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

Vladimír Štill and Jiří Barnat

Faculty of Informatics, Masaryk University

Relaxed Memory

Almost all recent processors exhibit relaxed memory behavior, which, together with the rise of multicore processors and parallel programs means that programmers have to deal with its complexities. Even on x86 processors, which have stronger memory model than most other architectures, programmers often have to decide whether to play safe with higher level synchronization constructs such as mutexes, or tap to the full power of the architecture and risk subtle unintuitive behavior of relaxed memory accesses.

Here, we present an extension of the DIVINE model checker which allows for analysis of C and C++ programs under the x86-TSO relaxed memory model. This means that DIVINE can now validate parallel programs intended to run on x86 processors more precisely. The novelty of our approach is in a careful design of an encoding of x86-TSO operations so that nondeterminism is minimized, and therefore the performance of analysis is increased. 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.

The x86-TSO Memory Model

```
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;
}
```

is $a = 0 \wedge b = 0$ reachable?

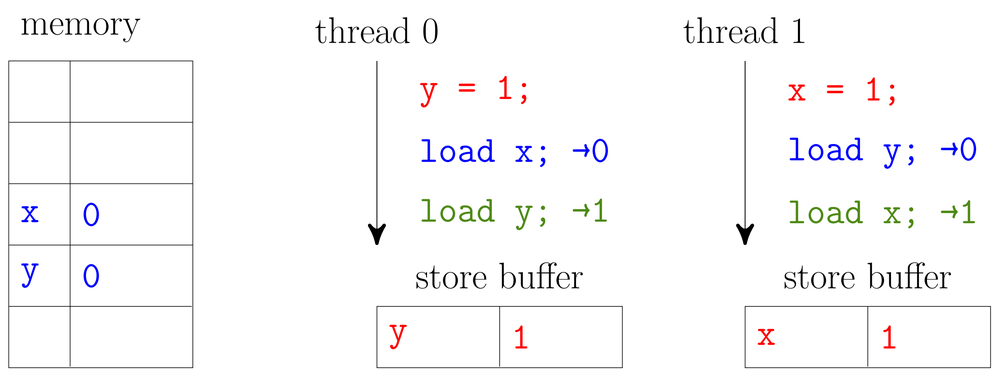


Figure 1: A demonstration of the x86-TSO memory model. The thread 0 stores 1 to variable `y` and then loads variables `x` and `y`. The thread 1 stores 1 to `x` and then loads `y` and `x`. Intuitively, we would expect it to be impossible for $a = 0$ and $b = 0$ to both be true at the end of the execution, as there is no interleaving of thread actions which would produce such a result. However, under x86-TSO, the stores are cached in thread-local 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).

Results

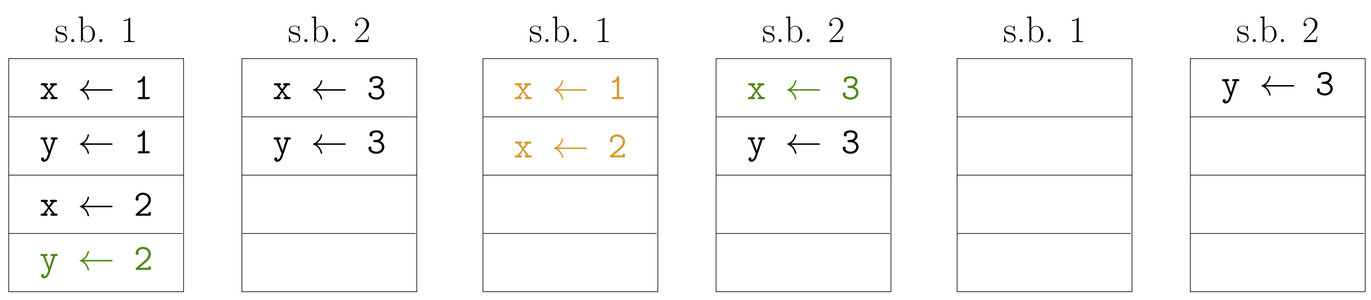
We compare our approach with state-of-the-art bounded model checker CBMC and stateless model checker Nidhugg and show that extended DIVINE competes well with both of them. The evaluation is performed on SV-COMP concurrency benchmarks without data nondeterminism. Of these benchmarks, DIVINE can solve more instances and discover more bugs than both of the other tools. The following table shows how many benchmarks were completed by each of the tools, how many of these contained errors under x86-TSO, but not without relaxed memory, and how many benchmarks were only solved by the given tool and not the other two tools.

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

https://divine.fi.muni.cz/2018/x86tso/

Delayed Flushing

x86-TSO is a Total Store Order memory model, which means store buffers maintain execution order of stores. The main novelty of our approach is *delayed flushing* which allows us to remove entries from the store buffer in order in which the corresponding values are read instead of the store order. This improvement limits nodeterminism by delaying it as long as possible. The reduced nondeterminism translates to better performance of verification using an explicit-state model checker.




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

Figure 2: 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;
}
```

Figure 3: 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
}
```

Figure 4: 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.