the same processor. This is because writer-tasks usually interact with the environment (e. g. sampler for temperature or pressure), and they are of high prioritiy. To the best of our knowledge the above assumption holds in most real-time systems.

3 The Protocol

3.1 Idea description

Our construction divides memory into memory slots and uses a pointer that points to the slot with the latest written data. Each slot has a flag field used by the special memory management mechanism in the protocol. Through this mechanism, i) writers can find a safe slot to write without corrupting the slots from where overlapping readers are getting their values and ii) readers find the slot with the latest information.

In the worst case, n readers occupy n slots to read and m writers allocate m slots to write and the pointer points to another slot. So, in total we need at most $n + m + 1$ slots for our protocol. In [8], Sorensen and Hemacher showed that $n + m + 1$ slots is necessary for this problem.

3.2 Protocol Description

Our protocol uses the instructions *Compare_and_Swap* [2] and *Fetch_and_Add.* The respective specifications of these instructions are shown in Figure 2 and Figure 3. Most multiprocessor systems either provide these primitives or provide others that can be used to emulate these primitives.

[2] IBM System 370 was the first computer system that introduced Compare_and_Swap

```
Boolean Compare_and_Swap(WORD *mem, register WORD old, new)
{
   WORD temp = *mem;
   if (temp == old)
   {
     *mem = new;
     return TRUE;
   }
   else
     return FALSE;
}
```

Figure 2: The *Compare_and_Swap* atomic primitive

```
int Fetch_and_Add(int *mem, int increment)
{
  int temp;
  temp = *mem;
  *mem = *mem + increment;
  return temp
}
```

Figure 3: The *Fetch_and_Add* atomic primitive

Figure 4 presents commented pseudo-code for our nonblocking protocol. We use $n + m + 1 = TASKS' NUM + 1$ slots. Each slot has a field, called 'used' that is used by the memory management layer of our protocol. There are a number of values that this field can carry, these values together with an informal description of the associated information can be described as follows:

- $k = 0$: indicates that a writer has finished writing in the respective slot but no reader has read it yet
- $k > 0$: indicates that this is the slot with the most recent value and there are k readers reading this slot at this moment
- $-(n + m) < k < 0$: indicates that this is not the slot with the most recent value but there are $k + (n + m)$ readers currently reading it. These readers started their operations long ago when this slot had the most recent value
- $k = -(n + m)$: indicates that this is not the slot with the most recent value and that there is no reader reading it; the combination of these two makes it ideal for a writer to allocate this slot to its current operation and use it
- $k = -2(n + m)$: indicates that a writer has allocated this slot and is currently writing in it

The read/write protocol can be described informally as follows.

```
structure readwrite_buf
{
data: array[LENGTH_BUF] of Character;
used: integer;
  /*-2*TASK'NUM means allocated & writing data
  -TASK'NUM means slot is free
  -TASK'NUM + 1 ~ -1 means slot is free but with readers accessing it
   0 means validate data  */
} LF_buffer /* Lock-free buffer */
workbuf: array[TASK'NUM+1] of LF_buffer
dataptr: pointer to LF_buffer

function newbufcell(): pointer to LF_buffer
 for i=0 to TASK'NUM
  if (cas(work_buf[i].used,
      -TASK'NUM ,-2*TASK'NUM )
  return &work_buf[i]
 return NULL

function initdata
  for i=0 to TASK'NUM
      work_buf[i].used=-TASK'NUM
  dataptr=&work_buf[0]
function writebuf(data: pointer to datatye)
  temp = newbufcell()
 /* allocate a new buffer cell for write */
  writingsth(temp,data)
 /* write data into new buffer cell */
  temp.used=0
 /* means data valid in the cell */
  swap(dataptr,old,temp)
 /* use cas to achieve non-blocking write */
  faa(old->used, -TASK'NUM )
 /* if no reader then it will change to -TASK'NUM */
function readbuf(): pointer to datatype
 /* use fetch and add to mark that current block is in use. */
  loop
    reading=dataptr
  until (faa(reading->used,1) >= (-TASK'NUM))
 /* increase used by 1 if it is greater than -TASK' NUM.
    So it can not be recycled*/
  return=readingsth(reading)
  faa(reading->used,-1)
```

Figure 4: The Structure and Operations description for our non-blocking shared object

A writer runs the following steps whenever it wants to write data into the object:

(1) First, it allocates a free slot from the slots with flags entry $k = -(n + m)$. The writer uses *Compare_and_Swap* to read and change the flag from $k = -(n + m)$ to $k = -2(n + m)$ in one atomic operation, in this way if several writers want to allocate the same slot, the *Compare_and_Swap* atomic operation available at a hardware level will guarantee that only one will succeed.
(2) Then, it writes the data that it wants to write into that slot. If one reader tries to access that slot during this writing, the reader will give up and will be forced to try again as we will see in the readers protocol description.
(3) After the writing has finished, the writer changes the flag entry of that slot from $k = -2(n + m)$ to 0.
(4) As a next step the writer changes the data pointer variable to this new slot (so that consequent reads find a pointer to the fresh value) with the atomic primitive *Swap* and gets the pointer to the old slot at the same time.
(5) Finally, the writer changes the flag entry of the old slot from $k > 0$ to $k < 0$ by subtracting $n + m$.

Any write operation will finish after the 5 steps. The worst case execution time can be analyzed directly.

A reader performs the following steps during each read operation:

(1) reads the data pointer variable to get a pointer to the slot with the most recently written value
(2) uses the atomic primitive *Fetch_and_Add* to get and to add 1 on the value k of the flag entry of that slot
(3) there are three possibilities for the value that the fetch_and_add will return to the reader
    (a) $k \geq 0$, the slot has the most recent value written by all writers until now, and the reader can return this value, so go to step 5 to get the value.
    (b) $0 > k \geq -(n + m)$, the slot has the most recent value with respect to this specific reader from the linearisabilty point of view and the reader can return this value, so go to step 5 to get the value.
    (c) $k < -(n + m)$, the slot has been "recycled" by some writer and the data that the slot holds is invalid.
(4) goto step 1
(5) reads the data out of the slot
(6) uses the atomic primitive *Fetch_and_Add* with value -1 to change the value k of the flag entry of the slot

From the above description, a reader has a possibility to retry whenever it want to read a slot. We must find out whether there exists a upper bound on the number of reties a reader have to experience and only if it exists, we can derive the worst case

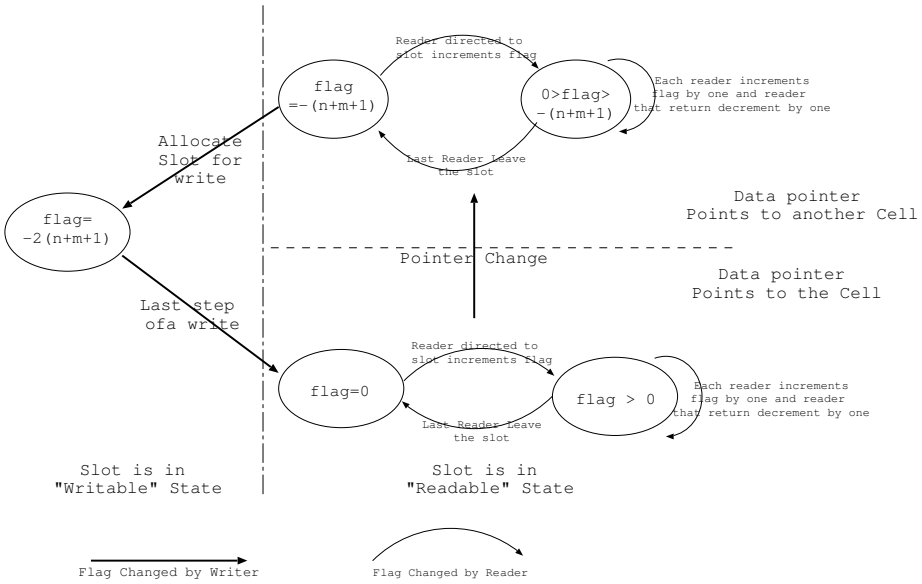


Figure 5: Slot state changing graph

execution time for any reader. The detailed analysis of the worst case behaviour of a reader will be given in Section 5.

A description of the different states that a slot can be in, together with the description of the actions that cause the change of these states, is depicted in Figure 5.

From the above description, it is easy to check that the protocol: i) guarantees continuous flow of information from the writers to the readers, without any blocking; ii) makes it possible to add or remove readers and writers on the fly without any change to the protocol of the other tasks; and iii) does not require information about how big the buffer has to be.

4 Correctness

4.1 Linearisability Model and Definitions

In [9] one can find a formalism for the notion of the atomic buffer and the global time assumption that we adopt. We assume that each operation Op has a time interval $[s_{Op}, f_{Op}]$ on a linear time axis. We can think of s_{Op} and f_{Op} as the starting and finishing times of Op. Moreover, we assume that there is a precedence relation on operations which is a strict partial order (denoted by '$<_h$'). Semantically, $a <_h b$ means that operation a ends before operation b starts. If two operations are incomparable under $<_h$, they are said to **overlap**.

A **reading function** R_B for a buffer B is a function that assigns a write operation W to each read operation R on B, such that the value returned by R is the value written by W. It is assumed that there exists a write operation, which initialises the buffer B, that precedes all other operations on B.

A complete history on a buffer is an execution of an arbitrary number of operations according to its protocol. With the linearisability definition described in section 2.2, a complete history on a buffer B is linearisable if there is a total order $\rightarrow_{\mathrm{h}}$ on the set of all the operations of the run such that: (i) The total order $\rightarrow_{\mathrm{h}}$ extends the precedence relation $<_h$, (ii) every read operation r returns a value which is equal to the value written by a write operation that directly precedes r in the total ordering $\rightarrow_{\mathrm{h}}$. An implementation of a buffer is linearisable if all its complete histories are linearisable.

4.2 Correctness Proof

In this subsection will show that any complete history is linearisable. Our proof uses a widely used lemma in the area that can be found also in [10].

Lemma: A complete history h is linearisable iff there exist a function mapping each operation in h to a rational number such that the following three conditions are satisfied:

Precedence if one operation precedes another, then the value of the latter is at least that of the former

Uniqueness different write operations have different numbers

Integrity for each read operation there exists a write operation with the same value which it doesn't precede.

Now we associate a tag field with the pointer variable that associates a tag value to each value that is written in this variable in the following way: every time a write operation updates the pointer with a swap operation the tag is increased by one. The tag field and variable are auxiliary and are introduce only to help the proof.

Now, it is easy to associate a tag value to each write operation and each read operation in a way that guarantees the above mentioned conditions. A write operation is associated with the tag value that it writes and a read operation is associated with the tag value that it read when it read the pointer variable; in this way different write task are associated to different values (tag values only increase) and read tasks are associated with the respective write operations that wrote the values that they return.

Theorem 1 *Each complete history of our protocol is linearisable.*

5 Schedulability Analysis

5.1 Worst Case Analysis

As we mentioned in Section 2 we assume that writers have the highest priority on the nodes where they are running. In our protocol i) write operations are guaranteed to finish independently from interleaving with other task operations after a finite number of steps; ii) read operations are subject to retry under certain circumstances.

For our analysis we will use the following notation:

- P_w: period of the writer
- C_i: Compute time of task i
- T_r: read latency caused by one retry
- R_i: the response time fo task i
- D_i: the deadline and period of task i
- N_i: maximum number of interventions
- W_i: the worst case executing time of task i

Because tasks on different processors are decoupled, the scheduling problem for a multiprocessor system is converted to the scheduling problem on several uni-processor systems. If we can find out the worst case executing time of each task then we can use any scheduling algorithm for uni-processor systems to schedule them.

Observation 1 *A read operation will do a retry only if there are more than two write operations overlap with it.*

A read operation will do a retry if it finds the slot which it want to read is recycled by a write operations. It is obvious that if no write operation overlaps with a read operation, the reader will succeed without any retry. An overlap between a reader operation and other read operations does not cause reader to retry. If only one write operation overlaps with a read operation, there are two possibilities:

(1) The write operation change the pointer variable with its *swap* statement before the read operation read the pointer variable. In this case, the read operation will read out the freshest value without any retry.
(2) The write operation change the pointer variable after the read operation read the pointer variable, the read operation will return a value satisfied linearisability without retry

If a read operation overlap with more than two write operations, the following scenaria might happen and lead the read operation retry.

(1) The read operation, R_0 reads the pointer variable and finds that slot A contains the freshest value.
(2) A write operation W_1 changes the pointer variable to another slot with a *Swap* and frees slot A by change the tag of A to $-(n+m)$ with a *Fetch_and_Add*.
(3) Another write operation W_2 allocate slot a by changing the tag of A to $-2*(n+m)$, with a *Compare_and_Swap*.
(4) The read operation R_0 uses *Fetch_and_Add* to change the tag of A and find the slot is recycled. R_0 does a retry.

From the above observation, we can tell that to let a reader retry, a *Swap*, a *Fetch_and_Add* and a *Compare_and_Swap* statement must take effect between the *read* and *Fetch_and_Add* of the read operation. If the read operation is not preempted by any other tasks, there is no time for the above statement to happen. This gives us the following lemma.

Lemma 1 *The number of retry of a read operation is less or equal to the number of preemption the operation experienced.*

The number of retry of a read operation is also effected by the maxium number of writers during the total time which the reader experience preemption.

Lemma 2 *Assume the total preemption time a read task i expereinced is Pre_i, then the max number of interventions for each read operation is bounded by*

$$N_i = \left\lceil \frac{Pre_i}{2 * P_w} \right\rceil$$

5.2 Rate-Monotonic

The response time of a reader task i can be calculated with the following fomular:

$$R_i = C_i + \sum_{j \in HP(i)} \left\lceil \frac{D_j}{R_i} \right\rceil C_j + \min\left(\sum_{j \in HP(i)} \left\lceil \frac{D_j}{R_i} \right\rceil, \left\lceil \frac{\sum_{j \in HP(i)} \left\lceil \frac{D_j}{R_i} \right\rceil C_j}{2 * P_w} \right\rceil \right) T_r$$

where $HP(i)$ represents all tasks who run on the same processor as task i and whose priorities is higher than the priority of task i. The first two items are the same as original response time analysis. The third item calculates the extra execution time introduced by retrying of the reader task in the worst case. The item has a minimum of two numbers: the maxium number of preemption given by Lemma 1 and the maxium number of interference given by Lemma 2.

5.3 Scheduling experiment

We conduct the following scheduling experiments:

In the first experiment all tasks need to access to the multi-word buffer. There are fixed number of writers that write to the buffer with period P_w. Every reader task needs the same computing time and all tasks have the same deadline and period. We try, in the experiments, to schedule as many reader tasks as possible. All the experiment parameters are listed as following:

C_i: Compute time of task i: 800usec
T_{rw}: Read/Write buffer time: 100usec
D_i: the deadline of task: 10msec
T_r: read latency caused by one retry: 10usec
P_w: period of the writer: 1msec
P_r: period of the reader: 10msec
$bcnt$: the number of buffers for the message, used by Kopetz's protocol: 4

For the non-blocking algorithm, we calculate the response time of reader tasks. In this experiment, we compare our algorithm with a lock-based algorithm. The lock-based protocol is using PCP to avoid Priority inversion. Figure 6 shows the result of the schedule simulation. The solid line represents the ideal number of reader tasks based on the lock algorithm that can be scheduled and the dashed line represents that of our non-blocking protocol. As it can be seen, the schedulability of our non-blocking protocol is the same as or better than the other protocol all the time.

In the second experiment, we consider different randomised computing times for different reader tasks. All parameters except the computing time for the reader tasks are the same as the first experiment. The worst case execution time for the non-blocking algorithm is calculated with the equation in section 5.2. We create two sets of randomised computing time tasks with different random seeds. Figure 7 shows the result of two task sets. The results also show that with randomised computing time our non-blocking protocol also gives better scheduling results compared to the lock-based one.

6 Conclusion

In this paper, we presented a non-blocking protocol that allows real-time processes to share data in a multiprocessor system. The protocol provides the means for concurrent real-time tasks to read and write the shared data; the protocol allows multiple write and multiple read operations to be executed concurrently. Together with the

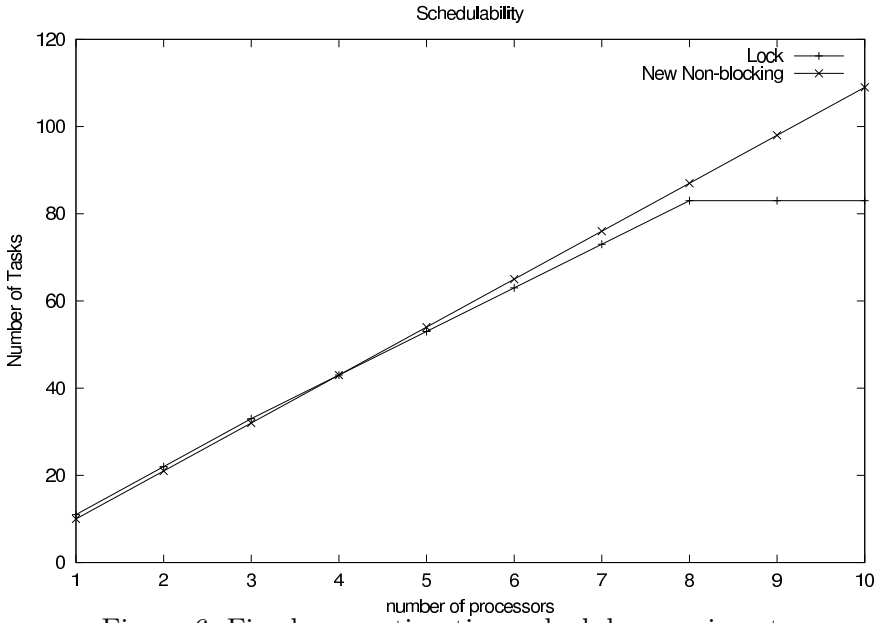


Figure 6: Fixed computing time schedule experiment

protocol its schedulability analysis and a set of schedulability tests are given. Both the schedulability analysis for a task set and the schedulability experiments show that the algorithm presented in this paper exhibit less overhead than the lock based protocols.

The space requirements of our protocol is the same with the lower bound shown by Sorencen and Hemacher [8]. Our protocol only needs $n + m + 1$ memory slots where n is the number of readers that perform their read operations concurrently and m is the number of writers that can write concurrently. Our protocol extends previous results by allowing any arbitrary number of tasks to perform read or write operations concurrently without compromising efficiency.

We believe that the memory management scheme that we introduce in this paper and used in our protocol can be used as an independent component with other non-blocking shared data object implementations; we are currently looking at it.

Acknowledgement

We'd like to thank Hans Hanson for his valuable comments and discussions.

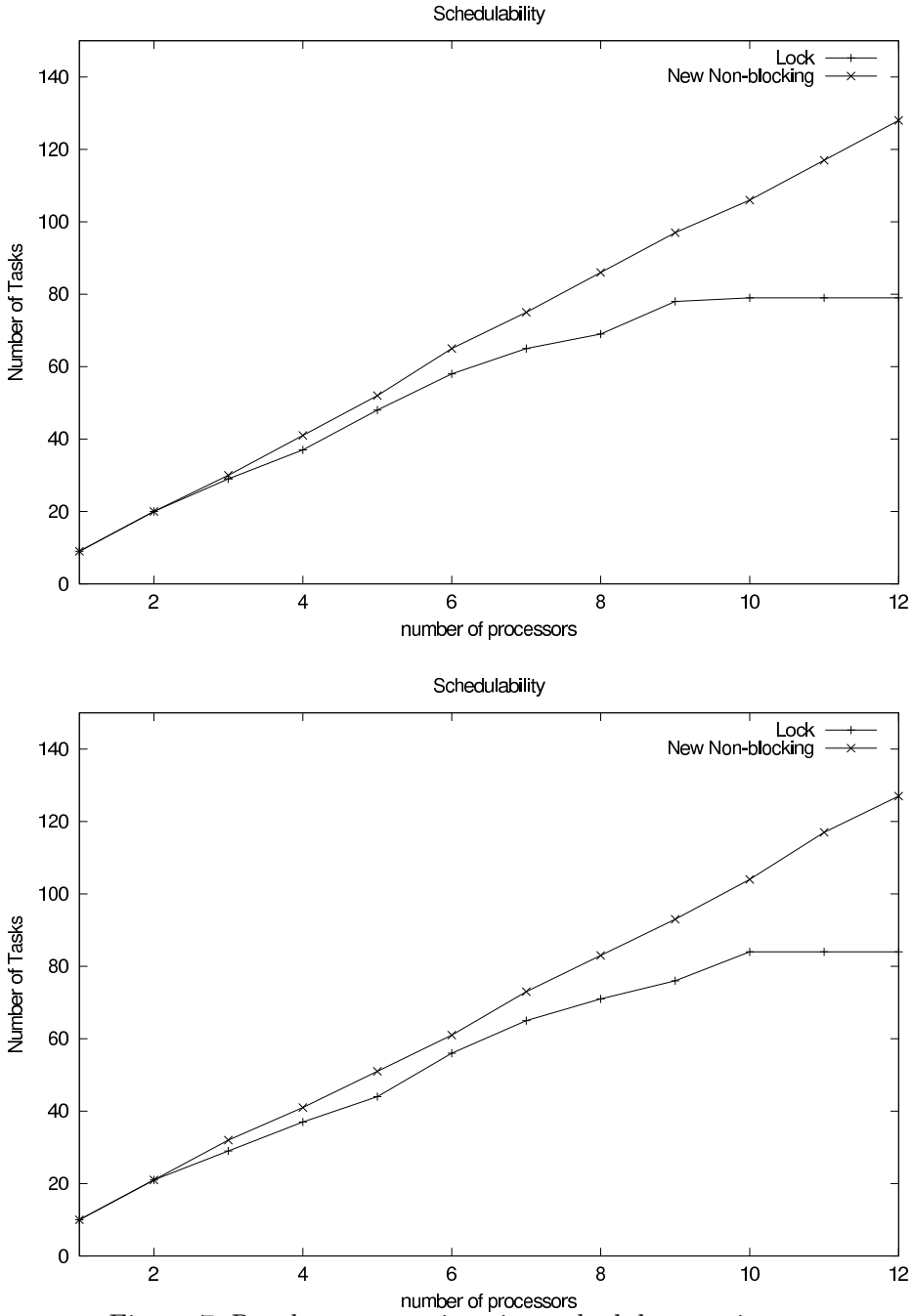


Figure 7: Random computing time schedule experiment

References

[1] Maurice P. Herlihy, Jeannette M. Wing Linearizability: A correctness condition for concurrent objects, *ACM Transactions on Programming Languages and Systems*, 12(3), July 1990, pp. 463-492.

[2] H. Kopetz, J. Reisinge The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronisation Problem, in *Proceedings of the Real-Time Systems Symposium*, 1993, pp. 131-137.

[3] R. Rajkumar Real-Time Synchronization Protocols for Shared Memory Multiprocessors, in *Proceedings of the 10th International Conference on Distributed Computing Systems*, 1990, pp. 116-123.

[4] L. Sha and R. Rajkumar, J. P. Lehoczky Priority Inheritance Protocols: An Approach to Real-Time Synchronization, *IEEE Transactions on Computers*, Vol. 39, 9 (Sep.) 1990, pp. 1175-1185.

[5] S. V. Adve, K. Gharachorloo Shared Memory Consistency Models: A Tutorial, *IEEE Computer*, Dec. 1996, pp. 66-76.

[6] H. R. Simpson Four-slot fully asynchronous communication mechanism, *IEE Proceedings*, Jan. 1990, pp. 17-30.

[7] J. Chen, 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.

[8] P. Sorensen, V. Hemacher A Real-Time System Design Methodology, *INFOR*, 13, 1 (Feb.) 1975, pp. 1-18.

[9] L. Lamport On interprocess communication, part i: basic formalism, part ii: basic algorithms, *Distributed Computing*, 1986, pp. 77-101.

[10] M. Li, J. Tromp, P. Vitanyi How to share concurrent wait-free variables *Journal of the ACM*, July 1996, pp. 723-746.

[11] Stuart Faulk, David Parnas On Synchronization in Hard-Real-Time Systems, *Communication of ACM*, Vol. 31, Mar. 1988, pp. 274-287.

[12] A. Silberschatz, Peter B. Galvin, *Operating System Concepts*, Addison Wesley, 1994.

Chapter 6

Efficient Wait-Free Queue Algorithms for Real-Time Synchronization

This paper is in submission. It is also available as the Technical Report 2002-05 of Department of Computing Science, Chalmers University of Technology.

Efficient Wait-Free Queue Algorithms for Real-Time Synchronization ⋆

Philippas Tsigas and Yi Zhang

Department of Computing Science,
Chalmers University of Technology,
SE-412 60, Gothenburg, Sweden

Abstract

The Real-Time Specification for Java provides protected, non-blocking, shared access to objects used by both regular Java threads (`java.lang.Threads`) and the time-critical `NoHeapRealtimeThreads`. Such access is offered via a set of *wait-free queue* classes. These classes are provided explicitly to enable communication between the real-time `NoHeapRealtimeThreads` and the regular Java threads; they have a unidirectional nature with one side of the queue for the real-time threads and the other one for the non-real-time ones. This set of *wait-free queue* classes is of big importance not only to real-time Java but also to any real-time synchronization system.

Efficient algorithmic implementations of these queue classes are presented in this paper. The algorithms are designed to exploit the unidirectional nature of these queues and they are more efficient, with respect to space complexity, compared to previous wait-free implementations, without losing in time complexity. The space complexity of our algorithms is $O(M + N)$ where N is the maximum number of concurrent tasks that the *Queue* supports and M is the size of the *Queue*. The space complexity of the previous best solution is $O(N * M)$. The time complexity of our algorithm and the previous best one is $O(N)$ for each task. Experiments we've performed suggest that our algorithms are typically $9\% - 36\%$ faster than the previous best one.

⋆ This work is partially funded by the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se), supported by the Swedish Foundation for Strategic Research.

1 Introduction

In this work, we present algorithmic implementations of the wait-free queue classes of the Real-time Specification for Java. These implementations are designed to have the unidirectional nature of these queues in mind and they are more efficient, with respect to space, compared to previous wait-free implementations, without losing in time complexity. The wait-free queue classes proposed in the Real-time Specification for Java are of general interest to any real-time synchronization system where hard real-time tasks have to synchronize with soft or even non real-time tasks.

Java was originally designed by Sun to facilitate the development of embedded system software [8], it was designed more to simplify programming than to enable programmers to write software that complies reliably with real-time constraints. In order to facilitate its major goal of operating system and hardware independence, in areas such as thread behavior, synchronization, interrupts, memory management, and input/output Java's expressiveness was designed to be weak. However, these are among the critical areas needing explicit management for meeting application timeliness requirements. On the other hand because the simplicity, the object orientation appeal and most significantly the Java platform's independence offer, there are greater cost-savings potential in the real-time domain than in the desktop and server domains: real-time computing systems use many different processor types and operating systems. The matching between Java and real-time software development led to the formation of the Real-Time for Java Experts Group (RTJEG) that began developing the Real-Time Specification for Java (RTSJ) in March 1999 under the Java Community Process [4]. The goal was to provide a platform that let programmers correctly reason about the temporal behavior of executing software. In their effort to do so, they enhanced Java in 7 areas [5]:

(1) thread scheduling and dispatching,
(2) memory management,
(3) synchronization and resource sharing,
(4) asynchronous event handling,
(5) asynchronous transfer of control,
(6) asynchronous thread termination,
(7) physical memory access.

In the area of synchronization and resource allocation the RTSJ introduces a set of new synchronization mechanisms, a set of wait-free queue classes. This wait-free queue classes became obligatory for the support of protected, concurrent access of data by both regular threads (`java.lang.Threads`) and the time-critical `NoHeapRealtimeThreads`[1] that were introduced by the RTSJ to have an implicit

[1] We are going to use the `NHRT` term to denote the `NoHeapRealtimeThread` for the rest of

execution eligibility logically higher than the garbage collector. `NHRT`s can not access (allocate or even reference) any objects in the heap, which means that they should be able to run while the garbage collector is running. `NHRT`s are introduced for code with a very low tolerance of non-scheduled delays. The RTSJ does not mandate algorithms or specific time constants for such, but requires that the semantics of the implementation are met.

1.1 Wait-Free Synchronization

In the area of synchronization and resource allocation applications often need to share serializable resources. Java also provides the ability to introduce concurrency mechanisms into applications. Traditionally, concurrency mechanisms for synchronization and resource allocation use mutual exclusion to protect the consistency of the shared data by allowing only one process at a time to access the method/class. If one declares a method to be `synchronized`, Java will prevent more than one thread from executing that method at any time. The keyword `synchronized` is the only mechanism required by the specification that can enforce mutual exclusion in the traditional sense in RTSJ as in Java. Mutual exclusion i) causes large performance degradation; ii) leads to a complex scheduling analysis since tasks can be delayed, because they were either preempted by other more urgent tasks, or because they are blocked before a critical section by another process that can in turn be preempted by another more urgent task and so on (This is also known as the convoy effect); iii) and most significantly for the real-time systems it leads to priority inversion in which a high priority task can be blocked for an unbounded time by a lower priority task [16]. Several synchronization protocols have been introduced to solve the priority inversion problem for uniprocessor [16] and multiprocessor [14] systems. The solution presented in [16] 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. The situation is even worse in a multiprocessor real-time system, where a task may be blocked by another task running on a different processor [14]. For the RTSJ, it was decided that the least intrusive specification for allowing real-time safe synchronization is to require that implementations of the Java keyword `synchronized` includes one or more algorithms that prevent priority inversion among real-time Java threads that share the serialized resource. But still the use of the `synchronized` keyword implementing the required priority inversion algorithm was not sufficient to both prevent priority inversion and allow the special `NHRT`s, that were introduced in the RTSJ to give the means to time-critical tasks to get an execution eligibility logically higher than the garbage collector, to do so [4]. The RTJEG realized that a non-blocking, protected access to objects shared between `NHRT` and regular Java threads is the only

---

the paper.

solution to the problem. The decision of the RTJEG to provide wait-free queues to enable communication between the regular Java threads and the real-time `NHRT` follows research in recent years, in which several researchers have investigated the use of wait-free shared-object algorithms as an alternative to lock-based mechanisms in object-based real-time systems [1, 2, 3, 7, 15, 17]. Moreover, research in real-time operating systems [6, 9, 11, 19] has also shown how to incorporate wait-free techniques in real-time kernels.

Wait-free implementation of shared data objects is an alternative approach for the problem of inter-task communication and synchronization. Wait-free mechanisms allow multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion to accomplish this. A wait-free implementation of a shared data object guarantees that every process accessing the object always completes its operation in a bounded number of its own steps, regardless of interleaving (process halts, failures, scheduler behavior). Wait-free inter-task communication does not allow one task to block another task and thus gives significant advantages over lock-based schemes because:

(1) it does not give priority inversion and avoids lock convoys that make scheduling analysis hard and delays longer.
(2) it provides high fault tolerance (processor failures will never corrupt shared data objects) and eliminates deadlock scenarios from two or more tasks both waiting for locks held by the other.
(3) more significantly, it completely eliminates the interference between process schedule and synchronization; thus, giving a more compositional framework to argue about the 'task' behavior under the effect of the scheduler and the synchronization mechanism. This gives the ability to a task to keep its execution eligibility during communication and synchronization, and this was the feature that was incumbent in the RTSJ.

All the above mentioned advantages come from the fact that wait-free solutions are not penalized from the negative effects of blocking.

1.2 Related Work and Our Contribution

Concurrent FIFO queue data structures are fundamental data structures used in many programs and algorithms and, as can be expected, many researchers have proposed implementations for them. Although there are many non-blocking implementations (see [18] for references), only few of them are wait-free. In a non-blocking algorithm, some operations are allowed to perform unbounded number of steps when they are concurrent with other operations; this, of course, is not allowed in a wait-free algorithm. All previously mentioned constructions (wait-free or not) were targeted to-

wards asynchronous systems; such constructions require hardware support for strong synchronization primitives such as `Compare-and-Swap` etc. These primitives are not available in the Real-Time Specification for Java. As a matter of fact in the RTSJ only read and write memory operations are supported. The reason is the hardware-independence property that the RTSJ wants to preserve.

Recent research at the University of North Carolina has shown that wait-free algorithms can be simplified considerably in real-time systems by exploiting the way that processes are scheduled for execution in such systems [1, 15]. In particular, if processes are scheduled by priority, then object calls by high-priority processes automatically appear to be atomic to lower-priority processes executing on the same processor. Consequently they show an implementation of the `Compare-and-Swap` from reads and writes in a priority-based uniprocessor system [15]. In a consequent paper [2], a wait-free implementation of a linked-list from compare-and-swap for priority-based systems is presented. These results combined can offer an efficient implementation, with respect to time complexity, that satisfies the specifications of the wait-free queue classes in RTSJ. The space complexity of this implementation is $O(N * M)$ where N and M is the maximum number of concurrent tasks that the queue supports and the size of the queue respectively; the time complexity of this implementation is $O(N)$ for each task.

In this paper implementations of RTSJ queue classes with $O(M+N)$ space complexity and $O(N)$ time complexity are presented. Experiments we've performed suggest that our algorithms are typically $9\% - 36\%$ faster than the previous best one. The wait-free queue classes that are provided by RTSJ have been designed to enable communication between the real-time `NoHeapRealtimeThreads` and the regular Java threads; they have a unidirectional nature with one side of the queue (read or write) for the real-time threads and the other one (write or read, respectively) for the non-real-time ones. The implementations presented in this paper are designed having the unidirectional nature of these queues in mind in order to gain efficiency; to the best of our knowledge our implementations are the first unidirectional wait-free queue implementations in the literature.

The remainder of the paper is organized as follows. In Section 2 we give a brief introduction to the basic design features of the RTSJ and Section 3 presents our algorithms. In Section 4 we summarize our experimental results. We conclude in Section 5.

2 Synchronization and Resource Sharing in RTSJ

In this section, a short description of the basic design features of RTSJ, that we had to take into account in our implementation, are presented.

The RTSJ is designed for multithreading priority-based uniprocessor systems. The application program must see the minimum 28 priorities as unique; for example, it must know that a thread with a lower priority will never execute if a thread with a higher priority is ready. If threads with the same exact priority are eligible to run, they will execute in FIFO order. The RTSJ provides wait-free queue classes to provide protected, non-blocking, shared access to objects accessed by both regular Java threads and `NHRT`. These classes are provided explicitly to enable communication between the real-time execution of `NHRT` and regular Java threads. Basically, there exist two different new queue classes in RTSJ: the `WaitFreeWriteQueue` class and the `WaitFreeReadQueue` class.

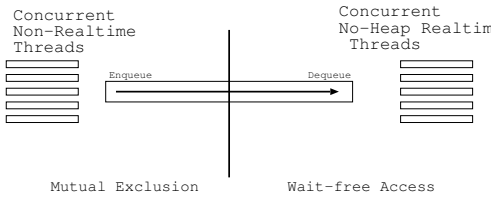


Figure 1: The `WaitFreeReadqueue` class

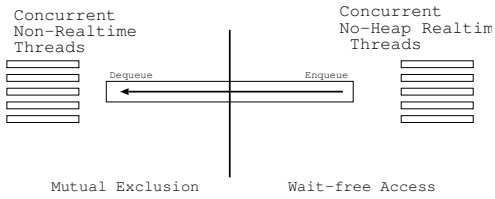


Figure 2: The `WaitFreeWriteQueue` class

Both these queue classes are unidirectional. The information flow for the `WaitFreeWriteQueue` is from the real-time side to the non-real-time one, as shown in Figure 1. The information flow for the `WaitFreeReadQueue` is from the non-real-time side to the real-time one, as shown in Figure 2. When a `NHRT` wants to send data to a regular Java thread, it uses the `write` (real-time) operation of `WaitFreeWriteQueue` class. Regular threads use the `read` (non-real-time) operation of the same class to read information. The `write` side is non-blocking and wait-free, so that `NHRT` will not experience delays from the garbage collection. The `read` operation, on the other hand, is blocking. Since the `write` is wait-free and the arrival dynamics are incompatible, data can be lost. To avoid delays of allocating memory elements, class constructors statically allocate all memory used for queue elements, giving the queue a finite limit. The `WaitFreeReadQueue` class, which is unidirectional from non-real-time to real-time, works in the converse manner.

The third queue class that is described in the RTSJ is the `WaitFreeDeQueue` class and is implemented by putting back-to-back a `WaitFreeWriteQueue` and a `WaitFreeReadQueue`. The formal specification for the `WaitFreeWriteQueue`, `WaitFreeReadQueue` and `WaitFreeDeQueue` classes, can be found in [5].

3 The Algorithms

Algorithmically the implementations of the two wait-free queue classes (`WaitFreeWriteQueue` and `WaitFreeReadQueue`) are quite similar. In this paper, we present the implementation of the `WaitFreeWriteQueue` class to illustrate the ideas behind the constructions.

```
public class SeqQueue {
  RTQueueCell head, tail;
  RTQueueCell dumbcell;
  void SeqQueue() {
    head = dumbcell;
    tail = dumbcell;
    dumbcell.data = null;
    dumbcell.next = null;
  }
  public java.lang.Object read() {
    RTQueueCell temp;
    temp = (RTQueueCell)head.next;
    if (temp != null)
      head = temp;
    return temp;
  }
  public boolean write(java.lang.Object object) {
    RTQueueCell temp;
    temp = new RTQueueCell();
    temp.data = object;
    temp.next = null;
    //tail and tail,next will be shared
    //read/write in concurrent implementation
    tail.next = temp;
    tail = temp;
    return TRUE;
  }
}
```

Figure 3: The Sequential implementation of the queue

3.1 Informal Description

To simplify the presentation of our algorithm, we start with a simple sequential queue implementation. We will then discuss how to extend this sequential algorithm to a concurrent queue implementation with the specifications that we are looking for. The Java-pseudo-code for this sequential queue is shown in Figure 3.

As it can be seen in Figure 3 we implemented algorithmically the queue using a singly

linked list. For efficiency reasons, we choose the front of the queue, where we only delete nodes, to be the head of the list, and the rear of the queue, where we only insert, to be the tail of the list. In this way we only use the operations of the linked list that modify the head and the tail of the list. In order to minimize the interference between the `write (enqueue)` and `read (dequeue)` [2] operations, we introduce a *dumbcell* in the empty list. In this way, when executing a `dequeue` operation, only the *head* needs to be checked in order to see whether the queue is empty or not. If the *next* field of the *head* is *null*, the queue is empty. Therefore, the `dequeue` operation needs only to check the *head* variable and only the *tail* needs to be checked for the `enqueue` operation. For the initialization for the simple sequential queue we define a *dumbcell*, with `null` in its next field, and let the *head* and the *tail* of the queue point to it, statements 5 to 8 in Figure 3.

Figure 4 shows the structure of the cells of the linked list that we are using. The class `RTQueueCell` has two public members: one is for the data entry, the other is the next pointer that singly links the elements of the list.

To extend the sequential version to a concurrent wait-free queue implementation, first we will use a simple announce-and-help scheme for the `enqueue` operations. The announce-and-help scheme uses the priority-based scheduler to achieve wait-freedom. This scheme is based on the task priorities to guarantee that an operation will finish in a bounded number of steps regardless of the status of the other operations, as follows: First, each enqueue-task announces the data (writes a pointer to the memory where the data are) that it wants to enqueue in a special *Announcement* array. The enqueue-task with priority i will use the ith position of the array. After the announcement step, the enqueue-task reads and helps the data that have been announced in the array one by one, starting from the lowest priority up to its own priority. During this helping phase, if an enqueue-task A is not going to be preempted by a higher priority task, then all current enqueue operations, with lower priority than the priority of A, that are announced will be helped/enqueued by A. If the enqueue task A is preempted by a higher priority task B during its helping phase, then there are two cases:

```
public class RTQueueCell {
        public java.lang.Object data = null;
        public java.lang.Object next = null;
 }
```

Figure 4: Definition of the queue cell

• B is not an enqueue-task on the same queue: then the task A will continue its

[2] Throughout the paper, the terms queue `write` and `enqueue` are used interchangeably. The same also holds for the terms queue `read` and `dequeue`. To distinguish between the queue `read/write` and normal read/write memory operation, we are using typewriter type style for the queue operations and serif type style for the memory ones.

program steps after B finishes, from the same queue-state from which it was preempted. Dequeue operations on the same queue are executed by tasks that have lower priority and therefore they can not preempt enqueue operations on the same queue.

• B performs an enqueue operation on the same queue: in this case B is going to announce its task and help all lower priority tasks that are announced and its own task that has just been announced. Therefore task A will be helped by B. Because the priorities are bounded, there always exists a task which will not be preempted by another enqueue task. Therefore, all tasks that announced their operations will be helped (either by themselves or by higher priority tasks).

The RTSJ has been designed for uniprocessor systems. The RTSJ as well as Java support only plain memory synchronization primitives like atomic reads and writes to memory locations, as opposed to other advanced synchronization primitives like `Compare-and-Swap`. The weak vocabulary of Java in memory synchronization comes again from the fact that different processors support different memory synchronization primitives and Java was designed to be hardware-independent. Although reads and writes are very weak synchronization primitives in the context of general asynchronous systems, by exploiting the fact that the tasks are executed by priority, it has been shown that they are universal primitives for priority-based uniprocessor systems [15].

During the design phase of any shared data object, a problem that arises from the use of memory read/write operations is the "enabled late-write" problem [15]. The "enabled late-write" problem arises when a low priority task A is preempted while it is about to write to a memory position, and is preempted by other tasks that access and modify the same memory position. When task A resumes running, it overwrites the previous "fresh" value with an "old" one. Anderson et al. [15] proposed a majority voting scheme to overcome the problem. Their scheme requires $2N-1$ memory words to solve "the enabled late-write" problem for 1 word.

In this paper we propose a new more efficient scheme to face the "enabled late-write". The new scheme tries to avoid the problem from the beginning by:

(1) Making sure that, when a task A is preempted before writing to position p, all other tasks that write on to p, (while A is preempted) write the same value that A wanted to write. In order to establish this, we guide the tasks to go through the same computational steps as A when they have to decide about the value that they want to write on the same memory location.
(2) When the above is possible, we organize the shared variables that might suffer from the "enabled late-write" problem as arrays that carry information that can be used algorithmically to determine the correct/new value of the variable.

We believe that, the same idea can be used when algorithmically designing other

shared objects for the RTSJ.

```
public class WaitFreeWriteQueue
{
... ...
private MemoryArea MemPool;
private java.lang.Object[] Announcement;
private RTQueueCell[] tail;
private RTQueueCell head;
// get the minPriority from the scheduler
private int minPriority;
// get the maxPriority from the scheduler
private int maxPriority;
        ... ...
}
```

Figure 5: Shared private variables for `WaitFreeWriteQueue`

```
RTQueueCell dumbcell = new RTQueueCell();
... ...
Announcement = new
java.lang.Object[maxPriority + 1];
tail = new RTQueueCell[maxPriority + 1];

for (i=minPriority;i<=maxPriority;i++) {
Announcement[i] = null;
tail[i] = null;
}
tail[minPriority] = dumbcell;
head = dumbcell;
dumbcell.data = null;
dumbcell.next = null;
```

Figure 6: Initialization for `WaitFreeWriteQueue`

The wait-free part in this class is the part that implements the `enqueue` operations. The wait-free `write` operations share also the private variable $MemPool$ that hold references to a `MemoryArea`[3]. The shared private variables for our `WaitFreeWriteQueue` are as shown in Figure 5. All `RTQueueCell`s should be allocated from the `MemoryArea`. The *Announcement* array is used to hold the different `enqueue` operations. The *tail* and *Announcement* arrays are of equal length, equal to the real-time priority level supported by the scheduler. For the head of the queue we use the simple variable *head*. The *minPriority* and *maxPriority* are the minimum and maximum priorities that real-time threads can be assigned, respectively. This information can be obtained from the scheduler. All shared variables will be initialized when constructing the queue.

[3] The RTSJ introduces the `memory area` concept, which is a region of memory outside the garbage-collected heap that you can use to allocate objects. The RTSJ uses the abstract class `MemoryArea` for this.

The initialization is similar to that of the sequential version. Because we now use an array to represent the tail, we need to initialize this array in a way that makes it easy for the algorithm to find the correct `tail` (the *dumbcell*), when a task accesses the queue for the first time. When a task accesses the *tail* array, it checks from the the cell of the lowest priority task to the highest to find a non-null cell. Henceforth, we initialize the cell corresponding to the lowest priority point to the *dumbcell*. During the initialization part, we also need to initialize the *Announcement* array with the value `null`, that means that there are no announced operations. The pseudo-code for the initialization is described in Figure 6. The initialization of the local variables is part of the pseudo-code description of the algorithm described in Figure 7.

Now, in order to extend the sequential version that we presented at the beginning of this section to the concurrent one that we are aiming for, we first need to make sure that the shared read/write operations to the *tail* and the *tail.next* variables (the shared variables of our implementation where overwriting might take place) do not suffer from the "enabled late-write" problem. The wait-free `enqueue` operation is presented in the `write` function below. The announce-and-help scheme, that is used in our implementation, uses the priority-based scheduler to achieve wait-freedom. Each priority is mapped to the respective entry of the array *Announcement*. An `enqueue` operation first gets the priority of its thread, then it allocates a free cell from the memory area assigned to the queue. The memory area is where the queue and its internal elements are allocated. After writing the data in the free cell, the task announces this cell in the *Announcement* array at the index that is associated to its priority. This constitutes the last part of the announcement phase. This is, as we will see, the "linearizability point" of the enqueue operation at the linearizability history. After it announces the object that it wants to enqueue in the *Announcement*, a task will enter the helping phase that was described at the beginning of this subsection. The helping phase is described in relation to the implementation pseudo-code in Figure 7. During the helping phase, an `enqueue` operation with priority i helps the tasks with priority $j \leq i$ that have been announced in the array *Announcement*, one at a time, starting from the operation with the smallest priority that it can find (statement 17 in the implementation). For each such operation, it finds the tail of the queue (statements 22-32 on the protocol); then puts the data announced at the end of the *tail* of the queue; then changes the *tail* variable to point to the new position; and finally cleans the *Announcement*$[j]$.

The wait-free queue class we designed are used to provide communication between NHRTs and regular Java threads. To untangle the effect of garbage collection, statical memory management is needed for nodes of the queue class. Statical memory management is not the subject of this paper; but, a simple scheme is presented here, in the Appendix A, to show the feasibility of such a scheme. Other better and more efficient schemes are possible. In the Appendix B, we describe implementations of the other methods supported by the `WaitFreeWriteQueue` class.

```
1 public boolean write(java.lang.Object object) {
2 boolean find = false;   int i,j, mypriority;
3 RTQueueCell tempcell, temptail=null;
4 java.lang.Object tempAnnounce;
5 java.lang.Thread currentone;
6 //Find your priority
7 currentone = java.lang.Thread.currentThread();
8 mypriority = currentone.getPriority();
9 //Allocate a cell in the MemoryArea
10 try {
11   tempcell = MemPool.newInstance(RTQueueCell);
12   tempcell.data = object; tempcell.next = null;
13 } catch(OutofMemoryError x) { return false; }
14 //Announce one's operation
15 Announcement[mypriority]=tempcell;
16 //Enter helping phase and help lower priorities and oneself
17 for(i=minPriority;i<=mypriority;i++) {
18   tempAnnounce=Announcement[i];
19   if (tempAnnounce == null) continue;
20   //Try to find the actual tail
21   find = false;
22   for(j=minPriority;j<=maxPriority;j++) {
23   if (tail[j]!=null)
24    if (tail[j].next == null) {
25      find = true;
26      break;
27    }
28   }
29   //Continue in Figure 8
```

Figure 7: Wait-free enqueue operation for the `WaitFreeWriteQueue`

Figure 9 shows the lock-based `read` operation of the `WaitFreeWriteQueue` class. It's a straight forward implementation that uses mutual exclusion to serialize concurrent dequeue operations.

3.2 Correctness Proof

In the helping phase two sets of variables are used, the *tail* array (tasks help to enqueue data at the tail of the queue) and the *Announcement* array; all of them are shared variables and can be read and written by different tasks. In the implementation, the value of a variable *Announcement*[i] changes from *null* to a non-null value, when a task with priority i announces its `enqueue` operation. The value of the same variable changes back to *null* when the item that the `enqueue` operation wanted to enqueue was enqueued by the same operation or by another higher priority enqueue operation. If there are e enabled writes that are ready to write to *Announcement*[i] then at least

```
30   if (find)
31   //No preemption detected, the actual tail has been found.
32     temptail =  tail[j];
33   else
34   //Preemption detected! There are 2 possibilities
35     if(Announcement[i]!=null) {
36   //Preempted, helped but not completely. Help the task with
37   //priority i to update tail and Announcement array
38       tail[i]=(RTQueueCell)tempAnnounce;
39       Announcement[i] = null;
40       continue;
41     }
42     else
43   //Preempted and helped by a higher priority task that helped
44   //the task that you were helping and you. Return!
45       return true;
46
47   //Did you preempt a lower priority task when it was on
48   //statements 60 and 61 ?
49   if(temptail==tempAnnounce)
50   {
51   //Help it to update the Announcement
52     Announcement[i] = null;
53     continue;
54   }
55   //Preempted by higher priority task that helped you completely?
56   if(Announcement[i]==null)
57     return true;
58   //Enqueue the announcement
59   temptail.next=tempAnnounce;
60   tail[i]=(RTQueueCell)tempAnnounce;
61   Announcement[i] = null;
62  }
63 return true;
64 }
```

Figure 8: Wait-free enqueue operation for the `WaitFreeWriteQueue` (continue)

$e-1$ of them are helping operations and have priority higher than i and want to change the value of the $Announcement[i]$ from *non-null* to *null*. The one "enabled late-write", that might exist, is the write with priority i that wants to announce a new enqueue operation. This write will not be scheduled before the other pending writes, with higher priority, take place, and thus, its write will not be overwritten by them. The above proves the following lemma:

Lemma 1 *$\forall i$, $minPriority \leq i \leq maxPriority$, $Announcement[i]$ will not suffer from the "enabled late-write" problem.*

Lemma 2 *When a task A is preempted just before it writes to the tail array, a higher*

```
1 public synchronized java.lang.Object read() {
2 RTQueueCell tempcell;
3 tempcell = head.next;
4 if (tempcell != null)
5 head = (RTQueueCell) tempcell;
6 return tempcell;
7 }
```

Figure 9: Lock-based dequeue operation for `WaitFreeWriteQueue`

priority task will write the same content to the same position in the tail array.

Proof: The decision of what to write on the tail is based on the contents of the *Announcement* and *tail* arrays. If a higher priority task preempted task A just before A was to write the *tail* array, then, since, nothing changed on the *Announcement* and *tail* of the object from the time that A read them, the higher priority task, that preempted A, will compute the same value to write to the *tail* array. □

If we would have used a simple *tail* variable for the queue, as it is used in the sequential implementation of the queue, the "enabled late-write" problem could have happened in the *tail* variable. To solve that problem, we organize the *tail* of the queue as an array. Each location in the array corresponds to the respective priority. All tasks with the same priority will be executed in a FIFO order and use the same location in the array. Each enqueued item from a task with priority i will become the tail of the queue once, and the ith index of the *tail* will point to it. In our construction, all tasks that try to help a task with priority i that has been announced, are going to write to the *tail* array at the index that corresponds to priority i. For example, when a task A is helping with task C, it is preempted by another high priority task B and has an enabled write on the *tail*array. Then the new task B will help the same task C also and will go through the same computational steps and will update the same entry of the *tail* array with the same value as the preempted enabled write of task A. This is guaranteed from Lemma 2. In this way, the "enabled late-write" problem can not take place in any $tail[i]$ variable. This sketches a proof of the following lemma.

Lemma 3 *$\forall i, minPriority \leq i \leq maxPriority$, $tail[i]$ will not suffer from the "enabled late-write" problem.*

Now, we need to give a way for the tasks to read the *tail* array and compute the real tail of the queue. Each item in the *tail* array has been the real tail of the queue at some point in time but only one of them is the current tail of the queue. In our implementation, there is at most one *tail* entry that has the value *null* on its next field. As we are designing a concurrent queue, an enqueue operation can be preempted anywhere; a task A can be preempted between statement 59 and 60 by a task B. The *tail* array then will have no element with the value *null* on its next field. The actual tail in this case should point to the object enqueued by a task C, which is being helped

by task A (A executes statement 59 and 60 i only when it is helping another task). During the helping phase of its enqueue operation, task B need to find the tail of the queue and uses the local variable *temptail* to store it. In the pseudo-code, when task B executes statements 22-28, it goes through the *tail* array from the lowest priority to the highest priority and tries to find the one index in the array with *null* in the 'next' field, if there is one. If there is no overlapping with enqueue operations, task B will find the index with *null* in the next field. It will store the value in *temptail* (statement 32).

Lemma 4 *temptail.next will not suffer from the "enabled late-write" problem.*

Proof: When a task A with priority j helps a task with priority i that has been announced, where $i < j$, all items in the announcement array from *minPriority* to $i - 1$ should have the value *null* because task A starts its helping phase from *minPriority*, and, tasks with priority less than j can not preempt task A and make changes in the announcement array. Before task A updates the next field of the tail of the queue (statement 59), nothing changes in the tail array and the announcement array. If a task B with priority k, where $k > j$, preempted task A, task B will add its own announcement in position k and nothing between *minPriority* to j in the announcement array will change. Therefore task B will help the announcement of the task with priority i and will find the same tail and make the same decision with task A and finally put the same value as A on *temptail.next*. □

If task B overlapped with other `enqueue` tasks, then task B might not find an index on the array with *null* in the next field. If this happens, task B has already enough information to find the actual tail of the queue and help the preempted task to update the tail of the queue. To see this, let us look at the possible ways that the above could have happened; there are two cases:

• Task B preempts the lower priority task A, when A was between statements 59 and 60; e.g. A had just finished enqueueing the data before updating the tail of the queue. The actual tail of the queue at this point is the task which is being helped by both task A and task B. Task B will help task C to update the tail when B runs statements 38-39.

• Task B is preempted by a higher priority task D and D updates the *tail* array in such a way that task B misses the actual tail of the queue when B is scheduled back. In this case, D will help all lower priority tasks. So, task B just needs to stop its helping phase and return. B will detect that it has been helped and return in statement 45.

The above sketches a proof that items are going to be put on the singly link-list one after the other.

Since different tasks are going to try to help the same task, we need to show that an item is not going to be enqueued more than one time. That is the reason that statement 49 is used from task B to detect that it has preempted a task A when A was between statements 60 and 61 of its pseudo-code. When the preemption happens, the announcement has been added to the queue as a tail but not been cleaned, which has been read by task B in *temptail*. If such a preemption is detected, the task B will help task A to clean the announcement array, when task B executes statement 52. As both of them want to write *null* at the same position, no "enabled late-write" problem exist. Statement 56-57 is used from task B to detect that it had been preempted by a higher priority tasks D and to conclude that task D has helped the task that B was helping when preempted.

The following lemma also proves that it is necessary and sufficient for a task to help other tasks with priority up to its own priority.

Lemma 5 *When a task A with priority i announces an enqueue data in the Announcement array, all elements of the array from $i+1$ to $maxPriority$ have the value null.*

Proof: Assume towards a contradiction that $Announcement[j]$ is not *null*, where $j > i$. Then there must exist a task B with priority j that announced its enqueue object in *Announcement* array and the announcement by task B hasn't been "cleaned". $Announcement[j]$ is cleaned as the last step of the enqueue operation. The task A must preempt task B to announce its enqueue object in *Announcement*, in order to preempt task B, $i > j$ must hold. This is a contradiction,

As the contents of the *Announcement* array from index $i+1$ to index $maxPriority$ are *null* when task A announce its operation, there is no need to help them. It is sufficient to help tasks with priority up to i. As task A can preempt any lower priority task after it has announced, it is necessary to help them. □

The access of the queue is modeled by a history h. A history h is a finite (or not) sequence of operation invocation and response events. Any response event is preceded by the corresponding invocation event. For our case there are two different operations that can be invoked, a `write` operation or a `read` operation. An operation is called complete if there is a response event in the same history h; otherwise, it is said to be pending. A history is called complete if all its operations are complete. In a global time model each operation q "occupies" a time interval $[s_q, f_q]$ on a linear time axis ($s_q < f_q$); we can think of s_q and f_q as the starting and finishing time instants of q. During this time interval the operation is said to be *pending*. There exists a precedence relation on operations in a history denoted by $<_h$, which is a strict partial order: $q_1 <_h q_2$ means that q_1 ends before q_2 starts; Operations incomparable under $<_h$ are called *overlapping*. A complete history h is linearizable if the partial order $<_h$ on its operations can be extended to a total order $\rightarrow_{\mathrm{h}}$ that respects the specification

of the object [10].

In the remains of this section, we prove that our implementation is a concurrent linearizable queue implementation. In order to do so, we will show that any possible history ($<_h$), produced by our implementation, can be extended to a total order ($\rightarrow_{\mathrm{h}}$) by using a "linearization point" for each operation. The "linearization point" of an operation is an atomic point on its execution, during which the operation takes effect.

Lemma 6 *The* write *to the announcement array is the "linearization point" for the* `write` *operations.*

Proof: By Lemma 5, when a task A with priority i executes statement 15, all items of the announcement array from $i+1$ to $maxPriority$ have the value $null$. Task A will help all operations announced in *Announcement* from the lowest to its own priority. Enqueue operations with lower priority than i that have been announced by executing statement 15, will be enqueued before A's announcement on the *anouncement* array. If the current task A is preempted by a higher priority task after executing statement 15, the announcement will be enqueued before the announcement of the task with higher priority. So, the execution order of statement 15 in the `write` operation extends the precedence partial order to a total order that respects the FIFO specifications of the `WaitFreeWriteQueue` class. □

Lemma 7 *The* read *of the next field of the head of the queue is the "linearization point" for the* `read`*s of the queue.*

Proof: Since, mutual exclusion is used between `read` operations on the queue, the order in which they get access to the critical section totally orders them. But as the `read` operations of the queue have lower priority than all the `write` operations of the queue, they can be preempted and run concurrently with `write` operations. As all high priority tasks will appear atomic to a low priority task, a `write` operation will only be observed if it starts executing before statement 3 of the `read` operation. By selecting then the execution of the statement 3 of a `read` operation as its "linearizability point", all operations are totally ordered with a relation that extends the precedence relation and respects the specification of the `WaitFreeWriteQueue` class. □

The lemma below proves that our queue implementation is a FIFO one and that no enqueued element gets lost. For simplicity we introduce `write`(*empty*) operations in the history when the queue is empty.

Lemma 8 *In a complete history such that* $\texttt{write}(x) \rightarrow_h \texttt{write}(y)$*, then* $\texttt{read}(x) \rightarrow_h \texttt{read}(y)$.

Proof: From the assumption, we have that `write`$(x) \rightarrow_{\mathrm{h}}$ `write`(y) which means x is announced before y. If there is no overlapping, x will be put in the list before y as in the sequential version. If overlapping exists, by lemma 5, the task A who announces y has higher priority than the task who announces x. As task A will help from the task with lowest priority to itself, it will put x in the list before y. As `read` uses mutual exclusion, only one `read` operation processes the list from the head to the tail. So `read` operations will find x first. □

The lemma below proves that dequeue operations dequeue items that have really been enqueued.

Lemma 9 *In a complete history, if x is* `read`*, then it has been* `write`*d, and* `write`$(x) \rightarrow_h$ `read`(x)

Proof: The linearizability point of the `read`(x) is the point where the `read` operation reads the *next* field of the *head*. Because a `write` operation announces its operation and the announcement takes place before the helping phase, and in the helping phase the announcement will be put in the next field of a $tail[i]$. If x is read, then some task must write in the field during its helping phase. Helping an announcement can only happen after it has been announced by some task in the announcement array. So, the x read by a task must have been written before the `read` operation. □

The above lemmas give us the following theorem.

Theorem 1 *Our algorithm for the* `WaitFreeWriteQueue` *is a linearizable FIFO concurrent queue without the "enabled late-write" problem.*

4 Experimental Results

To evaluate the performance of our algorithm, we compare it with the best previously known solution that was proposed in [2]. We performed our experiments on a real-time environment simulator based on the Real-Time Threads (RTT) Package [12]. The RTT provides priority-based preemptive scheduling for real-time threads that is similar to the real-time java virtual machine specification. To the best of our knowledge there is no implementation of a real-time java virtual machine available yet.

We performed experiments with 1, 2, 4, 8 and 16 concurrent enqueue tasks. The parameters of the task sets were selected so that the tasks are schedulable and are

based on the following formulas:

$$T_i = (n-i) * T, \qquad i = 0, ..., (n-1)$$

$$C_i = \begin{cases} (n-i) * C, & i = 1, ..., (n-1) \\ (18-(n-1)), & i = 0 \end{cases}$$

T and C are the period and computation time of the highest priority task respectively and $\frac{C}{T} = 20$. T_i and C_i are the period and computation time of the task i respectively. N is the number of concurrent enqueue tasks. The task sets are scheduled with the rate-monotonic scheduling algorithm. With the task parameters described above, the processor utilization is $U = \sum \frac{C_i}{T_i} = 90\%$ for all task sets and all task sets are schedulable under rate-monotonic scheduling [13].

The results of our experiments show that our algorithm does not decrease only dramatically the space requirements but is also from 9% to 36% faster than best previously known. The average response times of the highest, middle and lowest priority task of each task set are shown in Figure 10.

5 Conclusion

Efficient implementations of the RTSJ queue classes are presented in this paper. The wait-free queue classes proposed in the Real-time Specification for Java are of general interest to any real-time synchronization system where hard real-time tasks have to synchronize with soft or even non real-time tasks. The implementations presented here are designed with the unidirectional nature of these queues in mind and they are more efficient, with respect to space, compared to previous wait-free implementations without losing in time complexity. The space complexity of our algorithms is $O(M + N)$ where N is the maximum number of concurrent tasks that the *Queue* supports and M is the size of the *Queue*. The space complexity of the previous best solution is $O(N * M)$. The time complexity of our algorithm is $O(N)$. Experiments we've performed suggest that our algorithms are typically $9\% - 36\%$ faster than the previous best one.

There are several ways that future research in wait-free synchronizations can contribute to real-time Java. A very promising, we believe, is the investigation of practical wait-free implementations of garbage collection in the RTSJ model. The garbage collector is a central component of the Java environment. Wait-free implementation will improve the programmers ability to correctly reason about the temporal behavior of their Java programs.

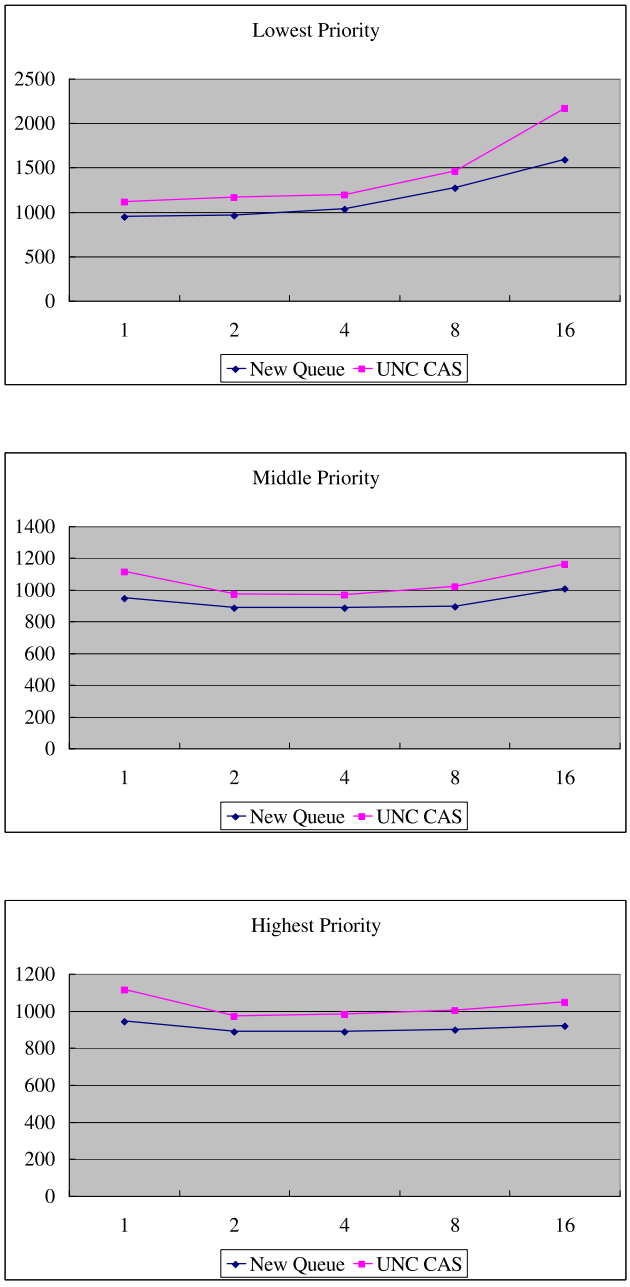


Figure 10: Response times of the lowest priority, middle priority and highest priority task

References

[1] J. Anderson, R. Jain, and S. Ramamurthy. Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. In *Proceedings of the 18th IEEE Real-Time*

Systems Symposium (RTSS '97), pages 111–122. IEEE, Dec. 1997.

[2] J. Anderson, S. Ramamurthy, and R. Jain. Implementing wait-free objects on priority-based systems. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC '97)*, pages 229–238. ACM, Aug. 1997.

[3] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems*, 15(2):134–165, May 1997.

[4] G. Bollella and J. Gosling. The real-time specification for java. *IEEE Computer*, 33(6):47–54, June 2000.

[5] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000. URL: `www.javaseries.com/rtj.pdf`.

[6] G. C. Buttazzo. HARTIK: A real-time kernel for robotics applications. In *Proceedings of the Real-Time Systems Symposium*, pages 201–205. IEEE Computer Society Press, Dec. 1993.

[7] A. Ermedahl, H. Hansson, M. Papatriantafilou, and P. Tsigas. Wait-free snapshots in real-time systems: Algorithms and their performance. In *Proceedings of the fifth International Conference on Real-Time Computing Systems and Applications (RTCSA '98)*, pages 257–266, Dec 1998.

[8] J. Gosling and H. McGilton. *The Java Language Environment: A White Paper*. SUN Microsystems, Inc., 1995.

[9] M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University, 1999.

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

[11] G. Lamastra, G. Lipari, G. C. Buttazzo, A. Casile, and F. Conticelli. Hartik 3.0: A portable system for developing real-time applications. In *Proceedings of the 4th IEEE International Workshop on Real-Time Computing Systems and Applications (RTCSA)*, pages 43–50, October 1997.

[12] S. L. A. Lo, N. C. Hutchinson, and S. T. Chanson. Architectural considerations in the design of real-time kernels. In *Proceedings of the 14th Real-Time Systems Symposium, December 1–3, 1993*, pages 138–147. IEEE, 1993.

[13] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, M. Gonza, and L. Harbour. *A Practitioners Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real Time Systems*. Kluwer Academic Publishers, 1993.

[14] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *10th International Conference on Distributed Computing Systems*, pages 116–123. IEEE, IEEE Computer Society Press, May–June 1990.

[15] S. Ramamurthy, M. Moir, and J. Anderson. Real-time object sharing with minimal system support. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 233–242. ACM, May 1996.

[16] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept. 1990.

[17] P. Tsigas and Y. Zhang. Non-blocking data sharing in multiprocessor real-time systems. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications(RTCSA '99)*, pages 247–254, Dec 1999.

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

[19] K. Zuberi, P. Pillai, and K. G. Shin. EMERALDS: a small-memory real-time microkernel. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 277–299, 1999.

Appendix A: Memory Management for Queue Node

A simple statical memory management scheme for the wait-free queue class is provided in this section. Other statical memory management schemes can be also used for the queue class presented here.

Each task will be statically associated an array of queue nodes during initialization. The length of the array depends on the expected length of the enqueue. A flag is associated to each item in the array; this flag is used to tell whether the node is free or not. When an enqueue task want to allocate a node, it will go through the array and return the first free node if such a node exists. The enqueue task will mark the node as 'not free'. A dequeue task will free the node by setting the flag to be 'free' after it finish its operation on the node.

No mutual exclusion is required for allocating and freeing a node with the above scheme. For each array, there is only one enqueue task associated with the array and the dequeue task which got this node will only write on the flag of this node. At any time, only one task can write the flag: if the node is free, only the enqueue task will update the flag to make the node occupied; if the node is not free, only a dequeue task will update the flag to make the node free. Therefore, the above scheme is wait-free. As mentioned before, memory management is not the subject of this paper, but we provide a simple and working statical memory management for our scheme to show the feasibility of such a scheme. Other better and more efficient schemes are possible.

Appendix B: Other Methods Supported by the Class

Here we describe how to implement the other methods supported by the `WaitFreeWriteQueue` class.

For wait-free queue classes in RTSJ, programmers need to manage nodes themselves. Functions `isFull` and `size` are related to memory management and their implementation can be varied according to the memory management scheme. We show how these two functions can be implement under the memory scheme presented in Appendix A. As each task has its own associated array, `isFull` will return **True** when there is no free node in the associated array; otherwise it will return **False**. The `size` of the wait-free queue is equal to the number of allocated nodes by all tasks. The `force` function requires a task to overwrite an old value when the queue is full. We can implement the `force` function with the helping of the memory management scheme. When allocating a node, the last allocated node will be recorded. When all nodes are occupied, a task can not enqueue a new data. If it calls the `force` function, it will enqueue the value into the last allocated node.

The implementations of `clear` and `isEmpty` functions are shown in Figure and .1 do not depend on memory management scheme.

```
public void clear()
{
int i;
for (i=minPriority;i<=maxPriority;i++)
{
Announcement[i] = null;
tail[i] = null;
}
tail[minPriority] = DumbNull;
head = DumbNull;
DumbNull.next = null;
}
public boolean isEmpty()
{
if (head.next == null)
return true;
else
return false;
}
```

Figure .1: Implementation of `clear` and `isEmpty` functions

Chapter 7

Lock-free Object-Sharing for Shared Memory Real-time Multiprocessors

This paper is in submission. It is also available as the Technical Report 2003-03 of Department of Computing Science, Chalmers University of Technology.

Lock-free Object-Sharing for Shared Memory Real-time Multiprocessors ⋆

Philippas Tsigas and Yi Zhang

Department of Computing Science,
Chalmers University of Technology,
SE-412 60, Gothenburg, Sweden

Abstract

Operations on lock-free shared data objects suffer from potential unbounded execution time particularly in multiprocessor systems. To apply lock-free technology in real-time systems, the problem of unbounded execution time must be solved. In this paper, we analyze the causes of unbounded execution time of operations on lock-free implementations and propose an inter-process coordination protocol that bounds the execution time of operations on lock-free shared data objects in real-time shared memory multiprocessors. The protocol that we propose works for the lock-free implementations in real-time multiprocessor systems the same way that the multiprocessor priority ceiling protocol (MPCP) works for mutual exclusion in real-time multiprocessors. With the new protocol, the worst case execution time of accessing a lock-free shared data object can be bounded. Scheduling analysis for the proposed protocol is presented and the advantages of applying such a protocol in real-time multiprocessors is discussed.

1 Introduction

Tasks in real-time multiprocessors are required to provide correct computation under time critical constrains. To satisfy time constrains, the worst case execution time of tasks must be bounded and predictable. With a simplified assumption that all tasks are independent of each other, most real-time scheduling algorithms guarantee that all tasks meet deadlines using the information of worst case behavior. However, in reality, not all real-time tasks are independent of each other; some of them need to

⋆ Partially supported by ARTES, a national Swedish strategic research initiative in Real-Time Systems and TFR the Swedish Research Council for Engineering Sciences.

communicate with each other in order to coordinate. In real-time shared memory multiprocessors, shared objects are the communication channels for tasks. Shared objects must be protected with synchronization protocols to guarantee the consistency of the objects. Synchronization protocols for shared objects can introduce unbounded or unpredictable worst case behavior into real-time systems. The design of synchronization protocols for shared object with predictable behavior in real-time systems has attracted the attention of many researchers.

In shared memory real-time multiprocessors, there are two ways to protect shared objects: mutual exclusion and non-blocking synchronization. When mutual exclusion is used to protect shared data objects in real-time systems without any other effort, tasks will have unpredictable worst case behavior because of priority inversion. To minimize the effect of priority inversion, Rajkumar, Sha, and Lehoczky have proposed the priority inheritance protocol (PIP) and priority ceiling protocol (PCP) [10, 12] for uniprocessor real-time systems with rate monotonic (RM) scheduling. Chen and Lin [8] extended the PCP protocol to work together with the earliest deadline first (EDF) scheduling algorithm. Baker [6] describes a stack-based resource allocation policy for real-time systems scheduled with RM or EDF. For real-time multiprocessor systems, Rajkumar et al. have proposed the distributed priority ceiling protocol (DPCP) [11] (for message passing systems) and the multiprocessor priority ceiling protocol (MPCP) [9] (for shared memory systems).

Compared to mutual exclusion, non-blocking synchronization has certain advantages that make it attractive for use in real-time systems. Non-blocking synchronization includes lock-free and wait-free synchronization. Objects with lock-free synchronization usually are implemented with retry loops [5] and guarantee the progress of the whole system. However, the retry loops in lock-free shared objects are potentially unbounded; unbounded retry loops make individual processes to experience unbounded execution times. Objects with wait-free synchronization are often implemented with a helping scheme to bound execution time and ensure for each operation individual progress. In [3, 5], Anderson, Ramamurthy and Jeffay show how to apply lock-free synchronization into hard real-time uniprocessor systems. Anderson et al. in [1, 4] have also shown how to apply wait-free synchronization into real-time uniprocessors and multiprocessors. In their approach, when interferences between operations happen, tasks help each other in certain ways to make sure that every one makes some progress. Later, they extend their results to work in quantum-based real-time systems [2].

In this paper, we address the problem of applying lock-free synchronization in real-time shared memory multiprocessor systems. First, we look into the causes of unbounded worst case execution behavior when accessing lock-free shared objects in such systems. Then, in order to make lock-free synchronization applicable to real-time shared memory multiprocessors, we propose inter-process an protocol that bounds the

worst case execution time of an operation accessing a lock-free shared object in such systems. The proposed protocol works for lock-free synchronization the same way that MPCP or DPCP work for mutual exclusion. With the help of such protocols, the worst case execution time of accessing shared objects is bounded and tasks which communicate through shared objects can be scheduled as independent tasks. Our results complement the results in [3, 5] that were proposed for uniprocessor systems. In [3, 5], the interference between tasks, running on the *same processor*, accessing lock-free shared data object was analyzed. Based on this analysis, a way was shown to bound the number of retries when accessing lock-free shared objects under certain conditions with RM and EDF scheduling for uniprocessors.

The rest of the paper is organized as follows. In Section 2, we present the problem addressed in this paper. We describe the real-time shared memory multiprocessor system model, the lock-free synchronization scheme and we analyze the causes of unbounded worst case execution time when accessing lock-free shared objects in multiprocessor systems. In section 3, we present our protocols for applying lock-free synchronization in real-time multiprocessors. We conclude in section 4.

2 The Problem

Resource sharing is an important problem in both uniprocessor and multiprocessor real-time systems. Tasks in real-time systems are required to deliver results within a deadline. To finish their computation, tasks may have to access shared resources. Synchronization mechanisms are used to guarantee the consistency of shared resources. Furthermore, because of real-time constrains of the system, the access time of a shared resource must also be bounded.

Non-blocking synchronization is a way to coordinate accessing of shared resources. It guarantees the consistency of shared data objects without using mutual exclusion. There are two types of non-blocking synchronization: wait-free and lock-free. Wait-free synchronization offers bounded accessing time for shared resources under any system conditions. This feature makes it very attractive to real-time systems. However wait-free synchronization typically is algorithmically more complex and has larger memory consumption than the respective lock-free synchronization. The obstacle of applying lock-free synchronization in real-time systems is that lock-free synchronization has unbounded resource accessing time in the worst case, it guarantees only progress of the whole system.

In the following subsection, first we will give a brief description of the system model and the notation used in this paper. Then we will introduce the general structure of lock-free synchronization and analyze the cause of the unbounded accessing time of

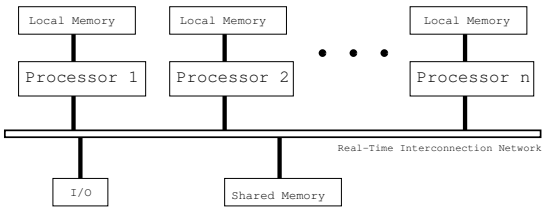


Figure 1: Shared Memory Multiprocessor System Structure

shared resources when using lock-free synchronizations.

2.1 Shared Memory Multiprocessor Real-time System

A generic shared memory multiprocessor system is depicted in Figure 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. In shared memory multiprocessor systems, communication architectures provide implicit and transparent communication as a result of conventional memory accessing primitives, such as `Read/Write`, `Test_And_Set` and `Compare_And_Swap`. A set of cooperating tasks (processes) with timing constraints are running on the system, performing their respective operations. Each task is sequentially executed on one of the processors, while each processor can be shared between many tasks (multiprocessing). The cooperating tasks which possibly run on different processes use shared data objects built in the shared memory to coordinate and communicate. Every task has a maximum computing time and has to be completed by a time specified by a deadline.

2.2 Lock-free Synchronization

An implementation of a shared data object is called lock-free if it first supports concurrency: several processes can perform operations on the shared data object concurrently; and moreover if it guarantees that at any point in time some of the non-fault concurrent processes will complete their operations on the object in a bounded time regardless of the speed or status of the other processes. This requirement rules out the use of locks: if a process crashes while holding a lock, all other processes waiting for the lock can make no progress. Operations on lock-free shared objects are usually implemented as a union of "retry loops". For detailed description of such lock-free shared data objects, the readers are referred to [5]. In this article, we use the term "lock-free" in the same way as it is used in [5]. Lock-free synchronization has some properties that fit the needs of real-time system: it does not suffer from priority inversion and is free from deadlock.

```
1 void access_object(ptr)
2 {
3   do
4   {
5         old = READ(ptr);
6         new = Computing(old);
7         result = CAS(ptr, old, new);
8   } while(result != SUCCESS);
9 }
```

Figure 2: A general scheme of accessing a lock-free data structure

A general retry loop that is used for accessing a lock-free shared data object is depicted in Figure 2. When a process wants to access a lock-free shared data object, it finds a pointer which points to the interesting part of the shared data object for this part of the operation. Then it reads out the content of the pointer and performs computation on the content of the shared data object and generates a new content. Finally, it uses an atomic primitive, `Compare_And_Swap (CAS)` in this example, to update the pointer with the new value in one atomic step. The atomic primitive `CAS` can be replaced by any other universal atomic primitive like the primitive-pair `Load_Linked/Store_Conditioned` or the primitive CAS2 [5]. If only one process accesses the part of the shared data object, it will succeed to perform the `CAS`. When several processes want to modify the same part of the data structure at the same time, the atomic primitive `CAS` will guarantee that only one of them will succeed; the other processes will have to retry the whole loop again and again until they succeed.

Terminology and Notation: In this paper, we assume that all tasks are periodic, can not migrate between processors and are scheduled with priority on uniprocessors. The deadline of each task is the end of the corresponding period of the task. Below we describe the notations used in the rest of this paper.

- N is the number of tasks in the system.
- P is the number of processors of the system. Processors are indexed from $[0, P-1]$.
- T_i is the ith task.
- $P(T_i)$ is the period of the task T_i.
- s is the worst-case execution time required for one lock-free computing loop iteration (statements 5 to 7), on a lock-free shared object regardless of success of the operation. For simplicity, we assume it is the same for all objects.
- $t(T_i, j)$ is the absolute time that statement j of an operation on a lock-free shared object by task T_i takes effect.
- $W_o(i)$ is the worst case execution time outside the lock-free operations for the ith task.
- $OP(j)$ is the set of all processors except from processor j.
- $TS(j, x)$ is the set of tasks which are running on processor j and perform operations on the lock-free shared object x.

- R_i^a is the retry-level of task T_i for shared object a. The definition and usage of retry-level will be described later.

If two tasks, T_i and T_j, execute the lock-free retry loop when they access the same object and T_i succeeds to update the shared object and T_j fails to update it, we say that task T_j is **interfered** by T_i or T_i causes **interference** on T_j. A task T_i can cause interference on many tasks. Operations on lock-free shared objects are concurrent operations and an operation can be interfered by other operations at any time and at any rate. Such interferences can cause any operation on a lock-free shared object to take arbitrarily long time to finish.

2.3 Causes of Interference

If we consider the relative location of tasks involved, there are two kinds of interferences that can take place:

Preemption Interference: Preemption interference takes place when a task T_i that is accessing a lock-free shared data object, is preempted by another task T_j that also causes **interference** on T_i. An example scenario is shown in Figure 3, T_i and T_j run on the same processor: Processor 0.

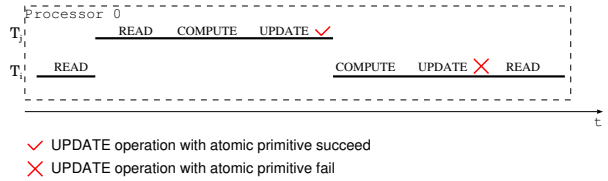


Figure 3: Failure of atomic primitive caused by preemption

Overlap Interference: Overlap Interference is the interference that takes place between two tasks T_i and T_j when task T_i and Task T_j run on two different processors. An example scenario is shown in Figure 4, T_i runs on processor 0 and T_j runs on processor 1.

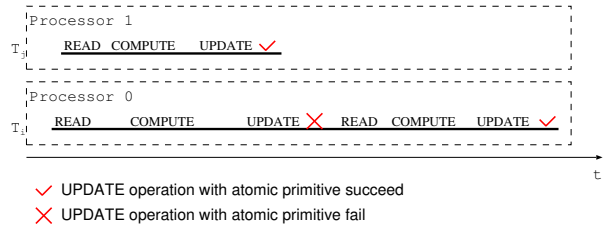


Figure 4: Failure of atomic primitive caused by overlap

In uniprocessor systems, overlap interference does not exist as all tasks are running on one processor. In such a system, only preemption will cause a task to do a retry loop

when accessing a lock-free shared data object. Anderson et al. [3, 5] have analyzed this case. In their work, they show that since all tasks of a real-time system are scheduled under a certain scheduling policy, the maximum number of preemptions of an operation on a lock-free shared object within a time period can be determined with the knowledge of scheduler. Furthermore, they induce a formula for calculating the worst case execution time of lock-free computing under different scheduling policies for real-time uniprocessor systems.

In multiprocessor systems, both preemption and overlapping can cause task interferences and force a task to execute a retry loop when it accesses a lock-free shared data object. Furthermore, these two causes of interference cross-effect each other. If an operation on a lock-free shared object by a task is preempted and interfered, the operation will be forced to execute a retry loop. Consequently the operation will take more time and thus the probability to be preempted and overlapped by other tasks again will increase. The same argument also holds when an operation on a lock-free shared object by a task is forced to do a retry loop because of overlap interference. Things are even worse when looking at overlap interferences since they have the potential to happen at any point in time and at any rate in a general multiprocessor system. Therefore, if we want to use lock-free shared data objects in real-time multiprocessors, we must bound the number of overlap interferences that can happen to an operation.

2.4 Observations

In this paper, we focus on analyzing first the overlap interference of lock-free shared objects in real-time multiprocessors. We assume tasks will not migrate among processors.

The worst case frequency of overlap interferences on different tasks running on the same processors is the same. When a task accesses a lock-free shared object, the number of overlap interference in one time unit is determined by the number of the other tasks which also want to access the same lock-free object and run on different processors. That is the reason that all tasks on the same processor will have the same worst case overlap interferences.

Overlapping interference has more impact on scheduling for tasks (scheduled on the same processor) with small periods than on tasks with long periods. When computing the worst case execution time of a task, we have to assume that the task we consider is the one that failed to access the object in every overlap interference. The extra accessing time of a lock-free object introduced because of overlap interference is constant for all tasks on the same processor. The smaller the period a task has, the larger the processor utilization the extra accessing time will introduce. As the worst

case frequency of overlap interference for one processor increases, at certain point the extra processor utilization introduced by overlap interference will make the task no schedulable.

If using the worst case frequency of overlap interference for scheduling, many extra processor time need to be reserved. Considering three tasks on three processors. All tasks have the same release time and access the same lock-free shared objects. In the worst case, all of them should finish access the lock-free objects in $3s$. As every task has a chance to be the last one accessing the object, we have to reserve $3s$ for each tasks. It means we reserve $9s$ processor time in total for accessing lock-free object.

From the above observations, we concluded that in order, to be able to use lock-free shared objects in real-time multiprocessors, the overlap interference must respect the priority nature already existing in real-time systems; and use it to reduce the time reserved for lock-free accessing. We will address these problem in the next section.

3 Retry-level Based Protocol for Lock-free Computing

In this section, we will first show how to bound the number of overlap interferences that can happen to an operation. Then we will propose schemes for using lock-free shared objects in real-time multiprocessors and give a method to calculate the worst case execution time of operations on lock-free shared objects.

3.1 The protocol

As said in the previous section, preemption interference and overlap interference may cross-effect each other. To analyze the behavior of overlapping and present our protocol that bounds the retry-time caused by overlapping, we will first decouple the two causes of interference. To simplify the presentation, non-preemptive scheduling is going to be assumed for the beginning, later we will show how to extend our results for preemptive scheduling. We assume tasks are periodic and fixed on processors. When using non-preemptive scheduling, an instance of a task which is running on a processor will not be stopped unless it relinquishes the processor itself. Therefore, in the non-preemptive scheduling case the cause of interference for operations on lock-free shared objects is only overlapping.

When a task T_l on processor i executes an operation on a lock-free shared object a, the number of overlap interferences on the task's operation can be determined by

the number of tasks which are running on all other processors of the system and access the same object. Hence, the number of overlap interferences on a task does not depend on its priority but rather depends on the number of processors which access the same object. For a certain period of time, $[0, t]$, the maximum number of overlap interferences, OI, on T_l is calculated by the following formula:

$$OI = \sum_{k \in OP(j)} \sum_{l \in TS(k,a)} \lceil \frac{t}{P(T_l)} \rceil$$

This estimation can be refined further if information concerning the release time of tasks and the start time of lock-free operation of each task are considered known. The estimation above is pessimistic since we have to assume that every overlap interference will cause a retry loop and not desirable for real-time systems. We propose a retry-level based protocol to work on top of a lock-free shared object implementation. In real-time systems, high priorities are given to some tasks to help them finish early. Every task has a priority when it runs on a processor; we introduce also a retry-level for every task accessing a lock-free shared object. A task with high priority experiences small delays; a task with high retry-level will experience less overlap interference when it accesses a lock-free shared object. Every task can have its own priority for the scheduling part and at the same time it has its own retry-level on each lock-free shared object. A requirement of the protocol is that the retry-level for each task on a lock-free shared object must be unique. Our protocol is designed to achieve the following goal: tasks with high retry-level will not be forced to do a retry loop by a task with low retry-level. In this way we will manage to bound the number of interferences on an operation by the number of tasks with higher retry-level on different processors.

A generic operation on a lock-free shared object with the retry-level protocol is described in figure 5. The operation with the retry-level protocol on a lock-free shared object is an extension of the original operation. We assume, like in MPCP [9], that there is a prioritized queue associated with the lock-free shared object. When a task wants to access a lock-free shared object, it first inserts its own retry-level into the priority queue, statement 5. Then, it calls a "delay" function that delays the process from reading for some time, the time that the "delay" function uses is going to be specified later in this section. Before updating the object, it finds the highest retry-level in the priority queue, statement 11. If it is its own retry-level, it will continue the update. Otherwise, it will perform a retry. If the task updates the object successfully, it will remove its own retry-level from the priority queue. The execution time of the delay function, t_{delay}, should be larger than or equal to the execution time of the statements 11, 12 and 13 together. The delay function can be implemented as several "nop" instructions which give the same execution time. The delay time, t_{delay}, is determined by the system architecture and is independent of the implementation of lock-free shared objects.

The proposed protocol has the following properties: if there is no overlapping, a task will access the shared object without any retry; a task with high retry-level will never be interfered by a task with low retry-level; the task with the lowest retry-level will suffer the same amount of the interference as it would have suffered without using our protocol.

```
1 Shared PriorityQueue pq;
2 void access_object(int mrl, ptr)  /* mrl: The retry-level of current tasi
3 {
4   int templevel;
5   enqueue(pq, mrl);
6   delay();
7   do
8   {
9         old = READ(ptr);
10        new = Computing(old);
11        templevel = read_min(pq);
12        if (templevel == mrl) {
13                result = CAS(ptr, new);
14                remove(pq, templevel);
15        }
16        else
17                result = FAIL
18  } while(result != SUCCESS);
19 }
```

Figure 5: A Generic Lock-free operation with the retry-level protocol

3.2 An Example

Before we analyze in detail the properties of our protocol, we will illustrate the retry-level protocol with an example. For the priority and the retry-level, we assume that small numbers mean high priorities or retry-levels. Let us consider a system with 3 processors, P_0, P_1, P_2, and six tasks, $J_1, ..., J_6$, and two global shared objects, SO_1, SO_2. The tasks need to access the two shared objects in the following sequence:

$J_1 : [...SO_1...SO_2]$

$J_2 : [...SO_2...SO_1]$

$J_3 : [...SO_2...SO_1...]$

$J_4 : [...SO_1...]$

$J_5 : [...SO_1...]$

$J_6 : [...SO_1...]$

Tasks	Priority	Retry-Level for SO_1	Retry-Level for SO_2	Run on Processor
J_1	1	1	3	P_0
J_2	2	2	2	P_1
J_3	3	3	1	P_2
J_4	4	4	N/A	P_2
J_5	5	5	N/A	P_0
J_6	6	6	N/A	P_1

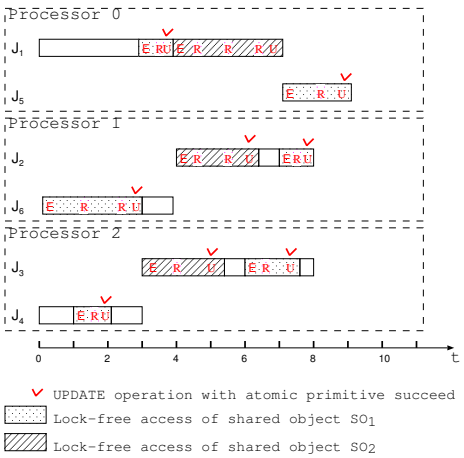


Figure 6: Example of the execution of the retry-level based protocol

The task set and system configure are shown above. "N/A" means a task does not have a retry-level for a shared object because it does not access it. Let us consider the following execution shown in Figure 6.

- At $t = 0$: tasks J_1, J_6, J_4 begin to execute on processors P_0, P_1, P_2, respectively. Task J_6 begins to access the global shared object SO_1.
- At $t = 1$: task J_4 begins to access the global shared object SO_1; at that moment task J_4 executes statement 5 then J_6 executes statement 11; J_6 finds out the intention of J_4 and gives up its updating operation and does a retry loop.
- At $t = 2$, task J_4 read the priority queue and finds its own retry-level and updates the object SO_1 successfully.
- At $t = 3$: task J_4 on P_2 finishes and task J_3 is released and begins to access shared object SO_2. J_6 on P_1 reads the priority queue and finds its own retry-level. Before J_6 starts to update object SO_1, task J_1 starts to access the same shared object

and inserts its retry-level into the priority queue. J_1 will be delayed for some time before it reads the shared object, delay that will allow J_6 to update the object SO_1 before J_1 reads it. J_6 finishes its operation on SO_1.

- At $t = 4$: task J_1 updates SO_1 successfully. J_1 then starts its operation on SO_2. Task J_6 on P_1 has finished and J_2 is scheduled begins access to SO_2. Both J_1 and J_2 insert their retry-levels into the priority queue for SO_2.
- At $t = 5$: tasks J_1, J_2, J_3 want to update shared object SO_2. When they check the priority queue, J_3 will go ahead and J_1 and J_2 will do a retry loop. After accessing SO_2, J_3 continues.
- At $t = 6$: tasks J_1 and J_2 read the priority queue of SO_2 and they find out that the highest retry-level is 2 (J_2's retry-level). J_1 skips its updating operation and J_2 updates SO_2 successfully. Task J_3 begins to access SO_1.
- At $t = 7$: task J_1 updates SO_2 successfully and exits. Task J_5 begins to execute and accesses the shared object SO_1. J_3 wants to update SO_1 and reads the priority queue. J_2 also accesses the shared object SO_1. When J_3 checks the priority queue, it finds its own retry-level. J_2 will wait some time before reading the shared object.
- At $t = 8$: task J_3 finishes its operation. Tasks J_2 and J_5 both want to update SO_1, J_2 will go ahead and J_5 will perform a retry loop. J_2 finishes after updating SO_1.
- At $t = 9$: task J_5 updates SO_1 and then releases the processor.

3.3 Schedulability Analysis

In this section, we analyze the worst case behavior of the protocol proposed in the previous section and also remove the non-preemptive restriction that we introduced for simplicity reasons before. First we present some lemmas which describe the properties of the protocol.

Lemma 1 *In real-time shared memory multiprocessor systems, where non-preemptive scheduling is applied on each processor and all tasks use the retry-level based protocol to access lock-free shared objects, a task with retry-level i will not be interfered by tasks with retry-level less than i.*

Proof: The `CAS` atomic primitive will succeed if and only if no other task updated the value of the pointer after it was read [1]. If a task T_i fails to modify the shared data object when executing the `CAS` operation, it means that T_i has suffer an overlap interference and task T_j has modified the object after T_i read it and before it executed the `CAS` operation. This means that the statement 9 of task T_i took effect before statement 13 of T_j effect. If task T_j has a lower retry-level than T_i, then the execution of statement 5 of T_i has happened after the execution of statement 11 of T_j. Otherwise, task T_j would

[1] We will not consider the ABA problem in this paper. We can use one of the standard techniques used in the literature to overcome it [13].

have observed the intention of T_i and would have not modified the object. By summing these up, we get the following relations: $t(T_j, 11) < t(T_i, 5) < t(T_i, 9) < t(T_j, 13)$. From these relations, we have that $(t(T_j, 13) - t(T_j, 11)) > (t(T_i, 9) - t(T_i, 5) > t_{delay}$, which contradicts the requirement that $t_{delay} > (t(T_j, 13) - t(T_j, 11))$. □

The above lemma implies that the task with the highest retry-level will never suffer from overlap interferences; for the task with the lowest retry-level, every time it overlaps with another task, it will be forced to do a retry loop. If the retry-level based protocol is not applied, every time a task overlaps with any other task, it is possible for that task to have to perform a retry loop. Therefore, in the worst case, tasks that use the retry-level based protocol will not behave worse than tasks without retry-level based protocol. The above sketches the proof of the following lemma.

Lemma 2 *In real-time shared memory multiprocessors, where non-preemptive scheduling is applied on each processor and all tasks use the retry-level based protocol to access lock-free shared objects, the number of overlap interferences in the worst case on any task is no more than the number of overlap interferences for the same task if it is running in a system without retry-level base protocol.*

The above two lemmas give us the number of overlap interferences (in the worst case) for the tasks with the highest and lowest retry-levels. The number of overlap interferences in the worst case for other tasks is determined by the following lemma.

Lemma 3 *In real-time shared memory multiprocessors, where non-preemptive scheduling is applied on each processor and all tasks use the retry-level based protocol to access lock-free shared objects, the number of overlap interferences (in the worst case) on the ith task running on processor j with retry-level $R(T_i, a)$ while accessing shared object a, is denoted as B_i, and is the smallest natural number that satisfies the following formula:*

$$B_i = \sum_{k \in OP(j)} \sum_{l \in TS'(k)} \lceil \frac{(B_i + 1) * s}{P(T_l)} \rceil$$

where $TS'(k) = \{t | t \in TS(k, a) \bigvee R_t^a < R_i^a\}$

Proof: From lemma 1, we know that all tasks with lower retry-level than the retry level of T_i cannot introduce any retry loop to T_i. The formula above represents the maximum retry time caused by overlap interferences from all tasks which are not running on the same processor and whose retry-levels are higher than i between the time T_i starts to access the object and the time it can start its loop without failing. If no such t exists, then the maximum possible blocking time is infinite for this task.

□

The above lemma computes the worst case number of overlap interferences for operations on a lock-free shared object for all tasks in a real-time multiprocessors with non-preemptive scheduling. The following theorem comes also from the previous lemma.

Theorem 1 *In real-time shared memory multiprocessors, where non-preemptive scheduling is applied on each processor and all tasks use the retry-level based protocol to access lock-free shared objects, the worst case execution time of the ith task is*

$$W_{cet} = W_o(i) + (B_i + 1) * s$$

where B_i is given by Lemma 3.

For simplicity reasons until now we considered non-preemptive scheduling policy on each processor. Now, we will show how to extend our result to also work with preemptive scheduling polices. When a preemptive scheduling policy is used, the number of retries depends not only on the overlapping with other tasks running on other processors but also on preemption because of high priority tasks that are running on the same processor. Therefore, we suggest to apply the proposed protocol with other protocols that have been proposed for uniprocessor systems to bound the execution time of lock-free computing. We show how to incorporate the protocol presented here with the immediate priority ceiling protocol (ICPP)[7] and we also compute the worst case execution time of an operation on a lock-free shared object when using such a scheme.

By using ICPP [7] we avoid preempting a task accessing a global lock-free shared data object. The priority ceiling, P_G, of a lock-free shared data object is the highest priority assigned to any task in the entire system. When a task wants to operate on a lock-free shared object, first it uses the ICPP protocol to promote the priority of the task to P_G. Then, the task will use the retry-level based protocol to access the lock-free shared data object. When it finishes its operation on the object, the priority of the task will be changed back to its own priority. The worst case time of accessing a lock-free shared object will be the same as the one computed with non-preemptive scheduling. This give us the following theorem.

Theorem 2 *In real-time shared memory multiprocessors, where preemptive scheduling is applied on each processor, and all tasks use the combination of the ICPP protocol on uniprocessor level, and the retry-level based protocol to access a lock-free object a, then the number of interferences on an operation while accessing a lock-free shared object a for task T_i, is denoted as B_i, and is the smallest natural number that satisfies the following formula:*

$$B_i = \sum_{k \in OP(j)} \sum_{l \in TS'(k)} \lceil \frac{(B_i + 1) * s}{P(T_l)} \rceil$$

where $TS'(k) = \{t | t \in TS(k,a) \bigvee R_t^a < R_i^a\}$

4 Discussion and Conclusion

One advantage of lock-free shared object over mutual exclusion is that faults or failure on one task will not effect the progress of other tasks. This property is weakened by the proposed protocol: the failure of a task with low retry-level will never effect a task with high retry-level. We think the weakened property will not have much impact in practice. If a real-time system is mission-critical, the system must impose some fault-tolerance technology. Otherwise, the faults will render the whole system useless.

The assignment of retry-levels on tasks is a decision problem. The retry-levels of tasks can be assigned the same order as priorities of tasks. In this way, the system will delivery the best fault tolerance in respect to the shared object accessing part. They can also be assign in other ways which can help people to schedule tasks. How to assign them is up to the designer of the real-time system.

Our contributions in this paper are the following: we analyze the cause of interference for lock-free shared object in real-time shared memory multiprocessors; we propose methods to bound the number of interferences for lock-free shared object in real-time shared memory multiprocessors; we analyze the worst case behaviors of the proposed methods. To the best of our knowledge, we are the first to show how to apply lock-free technology in real-time shared memory multiprocessors.

For systems with non-preemptive scheduling on each processor, the new protocols can perform in user level entirely. For systems with preemptive scheduling, the new protocols have the requirement of priority promotion as MPCP does, but without having to perform the task of waking up and transferring control of shared resources to tasks on different processors, which is required by MPCP. In the worst case, the task with the highest priority that uses MPCP will be blocked for one critical section; the task with the highest retry-level will not be interferenced by any other tasks in our protocol. Comparing it with MPCP, the methods in this paper have also better behavior in case of faults; the content of lock-free shared objects will be consistent all the time and the failure of a task with low retry-level will never effect a task with high retry-level.

In future work, we will compare the protocol with MPCP and wait-free schemes in schedulability and implementation requirements with experiments. Implementation of the new protocol in real-time kernel and profiling its performance are also interested by us.

References

[1] J. Anderson, R. Jain, and S. Ramamurthy. Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 111–122. IEEE, Dec. 1997.

[2] J. Anderson, R. Jain, and S. Ramamurthy. Efficient object sharing in quantum-based real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium(RTSS '98)*, pages 346–355. IEEE, Dec. 1998.

[3] J. Anderson and S. Ramamurthy. Using lock-free objects in hard real-time applications. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*, pages 272–272. ACM, Aug. 1995.

[4] J. Anderson, S. Ramamurthy, and R. Jain. Implementing wait-free objects on priority-based systems. In *Proceedings of the 16th annual ACM symposium on Principles of distributed computing*, pages 229–238. ACM Press, 1997.

[5] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems*, 15(1):134–165, Feb. 1997.

[6] T. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, 3(1), Mar. 1991.

[7] A. Burns and A. J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Addison-Wesley, Third edition, 2001.

[8] M.-I. Chen and K.-J. Lin. Dynamic priority ceilings: a concurrency control protocol for real-time systems. *Real-Time Systems*, 2(4):325–346, 1990.

[9] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 116–123, Paris (France), May–June 1990. IEEE, IEEE Computer Society Press.

[10] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[11] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the 1988 IEEE Real-Time Systems Symposium*, pages 259–269, 1988.

[12] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept. 1990.

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

Chapter 8

Conclusions

This thesis has performed a study on applying non-blocking synchronization in shared memory multiprocessors. It first explores the performance impact of non-blocking synchronization in shared memory multiprocessors. Then it demonstrates the advantages of non-blocking synchronization on real-time shared memory systems.

The performance advantage of non-blocking synchronization over mutual exclusion in shared memory multiprocessors is intuitive and has been advocated by the theoretical community for a long time. But non-blocking synchronization is still not popular among programs for shared memory multiprocessors. One of the reasons is that many non-blocking synchronization mechanisms are quite complex. Besides, there are no large application examples with non-blocking synchronization. The aim of this thesis is to address these issues. We developed non-blocking data structures which can be used as basic building blocks for applications. We worked on large benchmark applications, transfered mutual exclusion in applications into non-blocking synchronization, and showed the cost and performance impacts of such modifications. We developed non-blocking algorithms for shared memory multiprocessors. We hope that our effort can help application designers and programmers understand the performance advantage of non-blocking synchronization, accept ideas of non-blocking synchronization, and use them in their programs.

The advantages of non-blocking synchronization in real-time shared memory systems have also been known for a long time. Compared with mutual exclusion, non-blocking synchronization is freed from priority inversion problem and deadlocks. There are two ways to apply non-blocking synchronization in real-time systems. One of them is to design a general scheme which can allow non-blocking algorithms to be used in real-time systems and produce predictable behavior. The other one is to design special non-blocking synchronization algorithms with information provided by

the real-time systems themselves, such as time information of task and information of scheduler, etc. In this thesis, we investigated both ways: two non-blocking data structures are proposed and a general way of applying lock-free synchronization into real-time systems is investigated.

From our experience of working with non-blocking synchronization, we learn that:

- Non-blocking synchronization is more complex algorithmically than mutual exclusion. The design and understanding of algorithms using non-blocking synchronization require more time than those of mutual exclusion. This complexity is the main obstacle for programmers to accept non-blocking synchronization.

- Data-structure-specific non-blocking algorithms are desired. These algorithms generally outperform lock-based implementations and universal non-blocking implementations. At the same time, the implementations of these data structures can be provided to programmers as basic building blocks for their applications. This can lower the entry cost of using non-blocking synchronization.

- General memory management schemes for non-blocking synchronization are appreciated. In our research of non-blocking synchronization, we incorporate the idea of memory management into our non-blocking algorithms in several occasions. Such incorporation helps the designing and the efficiency of our algorithms. We believe that general memory management schemes for non-blocking synchronization will be crucial for non-blocking synchronization.