Weak Memory Models as LLVM-to-LLVM Transformations[⋆]

Vladimír Štill, Petr Ročkai[⋆⋆], and Jiří Barnat

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{xstill,xrockai,barnat}@fi.muni.cz

Abstract. Data races are among the most difficult software bugs to discover. They arise from multiple threads accessing the same memory location, a situation which is often hard to discern from source code alone. Detection of such bugs is further complicated by individual CPUs' use of relaxed memory models. As a matter of fact, proving absence of data races is a typical task for automated formal verification. In this paper, we present a new approach for verification of multi-threaded C and C++ programs under weakened memory models (using store buffer emulation), using an unmodified model checker that assumes Sequential Consistency. In our workflow, a C or C++ program is translated into LLVM bitcode, which is then automatically extended with store buffer emulation. After this transformation, the extended LLVM bitcode is model-checked against safety and/or liveness properties with our explicit-state model checker DIVINE.

1 Introduction

Finding concurrency-related errors, such as deadlocks, livelocks and data races and their various consequences, is extremely hard – the standard testing approach does not allow the user to control the precise timing of interleaved operations. As a result, some concurrency bugs that occur under a specific interleaving of threads may remain undetected even after a substantial period of testing. To remedy this weakness of testing, formal verification methods, explicit-state model checking in particular, can be of extreme help.

Concurrent access to shared memory locations is subject to the so called memory model of the specific CPU in use. Generally speaking, in relaxed memory models, the visibility of an update to a shared memory variable may be postponed or even reordered with other updates to different memory locations. Unfortunately, most programming and modelling languages were designed to

[⋆] This work has been partially supported by the Czech Science Foundation grant No. 15-08772S.

[⋆⋆] The contribution of Petr Ročkai has been partially supported by Red Hat, Inc.

[†] The final publication is available at Springer via `http://dx.doi.org/10.1007/978-3-319-29817-7_13`

merely mimic the principles of the underlying sequential computation machine, and therefore lack the syntactic and semantic constructs required to express low-level details of the concurrent computation and the memory model of the underlying hardware architecture in particular. Moreover, for obvious reasons, programmers design parallel algorithms with the *Sequential Consistency* [14] memory model in mind, under which any write to or read from a shared variable is instantaneous and immediately visible to all concurrent threads or processes – an assumption that is far from the reality of contemporary processors.

To protect from inconsistencies due to the reordered or delayed memory writes in the relaxed memory model architectures, specific low-level hardware mechanisms, such as memory barriers, have to be used. A memory barrier makes sure that all the changes done prior the barrier instruction are visible to all other processes before any other instruction *after* the barrier is executed. For more details on how memory barriers work we kindly refer the reader to technical literature. Naturally, the implementation details of a specific relaxed memory model depend on the brand and model of a CPU in use [19].

As a result, programs written in programming languages such as C do not contain enough information for the compiler to emit the code whose behaviour is both correct with respect to the incomplete specification given by the source code and at the same time as efficient as possible. A widely accepted compromise is that sequential code is guaranteed to be semantically correct, but any concurrent data access is the responsibility of the programmer. Such access needs to be guarded with various programming and modelling language addons such as builtin compiler functions, operating system calls, atomic variables with (optional) explicit memory ordering specification, or other non-language mechanisms. Since the correctness of behaviour depends on a human decision, often the resulting binary code does not do exactly what it was intended to do by its developer.

This is exactly where formal verification by model checking can help. The model checking procedure [7] systematically explores all configurations (states) of a program under analysis to discover any erroneous or unwanted behaviour of the program. The procedure can easily reveal states of the program that are only reachable under a very specific thread interleaving; clearly, such states may be very hard to reach with testing alone. Examples of explicit-state model checkers include SPIN [10], DIVINE [4], or LTSmin [12]. Unfortunately, none of the mentioned model checkers have direct support for model checking programs under relaxed memory models. Instead, should a user be interested in verification of a program under relaxed memory model, the program has to be manually (or semi-manually) augmented to capture relaxed memory behaviour.

The main contribution of our paper is in a new strategy to automate model checking of C and C++ programs under relaxed memory model without the need of modification of the interpreter used by the model checker itself. Note that interpreting C and C++ alone is a challenging task and any extension of the interpreter towards relaxed memory models would only make it harder. In fact model checkers do not typically rely on direct interpretation of C or C++ code, but use some other, syntactically simpler, representation of the original program. The model checker DIVINE, for example, interprets LLVM bitcode, which is an intermediate representation of the program created by an LLVM-based compiler.

In order to perform verification of C and C++ programs under relaxed memory model, we suggest to augment the original program and extend it with further data structures (store buffers and a cleanup thread) to simulate the behaviour of the original program under relaxed memory model. However, for the same reasons as above, we avoid direct transformation of C or C++ programs – it would require to parse the complex syntax of a high-level programming language. Instead, we apply the transformation at the level of LLVM bitcode, after the original program is translated by a C++ compiler, but before the representation is passed to the model checker for verification. This scenario allows us to completely separate the weak memory extension from the use of a model checker, hence, it allows us to use any model checker capable of processing LLVM bitcode under Sequential Consistency. Our LLVM bitcode to LLVM bitcode transformation adds store buffer data emulation to under-approximate Total Store Order (TSO) – a particular theoretical model of a relaxed memory model. The transformation is implemented within the tool called LART (LLVM Abstraction and Refinement Tool, Section 7.1 in [22]) that is distributed as a part of DIVINE model checker bundle, under the 2-clause BSD licence.

The rest of the paper is organised as follows. Section 2 lists the most relevant related work, Section 3 gives all the details of the LLVM transformation, Section 4 describes some relevant but rather technical implementation details, Section 5 gives details on an experimental evaluation of our approach, and finally Section 6 concludes the paper.

2 Related Work

The idea of using model checkers to verify programs under relaxed memory models has been discussed first in connection with the explicit-state model checker Murφ [8]. The tool was used to generate all possible outcomes of small, assembly language, multiprocessor programs using a given memory model [21]. This was achieved by encoding the memory model and program under analysis in the Murφ description language, which is an idea applied in many later papers, including this one.

To cope with the rather complex situation around memory models, theoretical models have been introduced to cover as many instances of different relaxed memory behaviours as possible. The currently most used theoretical models are the *Total Store Order* (TSO) [25], *Partial Store Order* (PSO) [25] and *x86-TSO* which is a Total Store Order enriched with interlocking instructions [16]. In those theoretical models, an update may be deferred for an infinite amount of time. Therefore, even a finite state program that is instrumented with a possibly infinite delay of an update may exhibit an infinite state space. It has been proven that for such an instrumented program, the problem of reachability of a particular system configuration is decidable, but the problem of repeated reachability of a given system configuration is not [2].

A particular technique that incorporates TSO-style store buffers into the model and uses finite automata to represent the possibly infinite set of possible contents of these buffers has been introduced in [16]. Since the state space explosion problem is even worse with TSO buffers incorporated into the model, authors of [16] extended their approach with a partial-order reduction technique later on [17].

A different approach has been taken in [11], where the algorithm to be analysed was transformed into a form where the statements of the algorithm could be reordered according to a particular weak memory ordering. The transformed algorithm was then analysed using a model-checking tool, SPIN in that case.

A lot of research has been conducted to actually detect deviation of an execution of the program on a relaxed memory model architecture from an execution under Sequential Consistency (SC). An SC deviation run-time monitor using operational semantics [18] of TSO and PSO was introduced in [6], where authors considered a concrete, sequentially consistent execution of the program, and simulated it on the operational model of TSO and PSO by buffering stores, as long as they generated the same trace as the SC execution. Another approach to detect discrepancies between a sequential consistency execution and real executions relied on axiomatic definition of memory models and (SAT-based) bounded model checking [5].

The problem of relaxed memory model computation has been addressed also in the program analysis community. Given a finite-state program, a safety specification and a description of the memory model, the framework introduced in [20] computes a set of ordering constraints that guarantee the correctness of the program under the memory model. The computed constraints are maximally permissive: removing any constraint from the solution would permit an execution that violates the specification. To address the undecidability of the problem, an abstraction from precise memory models has been considered by the BLENDER tool [13]. The tool employs abstract interpretation to deliver an effective verification procedure for programs running under relaxed memory models.

Another program analysis tool, called OFFENCE, was introduced to ensure program stability [1] by inserting a memory barrier instruction where needed – an approach also used in [17]. The problem of relaxed memory model and correct placement of synchronisation primitives is also relevant for the compiler community [9].

The problem of LTL model checking for an under-approximated TSO memory model using store buffers was also evaluated in [3], where authors proposed transformation of the DVE modelling language programs to simulate TSO.

3 Emulation of Relaxed Memory in LLVM Bitcode

We have chosen to provide an under-approximation of the TSO memory model, both for its simplicity and the fact that it closely resembles the memory model

```
   int x = 0, y = 0;

1  void thread0() {                      1  void thread1() {
2      y = 1;                            2      x = 1;
3      cout << "x = " << x << endl;      3      cout << "y = " << y << endl;
4    }                                   4  }
```

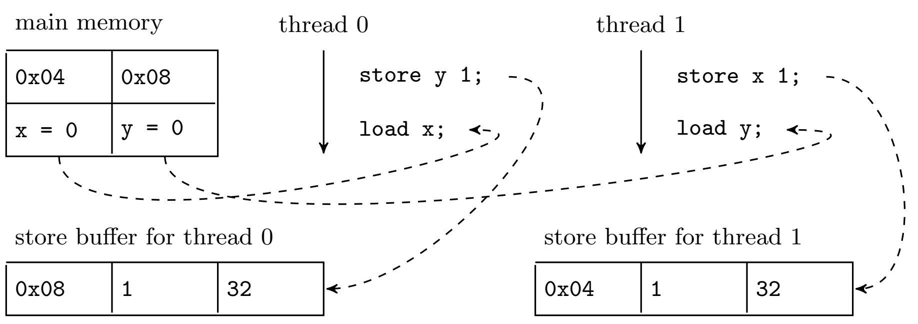

Fig. 1. In this example, each of the threads first writes into a global variable and later reads the variable written by the other thread. Under sequential consistency, the possible outcomes would be $x = 1, y = 1$; $x = 1, y = 0$; and $x = 0, y = 1$, since at least one write must proceed before the first read proceeds. However, under TSO $x = 0, y = 0$ is also possible: this corresponds to the reordering of the load on line 3 before the independent store on line 2, and can be simulated by performing the store on line 2 into a store buffer. The diagram shows (shortened) execution of the listed code. Dashed lines represent where given value is read from/stored to.

used by x86 computers. In this memory model, all stores are required to become visible in the same order as they are executed; however, loads can be executed before independent stores. This situation can be emulated by per-thread store buffers – stores are performed into store buffers and later flushed into main memory. Loads then have to first consult their thread's respective store buffer, and if it does not contain the address in question, proceed by consulting the main memory. Loads do not see changes that are recorded only in store buffers of other threads. We can see an illustration of the TSO memory model, and its simulation using store buffers, in Figure 1. While in the sequentially consistent case, the result $x = 0, y = 0$ would not be possible, under TSO it is a valid output of the program, and indeed it can be proved reachable by running DIVINE on the transformed code. Note that store buffers are flushed non-deterministically, using a dedicated thread; in particular, we run a dedicated flushing thread for each worker thread.

Note that we deliberately avoid precise (unbounded store-buffer) simulation of the theoretical TSO memory model, as this could easily result in infinite state space of the program under verification. However, the store buffer size can be passed as a parameter to the bitcode transformation. This way, we can make both reachability and LTL verification decidable and connect it seamlessly to an existing explicit-state framework. Please note that this approach only under-approximates the set of all TSO behaviours. I.e., when DIVINE finds a counterexample in the modified model, this counterexample can indeed occur in some runs of the given program on some real hardware with TSO semantics. On the other hand, not finding a counterexample does not guarantee error free execution on machines with store buffers deeper than specified for verification. Obviously, setting the size of store buffers is a matter of compromise – larger buffers will result in more precise verification, but also in a larger state spaces.

3.1 Infinite Delay Problem

For safety properties, such as assertion violation and/or memory safety, delaying writes indefinitely (never flushing them from a store buffer) is not a problem, as any violation of safety property is witnessed by finite path and for each run with infinite delay, there also exists (possibly finite) run where each write is eventually flushed. In infinite runs, however, such as those constructed as counterexamples to liveness properties, infinite delays could pose a problem. Imagine, for example, the following two threads:

```
bool x = false, y = false;
```

```
void thread0() {                    void thread1() {
    y = true;                           x = true;
    while ( !x ) { AP( w0 ) }           while ( !y ) { AP( w1 ) }
    for (;;) { /* work */ }             for (;;) { /* work */ }
}                                   }
```

and a liveness property written (using LTL) as $FG(\neg w_0 \wedge \neg w_1)$. Assuming a separate thread to perform store buffer flushes, it is easy to see that this property holds only if the buffers are actually flushed on every possible run. However, since flushing happens non-deterministically, it may actually never happen on an infinite run. While this can be viewed as theoretically correct, it does not correspond to any real-world behaviour, where delayed writes will eventually finish and the program eventually proceeds. To counteract this inconsistency, we ask our model checker to assume weak fairness [15], where it is guaranteed that every non-blocking thread has performed infinitely many actions in an infinite run.

In [3], authors proposed to handle this problem by extending LTL specification to include this store buffer fairness criteria. In our case though, we have chosen to implement our transformation in a way which does not require any additional specification and store buffer fairness is implied by the standard weak fairness.

3.2 Invalidated Variable Store Problem

Another issue to deal with are delayed flushes from a store buffer that come at the time when the object that should be written into does not exist anymore in the main memory. As both memory allocation and stack depth can change at the run-time, it might happen that an entry in the store buffer points to invalid location (either given memory chunk was deallocated by the user, or it lived in a stack frame that has already been abandoned). To solve this problem, we would need to make sure that inaccessible addresses are evicted from the store buffers. For dynamic memory, this can be done by overriding the function which deallocates objects from memory in such a way that it first iterates over all store buffers and evict entries into the to-be-freed memory before calling the original deallocate function.

For stack memory, however, the situation is more complicated – it is not sufficient to evict all the stack-frame-allocated memory from store buffers before returning from a function, because an exception can cause stack unwinding, which can also result in invalid references in store buffers. This means that cleanup handlers [24] need to be added to each function to deal with the situation.

4 Implementation

First of all, let us briefly explain how LLVM bitcode is used by our target model checker DIVINE to support for C/C++ verification. There are two levels below the LLVM bitcode of the program to be verified – an interpreter and an LLVM *userspace*. The interpreter is used directly by the model checker to generate and explore the state space graph by executing LLVM instructions. The interpreter detects errors such as invalid memory dereference, memory leaks, assertion violations, etc. The interpreter has to be aware of threads and dynamic memory management, hence, its role is similar to what the CPU and the core of the operating system do when executing the code natively. The userspace, on the other hand, corresponds to the runtime of the programming language, that is, it provides LLVM bitcode for the basic libraries required by the given programming language and/or threading model. The userpsace and interpreter together provide the user with a standards-compliant interface for user's programming language of choice.

While in general, the separation of work between the interpreter and userspace could be almost arbitrary (one could, for example, include the entire pthread library in the interpreter), it is advantageous to keep the interpreter as simple as possible, pushing most of the required functionality into the userspace. Therefore, DIVINE provides a fairly small set of intrinsic functions (sixteen in total), which give access to the necessary functionality provided by the interpreter. The rest is left to userspace.

The support for relaxed memory verification, such as functions that simulate store buffers, thus need not come separately for every program to be verified under relaxed memory model, but may actually become a part of the DIVINE LLVM userspace. However, it is not possible to implement weak memory simulation through addition of userspace functions alone – we need to change the behaviour of memory manipulation instructions (such as loads, stores, and fences). For this reason, we implemented an LLVM to LLVM bitcode transformation pass, which translates relevant instructions into calls to the relevant userspace functions. The actual simulation of the memory model is thus implemented within the userspace and is separate from the original program. As a result of this design choice, this transformation can be easily modified to work with other LLVM model checkers and with different weak memory models.

4.1 Updates to LLVM Userspace

Currently, LLVM userspace provides replacement functions for `load`, `store` and `fence`. The relevant userspace functions can be identified by their `__lart_weakmem` prefix. Store buffers are represented by a thread-local array with one record for each store – this record contains the address, the value itself and the bit width of the value. We have chosen to limit a single store to 64 bits, which is the usual size atomically written by modern CPUs and also the maximal size of standard integer types in C. Each store then pushes a record into the local store buffer, while loads first consult the local store buffer for an up-to-date value, and if it is not present proceed to load from memory. A fence flushes all the entries from the local store buffer.

Note that block memory manipulation functions have to be replaced too, to protect them from bypassing the store buffers. Hence, the userspace provides replacements for block memory manipulation functions such as `llvm.memmove`, `llvm.memcpy`, etc.

Further, atomic LLVM instructions, e.g. `cmpxchg`, are rewritten within the transformation to use only functions implemented within the userspace. However, we currently only support sequentially-consistent ordering of atomics (which is the default ordering for atomic variables in C++11). Further extensions to support all atomic access orderings supported by LLVM/C++11 are planned.

Finally, attention had to be paid to initialisation of the store buffers. Due to the nature of global variable constructors in C++ which can run in arbitrary order, we cannot use non-trivial constructors for store buffers, as this could cause the constructor to run after some calls to `__lart_weakmem_*` functions have already happened. Therefore, the store buffer array is initialised to a null pointer and allocated in the first call to one of the `__lart_weakmem_*` functions.

4.2 LLVM to LLVM Transformation

The transformation is implemented as part of the LART tool. It basically iterates over all the instructions in the original LLVM bitcode and replaces some of them with calls to the corresponding replacement functions.

To perform this transformation correctly, we had to introduced special LLVM function attributes: *bypass*, *tso*, and *sc*, denoting in what mode a particular function should operate. Functions marked *bypass* are not subject to the transformation at all, functions marked *tso* are fully processed by the transformation as indicated above. In functions marked *sc*, additional memory barriers are inserted at the beginning of the function and after a call to any non-SC function.

Note that it is important that the functions which implement the relaxed weak memory model itself are not transformed; for this reason, all `__llvm_weakmem_*` functions are annotated as *bypass*. The default behaviour of the transformation on functions that are not annotated with any of the attributes can be set by a parameter passed to the transformation.

Since LLVM allows loads and stores larger than 64 bits (either large scalar types, such as 128 bit integers, or aggregate values), we first break these large loads and stores into chunks of at most 64 bit-wide operations in a separate transformation pass and only after this is done, we perform the instruction substitution transformation as outlined above.

Finally, to avoid interference from compiler optimisations, some of the memory accesses in our functions had to be marked volatile and we had to prevent inlining of some of the functions (since inlining would discard function attributes). Likewise, all the exposed functions had to be marked `noinline`.

4.3 State Space Reduction

Store buffers substantially increase the size of the state space, hence it is necessary to counteract this growth. DIVINE provides powerful reduction techniques out of the box, based on analysis of instruction visibility. Those reductions are, however, rendered less effective by interactions with the store buffer: in particular, any TSO load or store is treated as visible by the $\tau+$ reduction due to global variable access within the TSO load/store implementation.

Fortunately, it is possible to reduce the overhead of store buffers by entirely bypassing their use for memory locations that are private to a particular thread. However, since the entire logic of TSO stores is handled in the userspace, it is necessary to expose an additional intrinsic (builtin) function in the model checker, which, for a given address, decides whether the address is visible from any other threads.

As far as correctness is concerned, when we realise that from the point of view of the model checker, store buffers are part of the global memory, the argument carries over from the analogical construct (store visibility) used in $\tau+$ reduction [23]. Any pointers currently residing in store buffers – and hence, capable of revealing new memory locations to foreign threads – are treated as global; hence, a delayed write of such a pointer cannot incorrectly hide intervening stores (into locations that were previously thread-private but revealed by the pointer living in a store buffer).

5 Evaluation

We evaluated our approach on a few models, all of which can be found in examples in source distribution of DIVINE[1]. Descriptions of the models used can be found in Table 1. All measurements were performed on a laptop with Intel Core i7-3520M, running at 3.4 GHz, with 8 GB of memory. DIVINE used

[1] online: https://divine.fi.muni.cz/trac/browser/examples/llvm/weakmem/

Table 1. Models used for evaluation

`simple_sc`	Model based on figure 1, SC, asserting that $x = 0, y = 0$.
`simple_mtso`	Same model, but manually modified to use TSO for relevant variables.
`simple_stso`	Same model, workers are auto-transformed to TSO, the rest is SC.
`simple_tso`	Same model, fully transformed to TSO.
`peterson_sc`	Peterson's mutual exclusion algorithm.
`peterson_tso`	The same, automatically transformed to TSO.
`fifo_sc`	First-in, first-out, lockless inter-thread queue, as used in DIVINE.
`fifo_tso`	Automated TSO transform of `fifo_sc` above.

4 threads for verification and never depleted available memory (loss-less state space compression was enabled).

5.1 Results

The results of verification with DIVINE can be seen in Table 2. In all cases, Context-Switch-Directed Reachability [26] was used, as it performed much faster than regular reachability for the TSO simulation case. From the results, we can see significant increase of state space size when store buffers are enabled. This is due to two factors – one of them is that the store buffers themselves increase the state space size, as they can be flushed non-deterministically anywhere between the given store and the nearest memory barrier. The other issue is the interference with $\tau+$ reduction mentioned in Section 4.3. As can be seen in the case of `peterson_sc` and `peterson_tso` with store buffers of size 0 (in this case value is stored into store buffer and immediately flushed out within one transition in the state space), this effect is quite strong.

As for the differences between different versions of the `simple` model, the state space size is clearly dependent on how many of the loads and stores are treated as TSO – in case of full TSO transformation all library functions are also in TSO, therefore state space size is increased far more. The difference between `simple_mtso` and `simple_stso` is more subtle: in the case of `simple_stso` our transformation adds memory barriers into SC functions, at their beginning and after any call to non-SC function. While the second case is rarely present in our model, the first case makes any function call observable, as a flush will be considered observable by $\tau+$ reduction (due to an accesses to the store buffer).

6 Conclusion

We have introduced an LLVM to LLVM transformation that extends a program with relaxed memory simulation and we have shown that such an extended program can be passed to a model checker to perform verification of C/C++ programs under a relaxed memory model. A key attribute of our approach is that no updates to the model checker (which is based on sequential consistency) are needed. The preliminary experiments show the approach as such is feasible,

Table 2. Results of `divine verify` for our examples.

model	store buffer size	assertion violated	# of states	reduced # states	memory [GB]	time [s]
`simple_sc`	N/A	no	205	N/A	0.16	1
`simple_mtso`	1	**yes**	6.89 k	N/A	0.17	3
`simple_stso`	1	**yes**	10.7 k	10.7 k	0.17	6
`simple_tso`	1	**yes**	24.7 M	537.2 k	3.18	20318
`peterson_sc`	N/A	no	1.68 k	N/A	0.16	1
`peterson_tso`	0	no	55.9 k	N/A	0.17	38
`peterson_tso`	2	**yes**	2.86 M	95.7 k	0.79	990
`peterson_tso`	3	**yes**	4.70 M	129.9 k	1.21	1610
`fifo_sc`	0	no	6951	N/A	0.73	20
`fifo_tso`	1	no	–	44 M	–	–

even though the growth of the state space is significant. Finally, the verification of the `fifo_tso` model is, in itself, a valuable result, as the code in question is sensitive to memory ordering and until now we were only able to verify it under the assumption of sequential consistency.

As our future work we intend to improve the implementation and also implement support for weaker memory models, such as Partial Store Order. As a research goal, we want to extend LART to automatically annotate some functions as SC, whenever it can be statically decided that such an annotation has no influence on the verification result, counteracting the growth of the state space. Further improvements of reductions supported by DIVINE and their interaction with store buffer simulation, and thread-local memory in general, could also significantly reduce the state space.

References

1. J. Alglave and L. Maranget. Stability in weak memory models. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 50–66, Berlin, Heidelberg, 2011. Springer.
2. M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 7–18, New York, NY, USA, 2010. ACM.
3. J. Barnat, L. Brim, and V. Havel. LTL Model Checking of Parallel Programs with Under-Approximated TSO Memory Model. In *Application of Concurrency to System Design (ACSD)*, pages 51–59. IEEE, 2013.
4. J. Barnat, L. Brim, V. Havel, and J. Havlíček et al. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *CAV*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013.
5. S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *CAV*, volume 5123 of *LNCS*, pages 107–120. Springer, 2008.
6. J. Burnim, K. Sen, and C. Stergiou. Sound and Complete Monitoring of Sequential Consistency in Relaxed Memory Models. Technical Report UCB/EECS-2010-31, EECS Department, University of California, Berkeley, March 2010.

7. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT press, 1999.
8. D. Dill. The Murphi Verification System. In *Computer Aided Verification*, volume 1102 of *LLNC*, pages 390–393. Springer, 1996.
9. X. Fang, J. Lee, and S. P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *International Conference on Supercomputing (ICS'03)*, pages 285–294. ACM, 2003.
10. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2004.
11. B. Jonsson. State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version). *SIGARCH Comput. Archit. News*, 36:65–71, June 2009.
12. G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk. LTSmin: High-Performance Language-Independent Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9035 of *LNCS*, pages 692–707. Springer, 2015.
13. M. Kuperstein, M. Vechev, and E. Yahav. Partial-coherence abstractions for relaxed memory models. In *Programming language design and implementation (PLDI'11)*, pages 187–198. ACM, 2011.
14. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.
15. D. J. Lehmann, A. Pnueli, and J. Stavi. Impartiality, Justice and Fairness: The Ethics of Concurrent Termination. In *Automata, Languages and Programming (ICALP)*, volume 115 of *LNCS*, pages 264–277. Springer, 1981.
16. A. Linden and P. Wolper. An Automata-Based Symbolic Approach for Verifying Programs on Relaxed Memory Models. In *Model Checking Software*, volume 6349 of *LNCS*, pages 212–226. Springer, 2010.
17. A. Linden and P. Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *Proceedings of the 18th international SPIN conference on Model checking software*, pages 144–160, Berlin, Heidelberg, 2011. Springer.
18. S. Mador-Haim, R. Alur, and M. M. K. Martin. Specifying relaxed memory models for state exploration tools. In $(EC)^2$*: Workshop on Exploting Concurrency Eficiently and Correctly*, 2009.
19. P. E. Mckenney. Memory Barriers: a Hardware View for Software Hackers, 2009.
20. M.Kuperstein, M. T. Vechev, and E. Yahav. Automatic inference of memory fences. In *Formal Methods in Computer-Aided Design*, pages 111–119. IEEE, 2010.
21. S. Park and D. Dill. An executable specification and verifier for relaxed memory order. *IEEE Trans. on Computers*, 48(2):227–235, 1999.
22. P. Ročkai. *Model Checking Software.* Disertation thesis, Masaryk University, Faculty of Informatics, 2015.
23. P. Ročkai, J. Barnat, and L. Brim. Improved State Space Reductions for LTL Model Checking of C & C++ Programs. In *NFM*, volume 7871 of *LNCS*, pages 1–15. Springer, 2013.
24. P. Ročkai, J. Barnat, and L. Brim. Model Checking C++ with Exceptions. *Automated Verification of Critical Systems*, 70, 2014.
25. CORPORATE SPARC International, Inc. *The SPARC architecture manual (version 9).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
26. V. Štill, P. Ročkai, and J. Barnat. Context-Switch-Directed Verification in DIVINE. In *Mathematical and Engineering Methods in Computer Science MEMICS 2014*, volume 8934 of *LNCS*, pages 135–146. Springer, 2014.