

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

---

Vladimír Štill    Jiří Barnat



Masaryk University  
Brno, Czech Republic

13th November 2018



- caches and inter-core communication in modern CPUs
- out-of-order execution
- has observable effects in multithreaded environments
- can cause nasty and hard to detect bugs

- caches and inter-core communication in modern CPUs
- out-of-order execution
- has observable effects in multithreaded environments
- can cause nasty and hard to detect bugs
- relaxed memory will probably be here to haunt us for a long time





# Analysis of Parallel Programs under Relaxed Memory

- designing fast correct parallel programs is hard
- formal analysis can help



- designing fast correct parallel programs is hard
- formal analysis can help

## Current Approaches

- many different ways of finding errors
  - stateless model checking (Nidhugg, RCMC, ...)
  - bounded model checking (CBMC, ...)
- also techniques which consider any relaxed behaviour as error



- designing fast correct parallel programs is hard
- formal analysis can help

## Current Approaches

- many different ways of finding errors
  - stateless model checking (Nidhugg, RCMC, ...)
  - bounded model checking (CBMC, ...)
- also techniques which consider any relaxed behaviour as error

## Our Focus

- x86-TSO memory model of the Intel and AMD CPUs (x86-TSO)
- C and C++
- based on explicit-state model checking



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

void thread1()
{
    x = 1;
    int b = y;
    int d = x;
}
```

- every write to the memory can be delayed
- writes are not reordered with each other but a CPU will not see writes done by other CPUs instantly

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

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

void thread0()
{
    y = 1;
    int a = x;
    int c = y;
}

void thread1()
{
    x = 1;
    int b = y;
    int d = x;
}
```

- every write to the memory can be delayed
- writes are not reordered with each other but a CPU will not see writes done by other CPUs instantly

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

shared memory

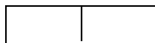
x	0
y	0

thread 0

y = 1;



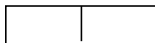
store buffer



thread 1



store buffer





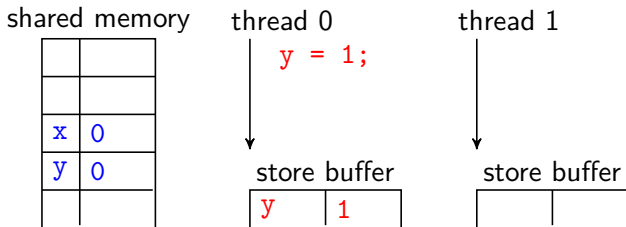
```
int x = 0;
int y = 0;

void thread0()
{
    y = 1;
    int a = x;
    int c = y;
}

void thread1()
{
    x = 1;
    int b = y;
    int d = x;
}
```

- every write to the memory can be delayed
- writes are not reordered with each other but a CPU will not see writes done by other CPUs instantly

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



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

void thread0()
{
    y = 1;
    int a = x;
    int c = y;
}

void thread1()
{
    x = 1;
    int b = y;
    int d = x;
}
```

- every write to the memory can be delayed
- writes are not reordered with each other but a CPU will not see writes done by other CPUs instantly

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

shared memory

x	0
y	0

thread 0

y = 1;  
load x;

store buffer

y	1
---	---

thread 1

store buffer

--	--

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

void thread0()
{
    y = 1;
    int a = x;
    int c = y;
}

void thread1()
{
    x = 1;
    int b = y;
    int d = x;
}
```

- every write to the memory can be delayed
- writes are not reordered with each other but a CPU will not see writes done by other CPUs instantly

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

shared memory

x	0
y	0

thread 0

y = 1;  
load x; → 0

store buffer

y	1
---	---

thread 1

store buffer

--	--

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

void thread0()
{
    y = 1;
    int a = x;
    int c = y;
}
```

```
void thread1()
{
    x = 1;
    int b = y;
    int d = x;
}
```

- every write to the memory can be delayed
- writes are not reordered with each other but a CPU will not see writes done by other CPUs instantly

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

shared memory

x	0
y	0

thread 0

y = 1;  
load x; → 0  
load y;

store buffer

y	1
---	---

thread 1

store buffer

--	--

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

void thread0()
{
    y = 1;
    int a = x;
    int c = y;
}
```

```
void thread1()
{
    x = 1;
    int b = y;
    int d = x;
}
```

- every write to the memory can be delayed
- writes are not reordered with each other but a CPU will not see writes done by other CPUs instantly

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

shared memory

x	0
y	0

thread 0

y = 1;  
load x; →0  
load y; →1

store buffer

y	1
---	---

thread 1

store buffer

--	--

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

void thread0()
{
    y = 1;
    int a = x;
    int c = y;
}
```

```
void thread1()
{
    x = 1;
    int b = y;
    int d = x;
}
```

- every write to the memory can be delayed
- writes are not reordered with each other but a CPU will not see writes done by other CPUs instantly

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

shared memory

x	0
y	0

thread 0

y = 1;  
load x; →0  
load y; →1

store buffer

y	1
---	---

thread 1

x = 1;

store buffer

--	--

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

void thread0()
{
    y = 1;
    int a = x;
    int c = y;
}

void thread1()
{
    x = 1;
    int b = y;
    int d = x;
}
```

- every write to the memory can be delayed
- writes are not reordered with each other but a CPU will not see writes done by other CPUs instantly

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

shared memory

x	0
y	0

thread 0

y = 1;  
load x; →0  
load y; →1

store buffer

y	1
---	---

thread 1

x = 1;

store buffer

x	1
---	---

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

void thread0()
{
    y = 1;
    int a = x;
    int c = y;
}

void thread1()
{
    x = 1;
    int b = y;
    int d = x;
}
```

- every write to the memory can be delayed
- writes are not reordered with each other but a CPU will not see writes done by other CPUs instantly

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

shared memory

x	0
y	0

thread 0

y = 1;  
load x; →0  
load y; →1

store buffer

y	1
---	---

thread 1

x = 1;  
load y;

store buffer

x	1
---	---



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

void thread0()
{
    y = 1;
    int a = x;
    int c = y;
}

void thread1()
{
    x = 1;
    int b = y;
    int d = x;
}
```

- every write to the memory can be delayed
- writes are not reordered with each other but a CPU will not see writes done by other CPUs instantly

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

shared memory

x	0
y	0

thread 0

y = 1;  
load x; →0  
load y; →1

store buffer

y	1
---	---

thread 1

x = 1;  
load y; →0

store buffer

x	1
---	---

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

void thread0()
{
    y = 1;
    int a = x;
    int c = y;
}

void thread1()
{
    x = 1;
    int b = y;
    int d = x;
}
```

- every write to the memory can be delayed
- writes are not reordered with each other but a CPU will not see writes done by other CPUs instantly

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

shared memory

x	0
y	0

thread 0

y = 1;  
load x; →0  
load y; →1

store buffer

y	1
---	---

thread 1

x = 1;  
load y; →0  
load x;

store buffer

x	1
---	---

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

void thread0()
{
    y = 1;
    int a = x;
    int c = y;
}

void thread1()
{
    x = 1;
    int b = y;
    int d = x;
}
```

- every write to the memory can be delayed
- writes are not reordered with each other but a CPU will not see writes done by other CPUs instantly

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

shared memory

x	0
y	0

thread 0

y = 1;  
load x; →0  
load y; →1

store buffer

y	1
---	---

thread 1

x = 1;  
load y; →0  
load x; →1

store buffer

x	1
---	---



- x86-TSO allows arbitrary delaying of stores



- x86-TSO allows arbitrary delaying of stores
- but only some possibilities are meaningful and observable



- x86-TSO allows arbitrary delaying of stores
- but only some possibilities are meaningful and observable
- consider flushing the store buffer only if someone reads buffered values



- x86-TSO allows arbitrary delaying of stores
- but only some possibilities are meaningful and observable
- consider flushing the store buffer only if someone reads buffered values
- still, store buffer can be large → many ways to flush it



- x86-TSO allows arbitrary delaying of stores
- but only some possibilities are meaningful and observable
- consider flushing the store buffer only if someone reads buffered values
- still, store buffer can be large  $\rightarrow$  many ways to flush it
- problem: store buffer keeps order





- x86-TSO allows arbitrary delaying of stores
- but only some possibilities are meaningful and observable
- consider flushing the store buffer only if someone reads buffered values
- still, store buffer can be large → many ways to flush it
- problem: store buffer keeps order
- solution: *delayed flushing* – some values can behave as flushed, but their order relative to other buffered values can be undetermined

s.b. 1	s.b. 2
x $\leftarrow$ 1	x $\leftarrow$ 3
y $\leftarrow$ 1	y $\leftarrow$ 3
x $\leftarrow$ 2	
y $\leftarrow$ 2	

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

- three threads, thread 0 executing

s.b. 1	s.b. 2
x $\leftarrow$ 1	x $\leftarrow$ 3
y $\leftarrow$ 1	y $\leftarrow$ 3
x $\leftarrow$ 2	
y $\leftarrow$ 2	

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

- nondeterministically choose which value to load from the store buffers

s.b. 1	s.b. 2
x ← 1	x ← 3
x ← 2	y ← 3

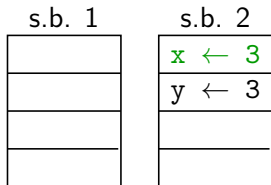
```
void thread0() {  
    int a = y; // →2  
    int b = x;  
}
```

- older values for y in s.b. 1 removed
- other older values in s.b. 1 marked as flushed

s.b. 1	s.b. 2
x ← 1	x ← 3
x ← 2	y ← 3

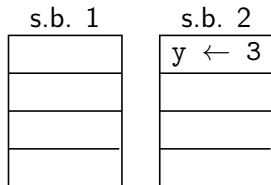
```
void thread0() {  
    int a = y; // →2  
    int b = x; // →3  
}
```

- nondeterministically choose to load value from s.b. 2



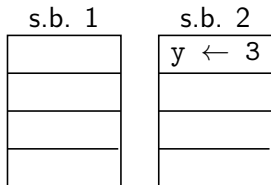
```
void thread0() {  
    int a = y; // →2  
    int b = x; // →3  
}
```

- flushed values for x has to be removed (overwritten)



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

- x is propagated to the memory



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

- $y \leftarrow 3$  stays in buffer unless  $y$  is read again



- evaluation on nonsymbolic SV-COMP concurrency benchmarks
- compared with CBMC and Nidhugg

(total = 54)	CBMC	<b>DIVINE</b>	Nidhugg
finished	20	23	24
TSO errors	3	10	3
uniquely solved	5	6	8



## Conclusion

- an efficient explicit-state approach to verification under the x86-TSO memory model



## Conclusion

- an efficient explicit-state approach to verification under the x86-TSO memory model
- relaxed behaviour encoded into the program



## Conclusion

- an efficient explicit-state approach to verification under the x86-TSO memory model
- relaxed behaviour encoded into the program
- works with C and C++ programs
- extendable to any LLVM-based language straight-forwardly



## Conclusion

- an efficient explicit-state approach to verification under the x86-TSO memory model
- relaxed behaviour encoded into the program
- works with C and C++ programs
- extendable to any LLVM-based language straight-forwardly
- analysis of non-terminating programs with finite state space



## Conclusion

- an efficient explicit-state approach to verification under the x86-TSO memory model
- relaxed behaviour encoded into the program
- works with C and C++ programs
- extendable to any LLVM-based language straight-forwardly
- analysis of non-terminating programs with finite state space

## Future Work

- combination with symbolic verification
- focus on lock-free programming



## Conclusion

- an efficient explicit-state approach to verification under the x86-TSO memory model
- relaxed behaviour encoded into the program
- works with C and C++ programs
- extendable to any LLVM-based language straight-forwardly
- analysis of non-terminating programs with finite state space

## Future Work

- combination with symbolic verification
- focus on lock-free programming

Thank You!