THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Concurrent Algorithms and Data Structures for Many-Core Processors

DANIEL CEDERMAN

Division of Network and Systems
Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2011

Concurrent Algorithms and Data Structures for Many-Core Processors

Daniel Cederman

ISBN 978-91-7385-503-7

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 3184
ISSN 0346-718X

Technical report 76D
Department of Computer Science and Engineering
Distributed Computing and Systems

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

Author e-mail: cederman@chalmers.se

Printed by Chalmers Reproservice
Göteborg, Sweden 2011

Concurrent Algorithms and Data Structures for Many-Core Processors

Daniel Cederman

Division of Network and Systems, Chalmers University of Technology

ABSTRACT

The convergence of highly parallel many-core graphics processors with conventional multi-core processors is becoming a reality. To allow algorithms and data structures to scale efficiently on these new platforms, several important factors needs to be considered.

(i) The algorithmic design needs to utilize the inherent parallelism of the problem at hand. Sorting, which is one of the classic computing components in computer science, has a high degree of inherent parallelism. In this thesis we present the first efficient design of Quicksort for graphics processors and show that it performs well in comparison with other available sorting methods.

(ii) The work needs to be distributed efficiently across the available processing units. We present an evaluation of a set of dynamic load balancing schemes for graphics processors, comparing blocking methods with non-blocking.

(iii) The required synchronization needs to be efficient, composable and easy to use. We present a methodology to easily compose the two most common operations provided by a data structure – the insertion and deletion of elements. By exploiting a common construction found in most non-blocking data structures, we created a move operation that can atomically move elements between different types of non-blocking data structures, without requiring a specific design for each coupling. We also present, to the best of our knowledge, the first application of software transactional memory to graphics processors. Two different STM designs, one blocking and one obstruction-free, were evaluated on the task of implementing different types of common concurrent data structures on a graphics processor.

Keywords: parallel, lock-free, graphics processors, multi-core, sorting, load balancing, composition, software transactional memory

Preface

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

- **Daniel Cederman**, Philippas Tsigas, "GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors," in *the ACM Journal of Experimental Algorithmics (JEA)*, Vol. 14, No. 4, ACM press 2009.
- **Daniel Cederman**, Philippas Tsigas, "On Dynamic Load Balancing on Graphics Processors," in *the Proceedings of the 11th Graphics Hardware (GH 2008)*, pages 57 - 64, ACM press 2008.
- **Daniel Cederman**, Philippas Tsigas, "Supporting Lock-Free Composition of Concurrent Data Objects," in *the Proceedings of the 7th ACM conference on Computing Frontiers (CF 10)*, pages: 53-62, ACM 2010.
- **Daniel Cederman**, Philippas Tsigas, Muhammad Tayyab Chaudhry "Towards a Software Transactional Memory for Graphics Processors," in *the Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization 2010*, pages 121-129, Eurographics Association 2010.

Acknowledgments

I would first like to thank my advisor, Philippas Tsigas, for offering me the position as one of his PhD students. This field of research that I have been invited to is a truly exciting one, especially at this point in time. Philippas extensive knowledge and experience in the field and, most important in my eyes, his ability to guide and encourage, has been crucial for the completion of this thesis. Seldom have I left a meeting with him without feeling inspired. I am honored to have worked with him and I hope to continue working with him in the future.

I would also like to thank my good friend and fellow master thesis worker Tord. Had he not suggested Philippas as the supervisor for our master thesis, I would probably never have considered a career in research. Thank you Tord!

I am honored to have Maged Michael, whose work I've cited countless times, as my opponent, and for having Christoph Kessler, Ulf Assarsson, Björn Lisper, Sally McKee and my examiner Aarne Ranta in the grading committee.

I am very grateful to Microsoft Research for their European PhD Scholarship Programme, which financed my position for the first three years. Thank you Fabien! I am also excited about the PEPPHER project, which has financed me these last few months, and which I am looking forward to continue working with.

I want to give a special thanks to my office mate Andreas for all the fun and interesting discussions over the years, and to Georgios, for putting up with said discussions for as long as he did. You two, and the rest of the Distributed Computing and Systems group – Elad, Farnaz, Marina, Nhan, Philippas and

Zhang – have made this a truly great place to work and I am very proud to be a member of this group. My thanks also goes out to the previous members of the group – Anders, Boris, Håkan, Niklas, Phuong and Yi – whom I've been lucky to work and discuss with in varying degrees. I would also like to thank the three master students that I have had the joy of working with – Rafia, Tayyab and Emad. I wish you all the best!

I would also like to show my gratitude to my current and former office neighbors – Dennis, Erland, Magnus, Phu, Pierre, Tomas, Vilhelm and Wolfgang – for providing a friendly and creative working environment. A special thanks to the Computer Graphics group for discussions and sharing of hardware. From the administrative and technical side of the department, I would like to especially thank Eva, Eva and Peter, for always being helpful and always providing solutions. I would also like to thank machine number 659943, for its selfless dispensing of nourishing coffee.

Lastly, I would like to thank my family, friends and relatives for all their support, which I have always been able to rely on. It has been *very* much appreciated!

Daniel Cederman
Göteborg, 2011

Contents

1 Introduction

The original Quicksort paper by Tony Hoare in 1962 featured an experimental evaluation of the algorithm where one thousand elements were to be sorted [Hoa62]. Using Quicksort, the feat was accomplished in just above 3 minutes and managed to beat the Merge-Sort competitor by over 2 minutes. Measurements on 1500 elements and more could only be performed with Quicksort, since the memory requirements of Merge-Sort were too large for it to be evaluated. With today's computers it is possible to sort millions of elements in less than a second [CT08a, CT09].

What is interesting is that although the performance has increased so much, very little has been changed in the original algorithm. Instead, most of the increases in speed have come from improvements done on the hardware side. For example, more bits have been added for addressing to allow for bigger

memories; the memory speed has increased, although not enough for it to lose its position as a bottleneck; cache-memories have been introduced and grown larger to mask the memory latency; the processors' clock speeds have been increased tremendously and instructions can be executed speculatively to be used or discarded later on, and many other changes. What all these changes have in common is that the programmers have not been forced to change anything in their programs to take advantage of the improvements. They have all been designed to exploit the normal usage patterns of most programs.

There have of course been hardware architectures that forced the programmer to adapt to it, but for the mainstream processors the manufacturers have tried to avoid that to ensure backward compatibility. Launching a new architecture that requires all programs to be rewritten is a very risky endeavor. So, for a long time, the main variable when comparing the performance between processors has been the clock frequency, since it allowed for an automatic increase in performance without the need for the program to be rewritten.

But recently it has become much harder to increase performance by just increasing the clock frequency. The memory speed has not increased at the same rate and an already high clock frequency has led to much greater power requirements with accompanying excessive heat. It has become increasingly difficult to lead off this energy without using techniques such as water-cooling, which so far seems difficult to bring to the average consumer, although it is already used by enthusiasts.

Instead, the major designers and manufacturers of processors have opted to go for parallelism. The clock frequency is lowered and cache memories are being replaced by more processing cores. The problem now is that programs have to be redesigned to take advantage of the available computing power on these multi-core processors. They need to be divided into tasks that can be performed in parallel on different cores. For these tasks to be able to communicate with each other, they need to have access to safe and efficient synchronization, to avoid problems such as dead-locks, live-locks, and convoying. This can be tricky for all but the most embarrassingly parallel programs. New techniques

are needed to efficiently exploit available parallelism, and lock-free data structures and transactional memory could aid in this task.

In recent years, graphics processors have become a new target in the search for extra computational power. In normal use, graphics processors are often faced with embarrassingly parallel problems, such as performing matrix multiplications to transform 3D vertices into 2D space. This has caused them to target parallelism much earlier than generic processors, which have mostly dealt with sequential programs. In an attempt to broaden the user base for high-end graphics processors beyond just gamers, NVIDIA introduced the CUDA platform to simplify general purpose programming on their graphics processors [NVI10]. It quickly became popular, and is now joined by the OpenCL initiative, which promises to bring support for platform independent parallelism to graphics processors, as well as other parallel hardware platforms [Gro08].

This move simplified the use of the graphics processor as an auxiliary unit for handling parallel code. The combination of highly parallel many-core graphics processors, for easy to parallelize algorithms, and conventional multi-core processors with large caches, for mostly sequential code, could be seen as a view of things to come. Processors will most likely become more and more heterogeneous in the future, with specialized cores for different purposes. For simplicity and efficiency, we will most likely see a unified memory architecture, such as for example the one used in INTEL's Larrabee [SCS+08].

In this thesis we have looked at methods to use the graphics processor efficiently. We have designed and implemented a variant of the quicksort algorithm that worked well for graphics processors. After finding that the hardware load balancing could be improved, we evaluated different software-based load balancing methods on different graphics processors. Non-blocking methods proved to achieve better performance, but have the problem that they are hard to compose. We looked into this problem and provided a methodology for easily composing certain non-blocking functions. We also evaluated more generic schemes for composition of non-blocking functions using software transaction memory.

In the following section we will take a look at some of the problems that one faces when trying to design an algorithm that scales well with the number of processing units. In section 1.2 we give an overview of the architecture used in graphics processors of today and in section 1.3 we give an overview of our work.

1.1 Synchronization

Whenever two entities communicate with each other, they need to utilize some kind of synchronization mechanism to be sure that the exchange of information can take place without problems. While talking at the same time and acting on information mid-sentence are poor ways to communicate between humans, in a computer setting it often has disastrous results.

1.1.1 A Small Example

On a shared memory system the communication between different threads and processes occurs whenever the same memory locations are read and/or written to by more than one entity. Simple operations such as incrementing an integer suddenly become more tricky when several parties try to do it concurrently. As it is the processor that performs the actual incrementing, the current value of the integer variable needs to first be transported to a register on the processor, where it is then incremented by one, and then transported back to the memory. For performance reasons, this is done as three separate operations so that other data might be transported on the data bus while the processor is busy incrementing the value in the register. Two threads that are concurrently trying to increment a counter could then yield the result shown in Figure 1.1.

In the example, the integer i_S is zero from the start and should be two after the two increment operations have been performed. But due to some unlucky ordering of instructions, the end result is one, with the effect that the work of one increment instruction disappeared. Now, while this problem occurred because we had some unlucky ordering of instructions, it could very well be

Thread A	**Thread B**
$i_A \leftarrow i_S$	
	$i_B \leftarrow i_S$
	$i_B \leftarrow i_B + 1$
	$i_S \leftarrow i_B$
$i_A \leftarrow i_A + 1$	
$i_S \leftarrow i_A$	

Before: $i_S = 0$ **After:** $i_S = 1$

Figure 1.1: *Two concurrent increments on a shared variable fails.*

that in 99% of the cases it works without any problem. This often makes these kinds of errors very hard to debug, as they are hard to repeat.

1.1.2 Atomic Primitives

One way to solve the synchronization problem when incrementing an integer is to sacrifice performance and use an atomic increment operation, often called a Fetch-And-Inc (FAI). It locks the memory bus whilst the operation is being performed, so that no other processor can access the memory location concurrently. This is less efficient, but it gives the correct result.

Modern computer architectures often have support for a set of atomic primitives that can be generalized to Fetch-And-Op (FAO), where *op* is usually a math or binary operation such as addition or exclusive or. Other popular atomic primitives include Test-And-Set (TAS), which atomically sets the value of a variable to 1 if not already set. It is often used to implement locks; see section 1.1.5. Compare-And-Swap (CAS) is a powerful primitive that can atomically change the value of a variable to x if the current value is y. It is often used to make sure that one does not overwrite the result from another process. It could, for example, be used to implement a Fetch-And-Inc by reading the variable that is supposed to be incremented, increasing the value by one, and then conditionally writing it back using CAS if it still has the old value. If the value

```
// Fetch-And-Op atomically performs
// an operation on the variable var.
// The operation could be an exclusive-
// or, addition, subtraction or similar.
procedure FAO(var, value)
    x ← var;
    x ← x op value;
    var ← x;

// Test-And-Set atomically sets the
// value of the variable var to 1,
// if it currently is 0.
procedure TAS(var)
    if var = 0 then
        var ← 1;
        return true;
    else
        return false;

// Compare-And-Swap atomically
// changes the value of the variable
// var to new, if it currently
// has the value expected.
procedure CAS(var, expected, new)
    if var = expected then
        var ← new;
        return true;
    else
        return false;

// Used together with SC.
// Marks the variable var
// and returns its value.
procedure LL(var)
    mark var;
    return var;

// Used together with LL.
// Writes the value new
// to var if not changed
// since marked by LL.
procedure SC(var, value)
    if hasMark var then
        var ← new;
        return true;
    else
        return false;
```

Figure 1.2: *Semantics for some commonly available synchronization primitives. All instructions are performed atomically.*

```
procedure ATOMICINC(var)
    repeat
        old ← var
        new ← old + 1
    until CAS(var,old,new)
```

Figure 1.3: *Implementation of an atomic increment function using Compare-And-Swap.*

has changed, the CAS operation will fail and we have to reread the new value and perform the increment again. See Figure 1.3. An overview of some of the commonly available synchronization primitives can be found in Figure 1.2.

Although CAS is powerful, it has one weakness in that it cannot detect if a variable has been changed from its value A, to a value B, and then back again to value A. This is known as the ABA-problem. The problem is often solved by adding a time-stamp to the variable that is increased every time it is changed. This lowers the probability that the ABA-problem will occur, but it also lowers the amount of information that can be stored in the variable. There is therefore a tradeoff between the number of useful bits and the probability that there will be an error.

The problem mostly appears when a memory object that has been freed is reclaimed, and used again, while a process is still holding a reference to the original object. A way to solve this problem, without using time-stamps, is through the use of reference counting or hazard pointers. By counting the number of processes that holds a reference to an object, it is possible to defer the reclamation until the reference count reaches zero [Val95, MS95]. Hazard pointers in turn can be used to mark memory objects that are not to be reclaimed [Mic04a]. Each process controls its own set of hazard pointers and uses them to mark objects that it is currently accessing. Hazard pointers are most useful when the number of memory references are bounded. It is also possible to combine the two methods, as done in the lock-free memory reclamation scheme by Gidenstam et al. [GPST09].

Consensus Number	Object
1	read/write registers
2	test-and-set, swap, fetch-and-add
⋮	⋮
$2n-2$	n-register assignment
⋮	⋮
∞	move, swap, compare-and-swap

Figure 1.4: *Consensus number for some common primitives. Based on a table by Herlihy [Her91].*

Another solution to the ABA-problem is to use the Load-Linked (LL)/Store-Conditionally (SC) pair of atomic primitives, if available. By reading a variable using LL, one can use SC to assure that a new value is written to the variable only if no other process has already changed it since the last call to LL. The semantics can be seen in Figure 1.2. A drawback of using LL/SC is that in many implementations the SC might fail even though the value has not been changed. This can happen when the check is done on the cache-line and not on the actual word, causing the SC to fail due to a write in the vicinity of the word by another process. While the LL/SC pair is not commonly available, in contrast to CAS, it is possible to implement its functionality in software using CAS [Mic04b].

The power of different synchronization primitives is often measured by its consensus number [Her91]. The consensus number tells how many processes can agree on a value, by using the primitive, in a finite number of their own steps. Normal read and write registers have a consensus number of one, which means that they cannot solely be used to achieve any non-blocking synchronization. Compare-And-Swap, on the other hand, has an infinite consensus number. Between these two extremes we have, for example, n-register assignment and coalesced memory accesses [HTA10]. Figure 1.4 shows the consensus number for some common primitives.

Initial values: $x \leftarrow 0, y \leftarrow 0$

Thread A	**Thread B**
$x \leftarrow 1$	$y \leftarrow 1$
$a \leftarrow y$	$b \leftarrow x$

Thread A (Reordered)	**Thread B (Reordered)**
$a \leftarrow y$	$b \leftarrow x$
$x \leftarrow 1$	$y \leftarrow 1$

Figure 1.5: *Reordering of memory reads and writes might cause unwanted behavior. By just looking at the program, it seems there is no way that both a and b can be zero after the execution of both threads. But the compiler or processor might find it more efficient to do the read before the write, a change that does not change the local semantics of the thread. Now it is possible for both a and b to be zero. To avoid this, one needs to insert memory barriers between the two instructions to prohibit the compiler and processor from reordering the instructions.*

1.1.3 Memory Consistency

On a uniprocessor it is possible to keep a strict ordering of memory accesses, that is, reads and writes occurring in the exact order they have been issued as measured by a common clock. However, this is not practical to achieve on a multiprocessor system, as it would require too much synchronization. A more practical consistency model is sequential consistency, as proposed by Lamport [Lam79]. There the ordering of instructions from a single processor is exactly as specified by its program, but the global ordering can tolerate any ordering of instructions from different processors. An even weaker consistency model, relaxed consistency, allows read and write operations that are not dependent upon each other to occur in an order that differs from the one specified by the program. For local operations this is no problem, but when communicating between different processes, the exact ordering might be critical, as shown by the example in Figure 1.5. Processors with relaxed consistency often have memory barriers that can be used in these settings, that guarantee that an op-

Blocking Uses mutual exclusion to only allow one process at a time to access the object.

Lock-Free Multiple processes can access the object concurrently. At least one operation in a set of concurrent operations finishes in a finite number of its own steps.

Wait-Free Multiple processes can access the object concurrently. Every operation finishes in a finite number of its own steps.

Figure 1.6: *Progress guarantees for shared data objects.*

eration have been performed before continuing past the barrier. These barrier operations are often expensive, and should be used sparingly.

1.1.4 Shared Data Objects

The methods used for synchronizing memory accesses to shared data objects can be divided into three categories, *blocking*, *lock-free*, and *wait-free*. Blocking methods use mutual exclusion primitives, such as locks, to only allow one process at a time to access the object. This is a pessimistic conflict control that assumes conflicts even when there are none.

Lock-free methods on the other hand employ an optimistic conflict control approach, allowing several processes to access the shared data object at the same time and suffering delays because of retries only when there is an actual conflict. This feature allows non-blocking algorithms to scale much better when the number of processes and contention increases.

Wait-free methods guarantee that every operation can finish in a finite number of its own steps. Unfortunately, it is often the case that the number of steps is quite large, which is why these types of synchronization mechanism are mostly used in real-time systems where predictability is the most important design parameter.

// Try to acquire lock
while $\neg TAS(lock)$ **do**
{}
do stuff...
// Release lock
$lock \leftarrow 0$

(A) Naive.

// Try to acquire lock
while $lock \neq 0 \vee \neg TAS(lock)$ **do**
{}
do stuff...
// Release lock
$lock \leftarrow 0$

(B) Better.

Figure 1.7: *Two lock implementations.*

The term non-blocking is usually used to describe methods that are either lock-free or wait-free.

1.1.5 Blocking Synchronization

The standard way of implementing shared data objects is often by the use of basic synchronization constructs, such as locks and semaphores. Blocking shared data objects that rely on mutual exclusion are often easier to design than their non-blocking counterpart, but a lot of time is spent in the actual synchronization, due to busy waiting and convoying.

Figure 1.7 (a) shows a naive implementation of a lock using the Test-And-Set primitive. The problem with this implementation is that during heavy contention the memory bus will be repeatedly locked by different processes trying to atomically change the value of the lock variable. A better way to implement the lock is to do a Test-And-Test-And-Set operation, as in Figure 1.7 (b). Now each process does a normal read first to see if the lock is available and only then it tries to do a TAS to acquire the lock. But even then this kind of busy waiting is expensive. Another way to lower the contention on a lock is to use some sort of backoff function that tells the process to wait for a certain amount of time before checking again. An often used method is to increase the backoff time exponentially for every failed attempt to acquire the lock. The problem here is

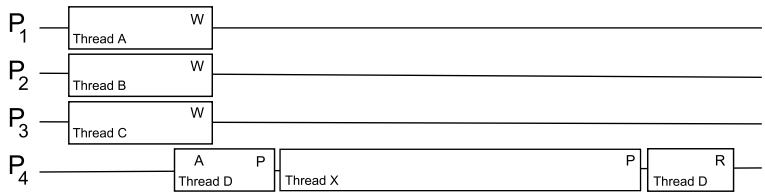


Figure 1.8: *The active thread on processor P_4 acquires the lock. P_1, P_2, and P_3 also need the lock to be able to proceed. The thread on P_4 that holds the lock is preempted by another thread before it can release the lock. All threads are now waiting for a thread that does not even use the lock.*

that it is hard to tune the backoff function and that some processes might have to wait much longer than other processes before acquiring the lock.

An alternative implementation of a lock is the queue lock. When a process fails to acquire a lock, it adds itself to a queue associated with the lock and makes a context switch so that another thread can use the processor while it is waiting for the lock to be released. When the process that is currently holding the lock finishes, it notifies the next process in the queue that the lock is free and switches in its context. Unfortunately, the context switching is often not free and it might be more efficient to just spin on the lock.

The convoying problem occurs when a process is preempted and is unable to release the lock before getting swapped out. This causes other processes to have to wait longer than necessary, potentially slowing the whole program, since it has to wait for another process that is not holding the lock. See Figure 1.8. In a real-time setting this could lead to priority inversion. If a high priority thread needs to wait for a lower priority thread holding a lock, and a middle priority thread comes and preempts the low priority lock, the high priority thread is forced to wait for a middle priority thread that does not have any connection to the lock. This problem is often solved by increasing the priority of the lower priority thread while it is holding the lock, either to a specified high priority, Priority-Ceiling-Protocol (PCP), or to the priority of highest priority thread waiting for the lock, Priority-Inheritance-Protocol (PIP).

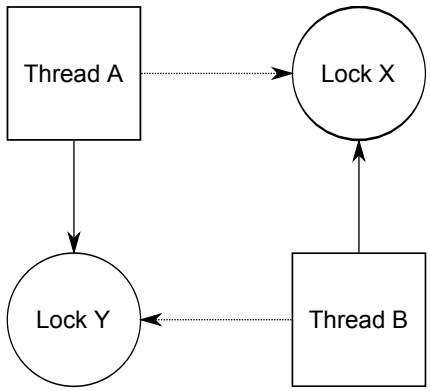


Figure 1.9: *Thread A holds lock Y but needs lock X to continue. Thread B holds lock X and needs lock Y. Since neither can release their locks, we have a deadlock situation.*

Another big problem is the potential for dead-locks, the case where two or more processes circularly wait for locks held by the other. See Figure 1.9. It might be hard to spot potential deadlock situations and the risk for deadlocks makes it hard to compose different blocking data structures since it is not always possible to know how closed source libraries do their locking.

1.1.6 Non-Blocking Synchronization

Non-blocking, in contrast to blocking, shared data objects allows access by several processes at the same time, without using mutual exclusion. By definition, they cannot block a process from accessing them, which means that they avoid the problems with convoys and lock contention. Such objects also offer greater fault-tolerance since one process can always make progress, whereas in a blocking scenario, if the process holding the lock would crash, the data structure would be locked permanently for all other processes. A non-blocking solution also eliminates the risk of deadlocks.

A standard way of implementing a non-blocking data structure is to prepare any changes to the data structure locally and then use an atomic primitive, such as Compare-And-Swap, to make them effective in one step. If the CAS

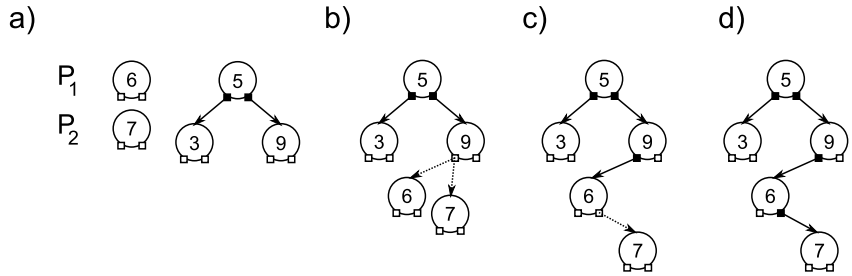


Figure 1.10: a) *Two different processes,* P_1 *and* P_2*, want to insert the elements* 6 *and* 7 *into the tree.* **b)** *They both want to insert their element as the left child of the node with value* 9*. Using Compare-And-Swap they both try to change the value from* null *to point to their node.* **c)** *As Compare-And-Swap is an atomic instruction, only one process will succeed. This time it was process* P_1*. Process* P_2 *now tries to insert its node as the child of the node with value* 6*.* **d)** *Process* P_2 *is not competing with another process, so the Compare-And-Swap will be successful.*

operation fails, it means that another process got to it first and we have to retry. Figure 1.10 shows an example where a new node is to be added to a tree.

Sometimes it is not possible to do all changes using just one atomic operation. For example, after inserting a node into a tree, it might be required to rebalance it. To make sure that any other concurrent process is not hindered, we can announce the operation that we are doing to all other processes, which will allow them to help us perform the operation, if we are not quick enough ourselves. When they have helped us, they can proceed with their own operations. A problem with helping is that it can potentially be unfair. A situation might occur where one process gets stuck helping other processes indefinitely, and never manages to perform its own operations.

The correctness of non-blocking operations is often proved by showing that they are linearizable, a correctness criterion introduced by Herlihy and Wing [HW87]. In this model, each operation on a shared data object consists of an invocation and a response. A sequence of such operations makes up a history. Operations in a concurrent history can be placed in any order if they occur concurrently, but an operation that finishes before another is invoked must

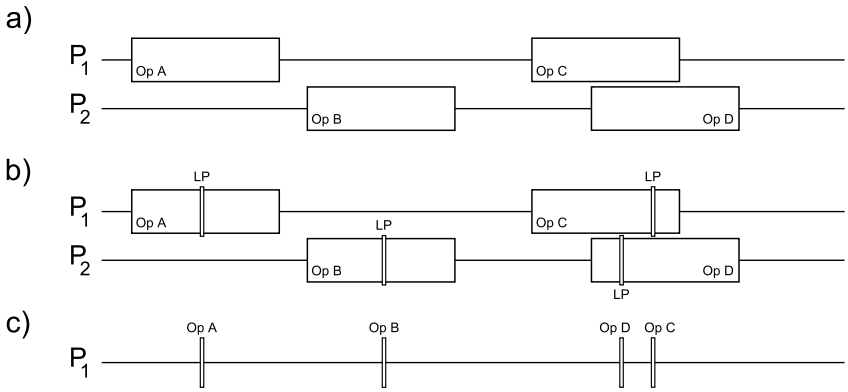


Figure 1.11: a) *A concurrent history where operation A finishes before operation B, and operations C and D occur concurrently.* **b)** *The same concurrent history with the linearization points marked out. As operation A finished before operation B, its linearization point must occur before the linearization point of operation B. Operations C and D occurred concurrently and there is thus no requirement on the order of their linearization points.* **c)** *A sequential history using the linearization points from the concurrent history.*

appear before the latter. If the operations in any actual concurrent history can be reordered in such a way that the history is equivalent to a correct sequential history, then the concurrent object is linearizable. One way of looking at linearizability is to think that an operation takes effect at a specific point in time, the linearization point. All operations can then be ordered according to the linearization point to form a sequential history. See Figure 1.11 for an example.

1.1.7 Composition

Whereas operations of sequential data structures can be quite easily composed, the composition of concurrent data structures is often a challenge. There is a lack of a general, efficient, lock-free method for composing operations that makes it difficult for the programmer to perform multiple operations together atomically. To efficiently glue together multiple objects and their respective

operations, one needs to perform an often challenging task that requires an efficient algorithmic design for every particular composition. The task is made difficult by the fact that lock-free data objects are often too complicated to be trivially altered.

Composing blocking data objects also puts the programmer in a difficult situation, as it requires knowledge of the way locks are handled internally (in the implementation of the objects themselves) in order to avoid deadlocks. It is not possible to build on lock-based components without examining their implementations and even then the drawbacks of locking will not go away.

Composing lock-free concurrent data objects has been an open problem in the area of lock-free data objects. There exist customized compositions of specific concurrent data objects, including the composition of lock-free flat-sets by Gidenstam et al. that constitute the foundation of a lock-free memory allocator [GPT09, GPT05], but no generic solution.

Using blocking locks to compose lock-free operations is not a viable solution, as it would reduce the concurrency and remove the lock-freedom guarantees of the operations. The reason for this is that the lock-free operations would have to acquire a lock before executing in order to ensure that they are not executed concurrently with any composed operations. This would cause the operations to be executed sequentially and lose their lock-free behavior. Simply put, a generic way to compose concurrent objects, without foiling the possible lock-freedom guarantees of the objects, has to be lock-free itself.

We will take a closer look at this problem in section 1.3.

1.1.8 Transactional Memory

Transactional memory (TM) is a potential future way of simplifying synchronization [Kni86, HM93]. The basic idea, taken from the database domain, is to mark sections of code that need to be performed atomically. See Figure 1.12 for an example. An invocation of this section is called a transaction. The transactional memory then tries to perform all instructions in the transaction atomically. If it notices a conflict it aborts and undoes any potential

```
...
atomic
    if x > 10 then
        x ← x^2
    else
        x ← x + 1
end
...
```

Figure 1.12: *Normal use of a software memory transaction.*

changes it has already done. It then tries to redo the transaction until it succeeds. There are no mainstream hardware implementations of transactional memories so far, but a lot of work has been done implementing transactional memory in software [ST95, DSS06, HF03, Moi97, Enn06, HLMS03, FH07, ATLM+06, HPST06, SATH+06], and creating hybrid versions that use both hardware and software [DFL+06, KCJ+06]. For a good overview we recommend the *Transactional Memory* paper by Larus and Kozyrakis [LK08].

A main problem so far has been to achieve good performance and making it composable with other non-blocking data structures. There are also the problems with how side-effects that cannot be undone should be handled inside the transaction.

1.2 Graphics Processors

Today's graphics processors contain very powerful many-core processors; for example, some of NVIDIA's highest-end graphics processors currently boast 512 stream processors. These processors are specialized for compute-intensive, greatly parallel computations and they could be used to assist the CPU in solving problems that can be efficiently data-parallelized.

Previous work on general purpose computation on GPUs have used the OpenGL interface; but since it was primarily designed for performing graphics operations, it gives a poor abstraction to the programmer who wishes to

use it for non-graphics related tasks. NVIDIA has improved the situation by providing programmers with CUDA, a programming platform for doing general purpose computation on GPUs. It consists of a compiler for a C-based language that can be used to create kernels that can be executed on the GPU. Also included are high performance numerical libraries for FFT and linear algebra [NVI07]. OpenCL is a similar initiative to CUDA, but has a wider industrial backing and will have support for multi-core platforms other than NVIDIA's graphics processors [Gro08]. As of the time of writing this text, OpenCL is still in a relative infancy, so the rest of this section will deal exclusively with CUDA-based graphics processors.

1.2.1 General Architecture

The high range graphics processor from NVIDIA that supports CUDA currently boasts 16 multiprocessors/stream-processors, each multiprocessor consisting of 32 processors that all execute the same instruction on different data in lock-step. Each multiprocessor supports up to 1536 threads, has 48 KB of fast processor local memory and a maximum of 32 K available registers that can be divided between the threads.

1.2.2 Scheduling

Threads are logically divided into *thread blocks* that are assigned to a specific multiprocessor. Depending on how many registers and how much local memory the block of threads requires, there could be multiple blocks assigned to a single multiprocessor. If more blocks are needed than there is room for on any of the multiprocessors, the leftover blocks will be scheduled sequentially.

The GPU schedules threads depending on which *warp* they are in. Threads with *id* $0..31$ are assigned to the first warp, threads with *id* $32..63$ to the next, and so on. When a warp is scheduled for execution, the threads that perform the same instructions are executed concurrently (limited by the size of the multiprocessor), whereas threads that deviate are executed sequentially. Hence it is important to try to make all threads in the same warp perform the same instruc-

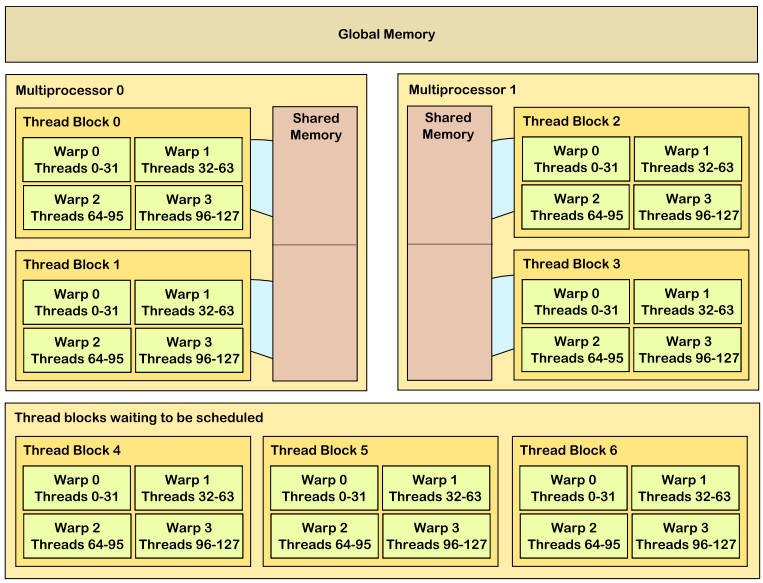


Figure 1.13: *A graphical representation of the CUDA hardware model.*

tions most of the time. See Figure 1.13 for a graphical description of the way threads are grouped together and scheduled.

Two warps cannot execute simultaneously on a single multiprocessor, so one could see the warp as the counter-part of the thread in a conventional SMP system. All instructions on the GPU are SIMD, so the threads that constitute a warp can be seen as a way to simplify the use of these instructions. Instead of each thread issuing SIMD instructions on 32-word arrays, the threads are divided into 32 sub-threads and each works on its own word.

1.2.3 Synchronization

Threads within a thread block can use the multiprocessors' shared local memory and a special thread barrier-function to communicate with each other. The barrier-function forces all threads in the same block to synchronize. There is, however, no barrier-function for threads from different blocks. The reason for

this is that when more blocks are executed than there is room for on the multiprocessors, the scheduler will wait for a thread block to finish executing before it swaps in a new block. This makes it impossible to implement a barrier function in software and the only solution is to wait until all blocks have completed.

All new graphics processors support atomic instructions, such as Compare-And-Swap and Fetch-And-Add, on both global and local memory.

1.2.4 Memory

Data is stored in a large global memory that supports both gather and scatter operations. On earlier NVIDIA graphics processors there was no caching available automatically when accessing this memory, but each thread block could use its own, very fast, shared local memory to temporarily store data and use it as a manual cache. By letting each thread access consecutive memory locations, the hardware can coalesce several read or write operations into one big read or write operation, which increases performance. With the new Fermi architecture, there is now an L1/L2 cache available when accessing the global memory.

The local shared memory is divided into memory banks that can be accessed in parallel. If two threads write to or read from the same memory bank, the accesses will be serialized. Due to this, one should always try to make threads in the same warp write to different banks. If all threads read from the same memory bank, a broadcasting mechanism will be used, making it just as fast as a single read. A normal access to the local shared memory takes the same amount of time as accessing a register.

1.3 Our work

In our work we have been investigating how the graphics processor differs from conventional multi-core processors when it comes to synchronization between different processing units. To learn more about the platform we created a Quicksort algorithm specifically designed for the graphics processor and managed to achieve good performance results, compared to already existing sorting solu-

tions. Sorting is an important component of numerous algorithms and programs, for example within database, graphics, and finance applications. Quicksort has, however, been seen earlier by some as unsuitable for use on graphics processors due to the extra overhead needed compared to other algorithms, but our implementation shows that it can perform on the same level as other algorithms.

When implementing the Quicksort algorithm we noticed that the scheduler on the graphics processor did a poor job of achieving a decent load balance, so we decided to try our hands on other, dynamic, load balancing schemes that have been much used on conventional multiprocessors. We compared a blocking global work queue, a non-blocking global work queue, and a non-blocking work-stealing scheme with the built-in load balancing mechanism on the graphics processor. We found that blocking data structures exhibited very poor performance when contention increased, as expected, and that non-blocking work-stealing could outperform the built-in load balancing.

A difficulty when dealing with concurrent data structures in general is that it is hard to compose their operations. With blocking implementations care needs to be taken to avoid dead-locks, and non-blocking algorithms are often too complicated to be trivially altered. We approached this problem by trying to compose two of the most common operations available – the operations for inserting and removing elements. We found a common theme in the use of the compare-and-swap operation as a linearization point. By combining the linearization points of two different operations with the help of a double- or multi-word compare-and-swap, it was possible to compose these operations with relative ease. This allowed for the creation of a generic move operation that could be used to atomically move elements between data structures that had been adapted using our methodology.

We then looked at a more generic way of composing data structures using software transactional memory. STM:s compose easily and can be used to speedily implement concurrent data structures. As no work had been done applying software transactional memory to graphics processors, we decided to evaluate two STM designs on a GPU, one blocking and one obstruction-free.

1.3.1 A Practical Quicksort Algorithm for Graphics Processors

In the first paper we presented GPU-Quicksort, a parallel Quicksort algorithm designed to take advantage of the high bandwidth of GPUs by minimizing the amount of bookkeeping and inter-thread synchronization needed [CT08a, CT09].

The bookkeeping is minimized by constraining all thread blocks to work with only one (or part of a) sequence of data at a time. This way pivot values do not need to be distributed to all thread blocks and thus no extra information needs to be written to the global memory.

The two-pass design of GPU-Quicksort has been introduced to keep the inter-thread synchronization low. First the algorithm goes through the sequence to sort, counting the number of elements that each thread sees that have a higher (or lower) value than the pivot. By calculating a cumulative sum of these sums, in the second phase, each thread will know where to write its assigned elements without any extra synchronization. The small number of inter-block synchronization that is required between the two passes of the algorithm can be reduced further by taking advantage of the atomic synchronization primitives that are available on newer hardware.

A previous implementation of Quicksort for GPUs by Sengupta et al. turned out not to be competitive enough in comparison to radix sort or even CPU-based sorting algorithms [SHZO07]. According to the authors, this was due to it being more dependent on the processor speed than on the bandwidth. This is in contrast with GPU-Quicksort, which is bandwidth bound, which means that it gains significantly from the high bandwidth of GPUs and scales in performance such as bandwidth increases.

In our experiments we compared GPU-Quicksort with some of the fastest known sorting algorithms for GPUs, as well as with the C++ Standard Library sorting algorithm, Introsort, for reference. We used several input distributions and two different graphics processors, the low-end 8600GTS with 32 cores and the high-end 8800GTX with 128 cores, both from NVIDIA. What we could

observe was that GPU-Quicksort performed better on all distributions on the high-end processor and on par with or better on the low-end processor.

A significant conclusion, we think, that can be drawn from this work, is that Quicksort is a practical alternative for sorting large quantities of data on graphics processors.

1.3.2 On Load Balancing on Graphics Processors

In the second paper we have compared four load balancing methods, a blocking queue, a non-blocking queue, a non-blocking work stealing scheme and a static list, on the task of creating an octree partitioning of a set of particles [CT08b].

We found that the blocking queue performed poorly and scaled badly when faced with more processing units, something that can be attributed to the inherent busy waiting. The non-blocking queue performed better but scaled poorly when the number of processing units got too high. Since the number of tasks increased quickly and the tree itself was relatively shallow the static queue performed well. The non-blocking work-stealing method perform very well and outperformed the static method.

The experiments showed that synchronization can be very expensive and that new methods that take more advantage of the graphics processors' features and capabilities might be required. They also showed that lock-free methods achieve better performance than blocking, and that they can be made to scale with increased numbers of processing units.

1.3.3 Supporting Lock-Free Composition of Concurrent Data Objects

Lock-free data objects offer several advantages over their blocking counterparts, such as being immune to deadlocks and convoying and, more importantly, being highly concurrent. However, composing the operations they provide into larger atomic operations, while still guaranteeing efficiency and lock-freedom, is a challenging algorithmic task.

In the third paper we present a lock-free methodology for composing highly concurrent linearizable objects together by unifying their linearization points [CT10]. This makes it possible to relatively easily introduce atomic lock-free move operations to a wide range of concurrent objects. Our experimental results demonstrate that the methodology presented in the paper, applied to the classical lock-free implementations, offers better performance and scalability than a composition method based on locking. These results also demonstrate that it does not introduce noticeable performance penalties to the previously supported operations of the concurrent objects.

Our methodology can also be easily extended to support n operations on n distinct objects, for example to create functions that remove an item from one object and insert it into n others atomically.

1.3.4 Towards a Software Transactional Memory for Graphics Processors

The introduction of general purpose computing on many-core graphics processor systems, and the general shift in the industry towards parallelism, has created a demand for ease of parallelization. Software transactional memory (STM) simplifies development of concurrent code by allowing the programmer to mark sections of code to be executed concurrently and atomically in an optimistic manner. In contrast to locks, STMs are easy to compose and do not suffer from deadlocks.

In our fourth paper we have designed and implemented two STMs for graphics processors, one blocking and one non-blocking [CTC10]. The design issues involved in the design of these two STMs are described and explained in the paper together with experimental results comparing the performance of the two STMs. We found that while a blocking STM is simpler to implement, providing additional progress guarantees such as obstruction-freeness, improves performance and lowers the number of aborted transactions.

Bibliography

[ATLM+06] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM.

[CT08a] Daniel Cederman and Philippas Tsigas. A Practical Quicksort Algorithm for Graphics Processors. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA 2008), Lecture Notes in Computer Science Vol.: 5193*, pages 246–258, 2008.

[CT08b] Daniel Cederman and Philippas Tsigas. On dynamic load balancing on graphics processors. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 57–64, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[CT09] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics (JEA)*, 14:1.4–1.24, 2009.

[CT10] Daniel Cederman and Philippas Tsigas. Supporting lock-free composition of concurrent data objects. In *CF '10: Proceedings of the 7th ACM international conference on Computing frontiers*, pages 53–62, New York, NY, USA, 2010. ACM.

[CTC10] Daniel Cederman, Philippas Tsigas, and Muhammad Tayyab Chaudhry. Towards a Software Transactional Memory for Graphics Processors. pages 121–129, 2010.

[DFL+06] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum. Hybrid Transactional Memory. In *Proceedings of the 12th International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 336–346, 2006.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[Enn06] Robert Ennals. Software Transactional Memory Should Not Be Obstruction-Free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[FH07] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2):5, 2007.

[GPST09] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems*, 20:1173–1187, 2009.

[GPT05] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. Allocating Memory in a Lock-Free Manner. In *ESA '05: Proceedings of the 13th Annual European Symposium on Algorithms*, pages 329–342, 2005.

[GPT09] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. NBmalloc: Allocating Memory in a Lock-Free Manner. *Algorithmica*, 2009.

[Gro08] Khronos Group. OpenCL (Open Computing Language). `www.khronos.org/opencl/`, 2008.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, volume 38, pages 388–402, New York, NY, USA, November 2003. ACM Press.

[HLMS03] M. Herlihy, V. Luchangco, M. Moir, and W.N. Scherer. Software Transactional Memory for Dynamic-sized Data Structures. In *Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2003.

[HM93] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support For Lock-Free Data Structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[Hoa62] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(4):10–15, 1962.

[HPST06] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 14–25, New York, NY, USA, 2006. ACM Press.

[HTA10] Phuong Hoai Ha, Philippas Tsigas, and Otto J. Anshus. The synchronization power of coalesced memory accesses. *IEEE Transactions on Parallel and Distributed Systems*, 21:939–953, 2010.

[HW87] Maurice Herlihy and Jeannette Wing. Axioms for concurrent objects. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 13–26, New York, NY, USA, 1987. ACM.

[KCJ+06] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid Transactional Memory. In

Proceedings of Symposium on Principles and Practice of Parallel Programming, Mar 2006.

[Kni86] Tom Knight. An architecture for mostly functional languages. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112, New York, NY, USA, 1986. ACM Press.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[LK08] James Larus and Christos Kozyrakis. Transactional memory: Is TM the answer for improving parallel programming? volume 51, New York, NY, USA, 2008. ACM.

[Mic04a] Maged M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[Mic04b] Maged M. Michael. Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In *Proceedings of the 18th International Symposium on Distributed Computing (DISC), vol. 3274 of LNCS*, pages 144–158, 2004.

[Moi97] Mark Moir. Transparent Support for Wait-Free Transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319. Springer-Verlag, 1997.

[MS95] Maged Michael and Michael Scott. Correction of a memory management method for lock-free data structures. Technical report, University of Rochester, 1995.

[NVI07] NVIDIA Corporation, nvidia.com/cuda. *CUDA SDK*, 1.0 edition, 2007.

[NVI10] NVIDIA Corporation, nvidia.com/cuda. *CUDA SDK*, 3.2 edition, 2010.

[SATH+06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[SHZO07] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan Primitives for GPU Computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 97–106, 2007.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[Val95] John D. Valois. Lock-free linked lists using compare-and-swap. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222, New York, NY, USA, 1995. ACM.

Part I

PAPERS

PAPER I

Daniel Cederman, Philippas Tsigas

GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors

In the ACM Journal of Experimental Algorithmics (JEA)

Vol. 14, No. 4, ACM press 2009.

2

A Practical Quicksort Algorithm for Graphics Processors

In this paper we describe GPU-Quicksort, an efficient Quicksort algorithm suitable for highly parallel multi-core graphics processors. Quicksort has previously been considered an inefficient sorting solution for graphics processors, but we show that in CUDA, NVIDIA's programming platform for general purpose computations on graphical processors, GPU-Quicksort performs better than the fastest known sorting implementations for graphics processors, such as radix and bitonic sort. Quicksort can thus be seen as a viable alternative for sorting large quantities of data on graphics processors.

2.1 Introduction

In this paper, we describe an efficient parallel algorithmic implementation of Quicksort, GPU-Quicksort, designed to take advantage of the highly parallel nature of graphics processors (GPUs) and their limited cache memory. Quicksort has long been considered one of the fastest sorting algorithms in practice for single processor systems and is also one of the most studied sorting algorithms, but until now it has not been considered an efficient sorting solution for GPUs [SHZO07]. We show that GPU-Quicksort presents a viable sorting alternative and that it can outperform other GPU-based sorting algorithms such as GPUSort and radix sort, considered by many to be two of the best GPU-sorting algorithms. GPU-Quicksort is designed to take advantage of the high bandwidth of GPUs by minimizing the amount of bookkeeping and inter-thread synchronization needed. It achieves this by i) using a two-pass design to keep the inter-thread synchronization low, ii) coalescing read operations and constraining threads so that memory accesses are kept to a minimum. It can also take advantage of the atomic synchronization primitives found on newer hardware to, when available, further improve its performance.

Today's graphics processors contain very powerful multi-core processors, for example, NVIDIA's highest-end graphics processor currently boasts 128 cores. These processors are specialized for compute-intensive, highly parallel computations and they could be used to assist the CPU in solving problems that can be efficiently data-parallelized.

Previous work on general purpose computation on GPUs have used the OpenGL interface, but since it was primarily designed for performing graphics operations it gives a poor abstraction to the programmer that wishes to use it for non-graphics related tasks. NVIDIA is attempting to remedy this situation by providing programmers with CUDA, a programming platform for doing general purpose computation on their graphics processors. A similar initiative to CUDA is OpenCL, which specification has just recently been released [Gro08].

Although simplifying the programming, one still needs to be aware of the strengths and limitations of the new platform to be able to take full advantage of

it. Algorithms that work great on standard single processor systems most likely need to be altered extensively to perform well on GPUs, which have limited cache memory and instead use massive parallelism to hide memory latency.

This means that directly porting efficient sorting algorithms from the single processor domain to the GPU domain would most likely yield very poor performance. This is unfortunate, since the sorting problem is very well suited to be solved in parallel and is an important kernel for sequential and multiprocessing computing and a core part of database systems. Being one of the most basic computing problems, it also plays a vital role in plenty of algorithms commonly used in graphics applications, such as visibility ordering or collision detection.

Quicksort was presented by C.A.R. Hoare in 1961 and uses a divide-and-conquer method to sort data [Hoa61]. A sequence is sorted by recursively dividing it into two subsequences, one with values lower and one with values higher than the specific pivot value that is selected in each iteration. This is done until all elements are sorted.

2.1.1 Related Work

With Quicksort being such a popular sorting algorithm, there have been a lot of different attempts to create an efficient parallelization of it. The obvious way is to take advantage of its inherent parallelism by just assigning a new processor to each new subsequence. This means, however, that there will be very little parallelization in the beginning, when the sequences are few and large [ED82].

Another approach has been to divide each sequence to be sorted into blocks that can then be dynamically assigned to available processors [HNR90, TZ03]. However, this method requires extensive use of atomic FAA[1] which makes it too expensive to use on graphics processors.

Blelloch suggested using prefix sums to implement Quicksort and recently Sengupta et al. used this method to make an implementation for CUDA [Ble93, SHZO07]. The implementation was done as a demonstration of their segmented

[1] Fetch-And-Add reads an integer from the memory, increments it by a given amount and writes it back to the memory, all in one atomic step.

scan primitive, but it performed quite poorly and was an order of magnitude slower than their radix-sort implementation in the same paper.

Since most sorting algorithms are memory bandwidth bound, there is no surprise that there is currently a big interest in sorting on the high bandwidth GPUs. Purcell et al. [PDC+03] have presented an implementation of bitonic merge sort on GPUs based on an implementation by Kapasi et al. [KDR+00]. Kipfer et al. [KSW04, KW05] have shown an improved version of the bitonic sort as well as an odd-even merge sort. Greß et al. [GZ06] introduced an approach based on the adaptive bitonic sorting technique found in the Bilardi et al. paper [BN89]. Govindaraju et al. [GRM05] implemented a sorting solution based on the periodic balanced sorting network method by Dowd et al. [DPRS89] and one based on bitonic sort [GRHM05]. They later presented a hybrid bitonic-radix sort that used both the CPU and the GPU to be able to sort vast quantities of data [GGKM06]. Sengupta et al. [SHZO07] have presented a radix-sort and a Quicksort implementation. Recently, Sintorn et al. [SA07] presented a hybrid sorting algorithm which splits the data with a bucket sort and then uses merge sort on the resulting blocks. The implementation requires atomic primitives that are currently not available on all graphics processors.

In the following section, Section 2.2, we present the system model. In Section 2.3.1 we give an overview of the algorithm and in Section 2.3.2 we go through it in detail. We prove its time and space complexity in Section 2.4. In Section 2.5 we show the results of our experiments and in Section 2.5.4 and Section 2.6 we discuss the result and conclude.

2.2 The System Model

CUDA is NVIDIA's initiative to bring general purpose computation to their graphics processors. It consists of a compiler for a C-based language which can be used to create kernels that can be executed on the GPU. Also included are high performance numerical libraries for FFT and linear algebra.

General Architecture The high range graphics processors from NVIDIA that supports CUDA currently boasts 16 multiprocessors, each multiprocessor

consisting of 8 processors that all execute the same instruction on different data in lock-step. Each multiprocessor supports up to 768 threads, has 16KiB of fast local memory and a maximum of 8192 available registers that can be divided between the threads.

Scheduling Threads are logically divided into *thread blocks* that are assigned to a specific multiprocessor. Depending on how many registers and how much local memory the block of threads requires, there could be multiple blocks assigned to a single multiprocessor. If more blocks are needed than there is room for on any of the multiprocessors, the leftover blocks will be run sequentially.

The GPU schedules threads depending on which *warp* they are in. Threads with *id* $0..31$ are assigned to the first warp, threads with *id* $32..63$ to the next and so on. When a warp is scheduled for execution, the threads which perform the same instructions are executed concurrently (limited by the size of the multiprocessor) whereas threads that deviate are executed sequentially. Hence it is important to try to make all threads in the same warp perform the same instructions most of the time. See Figure 2.1 for a graphical description of the way threads are grouped together and scheduled.

Two warps cannot execute simultaneously on a single multiprocessor, so one could see the warp as the counter-part of the thread in a conventional SMP system. All instructions on the GPU are SIMD, so the threads that constitute a warp can be seen as a way to simplify the usage of these instructions. Instead of each thread issuing SIMD instructions on 32-word arrays, the threads are divided into 32 sub-threads that each works on its own word.

Synchronization Threads within a thread block can use the multiprocessors shared local memory and a special thread *barrier-function* to communicate with each other. The barrier-function forces all threads in the same block to synchronize, that is, a thread calling it will not be allowed to continue until all other threads have also called it. They will then continue from the same position in the code. There is however no barrier-function for threads from different blocks. The reason for this is that when more blocks are executed than there is room for on the multiprocessors, the scheduler will wait for a thread block

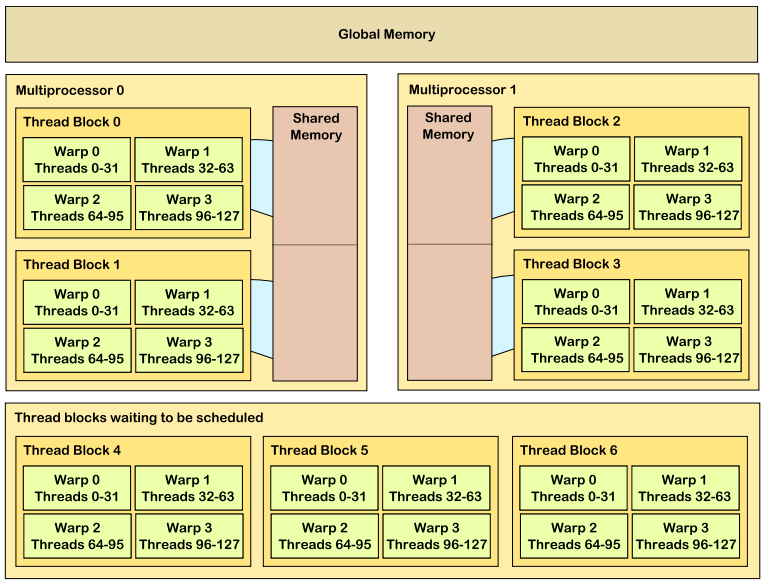


Figure 2.1: *A graphical representation of the CUDA hardware model*

function CAS($ptr, oldval, newval$)
    if $*ptr = oldval$ **then**
        $*ptr \leftarrow newval$;
        return $oldval$;
    return $*ptr$;

function FAA($ptr, value$)
    $*ptr \leftarrow *ptr + value$;
    return $*ptr - value$;

Figure 2.2: *The specification for the Compare-And-Swap and Fetch-And-Add operation. The two operations are performed atomically.*

to finish executing before it swaps in a new block. This makes it impossible to implement a barrier function in software and the only solution is to wait until all blocks have completed.

Newer graphics processors support atomic instructions such as Compare-And-Swap (CAS) and Fetch-And-Add (FAA). See Figure 2.2 for the specification of these synchronization operations.

Memory Data is stored in a large global memory that supports both gather and scatter operations. There is no caching available automatically when accessing this memory, but each thread block can use its own, very fast, shared local memory to temporarily store data and use it as a manual cache. By letting each thread access consecutive memory locations, the hardware can coalesce several read or write operations into one big read or write operation, which increases performance.

This is in direct contrast with the conventional SMP systems, where one should try to let each thread access its own part of the memory so as to not thrash the cache.

Because of the lack of caching, a high number of threads are needed to hide the memory latency. These threads should preferably have a high ratio of arithmetic to memory operations to be able to hide the latency well.

The shared memory is divided into memory banks that can be accessed in parallel. If two threads write to or read from the same memory bank, the accesses will be serialized. Due to this, one should always try make threads in the same warp write to different banks. If all threads read from the same memory bank, a broadcasting mechanism will be used, making it just as fast as a single read. A normal access to the shared memory takes the same amount of time as accessing a register.

2.3 The algorithm

The following subsection gives an overview of GPU-Quicksort. Section 2.3.2 will then go into the algorithm in more details.

2.3.1 Overview

The method used by the algorithm is to recursively *partition* the sequence to be sorted, i.e. to move all elements that are lower than a specific pivot value to a position to the left of the pivot and to move all elements with a higher value to the right of the pivot. This is done until the entire sequence has been sorted.

In each partition iteration a new pivot value is picked and as a result two new subsequences are created that can be sorted independently. In our experiments we use a deterministic pivot selection that is described in Section 2.5, but a randomized method could also be used. After a while there will be enough subsequences available that each thread block can be assigned one. But before that point is reached, the thread blocks need to work together on the same sequences. For this reason, we have divided up the algorithm into two, albeit rather similar, phases.

First Phase In the first phase, several thread blocks might be working on different parts of the same sequence of elements to be sorted. This requires appropriate synchronization between the thread blocks, since the results of the different blocks needs to be merged together to form the two resulting subsequences.

Newer graphics processors provide access to atomic primitives that can aid somewhat in this synchronization, but they are not yet available on the high-end graphics processors and there is still the need to have a thread block barrier-function between the partition iterations, something that cannot be implemented using the available atomic primitives.

The reason for this is that the blocks might be executed sequentially and we have no way of knowing in which order they will be run. So the only way to synchronize thread blocks is to wait until all blocks have finished executing. Then one can assign new sequences to them. Exiting and reentering the GPU is not expensive, but it is also not delay-free since parameters needs to be copied from the CPU to the GPU, which means that we want to minimize the number of times we have to do that.

When there are enough subsequences so that each thread block can be assigned its own, we enter the second phase.

Second Phase In the second phase, each thread block is assigned its own subsequence of input data, eliminating the need for thread block synchronization. This means that the second phase can run entirely on the graphics processor. By using an explicit stack and always recurse on the smallest subsequence, we minimize the shared memory required for bookkeeping.

Hoare suggested in his paper [Hoa62] that it would be more efficient to use another sorting method when the subsequences are relatively small, since the overhead of the partitioning gets too large when dealing with small sequences. We decided to follow that suggestion and sort all subsequences that can fit in the available local shared memory using an alternative sorting method. For the experiments we decided to use bitonic sort, but other sorting algorithms could also be used.

In-place On conventional SMP systems it is favorable to perform the sorting in-place, since that gives good cache behavior. But on the GPUs with their limited cache memory and the expensive thread synchronization that is required when hundreds of threads need to communicate with each other, the advantages of sorting in-place quickly fade. Here it is better to aim for reads and writes to be coalesced to increase performance, something that is not possible on conventional SMP systems. For these reasons it is better, performance-wise, to use an auxiliary buffer instead of sorting in-place.

So, in each partition iteration, data is read from the primary buffer and the result is written to the auxiliary buffer. Then the two buffers switch places and the primary becomes the auxiliary and vice versa.

Partitioning

The principle of two phase partitioning is outlined in Figure 2.3. A sequence to be partitioned is selected and it is then logically divided into m equally sized sections (Step a), where m is the number of thread blocks available. Each thread block is then assigned a section of the sequence (Step b).

The thread block goes through its assigned data, with all threads in the block accessing consecutive memory so that the reads can be coalesced. This is important, since reads being coalesced will significantly lower the memory access time.

Synchronization The objective is to partition the sequence, i.e. to move all elements that are lower than the pivot to a position to the left of the pivot in the auxiliary buffer and to move the elements with a higher value to the right of the pivot. The problem here is how to synchronize this in an efficient way. How

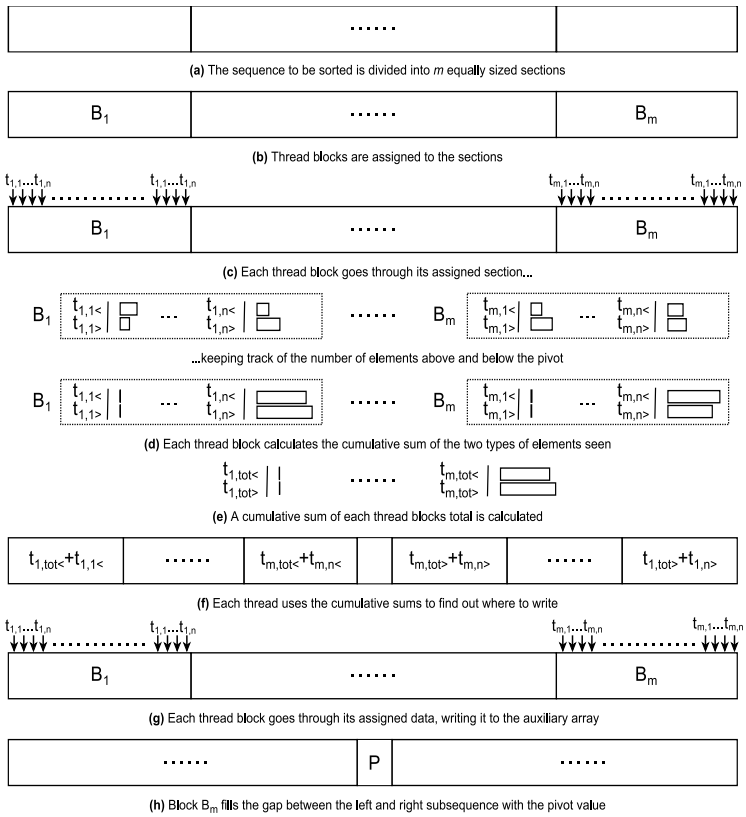


Figure 2.3: *Partitioning a sequence*

do we make sure that each thread knows where to write in the auxiliary buffer? It should also be noted that it is important to minimize the amount of synchronization communication between threads since it will be quite expensive as we have so many threads.

Cumulative Sum A possible solution is to let each thread read an element and then synchronize the threads using a barrier function. By calculating a cumulative sum[2] of the number of threads that want to write to the left and that wants to write to the right of the pivot, each thread would know that x

[2]The terms prefix sum or sum scan are also used in the literature.

threads with a lower thread id than its own are going to write to the left and that y threads are going to write to the right. Each thread then knows that it can write its element to either buf_{x+1} or $buf_{n-(y+1)}$, depending on if the element is higher or lower than the pivot.

A Two-Pass Solution But calculating a cumulative sum is not free, so to improve performance we go through the sequence two times. In the first pass each thread just counts the number of elements it has seen that have a value higher (or lower) than the pivot (Step c). Then when the block has finished going through its assigned data, we use these sums instead to calculate the cumulative sum (Step d). Now each thread knows how much memory the threads with a lower id than its own needs in total, turning it into an implicit memory-allocation scheme that only needs to run once for every thread block, in each iteration.

In the first phase, were we have several thread blocks accessing the same sequence, an additional cumulative sum need to be calculated for the total memory used by each thread block (Step e).

Now when each thread knows where to store its elements, we go through the data in a second pass (Step g), storing the elements at their new position in the auxiliary buffer. As a final step, we store the pivot value at the gap between the two resulting subsequences (Step h). The pivot value is now at its final position which is why it does not need to be included in any of the two subsequences.

2.3.2 Detailed Description

The following subsection describes the algorithm in more detail.

The First Phase

The goal of the first phase is to divide the data into a large enough number of subsequences that can be sorted independently.

Work Assignment In the ideal case, each subsequence should be of the same size, but that is often not possible, so it is better to have some extra sequences and let the scheduler balance the workload. Based on that observation,

Algorithm 1 GPU-Quicksort (CPU Part)

procedure GPUQSORT($size$, d, $\hat{d}$) ▷ d contains the sequence to be sorted.
$startpivot \leftarrow$median(d_0, $d_{size/2}$, d_{size}) ▷ d_x is the value at index x.
$work \leftarrow \{(d_{0 \to size}, startpivot)\}$ ▷ $d_{x \to y}$ is the sequence from index x to y.
$done \leftarrow \emptyset$
while $work \neq \emptyset \wedge |work| + |done| < maxseq$ **do** ▷ Divide into $maxseq$ subsequences.
  $blocksize \leftarrow \sum_{(seq,pivot) \in work} \frac{||seq||}{maxseq}$ ▷ $||seq||$ is the length of sequence seq.
  $blocks \leftarrow \emptyset$
  for all $(seq_{start \to end}, pivot) \in work$ **do** ▷ Divide sequences into blocks.
    $blockcount \leftarrow \lceil \frac{||seq||}{blocksize} \rceil$ ▷ Number of blocks to create for this sequence.
    $parent \leftarrow (seq, seq, blockcount)$ ▷ Shared variables for all blocks of this sequence.

    for $i \leftarrow 0, i < blockcount - 1, i \leftarrow i + 1$ **do**
      $bstart \leftarrow start + blocksize \cdot i$
      $blocks \leftarrow blocks \cup \{(seq_{bstart \to bstart + blocksize}, pivot, parent)\}$
    $blocks \leftarrow blocks \cup \{(seq_{start + blocksize \cdot (blockcount-1) \to end}, pivot, parent)\}$

  $news \leftarrow$ gqsort$\ll |blocks| \gg (blocks, d, \hat{d})$ ▷ Start $|blocks|$ thread blocks.
  $work \leftarrow \emptyset$
  for all $(seq, pivot) \in news$ **do** ▷ If the new sequences are too long; partition them further.
    if $||seq|| < size/maxseq$ **then**
      $done \leftarrow done \cup \{(seq, pivot)\}$
    else
      $work \leftarrow work \cup \{(seq, pivot)\}$
$done \leftarrow done \cup work$ ▷ Merge the sets done and work.
lqsort$\ll |done| \gg (done, d, \hat{d})$ ▷ Do final sort.

a good way to partition is to only partition subsequences that are longer than $minlength = n/maxseq$, where n is the total number of elements to sort, and to stop when we have $maxseq$ number of sequences. We discuss how to select a good value for $maxseq$ in Section 2.5.

In the beginning of each iteration, all sequences that are larger than the value of $minlength$ are assigned thread blocks relative to their size. In the first iteration, the original sequence will be assigned all available thread blocks. The sequences are divided so that each thread block gets an equally large section to sort, as can be seen in Figure 2.3 (Step a and b).

First Pass When a thread block is executed on the GPU, it will iterate through all the data in its assigned sequence. Each thread in the block will keep track of the number of elements that are greater than the pivot and the number

Algorithm 2 GPU-Quicksort (First Phase GPU Kernel)

function GQSORT($blocks, d, \hat{d}$)
var global $sstart, send, oldstart, oldend, blockcount$
var block local $lt, gt, pivot, start, end, s$
var thread local $i, lfrom, gfrom, lpivot, gpivot$

$(s_{start \to end}, pivot, parent) \leftarrow blocks_{blockid}$ ▷ Get the sequence block assigned to this thread block.
$lt_{threadid}, gt_{threadid} \leftarrow 0, 0$ ▷ Set thread local counters to zero.

$i \leftarrow start + threadid$ ▷ Align thread accesses for coalesced reads.
for $i < end, i \leftarrow i + threadcount$ **do** ▷ Go through the data...
  if $s_i < pivot$ **then** ▷ counting elements that are smaller...
    $lt_{threadid} \leftarrow lt_{threadid} + 1$
  if $s_i > pivot$ **then** ▷ or larger compared to the pivot.
    $gt_{threadid} \leftarrow gt_{threadid} + 1$

$lt_0, lt_1, lt_2, ..., lt_{sum} \leftarrow 0, lt_0, lt_0 + lt_1, ..., \sum_{i=0}^{threadcount} lt_i$ ▷ Calculate the cumulative sum.
$gt_0, gt_1, gt_2, ..., gt_{sum} \leftarrow 0, gt_0, gt_0 + gt_1, ..., \sum_{i=0}^{threadcount} gt_i$

if $threadid = 0$ **then** ▷ Allocate memory in the sequence this block is a part of.
  $(seq_{sstart \to send}, oseq_{oldstart \to oldend}, blockcount) \leftarrow parent$ ▷ Get shared variables.
  $lbeg \leftarrow FAA(sstart, lt_{sum})$ ▷ Atomic increment allocates memory to write to.
  $gbeg \leftarrow FAA(send, -gt_{sum}) - gt_{sum}$ ▷ Atomic is necessary since multiple blocks access this variable.

$lfrom = lbeg + lt_{threadid}$
$gfrom = gbeg + gt_{threadid}$

$i \leftarrow start + threadid$
for $i < end, i \leftarrow i + threadcount$ **do** ▷ Go through data again writing elements
  if $s_i < pivot$ **then** ▷ to the their correct position.
    $\neg s_{lfrom} \leftarrow s_i$ ▷ If s is a sequence in d, $\neg s$ denotes the corresponding
    $lfrom \leftarrow lfrom + 1$ ▷ sequence in $\hat{d}$ (and vice versa).
  if $s_i > pivot$ **then**
    $\neg s_{gfrom} \leftarrow s_i$
    $gfrom \leftarrow gfrom + 1$

if $threadid = 0$ **then**
  if $FAA(blockcount, -1) = 0$ **then** ▷ Check if this is the last block in the sequence to finish.
    for $i \leftarrow sstart, i < send, i \leftarrow i + 1$ **do** ▷ Fill in pivot value.
      $d_i \leftarrow pivot$
    $lpivot \leftarrow median(\neg seq_{oldstart}, \neg seq_{(oldstart+sstart)/2}, \neg seq_{sstart})$
    $gpivot \leftarrow median(\neg seq_{send}, \neg seq_{(send+oldend)/2}, \neg seq_{oldend})$
    result $\leftarrow$ **result** $\cup \{(\neg seq_{oldstart \to sstart}, lpivot)\}$
    result $\leftarrow$ **result** $\cup \{(\neg seq_{send \to oldend}, gpivot)\}$

Algorithm 3 GPU-Quicksort (Second Phase GPU Kernel)

procedure LQSORT($seqs, d, \hat{d}$)
var block local $lt, gt, pivot, workstack, start, end, s, newseq1, newseq2, longseq, shortseq$
var thread local $i, lfrom, gfrom$

push $seqs_{blockid}$ **on** $workstack$ ▷ Get the sequence assigned to this thread block.
while $workstack \neq \emptyset$ **do**
  pop $s_{start \to end}$ **from** $workstack$ ▷ Get the shortest sequence from set.
  $pivot \leftarrow median(s_{start}, s_{(end+start)/2)}, s_{end})$ ▷ Pick a pivot.
  $lt_{threadid}, gt_{threadid} \leftarrow 0, 0$ ▷ Set thread local counters to zero.
  $i \leftarrow start + threadid$ ▷ Align thread accesses for coalesced reads.
  for $i < end, i \leftarrow i + threadcount$ **do** ▷ Go through the data...
    if $s_i < pivot$ **then** ▷ counting elements that are smaller...
      $lt_{threadid} \leftarrow lt_{threadid} + 1$
    if $s_i > pivot$ **then** ▷ or larger compared to the pivot.
      $gt_{threadid} \leftarrow gt_{threadid} + 1$
  $lt_0, lt_1, lt_2, ..., lt_{sum} \leftarrow 0, lt_0, lt_0 + lt_1, ..., \sum_{i=0}^{threadcount} lt_i$ ▷ Calculate the cumulative sum.
  $gt_0, gt_1, gt_2, ..., gt_{sum} \leftarrow 0, gt_0, gt_0 + gt_1, ..., \sum_{i=0}^{threadcount} gt_i$
  $lfrom \leftarrow start + lt_{threadid}$ ▷ Allocate locations for threads.
  $gfrom \leftarrow end - gt_{threadid+1}$
  $i \leftarrow start + threadid$ ▷ Go through the data again, storing everything at its correct position.
  for $i < end, i \leftarrow i + threadcount$ **do**
    if $s_i < pivot$ **then**
      $\neg s_{lfrom} \leftarrow s_i$
      $lfrom \leftarrow lfrom + 1$
    if $s_i > pivot$ **then**
      $\neg s_{gfrom} \leftarrow s_i$
      $gfrom \leftarrow gfrom + 1$
  $i \leftarrow start + lt_{sum} + threadid$ ▷ Store the pivot value between the new sequences.
  for $i < end - gt_{sum}, i \leftarrow i + threadcount$ **do**
    $d_i \leftarrow pivot$
  $newseq1 \leftarrow \neg s_{start \to start+lt_{sum}}$
  $newseq2 \leftarrow \neg s_{end-gt_{sum} \to end}$
  $longseq, shortseq \leftarrow max(newseq1, newseq2), min(newseq1, newseq2)$
  if $||longseq||$ <MINSIZE **then** ▷ If the sequence is shorter than MINSIZE
    $altsort(longseq, d)$ ▷ sort it using an alternative sort and place result in d.
  else
    push $longseq$ **on** $workstack$
  if $||shortseq||$ <MINSIZE **then**
    $altsort(shortseq, d)$
  else
    push $shortseq$ **on** $workstack$

of elements that are smaller than the pivot. This information is stored in two arrays in the shared local memory; with each thread in a half warp (a warp being 32 consecutive threads that are always scheduled together) accessing different memory banks to increase performance.

The data is read in chunks of T words, where T is the number of threads in each thread block. The threads read consecutive words so that the reads coalesce as much as possible.

Space Allocation Once we have gone through all the assigned data, we calculate the cumulative sum of the two arrays. We then use the atomic FAA-function to calculate the cumulative sum for all blocks that have completed so far. This information is used to give each thread a place to store its result, as can be seen in Figure 2.3 (Step c-f).

FAA is as of the time of writing not available on all GPUs. An alternative if one wants to run the algorithm on the older, high-end graphics processors, is to divide the kernel up into two kernels and do the block cumulative sum on the CPU instead. This would make the code more generic, but also slightly slower on new hardware.

Second Pass Using the cumulative sum, each thread knows where to write elements that are greater or smaller than the pivot. Each block goes through its assigned data again and writes it to the correct position in the current auxiliary array. It then fills the gap between the elements that are greater or smaller than the pivot with the pivot value. We now know that the pivot values are in their correct final position, so there is no need to sort them anymore. They are therefore not included in any of the newly created subsequences.

Are We Done? If the subsequences that arise from the partitioning are longer than $minlength$, they will be partitioned again in the next iteration, provided we do not already have more than $maxseq$ sequences. If we do, the next phase begins. Otherwise we go through another iteration. (See Algorithm 1).

The Second Phase

When we have acquired enough independent subsequences, there is no longer any need for synchronization between blocks. Because of this, the entire phase two can be run on the GPU entirely. There is however still the need for synchronization between threads, which means that we will use the same method as in phase one to partition the data. That is, we will count the number of elements that are greater or smaller than the pivot, do a cumulative sum so that each thread has its own location to write to and then move all elements to their correct position in the auxiliary buffer.

Stack To minimize the amount of fast local memory used, there being a very limited supply of it, we always recurse on the smallest subsequence. By doing that, Hoare has shown [Hoa62] that the maximum recursive depth can never go below $\log_2(n)$. We use an explicit stack as suggested by Hoare and implemented by Sedgewick in [Sed78], always storing the smallest subsequence at the top.

Overhead When a subsequence's size goes below a certain threshold, we use an alternative sorting method on it. This was suggested by Hoare since the overhead of Quicksort gets too big when sorting small sequences of data. When a subsequence is small enough to be sorted entirely in the fast local memory, we could use any sorting method that can be made to sort in-place, does not require much expensive thread synchronization and performs well when the number of threads approaches the length of the sequence to be sorted. See Section 2.5.2 for more information about algorithm used.

Cumulative sum

When calculating the cumulative sum, it would be possible to use a simple sequential implementation, since the sequences are so short (≤ 512). But it is calculated so often that every performance increase counts, so we decided to use the parallel cumulative sum implementation described in [HSO07] which is based on [Ble93]. Their implementation was an *exclusive* cumulative sum so we had to modify it to include the total sum. We also modified it so that it

accumulated two arrays at the same time. By using this method, the speed of the calculation of the cumulative sum was increased by 20% compared to using a sequential implementation.

Another alternative would have been to let each thread use FAA to create a cumulative sum, but that would have been way too expensive, since all the threads would have been writing to the same variable, leading to all additions being serialized. Measurements done using 128 threads show that it would be more than ten times slower than the method we decided to use.

2.4 Complexity

THEOREM 2.1. *The average time complexity for GPU-Quicksort on a CRCW PRAM is* $O(\frac{n}{p}\log(n))$.

Proof. For the analysis we combine phase one and two since there is no difference between them from a complexity perspective. We assume a p-process arbitrary CRCW PRAM [Jaj92]. Each partition iteration requires going through the data, calculating the cumulative sum and going through the data again writing the result to its correct position. Going through the data twice takes $O(\frac{n}{p})$ steps, where n is the length of the sequence to sort and p is the number of processors.

The accumulate function has a time complexity of $O(\log(T))$, where T is the number of threads per thread block [Ble93]. Since T does not vary with n or p, it is a constant cost. The alternative sort only needs to sort sequences that are smaller than q, where q is dependent on the amount of available shared memory on the graphics processor. This means that the worst case complexity of the alternative sort is not dependent on n or p and is thus constant.

Assuming that all elements are equally likely to be picked as a pivot, we get an average running time of

$$T(n) = \begin{cases} O(\frac{n}{p}) + \frac{2}{n}\sum_{i=0}^{n-1} T(i) & n > q, \\ O(1) & n \leq q. \end{cases}$$

Assuming that $q \ll n$ we can set $q = 1$ which gives us the standard Quicksort recurrence relation, which is proved to be $O(\frac{n}{p}\log(n))$. □

THEOREM 2.2. *The space complexity for GPU-Quicksort is $2n + c$, where c is a constant.*

Proof. Phase one is bounded in that it only needs to keep track of a maximum of $maxseq$ subsequences. Phase two always recurses on the smaller sequence, giving a bound of $\log_2(keysize)$ subsequences that needs to be stored. $maxseq$ and $\log_2(keysize)$ do not depend on the size of the sequence and can thus be seen as constants. The memory complexity then becomes $2n + c$, the size of the sequence to be sorted plus an equally large auxiliary buffer and a constant needed for the bookkeeping of $maxseq$ and $B\log_2(keysize)$ sequences, where B is the amount of thread blocks used. □

2.5 Experimental Evaluation

2.5.1 Hardware

We ran the experiments on a dual-processor dual-core AMD Opteron 1.8 GHz machine. Two different graphics processors were used, the low-end NVIDIA 8600GTS 256 MiB with 4 multiprocessors and the high-end NVIDIA 8800GTX 768 MiB with 16 multiprocessors, each multiprocessor having 8 processors each.

The 8800GTX provides no support for atomic operations. Because of this, we used an implementation of the algorithm that exits to the CPU for block-synchronization, instead of using FAA.

2.5.2 Algorithms used

We compared GPU-Quicksort to the following state-of-the-art GPU sorting algorithms:

GPUSort Uses bitonic merge sort [GRHM05].

Radix-Merge Uses radix sort to sort blocks that are then merged [HSO07].

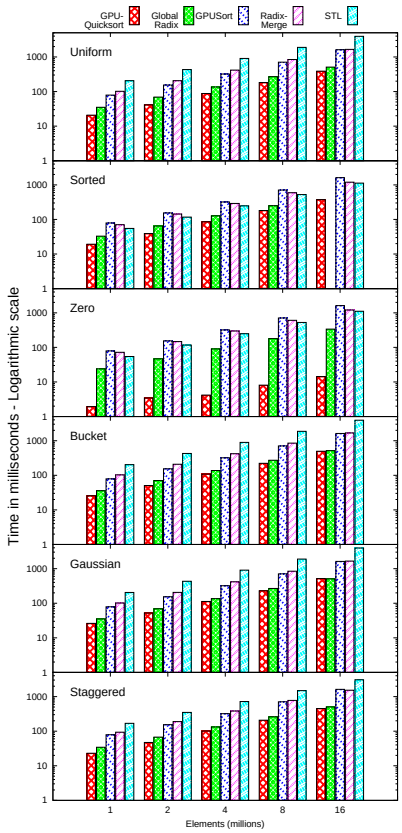


Figure 2.4: *Results on the 8800GTX*

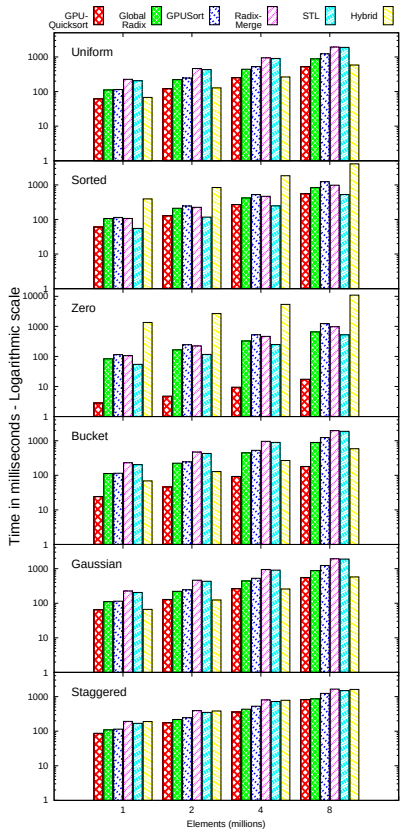


Figure 2.5: *Results on the 8600GTS*

Global Radix Uses radix sort on the entire sequence [SHZO07].

Hybridsort Splits the data with a bucket sort and uses merge sort on the resulting blocks [SA07].

STL-Introsort This is the Introsort implementation found in the C++ Standard Library [Mus97]. Introsort is based on Quicksort, but switches to heapsort when the recursion depth gets too large. Since it is highly dependent on the computer system and compiler used, we only included it to give a hint as to what could be gained by sorting on the GPU instead of on the CPU.

We could not find an implementation of the Quicksort algorithm used by Sengupta et al., but they claim in their paper that it took over 2 seconds to sort 4M uniformly distributed elements on an 8800GTX, which makes it much slower than STL sort [SHZO07].

We only measured the actual sorting phase, we did not include in the result the time it took to setup the data structures and to transfer the data on and off the graphics memory. The reason for this is the different methods used to transfer data which wouldn't give a fair comparison between the GPU-based algorithms. Transfer times are also irrelevant if the data to be sorted is already available on the GPU. Because of those reasons, this way of measuring has become a standard in the literature.

We used different pivot selection schemes for the two phases. In the first phase we took the average of the minimum and maximum element in the sequence and in the second we picked the median of the first, middle and last element as the pivot, a method suggested by Singleton [Sin69].

We used the more computationally expensive method in the first phase to try to have a more even size of the subsequences that are assigned to each thread block in the next phase. This method works perfectly well with numbers, but for generic key types it has to be replaced with e.g. picking a random element or using the same method as in phase two.

The source code of GPU-Quicksort is available for non-commercial use [CT07].

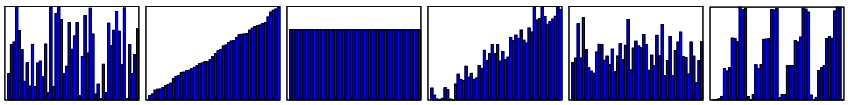
2.5.3 Input Distributions

For benchmarking we used the following distributions which are commonly used yardsticks in the literature to compare the performance of different sorting algorithms [HBJ98]. The source of the random uniform values is the Mersenne Twister [MN98].

Uniform Values are picked randomly from $0 - 2^{31}$.

Sorted Sorted uniformly distributed values.

Zero A constant value is used. The actual value is picked at random.



(a) *Uniform* **(b)** *Sorted* **(c)** *Zero* **(d)** *Bucket* **(e)** *Gaussian* **(f)** *Staggered*

Figure 2.6: *Visualization of sequences generated with different distributions.*

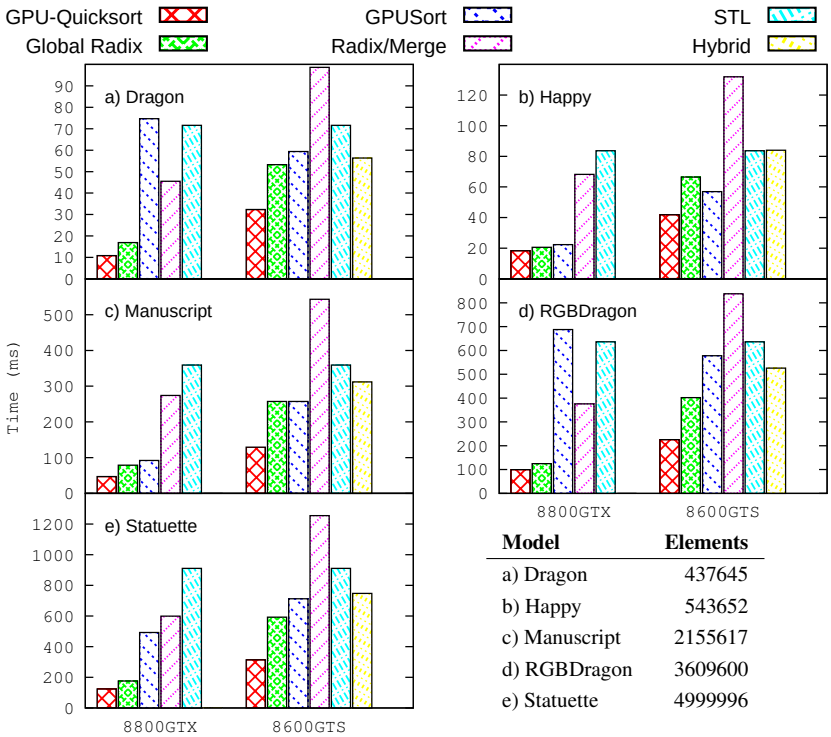
Bucket The data set is divided into p blocks, where $p \in \mathbb{Z}^+$, which are then each divided into p sections. Section 1 in each block contains randomly selected values between 0 and $\frac{2^{31}}{p} - 1$. Section 2 contains values between $\frac{2^{31}}{p}$ and $\frac{2^{32}}{p} - 1$ and so on.

Gaussian The Gaussian distribution is created by always taking the average of four randomly picked values from the uniform distribution.

Staggered The data set is divided into p blocks, where $p \in \mathbb{Z}^+$. The staggered distribution is then created by assigning values for block i, where $i \leq \lfloor \frac{p}{2} \rfloor$, so that they all lie between $(2i+1)\frac{2^{31}}{p}$ and $(2i+2)\frac{2^{31}}{p} - 1$. For blocks where $i > \lfloor \frac{p}{2} \rfloor$, the values all lie between $(2i-p)\frac{2^{31}}{p}$ and $(2i-p+1)\frac{2^{31}}{p} - 1$. We decided to use a p value of 128.

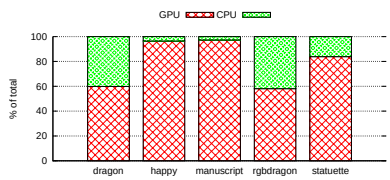
The results presented in Figure 2.4 and 2.5 are based on experiments sorting sequences of integers. We have done experiments using floats instead, but found no difference in performance.

To get an understanding of how GPU-Quicksort performs when faced with real-world data we have evaluated it on the task of visibility ordering, i.e. sorting a set of vertices in a model according to their distance from a point in 3D-space. The 3D-models have been taken from *The Stanford 3D Scanning Repository* which is a resource commonly used in the literature [Sta]. By calculating the distance from the viewer (in the experiments we placed the viewer at origin (0,0,0)), we get a set of floats that needs to be sorted to know in which order the vertices and accompanying faces should be drawn. Figure 2.7 shows the result from these experiments on the two different graphics processors. The size of the models ranges from 5×10^6 elements to 5×10^7 elements. The hybrid algorithm this time comes in two versions, the normal one and one where we first randomize the sequence as suggested in the paper [SA07].

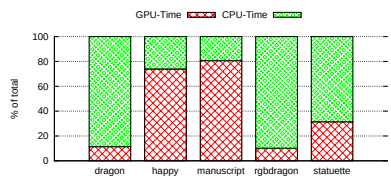


Model	Elements
a) Dragon	437645
b) Happy	543652
c) Manuscript	2155617
d) RGBDragon	3609600
e) Statuette	4999996

Figure 2.7: *Shows the time taken to sort the distances between origin (0,0,0) and each vertice in five different 3D-models. a) Dragon, b) Happy, c) Manuscript, d) RGBDragon and e) Statuette. Also shown is a table with the number of vertices in each figure, i.e. the size of the sequence to sort.*



(a) *Percentage of data sorted on the GPU vs CPU when using GPUSort.*



(b) *Percentage of time spent sorting on the GPU vs CPU when using GPUSort.*

Figure 2.8: *GPUSort on 8600GTS.*

Phase	Registers	Shared Memory (32-bit words)
Phase one	14	$4T + 14$
Phase two	14	$\max(2T, S) + 98$

Table 2.1: *Registers and shared memory per block required by each phase. T is the number of threads per block and S is the minimum sequence sorted by Quicksort.*

optp(s,k,m)	8800GTX	8600GTS
Threads per Block	$k = \frac{1.172}{10000}, m = 53$	$k = 0, m = 64$
Max Blocks in Phase One	$k = \frac{3.7480}{10000}, m = 476$	$k = \frac{9.5160}{10000}, m = 203$
Min Sequence to Sort	$k = \frac{4.6850}{10000}, m = 211$	$k = \frac{3.2160}{10000}, m = 203$

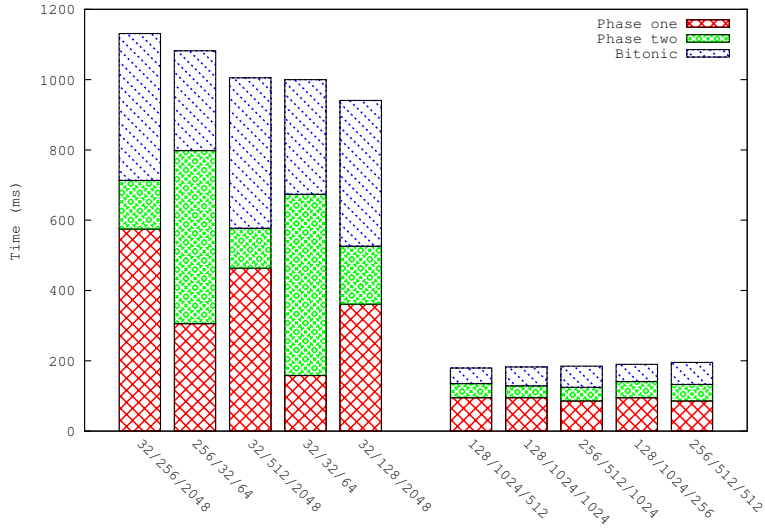
Table 2.2: *Shows the constants used to select suitable parameters for a given sequence length* s.

GPUSort can only sort sequences that have a length which is a power of two, due to the use of a bitonic sorting network. In Figure 2.8a we have visualized the amount of data that is sorted on the GPU versus the CPU. In Figure 2.8b we also show the relative amount of time spent on the GPU versus the CPU.

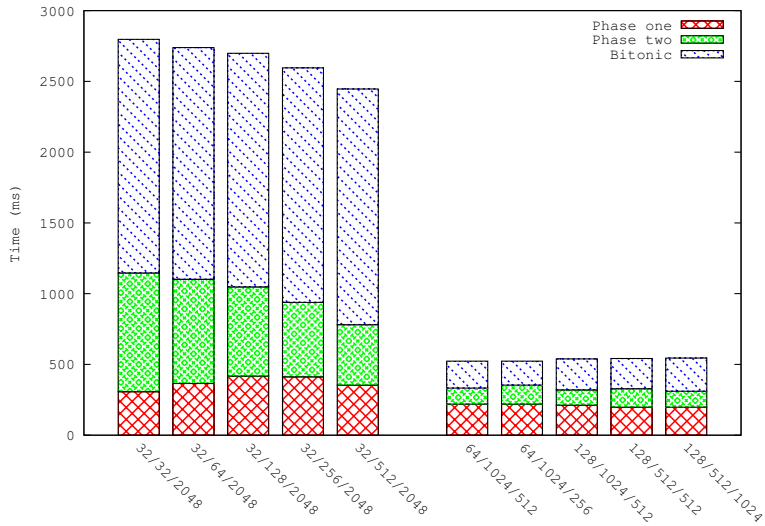
Table 2.1 shows the number of registers and amount of shared memory needed by each phase. The higher the number of registers used the fewer the threads that can be used per block and the higher the amount of shared memory used the fewer number of blocks can be run concurrently.

The GPU-Quicksort algorithms can be seen as having three parameters, the maximum number of subsequences created in the first phase, *maxseq*, the number of threads per block, *threads*, and the minimum size of sequence to sort with Quicksort before switching to the alternative sort, *sbsize*. In Figure 2.9 we have tried several different combinations of these parameters on the task of sorting a uniformly distributed sequence with 8 million elements. The figure shows the five best and the five worst results divided into how much time each phase takes.

Figure 2.10 shows how the performance changes when we vary one parameter and then pick the best, the worst and the average result among all other combinations of the two other parameters.

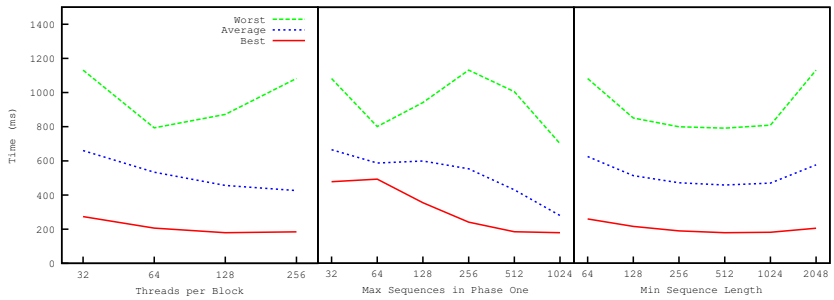


(a) *8800GTX.*

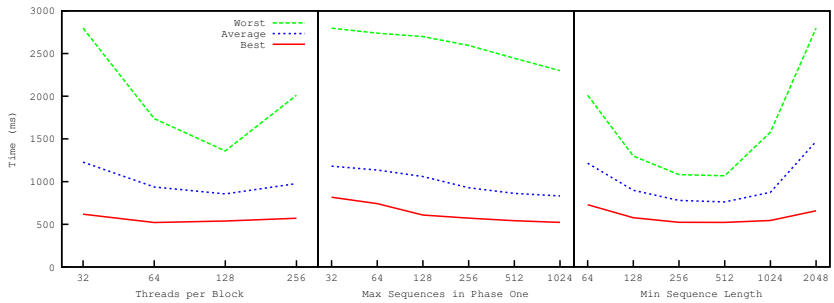


(b) *8600GTS.*

Figure 2.9: *The five best and worst combinations of* threads per block/maximum number of subsequences created in phase one/minimum size of sequence to sort using Quicksort.



(a) *8800GTX.*



(b) *8600GTS.*

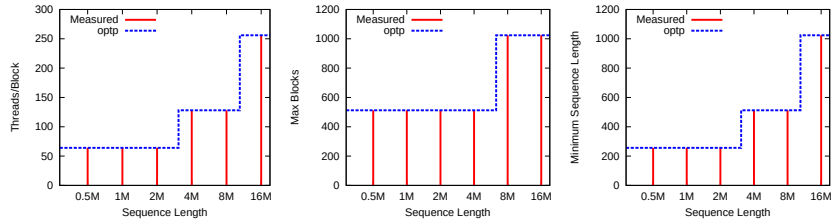
Figure 2.10: *Varying one parameter, picking the best, worst and average result from all possible combination of the others parameters.*

The optimal selection of these parameters varies with the size of the sequence. Figure 2.11 shows how the values that gives the best result changes when we run larger sequences. All variables seems to increase with the size of the sequence.

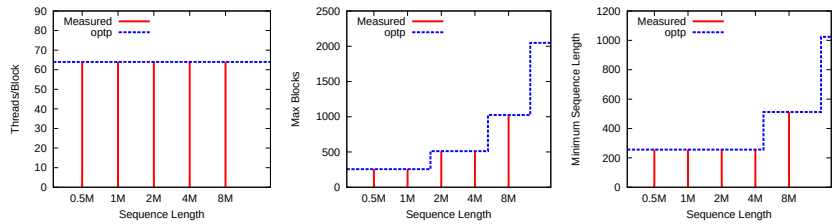
To get the best parameters for any given sequence length we use a linear function for each parameter to calculate its value.

$$optp(s,k,m) := 2^{\lfloor \log_2(sk+m)+0.5 \rfloor}$$

The parameters for this function are presented in Table 2.2. They were calculated by doing a linear regression using the measured values presented in Figure 2.11.



(a) *8800GTX.*



(b) *8600GTS.*

Figure 2.11: *The parameters vary with the length of the sequence to sort.*

2.5.4 Discussion

In this section we discuss GPU sorting in the light of the experimental result.

Since sorting on GPUs has received a lot of attention it might be reasonable to start with the following question; **is there really a point in sorting on the GPU?** If we take a look at the radix-merge sort in Figure 2.5 we see that it performs comparable to the CPU reference implementation. Considering that we can run the algorithm concurrently with other operations on the CPU, it makes perfect sense to sort on the GPU.

If we look at the other algorithms we see that they perform at twice the speed or more compared to Introsort, the CPU reference. On the higher-end GPU in Figure 2.4, the difference in speed can be up to 10 times the speed of the reference! Even if one includes the time it takes to transfer data back and forth to the GPU, less than 8ms per 1M element, it is still a massive performance gain that can be made by sorting on the GPU. Clearly there are good reasons to use the GPU as a general purpose co-processor.

But why should one use Quicksort? Quicksort has a worst case scenario complexity of $O(n^2)$, but in practice, and on average when using a random pivot, it tends to be close to $O(n \log(n))$, which is the lower bound for comparison sorts. In all our experiments GPU-Quicksort has shown the best performance or been among the best. There was no distribution that caused problems to the performance of GPU-Quicksort. As can be seen when comparing the performance on the two GPUs, GPU-Quicksort shows a speedup by around 3 times on the higher-end GPU. The higher-end GPU has a memory bandwidth that is 2.7 times higher but has a slightly slower clock speed, indicating that the algorithm is bandwidth bound and not computation bound, which was the case with the Quicksort in the paper by Sengupta et al. [SHZO07].

Is it better than radix? On the CPU, Quicksort is normally seen as a faster algorithm as it can potentially pick better pivot points and does not need an extra check to determine when the sequence is fully sorted. The time complexity of radix sort is $O(n)$, but that hides a potentially high constant which is dependent on the key size. Optimizations are possible to lower this constant, such as constantly checking if the sequence has been sorted, but when dealing with longer keys that can be expensive. Quicksort being a comparison sort also means that it is easier to modify it to handle different key types.

Is the hybrid approach better? The hybrid approach uses atomic instructions that were only available on the 8600GTS. We can see that it performs very well on the uniform, bucket and gaussian distribution, but it loses speed on the staggered distributions and becomes immensely slow on the zero and sorted distribution. In the paper by Sintorn and Assarsson they state that the algorithm drops in performance when faced with already sorted data, so they suggest randomizing the data first [SA07]. This however lowers the performance and wouldn't affect the result in the zero distribution.

How are the algorithms affected by the higher-end GPU? GPUSort does not increase as much in performance as the other algorithms when run on the higher-end GPU. This is an indication that the algorithm is more computationally bound than the other algorithms. It goes from being much faster than the slow radix-merge to perform on par and even a bit slower than it.

The global radix sort showed a three time speed improvement, as did GPU-Quicksort. As mentioned earlier, this shows that the algorithms most likely are bandwidth bound.

How are the algorithms affected by the different distributions? All algorithms showed about the same performance on the uniform, bucket and gaussian distributions. GPUSort takes the same amount of time for all of the distribution since it is a sorting network, which means it always performs the same number of operations regardless of the distribution.

The zero distribution, which can be seen as an already sorted sequence, affected the algorithms to different extent. The STL reference implementation increased dramatically in performance since its two-way partitioning function always returned even partitions regardless of the pivot chosen. GPUSort performs the same number of operations regardless of the distribution, so there was no change there. The hybrid sort placed all elements in the same bucket which caused it to show much worse performance than the others. GPU-Quicksort shows the best performance since it will pick the only value that is available in the distribution as the pivot value, which will then be marked as already sorted. This means that it just has to do two passes through the data and can sort the zero distribution in $O(n)$ time.

On the sorted distribution all algorithms gain in speed except GPUSort and the hybrid. The CPU reference becomes faster than GPUSort and radix-merge on the high-end graphics processor and is actually the fastest when compared to the algorithms run on the low-end graphics processor.

On the real-world data experiments, Figure 2.7, we can see that GPU-Quicksort performs well on all models. Other than that the relative differences stay the same as they did with the artificial distributions. One interesting thing though is the inconsistency in GPUSort. It is much faster on the larger manuscript and statuette models than it is on the smaller dragon models. This has nothing to do with the distribution since bitonic sort is not affected by it, so this is purely due to the fact that GPUSort needs to sort part of the data on the CPU since sequences needs to have a length that is a power of two to be sorted with bitonic sort. In Figure 2.8a we can see that for the two dragon-models

40% is spent sorting on the CPU instead of on the GPU. Looking at Figure 2.8b we see that this translates to 90% of the actual sorting time. This explains the strange variations in the experiments.

2.6 Conclusions

In this paper we present GPU-Quicksort, a parallel Quicksort algorithm designed to take advantage of the high bandwidth of GPUs by minimizing the amount of bookkeeping and inter-thread synchronization needed.

The bookkeeping is minimized by constraining all thread blocks to work with only one (or part of a) sequence of data at a time. This way pivot values do not need to be distributed to all thread blocks and thus no extra information needs to be written to the global memory.

The two-pass design of GPU-Quicksort has been introduced to keep the inter-thread synchronization low. First the algorithm traverses the sequence to sort, counting the number of elements that each thread sees that have a higher (or lower) value than the pivot. By calculating a cumulative sum of these sums, in the second phase, each thread will know where to write its assigned elements without any extra synchronization. The small amount of inter-block synchronization that is required between the two passes of the algorithm can be reduced further by taking advantage of the atomic synchronization primitives that are available on newer hardware.

A previous implementation of Quicksort for GPUs by Sengupta et al. turned out not to be competitive enough in comparison to radix sort or even CPU based sorting algorithms [SHZO07]. According to the authors this was due to it being more dependent on the processor speed than on the bandwidth.

In our experiments we compared GPU-Quicksort with some of the fastest known sorting algorithms for GPUs, as well as with the C++ Standard Library sorting algorithm, Introsort, for reference. We used several input distributions and two different graphics processors, the low-end 8600GTS with 32 cores and the high-end 8800GTX with 128 cores, both from NVIDIA. What we could

observe was that GPU-Quicksort performed better on all distributions on the high-end processor and on par with or better on the low-end processor.

A significant conclusion, we think, that can be drawn from this work, is that Quicksort is a practical alternative for sorting large quantities of data on graphics processors.

Acknowledgments

We would like to thank Georgios Georgiadis and Marina Papatriantafilou for their valuable comments during the writing of this paper. We would also like to thank Ulf Assarsson and Erik Sintorn for insightful discussions regarding CUDA and for providing us with the source code to their hybrid sort. Last but not least, we would like to thank the anonymous reviewers for their comments that helped us improve the presentation of the algorithm significantly.

Bibliography

[Ble93] Guy E. Blelloch. Prefix Sums and Their Applications. In John H. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993.

[BN89] Gianfranco Bilardi and Alexandru Nicolau. Adaptive Bitonic Sorting. An Optimal Parallel Algorithm for Shared Memory Machines. *SIAM Journal on Computing*, 18(2):216–228, 1989.

[CT07] Daniel Cederman and Philippas Tsigas. GPU Quicksort Library. `www.cs.chalmers.se/~dcs/gpuqsortdcs.html`, December 2007.

[DPRS89] Martin Dowd, Yehoshua Perl, Larry Rudolph, and Michael Saks. The Periodic Balanced Sorting Network. *Journal of the ACM*, 36(4):738–757, 1989.

[ED82] D. J. Evans and R. C. Dunbar. The Parallel Quicksort Algorithm Part 1 - Run Time Analysis. *International Journal of Computer Mathematics*, 12:19–55, 1982.

[GGKM06] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 325–336, 2006.

[GRHM05] N. Govindaraju, N. Raghuvanshi, M. Henson, and D. Manocha. A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors. Technical report, University of North Carolina-Chapel Hill, 2005.

[GRM05] Naga K. Govindaraju, Nikunj Raghuvanshi, and Dinesh Manocha. Fast and Approximate Stream Mining of Quantiles and Frequencies Using Graphics Processors. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 611–622, 2005.

[Gro08] Khronos Group. OpenCL (Open Computing Language). `http://www.khronos.org/opencl/`, 2008.

[GZ06] Alexander Greß and Gabriel Zachmann. GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, 2006.

[HBJ98] David R. Helman, David A. Bader, and Joseph JáJá. A Randomized Parallel Sorting Algorithm with an Experimental Study. *Journal of Parallel and Distributed Computing*, 52(1):1–23, 1998.

[HNR90] P. Heidelberger, A. Norton, and John T. Robinson. Parallel Quicksort Using Fetch-And-Add. *IEEE Transactions on Computers*, 39(1):133–138, 1990.

[Hoa61] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.

[Hoa62] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(4):10–15, 1962.

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel Prefix Sum (Scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley, August 2007.

[Jaj92] Joseph Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[KDR+00] Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucek Khailany. Efficient Conditional Operations for Data-parallel Architectures. In *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*, pages 159–170, 2000.

[KSW04] Peter Kipfer, Mark Segal, and Rüdiger Westermann. UberFlow: A GPU-based Particle Engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 115–122, 2004.

[KW05] Peter Kipfer and Rüdiger Westermann. Improved GPU Sorting. In Matt Pharr, editor, *GPUGems 2*, chapter 46, pages 733–746. Addison-Wesley, 2005.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: a 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[Mus97] David R. Musser. Introspective Sorting and Selection Algorithms. *Software - Practice and Experience*, 27(8):983–993, 1997.

[PDC+03] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon Mapping on Programmable Graphics Hardware. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware*, pages 41–50, 2003.

[SA07] Erik Sintorn and Ulf Assarsson. Fast Parallel GPU-Sorting Using a Hybrid Algorithm. In *Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[Sed78] Robert Sedgewick. Implementing Quicksort Programs. *Communications of the ACM*, 21(10):847–857, 1978.

[SHZO07] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan Primitives for GPU Computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 97–106, 2007.

[Sin69] Richard C. Singleton. Algorithm 347: an Efficient Algorithm for Sorting with Minimal Storage. *Communications of the ACM*, 12(3):185–186, 1969.

[Sta] Stanford. The Stanford 3D Scanning Repository.

[TZ03] Philippas Tsigas and Yi Zhang. A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000. In *Proceedings of the 11th Euromicro Conference on Parallel Distributed and Network-based Processing*, pages 372–381, 2003.

PAPER II

Daniel Cederman, Philippas Tsigas
On Dynamic Load Balancing on Graphics Processors
In the Proceedings of the 11th Graphics Hardware (GH 2008)
pages 57 - 64, ACM press 2008.

3

On Dynamic Load Balancing on Graphics Processors

To get maximum performance on the many-core graphics processors it is important to have an even balance of the workload so that all processing units contribute equally to the task at hand. This can be hard to achieve when the cost of a task is not known beforehand and when new sub-tasks are created dynamically during execution. With the recent advent of scatter operations and atomic hardware primitives it is now possible to bring some of the more elaborate dynamic load balancing schemes from the conventional SMP systems domain to the graphics processor domain.

We have compared four different dynamic load balancing methods to see which one is most suited to the highly parallel world of graphics processors.

Three of these methods were lock-free and one was lock-based. We evaluated them on the task of creating an octree partitioning of a set of particles. The experiments showed that synchronization can be very expensive and that new methods that take more advantage of the graphics processors features and capabilities might be required. They also showed that lock-free methods achieves better performance than blocking and that they can be made to scale with increased numbers of processing units.

3.1 Introduction

Today's graphics processors have ventured from the multi-core to the many-core domain and, with many problems in the graphics area being of the so called embarrassingly parallel kind, there is no question that the number of processing units will continue to increase.

To be able to take advantage of this parallelism in general purpose computing, it is imperative that the problem to be solved can be divided into sufficiently fine-grained tasks to allow the performance to scale when new processors arrive with more processing units. However, the more fine-grained a task set gets, the higher the cost of the required synchronization becomes.

Some problems are easily divided into fine-grained tasks with similar processing time, such as for example the N-body problem [NHP07]. But with other problems this information can be hard to predict and new tasks might be created dynamically during execution. In these cases dynamic load balancing schemes are needed which can adapt to shifts in work load during runtime and redistribute tasks evenly to the processing units.

Several popular load balancing schemes have been hard to implement efficiently on graphics processors due to lack of hardware support, but this has changed with the advent of scatter operations and atomic hardware primitives such as Compare-And-Swap. It is now possible to design more advanced concurrent data structures and bring some of the more elaborate dynamic load balancing schemes from the conventional SMP systems domain to the graphics processor domain.

The load balancing in these schemes is achieved by having a shared data object that stores all tasks created before and under execution. When a processing unit has finished its work it can get a new task from the shared data object. As long as the tasks are sufficiently fine-grained the work load will be balanced between processing units.

The methods used for synchronizing the memory accesses to these shared data objects can be divided into two categories, *blocking* and *non-blocking*. Blocking methods use mutual exclusion primitives such as locks to only allow one processing unit at a time to access the object. This is a pessimistic conflict control that assumes conflicts even when there are none.

Non-blocking methods on the other hand employ an optimistic conflict control approach allowing several processing units to access the shared data object at the same time and suffering delays because of retries only when there is an actual conflict. This feature allows non-blocking algorithms to scale much better when the number of processing units increases. Section 3.3 discusses more about the differences between the two methods.

In the paper we compare four different methods of dynamic load balancing:

Centralized Blocking Task Queue Tasks are stored in a queue using mutual exclusion.

Centralized Non-blocking Task Queue Tasks are stored in a (lock-free) non-blocking queue.

Centralized Static Task List Tasks are stored in a static list.

Task Stealing Each processing unit has a local double ended queue where it stores new tasks. Tasks can be stolen from other processing units if required.

The first method is lock-based while the other three are non-blocking ones. The schemes are evaluated on the task of creating an octree partitioning of a set of particles. An octree is a tree-based spatial data structure that repeatedly divides the space by half in each direction to form eight octants. The fact that

there is no information beforehand on how deep each branch in the tree will be, makes this a suitable problem for dynamic load balancing.

3.1.1 Related Work

Load balancing is a very basic problem and as such there has been a lot of work done in the area. In this subsection we present a small set of papers that we deem most relevant to our work.

Korch et al. have made a comparison between different dynamic balancing schemes on the radiosity problem [KR03]. Heirich and Arvo have compared static and dynamic load balancing methods for ray-tracing and found static methods to be inadequate [HA98]. Foley and Sugerman have implemented an efficient kd-tree traversal algorithm for graphics processors and point out that load balancing is one of the central issues for any highly parallel ray-tracer [FS05].

When it comes to using task stealing for load balancing, Blumofe and Leiserson gave "the first provably good task stealing scheduler for multithreaded computations with dependencies" [BL94]. Arora et al. have presented an efficient lock-free method for task stealing where no synchronization is required for local access when the local stack has more than one elements [ABP98]. Soares et al have used task stealing as load balancing for voxel carving with good performance [SMRR07].

In the following section the system model is presented. In Section 3.3 the need for non-blocking algorithms is discussed. Section 3.4 has an overview of the load balancing methods compared in the paper and Section 3.5 describes the octree partitioning task used for evaluation. In section 3.6 we describe the test settings and discuss the result.

3.2 The System Model

All the load balancing methods in this work have been implemented using CUDA, a compiler and run-time from NVIDIA that can compile normal C code

into binary or byte-code that can be executed on CUDA-enabled graphics processors.

General Architecture The graphics processors consist of up to 16 multiprocessors each, which can perform SIMD (Single Instruction, Multiple Data) instructions on 8 memory positions at a time. Each multiprocessor has 16 kB of a very fast local memory that allows information to be communicated between threads running on the same multiprocessor.

Memory Access Each multiprocessor has access to the large, but relatively slow, global memory. It is possible to speed up access to the memory by arranging memory accesses in such a way so that the graphics processor can coalesce them into one big read or write operation. This is done by letting threads access consecutive memory locations with the first location being a multiple of 16 times the word size read or written.

The NVIDIA 8600GTS and newer graphics processors support atomic operations such as CAS (Compare-And-Swap) and FAA (Fetch-And-Add) when accessing the memory, which can be used to implement efficient parallel data structures.

Scheduling The graphics processor uses a massive number of threads to hide memory latency instead of using a cache memory. These threads are divided into *thread blocks* of equal size where all threads in a specific thread block is assigned to a specific multiprocessor. This allows threads in the same thread block to communicate with each other using the fast local memory and a special hardware barrier function which can be used to synchronize all threads in a block.

Thread blocks are run from start to finish on the same multiprocessor and can't be swapped out for another thread block. If all thread blocks started can't fit on the available multiprocessors they will be run sequentially and will be swapped in as other thread blocks complete. A common scenario is to divide work into small tasks and then create a thread block for each task. When a multiprocessor has finished with one thread block/task, it schedules a new one and can thus achieve a relatively balanced load.

The threads in a thread block are scheduled according to which *warp* they are a part of. A warp consists of 32 threads with consecutive id, such as $0..31$ and $32..63$. The graphics processor tries to execute the threads in a warp using SIMD instructions, so to achieve optimal performance it is important to try to have all threads in a warp perform the same instruction at any given time. Threads in the same warp that perform different operations will be serialized and the multiprocessor will not be used to its full capabilities.

3.3 Synchronization

Synchronization schemes for designing shared data objects can be divided into three categories:

Blocking Uses mutual exclusion to only allow one process at a time to access the object.

Lock-Free Multiple processes can access the object concurrently. At least one operation in a set of concurrent operations finishes in a finite number of its own steps.

Wait-Free Multiple processes can access the object concurrently. Every operation finishes in a finite number of its own steps.

The term non-blocking is also used in order to describe methods that are either lock-free or wait-free.

The standard way of implementing shared data objects is often by the use of basic synchronization constructs such as locks and semaphores. Such blocking shared data objects that rely on mutual exclusion are often easier to design than their non-blocking counterpart, but a lot of time is spent in the actual synchronization, due to busy waiting and convoying. Busy waiting occurs when multiple processes repeatedly checks if, for example, a lock has been released or not, wasting bandwidth in the process. This lock contention can be very expensive.

The convoying problem occurs when a process (or warp) is preempted and is unable to release the lock quick enough. This causes other processes to have to wait longer than necessary, potentially slowing the whole program down.

Non-blocking shared data objects, on the other hand, allows access from several processes at the same time without using mutual exclusion. So since a process can't block another process they avoid convoys and lock contention. Such objects also offer higher fault-tolerance since one process can always continue, whereas in a blocking scenario, if the process holding the lock would crash, the data structure would be locked permanently for all other processes. A non-blocking solution also eliminates the risk of deadlocks, cases where two or more processes circularly waits for locks held by the other.

3.4 Load Balancing Methods

This section gives an overview of the different load balancing methods we have compared in this paper.

3.4.1 Static Task List

The default method for load balancing used in CUDA is to divide the data that is to be processed into a list of blocks or tasks. Each processing unit then takes out one task from the list and executes it. When the list is empty all processing units stop and control is returned to the CPU.

This is a lock-free method and it is excellent when the work can be easily divided into chunks of similar processing time, but it needs to be improved upon when this information is not known beforehand. Any new tasks that are created during execution will have to wait until all the statically assigned tasks are done, or be processed by the thread block that created them, which could lead to an unbalanced workload on the multiprocessors.

The method, as implemented in this work, consists of two steps that are performed iteratively. The only data structures required are two arrays containing tasks to be processed and a tail pointer. One of the arrays is called the in-array

and only allows read operations while the other, called the out-array, only allows write operations.

In the first step of the first iteration the in-array contains all the initial tasks. For each task in the array a thread block is started. Each thread block then reads task i, where i is the thread block ID. Since no writing is allowed to this array, there is no need for any synchronization when reading.

If any new task is created by a thread block while performing its assigned task, it is added to the out-array. This is done by incrementing the tail pointer using the atomic FAA-instruction. FAA returns the value of a variable and increments it by a specified number atomically. Using this instruction the tail pointer can be moved safely so that multiple thread blocks can write to the array concurrently.

The first step is over when all tasks in the in-array has been executed. In the second step the out-array is checked to see if it is empty or not. If it is empty the work is completed. If not, the pointers to the out- and in-array are switched so that they change roles. Then a new thread block for each of the items in the new in-array is started and this process is repeated until the out-array is empty. The algorithm is described in pseudo-code in Algorithm 4.

The size of the arrays needs to be big enough to accommodate the maximum number of tasks that can be created in a given iteration.

3.4.2 Blocking Dynamic Task Queue

In order to be able to add new tasks during runtime we designed a parallel dynamic task queue that thread blocks can use to announce and acquire new tasks.

As several thread blocks might try to access the queue simultaneously it is protected by a lock so that only one thread block can access the queue at any given time. This is a very easy and standard way to implement a shared queue, but it lowers the available parallelism since only one thread block can access the queue at a time, even if there is no conflict between them.

Algorithm 4 Static Task List Pseudocode.

```
function DEQUEUE(q, id)
    return q.in[id]
function ENQUEUE(q, task)
    localtail ← atomicAdd(&q.tail, 1)
    q.out[localtail] = task
function NEWTASKCNT(q)
    q.in, q.out, oldtail, q.tail ← q.out, q.in, q.tail, 0
    return oldtail
procedure MAIN(task_init)
    q.in, q.out ← newarray(maxsize), newarray(maxsize)
    q.tail ← 0
    enqueue(q, task_init)
    blockcnt ← newtaskcnt(q)
    while blockcnt ≠ 0 do
        run blockcnt blocks in parallel
            t ← dequeue(q, TB_id)
            subtasks ← doWork(t)
            for each nt in subtasks do
                enqueue(q, nt)
        blocks ← newtaskcnt(q)
```

The queue is array-based and uses the atomic CAS (Compare-And-Swap) instruction to set a lock variable to ensure mutual exclusion. When the work is done the lock variable is reset so that another thread block might try to grab it. The algorithm is described in pseudo-code in Algorithm 5.

Since it is array-based the memory required is equal to the number of tasks that can exist at any given time.

Algorithm 5 Blocking Dynamic Task Queue Pseudocode.

```
function DEQUEUE(q)
    while atomicCAS(&q.lock, 0, 1) == 1 do
    if q.beg! = q.end then
        q.beg + +
        result ← q.data[q.beg]
    else
        result ← NIL
    q.lock ← 0
    return result
function ENQUEUE(q, task)
    while atomicCAS(&q.lock, 0, 1) == 1 do
    q.end + +
    q.data[q.end] ← task
    q.lock ← 0
```

3.4.3 Lock-free Dynamic Task Queue

A lock-free implementation of a queue was implemented to avoid the problems that comes with locking and also in order to study the behavior of lock-free synchronization on graphics processors. A lock-free queue guarantees that, without using blocking at all, at least one thread block will always succeed to enqueue or dequeue an item at any given time even in presence of concurrent operations. Since an operation will only have to be repeated at an actual conflict it can deliver much more parallelism.

The implementation is based upon the simple and efficient array-based lock-free queue described in a paper by Zhang and Tsigas [TZ01]. A tail pointer keeps track of the tail of queue and tasks are then added to the queue using CAS. If the CAS-operation fails it must be due to a conflict with another thread block, so the operation is repeated on the new tail until it succeeds. This way at least one thread block is always assured to successfully enqueue an item.

The queue uses lazy updating of the tail and head pointers to lower contention. Instead of changing the head/tail pointer after every enqueue/dequeue operation, something that is done with expensive CAS-operations, it is only updated every x:th time. This increases the time it takes to find the actual head/tail since several queue positions needs to be checked. But by reading consecutive positions in the array, the reads will be coalesced by the hardware into one fast read operation and the extra time can be made lower than the time it takes to try to update the head/tail pointer x times.

Algorithm 6 gives a skeleton of the algorithm without the essential optimizations. For a detailed and correct description, please see the original paper [TZ01]. This method requires just as much memory as the blocking queue.

Algorithm 6 Lock-free Dynamic Task Queue Pseudocode.

```
function DEQUEUE(q)
    oldbeg ← q.beg
    lbeg ← oldbeg
    while task = q.data[lbeg] == NIL do
        lbeg++
    if atomicCAS(&q.data[lbeg], task, NIL)! = task then
        restart
    if lbeg mod x == 0 then
        atomicCAS(&q.beg, oldbeg, lbeg)
    return task
function ENQUEUE(q, task)
    oldend ← q.end
    lend ← oldend
    while q.data[lend]! = NIL do
        lend++
    if atomicCAS(&q.data[lend], NIL, task)! = NIL then
        restart
    if lend mod x == 0 then
        atomicCAS(&q.end, oldend, lend)
```

3.4.4 Task Stealing

Task stealing is a popular load balancing scheme. Each processing unit is given a set of tasks and when it has completed them it tries to steal a task from another processing unit which has not yet completed its assigned tasks. If a unit creates a new task it is added to its own local set of tasks.

One of the most used task stealing methods is the lock-free scheme by Arora et al. [ABP98] with multiple array-based double ended queues (*deques*). This method will be referred to as ABP task stealing in the remainder of this paper.

In this scheme each thread block is assigned its own deque. Tasks are then added and removed from the tail of the deque in a LIFO manner. When the deque is empty the process tries to steal from the head of another process' deque.

Since only the owner of the deque is accessing the tail of the deque there is no need for expensive synchronization when the deque contains more than one element. Several thread blocks might however try to steal at the same time, requiring synchronization, but stealing is assumed to occur less often than a normal local access. The implementation is based on the basic non-blocking version by Arora et al. [ABP98]. The stealing is performed in a global round robin fashion, so thread block i looks at thread block $i + 1$ followed by $i + 2$ and so on.

The memory required by this method depends on the number of thread blocks started. Each thread block needs its own deque which should be able to hold all tasks that are created by that block during its lifetime.

3.5 Octree Partitioning

To evaluate the dynamical load balancing methods described in the previous section, they are applied to the task of creating an octree partitioning of a set of particles [SG91]. An octree is a tree-based spatial data structure that recursively divides the space in each direction, creating eight octants. This is done until an octant contains less than a specific number of particles.

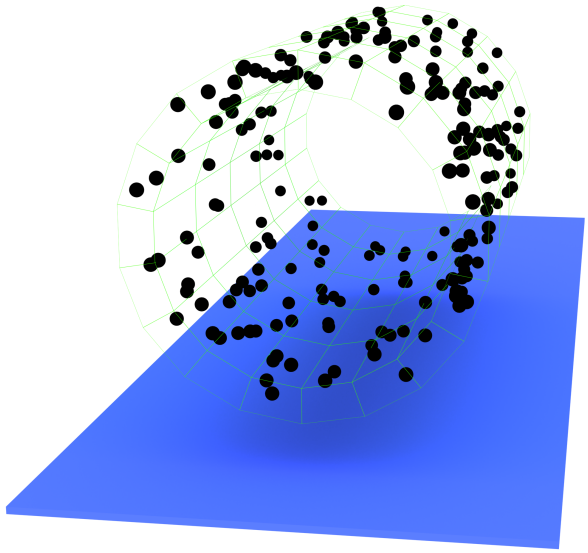


Figure 3.1: *Tube Distribution (An example with 200 elements.)*

The fact that there is no information beforehand on how deep each branch in the tree will be, makes this a suitable problem for dynamic load balancing.

A task in the implementation consists of an octant and a list of particles. The thread block that is assigned the task will divide the octant into eight new octants and count the number of elements that are contained in each. If the count is higher than a certain threshold, a new task is created containing the octant and the particles in it. If it is lower than the threshold, the particle count is added to a global counter. When the counter reaches the total number of particles the work is completed.

Particles found to be in the same octant are moved together to minimize the number of particles that has to be examined for further division of the octant. This means that the further down the tree the process gets, the less time it takes to complete a task.

This implementation of the octree partitioning algorithm should not be seen as the best possible one, but only as a way to compare the different work balancing methods.

3.6 Experimental Evaluation

Two different graphics processors were used in the experiments, the 9600GT 512MiB NVIDIA graphics processor with 64 cores and the 8800GT 512MiB NVIDIA graphics processor with 112 cores.

We used two input distributions, one where all particles were randomly picked from a cubic space and one where they were randomly picked from a space shaped like a geometrical tube, see Figure 3.1.

All methods where initialized by a single iteration using one thread block. The maximum number of particles in any given octant was set to 20 for all experiments.

3.6.1 Discussion

Figure 3.2 and 3.3 shows the time it took to partition two different particle sets on the 8800GT and 9600GT graphics processors using each of the load balancing methods while varying the number of threads per block and blocks per grid. The static method always uses one block per task and is thus shown in a 2D graph.

Figure 3.4 shows the time taken to partition particle sets of varying size using the combination of threads per block and blocks per grid found to be optimal in the previously described graph. The shapes of the graphs maps nicely to the total number of tasks created for each distribution and particle count, as shown in Figure 3.5. The task count is higher with the tube distribution since the octree is more unbalanced.

Figure 3.2 (a) clearly shows that using less than 64 threads with the blocking method gives us the worst performance in all of the experiments. This is due to the expensive spinning on the lock variable. These repeated attempts to acquire

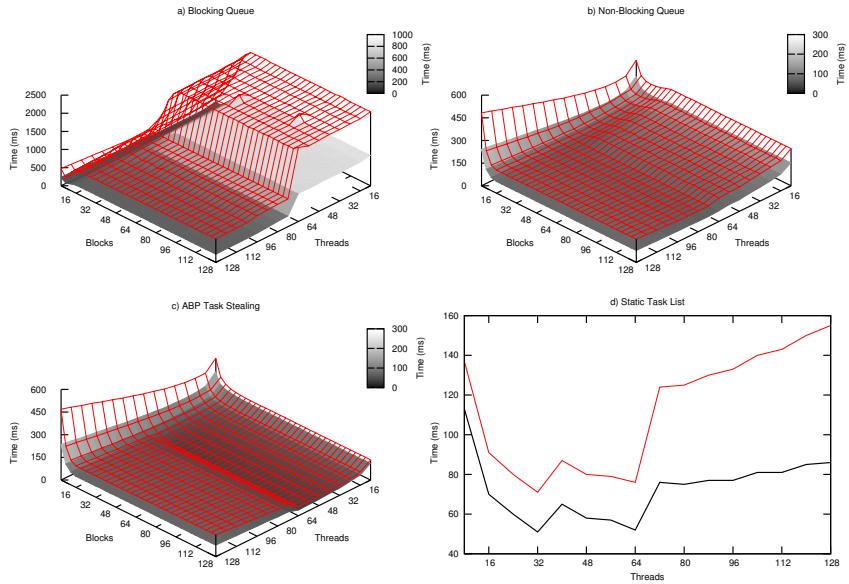


Figure 3.2: *Comparison of load balancing methods on the 8800GT. Shows the time taken to partition a* ***Uniform*** *(filled grid) and* ***Tube*** *(unfilled grid) distribution of half a million particles using different combinations of threads/block and blocks/grid.*

the lock causes the bus to be locked for long amounts of times during which only 32-bit memory accesses are done. With more than 64 threads the number of concurrent thread blocks is lowered from three to one, due to the limited number of available registers per thread, which leads to less lock contention and much better performance. This shows that the blocking method scales very poorly. In fact, we get the best result when using less than ten blocks, that is, by not using all of the multiprocessors! The same can be seen in Figure 3.3 (d) and 3.4 where the performance is better on the graphics processor with fewer cores. We used 72 threads and 8 blocks to get the best performance out of the blocking queue when comparing it with the other methods.

The non-blocking queue-based method, shown in Figure 3.2 (b), can take better advantage of an increased number of blocks per grid. We see that the performance increases quickly when we add more blocks, but after around 20

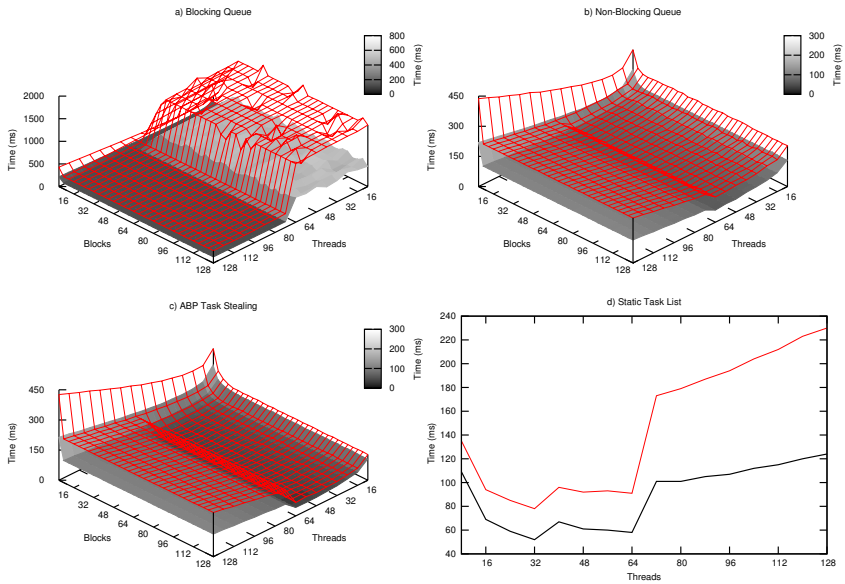


Figure 3.3: *Comparison of load balancing methods on the 9600GT. Shows the time taken to partition a* ***Uniform*** *(filled grid) and* ***Tube*** *(unfilled grid) distribution of half a million particles using different combinations of threads/block and blocks/grid.*

blocks the effect fades. It was expected that this effect would be visible until we increased the number of blocks beyond 42, the number of blocks that can run concurrently when using less than 64 threads. This means that even though its performance is much better than the its blocking counterpart, it still does not scale as well as we would have wanted. This can also clearly be seen when we pass the 64 thread boundary and witness an increase in performance instead of the anticipated drop. On the processor with fewer cores, Figure 3.3 (b), we do get a drop, indicating that conflicts are expensive for this queue implementation. Looking at Figure 3.4 we see the same thing, the non-blocking queue performs better on the processor with fewer cores. The measurements were done using 72 threads and 20 blocks.

In Figure 3.2 (c) we see the result from the ABP task stealing and it lies more closely to the ideal. Adding more blocks increases the performance until we get

to around 30 blocks. Adding more threads also increases performance until we get the expected drop 64 threads per block. We also get a slight drop after 32 threads since we passed the warp size and now have incomplete warps being scheduled. Figure 3.4 shows that the work stealing gives great performance and is not affected negatively by the increase in number of cores on the 8800GT. When we compared the task stealing with the other methods we used 64 threads and 32 blocks.

In Figure 3.2 (d) we see that the static method shows similar behavior as the task stealing. When increasing the number of threads used by the static method from 8 to 32 we get a steady increase in performance. Then we get the expected drops after 32 and 64, due to incomplete warps and less concurrent thread blocks. Increasing the number of threads further does not give any increase in speed as the synchronization overhead in the octree partitioning algorithm becomes dominant. The optimal number of threads for the static method is thus 32 and that is what we used when we compared it to the other methods in Figure 3.4.

As can be seen in Figure 3.2 and 3.3, adding more blocks than needed is not a problem since the only extra cost is an extra read of the finishing condition for each additional block.

Figure 3.5 shows the total number of tasks created for each distribution and particle count. As can be seen, the number of tasks increases quickly, but the tree itself is relatively shallow. A perfectly balanced octree has a depth of $\log_8(n/t)$, where t is the threshold, which with 500,000 elements and a threshold of 20 gives a depth of just 4.87. In practice, for the tube distribution, the static queue method required just 7 iterations to fully partition the particles and after just 3 iterations there were a lot more tasks than there were processing units.

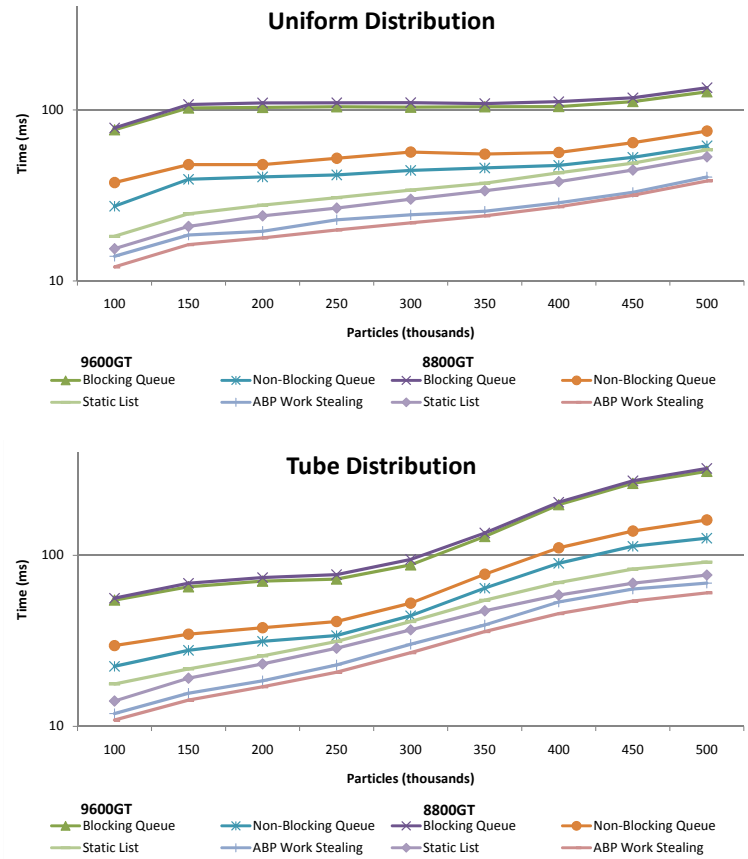


Figure 3.4: *A comparison of the load balancing methods on the* ***uniform*** *and* ***tube*** *distribution.*

3.7 Conclusions

We have compared four different load balancing methods, a blocking queue, a non-blocking queue, ABP task stealing and a static list, on the task of creating an octree partitioning of a set of particles.

We found that the blocking queue performed poorly and scaled badly when faced with more processing units, something which can be attributed to the inherent busy waiting. The non-blocking queue performed better but scaled poorly when the number of processing units got too high. Since the number of tasks increased quickly and the tree itself was relatively shallow the static queue performed well. The ABP task stealing method perform very well and outperformed the static method.

The experiments showed that synchronization can be very expensive and that new methods that take more advantage of the graphics processors features and capabilities might be required. They also showed that lock-free methods achieves better performance than blocking and that they can be made to scale with increased numbers of processing units.

Future Work

We are planning to compare the load balancing methods used in the paper on other problems, such as global illumination. Based on the conclusions from this work we are trying to develop new methods, tailored to graphics processors, for load balancing.

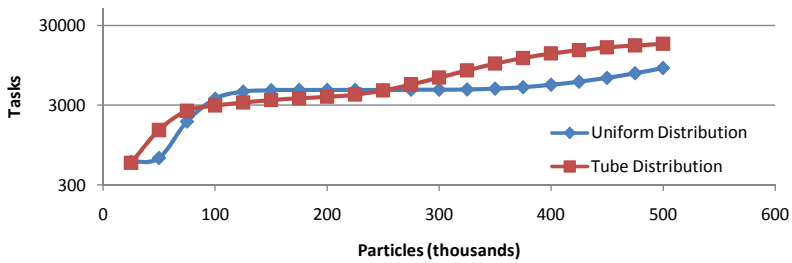


Figure 3.5: *The total number of tasks caused by the two distributions.*

Bibliography

[ABP98] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998.

[BL94] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico.*, pages 356–368, 1994.

[FS05] Tim Foley and Jeremy Sugerman. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, 2005.

[HA98] Alan Heirich and James Arvo. A Competitive Analysis of Load Balancing Strategies for Parallel Ray Tracing. *J. Supercomput.*, 12(1-2):57–68, 1998.

[KR03] Matthias Korch and Thomas Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms: Research articles. *Concurrency and Computation: Practice & Experience*, 16(1):1–47, 2003.

[NHP07] Lars Nyland, Marks Harris, and Jan Prins. Fast N-Body Simulation with CUDA. In *GPU Gems 3*, chapter 31, pages 677–695. Addison-Wesley, 2007.

[SG91] M. Shephard and M. Georges. Automatic three-dimensional mesh generation by the finite Octree technique. *International Journal for Numerical Methods in Engineering*, 32:709–749, 1991.

[SMRR07] Luciano Soares, Clément Ménier, Bruno Raffin, and Jean-Louis Roch. Work Stealing for Time-constrained Octree Exploration:

Application to Real-time 3D Modeling. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, 2007.

[TZ01] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *Proceedings of the thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 134–143, 2001.

PAPER III

Daniel Cederman, Philippas Tsigas

Supporting Lock-Free Composition of Concurrent Data Objects

In the Proceedings of the 7th ACM conference on Computing Frontiers (CF 10)
pages: 53-62, ACM 2010.

4

Supporting Lock-Free Composition of Concurrent Data Objects

Lock-free data objects offer several advantages over their blocking counterparts, such as being immune to deadlocks and convoying and, more importantly, being highly concurrent. However, composing the operations they provide into larger atomic operations, while still guaranteeing efficiency and lock-freedom, is a challenging algorithmic task.

We present a lock-free methodology for composing highly concurrent linearizable objects together by unifying their linearization points. This makes it possible to relatively easily introduce atomic lock-free move operations to a wide range of concurrent objects. Experimental evaluation has shown that

the operations originally supported by the data objects keep their performance behavior under our methodology.

4.1 Introduction

Lock-free data objects offer several advantages over their blocking counterparts, such as being immune to deadlocks, priority inversion, and convoying, and have been shown to work well in practice [ST02, TZ01, TZ02]. They have been included in Intel's Threading Building Blocks Framework [Int09], the NOBLE library [ST02] and the Java concurrency package [Lea09], and will be included in the forthcoming parallel extensions to the Microsoft .NET Framework [Mic09]. However, the lack of a general, efficient, lock-free method for composing them makes it difficult for the programmer to perform multiple operations together atomically. To efficiently glue together multiple objects, and their respective operations, one needs to perform an often challenging task that requires an efficient algorithmic design for every particular composition. The task is made difficult by the fact that lock-free data objects are often too complicated to be trivially altered.

Composing blocking data objects also puts the programmer in a difficult situation, as it requires knowledge of the way locks are handled internally (in the implementation of the objects themselves), in order to avoid deadlocks. It is not possible to build on lock-based components without examining their implementations and even then the drawbacks of locking will not go away.

Software Transactional Memories provide good composability, but have problems with high overhead and have poor support for dealing with non-transactional code [HMPJH05, CBM$^+$08, LK08]. They require, with few exceptions, that the data objects are rewritten to be handled completely inside the STM, which lowers performance compared to pure non-blocking data objects.

4.1.1 Composing

With the term composing we refer to the task of binding together multiple operations in such a way that they can be performed as one, without any intermediate state being visible to other processes. In the literature the term is also used for nesting, making one data object part of another, which is an interesting problem, but outside the scope of this paper.

Composing lock-free concurrent data objects, in the context that we consider in this paper, has been an open problem in the area of lock-free data objects. There exists customized compositions of specific concurrent data objects, including the composition of lock-free flat-sets by Gidenstam et al. that constitute the foundation of a lock-free memory allocator [GPT09, GPT05], but no generic solution.

Using blocking locks to compose lock-free operations is not a viable solution, as it would reduce the concurrency and remove the lock-freedom guarantees of the operations. The reason for this is that the lock-free operations would have to acquire a lock before executing, in order to ensure that they are not executed concurrently with any composed operations. This would cause the operations to be executed sequentially and lose their lock-free behavior. Simply put, a generic way to compose concurrent objects, without foiling the possible lock-freedom guarantees of the objects, has to be lock-free itself.

4.1.2 Contributions

The main contribution of this paper is to provide a methodology to introduce atomic move operations, that can move elements between objects of different types, to a large class of already existing concurrent objects without having to make significant changes to them. It manages this while preserving the lock-free guarantees of the object and without introducing significant performance penalties to the previously supported operations. Move operations are an important part of the core functionality needed when composing any kind of containers, as they provide the possibility to shift items between objects.

In our methodology we present a set of properties that can be used to identify suitable concurrent objects and we describe the mostly mechanical changes needed for our move operation to function together with the objects. The properties required by our methodology are fulfilled by a wide variety of lock-free data objects, among them lock-free stacks, queues, lists, skip-lists, priority queues, hash-tables and dictionaries [Tre86, MS96, ST05, FR04, ST04, Val95, Mic02, Har01b].

Our methodology is based on the idea of decomposing and then arranging lock-free operations appropriately so that their linearization points can be combined to form new composed lock-free operations. The linearization point of a concurrent operation is the point in time where the operation can be said to have taken effect. Most concurrent data objects that are not read- or write-only support an insert and a remove operation, or a set of equivalent operations that can be used to modify its content. These two types of operations can be composed together using the method presented in this paper to make them appear to take effect simultaneously. By doing this we provide a lock-free atomic operation that can move elements between objects of different types. To the best of our knowledge this is the first time that such a general scheme has been proposed.

As a proof of concept we show how to apply our method on two commonly used concurrent data objects, the lock-free queue by Michael and Scott [MS96] and the lock-free stack by Treiber [Tre86]. Experimental results on an Intel multiprocessor system show that the methodology presented in the paper, applied to the previously mentioned lock-free implementations, offers significantly better performance and scalability than a composition method based on locking. The proposed method does this in addition to its qualitative advantages regarding progress guarantees that lock-freedom offers. Moreover, the experimental evaluation has shown that the operations originally supported by the data objects keep their performance behavior while used as part of our methodology.

4.2 The Model

The model considered is the standard shared memory model, where a set of memory locations can be read from and written to, by a number of processes that progress asynchronously. Concurrent data objects are composed of a subset of these memory locations together with a set of operations that can use read and write instructions, as well as other atomic instructions, such as compare-and-swap (CAS). We require all concurrent data objects to be linearizable to assure correctness.

Linearizability is a commonly used correctness criterion introduced by Herlihy and Wing [HW90]. Each operation on a concurrent object consists of an invocation and a response. A sequence of such operations makes up a history. Operations in a concurrent history can be placed in any order if they occur concurrently, but an operation that finishes before another one is invoked must appear before the latter. If the operations in any actual concurrent history can be reordered in such a way, so that the history is equivalent to a correct sequential history, then the concurrent object is linearizable. One way of looking at linearizability is to think that an operation takes effect at a specific point in time, the linearization point. All operations can then be ordered according to the linearization point to form a sequential history.

4.3 The Methodology

The methodology that we present can be used to unify the linearization points of a remove and an insert operation for any two concurrent objects, given that they fulfill certain requirements. We call a concurrent object that fulfills these requirements a *move-candidate* object.

4.3.1 Characterization

Definition 4.1. *A concurrent object is a* move-candidate *if it fulfills the following requirements:*

1. *It implements linearizable operations for insertion and removal of a single element.*

2. *Insert and remove operations invoked on different instances of the object can succeed simultaneously.*

3. *The linearization points of the successful insert and remove operations can be associated with successful CAS operations, (on a pointer), by the process that invoked it. Such an associated successful CAS can never lead to an unsuccessful insert or remove operation.*

4. *The element to be removed is accessible before the linearization point.*

To implement a move operation, the equivalent of a remove and insert operation needs to be available or be implemented. A generic insert or remove operation would be very difficult to write, as it must be tailored specifically to the concurrent object, which motivates the first requirement.

Requirement 2 is needed since a move operation tries to perform the removal and insertion of an element at the same time. If a successful removal invalidates an insertion, or the other way around, then the move operation can never succeed. This could happen when the insert and remove operations share locks between them or when they are using memory management schemes such as hazard pointers [Mic04], if not dealt with explicitly. With shared locks there is the risk of deadlocks, when the process could be waiting for itself to release the lock in the remove operation, before it can acquire the same lock in the insert operation. Hazard pointers, which are used to mark memory that cannot yet be reused, could be overwritten if the same pointers are used in both the insert and remove operations.

Requirement 3 requires that the linearization points can be associated with successful CAS operations. The linearization points are usually provided together with the algorithmic description of each object. Implementations that use the LL/SC[1] pair for synchronization can be translated to ones that use CAS

[1] LL (Load-Link) and SC (Store-Conditional) are used together. LL reads a value from a memory location and SC can then only write a new value at the same location if the memory location has not been written to since the last LL.

by using the construction by Doherty et al. that implements the LL/SC functionality from CAS [DHLM04]. The requirement also states that the CAS operation should be on a variable holding a pointer. This is not a strict requirement; the reason for it is that the DCAS operation used in our methodology often needs to be implemented in software due to lack of hardware support for such an operation. By only working with pointers it makes it easier to identify words that are taking part in a DCAS operation. The last part, which requires the linearization point of an operation to be part of the process that invoked it, prevents concurrent data objects from using some of the possible helping schemes, but not the majority of them. For example, it does not prevent using the commonly used helping schemes where the process that helps another process is not the one that defines the linearization point of the process helped. As described in Section 4.1.2, there is a large class of well-known basic and advanced data objects that fulfills this requirement.

Requirement 4 is necessary as the insert operation needs to be invoked with the removed element as an argument. The element is usually available before the linearization point, but there are data objects where the element is never returned by the remove operation, or is accessed after the linearization point for efficiency reasons.

4.3.2 The Algorithm

The main part of the algorithm is the actual move operation, which is described in the following section. Our move operation makes heavy use of a DCAS operation that is described in detail in Section 4.3.2.

The Move Operation

The main idea behind the move operation is based on the observation that the linearization points of many concurrent objects' operations is a CAS and that by combining these CASs and performing them simultaneously, it would be possible to compose operations. A move operation does not need an expensive general multi-word CAS, so an efficient two word CAS customized for this

particular operation is good enough. We would like to simplify the utilization of this idea as much as possible, and for this reason we worked towards three goals when we designed the move operation:

- The changes required to adapt the concurrent data object should be minimal and be possible to perform mechanically.
- The performance impact on the normal operations of the concurrent data objects should be minimized.
- The move operation should be lock-free if the insert and remove operations are lock-free.

With these goals in mind we decided that the easiest and most generic way would be to reuse the remove and insert operations that are already supported by the object. By definition a move-candidate operation has a linearization point that consists of a successful CAS. We call the part of the operation prior to this linearization point the init-phase and the part after it the cleanup-phase. The move can then be seen as taking place in five steps:

1st step. The init-phase of the remove operation is performed. If the removal fails, due for example to the element not existing, the move is aborted. Otherwise the arguments to the CAS at the potential linearization point are stored. By requirement 4 of the definition of a move-candidate, the element to be moved can now be accessed.

2nd step. The init-phase of the insert operation is performed using the element received in the previous step. If the insertion fails, due for example to the object being full, the move is aborted. Otherwise the arguments to the CAS at the potential linearization point are stored.

3rd step. The CASs that define the linearization points, one for each of the two operations, are performed together atomically using a DCAS operation with the stored CAS arguments. Step two is redone if the DCAS failed due to a conflict in the insert operation. Steps one and two are redone if the conflict was in the remove operation.

4th step. The cleanup-phase for the insert operation is performed.

5th step. The cleanup-phase for the remove operation is performed.

To be able to divide the insert and remove operations into the init- and cleanup-phases without resorting to code duplication, it is required to replace all possible linearization point CASs with a call to the `scas` operation. The task of the `scas` operation is to restore control to the move operation and store the arguments intended for the CAS that was replaced. The `scas` operation is described in Algorithm 9 and comes in two forms, one to be called by the insert operations and one to be called by the remove operations. They can be distinguished by the fact that the `scas` for removal requires the element to be moved as an argument. If the `scas` operation is invoked as part of a normal insert or remove, it reverts back to the functionality of a normal CAS. This should minimize the impact on the normal operations.

If the DCAS operation used is a software implementation that uses helping, it might be required to use hazard pointers to disallow reclaiming of the memory used by it. In those cases the hazard pointers can be given as an argument to the `scas` operation and they will be brought to the DCAS operation. The DCAS operation provided in this paper uses helping and takes advantage of the support for hazard pointers.

If the DCAS in step 3 should fail, this could be for one of two reasons. First, it could fail because the CAS for the insert failed. In this case the init-phase for the insert needs to be redone before the DCAS can be invoked again. Second, it could fail because the CAS for the remove failed. Now we need to redo the init-phase for the remove, which means that the insert operation needs to be aborted. For concurrent objects such as linked lists and stacks there might not be a preexisting way for the insert to abort, so code to handle this scenario must be inserted. The code necessary usually amounts to freeing allocated memory and then return. The reason for this simplicity is that the abort always occurs before the operation has reached its linearization point. If the insertion operation can fail for reasons other than conflicts with another operation, there is also a need for the remove operation to be able to handle the possibility of aborting.

Depending on whether one uses a hardware implementation of a DCAS or a software implementation, it might also be required to alter all accesses to memory words that could take part in DCAS, so that they access the word via a special read-operation designed for the DCAS.

A concurrent object that is a move-candidate (Definition 4.1) and has implemented all the above changes is called a *move-ready* concurrent object. This is described formally in the following definition.

Definition 4.2. *A concurrent object is* move-ready *if it is a* move-candidate *and has implemented the following changes:*

1. *The CAS at each linearization point in the insert and remove operations have been changed to* `scas`*.*
2. *The insert (and remove) operation(s) can abort if the* `scas` *operation returns* `ABORT`*.*
3. *(All memory locations that could be part of a* `scas` *are accessed via the* read *operation.)*

The changes required are mostly mechanical once the object has been found to adhere to the move-ready definition. This object can then be used by our move operation to move items between different instances of any concurrent move-ready objects. Requirement 3 is not required for systems with a hardware based DCAS.

Theorem 4.2 in Section 4.4 states that the move operation is linearizable and lock-free if used together with two move-ready lock-free concurrent data objects.

DCAS

The DCAS operation performs a CAS on two distinct words atomically (See Algorithm 7 for its semantics). It is unfortunately not commonly available in hardware, some say for good reasons [DDG+04], so for our experiments it had to be implemented in software. There are several different multi-word compare-and-swap methods available in the literature [IR94, ARJ97, Her93, HT03, AM95,

Moi97, ST95, HFP02] and ours uses the same basic idea as in the solution by Harris et al.

Lock-freedom is achieved by using a two-phase locking scheme with helping[2]. First an attempt is made to change both the words involved, using a normal CAS, to point to a descriptor that holds all information required for another process to help the DCAS complete. See lines D10 and D14 in Algorithm 12. If any of the CASs fail, the DCAS is unsuccessful as both words need to match their old value. In this case, if one of the CASs succeeded, its corresponding word must be reverted back to its old value. When a word holds the descriptor it cannot be changed by any other non-helping process, so if both CASs are successful, the DCAS as a whole is successful. The two words can now be changed one at a time to hold their respective new values. See lines D28 and D29.

If another process wants to access a word that is involved in a DCAS, it first needs to help the DCAS operation finish. The process knows that a word is used in a DCAS if it is pointing to a descriptor. This is checked at line D34 in the read operation. In our experiments we have marked the descriptor pointer by setting its least significant bit to one. This is a method introduced by Harris et al. [Har01a] and it is possible to use since we assume that the word will contain a pointer and that pointers will be aligned to the word size of the system. Using the information in the descriptor it tries to perform the same steps as the initiator, but marks the pointer to the descriptor it tries to swap in with its thread id. This is done to avoid the ABA-problem, which can occur since CAS cannot distinguish a word that has been changed from A to B and then back to A again, from a word whose value has remained A. Unless taken care of in this manner, the ABA-problem could cause the DCAS to succeed multiple times, one for each helping process.

Our DCAS differs from the one by Harris et al. in that i) it has support for reporting which, if any, of the operations has failed, ii) it does not need to allocate an `RDCSSDescriptor` as it only changes two words, iii) it has support for hazard pointers, and iv) it requires two fewer CASs in the uncontended case.

[2]Lock-freedom does not exclude the use of locks, in contrast to its definition-name, if the locks can be revoked.

Algorithm 7 Semantics of the DCAS operation.

```
struct DCASDesc
  word old1, old2, new1, new2
  word *ptr1, *ptr2
  [word *hp1, *hp2]
  word res

dres DCAS(desc)
  if(*desc.ptr1 ≠ desc.old1)
    return FIRSTFAILED
  if(*desc.ptr2 ≠ desc.old2)
    return SECONDFAILED
  *desc.ptr1 ← desc.new1
  *desc.ptr2 ← desc.new2
  return SUCCESS
```

These are, however, minor differences and for our methodology to function it is not required to use our specific implementation. Performance gains and practicality reasons account for the introduction of the new DCAS. The DCAS is linearizable and lock-free according to Theorem 4.1.

4.4 Proof

Lemma 4.1. *The DCAS descriptor's* `res` *variable can only be changed from* `UNDECIDED` *to* `SECONDFAILED` *or from* `UNDECIDED` *to a marked descriptor and consequently to* `SUCCESS`*.*

Proof. The `res` variable is set at lines D17, D24, and D30. On lines D17 and D24 the change is made using CAS, which assures that the variable can only change from `UNDECIDED` to `SECONDFAILED` or to a marked descriptor. Line D30 writes `SUCCESS` directly to `res`, but it can only be reached if `res` differs from `SECONDFAILED` at line D25, which means that it must hold a marked descriptor as set on line D24 or already hold `SUCCESS`. □

Lemma 4.2. *The initiating and all helping processes will receive the same result value.*

Algorithm 8 Basic operations.

```
bool remove([key],*item)
  ...
  while(unsuccessful)
    ...
    result ← scas(ptr, old, new, element, [hp])
    // Only needed when insert can fail
    [if(result=ABORT)]
      [abort]
      [return false]
    ...
  ...
```

```
bool insert([key],item)
  ...
  while(unsuccessful)
    ...
    result ← scas(ptr, old, new,[hp])
    if(result=ABORT)
      abort
      return false
    ...
  ...
```

Proof. DCAS returns the result value at lines D9, D11, D19, D22, D27, and D31. Lines D22 and D27 are only executed if `res` is equal to `SECONDFAILED` and we know by Lemma 4.1 that the result value cannot change after that. Lines D31 and D19 can only be executed when `res` is `SUCCESS` and by the same Lemma the value can not change. Line D9 only returns when the result value is either `SUCCESS` or `SECONDFAILED` and as stated before these value cannot change. Line D11 returns `FIRSTFAILED` when the initiator process fails to announce the DCAS, which means that no other process will help the operation to finish. □

Lemma 4.3. *Iff the result value of the DCAS is* `SUCCESS`*, then* $*ptr_1$ *has changed value from* old_1 *to the descriptor to* new_1 *and* $*ptr_2$ *has changed value from* old_2 *to a marked descriptor to* new_2 *once.*

Proof. On line D10 $*ptr_1$ is set to the descriptor by the initiating process as otherwise the result value would be `FIRSTFAIL`. On line D14, $*ptr_2$ is set to a

Algorithm 9 Move operation.

```
thread local variables
  desc, ltarget, lskey, ltkey, insfailed

M1 bool move(source, target, [skey, tkey])
M2 desc ← new DCASDesc
M3 desc.res ← UNDECIDED
M4 [lskey ← skey, ltkey ← tkey]
M5 ltarget ← target
M6 result ← source.remove([lskey], tmp)
M7 desc ← 0
M8 return result
```

Algorithm 10 Move operation. scas for dequeue.

```
M9 fbool scas(ptr, old, new, element, [hp])
M10 if(desc ≠ 0)
M11   desc.ptr1 ← ptr
M12   desc.old1 ← old
M13   desc.new1 ← new
M14   [desc.hp1 ← hp]
M15   insfailed ← true
M16   result ← ltarget.insert([ltkey], element)
M17   if(insfailed)
M18     return ABORT
M19   return result
M20 else
M21   return cas(ptr,old,new)
```

marked descriptor by any of the processes. By contradiction, if all processes failed to change the value of $*ptr_2$ on line D14, the result value would be set to SECONDFAILED on line D17. On line D24 the res variable is set to point to a marked descriptor. This change is a step on the path to the SUCCESS result value and thus must be taken. On line D28 $*ptr_1$ is changed to new_1 by one process. It can only succeed once as the descriptor is only written once by the initiating process. This is in contrast to $*ptr_2$ which can hold a marked descriptor multiple times due to the ABA-problem at line D14. When $*ptr_2$ is changed to new_2 it could be changed back to old_2 by a process outside of the DCAS. The CAS at line D14 has no way of detecting this. This is the reason why we

Algorithm 11 Move operation. scas for enqueue.

```
M22 fbool scas(ptr, old, new, [hp])
M23 if(desc ≠ 0)
M24   desc.ptr2 ← ptr
M25   desc.old2 ← old
M26   desc.new2 ← new
M27   [desc.hp2 ← hp]
M28   result ← DCAS(desc, true)
M29   if(result != SUCCESS)
M30     desc ← new DCASDesc(desc)
M31     desc.res ← UNDECIDED
M32   insfailed ← false
M33   if(result = FIRSTFAILED)
M34     return ABORT
M35   if(result = SECONDFAILED)
M36     return false
M37   return true
M38 else
M39   return cas(ptr,old,new)
```

are using a marked descriptor that is stored in the `res` variable using CAS, as this will allow only one process to change the value of $*\text{ptr}_2$ to new_2 on line D29. A process that manages to store its marked descriptor to $*\text{ptr}_2$, but was not the first to set the `res` variable, will have to change it back to its old value. □

Lemma 4.4. *Iff the result value of the DCAS operation is* `FIRSTFAILED` *or* `SECONDFAILED`*, then* $*ptr_1$ *was not changed to* new_1 *in the DCAS and* $*ptr_2$ *was not changed to* new_2 *in the DCAS due to either* $*ptr_1 \neq old_1$ *or* $*ptr_2 \neq old_2$.

Proof. If the CAS at line D10 fails, nothing is written to $*\text{ptr}_1$ by any processes since the operation is not announced. The CAS at line D24 must fail, since otherwise the result value would not be `SECONDFAILED`. This means the test at line D25 will succeed and the operation will return before line D29, which is the only place that $*\text{ptr}_2$ can be changed to new_2. □

Lemma 4.5. *Iff the result value of the DCAS is* `SUCCESS`*, then* $*ptr_1$ *held a descriptor at the same time as* $*ptr_2$ *held a marked descriptor.*

Algorithm 12 Double word compare-and-swap.

```
D1 dres DCAS(desc, initiator)
D2 [if(¬initiator)]
D3     [hp1 ← desc.hp1, hp2 ← desc.hp2]
D4 if(desc.res = SUCCESS ∨ SECONDFAILED)
D5   if(desc is marked)
D6     cas(desc.ptr2, desc, desc.old2)
D7   else
D8     cas(desc.ptr1, desc, desc.old1)
D9   return desc.res
D10 if(initiator∧¬cas(desc.ptr1, desc.old1, desc))
D11   return FIRSTFAILED
D12
D13 mdesc ← mark(unmark(desc),threadID)
D14 p2set ← cas(desc.ptr2, desc.old2, mdesc)
D15 if(¬p2set)
D16   if(*desc.ptr2.ptr ≠ desc)
D17     cas(desc.res, UNDECIDED, SECONDFAILED)
D18     if(desc.res = SUCCESS)
D19       return desc.res
D20     if(desc.res = SECONDFAILED)
D21       cas(desc.ptr1, desc, desc.old1)
D22       return desc.res
D23
D24 cas(desc.res, UNDECIDED, mdesc)
D25 if(desc.res = SECONDFAILED)
D26   if(p2set) cas(desc.ptr2, mdesc, desc.old2)
D27   return desc.res
D28 cas(desc.ptr1, desc, desc.new1)
D29 cas(desc.ptr2, desc.res, desc.new2)
D30 desc.res ← SUCCESS
D31 return desc.res
```

Proof. Line D28 can only be reached if the CASs at lines D10 and D14 were successful. The values of $*\text{ptr}_1$ and $*\text{ptr}_2$ are not changed back until lines D28 and D29, so just before the first process reaches line D28 $*\text{ptr}_1$ holds a descriptor and $*\text{ptr}_2$ holds a marked descriptor. □

Lemma 4.6. *If the initiating process protects $*ptr_1$ and $*ptr_2$ with hazard pointers, they will not be written to by any helping process unless that process also protects them with hazard pointers.*

Algorithm 13 Double word compare-and-swap. Read operation.

```
D32 word read(*ptr)
D33   result ← *ptr
D34   while(result is DCASDesc)
D35     hp_d ← result
D36     if(hp_d = *ptr)
D37       DCAS(result, false)
D38     result ← *ptr
D39   return result
```

Proof. If the initiating process protects the words, they will not be unprotected until that process returns, at which point the final result value must have been set. This means that if the test at D4 fails for a helping process, the words were protected when the process local hazard pointers were set at line D3. If the test did not fail, then the words are not guaranteed to be protected. But in that case the word that is written to at line D6 or D8 is the same word that was read in the `read` operation. That word must have been protected earlier by the process calling it, as otherwise it could potentially read from invalid space. Thus the words are either protected by the hazard pointers set at line D3 or by hazard pointers set before calling the `read` operation. □

Lemma 4.7. *DCAS is lock-free.*

Proof. The only loop in DCAS is part of the read operation that is repeated until the word read is no longer a DCAS descriptor. The word can be assigned the same descriptor, with different process id, for a maximum number of $p-1$ times, where p is the number of processes in the system. This can happen when each helping process manages to write to $*\text{ptr}_2$ due to the ABA-problem mentioned earlier. This can only happen once for each process per descriptor as it will not get past the test on line D4 a second time.

So, a descriptor appearing on a word means that either a process has started a new DCAS operation or that a process has made an erroneous helping attempt. Either way, one process must have made progress for this to happen, which makes the DCAS lock-free. □

THEOREM 4.1. *The DCAS is lock-free and linearizable with possible linearization points at* D10, D17, *and* D24, *and follows the semantics as specified in Algorithm 7.*

Proof. Lemma 4.2 gives that all processes return the same result value. According to Lemmata 4.3 and 4.4, the result value can be seen as deciding the outcome of the DCAS. The result value is set at D17 and D24, which become possible linearization points. It is also set at D30, but that comes as a consequence of the CAS at line D24. The final candidate for linearization point happens when the CAS at line D10 fails. This happens before the operation is announced so we do not need to set the `res` variable.

Lemma 4.3 proves that when the DCAS is successful it has changed both $*\text{ptr}_1$ and $*\text{ptr}_2$ to an intermediate state from a state where they were equal to old_1 and old_2, respectively. Lemma 4.5 proves that they were in this intermediate state at the same time before they got their new values, according to Lemma 4.3 again. If the DCAS was unsuccessful then nothing is changed due to either $*\text{ptr}_1 \neq \text{old}_1$ or $*\text{ptr}_2 \neq \text{old}_2$. This is in accordance with the semantics specified in Algorithm 7.

Lemma 4.7 gives that DCAS is lock-free. □

THEOREM 4.2. *The* move *operation is linearizable and lock-free if applied to two lock-free move-ready concurrent objects.*

Proof. We consider DCAS an atomic operation as shown by Theorem 4.1. All writes, except the ones done by the DCAS operation, are process local and can as such be ignored.

The move operation starts with an invocation of the remove operation. If it fails, it means that there were no elements to remove from the object and that the linearization point must lie somewhere in the remove operation, since requirement 1 of the definition of a move-candidate states that the operations should be linearizable. If the process reaches the first `scas` call, the insert operation is invoked with the element to be removed as an argument. If the insert fails before it reached the second `scas` call, it was not possible to insert the element. In this case the `insfailed` variable is not set at line M32 and `scas` will

abort the remove operation. The linearization point in this case is somewhere in the insert operation. In both these scenarios, whether it is the remove or the insert operation that fails, the move operation as a whole is aborted.

If the process reached the second `scas` call, the one in the insert operation, the DCAS operation is invoked. If it is successful, then both the insert and remove operation must have succeeded according to requirement 3 of the definition of a move-candidate. By requirement 1, they can only succeed once, which makes the DCAS the linearization point. If the DCAS fails nothing is written to the shared memory and either the insert or both the remove and the insert operations are restarted.

Since the insert and remove operations are lock-free, the only reason for the DCAS to fail is that another process has made progress in their insertion or removal of an element. This makes the move operation as a whole lock-free. □

4.5 Case Study

To get a better understanding of how our methodology can be used in practice, we apply it to two commonly used concurrent objects, the lock-free queue by Michael and Scott [MS96] and the lock-free stack by Treiber [Tre86]. The objects use hazard pointers for memory management and the selection of them is motivated in the paper by Michael [Mic04].

4.5.1 Queue

The first task is to see if the queue is a move-candidate as defined by Definition 4.1:

1. The queue fulfills the first requirement by providing dequeue and enqueue operations, which have been shown to be linearizable [MS96].

2. The insert and remove operations share hazard pointers in the original implementation. By using a separate set of hazard pointers for the de-

Algorithm 14 Lock-free queue by Michael and Scott [MS96]. Enqueue operation.

```
Q1 bool enqueue(val)
Q2   node ← new Node
Q3   node.next ← 0
Q4   node.val ← val
Q5   while(true)
Q6     ltail ← read(tail)
Q7     hp1 ← ltail; if(hp1 != read(tail)) continue
Q8     lnext ← read(ltail.next)
Q9     hp2 ← lnext
Q10    if(ltail != read(tail)) continue
Q11    if(lnext != 0)
Q12      cas(tail,ltail,lnext)
Q13      continue
Q14    res ← scas(ltail.next,0,node,hp1)
Q15    if(res = abort)
Q16      free node
Q17      return false
Q18    if(res = true)
Q19      cas(tail,ltail,node)
Q20      return true
```

queue operation we fulfill requirement number 2, as no other information is shared between two instances of the object.

3. The linearization points can be found on lines Q34, and Q14 and both consist of a successful CAS, which is what requirement number 3 asks for. There is also a linearization point at line Q29, but it is not taken in the case of a successful dequeue. These linearization points were provided together with the algorithmic description of the object, which is usually the case for the concurrent linearizable objects that exist in the literature.

4. The linearization point for the dequeue is on line Q34 and the value that is read in case of a successful CAS is available on line Q33, which must be executed before line Q34.

The above simple observations give us the following lemma in a straightforward way.

Algorithm 15 Lock-free queue by Michael and Scott [MS96]. Dequeue operation.

```
Q21 bool dequeue(*val)
Q22  while(true)
Q23    lhead ← read(head)
Q24    hp3 ← lhead; if(hp3 != read(head) continue
Q25    ltail ← read(tail)
Q26    lnext ← read(lhead.next)
Q27    hp4 ← lnext
Q28    if(lhead!=read(head)) continue
Q29    if(lnext=0) return false
Q30    if(lhead==ltail)
Q31      cas(tail,ltail,lnext)
Q32      continue
Q33    *val ← lnext.val
Q34    if(scas(head,lhead,lnext,val,hp3)
Q35      free lhead
Q36      return true
```

Lemma 4.8. *The queue by Michael and Scott is a move-candidate.*

After making sure that the queue is a move-candidate we need to replace the CAS operations at the linearization points on lines Q34 and Q14 with calls to the `scas` operation. If we are using a software implementation of DCAS we also need to alter all lines where words are read that could be part of a DCAS, so that they access them via the read operation. For the queue these changes need to be done on lines Q6, Q7, Q8, Q10, Q23, Q24, Q25, Q26, and Q28.

One must also handle the case of `scas` returning `ABORT` in the enqueue. Since there has been no change to the queue, the only thing to do before returning from the operation is to free up the allocated memory on line Q16. The enqueue cannot fail so there is no need to handle the `ABORT` result value in the dequeue operation.

The move operation can now be used with the queue. In Section 4.6 we evaluate the performance of the move-ready queue when combined with another queue, and when combined with the Treiber stack.

Algorithm 16 Lock-free stack by Treiber [Tre86]. Push operation.

```
s1 bool push(val)
s2   node ← new Node
s3   node.val ← val
s4   while(true)
s5     ltop ← read(top)
s6     node.next ← ltop
s7     res ← scas(top,ltop,node)
s8     if(res = abort)
s9       free node
s10      return false
s11    if(res = true)
s12      return true
```

Algorithm 17 Lock-free stack by Treiber [Tre86]. Pop operation.

```
s13 bool pop(val)
s14   while(true)
s15     ltop ← read(top)
s16     if(ltop = 0)
s17       return false
s18     hp ← ltop
s19     if(read(top) != ltop)
s20       continue
s21     val ← ltop.val
s22     if(scas(top,ltop,ltop.next,val))
s23       free ltop
s24       return true
```

4.5.2 Stack

Once again we first check to see if the stack fulfils the requirements of the move-candidate definition:

1. The push and pop operations are used to insert and remove elements and it has been shown that they are linearizable. Vafeiadis has, for example, given a formal proof of this [Vaf09].

2. There is nothing shared between instances of the object, so the push and pop operations can succeed simultaneously.

3. The linearization points on lines s7 and s22 are both CAS operations. The linearization point on line s17 is not a CAS, but it is only taken when the source stack is empty and when the move can not succeed. The conditions in the definition only require successful operations to be associated to a successful CAS.

4. The element to be removed is available on line s21, which is before the linearization point on line s22.

The above simple observations give us the following lemma in a straightforward way.

Lemma 4.9. *The stack by Treiber is a move-candidate.*

To make the stack object move-ready we change the CAS operations on lines s7 and s22 to point to `scas` instead. We also need to change the read of *top* on lines s5, s15, and s19, if we are using a software implementation of DCAS, so that it goes via the read operation. Since push can be aborted we also need to add a check after line s7 that looks for this condition and frees allocated memory.

The stack is now move-ready and can be used to atomically move elements between instances of the stack and other move-ready objects, such as the previously described queue. In the next section we evaluate the performance of the move-ready stack when combined with another stack as well as when combined with the Michael and Scott queue.

4.6 Experiments

The evaluation was performed on a machine with an Intel Core i7 950 3 GHz processor and 6 GB DDR3-1333 memory. The processor has four cores with hyper-threading, providing us with eight virtual processors in total. All experiments were based on either two queues, two stacks, or one queue and one stack. The stack used was the lock-free stack by Treiber and the queue was the lock-free queue by Michael and Scott [Tre86, MS96].

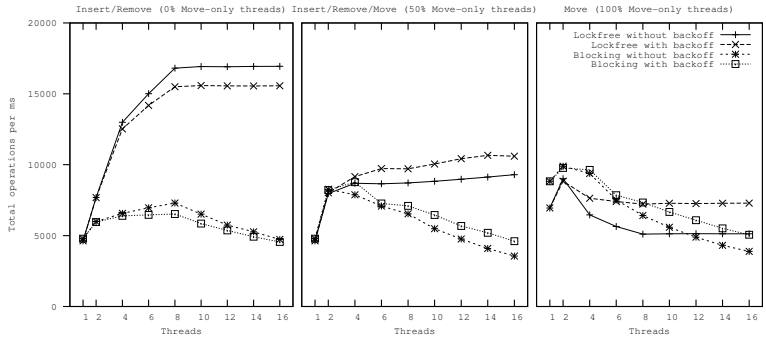


Figure 4.1: *Results from the queue/stack evaluation under* high *contention.*

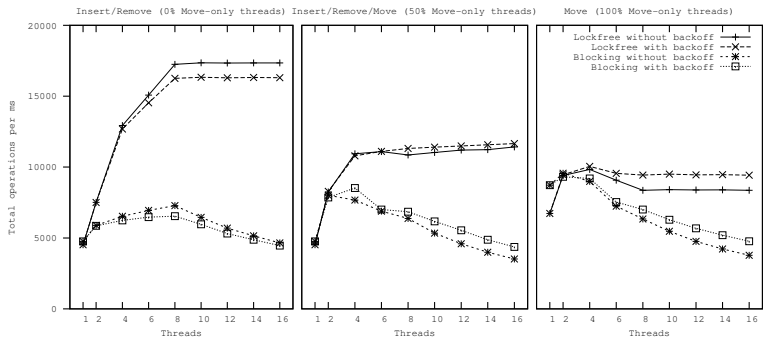


Figure 4.2: *Results from the queue evaluation under* high *contention.*

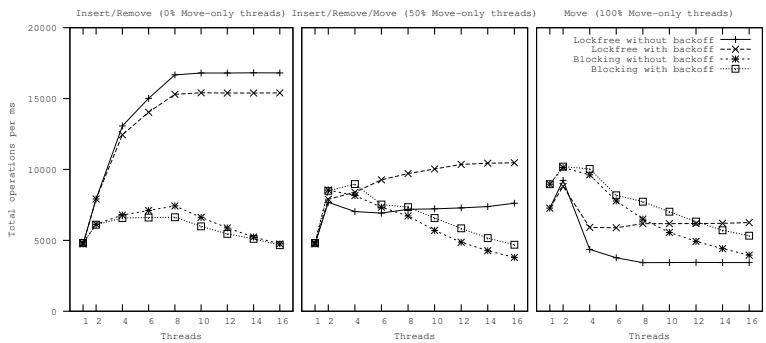


Figure 4.3: *Results from the stack evaluation under* high *contention.*

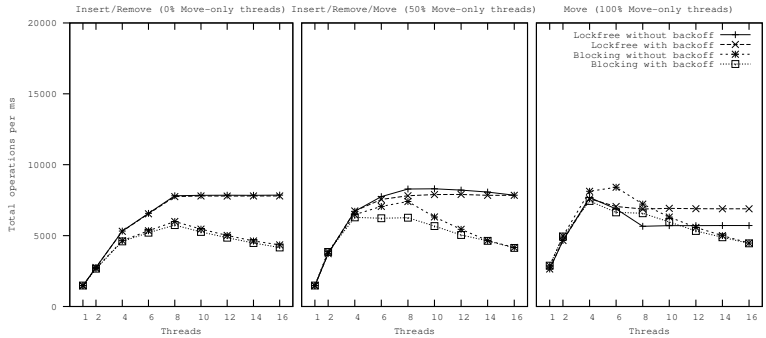


Figure 4.4: *Results from the queue/stack evaluation under* low *contention.*

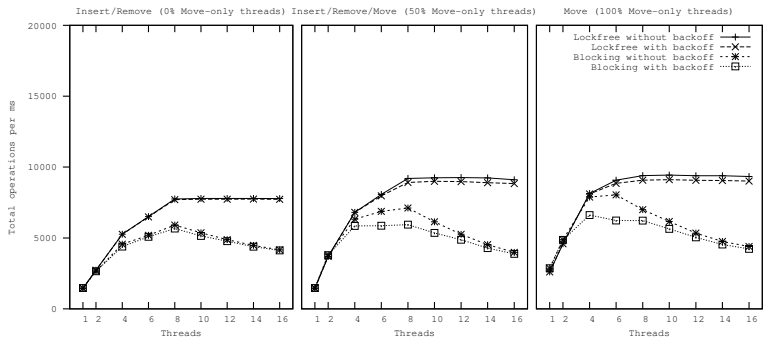


Figure 4.5: *Results from the queue evaluation under* low *contention.*

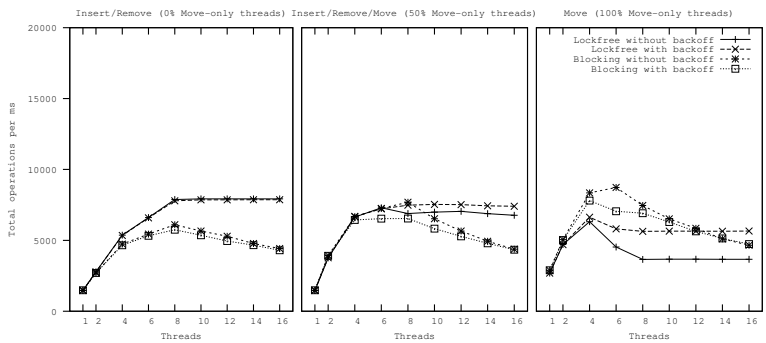


Figure 4.6: *Results from the stack evaluation under* low *contention.*

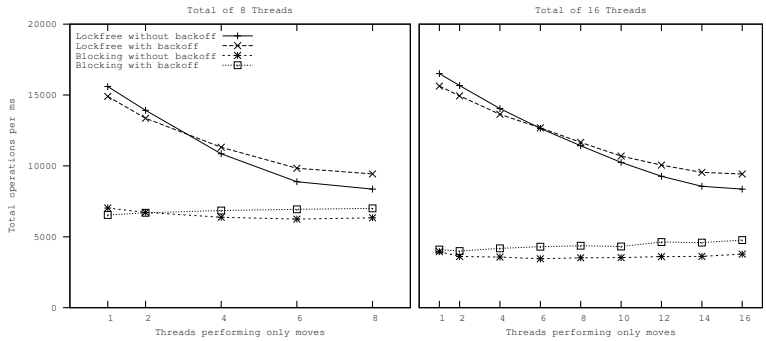


Figure 4.7: *Queue/Queue - Varying number of move-only threads under* high *contention.*

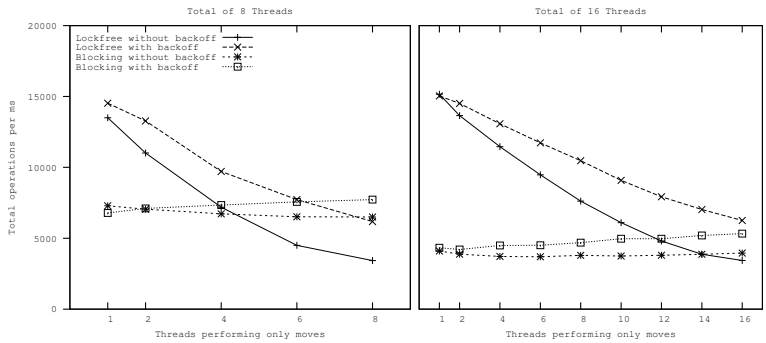


Figure 4.8: *Stack/Stack - Varying number of move-only threads under* high *contention.*

In the experiments two types of threads were used, one that performed only insert/remove operations, and one that only performed move operations. The number of threads, as well as the number of move-only threads, were varied between one and sixteen. We ran each experiment for five seconds and measured the number of operations performed in total per millisecond. Move operations were counted as two operations to normalize the result. Each experiment was run fifty times, taking the average as the results.

For reference we compared the lock-free concurrent objects with blocking implementations of the same objects, using test-test-and-set to implement the locks. We did the experiments both with and without a backoff function. The

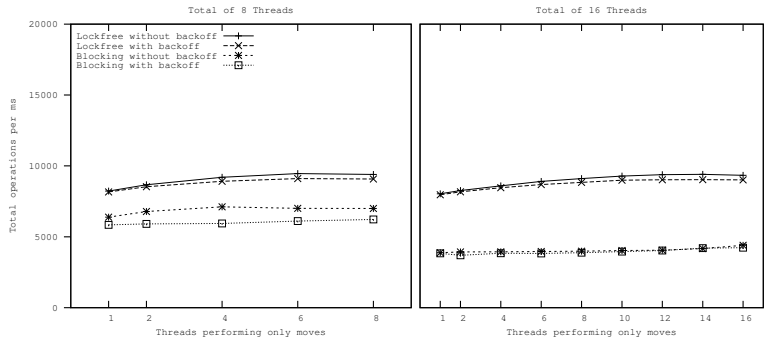


Figure 4.9: *Queue/Queue - Varying number of move-only threads under* low *contention.*

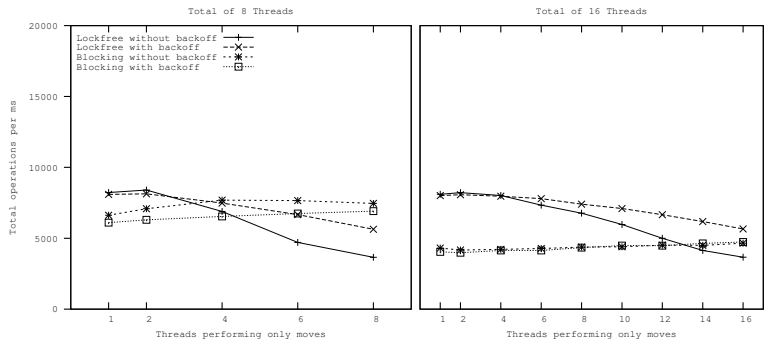


Figure 4.10: *Stack/Stack - Varying number of move-only threads under* low *contention.*

backoff function was used to lower the contention so that every time a process failed to acquire the lock, or, in case of the lock-free objects, failed to insert or remove an element due to a conflict, the time it waited before trying again was doubled.

All implementations used the same lock-free memory manager. Freed nodes are placed on a local list with a capacity of 200 nodes. When the list is full it is placed on a global lock-free stack. A process that requires more nodes accesses the global stack to get a new list of free nodes. Hazard pointers were used to prevent nodes in use from being reclaimed.

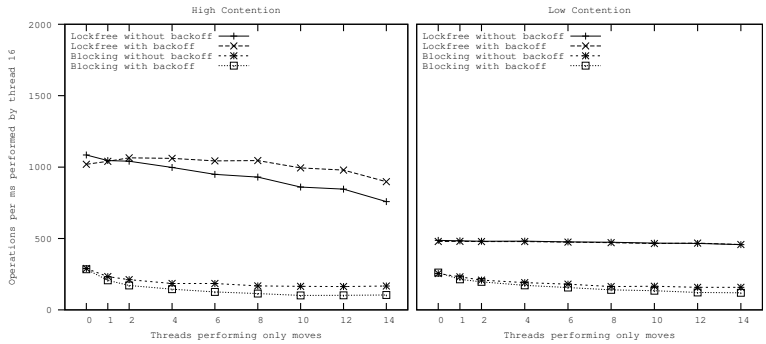


Figure 4.11: *Queue/Queue - Effect on thread performing insert/remove operations when other threads gradually turn into move-only threads.*

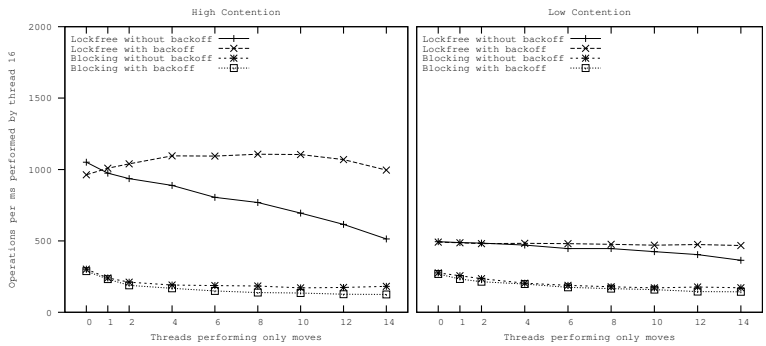


Figure 4.12: *Stack/Stack - Effect on thread performing insert/remove operations when other threads gradually turn into move-only threads.*

Two load distributions were tested, one with high contention and one with low contention, where each process did some local work for a variable amount of time after they had performed an operation on the object. The work time is picked from a normal distribution and the work takes around $0.1\mu s$ per operation on average for the high contention distribution and $0.5\mu s$ per operation on the low contention distribution.

Figures 4.1, 4.2 and 4.3 shows the number of operations in total per millisecond that was achieved performing either just insert/remove operations, or

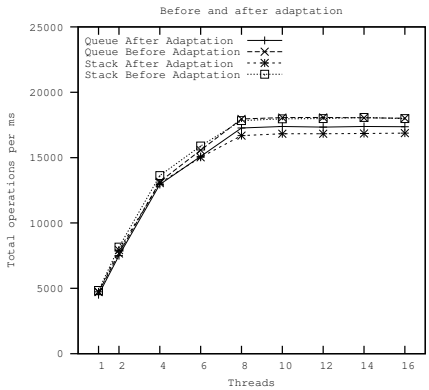


Figure 4.13: *Total number of operations per millisecond* before *and* after *adaptation of the stack and queue.*

just move operations, or an even mix of the two, under heavy contention. Figures 4.4, 4.5 and 4.6 shows the result for the same experiment performed under lower contention. In Figures 4.7 and 4.8 we take a closer look at what happens when the ratio of processes performing only move operations is varied using a total of either eight or sixteen processes under heavy contention. Figures 4.9 and 4.10 shows the results for the same experiments performed under lower contention.

Figure 4.13 shows how the performance of the insert/remove operations was affected by the adaptation to the move operation. In Figure 4.11 and 4.12 we take a closer look at how the performance of a thread, that is only performing insert/remove operations, is affected when gradually more threads go from performing just insert/remove operations to performing just move operations.

4.7 Discussion

The stack and queue have very few access points, which limits the offered parallelism. The experiments here are thus to be seen as showing some of the worst case scenarios.

In Figure 4.2, in the leftmost graph, we see that the performance increase sharply up to four threads, the number of cores on the processor, and then increases more slowly up to eight threads, the number of cores times two for hyper-threading. After eight threads there is no increase in performance as there are no more processing units. After this point the blocking version drops in performance when more threads are added.

When more move operation are performed, the performance does not scale as well, as can be seen in the two other graphs in Figure 4.2. The move operations are more expensive as they involve performing two operations and affects both data objects, which lowers the possible parallelism. For the queue the performance is still better than the blocking, however on the stack it is actually worse. Looking at Figure 4.8 we see that there is a threshold were the ratio of move-only threads makes the lock-free version worse than the blocking. This threshold does not appear to exist for the queue, which regardless of the ratio is faster than the blocking version, though more moves lowers the performance for the high contention case. For the low contention case the performance remains the same. In general the difference in performance between the blocking and lock-free methods becomes lower when the contention decreases. It should also be noted that it is not possible to combine a blocking move operation with non-blocking insert/remove operations.

In Figure 4.11 and 4.12 we see that the performance of a thread doing insert/remove operations is not affected much by the kind of operations that the other threads are performing, except for the lock-free stack without backoff. We can also see in Figure 4.13 that the adaptation, that is necessary for the data objects to support the generic move operation, hardly affects the performance of the normal operations.

Regarding the backoff, we can see in, for example, Figure 4.3, that with few move operations it hurts performance, whereas when the number of move operations increases it helps the performance. Unfortunately, it is typically hard to predict when this happens, making it difficult to design an optimal backoff function that works well in all scenarios.

4.8 Conclusion

We present a lock-free methodology for composing highly concurrent linearizable objects by unifying their linearization points. Our methodology introduces atomic move operations that can move elements between objects of different types, to a large class of already existing concurrent objects without having to make significant changes to them.

Our experimental results demonstrate that the methodology presented in the paper, applied to the classical lock-free implementations, offers better performance and scalability than a composition method based on locking. These results also demonstrate that it does not introduce noticeable performance penalties to the previously supported operations of the concurrent objects.

Our methodology can also be easily extended to support n operations on n distinct objects, for example to create functions that remove an item from one object and insert it into n others atomically.

Acknowledgments

This work was partially supported by the EU as part of FP7 Project PEPPHER (www.peppher.eu) under grant 248481 and the Swedish Research Council under grant number 37252706. Daniel Cederman was supported by Microsoft Research through its European PhD Scholarship Programme.

Bibliography

[AM95] James H. Anderson and Mark Moir. Universal Constructions for Multi-Object Operations. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 184–193, 1995.

[ARJ97] James H. Anderson, Srikanth Ramamurthy, and Rohit Jain. Implementing Wait-Free Objects on Priority-Based Systems. In *PODC*

'97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing, pages 229–238, 1997.

[CBM+08] Călin Caşcaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *Queue*, 6(5):46–58, 2008.

[DDG+04] Simon Doherty, David L. Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit, and Guy L. Steele, Jr. DCAS is not a Silver Bullet for Nonblocking Algorithm Design. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 216–224, 2004.

[DHLM04] Simon Doherty, Maurice Herlihy, Victor Luchangco, and Mark Moir. Bringing Practical Lock-Free Synchronization to 64-bit Applications. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 31–39, 2004.

[FR04] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59, 2004.

[GPT05] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. Allocating Memory in a Lock-Free Manner. In *ESA '05: Proceedings of the 13th Annual European Symposium on Algorithms*, pages 329–342, 2005.

[GPT09] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. NBmalloc: Allocating Memory in a Lock-Free Manner. *Algorithmica*, 2009.

[Har01a] Tim Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, 2001.

[Har01b] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, 2001.

[Her93] Maurice P. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[HFP02] Tim Harris, Keir Fraser, and Ian A. Pratt. A Practical Multi-word Compare-and-Swap Operation. In *DISC '02: Proceedings of the 16th International Conference on Distributed Computing*, pages 265–279, 2002.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice P. Herlihy. Composable Memory Transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, 2005.

[HT03] Phuong Hoai Ha and Philippas Tsigas. Reactive Multi-Word Synchronization for Multiprocessors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 184–193, 2003.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[Int09] Intel. Threading Building Blocks, 2009.

[IR94] Amos Israeli and Lihu Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *PODC '94: Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 151–160, 1994.

[Lea09] Doug Lea. The Java Concurrency Package (JSR-166), 2009.

[LK08] James Larus and Christos Kozyrakis. Transactional Memory. *Communications of the ACM*, 51(7):80–88, 2008.

[Mic02] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82, 2002.

[Mic04] Maged M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[Mic09] Microsoft. Parallel Computing Developer Center, 2009.

[Moi97] Mark Moir. Transparent Support for Wait-Free Transactions. In *WDAG '97: Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319, 1997.

[MS96] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

[ST95] Nir Shavit and Dan Touitou. Software Transactional Memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, 1995.

[ST02] Håkan Sundell and Philippas Tsigas. NOBLE: A Non-Blocking Inter-Process Communication Library. In *Proceedings of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*, Lecture Notes in Computer Science, 2002.

[ST04] Håkan Sundell and Philippas Tsigas. Scalable and lock-free concurrent dictionaries. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1438–1445, 2004.

[ST05] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, 2005.

[Tre86] R. K. Treiber. Systems programming: Coping with parallelism. In *Technical Report RJ 5118*, April 1986.

[TZ01] Philippas Tsigas and Yi Zhang. Evaluating the Performance of Non-Blocking Synchronization on Shared-Memory Multiprocessors. *ACM SIGMETRICS Performance Evaluation Review*, 29(1):320–321, 2001.

[TZ02] Philippas Tsigas and Yi Zhang. Integrating Non-Blocking Synchronisation in Parallel Applications: Performance Advantages and Methodologies. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, pages 55–67, 2002.

[Vaf09] Viktor Vafeiadis. Shape-Value Abstraction for Verifying Linearizability. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 335–348, 2009.

[Val95] John D. Valois. Lock-free linked lists using compare-and-swap. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222, 1995.

PAPER IV

Daniel Cederman, Philippas Tsigas, Muhammad Tayyab Chaudhry
Towards a Software Transactional Memory for Graphics Processors
In the Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization (EGPGV 2010)
pages 121-129, Eurographics Association 2010.

5

Towards a Software Transactional Memory for Graphics Processors

The introduction of general purpose computing on many-core graphics processor systems, and the general shift in the industry towards parallelism, has created a demand for ease of parallelization. Software transactional memory (STM) simplifies development of concurrent code by allowing the programmer to mark sections of code to be executed concurrently and atomically in an optimistic manner. In contrast to locks, STMs are easy to compose and do not suffer from deadlocks. We have designed and implemented two STMs for graphics processors, one blocking and one non-blocking. The design issues involved in the development of these two STMs are described and explained in

the paper together with experimental results comparing the performance of the two STMs.

5.1 Introduction

Computer processor research has previously been focused on increasing the clock speed, but as of late the trend has shifted towards increasing the number of processors instead. This has led to increased pressure for applications to become multi-threaded to take full advantage of the new computing power. But with increased parallelism comes the problem of efficient synchronization. Threads that concurrently access shared memory have to synchronize in order to maintain a non-corrupted view of the data.

The traditional way of synchronizing memory accesses has been to use mutual exclusion, using locks to only allow one process to access shared memory areas at any given time. However, this kind of lock-based synchronization makes it hard to compose function calls and leads to problems such as deadlocks, where two processes are both waiting for the other to give up a lock, and convoying, where a process that holds a lock gets swapped out causing other processes to wait unnecessarily long to acquire that lock.

Transactional memory (TM) provides an alternative concurrency control that can eliminate these problems or at least minimize them. A TM allows the programmer to mark a section of the code that is to run atomically, i.e., it should appear to take place instantly. The TM logs all read and write operations in the code block and only store the new data if there was no conflict with another process. If a transaction notices that another transaction has written to memory read in the transaction, the transaction will be restarted. The lack of commonly available hardware transactional memories has led to most implementations of transactional memory being completely software based, so-called Software Transactional Memories (STM).

An STM tries to automatically offer some degree of parallelism to the application without having the programmers concentrate on the mechanism of synchronization, as this is taken care of by the STM itself. There is, however,

a tradeoff when it comes to performance. Code written using STMs often has difficulties competing in performance compared to solutions that are highly optimized by hand. Caşcaval et al. argue that the overhead introduced by STMs might be hard to overcome [CBM+08].

The high bandwidth and many-core design of current graphics processors have caused a big interest in applying them for general purpose computing. With APIs such as CUDA and OpenCL it is only a matter of time before they become standard auxiliary processing units that most programmers would like to take advantage of in their applications.

In this paper we examine if an introduction of STMs to graphics processors could help simplify the relatively complex amount of synchronization needed when running programs on a many-core platform. More specifically, we evaluate two STM designs, with two different types of progress guarantees, in an effort to better understand the specific challenges involved providing synchronization for many-core graphics processor systems.

5.2 Related work

Transactional memory was originally intended for hardware and was first introduced in a paper by Knight and a paper by Herlihy and Moss [Kni86, HM93]. Shavit and Touitou then later introduced the concept of a pure software transactional memory [ST95]. Their STM required the programmer to specify beforehand which memory locations to access and could not adapt to values read in the transaction.

This was changed in Herlihy et al.'s dynamic STM (DSTM) where they also introduced the concept of a contention manager that should decide which transaction to abort [HLMS03]. Harris and Fraser have presented an STM that works on the word level as opposed to the object level, called WSTM [FH07].

The previously mentioned STMs perform their operations on local copies of the objects or words which are then either discarded or written back, but it is also possible to take a more optimistic approach and write directly to the objects or words [ATLM+06, HPST06, SATH+06]. This, however, requires the

STM to store the original values so that the changes can be undone if there is a conflict and also introduces the problem of visibility – should other transactions be able see the values written?

There has also been work on creating hybrid transactional memories that use both hardware and software [DFL+06, KCJ+06]. If the hardware has support for transactional memory, the transaction is started in hardware and handled there until it gets too large for the hardware to support. In those cases the transaction is taken over by the STM.

Ennals argues that STMs should be blocking as the advantages of doing a non-blocking implementation are small and prevents several optimizations that can be done in a blocking system [Enn06].

There is a large amount of literature on designing STMs [DSS06, HF03, Moi97] and for a good overview we recommend the *Transactional Memory* paper by Larus and Kozyrakis [LK08].

5.3 System Model

CUDA was introduced by NVIDIA as a general purpose parallel computing architecture, making it possible to execute computationally complex problems on NVIDIA's graphics processors. The software part of CUDA provides a compiler for a language based on C, but with extensions that allow functions to be executed on the graphics processor instead of on the CPU.

A CUDA compatible graphics processor consists of several so called multiprocessors, each of which can execute SIMD instructions on eight memory locations at a time. Threads are scheduled on the multiprocessors in groups called thread blocks. All threads in a thread block remain at the same multiprocessor until they have finished executing and can use the processor's extremely fast local memory, called the shared memory, to communicate with each other. It is up to the programmer to decide how many threads should be in each block and how many blocks to start in total. Depending on how many threads there are in a block, one or more blocks could run on the same multiprocessor.

```
atomic {
    ltail = read(tail);
    write(queue[ltail], value);
    ltail++;
}
```

Figure 5.1: *Example use of a software transactional memory.*

Threads of the same as well as different blocks can perform read and write operations on the main graphics memory known as the global memory. There is no cache support when using the global memory, but if threads with consecutive thread id's are accessing consecutive memory locations, the memory accesses could be coalesced by the hardware to dramatically speed up the reading and writing speed to the memory. There are also texture and constant memory that have cache support, but they are read-only.

Newer versions of CUDA-compatible graphics processors support atomic primitives, such as Compare-And-Swap, which can be used, for example, to implement locks or more advanced lock-free data-structures [Her88, PDC09]. Cederman and Tsigas took advantage of this to compare blocking and non-blocking dynamic load balancing schemes on graphics processors [CT08].

5.4 STM Design

On the user level, a basic software transactional memory needs to support, either directly or indirectly, four operations. A *begin* operation that marks the start of a transaction, a *read* operation that provides a snapshot of a memory location, a *write* operation that logs the updates to the memory that should be performed if the transaction is successful, and finally a *commit* operation that performs the writes if no other processes have touched the memory read by the transaction and restarts the transaction otherwise. The begin and commit operations are often performed indirectly, as in Figure 5.1, where they are part of the *atomic* keyword.

Features	**Blocking STM**	**Non-Blocking STM**
Progress Guarantee	Blocking	Obstruction-free
Conflict Detection Time	Commit-time	Commit-time
Locks	Shared	Unique
Conflict Handling	Aborts the transaction	Steals locks or aborts other transaction
Conflict Detection	Object based	Object based
Visibility	Local updates	Local updates
Log or Undo-Log	Log	Log

However, when deciding how to implement the functionality behind these operations, there are several important design decisions that have to be made. There is a vast design space for STMs and as of yet there is no definitive way to design an STM. One major divisional line is that which progress guarantees to provide. More basic guarantees can achieve better performance under low contention, while more advanced guarantees can give more independence from the scheduler at a cost in complexity. We argue that this design choice is one of the most important, as most graphics processors perform their scheduling in hardware in non-standard ways. For this reason we have implemented two different STMs that differ mainly in the type of progress guarantees that they provide. The first STM is designed to be as simple as possible to lower resource requirements and improve performance. This will be known for the remainder of this paper as the *blocking STM*. The second is based on the STM by Harris and Fraser, which is more complex and designed for general multiprocessors, but offers better progress guarantees [HF03].

In the following subsections we will go through some of the different design parameters and elaborate on our design decisions.

5.4.1 Progress Guarantees

Progress guarantees are often divided into one of four categories. The strongest is *wait-freedom*, which guarantees that, in the context of STMs, a transaction always succeeds in a bounded number of its own steps. This guarantee is typically only provided in real-time systems, where predictability is critical, as it often hampers performance. A weaker and more practical guarantee is *lock-freedom*, which guarantees that at least one transaction will be successful in a bounded number of its own steps. A transaction in a lock-free STM is always able to make progress, even in the case of all other threads controlling transactions being suspended. Despite the name, lock-freedom does not preclude locks, as long as these can be revoked. An even weaker guarantee is *obstruction-freedom*. It guarantees that a transaction will always succeed if it is executed without conflicts with other transactions. A contention manager is often used to achieve this, by arranging for one of the conflicting transactions to back off. The final category includes the *blocking* algorithms, which uses irrevocable locks and provides no guarantees at all.

Despite the lack of progress guarantees, we decided to make the first STM we designed blocking. This allowed for a simpler design and, according to Ennals, a potentially more efficient implementation [Enn06]. The disadvantage of using a blocking implementation is that it makes the STM much more dependant on the scheduler. We had concerns of whether the hardware scheduler on the graphics processor would be able to handle locks or not, as if the scheduler is not fair, it could swap out the lock holder and repeatedly just schedule the processes waiting for the lock. However, we experienced no such problems during our experimentation.

The second STM that we designed is based on the STM by Harris and Fraser and is an obstruction-free one [HF03]. When a transaction is to be committed, it tries to acquire all the locks it requires to get exclusive access to its write locations. But, at the same time, it publicly announces the actual values that it is going to write. This gives conflicting transactions two options. If the transaction has managed to acquire all locks, but not yet written the new values, the conflicting transaction can steal locks from it, using the new value that is to

be written. If the transaction has not yet acquired all its locks, the conflicting transaction may abort the original transaction before attempting to acquire its own locks. As a transaction never has to wait for another transaction to finish, the STM is non-blocking.

5.4.2 Conflict Detection Granularity

STMs are often designed with different levels of granularity for conflict detection depending on the language they are written for. For object oriented languages, such as Java, it is often more convenient to use objects as the basic unit. Two transactions accessing the same object will then conflict, even if they are accessing different fields in the object. This is known as a false conflict. For languages such as C, with no standard object type, it is more common to use individual words as the basic unit. Often, to lower the overhead of having a lock for each individual word, the memory is divided into several stripes, where every n:th word shares a lock. As multiple words might share the same lock there is a potential for false conflict, but this can be mitigated by increasing the number of strips.

For both the blocking and non-blocking STM we decided to put the granularity at the object level. The reason for this is that we wanted to take advantage of the graphics processor's ability to coalesce memory reads and writes into larger memory operations. For the blocking we shared locks between objects, whereas for the non-blocking we had one lock per object.

5.4.3 Log or Undo-Log

As there is normally no way of knowing if a transaction will succeed before it has tried to commit, there must be a way to undo transactions. The most common way is to keep a thread-local log where the changes to be performed are stored. The first time a word or object is read, a copy of it is stored in the log. All subsequent writes are then performed on the local copy. When all locks have been acquired at the end of the transaction, the items in the log are written to the shared memory. An alternative, and more optimistic approach, is

to acquire the write locks immediately at the first write and then store the data written inside the transaction directly to the shared memory. This is faster in cases where there are no conflicts, but to be able to abort, there needs to be an undo log that holds the old values of the words or objects written to.

Both STMs use the log method as we expect much contention and we want to avoid the problem with visibility, which occurs when other transactions read data that is yet to be committed. In a non-managed environment, this might lead to infinite loops or crashes.

5.4.4 Conflict Detection Time

Most STMs use some incarnation of a lock to provide mutual exclusion when writing the result from the transaction. These locks can either be acquired early, the first time that the word or object they protect is accessed, or as late as at commit time. Acquiring locks immediately have the advantage that conflicts will be detected early, but this might also, unfortunately, lead to more false conflicts. To assure that the transaction is not working on inconsistent data, it is possible to do the read validation whenever data is read or written.

The design decision here was that the blocking STM should use the same method as the non-blocking and lock at commit time.

5.4.5 Backoff

A backoff function is used whenever there is a conflict and a transaction needs to abort. It forces the process to wait before it tries to perform the transaction again. This lowers contention and increases the probability that at least one transaction is successful.

Backoff is often an important part of the contention management in STMs, together with the policy choice of which transaction to abort in case of a conflict. There are several ways of backing off, including linear, where the time to back off is increased linearly for every abort, and exponential, where the time to wait is, for example, doubled each time [GHP05, SS05]. We designed the STMs to be able to use both linear and exponential backoff.

5.5 Implementation

As mentioned in the STM Design section, an STM needs to support four basic operations. The following subsections will detail the implementation specifics for the blocking STM. For the non-blocking STM we refer to the paper by Harris and Fraser [HF03].

5.5.1 Begin

Each thread block is assigned a transaction descriptor to keep track of objects that have been read and objects that should be written back at commit time. Figure 5.2 shows an entry in the transaction descriptor. When a new transaction is initiated the transaction descriptor is cleared. The member variables in the descriptor will be motivated in the following subsections.

```
struct TransDescItem {
    int version;
    void* global;
    void* local;
    int size;
    bool readonly;
}
```

Figure 5.2: *Transaction descriptor item.*

```
struct VersionLock {
    int version(31bit);
    bool lock(1bit);
}
```

Figure 5.3: *Combined version number and lock.*

5.5.2 Read

The read operation transfers an object from the publicly available global memory into a private part that only the reading thread block has access to. A check is made to see if the lock that covers the object is taken. This is to make sure that no other thread block is currently writing to the object and to wait if anyone is. The lock consists of a version number with a lock-bit, as can be seen in Figure 5.3. As the lock-bit is the least significant bit, one can interpret odd values of the lock as the lock being taken and even values as the lock being available.

A copy of the version number of the object is stored in the transaction descriptor and the object is copied to the local part of the memory. The version number is read again and compared to the one in the transaction descriptor. If they match, then the local copy of the object represents a consistent snapshot of the object. If they do not match, another thread block must have written to it and we have to read it again until the version numbers matches.

A pointer to the local copy and a pointer to the public object is stored in the transaction descriptor once the object have been successfully read, together with a marker that indicates whether the local copy has been updated or not. The pointers are needed so that the commit operation knows where to write back the local copy.

The read operation then returns a pointer to the local copy of the object. If there already exists a local copy of the object, due to it being read earlier in the transaction, it is just a matter of returning the pointer to that local copy.

5.5.3 Write

When a local copy of an object has been updated, it needs to be marked as such so that it is updated at commit time. This is done by going through the transaction descriptor looking for the pointer to the object and then marking it when found. Since all writes are being performed locally, there is no need for any locks in this phase.

5.5.4 Commit

At commit time the STM needs to make sure that no other thread block has changed any of the objects read or written to inside the transaction before it can write back the updated objects. The objects that are to be updated are therefore checked to see if their current version number matches the ones in the transaction descriptor. If they do, the version number is incremented by one atomically using Compare-And-Swap to lock the objects. Using Compare-and-Swap, this will only succeed as long as the version number has not changed in the mean time. The thread that locked the object now has exclusive access to it. Any failure in acquiring write locks, or version numbers that do not match, causes the transaction to abort. The updated objects are then written back to public memory once all locks have been acquired. The locks are released by increasing their version number by one and the transaction is successful. By combining the version number and the lock we make sure that any concurrent read invocation does not see any intermediate state during the writing back of the updated object to the global memory.

5.6 Experimental Evaluation

For evaluation we used four concurrent data-structures. All of them used software transactional memory in their design. We measured their respective performance when faced with different contention levels and using different back-off strategies. In addition to measuring the number of operations per second, we also measured the number of aborted transactions in order to better understand their respective behavior and how it affects the given performance.

5.6.1 Hardware

The experiments were performed on the high-end graphics processor GTX280 with 30 multiprocessors. Each multiprocessor has 8 cores, giving us a total of 240 cores. The processor clock rate is close to 1.3 GHz and the optimal memory bandwidth is 141 GB per second.

5.6.2 Test-Bed Applications

Binary Tree Each thread block inserts a fixed quantity of randomly picked values, uniformly distributed, into a binary tree. As the tree grows wider there should be fewer conflicts.

Queue Each thread block performs an even amount of enqueue and dequeue operations on a single queue. This benchmark should provide the highest level of contention as only one enqueue or dequeue operation can take place at any given time.

Hash-map Each thread block inserts a fixed amount of randomly picked values, uniformly distributed, into a hash-map with 128 buckets, each bucket being an individual list. This benchmark is similar to the queue benchmark, but lowers contention by dividing access to it over several buckets. The transactions are longer since they need to find the end of the list before they can insert their element.

Skip-list Each thread block performs an even amount of insert, find, and delete operations on a skip-list with a maximum of 7 levels. This is a more complex benchmark that is expected to scale similarly to the tree. To compare the performance of the respective STMs with a highly parallel design of an advanced data-structure, we also compared the respective STM skip-list implementations with the lock-free skip-list by Sundell and Tsigas [ST04].

5.6.3 Experiment Settings

To see how the STMs react to different contention levels we have tested them with two scenarios. One where the test application performs some local work before accessing the data-structure, a low contention scenario, and one high contention scenario where there is no pause between transactions. The time for the local work is picked randomly after each transaction from a uniform

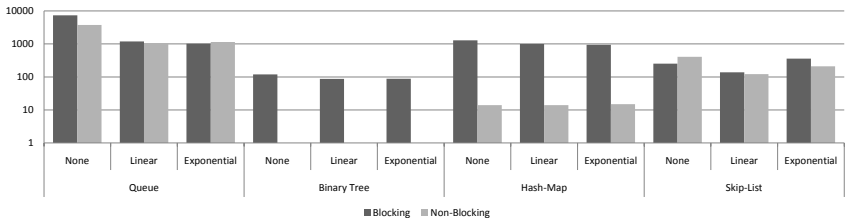


Figure 5.4: *Average number of aborts per transaction with* **low** *level of contention using 60 thread blocks (***logarithmic scale***).*

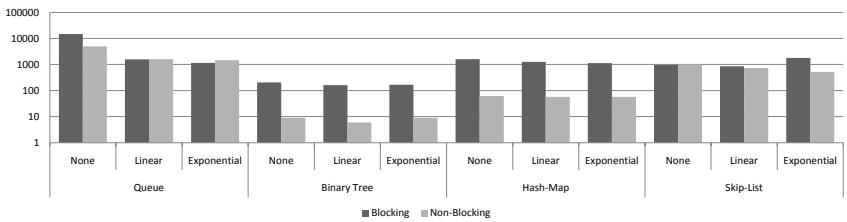


Figure 5.5: *Average number of aborts per transaction with* **full** *level of contention using 60 thread blocks (***logarithmic scale***).*

distribution and takes a total of ∼450 ms for one thread block to complete and around ∼500 ms for 60 thread blocks to complete in parallel.

Since the choice of backoff-function is important, we did each experiment with two types of backoff, one linear and one exponential. We also performed the experiments using no backoff at all.

Each of the data-structures were evaluated with a varying number of thread blocks. We did not vary the number of threads in each thread block, as we are only synchronizing the accesses by the thread blocks and not the individual threads within a block. The measurements were repeated 50 times.

5.7 Discussion

At a low level of contention, the blocking and the non-blocking skip-list and binary tree both scale well, see Figure 5.8. Looking at the number of aborts for

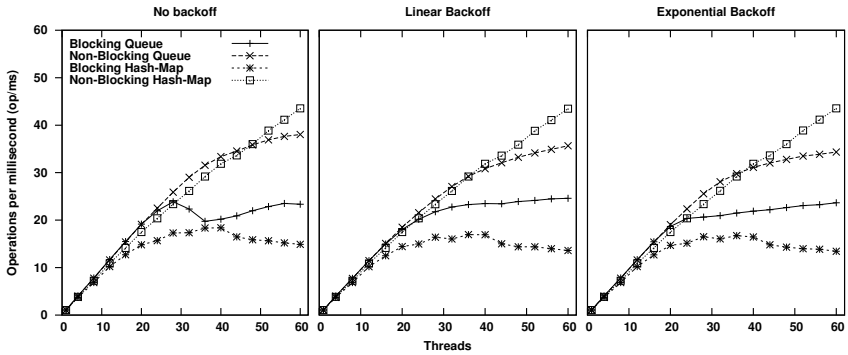


Figure 5.6: *Experimental result for the queue and hash-map with* **low** *level of contention.*

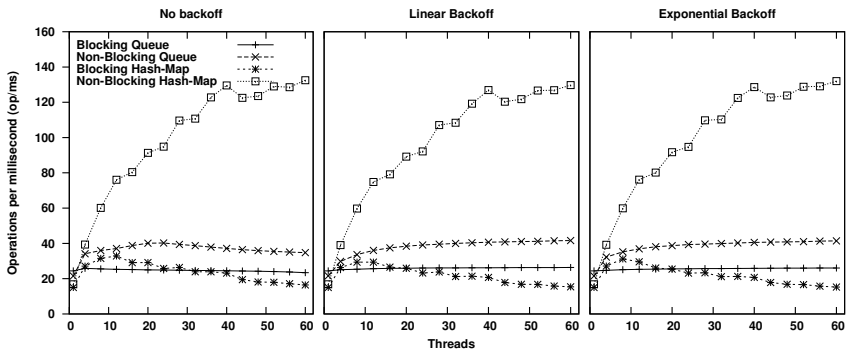


Figure 5.7: *Experimental result for the queue and hash-map with* **full** *level of contention.*

the skip-list, Figure 5.4, one can see that the average number of aborts is about the same, whereas for the binary tree there is a distinct difference. The blocking STM has an average of over one hundred aborted transactions for each thread block, while the non-blocking STM has none at all. With greater contention, Figure 5.5, the number of aborts remains the same for the blocking STM while it has increased to ten for the non-blocking STM. Despire this, the performance in number of operations per ms is much better for the non-blocking STM; see Figure 5.9.

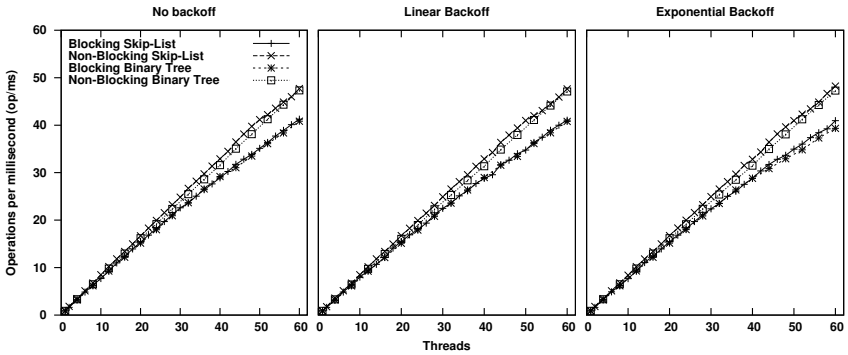


Figure 5.8: *Experimental result for the binary tree and skip-list with* **low** *level of contention.*

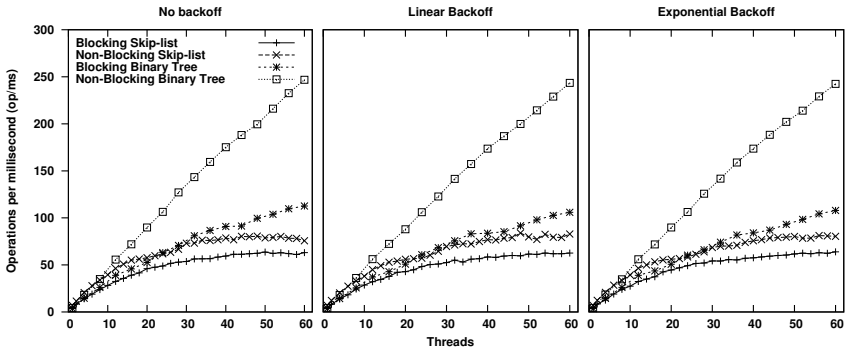


Figure 5.9: *Experimental result for the binary tree and the skip-list with* **full** *level of contention.*

In Figures 5.4 and 5.5 we see that the backoff has quite a large effect on the average number of aborts for the queue and that there does not seem to be any difference between the linear and the exponential backoff. However, the backoff does only slightly alter the number of operations per ms, which can be seen in Figures 5.6 and 5.7. For the other benchmarks there is hardly any difference between the results for the different backoff schemes, both when it comes to operations per second and when it comes to aborted transactions per number of thread blocks.

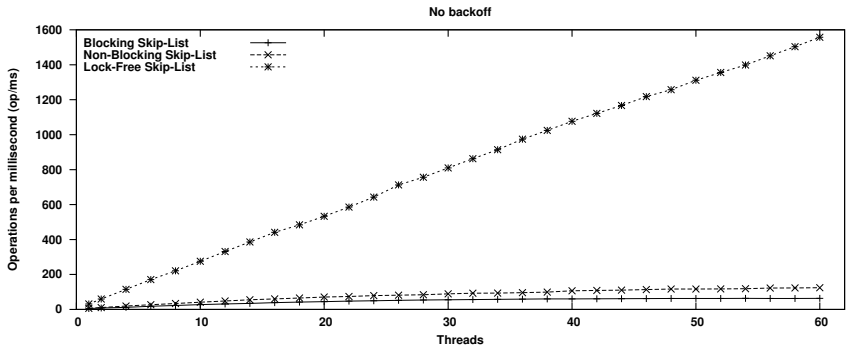


Figure 5.10: *Experimental result for the STM skip-lists compared with the skip-list by Sundell and Tsigas with* **full** *level of contention [ST04].*

The queue does not scale well for either the blocking or non-blocking STM. This is not surprising since only one enqueue or dequeue operation can take place at any given time. The hash-map and binary tree both scale much better with the non-blocking STM and it can be clearly seen that the non-blocking has a lot fewer aborted transactions when it comes to these benchmarks. This can be attributed to the fact that the non-blocking version can steal locks to continue working without aborting.

In Figure 5.10 the result from the comparison between the respective STM skip-lists and the lock-free skip-list by Sundell and Tsigas is presented [ST04]. By the figure it is clear that the respective STM skip-lists are significantly slower than the non-STM skip-list. This is to be expected, as one can gain a lot in performance by using more complex synchronization techniques during the design phase of the data-structure. However, there is a trade-off, as these techniques require much more time and expertise to get the design right, than to use an STM.

5.8 Conclusion

Software Transactional Memory has attracted the interest of many researchers over recent years. We have designed and implemented two STMs for graphics

processors, one blocking and one non-blocking. The design issues involved in the designing of these two STMs are described and explained in the paper together with experimental results comparing the performance of the two STMs. We found that while a blocking STM is simpler to implement, providing additional progress guarantees, such as obstruction-freeness, improves performance and lowers the number of aborted transactions.

Acknowledgements

This work was partially supported by the EU as part of FP7 Project PEPPHER (www.peppher.eu) under grant 248481 and the Swedish Research Council under grant number 37252706. Daniel Cederman was supported by Microsoft Research through its European PhD Scholarship Programme.

Bibliography

[ATLM+06] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM.

[CBM+08] Călin Caşcaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *Queue*, 6(5):46–58, 2008.

[CT08] Daniel Cederman and Philippas Tsigas. On Dynamic Load Balancing on Graphics Processors. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 57–64, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[DFL+06] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum. Hybrid Transactional Memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 336–346, 2006.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[Enn06] Robert Ennals. Software Transactional Memory Should Not Be Obstruction-Free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[FH07] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2):5, 2007.

[GHP05] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)*, volume 3724 of *Lecture Notes in Computer Science*, pages 303–323, 2005.

[Her88] Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 276–290, New York, NY, USA, 1988. ACM.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, volume 38, pages 388–402, New York, NY, USA, November 2003. ACM Press.

[HLMS03] M. Herlihy, V. Luchangco, M. Moir, and W.N. Scherer. Software Transactional Memory for Dynamic-sized Data Structures.

In *Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2003.

[HM93] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support For Lock-Free Data Structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[HPST06] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 14–25, New York, NY, USA, 2006. ACM Press.

[KCJ+06] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid Transactional Memory. In *Proceedings of Symposium on Principles and Practice of Parallel Programming*, Mar 2006.

[Kni86] Tom Knight. An architecture for mostly functional languages. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112, New York, NY, USA, 1986. ACM Press.

[LK08] James Larus and Christos Kozyrakis. Transactional memory: Is TM the answer for improving parallel programming? volume 51, New York, NY, USA, 2008. ACM.

[Moi97] Mark Moir. Transparent Support for Wait-Free Transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319. Springer-Verlag, 1997.

[PDC09] L. Santos P. Dubla, K. Debattista and A. Chalmers. Wait-Free Shared-Memory Irradiance Cache. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 57–64. Eurographics, March 2009.

[SATH+06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM.

[SS05] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 2005. ACM.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[ST04] Håkan Sundell and Philippas Tsigas. Scalable and lock-free concurrent dictionaries. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1438–1445, New York, NY, USA, 2004. ACM.

6
Future Work

The algorithmic design of data structures needs to be updated to meet the new challenges drawn up by the move from single- to multi- to many-core processors. The blocking mechanisms conventionally used to synchronize access to data needs to be replaced, as they limit scalability and are prone to suffer from dead-locks, convoying and busy-waiting. We argue in this thesis that lock-free synchronization, with its positive approach towards conflict handling, offers better scalability and is, by definition, immune to many of the problems associated with blocking schemes. So far, most work on lock-free synchronization has been focused on data structures that are based on lists. As future work it would be interesting to take a closer look at different types of trees and graphs as possible candidates for new lock-free data structures.

In this thesis we have discussed load balancing and work-stealing, and it is clear that the choice of data structures used in work-stealing schemes play a significant role. Much work has been done in the area of load balancing, and we will continue to look into this field and provide lock-free versions of the data structures that are commonly used.

As part of the thesis we have presented some initial steps towards practical and efficient composition of lock-free data structures. But while the move operation and the ability to insert/remove elements into/from multiple data structures are important pieces, much work remains before we have well defined support for generic compositions. An important next step is to extend the methodology presented, so that it in addition to the previously mentioned compositions, it also can support composition of multiple operations on the *same* data structure. This will be, by necessity, more complex to do efficiently, as it is likely that more speculative changes to the data structure needs to be made. Another interesting venue connected to composition, is the combination of lock-free data structures, for high performance, with software transactional memories, for ease of use. So far these have been hard to combine in an efficient manner, but if successful, they could prove to be a powerful combination.

It is likely that computer systems will become more and more heterogenous, and data structures will need to be adapted to take full advantage of this change. Support needs to be added for multiple address-spaces and the movement between them. A data structure that is used on a heterogenous platform will need to provide operations that are targeted towards the different kinds of processors and access patterns. For example, insert operations that are accessed from graphics processors needs, for efficiency, to support SIMD instructions. These operations in turn needs to be able to cooperate with insert operations that uses normal SISD instructions.

Other possible research directions include reactive and context aware data structures. Access patterns and compositions affects the performance as well as the progress guarantees of the data structures operations. Efficient ways to adapt to these changes automatically could provide significant performance improvements.