Contents

6.3 Producer Consumer Queue

To evaluate our technique for creating a non-blocking algorithm we compare the ease of use of a bounded non-blocking Producer Consumer Queue with that of a similar queue described in the literature. We also compare the performance of our queue with that of its counterpart implemented using mutual exclusion. We find that our queue is more flexible than the non-blocking queues described in the literature. We also find that our queue performs similarly to a queue implemented using mutual exclusion.

The Producer Consumer Queue is a concurrent design pattern. A bounded Producer Consumer Queue can act as a message queue for Inter-Processor Communication. Two classes of processors, the producers and the consumers, share a common buffer which acts as a queue of messages between them. A producer adds a message to the queue and a consumer removes it. The Producer Consumer Queue guarantees that a producer cannot add a message onto the queue when it is full and that a consumer cannot remove a message when it is empty and that each message is consumed exactly once.

The main contribution of this section is an evaluation of a Producer Consumer Queue implemented by an Immutable Data Structure. This section focuses on comparing: throughput, ease of implementation, ease of use, scalability and progress guarantees.

6.3.1 Experiment

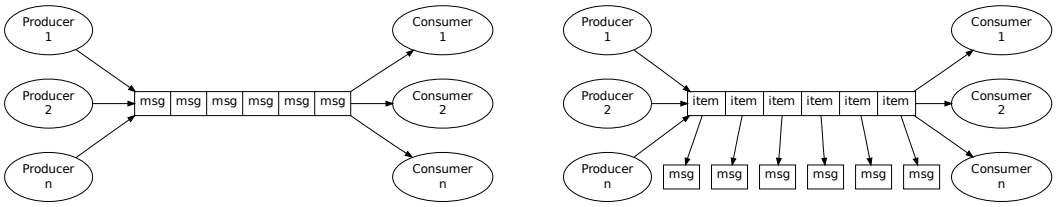
Our experiment compares the performance of a lock-free bounded Producer Consumer Queue implemented by an Immutable Data Structure with that of a blocking bounded Producer Consumer Queue implemented using mutual exclusion.

Section 6.3.3 describes the implementation of the Producer Consumer Queues and the experimental set up.

We call a Producer Consumer Queue that transmits messages in a buffer a Mailbox Queue and a queue that transmits references to messages a Messaging Queue. We are primarily interested in transmitting messages between physical processors so each end of the queue is accessed by thread of execution on a different physical processor.

Figure 6.1 illustrates the Producer Consumer Queue design pattern.

The production and consumption of messages by an application effects the



(a) Mailbox Queue (b) Messaging Queue

Figure 6.1: **The Producer Consumer Queue** design pattern.
(a) A Mailbox Queue acts as a buffer for fixed sized messages sent between producers and consumers.
(b) A Messaging Queue transmits messages referenced by items in the buffer between producers and consumers.

performance of the queue by introducing latency. Our experiment examines how the throughput of the Messaging Queue varies depending upon this latency.

Section 6.3.4 describes a simulated message workload.

We examine whether our lock-free Producer Consumer Queue is easier to implement than a similar non-blocking queue. We also compare the ease of use, the scalability and the progress guarantees offered by our queue with those of other blocking and non-blocking queues.

6.3.2 Results

This thesis does not make any claims about the absolute performance of Transactional Data Structures. However, the results of our experiment show that the performance of the non-blocking Producer Consumer Queue was broadly similar to that of a queue implemented using mutual exclusion.

Maximum throughput of the Mailbox Queue

The maximum throughput of a Mailbox Queue implemented by the non-blocking Producer Consumer Queue is compared with that of blocking queues from the Boost C++ library [Kar05].

Table 6.1 lists the elapsed time taken to transmit messages between two processors for various queue types.

Our non-blocking Producer Consumer Queue has the lowest overall execution time. The implementation based on a deque from the standard library has the

Algorithm	Elapsed time (s)
Non-blocking Producer Consumer Queue	**0.24**
boost bounded circular buffer	0.28
boost bounded space optimised circular buffer	0.30
boost bounded std::deque container	0.25
boost bounded std::list container	0.85

Table 6.1: **The maximum throughput of a Mailbox Queue.** The elapsed time taken to transmit one million mailbox messages between two processors for various queue types is listed. The experiment determines the maximum throughput of a Mailbox Queue with a capacity of one thousand 8 byte messages. Figures given are the mean of 10 observations.

lowest elapsed execution time of the blocking implementations.

Section 6.3.6 discusses the performance of the Mailbox Queue in detail.

We conclude that the maximum throughput of our mailbox Producer Consumer Queue is similar to that of its blocking counterpart.

Maximum throughput of the Messaging Queue

The maximum throughput of a Messaging Queue implemented by the non-blocking Producer Consumer Queue is compared with that of a blocking queue from the Boost C++ library.

Figure 6.2 and figure 6.3 illustrate the elapsed time taken to transmit messages between processors while varying the latency of production and consumption.

The blocking queue has a lower elapsed time than the non-blocking queue, regardless of the latency incurred by either the producer or the consumer. The difference between the throughput of the queues becomes more pronounced as the latency increases. When there is an imbalance between the latency of the producer and that of the consumer the elapsed time taken by the non-blocking queue is significantly longer than that taken by the blocking queue.

Section 6.3.7 discusses the performance of the Messaging Queue in detail.

We conclude that the maximum throughput of our messaging Producer Consumer Queue is lower than that of its blocking counterpart.

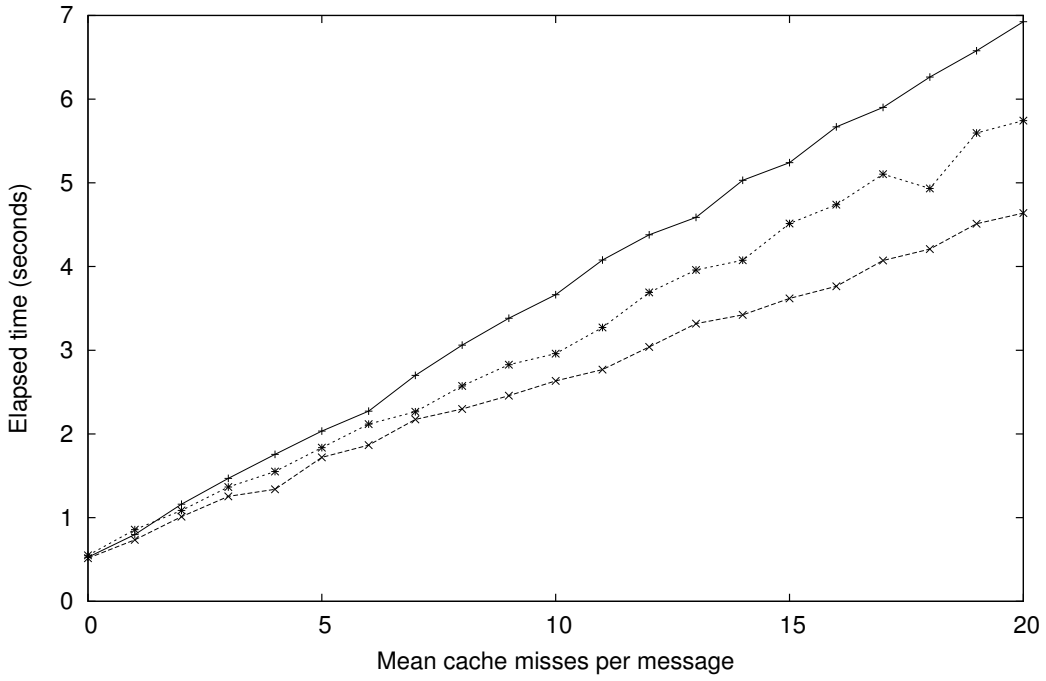


Figure 6.2: **The maximum throughput of a non-blocking bounded Messaging Queue implemented by a confluently persistent Immutable Data Structure.** The elapsed time taken to transmit one million messages between two processors is plotted against a varying number of forced cache misses incurred while: producing (*), consuming (x) and both producing and consuming the messages (+). Figures given are the mean of 10 observations.

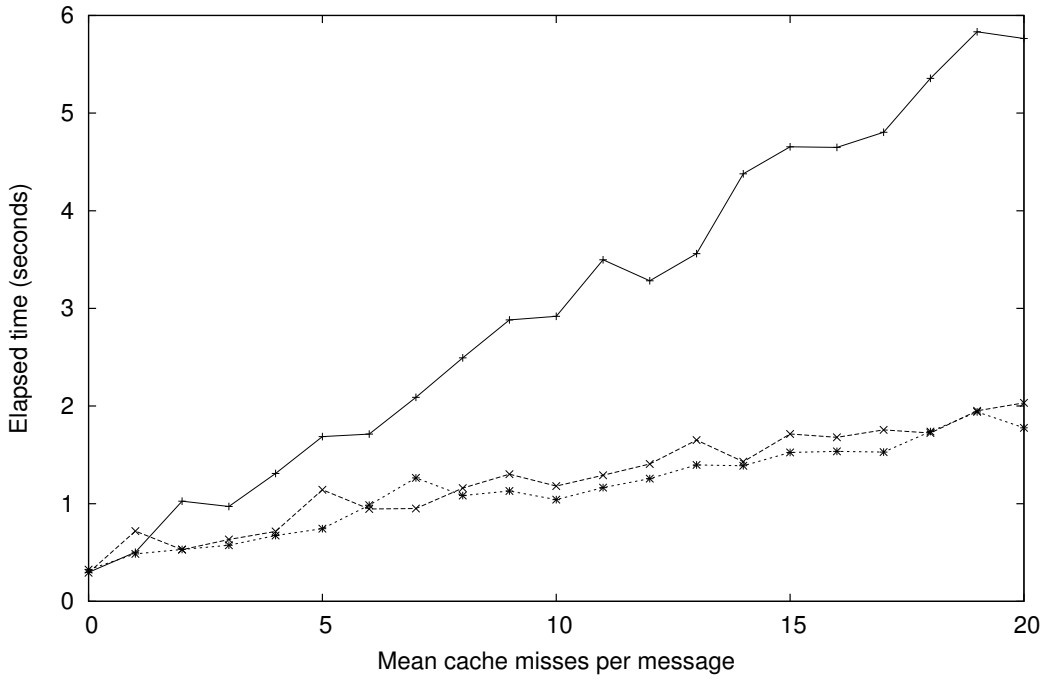


Figure 6.3: **The maximum throughput of a blocking Producer Consumer Queue from the Boost library, implemented by the std::deque container.** The elapsed time taken to transmit one million messages between two processors is plotted against a varying number of forced cache misses incurred while: producing (*), consuming (x) and both producing and consuming the messages (+). Figures given are the mean of 10 observations.

Ease of implementation

The Canonical Binary Tree on which our Producer Consumer Queue is based is a general solution to problems in concurrency, whereas a non-blocking algorithm is typically a specialised solution to a particular problem.

Section 6.3.8 compares the ease of implementing our queue with that of other queues.

We conclude that our Producer Consumer Queue implementation is more flexible than either its blocking or non-blocking counterparts.

Ease of use

This thesis claims that Transactional Data Structures make concurrent programs easier to write.

From the prospective of an application programmer our Producer Consumer Queue is as easy to use as either a blocking queue or the non-blocking Producer Consumer Queue developed by Scherer [SLS09].

Section 6.3.9 compares the ease of use of our queue with that of other queues.

We conclude that our Producer Consumer Queue implementation is as easy to use as either its blocking or non-blocking counterparts.

Scalability

This thesis claims that Transactional Data Structures facilitate the development of scalable concurrent programs.

We found that the throughput of both our queue and a queue implemented using mutual exclusion were unaffected by the number of processors concurrently accessing them.

Section 6.3.10 compares the scalability of our queue with that of other queues.

We conclude that our Producer Consumer Queue implementation is as scalable as either its blocking or non-blocking counterparts.

Progress guarantees

This thesis claims that concurrent programs that use Transactional Data Structures can guarantee progress.

A Producer Consumer Queue implemented by mutual exclusion makes no progress guarantees, whereas our non-blocking Producer Consumer Queue guarantees lock-free progress.

Section 6.3.11 compares the progress guarantees offered by our queue with those offered by other queues.

We conclude that our Producer Consumer Queue implementation is preferable to its blocking counterpart because it offers a progress guarantee.

6.3.3 Method

Experiments are performed using a PC with an Intel Core i7 860 processor operating at 2.8GHz with 8 MB of cache and 4GB of DDR3 SDRAM running at 1333 MHz. The examples are compiled using the Intel 64 bit C++ compiler with the maximum optimisation level.

We use the bounded blocking Producer Consumer Queue coding example from the Boost C++ library [Kar05]. Calls to the Boost Thread library are replaced by corresponding calls to the Threading Building Blocks library and the scalable memory allocator from this library is also used.

We use a bounded non-blocking Producer Consumer Queue based on a deque implemented by the Canonical Binary Tree. The Canonical Binary Tree is balanced but none of the optimisations, suggested in section **??** are implemented.

The memory occupied by the nodes is pre-allocated by the Threading Building Blocks scalable memory allocator. Nodes that become unreachable are periodically garbage collected without blocking the execution. The memory occupied by the leaves is allocated from a circular buffer. These locations are re-used but the Canonical Binary Tree ensures that they appear immutable. The queue is bounded by the leaf allocator which ensures that the front and back of the queue do not meet.

The queueing applications we compare behave differently because the access function of the blocking queue waits when the queue is empty, whereas the access function of the non-blocking queue may fail and must be re-tried. However, both applications transmit messages as fast as their queues allow.

6.3.4 Workload simulation

Inter-processor traffic is more difficult to characterise than network traffic. Standard protocols and benchmarks aid the evaluation of algorithms related to network traffic, whereas Inter-Processor Communication is generally based on bespoke protocols. Concurrent applications use a mixture of message sizes and perform varying amounts of work when preparing and processing messages.

The producer writes a message to memory and these writes are cached. The atomic compare-and-swap instruction in both the blocking and non-blocking implementations forces outstanding write operations buffered by the processor to be written to memory, so when the consumer reads the message from memory the operations are cache misses. The elapsed time taken to transmit messages is dominated by the latency of these cache misses.

We simulate the work done during the production and consumption of messages by inducing cache misses, but it is not sufficient to assume that a fixed number of cache misses is associated with each message. Messages are created by applications which do a varying amount of work during the production and consumption of messages and this behaviour must also be simulated.

We assume that the program issues memory operations that result in a cache miss at random intervals and that the latency of these operations dominates the production and consumption of the message. The number of cache misses per message is modelled by a Poisson distribution. A Poisson distribution is a discrete probability distribution that expresses the probability of a number of independent events occurring in a fixed period of time. A cache miss can be induced by accessing an array much larger than the processor cache.

6.3.5 Previous work

The C programming language does not specify a memory model so a concurrent application written in C relies on the memory model implemented by the underlying hardware architecture, but memory models implemented by hardware architectures differ. Adve provides a comprehensive tutorial on shared memory consistency models [AG95]. Non-blocking structures implemented in C tend not to be portable between different hardware architectures because the memory models implemented by these architectures are different. For example, The Intel

architecture software developer's manual describes how the memory models implemented by Intel IA-32 and Intel 64 bit processors differ [Int07]. It is difficult to construct a Non-blocking algorithm in C that is portable between the IA-32 and Intel 64 bit platforms.

Marginean describes a simple lock-free Producer Consumer Queue, implemented in C, in the mainstream magazine Dr Dobb's journal [Mar08]. This queue suffers from several problems including a misplaced memory barrier and false assumptions about the effect of atomic instructions on the iterators implemented by the Microsoft template library. The magazine published a revised version of the download code the following month but this too contained errors. Shutter described a working version of the queue, albeit with a restricted interface, four months after publication of the initial article [Shu08].

Non-blocking algorithms described in the literature may appear simple but getting them right is very difficult. Herlihy's book 'The art of Multiprocessor Programming' unintentionally illustrates the difficulty of finding errors in non-blocking algorithms. This book has an extensive on-line errata, even though it was clearly written and reviewed by experts [HS08]. Erroneous non-blocking algorithms, such as double-checked locking, have even appeared in peer reviewed conference publications [BBB+06].

The Java programming language has a clearly defined memory model. Manson describes the Java memory model in detail [MPA05]. The Java virtual machine for a particular hardware architecture implements memory barriers to ensure the correctness of functions in the Java libraries. Lea describes how portable concurrent programs can be constructed using the Java language [Lea06].

Scherer describes a lock-free unbounded Producer Consumer Queue which is called a scalable synchronous queue [SLS09]. This queue outperformed the queue included in the Java SE 5.0 version of the `java.util.concurrent` library and was subsequently included in Java 6. This queue does not contain messages in the way that our Messaging Queue does. Instead, it queues instances of the producers and matches them to available consumers to allow the handover of a single message. Scherer's thesis lists the program code which implements the queue and describes its operation in detail [Sch06]. The program code required to implement this queue is much shorter than that of our Canonical Binary Tree implementation but this belies its complexity.

6.3.6 Mailbox Queue performance

The throughput of a Mailbox Queue implemented by mutual exclusion is dependent on the standard library data structure that implements it. The std::list container is implemented by nodes with both forward and backward pointers, whereas the std::deque is implemented in managed blocks of storage. The size of an element in a std::list is larger than that of the std::deque. A single atomic compare-and-swap instruction is performed by each operation and the amount of memory written by the synchronisation depends upon the implementation of the data structure. The memory written by the synchronisation results in coherency cache misses when it is read by the consumer. The latency of cache misses dominates the execution time so the throughput of Mailbox Queue is dependent on the size of the elements of the underlying data structure.

Each message is written to memory by the producer and then read by the consumer. We estimate that this operation takes 800 cycles to complete so with a processor speed of 2.8GHz one million operations take about $(1000000*800/2.8*10^9) = 0.29$ seconds to complete.

The throughput of both our Mailbox Queue and the blocking queue is similar because they are both bounded by the latency of a similar number of coherency cache misses per message. To verify this we increased the size of the node in our Canonical Binary Tree and found that the throughput of the mailbox queue was reduced.

We did not expect the node size to make such a large difference to the performance of Transactional Data Structures. This observation motivated the search for ways of optimising the performance of the Canonical Binary Tree by reducing both the size of the node and the number of nodes accessed. These optimisations are described in section **??**.

6.3.7 Messaging Queue performance

The throughput of the Messaging Queue is, like the Mailbox Queue, bounded by the latency of cache misses. However, some misses are a result of the simulated processing of the messages.

When the production and consumption of messages is balanced the throughput of the blocking and non-blocking Messaging Queues are broadly similar. However, when the production and consumption of messages is imbalanced the

throughput of the blocking queue is higher than that of the non-blocking queue. When rate of production of messages is higher than the rate of consumption the instantaneous size of the queue is larger and consequently the path in the Canonical Binary Tree is longer.

The number of coherency cache misses incurred by each message processed by the non-blocking queue is dependent on the length of the path in the Canonical Binary Tree, whereas the number of coherency cache misses incurred by the blocking queue is independent of the size of the queue. Consequently the throughput of the non-blocking queue is dependent on the balance between the producer and the consumer, but the throughput of the blocking queue is not.

6.3.8 Ease of implementation

Both the non-blocking Producer Consumer Queue of Scherer and our Canonical Binary Tree took a similar amount of time to develop, so our non-blocking Producer Consumer Queue is no easier to implement from scratch than a comparable non-blocking queue [SLS09]. However, it is difficult to modify the ADT presented by the queue of Scherer without affecting its progress guarantee, whereas our queue can easily be tailored to the requirements of a particular application.

For example, a work stealing scheduler may be used to load-balance work among multiple consumers. A work stealing scheduler associates a unique Producer Consumer Queue with each consumer and it permits an idle consumer to remove messages from the back of a queue associated with a busy consumer to balance the work between consumers. Our Producer Consumer Queue can easily be adapted to permit equal access to both ends. It is more difficult to adapt the queue of Scherer to permit equal access to both ends.

6.3.9 Ease of programming

From the prospective of an application programmer our Producer Consumer Queue is as easy to use as either a blocking queue or the non-blocking Producer Consumer Queue of Scherer. However, Scherer's queue is more portable than our queue because it relies on the clearly defined Java memory model, whereas our queue is implemented in C which relies on the model implemented by the hardware architecture.

Our Producer Consumer Queue is more portable than other non-blocking

queues implemented in C because it relies on a single atomic compare-and-swap instruction for synchronisation, whereas other non-blocking queues rely on separate memory barriers which are architecture dependent [Shu08].

For example, a windowing queue allows more than one message to be added or removed by a single operation. Our Producer Consumer Queue can easily be adapted to support windowing by applying concurrency control to a path created by more than one access function. However, windowing is difficult to implement using mutual exclusion and we were unable to find an open source implementation of a windowing queue to compare our implementation against.

Ease of programming is a vague concept but we found our Producer Consumer Queue to be both portable and adaptable. It is at least as easy to use as either its blocking or non-blocking counterparts.

6.3.10 Scalability

Non-blocking algorithms are preferable to blocking algorithms because they are potentially scalable, whereas the scalability of algorithms that use mutual exclusion is fundamentally limited by Amdahl's law. Even a non-blocking algorithm that performs poorly on a modern Chip Multi-Processor is preferable to its blocking counterpart because the non-blocking algorithm is potentially scalable, whereas a blocking algorithm has limited scalability on any future hardware.

Goetz examines the scalability of the Producer Consumer Queues in the Java library [GBB+06]. Goetz found that the throughput of the queue is unaffected by the number of producers and consumers using it. We also found that the number of processors accessing a queue did not make any difference to its throughput.

6.3.11 Progress

Non-blocking algorithms are preferable to blocking algorithms because they offer a progress guarantee, whereas blocking algorithms do not. Even a non-blocking algorithm that performs less well than its blocking counterpart is preferable because the non-blocking algorithm guarantees progress, whereas its blocking counterpart has the potential to block indefinitely.

A lock-free queue may suffer from the progress pathology of live-lock. This occurs when two processors repeatedly prevent each other from successfully accessing the queue. In practice, our queue is unlikely to suffer from this pathology

because the Time Stamp Ordering concurrency control protocol ensures that one or other of the conflicting access functions takes precedence.

In practice, a Producer Consumer Queue is so simple and Chip Multi-Processors are so reliable that the lack of a progress guarantee makes little difference once the concurrent application is tested and shown to be working. However, programmers do not always get things right first time. During the development of a concurrent application a strong progress guarantee often makes the difference between an application that does not work correctly and one that requires a system restart to resolve dead-lock.

Bibliography

[AG95] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1995.

[BBB+06] David Bacon, Joshua Bloch, Jeff Bogda, Cliff Click, Paul Haahr, Doug Lea, Tom May, Jan-Willem Maessen, Jeremy Manson, John D. Mitchell, Kelvin Nilsen, Bill Pugh, and Emin Gun Sirer. The "double-checked locking is broken" declaration. http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html, 2006.

[GBB+06] Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, David Holmes, and Tim Peierls. *Java Concurrency in Practice*. Addison-Wesley Longman, Amsterdam, 2006.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[Int07] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, August 2007.

[Kar05] Björn Karlsson. *Beyond the C++ Standard Library, an introduction to Boost*. Addison-Wesley Professional, 2005.

[Lea06] Douglas Lea. *Concurrent Programming in Java(TM): Design Principles and Patterns (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2006.

[Mar08] Petru Marginean. Lock-free Queues. *Dr. Dobb's Journal*, July 2008.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. *SIGPLAN Not.*, 40:378–391, January 2005.

[Sch06] William N. Scherer, III. *Synchronization and concurrency in user-level software systems*. PhD thesis, Department of Computer Science, Rochester, NY, USA, 2006. AAI3204565.

[Shu08] Herb Shutter. Writing Lock-Free Code: A corrected queue. *Dr. Dobb's Journal*, October 2008.

[SLS09] William N. Scherer, Doug Lea, and Michael L. Scott. Scalable synchronous queues. *Commun. ACM*, 52:100–111, May 2009.