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

Vladimír Štill Jiří Barnat

Masaryk University

Brno, Czech Republic

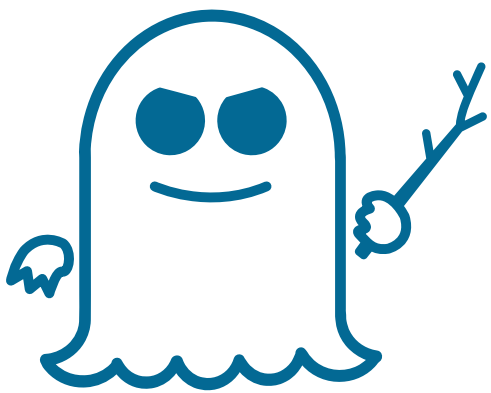
13th November 2018

Relaxed Memory

- 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

- 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



- 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

The x86-TSO Memory Model

```
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

$$\text{Is } a = 0 \wedge b = 0 \text{ reachable?}$$

The x86-TSO Memory Model

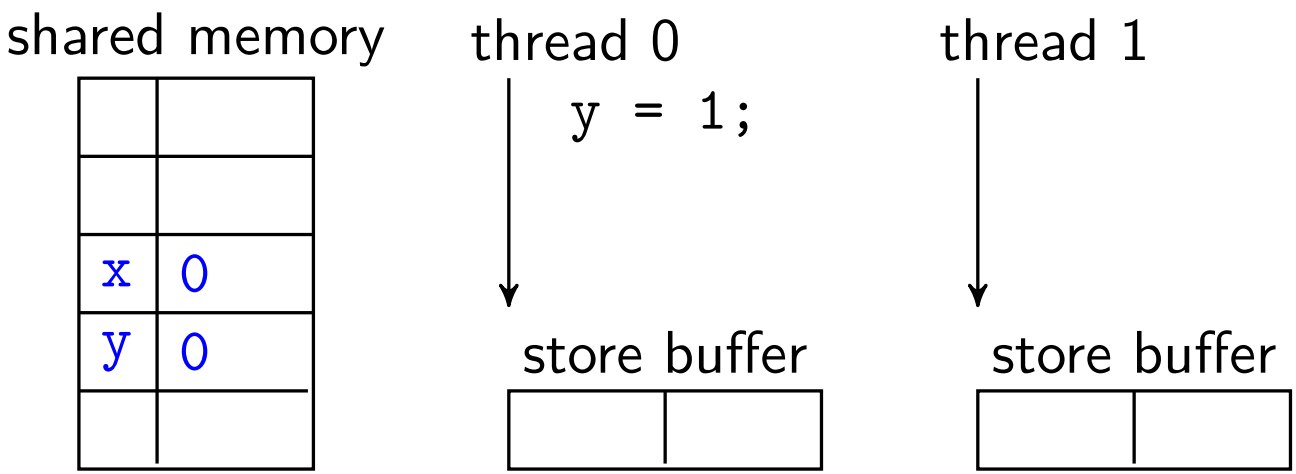
```
int x = 0;
int y = 0;

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?



The x86-TSO Memory Model

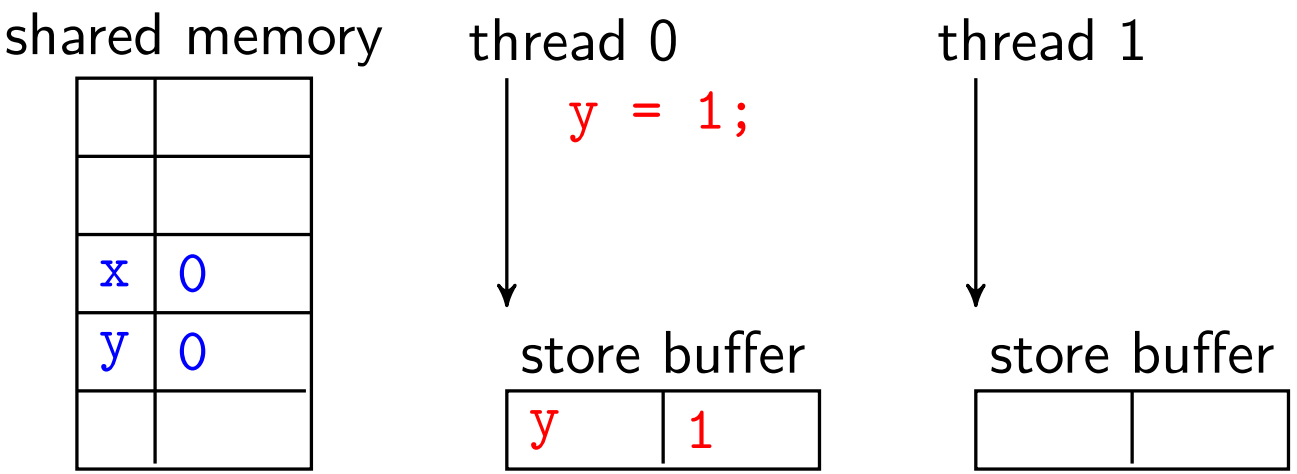
```
int x = 0;
int y = 0;

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?



The x86-TSO Memory Model

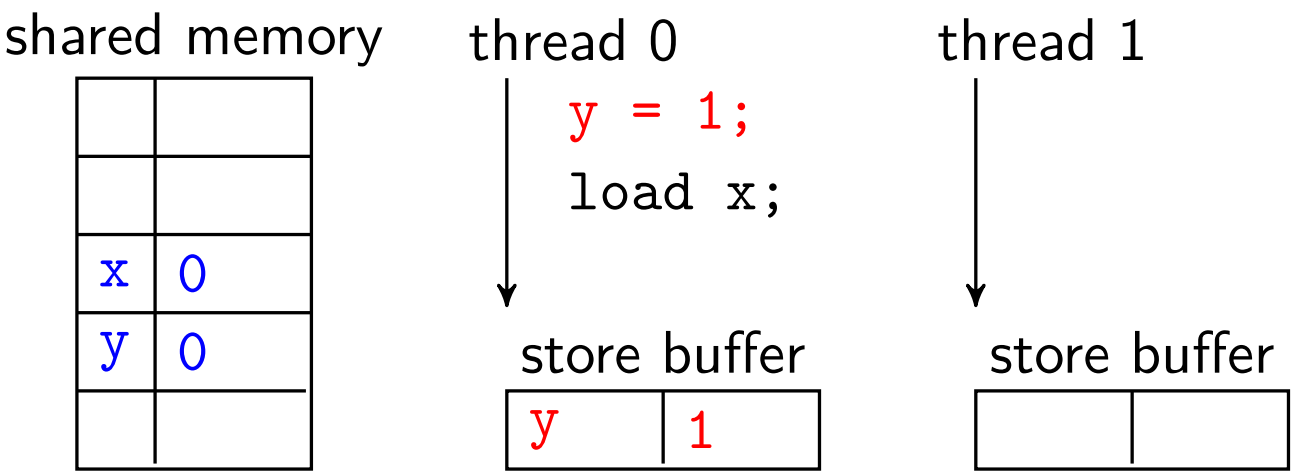
```
int x = 0;
int y = 0;

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?



The x86-TSO Memory Model

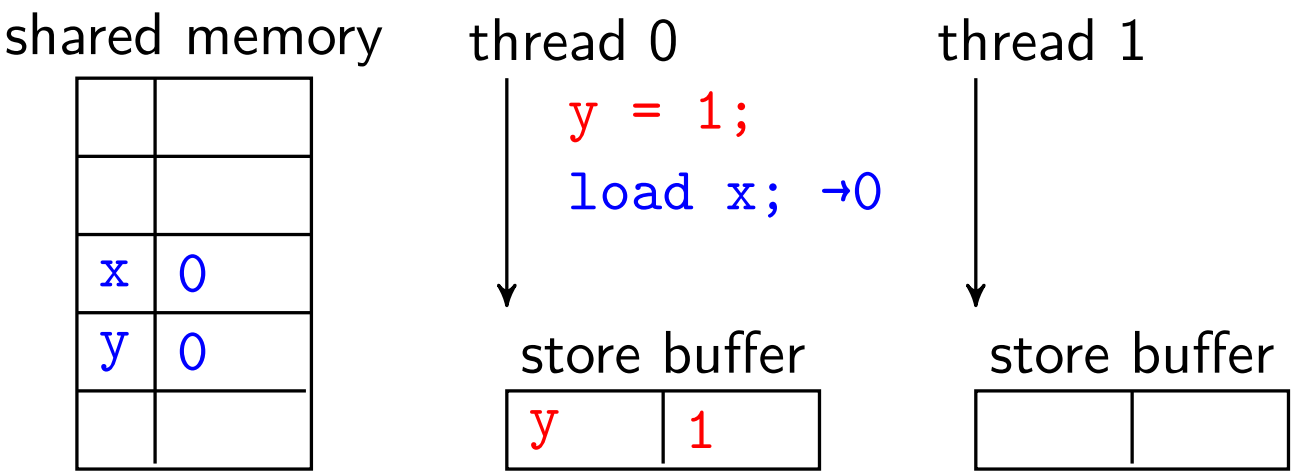
```
int x = 0;
int y = 0;

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?



The x86-TSO Memory Model

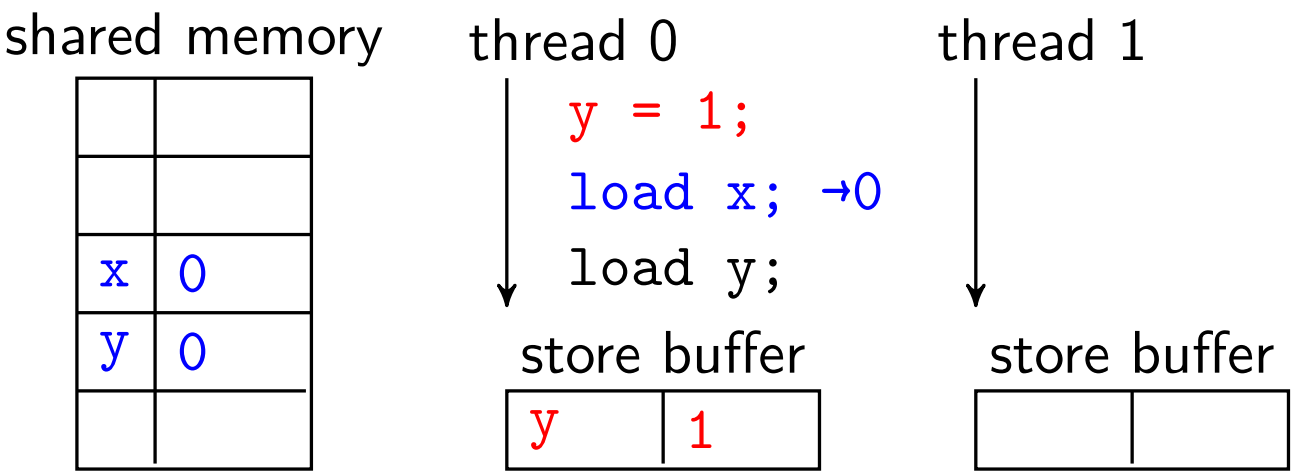
```
int x = 0;
int y = 0;

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?



The x86-TSO Memory Model

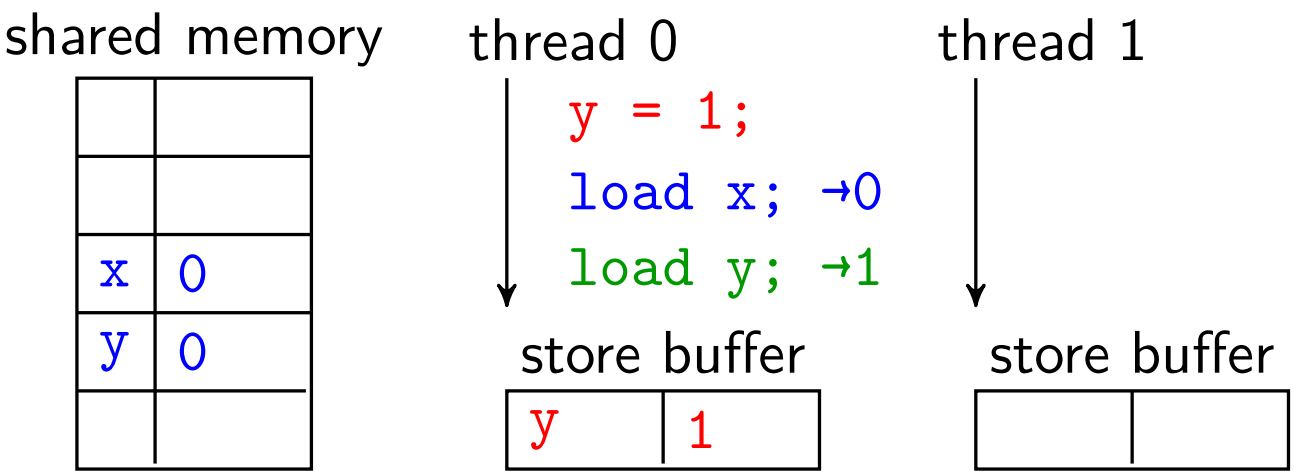
```
int x = 0;
int y = 0;

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?



The x86-TSO Memory Model

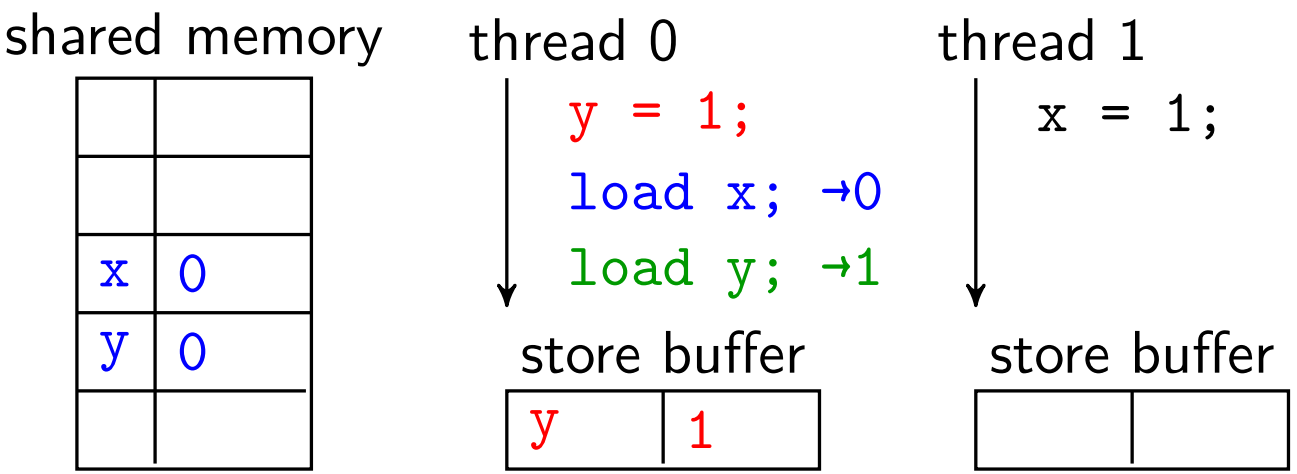
```
int x = 0;
int y = 0;

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?



The x86-TSO Memory Model

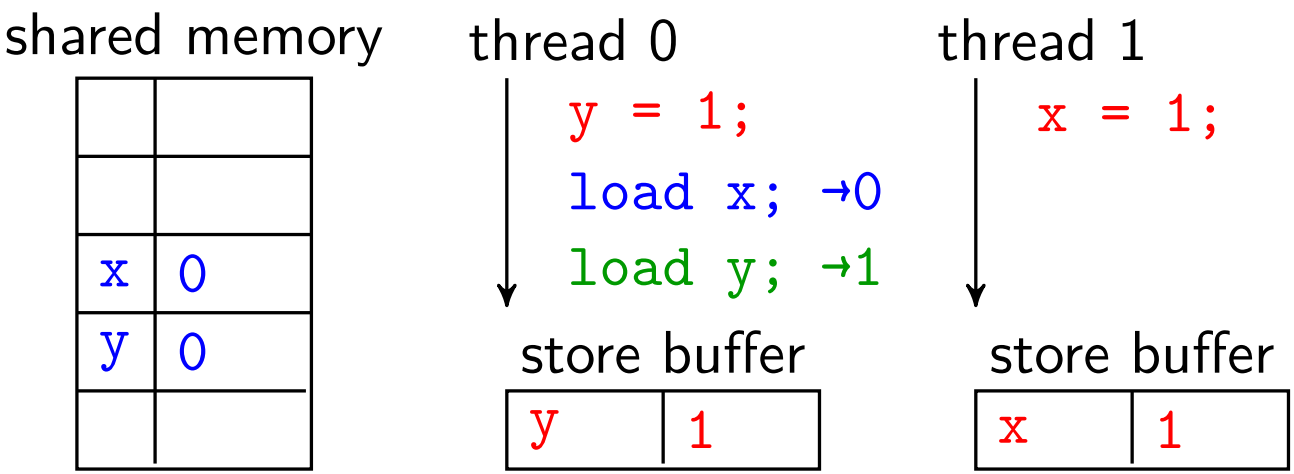
```
int x = 0;
int y = 0;

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?



The x86-TSO Memory Model

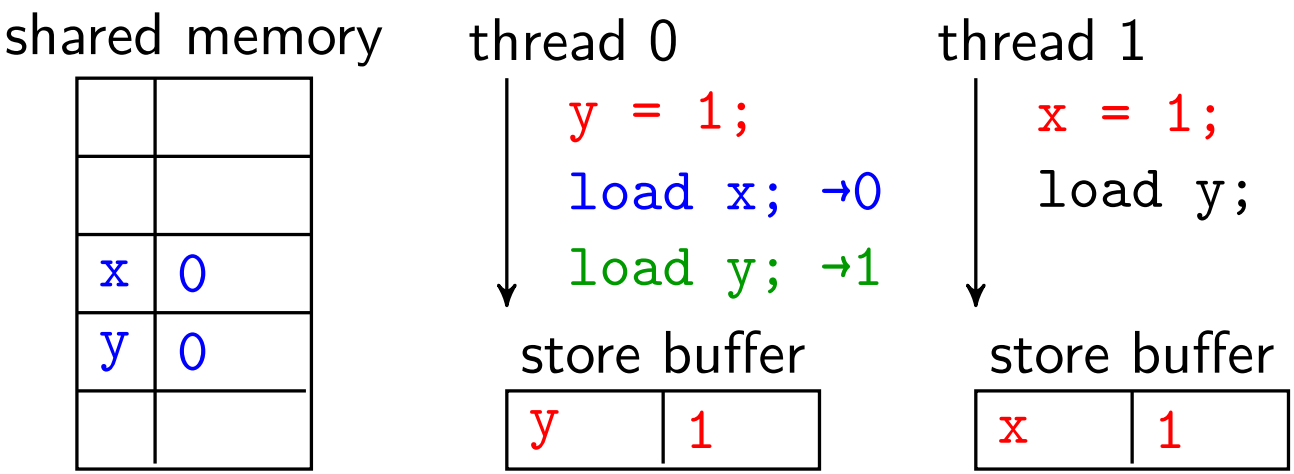
```
int x = 0;
int y = 0;

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?



The x86-TSO Memory Model

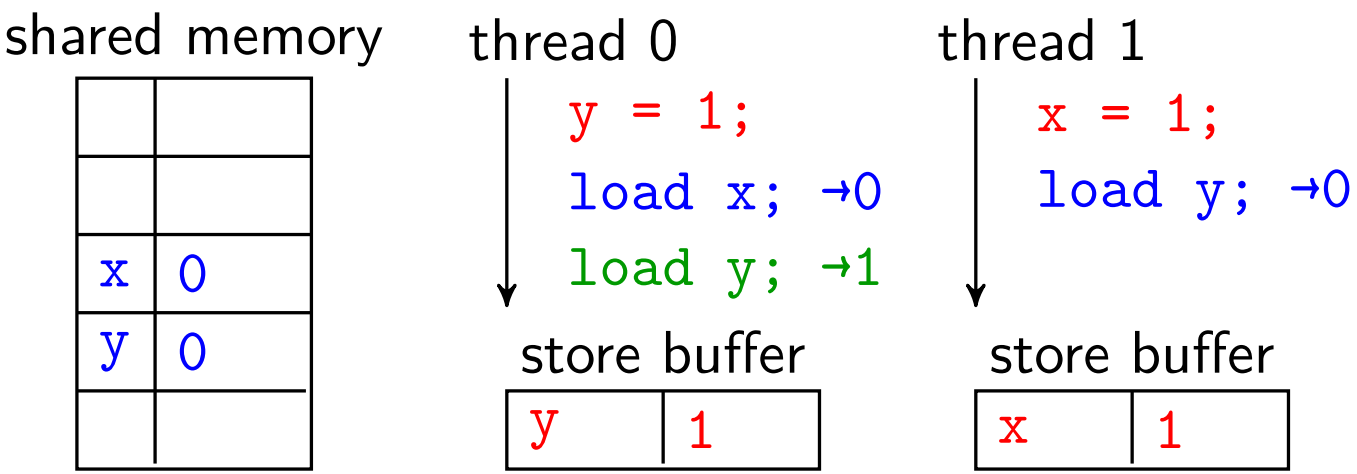
```
int x = 0;
int y = 0;

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?



The x86-TSO Memory Model

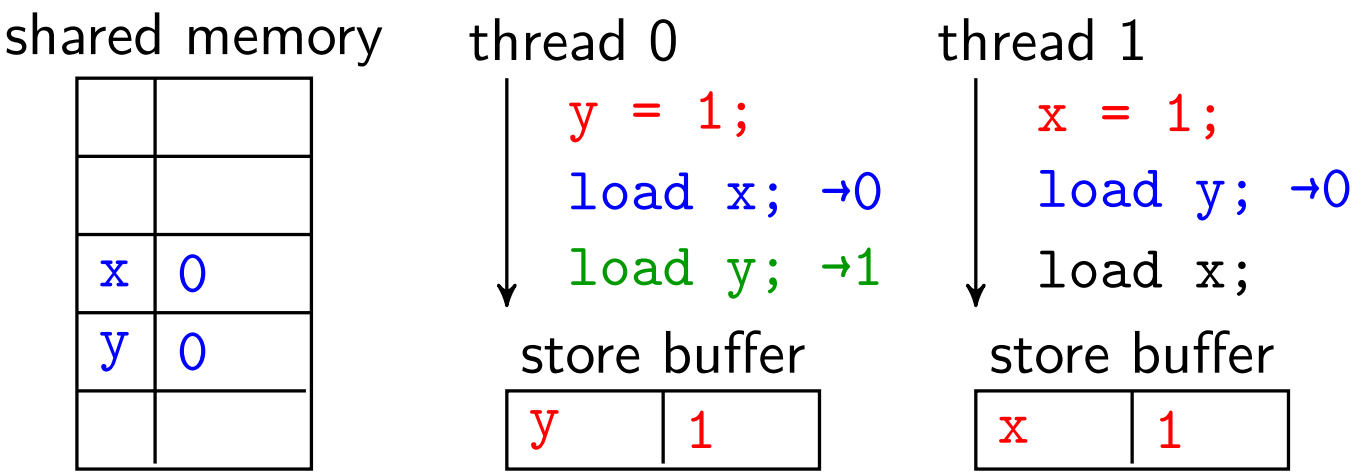
```
int x = 0;
int y = 0;

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?



The x86-TSO Memory Model

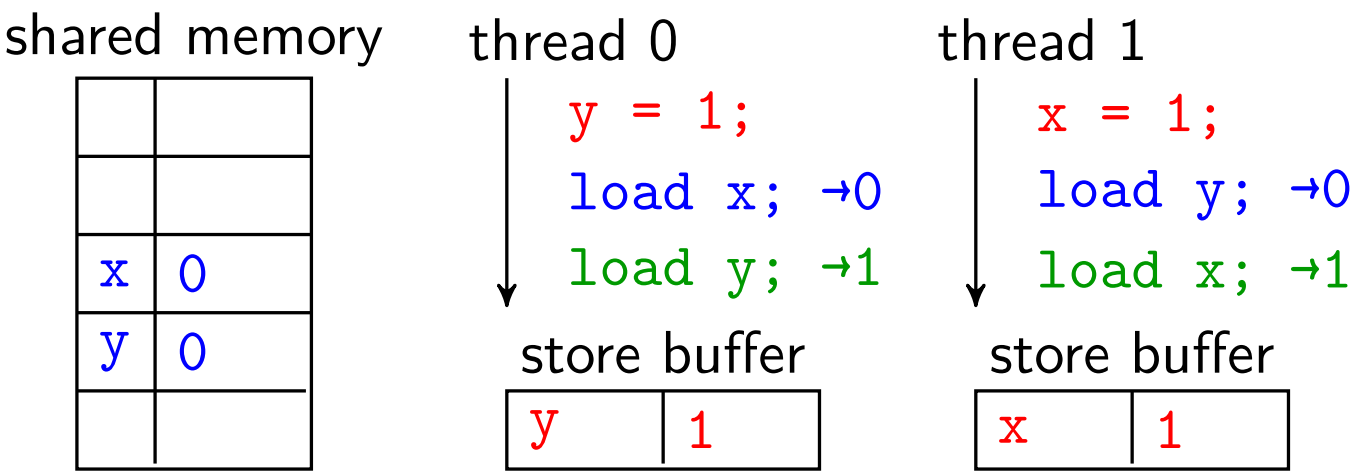
```
int x = 0;
int y = 0;

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?



Our Approach

- x86-TSO allows arbitrary delaying of stores

Our Approach

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

Our Approach

- 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

Our Approach

- 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

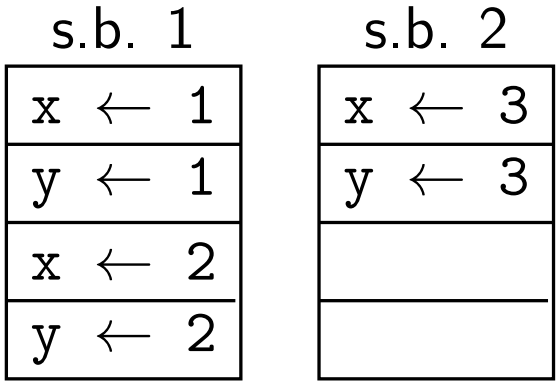
Our Approach

- 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

Our Approach

- 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
- solution: *delayed flushing* – some values can behave as flushed, but their order respective to other buffered values can be undetermined

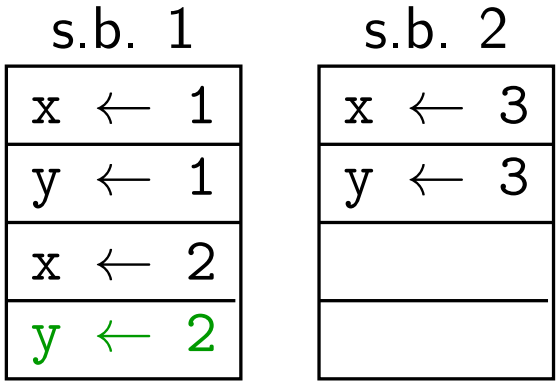
Delayed Flushing




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

- three threads, thread 0 executing

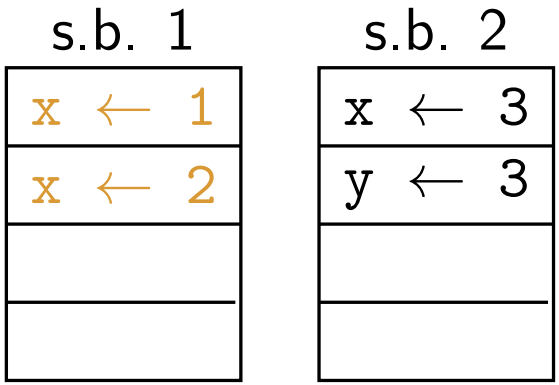
Delayed Flushing




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

- nondeterministically choose which value to load from the store buffers

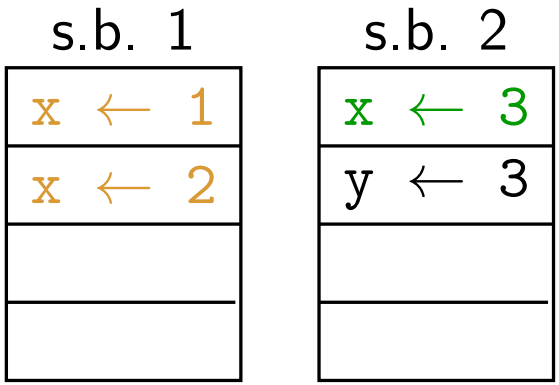
Delayed Flushing




```
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

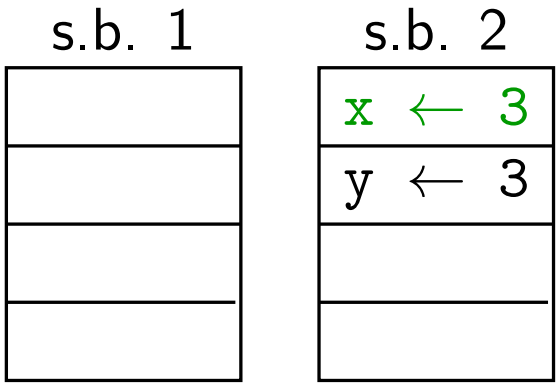
Delayed Flushing




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

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

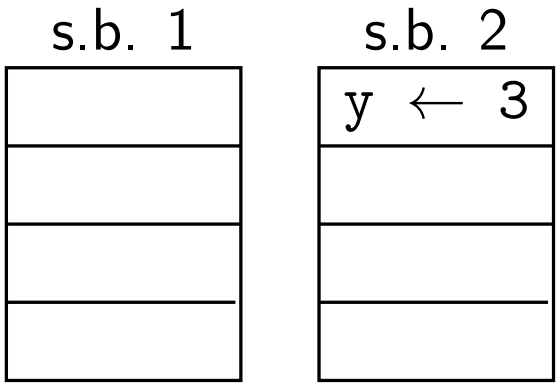
Delayed Flushing




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

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

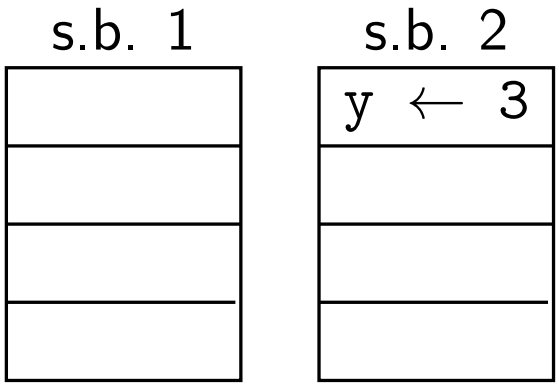
Delayed Flushing




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

- `x` is propagated to the memory

Delayed Flushing




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

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

Results

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

(total = 54)	CBMC	**DIVINE**	Nidhugg
finished	20	23	24
TSO errors	3	10	3
uniqly 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 & Future Work

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 & Future Work

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!