Nonblocking Concurrent Queue for Weak Memory Model

ALEXIS LE GLAUNEC, Rice University, USA

FIFO Concurrent Queues are used in many parallel applications to share data between threads. With hardwares exhibiting more and more parallelism, it has become essential to provide performant and scalable data structures in the context of multithreading. A class of concurrent queues is called nonblocking queues, based on the nonblocking property which guarantees that at least one thread is making progress. A famous nonblocking queue is the Michael & Scott (MS) queue. The advantages of the MS queue is it simplicity and efficiency. Many machines now run on architectures with a weak memory model where reordering of instructions can happen. The original Michael & Scott queue assumed a strong memory ordering such as sequential consistency, and therefore is not adapted to those weak architectures.

In this paper, we present an adaptation of the Michal & Scott queue for a weak memory model. In contrast with the original Michael & Scott queue, we allow for some instruction reordering for as long as it does not break the program's correctness. We have implemented the weak and strong MS queues in C++11 using its low-level atomic primitives. We have model checked our weak MS queue as well as the original MS queue using CDSChecker, and found no bug. We evaluate our Weak Michael & Scott queue against its strong counterpart over 3 architectures. On a x86 architecture, the weak queue allows for more compiler optimizations with an average speedup of 8% over the strong queue. On the POWER9 architecture, the benefit is visible in the case of multiprogramming with a 10% speedup over the strong queue.

CCS Concepts: • **Software and its engineering**; • **Computing methodologies** → **Parallel algorithms**;

ACM Reference Format:
Alexis Le Glaunec. 2024. Nonblocking Concurrent Queue for Weak Memory Model. 1, 1 (March 2024), 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Concurrent FIFO queues are widely employed in parallel programs. There are two sorts of concurrent queues: blocking and nonblocking queues. Blocking queues allow for unbounded delays due to slow threads preventing fast threads from making any progress. This is the case, for instance, when a slow thread holds a lock and a fast thread has to wait. Moreover, this issue also arises for blocking algorithms that are not based on locks [Gottlieb et al. 1983; Mellor-Crummey 1987]. While those algorithms perform well for a low contention settings, it has been shown by [Herlihy et al. 2003; Michael and Scott 1996] that those algorithms tend to perform poorly in a multiprogramming settings where multiple threads share one core. This is because once a thread's quantum runs out while holding a lock, the scheduler chooses another thread which cannot make progress while the lock-holder thread is asleep. On the opposite, nonblocking algorithms do not suffer from this scheduling problem and generally exhibit greater performance in a high contention setting. Prior work [Michael and Scott 1996; Prakash et al. 1991, 1994] on nonblocking queues have focused its effort on implementing queues assuming a strong memory model. Those algorithms are adapted for running on hardware with strong memory model (sequential consistency or TSO) such as x86 or

Author's address: Alexis Le Glaunec, Department of Computer Science, Rice University, USA, alexis.leglaunec@rice.edu.

XXXX-XXXX/2024/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

SPARC TSO. Nevertheless, they are not optimized for the weak memory model that is ubiquitous in smartphones (ARM) or gaming consoles (PowerPC) multicore processor architectures. In this paper, we propose a new version of the Michael and Scott nonblocking queue [Michael and Scott 1996] especially designed for a weak memory model. We compare the performance of our algorithm against the sequentially consistent version of the algorithm on the new Apple's silicon chip (ARM). The new algorithm is implemented in C++ using the low-level atomics from C++11. We have tested the implementation of the original Michael and Scott nonblocking queue from [Michael and Scott 1996] for a strong memory model as well as our new algorithm for weak memory model using the CDSChecker [Norris and Demsky 2013] model checker. We have found no bugs for the two algorithms.

In summary, the main contributions of our paper are the following:

(1) An implementation of the Michael & Scott nonblocking queue for a Weak Memory Model (ARM, POWER9, etc).
(2) The Model Checking of both the original Michael & Scott queue for a Strong Memory Model, and our implementation for a Weak Memory Model.
(3) A comparison of the performance benefit from a weak implementation over CPU architectures with strong and weak memory models: x86 (Intel i9-12900K), aarch64 (Apple M1 Pro) and IBM POWER9. Thus, we can measure both the gain due to compiler reordering and processor reordering (limited to weak architectures).

Outline of Paper: In Section 2, we explain the memory models and present the original and new algorithms for the nonblocking concurrent queue. In Section 3, we expose the methodology we used to verify the correctness of both the original and our new nonblocking queue. In Section 4, we compare the weak and strong versions of the concurrent queue. Finally, we conclude in Section 6 with a brief summary of our contributions.

2 ALGORITHMS

2.1 Memory Models

A memory consistency model defines the semantics of concurrent operations, in what order program instructions can be executed and the memory values that can be observed in a multithreaded execution. Instruction reordering can occur at the level of the compiler to enable optimizations, or at the level of the processor. There are 4 types of memory reordering:

- Load → Load: reordering of a load followed by a load,
- Load → Store: reordering of a load followed by a store,
- Store → Store: reordering of a store followed by a store, and
- Store → Load: reordering of a store followed by a load.

Memory models can be defined with respect to the memory reordering that are allowed. Memory models are generally classsided in two categories: strong memory models, where most of the memory reordering are disallowed, and weak memory models where most of them are allowed. In this paper, we consider the three memory models that are defined in the C++11 atomic library, e.g. Sequential Consistency, Release-Acquire and Relaxed memory models.

Sequential Consistency. In the Sequential Consistency Memory Model, none of the four memory reordering presented above are allowed. It is a widely used strong memory model, as it follows the order of instructions of the program and imposes a sequential order on instructions between threads. In the C++ concurrency memory model defined in [Boehm and Adve 2008], sequential consistency is the default memory model for atomic operations. This is the model that we will use to design our strong version of the Michael and Scott nonblocking queue. Note that while this

model prevents reordering of atomic operations, often referred to as synchronizations, non-atomic operations can still be reordered for as long as it does not change the intra-thread semantics.

However, the sequential consistency model enforces strong restrictions on reordering, which can incur a high cost to the hardware. This is why weaker memory models such as Release-Acquire and Relaxed memory models have been designed. They allow for more instruction reordering and are inspired by the memory fences that can be found in weak memory hardwares.

Release-Acquire. Release-Acquire defines a memory model with two fence instructions: Release and Acquire. In the Acquire semantics, all the instructions after a read instruction followed by acquire fence cannot be reordered before it. In the analogy with the lock algorithm, it means that you should never reorder an instruction in the critical section before the lock is acquired. Nevertheless, you can always add instructions into the critical section. In the Release semantics, all the instructions before a write instruction followed by release fence cannot be reordered after it. For the lock algorithm, it means that no instruction inside the critical section should be executed after the lock is released. This memory model is weaker compared to sequential consistency. On one hand, the Acquire semantics prevent Load $\longrightarrow$ Load and Load $\longrightarrow$ Store reordering. On the other hand, the Release semantics prevent Load $\longrightarrow$ Store and Store $\longrightarrow$ Store reordering. Therefore, this model limitation is that it cannot disallow Store $\longrightarrow$ Load reordering. We will use this memory model in our implementation of the weak version of the Michael and Scoot nonblocking queue.

Relaxed. In the Relaxed memory model, all of the four memory reordering are allowed. In our weak version of the Michael and Scoot queue, this will be useful for some operations that are optional, and therefore can be reordered.

2.2 Strong Nonblocking Queue

In this section, we present our implementation of the Michael and Scott nonblocking queue for the sequential consistency memory model. This is a straightforward adaptation of the original algorithm presented in [Michael and Scott 1996] adapted to the C++11 atomic library and templated. The idea behind the Michael and Scoot queue is to allow for the tail to lag behind behind by one node, i.e. to point to the second to last node instead of the last node. Therefore, when a thread succeeds enqueuing and before it updates the tail, another thread can update the tail and therefore prevent any blocking. This is done thanks to the Compare-And-Swap Read-Modify-Write atomic operation.

Strong Enqueue. Algorithm 1 presents the pseudocode for the enqueueing of a new node. When enqueuing, the tail may be lagging behind by at most one position. If so, to make the algorithm nonblocking, either the thread that has successfully enqueued (line 14) or the thread that failed (line 13) update the tail. The main purpose of line 6 is to prevent an unsuccessful compare-and-swap at line 8 if the tail is already lagging behind.

Strong dequeue. Algorithm 2 presents the pseudocode for the dequeuing of the first node. There are two cases to consider: when the queue is empty and when it is not empty. If the pointers of tail and head are equal (line 7), either the queue is empty and we return false or the tail is lagging behind (line 10) and we need to update the tail. If the queue is not empty, we try to move the head to the next node. Note that we read the value before performing the Compare-And-Swap as the value may be freed by another concurrent dequeue.

Compared to the original algorithm, we have decided not to include the Treiber's nonblocking stack algorithm to allocate reuse previously freed memory as it would complexify the weak version of the algorithm.

Algorithm 1: Enqueue on Strong Memory Model

```
1  Fn enqueue(val : T):
       // Create a new node
2      node = new Node();
3      node.value = val;
4      node.next = nullptr;
5      while true do
           // Retrieve the current tail and last inserted node
6          cur_tail = this→tail.load();
7          next = cur_tail.ptr→next.load();
           // If the tail has changed, try again (like Test-And-Test-And-Set)
8          if cur_tail == this→tail.load() then
               // Is the tail lagging behind?
9              if next.ptr == nullptr then
                   // Try to insert the node
10                 if cur_tail.ptr→next.CAS(next, (node, next.count + 1)) then
                       // Successfully added the node
11                     break;
12             else
                   // Try to update the tail
13                 this→tail.CAS(cur_tail, (next.ptr, cur_tail.count + 1));
       // Try to update the tail
14     this→tail.CAS(cur_tail, (node, cur_tail.count + 1));
```

2.3 Weak Nonblocking Queue

We have modified the Strong enqueue and dequeue algorithms for a Weak Memory Model. Instead of the default sequential consistency implied when using *load*, *store* and *CAS* operations, we indicate one of the C++ standard memory models: `acquire` for the acquire semantics, `release` for the release semantics and finally `release` when there is no reordering constraint.

Weak Enqueue. Algorithm 3 presents the dequeuing function specialized for a Weak Memory Model. Compared to Algorithm 1, we do not assume a sequential consistency memory order by default on the atomic operations. Instead, for each atomic operation, we label the operation with a memory ordering. In practice, we have modified Algorithm 1 by removing memory fences where it is unnecessary. For atomic `load`, valid memory orderings are sequential consistency, acquire and relaxed. For atomic `store`, those are sequential consistency, release and relaxed. Compare-And-Swap can either succeed or fail, therefore we need to indicate the memory model a) of the RMW (Read-Modify-Write) if the CAS succeeds or b) of the load if it fails. Note that the memory model in case of failure should always be more relaxed than in case of success. Moreover, on some hardwares, Compare-And-Swap can spuriously fail. This is a called a weak CAS. Strong CAS is build on top of weak CAS for those hardwares with additional synchronization costs. Therefore, we should use weak CAS whenever it is possible. To begin with, at lines 13 and 14, failing spuriously is fine because we do not check the return value of CAS. Thus, we use `weak_CAS` instead of `strong_CAS` and for the same reason the semantics both in case of success or failure can be relaxed. For the other CAS at line 10, we cannot make it weak because it could cause enqueing twice the same

Algorithm 2: Dequeue on Strong Memory Model

```
1  Fn bool dequeue(val : &T):
2      while true do
3          cur_head = this→head.load();
4          cur_tail = this→tail.load();
5          next = cur_head.ptr→next.load();
6          if cur_head == this→head.load() then
               // Check if the queue is empty or tail lags behind
7              if cur_head.ptr == cur_tail.ptr then
                   // Is the queue really empty?
8                  if next.ptr == nullptr then
9                      return false ;
                   // Tail is lagging behind
10                 this→tail.CAS(cur_tail, (next.ptr, cur_tail.count + 1)) ;
11             else
                   // Queue is not empty, try to dequeue
12                 *value = next.ptr→value ;
13                 if this→head.CAS(cur_head, (next.ptr, cur_head.count + 1)) then
                       // Dequeued successfully
14                     break ;
       // Free the node
15     delete cur_head.ptr;
16     return true ;
```

item. In case of success, we do not want reordering of this Compare-And-Swap with previous loads, so we use the release semantics. However, if it fails, the loading can be done later so we use the relaxed semantics. By default, we use the release-acquire semantics for all other atomic accesses to prevent aggressive reordering and of successive Compare-And-Swap. We need for the result of the CAS at line 10 to be visible by other threads, or we would break the invariant that the tails lags by at most 1. The only load that can be relaxed is in the comparison at line 8 because it is not essential to the program's correctness and its purpose is to prevent executing a Compare-And-Swap if you already know it will fail.

Weak Dequeue. Algorithm 4 presents an adaptation of the Michael & Scott queue for a Weak Memory Model. Essentially, we followed the same guideline here as for the weak enqueuing. The load at line 4 can be made relaxed because even if it is reorder after the comparison at line 8, the dequeue will fail in the else statement later and we will go back to that line in a next iteration of the loop.

2.4 Memory reclamation

In nonblocking programming, memory reclamation is a serious problem that can materialize with the ABA problem when using a Compare-And-Swap operation. In Compare-And-Swap, we compare an object `obj` to a `desired` object, and replace it with `new_obj` if `obj` = `desired`. Suppose that, in the enqueue of Algorithm 1, tail points to memory address A in thread 1. In the meantime, new nodes are enqueued and the node at memory address A is freed. Its memory address is then reused for a new node. Compare-And-Swap considers the two nodes as identical because they shared the

Algorithm 3: Enqueue on Weak Memory Model

```
1  Fn enqueue(val : T):
       // Create a new node
2      node = new Node();
3      node.value = val;
4      node.next = nullptr;
5      while true do
           // Retrieve the current tail and last inserted node
6          cur_tail = this→tail.load(acquire);
7          next = cur_tail.ptr→next.load(acquire);
           // If the tail has changed, try again (like Test-And-Test-And-Set)
8          if cur_tail == this→tail.load(relaxed) then
               // Is the tail lagging behind?
9              if next.ptr == nullptr then
                   // Try to insert the node
10                 if cur_tail.ptr→next.strong_CAS(next, (node, next.count + 1), release, relaxed)
                       then
                       // Successfully added the node
11                     break;
12             else
                   // Try to update the tail
13                 this→tail.weak_CAS(cur_tail, (next.ptr, cur_tail.count + 1), relaxed, relaxed);
       // Try to update the tail
14     this→tail.weak_CAS(cur_tail, (node, cur_tail.count + 1), relaxed, relaxed);
```

same memory address. As a result, the old node is enqueued and breaks the queue consistency. To tackle the ABA problem, several solutions have been proposed: reference counter where a counter is appended to all pointers and increased at every atomic operation, and hazard pointer [Michael 2004] where nodes are feed only after making sure that all the other threads do not keep a local version of the node. In our implementation, we use reference counters as it is simple and efficient. The main disadvantages of reference counters over hazard pointers is that Compare-And-Swap is performed over a double word (64-bit pointer and 64-bit counter) which can be more costly on some architecture, and the ABA problem may still occur if the counter overflows. The first issue can be solved by encoding the counter within the pointer. We ultimately decided to keep to the reference counter because it makes designing the weak algorithm easier, and most modern architectures natively support a double word Compare-And-Swap.

3 CORRECTNESS

Various methods have been explored the test the consistency of concurrent data structures, the main ones being Model Checking and Proof Assistant. In this section, we present the testing methods we used to check the correctness of our algorithms.

3.1 Unit Testing

We have tested our strong and weak implementations of the Michael & Scott queue both sequentially and in parallel. Our tests are parametric (number of threads, number of enqueues and dequeues)

Algorithm 4: Dequeue on Weak Memory Model

```
1  Fn bool dequeue(val : &T):
2      while true do
3          cur_head = this→head.load(acquire);
4          cur_tail = this→tail.load(relaxed);
5          next = cur_head.ptr→next.load(acquire);
6          if cur_head == this→head.load(relaxed) then
               // Check if the queue is empty or tail lags behind
7              if cur_head.ptr == cur_tail.ptr then
                   // Is the queue really empty?
8                  if next.ptr == nullptr then
9                      return false ;
                   // Tail is lagging behind
10                 this→tail.weak_CAS(cur_tail, (next.ptr, cur_tail.count + 1), relaxed, relaxed) ;
11             else
                   // Queue is not empty, try to dequeue
12                 *value = next.ptr→value ;
13                 if this→head.strong_CAS(cur_head, (next.ptr, cur_head.count + 1), release,
                     relaxed) then
                       // Dequeued successfully
14                     break ;
       // Free the node
15     delete cur_head.ptr;
16     return true ;
```

to test many possible settings. One test consists in K threads performing 1 million enqueue and blocking dequeues concurrently. We also checked that all the values enqueued are found after dequeuing all the elements. However, this testing is not sufficient to find elaborated bugs.

3.2 Model Checking

For a given program, Model Checking consists in exploring all the possible interleaving looking for invalid executions, i.e. that do not have the behavior expected by the programmer. To model check our algorithms, we use CDSChecker [Norris and Demsky 2013], a tool that allows model checking programs written in C++11 with the atomics library. CDSChecker has some limitations: limited support for modern C++ atomics and it only works for relatively simple programs.

To model check our algorithm using CDSChecker, we performed two simplifications:

(1) No memory reclamation, and
(2) No templating.

The first simplification is necessary because CDSChecker does not support double word Compare-And-Swap, and therefore would not compile our implementation. It also seems that CDSChecker does not like functions with templates. Thus, we removed those two things from the implementation that we tested CDSChecker. Note that for the benchmarking we do not make those simplifications.

As a result, we were able to verify our strong and weak queues implementations over a simple program with 2 threads: each thread does one enqueue and then one dequeue. We tried several other examples with more threads and enqueue/dequeue but the running time was exponentially

longer. CDSChecker verified our implementations and found no invalid execution over this example. While this does not prove that our algorithms are correct, this is useful testing to check that our algorithm performs as expected over simple executions.

4 EXPERIMENTS

4.1 Experimental Settings

We have run the experiments of three machines:

(1) Machine A equipped with an Intel(R) Core(TM) i9-12900K CPU (8 performance + 8 efficiency cores) processor and 32GB of RAM on Ubuntu 22.04 with GCC 11.4,
(2) Machine B equipped with a M1 Pro processor (10 cores) and 16GB of RAM on MacOS Monterrey with Clang, and
(3) Machine C equipped with a POWER9 processor (160 cores) and 260GB of RAM on Red Hat Entreprise 8.8 with GCC 8.

For each experiment, we run 5 trials and report the mean value.

4.2 Performance Comparison

Benchmark. To compare the weak and strong queues, we run the following experiment: k threads execute 10M enqueues followed by 10M dequeues. Between each enqueue or dequeue call, a thread performs "some work" by spinning for about 200 nanoseconds to prevent one thread from executing many queue operations successively and benefit from high cache hit rate. Each thread execute $10^7/k$ enqueues and dequeues where k is the number of threads. Therefore, as k becomes larger, we measure more and more the critical section.

Compiler optimizations. While x86 multiprocessors have a strong memory model (TSO), it does not mean that no instruction reordering can occur at the compiler level. As a matter of fact, every non-relaxed atomic access acts as a compiler barrier as well in C++. Therefore, by relaxing the memory model, we also enable more compiler optimizations. Fig. 1 presents the results of the experiment over the x86 architecture. We notice a clear benefit coming from our weak version of the Michael & Scott queue. Up to 5 threads, we observe an increase in performance as we add more parallelism. Then, it stabilizes after 8 cores, i.e. the number of performance cores. After 16 cores, we hyperthreads that share the same cores, therefore the performance start degrading again. Overall, the weak version of the queue consistently outperforms the strong queue even for a strong memory model, with an **average speedup of 8%**. This shows that a fine-grain implementation of an algorithm using low-level atomics is also beneficial to harwares with strong memory ordering thanks to more compiler optimizations.

Processor reordering. On a Weak Memory Model, we expect to see both the benefits from compiler reordering as well as processor reordering by lifting unnecessary memory fences. Fig. 2 presents the results of the experiment over the POWER9 architecture. Compared to x86, it is surprisingly worse with an average speedup of only 2%. However, as the number of threads becomes higher, the speedup increases to about 5%.

Multiprogramming. When it comes tu multiprogramming, POWER9 cores are able to execute up to 8 threads on a single concurrently. We now compare the performance of the strong and weak queues in a multiprogramming settings. Fig. 3 presents the results of the experiment for up to 320 threads (2 threads per core) over the POWER9 architecture. When we reach the multiprogramming area (after 160 threads), we observe a large speedup of the weak queue compared to the strong. Between 160 and 320 threads, the average speedup is of about **10%**. It is likely that when multiple

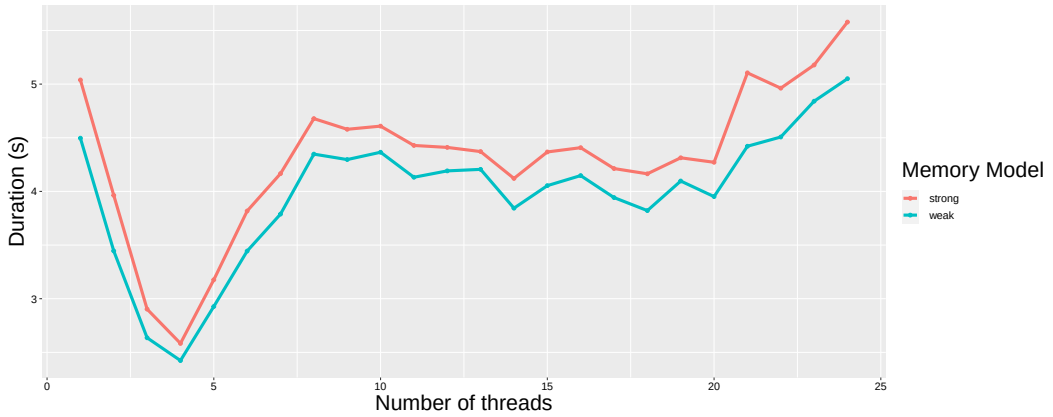

Fig. 1. Performance results of 1M enqueues and dequeues over an x86 multiprocessor

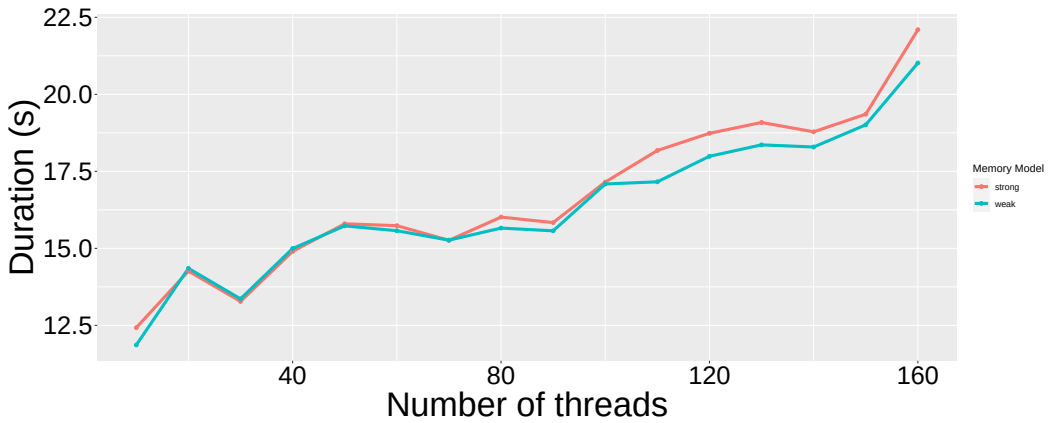

Fig. 2. Performance results of 1M enqueues and dequeues over the IBM POWER9 multiprocessor

threads are executed in parallel on one core of the POWER9 multiprocessor, the weak memory order allows the processor to interleave executions of threads, which is not allowed when using the sequential consistency model. Fig. 4 presents the results of the experiment for up to 20 threads (2 threads per core) over the M1 Pro multiprocessor (aarch64). The M1 Pro's ARM architecture is less relaxed than the POWER9, so we expect it to benefit less from a weaker memory ordering. In fact, the average speedup is of about 8.5%, and is largely due to the peak for 10 threads. This peak may be due to some internals of the M1 Pro multiprocessor at the edge of multiprogramming.

5 RELATED WORK

Software Memory Ordering. There is a rich set of prior works exploring the definition of Memory Ordering for programming languages. [Manson et al. 2005] presents the Memory Model of the Java programming model that is based on sequential consistency using the synchronized primitive. It also defines the semantics of programs with data races in that can lead to undefined behavior. Later on, the C++ Concurrency Memory Model was defined in [Boehm and Adve 2008] that is based on

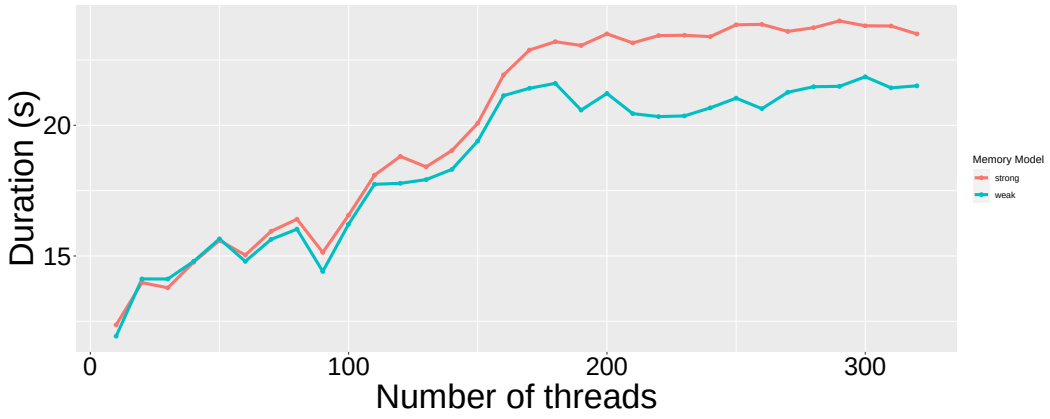

Fig. 3. Performance results of 1M enqueues and dequeues with up to 320 threads over the IBM POWER9 multiprocessor

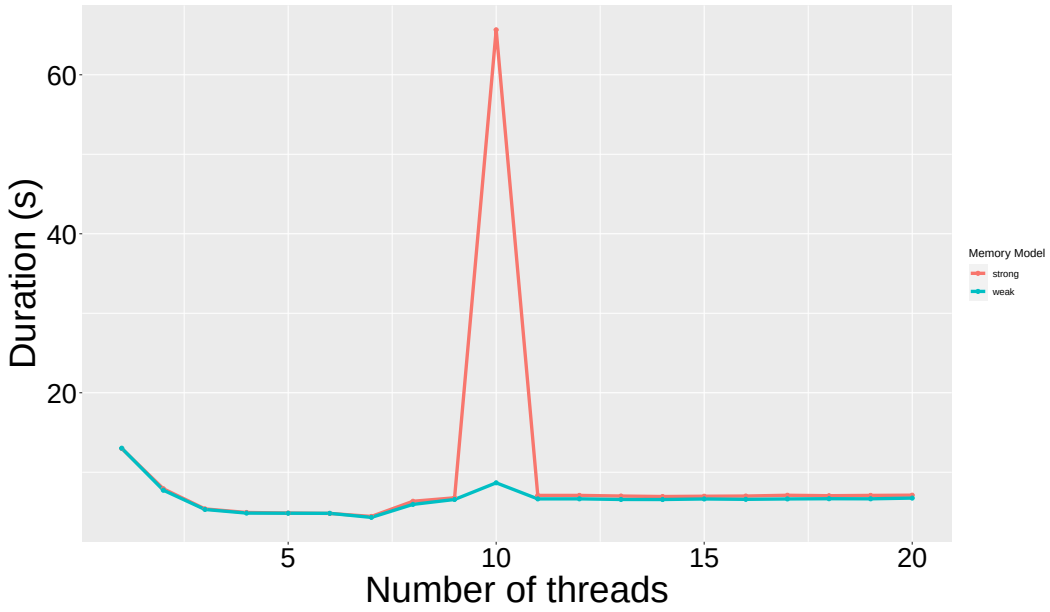

Fig. 4. Performance results of 1M enqueues and dequeues with up to 320 threads over the M1 Pro

the happens-before order of memory actions similar to the release-acquire semantics. It allows for low-level atomis with either the release, acquire, sequential consistency or relaxed semantics.

Hardware Memory Ordering. Memory ordering has also be defined for multiple hardware architectures. [Adve and Boehm 2010] gives an overview of the memory models used both in hardware and software, and raises the question of the formal definition of a memory model. The authors point out that there is a disconnection between the hardware and software memory models, and advocate for codesign between hardware designers and programmers. In fact, most programming languages assume a sequential consistency model that is not provided in real hardware. The TSO

memory ordering implemented on x86 multiprocessors was first defined in [Sindhu et al. 1992]. However, it has been shown that the TSO, as defined by Intel and AMD, differs from the TSO formal definition and several works have tried to defined the exact memory order used by x86 processors [Owens et al. 2009; Sarkar et al. 2009; Sewell et al. 2010].

6 CONCLUSION

We have consider the problem of adapting a nonblocking algorithm for a weak memory model. The original Michael & Scott nonblocking queue only proposed an algorithm assuming a strong memory model such as TSO. We have expanded the Michael & Scott Queue to accelerate the execution on hardwares with more relaxed memory models. We have verified both the strong and weak versions of the Michael & Scott queue using Model Checking. Our experimental results show that the weak version of the queue outperforms the strong version for both weak and strong hardware architectures. For the x86 architecture, we have an average speedup of about 8%. For the POWER9 architecture, the best performance is obtained for multithreading with a 10% speedup.

REFERENCES

Sarita V. Adve and Hans-J. Boehm. 2010. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *Commun. ACM* 53, 8 (aug 2010), 90–101. https://doi.org/10.1145/1787234.1787255

Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. *SIGPLAN Not.* 43, 6 (jun 2008), 68–78. https://doi.org/10.1145/1379022.1375591

Allan Gottlieb, Boris D. Lubachevsky, and Larry Rudolph. 1983. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *ACM Trans. Program. Lang. Syst.* 5, 2 (apr 1983), 164–189. https://doi.org/10.1145/69624.357206

Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. 2003. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing* (Boston, Massachusetts) *(PODC '03)*. Association for Computing Machinery, New York, NY, USA, 92–101. https://doi.org/10.1145/872035.872048

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 378–391. https://doi.org/10.1145/1040305.1040336

John Mellor-Crummey. 1987. Concurrent queues: Practical fetch-and-φ algorithms. (1987).

M.M. Michael. 2004. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (2004), 491–504. https://doi.org/10.1109/TPDS.2004.8

Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (Philadelphia, Pennsylvania, USA) *(PODC '96)*. Association for Computing Machinery, New York, NY, USA, 267–275. https://doi.org/10.1145/248052.248106

Brian Norris and Brian Demsky. 2013. CDSchecker: Checking Concurrent Data Structures Written with C/C++ Atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 131–150. https://doi.org/10.1145/2509136.2509514

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better X86 Memory Model: X86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics* (Munich, Germany) *(TPHOLs '09)*. Springer-Verlag, Berlin, Heidelberg, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

Sundeep Prakash, Yann Hang Lee, and Theodore Johnson. 1991. *Non-blocking algorithms for concurrent data structures.* Master's thesis. Citeseer.

S. Prakash, Yann Hang Lee, and T. Johnson. 1994. A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap. *IEEE Trans. Comput.* 43, 5 (may 1994), 548–559. https://doi.org/10.1109/12.280802

Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. 2009. The Semantics of X86-CC Multiprocessor Machine Code. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 379–391. https://doi.org/10.1145/1480881.1480929

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer's Model for X86 Multiprocessors. *Commun. ACM* 53, 7 (jul 2010), 89–97. https://doi.org/10.

1145/1785414.1785443
Pradeep S Sindhu, Jean-Marc Frailong, and Michel Cekleov. 1992. Formal specification of memory models. In *Scalable Shared Memory Multiprocessors*. Springer, 25–41.