Model Checking of C++ Programs under the x86-TSO Memory Model[⋆]

Vladimír Štill and Jiří Barnat

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{xstill,xbarnat}@mail.muni.cz

Abstract. In this work, we present an extension of the DIVINE model checker that allows for analysis of C and C++ programs under the x86-TSO relaxed memory model. We use an approach in which the program to be verified is first transformed, so that it itself encodes the relaxed memory behavior, and after that it is verified by an explicit-state model checker supporting only the standard sequentially consistent memory. The novelty of our approach is in a careful design of an encoding of x86-TSO operations so that the nondeterminism introduced by the relaxed memory simulation is minimized. In particular, we allow for nondeterminism only in connection with memory fences and load operations of those memory addresses that were written to by a preceding store. We evaluate and compare our approach with the state-of-the-art bounded model checker CBMC and stateless model checker Nidhugg. For the comparison we employ SV-COMP concurrency benchmarks that do not exhibit data nondeterminism, and we show that our solution built on top of the explicit-state model checker outperforms both of the other tools. The implementation is publicly available as an open source software.

1 Introduction

Almost all contemporary processors exhibit relaxed memory behavior, which is caused by cache hierarchies, instruction reordering, and speculative execution. This, together with the rise of parallel programs, means that programmers often have to deal with the added complexity of programming under relaxed memory. The behavior of relaxed memory can be highly unintuitive even on x86 processors, which have stronger memory model than most other architectures. Therefore, programmers often have to decide whether to stay relatively safe with higher level synchronization constructs such as mutexes, or whether to tap to the full power of the architecture and risk subtle unintuitive behavior of relaxed

[⋆] This work has been partially supported by the Czech Science Foundation grant No. 18-02177S and by Red Hat, Inc.

[†] The final publication is available at Springer via https://doi.org/10.1007/978-3-030-02450-5_8

memory accesses. For these reasons, it is highly desirable to have robust tools for finding bugs in programs running under relaxed memory.

Our aim is primarily to help with the development of lock-free data structures and algorithms. Instead of using higher level synchronization techniques such as mutexes, lock-free programs use low-level atomic operations provided by the hardware or programming language to ensure correct results. This way, lock-free programs can exploit the full power of the architecture they target, but they are also harder to design, as the ordering of operations in the program has to be considered very carefully. We believe that by providing a usable validation procedure for lock-free programs, more programmers will find courage to develop fast and correct programs.

Sadly, conventional validation and verification techniques often fail to detect errors caused by relaxed memory. Many of these techniques work best for deterministic, single-threaded programs, and techniques applicable to parallel programs often assume the memory is *sequentially consistent*. With sequentially consistent memory, any memory action is immediately visible to all processors and cores in the system, there is no observable caching or instruction reordering. That is, an execution of a parallel program under sequential consistency is an interleaving of actions of its threads [25]. Recently, many techniques for analysis and verification which take relaxed memory into account have been developed, and research in this field is still pretty active. In this work, we are adding a new technique which we hope will make the analysis of C and C++ programs targeting x86 processors easier.

Our technique is built on top of DIVINE, an explicit-state model checker for C and C++ programs [8]. DIVINE targets both sequential and parallel programs and can check a range of safety properties such as assertion safety and memory safety. We extend DIVINE with the support for the x86-TSO memory model [34] which describes the relaxed behavior of x86 and x86_64 processors. Due to the prevalence of the Intel and AMD processors with the x86_64 architecture, the x86-TSO memory model is a prime target for program analysis. It is also relatively strong and therefore underapproximates most of the other memory models – any error which is observable on x86-TSO is going to manifest itself under the more relaxed POWER or ARM memory models.

To verify a program under x86-TSO, we first transform it by encoding the semantics of the relaxed memory into the program itself, i.e. the resulting transformed program itself simulates nondeterministically relaxed memory operations. To reveal an error related to the relaxed memory behavior, it is then enough to verify the transformed program with a regular model checker supporting only the standard sequentially consistent memory.

In this paper we introduce a new way of encoding the relaxed memory behaviour into the program. Our new encoding introduces low amount of nondeterminism, which is the key attribute that allows us to tackle model checking of nontrivial programs efficiently. In particular, we achieve this by delaying nondeterministic choices arising from x86-TSO as long as possible. Our approach is based on the standard operational semantic of x86-TSO with store buffers, but it removes entries from the store buffer only when a load or a fence occurs (or if the store buffer is bounded and full). Furthermore, in loads we only remove those entries from store buffers that relate to the address being loaded, even if there are some older entries in the store buffer.

The rest of the paper is structured as follows: Section 2 contains preliminaries for our work, namely information about relaxed memory models in general and the x86-TSO memory model in particular, and about DIVINE. Section 3 then presents our contribution, details about its implementation, and integration with the rest of DIVINE. Section 4 provides evaluation results which compare DIVINE to Nidhugg [1] and CBMC [14] on a range of benchmarks from SV-COMP [9]. Section 5 summarizes related work and Section 6 concludes this work.

2 Preliminaries

2.1 Relaxed Memory Models

The relaxed behavior of processors arises from optimizations in cache consistency protocols and observable effects of instructions reordering and speculation. The effect of this behavior is that memory-manipulating instructions can appear to be executed in a different order than the order in which they appear in the binary, and their effect can even appear to be in different order on different threads. For efficiency reasons, virtually all modern processors (except for very simple ones in microcontrollers) exhibit relaxed behavior. The extent of this relaxation is dependent on the processor architecture (e.g., x86, ARM, POWER) but also on the concrete processor model. Furthermore, the actual behavior of the processor is often not precisely described by the processor vendor [34]. To abstract from the details of particular processor models, *relaxed memory models* are used to describe (often formally) behavior of processor architectures. Examples of relaxed memory models of modern processors are the memory model of x86 and x86_64 CPUs described formally as x86-TSO [34] and the multiple variants of POWER [33, 27] and ARM [19, 5, 31] memory models.

For the description of a memory model, it is sufficient to consider operations which affect the memory. These operations include loads (reading of data from the memory to a register in the processor), stores (writing of data from a register to the memory), memory barriers (which constrain memory relaxation), and atomic compound operations (read-modify-write operations and compare-and-swap operation).

2.2 The x86-TSO Memory Model

The x86-TSO is very similar to the SPARC Total Store Order (TSO) memory model [35]. It does not reorder stores with each other, and it also does not reorder loads with other loads. The only relaxation allowed by x86-TSO is that store can appear to be executed later than a load which succeeds it. The memory model does not give any limit on how long a store can be delayed. An example

```
int x = 0, y = 0;
void thread0() {
    y = 1;
    int a = x;
    int c = y;
}
void thread1() {
    x = 1;
    int b = y;
    int d = x;
}
```

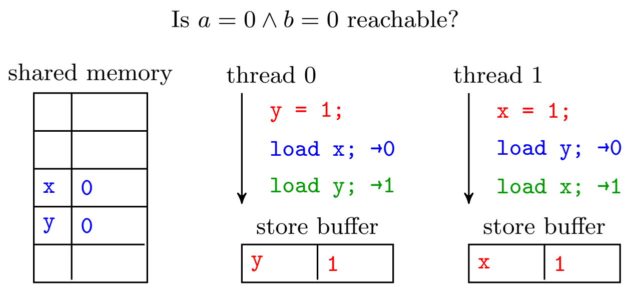


Fig. 1. A demonstration of the x86-TSO memory model. The thread 0 stores 1 to variable `y` and then loads variables `x` and `y`. The thread 1 stores 1 to `x` and then loads `y` and `x`. Intuitively, we would expect it to be impossible for $a = 0$ and $b = 0$ to both be true at the end of the execution, as there is no interleaving of thread actions which would produce such a result. However, under x86-TSO, the stores are cached in the store buffers (marked red). A load consults only shared memory and the store buffer of the given thread, which means it can load data from the memory and ignore newer values from the other thread (blue). Therefore `a` and `b` will contain old values from the memory. On the other hand, `c` and `d` will contain local values from the store buffers (locally read values are marked green).

of non-intuitive execution of a simple program under x86-TSO can be found in Figure 1.

The operational semantics of x86-TSO is described by Sewell et al. in [34]. The corresponding machine has hardware threads (or cores), each with associated local store buffer, a shared memory subsystem, and a shared memory lock. Store buffers are first-in-first-out caches into which store entries are saved before they are propagated to the shared memory. Load instructions first attempt to read from the store buffer of the given thread, and only if they are not successfull, they read from the shared memory. Store instructions push a new entry to the local store buffer. Atomic instructions include various read-modify-write instructions, e.g. atomic arithmetic operations (which take memory address and a constant),[1] or compare-and-swap instruction.[2] All atomic instructions use the shared memory lock so that only one such instruction can be executed at a given time, regardless of the number of hardware threads in the machine. Furthermore, atomic instructions flush the store buffer of their thread before they release the lock. This means that effects of atomic operations are immediately visible, i.e., atomics are sequentially consistent on x86-TSO. On top of these instructions,

[1] These instructions have the `lock` prefix in the assembly, for example `lock xadd` for atomic addition.

[2] `lock cmpxchg`

x86-TSO has a full memory barrier (`mfence`) which flushes the store buffer of the thread that executed it.[3]

To recover sequential consistency on x86, it is necessary to make memory stores propagate to the main memory before subsequent loads execute. This is most commonly done in practice by inserting memory fence after each store. An alternative approach would be to use atomic exchange instruction (`lock xchg`) which can atomically swap value between a register and a memory slot.

One of the specifics of x86 is that it can handle unaligned memory operations.[4] While the x86-TSO paper does not give any specifics about handling unaligned and mixed memory operations (e.g., writing a 64-bit value and then reading a 16-bit value from inside it) it seems from our own experiments that such the operations are not only fully supported, but they are also correctly synchronized if atomic instructions are used. This is in agreement with the aforementioned operational semantics of x86-TSO in which all the atomic operations share a single global lock.

2.3 DIVINE

DIVINE is an explicit-state model checker for C and C++ code that utilizes the clang compiler to translate the input program into the LLVM bitcode. This bitcode is then instrumented and interpreted by DIVINE's execution engine, DiVM. The complete workflow is illustrated in Figure 2. DIVINE focuses on both parallel and sequential programs and is capable of finding a wide range of problems such as memory corruptions, assertion violations, and deadlocks caused by improper use of mutexes. DIVINE also has very good support for C and C++, which it achieves by employing the standard clang compiler, and the libc++ standard library. Moreover, a few custom-built libraries are provided to enable full support of C++14 and C11 [8, 41]. To efficiently handle parallel programs, DIVINE employs state space reductions and has a graph based representation of program memory. More details about the internal architecture of DIVINE can be found in [32].

2.4 Relaxed Memory in C/C++ and LLVM

There are several ways in which C and C++ code can use atomic instructions and fences. These include inline assembly, compiler-provided intrinsic functions, and (since C11 and C++11) standard atomic variables and operations. While the constructs used to define atomic variables differ between C and C++, the memory model itself is the same for C11 and C++11. The C and C++ atomics are designed so that programmers can use the full potential of most platforms: the atomic operations are parametrized by a *memory order* which constrains how

[3] There are two more fence instructions in the x86 instruction set, but according to [34] they are not relevant to normal program execution.

[4] Other architectures, for example ARM, require loaded values to be aligned, usually so that the address is divisible by the value size.

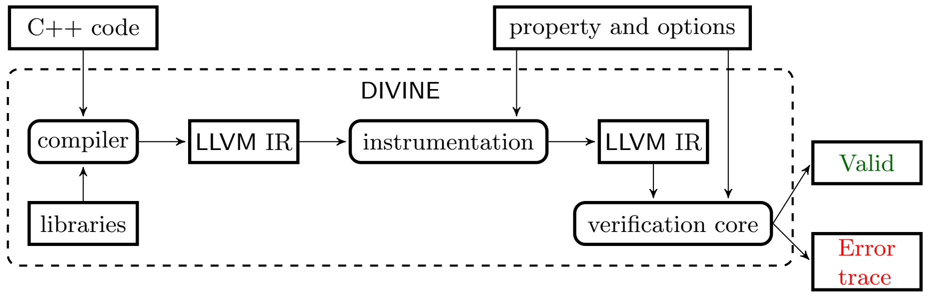


Fig. 2. Verification workflow of DIVINE when it is given a C++ file as an input. Boxes with rounded corners represent stages of input processing.

instructions can be reordered. The compiler is responsible for emitting assembly code which makes sure these ordering requirements are met. From the point of x86-TSO, all memory orderings except for sequential consistency amount to unconstrained execution, as such they exhibit non-atomic memory accesses.

When the C or C++ code is compiled to LLVM bitcode, the intrinsic functions and the standard atomic operations of the high-level programming language are mapped in the very same way to the corresponding LLVM instructions. The semantics of LLVM memory operations mostly copies the C++ memory model and behavior of the C++ atomic operations.

3 x86-TSO in DIVINE

DIVINE does not natively support relaxed memory, and we decided not to complicate the already complex execution engine and memory representation with a simulation of relaxed behavior. Instead, we encode the relaxed behavior into the program itself on the level of LLVM intermediate representation. The modified program running under sequential consistency simulates all x86-TSO runs of the original program, up to some bound on the number of stores which can be delayed. The program transformation is rather similar to the one presented in our previous work in [40]. The main novelty is in the way of simulation of x86-TSO which produces significantly less nondeterminism and therefore substantial efficiency improvements.

3.1 Simulation of the x86-TSO Memory Model

The most straight-forward way of simulating x86-TSO is to add store buffers to the program and flush them nondeterministically, for example using a dedicated flusher thread which flushes one entry at a time and interleaves freely with all other threads. We used this technique in [40]. This approach does, however, create many redundant interleavings as the flusher thread can flush an entry at any point, regardless of whether or not it is going to produce a run with a different memory access ordering, i.e. without any respect to the fact whether the flushed value is going to be read or not.

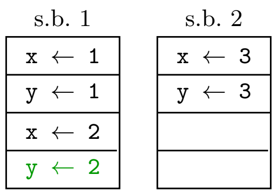
To alleviate this problem, it is possible to delay the choice whether to flush an entry from a store buffer to the point when the first load tries to read a buffered address. Only when such a load is detected, all possible ways the store buffers could have been flushed are simulated. In this case, the load can trigger flushing from any of the store buffers, to simulate that they could have been flushed before the load. To further improve the performance, only entries relevant to the loaded address are affected by the flushing. These are the entries with matching addresses and any entries which precede them in the corresponding store buffers (that are flushed before them to maintain the store order).

A disadvantage of this approach is that there are too many ways in which a store buffer entry can be flushed, especially if this entry is not the oldest in its store buffer, or if there are entries concerning the same addresses in multiple store buffers. All of these cases can cause many entries to be flushed, often with a multitude of interleavings of entries from different store buffers which has to be simulated.

Therefore, we propose a *delayed flushing*: entries in the store buffers can be kept in the store buffer after newer entries were flushed if the retained entries are marked as *flushed*. Such the entries behave as if they were already written to the main memory, but can still be reordered with entries in other store buffers. That is, when there is a flushed entry for a given location in any store buffer, the value stored in the memory is irrelevant as any load will either read the flushed entry or entry from the other store buffer (which can be written after the flushed entry). Flushed entries make it possible to remove store buffer entries out of order while preserving total store order. This way a load only affects entries from the matching addresses and not their predecessors in the store order. This improvement is demonstrated in Figures 3 to 5.

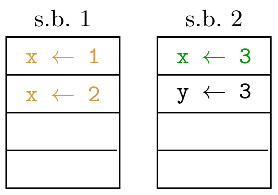
DIVINE handles C and C++ code by translating it to LLVM and instrumenting it (see Figure 2 for DIVINE's workflow). The support for relaxed memory is added in the instrumentation step, by replacing memory operations with calls to functions which simulate relaxed behavior. Essentially, all loads, stores, atomic instructions, and fences are replaced by calls to the appropriate functions.

All of the x86-TSO-simulating functions are implemented so that they are executed atomically by DIVINE (i.e., not interleaved). The most complex of these is the load operation. It first finds all entries with overlap the loaded address (*matching entries*) and out of these matching entries, it nondeterministically selects entries which will be written before the load (*selected entries*). All matching entries marked as flushed have to be selected for writing. Similarly, all matching entries which occur in a store buffer before a selected entry also have to be selected. Out of the selected entries, one is selected to be written last – this will be the entry read by the load. Next, selected entries are written, and all non-matching entries which precede them are marked as flushed. Finally, the load is performed, either from the local store buffer if matching entry exists there, or from the shared memory.



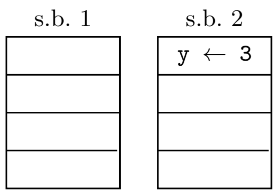
```
void thread0() {
    int a = y;
    int b = x;
}
```

Fig. 3. Suppose `thread0` is about to execute with the displayed contents of store buffers of two other threads and suppose it had nondeterministically chosen to load value 2 from `y` (denoted by green in the figure). The entries at the top of the store buffers are the oldest entries.



```
void thread0() {
    int a = y; // →2
    int b = x;
}
```

Fig. 4. At this point, `x` entries of store buffer 1 are marked as flushed (orange) and the `y ← 1` entry was removed as it was succeeded by the used entry `y ← 2`. The thread had nondeterministically selected to load `x` from store buffer 2.



```
void thread0() {
    int a = y; // →2
    int b = x; // →3
}
```

Fig. 5. In the load of `x`, all `x` entries were evicted from the buffers – all the flushed entries for `x` (which were not selected) had to be dropped before `x ← 3` was propagated to the memory. The last entry (`y ← 3`) will remain in the store buffer if `y` will never be loaded in the program again.

The remaining functions are relatively straightforward – stores push a new entry to the store buffer, possibly evicting the oldest entry from the store buffer if the store buffer exceeds its size bound; fences flush all entries from the store buffer of the calling thread to the memory; atomic operations are basically a combination of a load, store, and a fence. The only intricate part of these operations is that if an entry is flushed out of the store buffer, the entries from other store buffers which involve the same memory location can also be nondeterministically flushed (to simulate they could have been flushed before the given entry). This flushing is similar to flushing performed in load. An example which shows a series of loads can be found in Figures 3 to 5.

We will now argue that this way of implementing x86-TSO is correct. First, the nondeterminism in selecting entries to be flushed before a load serves the same purpose as the nondeterminism in the flusher thread of the more conventional implementation. The only difference is that in the flusher-thread scenario the entries are flushed in order, while in our new approach we are selecting only from the matching entries. Therefore, the difference between the two approaches is only on those entries which are not loaded by the load causing the flush, hence cannot be observed by the load. However, any entry which would be flushed before the selected entries in the flusher-thread approach is now marked with the flushed flag. This flag makes sure that such an entry will be flushed before an address which matches it is loaded, and therefore it behaves as if it was flushed. This way, the in-thread store order is maintained.

3.2 Stores to Freed Memory

As x86-TSO simulation can delay memory stores, special care must be taken to preserve memory safety of the program. More precisely, it is necessary to prevent the transformed program from writing into freed memory. This problem occurs if a store to dynamically allocated memory is delayed after the memory is freed, or if a store to stack location is delayed after the corresponding function had returned. This problem does not require special handling in normal program execution as both stack addresses as well as dynamic memory addresses remain to be writable for the program even after they are freed (except for memory mapped files, but these have to be released by a system call which includes sufficiently strong memory barrier).

To solve the problem of freed memory, it is necessary to evict store buffer entries which correspond to the freed memory just before the memory is freed. For entries not marked as flushed, this eviction concerns only store buffer of the thread which freed the memory. If some other thread attempted to write to the freed memory, this is an error as there is insufficient synchronization between the freeing and the store to the memory. However, corresponding entries marked as flushed should be evicted from all store buffers, as these entries correspond to changes which should have been already written to the shared memory. The program transformation takes care of inserting code to evict entries corresponding to freed memory from the store buffer.

3.3 Integration with Other Parts of DIVINE

The integration of x86-TSO simulation with the rest of DIVINE is rather straightforward. No changes are required in the DIVINE's execution engine or state space exploration algorithms. As for the libraries shipped with DIVINE, only minor tweaks were required. The `pthread` implementation had to be modified to add full memory barrier both at the beginning and at the end of every synchronizing functions. This corresponds to barriers present in the implementations used for normal execution, `pthread` mutexes and other primitives have to guarantee sequential consistency of the guarded operations (provided all accesses are properly guarded).

The DIVINE's operating system, DiOS, is used to implement low-level threading as well as simulation of various filesystem APIs [8]. We had to add memory barrier into the system call entry which hands control to DiOS. DiOS itself does not use relaxed memory simulation – the implementation of x86-TSO operations detects that the code is executed in the kernel mode and bypasses store buffers. In this way, the entire DiOS executes as if under sequential consistency. This synchronization is easily justifiable – system calls require a memory barrier or kernel lock in most operating systems.

3.4 Improvements

We have implemented two further optimizations of our x86-TSO simulation.

Static Local Variable Detection Accesses of local variables which are not accessible to other threads need not use store buffering. For this reason, we have inserted a static analysis pass which annotates accesses to local memory before the x86-TSO instrumentation. The instrumentation ignores such annotated accesses. The static analysis can detect most local variables which are never accessed using pointers.

Dynamic Local Memory Detection DIVINE can also dynamically detect if the given memory object is shared between threads (i.e., it is accessible from global variables or stacks of more then one thread). Using this information, it is possible to dynamically bypass store buffers for operations with non-shared memory objects. This optimization is correct even though the shared status of memory can change during its lifetime. A memory object o can become shared only when its address is written to some memory object s which is already shared (or o can become shared transitively through a series of pointers and intermediate objects). For this to happen, there has to be a store to the already shared object s, and this store has to be propagated to other threads. Once the store of the address of o is executed and written to the store buffer, o becomes shared, and any newer stores into it will go through the store buffer. Furthermore, once this store is propagated, any store which happened before turning o into a shared object also had to be propagated as x86-TSO does not reorder stores. Therefore, there is no reason to put stores to o through the store buffer if o is not shared. This optimization is not correct for memory models which allow store reordering – for such memory models, we would need to know that the object will never be shared during its lifetime.

3.5 Bounding the Size of Store Buffers

The complexity of analysis of programs under the x86-TSO memory model is high. From the theoretical point of view, we know due to Atig et al. [6] that reachability for programs with finite-state threads which run under TSO is decidable, but non-primitive recursive (it is in PSPACE for sequential consistency). The proof uses the so called SPARC TSO memory model [35] that is very similar to x86-TSO. However, the proof of decidability does not translate well to an efficient decision procedure, and real-world programs are much more complex than the finite-state systems used in the decidability proof.

For this reason, we would need to introduce unbounded store buffers to properly verify real-world programs. Unfortunately, this can be impractical, especially for programs which do not terminate. Therefore, our program transformation inserts store buffers of limited size, limiting thus the number of store operations that can be delayed at any given time. The size of the store buffers is fully configurable, and it currently defaults to 32, a value probably high enough to discover most bugs which can be observed on a real hardware.

Our implementation also supports the store buffers of unlimited size (when size is set to 0). In this mode, programs with infinite loops that write into shared memory will not have finite state space. Therefore, DIVINE will not terminate

Table 1. Comparison of the default configuration of DIVINE with CBMC and Nidhugg.

	CBMC	Nidhugg	DIVINE
finished	21	25	27
TSO bugs	3	3	9
unique	5	3	5

unless it discovers an error in the program. Verification with unbounded buffers will still terminate for terminating programs and for all programs with errors.

4 Evaluation

The implementation is available at `https://divine.fi.muni.cz/2018/x86tso/`, together with information about how to use it. We compared our implementation with the stateless model checker Nidhugg [1] and the bounded model checker CBMC [14, 24]. For evaluation we used SV-COMP benchmarks from the Concurrency category [9], excluding benchmarks with data nondeterminism[5] as our focus is on performance of relaxed memory analysis, not on handling of nondeterministic values. Furthermore, due to the limitation of stateless model checking with DPOR, Nidhugg cannot handle data nondeterminism at all. There are 55 benchmarks in total.

The evaluation was performed on a machine with 2 dual core Intel Xeon 5130 processors running at 2 GHz with 16 GB of RAM. Each tool was running with memory limit set to 10 GB and time limit set to 1 hour. The tools were not limited in the number of CPUs they can use.

We have used CBMC version 5.8 with the option `--mm tso`. Since there is no official release of Nidhugg, we have used version 0.2 from git, commit id `375c554` with `-tso` option to enable relaxed memory support and inserted a definition of the `__VERIFIER_error` function. For DIVINE, we have used the `--svcomp` option to enable support for SV-COMP atomic sections (which are supported by default by CBMC and Nidhugg), and we disabled nondeterministic memory failure by using the `divine check` command (SV-COMP does not consider the possibility of allocation failure). To enable x86-TSO analysis, `--relaxed-memory tso` is used for DIVINE.[6] The buffer bound was the default value (32) unless stated otherwise.

Table 1 compares performance of the default configuration of DIVINE with the remaining tools. The line "finished" shows the total number of benchmarks for which the verification task finished with the given limits. From these the line "TSO bugs" shows the number of errors caused by relaxed memory in benchmarks which were not supposed to contain any bugs under sequential consistency.

[5] I.e., all the benchmarks which contain calls to functions of the `__VERIFIER_nondet_*` family were excluded.

[6] The complete invocation is `divine check --svcomp --relaxed-memory tso BENCH.c`.

All discovered errors were manually checked to really be observable under the x86-TSO memory model. Finally, "unique" shows the number of benchmarks solved only by the given tool and not the other two. There were only 8 benchmarks solved by all three tools, all of them without any errors found.

Table 2. Comparison of various configurations of DIVINE. The "base" version uses none of the improvements from Section 3.4. The configurations marked with "s" add the static local variable optimization, while the configurations marked with "d" add the dynamic detection of non-shared memory objects. The "+sdu" configuration has both optimizations enabled and it has unbounded buffers. Finally, the "+sd4" has buffer bound set to 4 entries instead of the default 32 entries. The default version is "+sd".

	base	+s	+d	+sd	+sdu	+sd4
finished	26	26	27	27	27	27
TSO bugs	8	8	9	9	9	9

Table 3. Comparison of various versions of DIVINE on benchmarks on the 26 which all the versions finished. For the description of these versions, please refer to Table 2.

	base	+s	+d	+sd	+sdu	+sd4
states	252 k	263 k	250 k	231 k	206 k	296 k
time	2:14:49	2:17:13	1:09:23	1:05:05	0:58:28	1:24:59

Table 2 shows effects of buffer size bound and improvements described in Section 3.4. It can be seen that all versions perform very similarly, only one more benchmark was solved by the versions with dynamic shared object detection (the remaining solved benchmarks were the same for all versions). The number of solved benchmarks remains the same regardless of used store buffer bound.

Table 3 offers more detailed look at the 26 benchmarks solved by all versions of DIVINE. It shows the aggregate differences in state space sizes and solving times. It can be seen that the dynamic shared object detection improves performance significantly. Interestingly, we can see that of the 3 versions which differ only in store buffer size ("+sd", "+sdu", and "+sd4"), the unbounded version performs the best. We expect this to be caused by the nondeterminism in flushing the excessive entries out of the store buffer when the bound is reached – this can trigger flushing of matching entries from other store buffers and therefore increase nondeterminism.

5 Related Work

There are numerous techniques for analysis of programs with respect to relaxed memory.

Verification of Absence of SC Violations For these methods, the question is whether a program, when running under a relaxed memory model, exhibits any runs not possible under sequential consistency. This problem is explored under many names, e.g. (TSO-)safety [12], robustness [11, 16], stability [4], and monitoring of sequential consistency [13]. A similar techniques are used in [38] to detect data races in Java programs. A related problem of correspondence between a parallel and sequential implementation of a data structure is explored in [29]. Some of these techniques can also be used to insert memory fences into the programs to recover sequential consistency.

Neither of these techniques is directly comparable to our method. For these techniques, a program is incorrect if it exhibits relaxed behavior, while for us, it is incorrect if it violates specification (e.g., assertion safety and memory safety). In practice, the appearance of relaxed behavior is often not a problem, provided the overall behavior of the data structure or algorithm matches desired specification. In many lock-free data structures, a relaxed behavior is essential to achieving high performance.

Direct Analysis Techniques There are multiple methods for analysis of relaxed memory models based on program transformation. In [3] a transformation-based technique for the x86, POWER, and ARM memory models is presented. Another approach to program transformation is taken in [7], in this case, the transformation uses context switch bounding but not buffer bounding, and it uses additional copies for shared variables for TSO simulation. In [2] the context-bounded analysis using transformation is applied to the POWER memory model. Our work in [40] presents a transformation of LLVM bitcode to simulate buffer-bounded x86-TSO runs; compared to this work it has significantly less efficient implementation of the x86-TSO simulation.

A stateless model checking [20] approach to the analysis of programs running under the C++11 memory model (except for the release-consume synchronization) is presented in [28]. In [39] the authors focus mostly on modeling of TSO and PSO and its interplay with dynamic partial order reduction (DPOR, [18]). They combine modeling of thread scheduling nondeterminism and memory model nondeterminism using store buffers to a common framework by adding shadow thread for each store buffer which is responsible for flushing contents of this buffer to the memory. Another approach to combining TSO and PSO analysis with stateless model checking is presented in [1]. The advantage of this approach is that for a program without relaxed behavior it should produce no additional traces compared to sequential consistency. Another approach to stateless model checking is taken in [23], which uses execution graphs to explore all behavior of a C/C++ program under a modified C++11 memory model without exploring its interleaving directly.

So far, all of the described techniques used some kind of bounding to achieve efficiency – either bounding number of reordered operations, number of context switches, or number of iterations of loops. An unbounded approach to verification of programs under TSO is presented in [26]. It uses store buffers represented by automata and leverages cycle iteration acceleration to get a representation of store buffers on paths which would form cycles if values in store buffers were disregarded. It does not, however, target any real-world programming language. Instead, it targets a modified Promela language [21]. Another unbounded approach is presented in [10] – it introduces TSO behaviors lazily by iterative refinement, and while it is not complete, it should eventually find all errors.

Other Methods In [30], the SPARC hierarchy of memory models (TSO, PSO, RMO) is modeled using encoding from assembly to Murφ [17]. In [22] an explicit state model checker for C# programs (supporting subset of C#/.NET bytecode) which uses the .NET memory model is presented. The verifier first verifies program under SC and then it explores additional runs allowed under the .NET memory model. The implementation of the exploration algorithm uses a list of delayed instructions to implement instruction reordering. The work [15] presents verification of (potentially infinite state space) programs under TSO and PSO (with bounded store buffers) using predicate abstraction.

A completely different approach is taken in [36]. This work introduces a separation logic GPS, which allows proving properties about programs using (a fragment of) the C11 memory model. That is, this work is intended for manual proving of properties of parallel programs, not for automatic verification. The memory model is not complete; it lacks relaxed and consume-release accesses. Another fragment of the C11 memory model is targeted by the RSL separation logic introduced in [37].

6 Conclusion

We showed that by careful design of simulation of relaxed memory behaviour we can use the standard model checker supporting only the sequential consistency to efficiently detect relaxed memory errors in programs that are otherwise correct under sequentially consistent memory. Moreover, according to our experimental evaluation, our explicit-state model checking approach outperforms a state-of-the-art stateless model checker as well as bounded model checker, which is actually quite an unexpected result. We also show that many of the used benchmarks can be solved only by one or two of the three evaluated tools, which highlights the importance of employing different approaches to analysis of programs under relaxed memory. Finally, we show that for terminating programs, our approach is viable both with bounded and unbounded store buffer size.

References

1. Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless Model Checking for TSO and PSO. In *TACAS*, pages 353–367, Berlin, Heidelberg, 2015. Springer.
2. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. Context-bounded analysis for power. In *TACAS*, pages 56–74, Berlin, Heidelberg, 2017. Springer.

3. Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software Verification for Weak Memory via Program Transformation. In *ESOP*, pages 512–532, Berlin, Heidelberg, 2013. Springer.
4. Jade Alglave and Luc Maranget. Stability in Weak Memory Models. In *CAV*, pages 50–66, Berlin, Heidelberg, 2011. Springer.
5. Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Transactions on Programming Languages and Systems*, 36(2):7:1–7:74, July 2014.
6. Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the Verification Problem for Weak Memory Models. In *POPL*, pages 7–18, New York, NY, USA, 2010. ACM.
7. Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. Getting rid of store-buffers in tso analysis. In *CAV*, pages 99–115, Berlin, Heidelberg, 2011. Springer.
8. Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. Model Checking of C and C++ with DIVINE 4. In *ATVA*, volume 10482 of *LNCS*. Springer, 2017.
9. Dirk Beyer. *Software Verification with Validation of Results*, pages 331–349. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017.
10. Ahmed Bouajjani, Georgel Calin, Egor Derevenetc, and Roland Meyer. Lazy TSO Reachability. In *FASE*, pages 267–282, Berlin, Heidelberg, 2015. Springer.
11. Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and Enforcing Robustness against TSO. In *ESOP*, pages 533–553, Berlin, Heidelberg, 2013. Springer.
12. Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In *CAV*, pages 107–120, Berlin, Heidelberg, 2008. Springer.
13. Jabob Burnim, Koushik Sen, and Christos Stergiou. Sound and Complete Monitoring of Sequential Consistency for Relaxed Memory Models. In *TACAS*, pages 11–25, Berlin, Heidelberg, 2011. Springer.
14. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *TACAS*, pages 168–176, Berlin, Heidelberg, 2004. Springer.
15. Andrei Marian Dan, Yuri Meshman, Martin Vechev, and Eran Yahav. Predicate Abstraction for Relaxed Memory Models. In *SAS*, pages 84–104, Berlin, Heidelberg, 2013. Springer.
16. Egor Derevenetc and Roland Meyer. Robustness against Power is PSpace-complete. In *ICAPL*, pages 158–170, Berlin, Heidelberg, 2014. Springer.
17. David L. Dill. The Murphi Verification System. In *CAV*, pages 390–393, London, UK, UK, 1996. Springer-Verlag.
18. Cormac Flanagan and Patrice Godefroid. Dynamic Partial-order Reduction for Model Checking Software. In *POPL*, pages 110–121, New York, NY, USA, 2005. ACM.
19. Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *POPL*, pages 608–621, New York, NY, USA, 2016. ACM.
20. Patrice Godefroid. Model Checking for Programming Languages Using VeriSoft. In *POPL*, pages 174–186, New York, NY, USA, 1997. ACM.
21. Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 5 1997.

22. Thuan Quang Huynh and Abhik Roychoudhury. A Memory Model Sensitive Checker for C#. In *FM*, pages 476–491, Berlin, Heidelberg, 2006. Springer.
23. Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective Stateless Model Checking for C/C++ Concurrency. *Proceedings of the ACM on Programming Languages*, 2:17:1–17:32, December 2017.
24. Daniel Kroening and Michael Tautschnig. CBMC – C Bounded Model Checker. In *TACAS*, pages 389–391, Berlin, Heidelberg, 2014. Springer.
25. Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.
26. Alexander Linden and Pierre Wolper. An Automata-Based Symbolic Approach for Verifying Programs on Relaxed Memory Models. In *SPIN*, pages 212–226, Berlin, Heidelberg, 2010. Springer.
27. Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An Axiomatic Memory Model for POWER Multiprocessors. In *CAV*, pages 495–512, Berlin, Heidelberg, 2012. Springer.
28. Brian Norris and Brian Demsky. CDSchecker: Checking Concurrent Data Structures Written with C/C++ Atomics. In *OOPSLA*, pages 131–150, New York, NY, USA, 2013. ACM.
29. Peizhao Ou and Brian Demsky. Checking Concurrent Data Structures Under the C/C++11 Memory Model. *SIGPLAN*, 52(8):45–59, January 2017.
30. Seungjoon Park and David L. Dill. An Executable Specification, Analyzer and Verifier for RMO (Relaxed Memory Order). In *SPAA*, pages 34–41, New York, NY, USA, 1995. ACM.
31. Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. *Proceedings of the ACM on Programming Languages*, 2:19:1–19:29, December 2017.
32. Petr Ročkai, Vladimír Štill, Ivana Černá, and Jiří Barnat. DiVM: Model Checking with LLVM and Graph Memory. *Journal of Systems and Software*, 2018.
33. Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER Multiprocessors. In *PLDI*, pages 175–186, New York, NY, USA, 2011. ACM.
34. Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.
35. CORPORATE SPARC International, Inc. *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
36. Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. In *OOPSLA*, pages 691–707, New York, NY, USA, 2014. ACM.
37. Viktor Vafeiadis and Chinmay Narayan. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In *OOPSLA*, pages 867–884, New York, NY, USA, 2013. ACM.
38. Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Memory-Model-Sensitive Data Race Analysis. In *ICFEM*, pages 30–45, Berlin, Heidelberg, 2004. Springer.
39. Naling Zhang, Markus Kusano, and Chao Wang. Dynamic Partial Order Reduction for Relaxed Memory Models. In *PLDI*, pages 250–259, New York, NY, USA, 2015. ACM.

40. Vladimír Štill, Petr Ročkai, and Jiří Barnat. Weak Memory Models as LLVM-to-LLVM Transformations. In *MEMICS*, volume 9548 of *LNCS*, pages 144–155. Springer International Publishing, 2016.
41. Vladimír Štill, Petr Ročkai, and Jiří Barnat. Using Off-the-Shelf Exception Support Components in C++ Verification. In *QRS*, pages 54–64. IEEE, 7 2017.