Comparison of Verification Methods for Weak Memory Models

Niels ten Dijke
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
n.t.tendijke@student.utwente.nl

ABSTRACT

Modern multi-core systems allow for reordering of memory instructions for performance reasons. This reordering of instructions can be a source of bugs, especially in algorithms which do not use locks for synchronization. Verification of concurrent code on relaxed memory models is a computationally hard problem and many approaches to verify software under these models have been proposed, however no overview or comparison of current methods exists. In this project we give an overview of three recent verification methods for relaxed memory models. These methods are mainly chosen because, next to being cited, they have tools which can be used to verify concurrent code. Furthermore, these methods are recent and should give a good picture of the current state of the art. We compare these methods on verification of real world software, i.e. algorithms which have been proposed in research or are used in industry, under commonly used weak memory models. This is done by running the tools against a benchmark of concurrent C programs.

Keywords

Weak memory models, Program verification, Comparison, Overview

1. INTRODUCTION

Nowadays, shared-memory multiprocessors are pervasive [18, 11] and have led to concurrent programming to become mainstream. The main reason for using a multi-core architecture is performance: the increase in single core performance has stagnated since the mid-2000 and the use of multiple cores has been the way to boost chip speed. However, concurrent code running on multi-core architectures is more complex to understand and a source of bugs. Modern multi-core architectures such as x86, ARM or Power allow for reordering of memory instructions for improving performance [1]. In [21] Zucker et al investigate the performance benefits of reordering, also called *weak* or *relaxed* memory ordering and found a ten to forty percent increase in performance. For example, a processor may use a store buffer, to queue pending stores, while performing load instructions. Reordering of memory instructions especially is a problem in lockless algorithms, which are not data race free and must use explicit memory fence instructions to prevent certain reordering. Insufficient fences lead to concurrency bugs; on the other hand, too many fences impact performance. In order to reason about the correctness of programs running on these systems a formal description of memory with respect to read and write actions, or *memory consistency model*, is needed.

20th *Twente Student Conference on IT* January 24th, 2014, Enschede, The Netherlands.

The complex behavior these models exude has lead to several bugs in the past, even in production-level code. Model-checking techniques are powerful tools to automatically find these bugs [9]. More specifically, given a program, a memory model and a set of safety properties, a model checker outputs either that it has found no bugs or it gives an error trace that leads to incorrect execution. However, verifying programs running on weak memory models is generally very hard. Several methods exist to tackle the verification problem [3, 14, 13, 7, 6]. Although these often contain benchmarks, to our knowledge, no overview, let alone comparison of current methods exists. The primary aim of the project is to give an overview of current methods and to compare them on practicability in real world use.

More specifically, we describe the techniques used and run the tools corresponding to the methods on various concurrent algorithms used in research and industry. In particular we compare three contemporary methods (listed in section 3) on the verification of concurrent programs running on various weak memory models.

Ideally the verification method should:

1. Be correct in as many instances as possible, this means finding bugs in faulty code and not finding them in correct code.
2. Be able to deal with various memory relaxations.
3. Be efficient enough for practical use.

The last point is hard to evaluate, in this project we run a tool on an example for a maximum of 24 hours. The methods are compared on these above points by running a benchmark comprised of concurrent C programs of which some of these programs are known to have bugs under particular memory models. For each example the tools are run using the supported memory models and check if the output matches the expected output to compare the methods on the first and second point. Lastly, we time the tools for each example for the comparison of point 3. We found that the verification tools manage to find bugs in certain examples, however, the tools lack an easy to use user interface and fail on some examples.

This paper is organized as follows: section 2 briefly describes the abstract behavior of the relaxed memory models which are used in the example cases for the tools and shortly describes the verification problem. The next section provides some information on how the verification methods work. The later sections describe and discuss the examples and the results for the tools.

2. MEMORY CONSISTENCY MODELS

This section provides background on weak memory models which are supported by the methods under comparison. TSO, PSO and RMO are defined in the SPARC architecture manual [20]. The POWER model is described in [2], however [3] only implements part of the POWER specification. These models provide an abstract view of the possible behavior of multi-core architectures with respect to memory operations.

2.1 Sequential Consistency

The simplest and most intuitive for programmers is *sequential consistency* defined in [12] as follows: "A multiprocessor architecture where the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." What this means is that there is some total order on the instructions of all the processes and the per-process program order, i.e. the order in which instructions should be executed according to the program code, is preserved in the total order. In particular the following ordering is preserved:

- **Write-to-read order:** Stores are not reordered after loads.
- **Write-to-write order:** Stores are not reordered after stores.
- **Read-to-read/write:** Stores and loads are not reordered before loads.
- **Write Atomicity:** All write to a location should appear to all processors to have occurred in the same order. This means a processor does not read its own or other's writes early.

This simplified model is often assumed when verifying code [8, 5], however, in general does not properly characterize all the behavior of memory of modern multi-core architectures and may therefore not find all bugs. Though in most programs this assumption often is sufficient, since standard practice is to program shared memory using locks to prevent data races. However there are situations where correctness for SC is only a necessary condition for correctness, for example in lockless algorithms, one of which we will study in the benchmarks.

2.2 Total Store Order

The *Total Store Order*(TSO) model relaxes the write-to-read order. Possible behavior is illustrated by the program below, which is a simplified version of Dekker's mutual exclusion protocol.

Proc1	Proc2

x=1
lock1=y
if(lock1==0)
{critical section}
	y=1
	lock2=x
	if(lock2==0)
	{ critical section }

If the reads of the values of x and y happen before they are written (note that this does not violate the constraint for a single processor that operations on the same location occur in program order) both processes can enter the critical section at the same time. This can happen in the x86 architecture where stores are pushed onto a store buffer; a read to another location than a store that was put in the buffer previously can be executed before the store has been written to memory. Note that this does not occur if the multiprocessor system is sequentially consistent.

2.3 Partial Store Order

The next relaxation is *Partial Store Order* (PSO) model. In essence it is a weakening of the TSO model which allows for reordering of store operations, that is, it relaxes the write-to-write memory order. This can happen if the stores are written to different memory modules through a general interconnection network. A store to one memory module may complete faster than another module and thus reorder stores. The example below illustrates how this relaxation can lead to undesirable behavior.

Proc1	Proc2

val=42
lock=1
	while(lock==0)
	{ continue; }
	result=val

Here it is possible that the result will not equal 42, if the order of the write to *val* and *lock* by Process 1 is changed and can occur under the the PSO model. This is not possible under TSO since the write of 42 to val has to occur before the write of 1 to lock.

2.4 Relaxed Memory Order

The *Relaxed Memory Order*(RMO) is a further weakening of PSO and TSO and allows read-to-read/write relaxation; this means the reads are non-blocking, i.e. the processor does not stall when performing a read operation. In the previous example the second process can issue both reads at the same time, resulting again in the value of result to not be equal to 42. This is the case even if the write-to-write order is enforced by memory fences (see section 2.6). Another example to illustrate this behavior is given below:

Proc1	Proc2

val1=1
val2=1
	lock1 = val2
	lock2 = val1

Assuming val1 and val2 are both initialized at 0 and the writes are executed in program order, it is possible that after execution of this code under the RMO model that lock1 equals 1 and lock2 equals 0. The read of val2 to lock2 can be serviced by a non-blocking cache, with the old value (0), while other read to lock1 reads the new value.

2.5 POWER

The POWER models is the model used in the ARM and IBM POWER architectures and is a weakening of RMO. POWER relaxes write atomicity, as explained earlier this means writes do not reach all processors at the same time. It is both possible that a processor reads its own writes early and its possible for a processor to read others writes early, which means a processor can read a write before it is visible to the other processors. An example of possible behavior is given below.

Proc1	Proc2	Proc3
val1=1	lock1 = val1	lock2= val1

Table 1. Summary of weak memory consistency models

	W->R	W->W	R->RW	Write atomicity
SC				
TSO	X			
PSO	X	X		
RMO	X	X	X	
POWER	X	X	X	X

With val1 initially at 0. It is possible for lock1 differ from lock2, this can happen if the system uses a general interconnection network where the network does not guarantee when stores are delivered to different processors. Proc2 could read the new value whereas Proc3 reads the old value.
A summary of the models is given in table 1. An X indicates the model relaxes that particular ordering.

2.6 Memory Fences

Certain reordering of memory instructions can be unwanted. It is possible for the program to enforce certain ordering (and thus prevent incorrect execution) using *memory fences* or *barriers*. These instructions guarantee that certain operations(read and/or write) finish before the memory fence. There are several types of fences, summarized below.

Store barriers

A store barrier guarantees that all store instructions before the barrier are executed an thus visible to the other processors. It does, however, have no guaranteed effect on the loads.

Data dependency barriers

A data dependency barrier is used to maintain the ordering of loads that are interdependent. If the result of a load after a barrier depends on the result of a load before the barrier, the data dependency barrier guarantees the first load will be committed before the barrier and is accessible for the second load.

Load barriers

The load barriers guarantees all load instructions before the barrier are visible to the other processors. The partial ordering on the loads has no guaranteed effect on the stores.

General memory barrier

A general memory barrier is the load barrier and store barrier combined: it is a partial ordering on the loads and stores and guarantees all stores and loads before the barrier are committed.

2.7 Compare-and-swap

The compare-and-swap (CAS) instruction is an atomic operation which is often used in lockless concurrent programs and is used to atomically update the value of a memory location. An implementation is given in listing 1. The CAS instruction receives three parameters. The first parameter is the address which is to be updated, the second parameter is the value that is expected to be updated and the last parameter is the new value. Starting with line 3 it goes into an atomic section where the if the value in the memory location is updated if the current value in the memory location equals the old (expected) value, i.e. the memory location has not been updated by another thread using the compare-and-swap operation before entering the atomic section. The operation always returns the value read in the atomic section (line 4 in the program). Based on this value the caller of this function can conclude the CAS operation succeeded or that another thread modified the value. In general, a CAS instruction only guarantees atomic updates to one particular memory location. In some cases, however, for example in x86, a compare-and-swap implies a general memory barrier.

Listing 1. Compare-and-swap

```
1  word cas(word *mem, word old, word new)
2  {
3          begin_atomic();
4          int current = *mem;
5          if(current == old)
6          {
7                  *mem = new;
8          }
9          end_atomic();
10         return current;
11 }
```

2.8 Verification

Verification of programs running under weak memory models is a hard problem.[4] studies the complexity theoretical aspects of verifying finite state concurrent programs running under the TSO, PSO and RMO relaxed memory models. As mentioned before, a model checker, checks if particular properties are violated. In particular: given a (potentially infinite) system with transitions between states, it tries to find particular states (which could be violations of safety properties). This problem is referred to as the reachability problem and is decidable but non-primitive recursive ([19] for definition) for TSO and PSO. Relaxing the read->read/write order makes it undecidable, which is the case for RMO and weaker models.

3. RELATED WORK

Several papers have been written on the subject; ranging from better descriptions of weak memory models[16, 17] to techniques to do verification. The strategy seems often to be to prove a program running on a particular relaxed model is sequentially consistent[6, 7, 14]. Since we are interested in the practical state of research and comparing methods purely theoretically is hard, the methods we chose to compare have tools which can verify concurrent C code. To our knowledge no other tools to verify concurrent C code under relaxed memory models exist.

CBMC

CBMC incorporates the method described in *Software verification for weak memory via program transformation* [3]. The C program is first compiled into a GOTO-program, which is an control-flow graph. This program is then used as input for goto-instrument which generates an abstract event graph. This graph is comprised of read and write

events which are related to each other. It then finds cycles in the graph in which an execution which are valid on the weak architecture but not on SC. In these cycles it chooses one pair (for example a read and write pair) to delay. In the Dekker example a write-read pair can be delayed. These instructions are to be instrumented, in the example this means the write gets appended to a buffer which will flush nondeterministically.
The instrumented program can then be verified using SC tools. In the experiments we used SatAbs since it provided the best results according to the benchmarks performed for[3].
CBMC supports TSO, PSO, RMO and POWER. The method claims to be sound but not complete. The tool is open-source and available on
http://www.cprover.org/wmm/.

Fender

In *Dynamic Synthesis for Relaxed Memory Models* [14] the method behind the tool DFENCE is presented.
Weak behavior is simulated by using buffers; a per-thread buffer for TSO and a per-variable for PSO. A scheduler is used to find bad executions. At every scheduling point a thread is selected. The scheduler can then either flush the write buffer for TSO or flush the value of a particular variable for PSO with a certain probability. At every step it checks for safety violations (through assertions) and for sequential consistency and linearizability. Once an illegal execution is found, it can automatically be repaired through a process Vechev et al. call *Dynamic Synthesis.* First all the possible ways to avoid the executions are computed. These are then appended to all pending repairs. The repairs are then either enforced or accumulated. This process continues until no violating execution is found. When trying to find bad executions, the program is run several tiimes and is in essence a probabilistic method.
DFENCE supports TSO and PSO. The tool DFENCE is available on http://practicalsynthesis.org/fender/.

CheckFence

The method behind Checkfence is described in *CheckFence: checking consistency of concurrent data types on relaxed memory models* [6]. First it goes to a process called "specification mining": the program is encoded in such a way that it becomes a satisfiability problem whose solutions are possible serial executions. The observation of input and output values during execution is added to a set called the observation set. It then checks if the observations of executions under weak memory model are contained in the observation set. If one is found that not is included in the set, then this is a counterexample. CheckFence supports TSO and RMO. The tool is open-source and available on http://checkfence.sourceforge.net/.

4. EXPERIMENTS

In order to compare the methods on the three points described in introduction a benchmark of examples of concurrent C programs is used. [3, 6, 14] already have done some benchmarks. These and one other program is listed below, with their description, safety properties and expected result for SC, TSO, PSO, RMO and POWER (if known). The different methods are tested on whether they provide the right output. Next to that the time to verify the programs will be measured, although this is not a very precise method to measure efficiency, there is correlation between the two.

4.1 Examples

The following examples have been taken from the papers of the methods described in section 3 and one unpublished paper.

Dekker's mutual exclusion protocol

Dekker's algorithm is the first known correct solution of the mutual exclusion problem[10]. It fails however under TSO and more relaxed memory models, since allowing reads to be reordered before the writes can lead to the threads entering the critical section at the same time.

Microsoft producer consumer

In [7] Burckhardt et al. find a bug in a Microsoft production level concurrency library using the Sober tool. The algorithm fails under TSO.

Michael and Scott Queue

The Michael and Scott queue [15] is a non-blocking concurrent queue, which without memory fences fails under PSO and weaker.

Worker synchronization in PostgreSQL

In [3] a bug in the PostreSQL worker synchronization algorithm is found when relaxing write atomicity which is specific to the POWER memory model. Details on the bug can be found in the mentioned paper.

Lockless split deque

This example is taken from an unpublished paper by Tom van Dijk on a new concurrent work stealing deque algorithm. A work stealing deque is a concurrent datastructure where new tasks can be pushed on one end by the deque's owner and taken from the other end by the owner(pop) or other worker threads(steal). In essence there can be an infinite amount of worker threads, however we simplify without loss of generality(symmetry) by having one thread pushing and popping and 2 threads stealing (again by symmetry). The number of pushes is the same as the number of pops; the number of steals can be arbitrary. Mutual exclusion for stealing threads is provided trough a CAS instruction. Since we want each task to be executed, but not more than once the safety property are that each task should only be taken exactly once, this means each task is either taken by the owner or stolen by another thread. As this algorithm has not been published it has been added to the appendix.

4.2 Experimental setup

The results of the benchmark is presented per example with a small discussion on the output for the tools of that example in the Analysis section. The computer on which the tools ran was an AMD Dual Core processor running at 3 GHz with 2GB of RAM. We measured time by using bash's built in command *time*. The times displayed in the results section will be the elapsed real (wall clock) time in seconds. The examples are run five times to record mean and variance.

5. RESULTS

Below are the results for the tools for the examples described in the methods section. They are subdivided based on the example and then further on verification method. Each table under a particular example gives the results for one method for that example. Each row depicts the outcome for one specific model. The result "Successful" means there are no bugs found, while "Failed" indicates a bug. If the result of the example is known the expected columns shows what the outcome should be. Detailed explanation and analysis of the results can be found in the analysis section.

5.1 Dekker's algorithm

CBMC

Memory model	Expected	Result	Mean	SD
SC	Successful	Successful	0.2	0.12
TSO	Failed	Failed	1.1513	0.019
PSO	Failed	Failed	24.3	0.14
RMO	Failed	Failed	9.4	0.12
POWER	Failed	Failed	6.75	0.049

CheckFence

Memory model	Expected	Result	Mean	SD
TSO	Failed	-	-	-
RMO	Failed	-	-	-

CBMC works as expected and further inspection of the trace points to the problem in the example. CheckFence returns with a vacuous pass, which means it did not find an execution and something did not go right. The source of this problem is unknown.

5.2 Microsoft producer consumer

CBMC

Memory model	Expected	Result	Mean	SD
SC	Successful	Successful	0.6	0.12
TSO	Failed	Successful	3.04	0.019
PSO	Failed	Successful	3.01	0.022
RMO	Failed	Successful	3.02	0.032
POWER	Failed	Successful	3.03	0.018

CheckFence

Memory model	Expected	Result	Mean	SD
TSO	Failed	-	-	-
RMO	Failed	-	-	-

CBMC fails to find the error, CheckFence fails as before.

5.3 Micheal and Scott queue

CBMC

Memory model	Expected	Result	Mean	SD
SC	Successful	Aborts	-	-
TSO	Successful	-	-	-
PSO	Failed	-	-	-
RMO	Failed	-	-	-
POWER	Failed	-	-	-

CheckFence

Memory model	Expected	Result	Mean	SD
TSO	Failed	Failed	0.8	0.42
RMO	Failed	Failed	1.2	0.46

We reproduce the results with CheckFence. CBMC is not able to run this example since it does not support dynamic memory.

5.4 Worker synchronization in PostgreSQL

CBMC

Memory model	Expected	Result	Mean	SD
SC	Successful	Successful	0.05	0.016
TSO	Successful	Successful	8.02	0.062
PSO	Successful	Successful	17.51	0.077
RMO	Successful	Successful	18.3	0.16
POWER	Failed	Failed	19.3	0.36

CheckFence

Memory model	Expected	Result	Mean	SD
TSO	Failed	-	-	-
RMO	Failed	-	-	-

Checkfence fails(vacuous pass). The results described in [3] are reproduced using CBMC.

5.5 Lockless split deque

CBMC

Memory model	Result	Mean	SD
SC	Successful	307	6.9
TSO	Aborts		
PSO	Aborts		
RMO	Aborts		
POWER	Aborts		

CheckFence

Memory model	Expected	Result	Mean	SD
TSO	Failed	-	-	-
RMO	Failed	-	-	-

CheckFence fails (vacuous pass). CBMC aborts under weak memory models, this seems to be caused by a bug in the instrumentation tool.

6. ANALYSIS

In this section we discuss the results of the benchmarks. Firstly we will describe the results of CBMC, next we will discuss the results of CheckFence. Unfortunately, we were not able to run the benchmarks on DFENCE, this is left for future work. We did run it on a provided example, however it is not sufficient to draw conclusions. Lastly we briefly describe how errors in programs can be traced by the tools.

For CBMC we reproduce the published results in the Dekker and PostgreSQL algorithms. For the example found by the Sober tool instrumentation finds cycles, however SatAbs does not find a bug. The more complex lockless split deque verifies successful under sequential consistency. The instrumentation tool finds a lot of cycles (more than one million for PSO), however SatAbs aborts when running the instrumented program. Dumping the instrumented program to C code seems to point to the source of this problem which seems to be a bug in goto-instrument in which pointer dereference does not occur correctly. Fixing these errors manually and verifying the code with SatAbs results in a running time over 24 hours. The Michael and Scott queue example can not be implemented on CBMC, since as far as we know goto-cc does not support dynamic memory allocation. Using static memory (arrays) results in the same bug observed in the Lockless split deque example. Under SC, SatAbs fails with an error. The exact reason is unknown.

CheckFence fails in all examples except for the Michael Scott queue, which is supplied as an example with the tool. The reason for the failure is not clear. Moreover, TSO does not seem to be implemented or working on CheckFence, since it provides the same output as RMO. Another point to mention is that CheckFence requires the user to define the input and output actions to observe, whereas CBMC uses assertions.

When it comes to finding errors SatAbs provides a trace of instructions of lines executed leading to the property violation, but without the actual stores and loads. When CheckFence finds an execution that is not included in the observation set it provides a view of execution per thread and per line it displays which values are stored and loaded and to what location.

7. CONCLUSION

Firstly it can be said that the tools are not easily accessible: Although the tools are open-source software, only CBMC provides a working binary. Both CheckFence and DFENCE require changes to the source code in order to compile without errors. Furthermore, the tools do not have a manual on how to use them. This is especially important considering they each support a subset of the C language and use their own built-in constructs for spawning threads, using locks, atomic operations, etc.
As far as the results go, we can conclude the tools fail on each other's examples. The instrumentation tool of CBMC does not support dynamic memory allocation and there seems to be a bug when accessing array elements. CheckFence fails on the other examples; the reason for this however is not known. When it comes to memory model support, CBMC supports the most models. Efficiency wise, it is hard to quantitatively conclude anything about the results. It seems to be very discrete, either it fails, or it runs for a few seconds, or it runs for longer than 24 hours.

8. FURTHER WORK

This project serves as an overview of the state of research in verification of software under weak memory models. An important goal of research in this field is a tool which can be used to analyze real-world concurrent software under various relaxed models. Clearly, current tools are insufficient: DFENCE and CheckFence lack a good user interface and overall it was hard to find a non-trivial example which all tools could handle. It is hard to compare the verification methods solely on the information provided in the papers. Tool-wise CBMC supports the most memory models and has a very flexible and straightforward way of doing the actual verification. Fixing the pointer bug and allowing for dynamic memory allocation should open it up to more examples. CheckFence seems to have some bugs which need to be fixed (see results and analysis section), the source of the bugs have not been found. DFENCE could definitely use a better user interface which would open it up for testing more examples. The program now needs to be recompiled for each new example. We recommend new tools or improvements of these tools to at least run the benchmarks described in the method section. A further study in the current state of research in the verification of relaxed memory models could expand on the results gathered by running DFENCE and other new tools on the benchmark.

9. REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

[2] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli. The semantics of power and arm multiprocessor machine code. In *Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pages 13–24. ACM, 2009.

[3] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *Programming Languages and Systems*, pages 512–532. Springer, 2013.

[4] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *ACM Sigplan Notices*, volume 45, pages 7–18. ACM, 2010.

[5] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

[6] S. Burckhardt, R. Alur, and M. M. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *ACM SIGPLAN Notices*, volume 42, pages 12–21. ACM, 2007.

[7] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *Computer Aided Verification*, pages 107–120. Springer, 2008.

[8] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Satabs: Sat-based predicate abstraction for ansi-c. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 570–574. Springer, 2005.

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT press, 1999.

[10] E. W. Dijkstra. Co-operating sequential processes. f. *Programming Languages. Academic Press, New York*, 1968.

[11] P. Gepner and M. F. Kowalik. Multi-core processors: New way to achieve high system performance. In *Parallel Computing in Electrical Engineering, 2006. PAR ELEC 2006. International Symposium on*, pages 9–13. IEEE, 2006.

[12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[13] A. Linden and P. Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *Model Checking Software*, pages 144–160. Springer, 2011.

[14] F. Liu, N. Nedev, N. Prisadnikov, M. Vechev, and E. Yahav. Dynamic synthesis for relaxed memory models. In *ACM SIGPLAN Notices*, volume 47, pages 429–440. ACM, 2012.

[15] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.

[16] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics*, pages 391–407. Springer, 2009.

[17] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding power multiprocessors. In *ACM SIGPLAN Notices*, volume 46, pages 175–186. ACM, 2011.

[18] B. Schauer. Multicore processors–a necessity. *ProQuest Discovery Guides1–14*, 2008.

[19] T. A. Sudkamp and A. Cotterman. *Languages and machines: an introduction to the theory of computer science*, volume 2. Addison-Wesley Reading, Mass., 1988.

[20] D. L. Weaver and T. Gremond. *The SPARC architecture manual*. PTR Prentice Hall Englewood Cliffs, NJ 07632, 1994.

[21] R. N. Zucker and J.-L. Baer. *A performance study of memory consistency models*, volume 20. ACM, 1992.

APPENDIX

Listing 2. Lockless split deque by T. van Dijk

```
def steal():
        if allstolen:
                return NOWORK
        (t,s) = (tail,split)
        if t < s:
                if cas((tail,split), (t,s), (t+1,s)):
                        return WORK(t)
        else:
                return BUSY
        elif not movesplit: movesplit = 1
        return NOWORK

def push(data):
        if head == size: return FULL
        write task data at head
        head = head + 1
        if o_allstolen:
                (tail,split) = (head-1,head)
                if movesplit:
                        movesplit = 0
                allstolen = 0
                o_split = head
                o_allstolen = 0
        elif movesplit:
                grow_shared()

def grow_shared():
        new_s = (o_split+head+1)/2
        split = new_s
        o_split = new_s
        movesplit = 0

def shrink_shared():
        (t,s) = (tail,split)
        if t != s:
                new_s = (t+s)/2
                split = new_s
                MFENCE
                t = tail # read again
                if t != s:
                        if t > new_s:
                                new_s = (t+s)/2
                                split = new_s
                        o_split = new_s
                        return False
        allstolen = 1
        o_allstolen = 1
        return True

def pop():
        if head == 0:
                return EMPTY
        if o_allstolen or (o_split == head and shrink_shared()):
                head = head-1
                return STOLEN(head)
        head = head-1
        if movesplit:
                grow_shared()
        return WORK(head)
```