Lazy x86-TSO Memory Model for C++ Verification

Vladimír Štill

Masaryk University

Brno, Czech Republic

26th February 2018

Verification of Parallel Programs I

- design of parallel programs is hard
- easy to make mistakes – data races, deadlocks

Verification of Parallel Programs I

- design of parallel programs is hard
- easy to make mistakes – data races, deadlocks
- **memory behaviour is very complex**
  - effects of caches, out-of-order and speculative execution

Verification of Parallel Programs I

- design of parallel programs is hard
- easy to make mistakes – data races, deadlocks
- **memory behaviour is very complex**
  - effects of caches, out-of-order and speculative execution

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

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

- is it possible to end with `a == 0 && b == 0`?

Verification of Parallel Programs I

- design of parallel programs is hard
- easy to make mistakes – data races, deadlocks
- **memory behaviour is very complex**
  - effects of caches, out-of-order and speculative execution

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

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

- is it possible to end with `a == 0 && b == 0`? **yes**

Verification of Parallel Programs II

- C and C++

Verification of Parallel Programs II

- C and C++
- program is translated into LLVM intermediate language
- LLVM is executed by the model checker
- exploration of all possible runs of the program

Verification of Parallel Programs II

- C and C++
- program is translated into LLVM intermediate language
- LLVM is executed by the model checker
- exploration of ~~all possible~~ runs of the program
  - actually of some representants of classes of equivalent runs

Verification of Parallel Programs II

- C and C++
- program is translated into LLVM intermediate language
- LLVM is executed by the model checker
- exploration of ~~all possible~~ runs of the program
  - actually of some representants of classes of equivalent runs
- detect assertions, memory errors, compiler traps, ...

Relaxed Memory Example

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

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

Relaxed Memory Example

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

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

memory

x	0
y	0

thread 0

```
y = 1;
load x;
```

store buffer of t. 0

thread 1

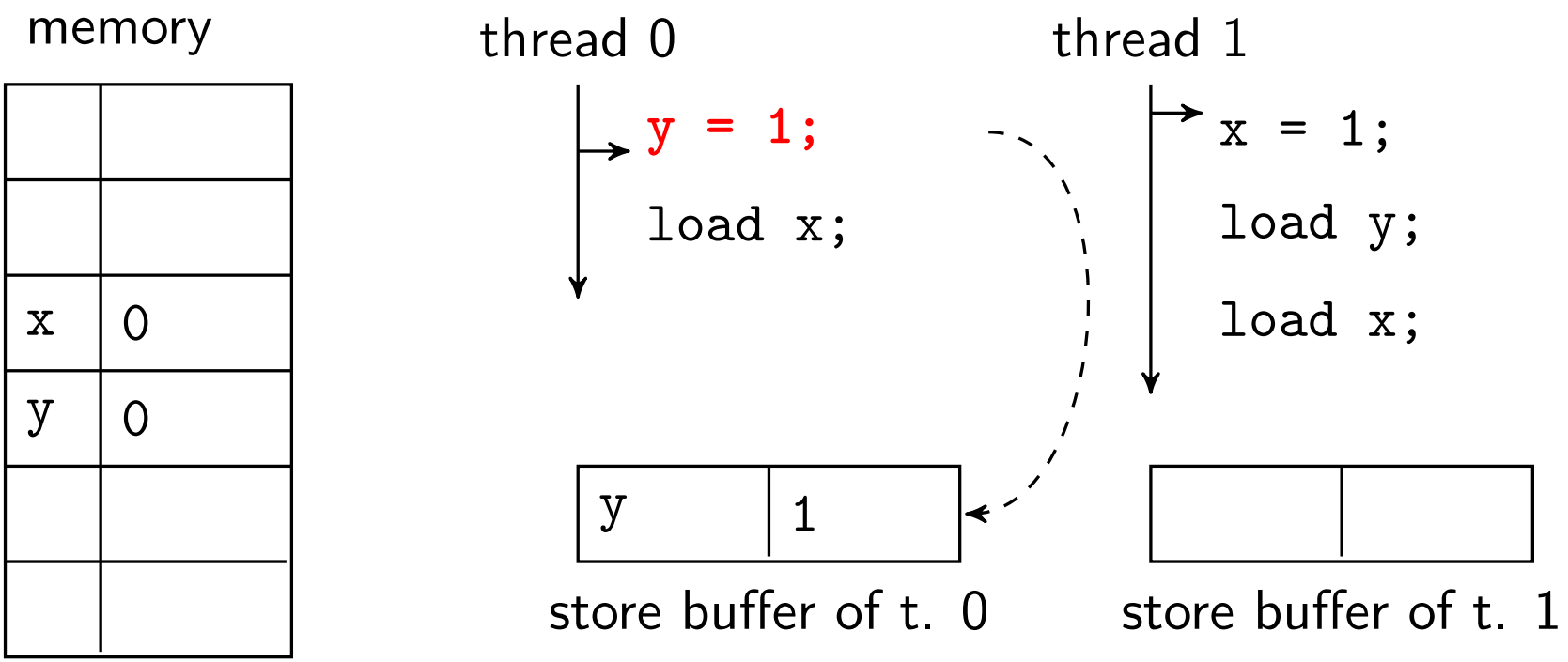
```
x = 1;
load y;
load x;
```

store buffer of t. 1

Relaxed Memory Example

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

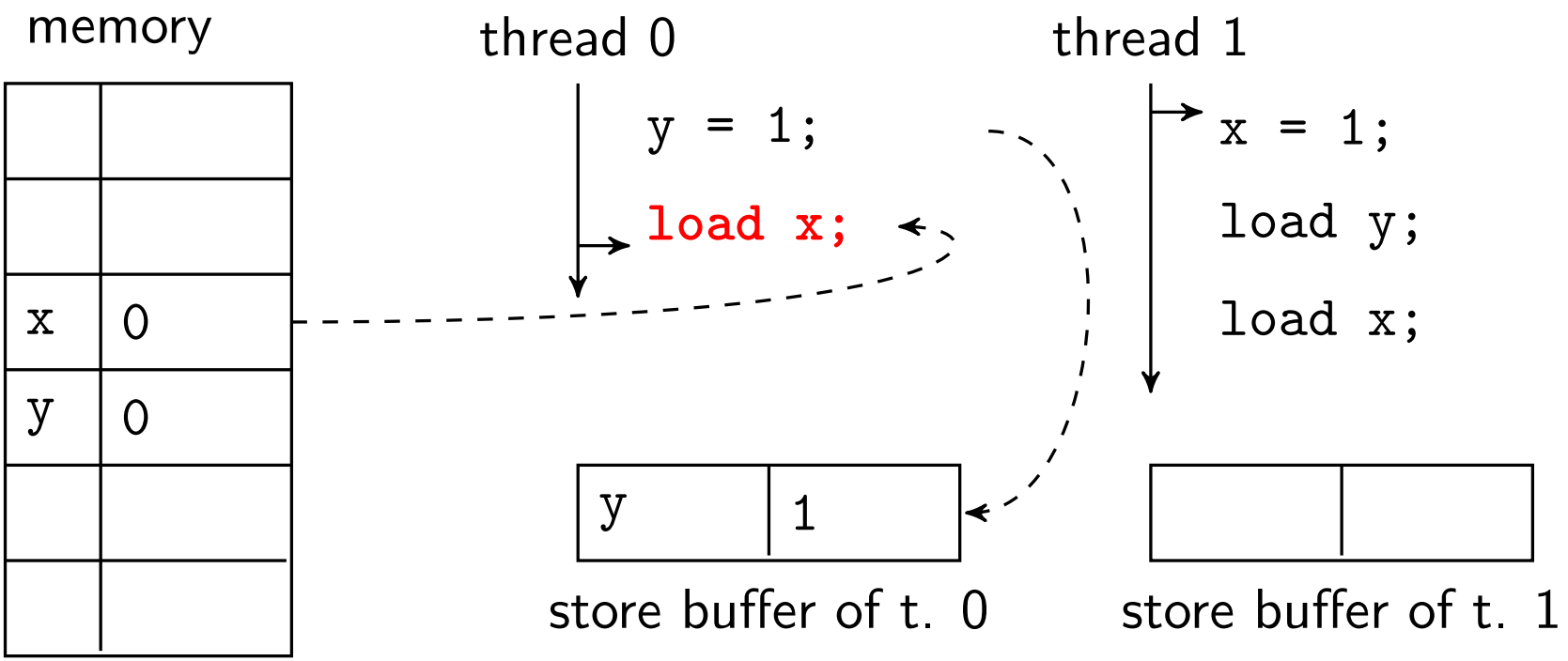
```
void thread1() {
    x = 1;
    int b = y;
    int c = x;
}
```



Relaxed Memory Example

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

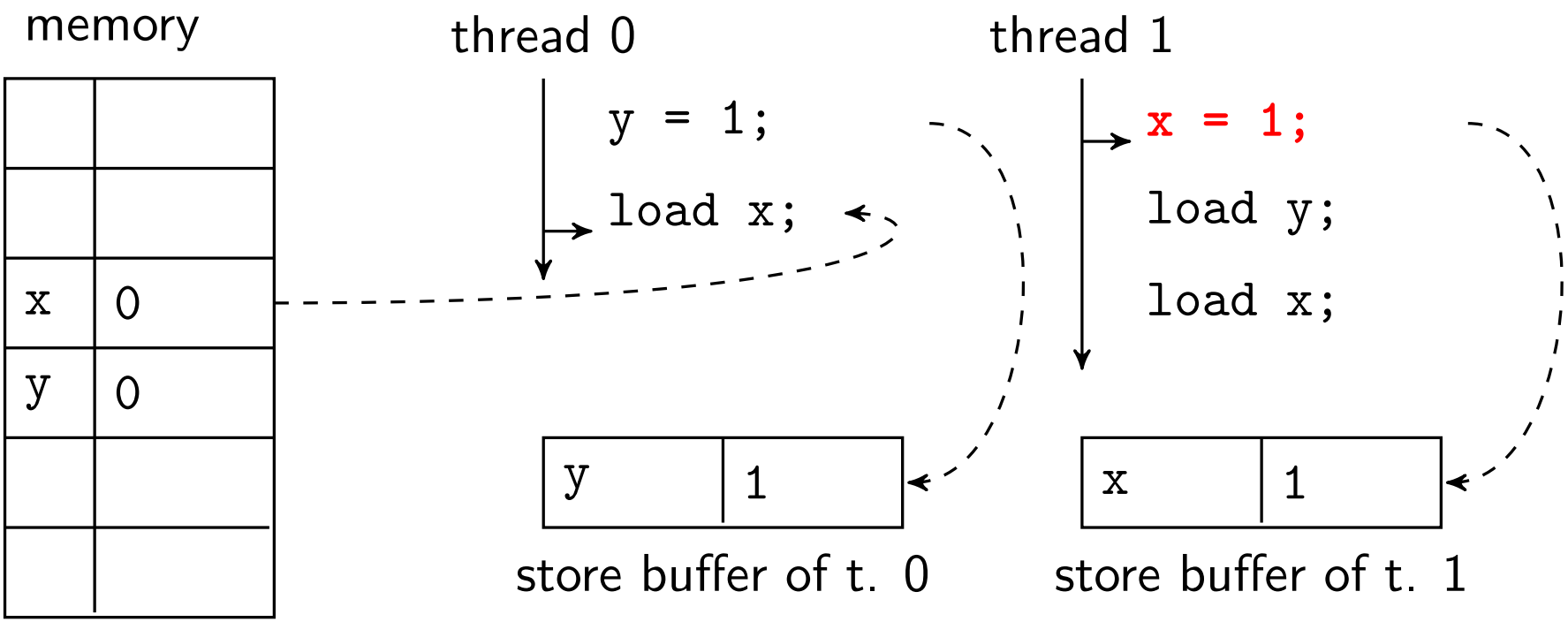
```
void thread1() {
    x = 1;
    int b = y;
    int c = x;
}
```



Relaxed Memory Example

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

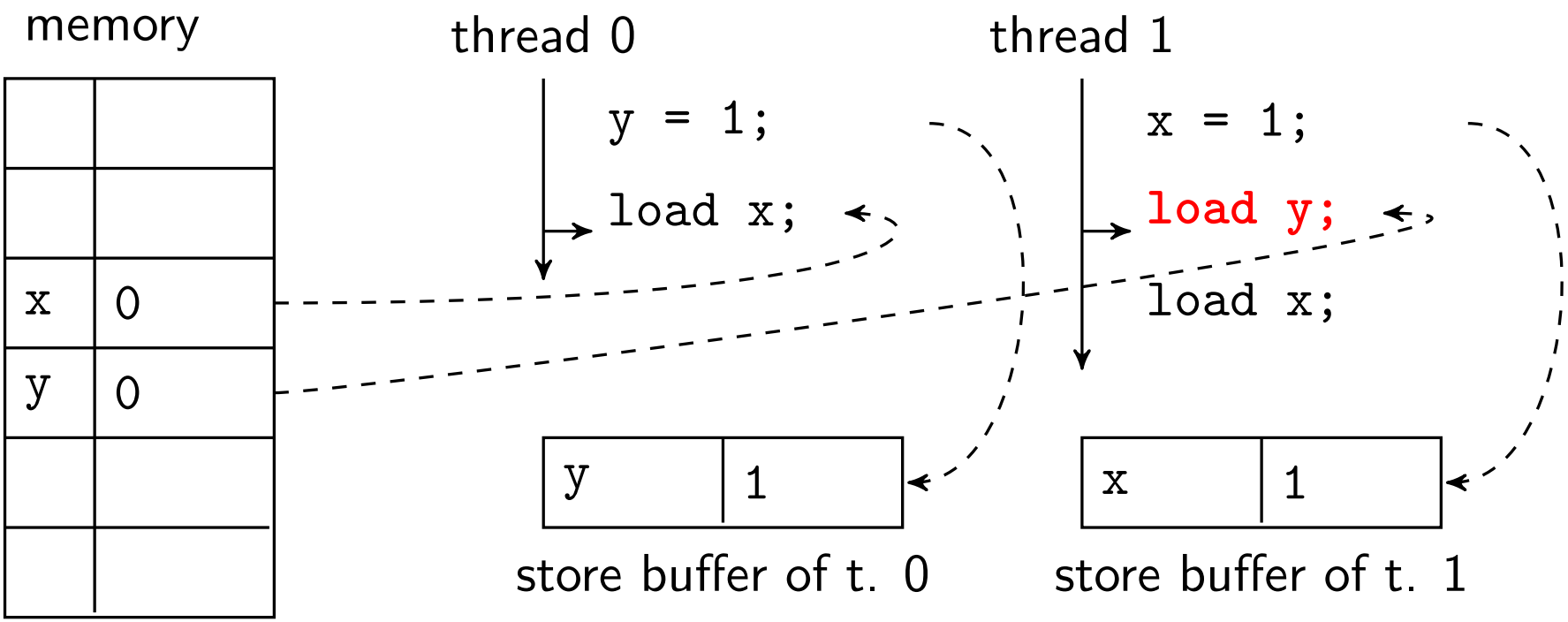
```
void thread1() {
    x = 1;
    int b = y;
    int c = x;
}
```



Relaxed Memory Example

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

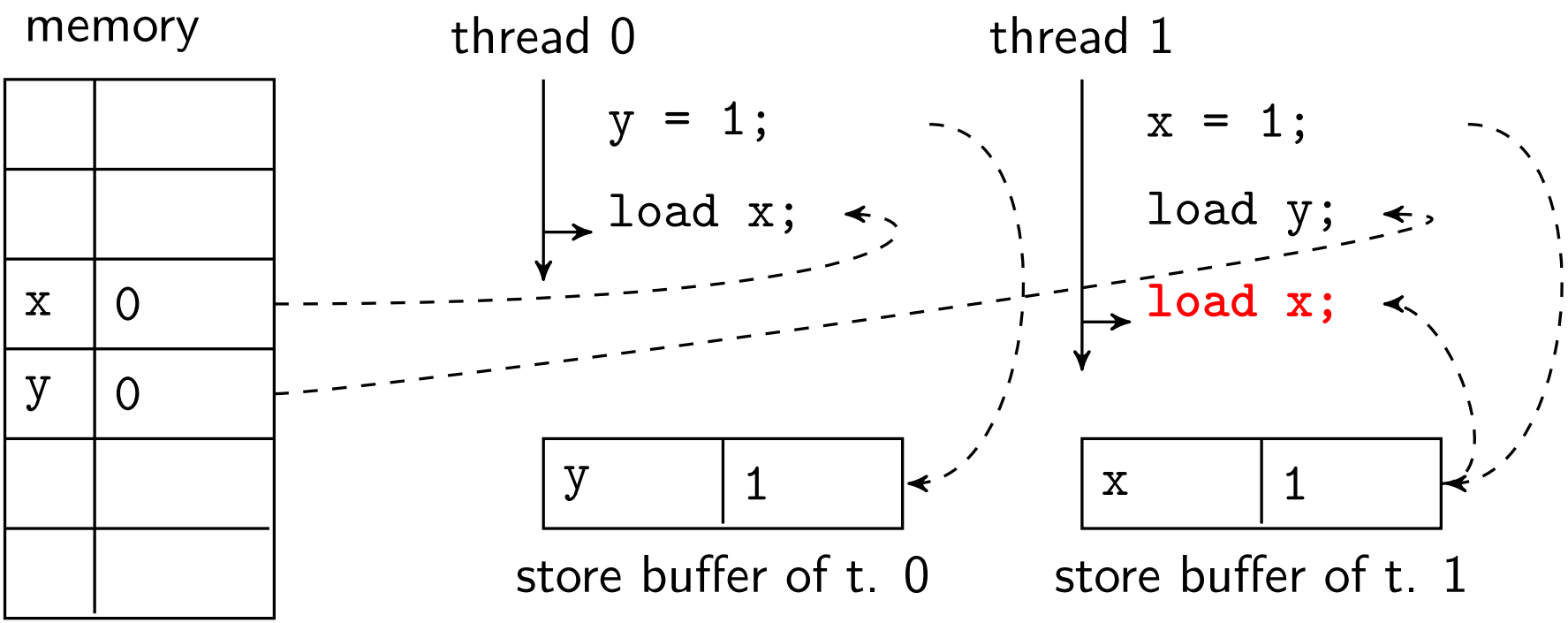
```
void thread1() {
    x = 1;
    int b = y;
    int c = x;
}
```



Relaxed Memory Example

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

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



Relaxed Memory Example

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

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

memory

x	0
y	0

thread 0

```
y = 1;
load x;
```

y	1

store buffer of t. 0

thread 1

```
x = 1;
load y;
load x;
```

x	1

store buffer of t. 1

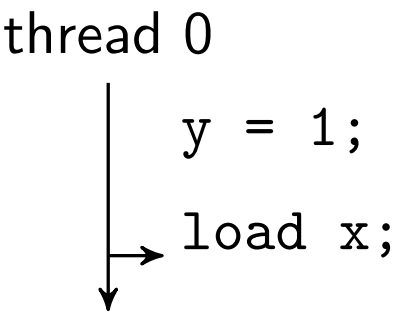
Relaxed Memory Example

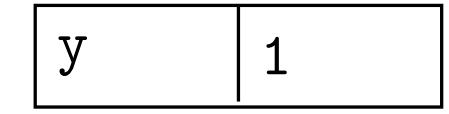
```
int x, y = 0;
void thread0() {
    y = 1;
    int a = x;
}
```

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

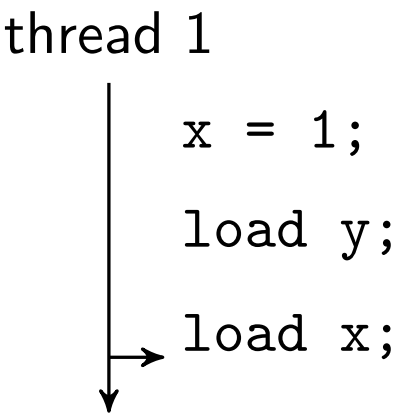
memory

x	1
y	0





store buffer of t. 0



store buffer of t. 1

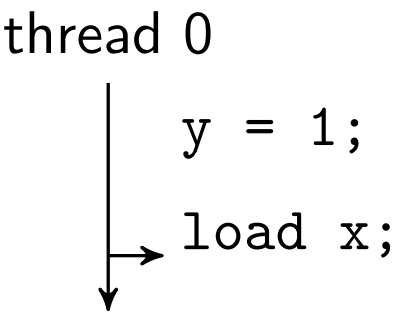
Relaxed Memory Example

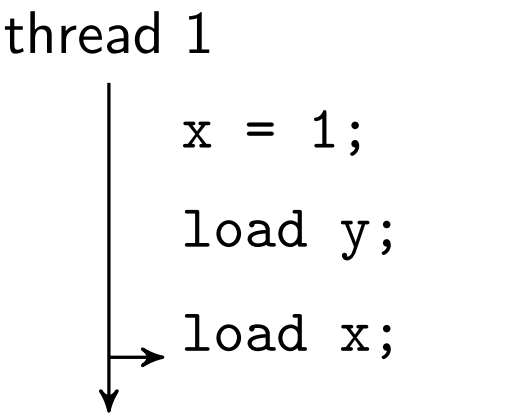
```
int x, y = 0;
void thread0() {
    y = 1;
    int a = x;
}
```

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

memory

x	1
y	1





store buffer of t. 0

store buffer of t. 1

Why Relaxed Memory?

- memory is significantly slower than processor cores
- processor has caches to speed up execution

Why Relaxed Memory?

- memory is significantly slower than processor cores
- processor has caches to speed up execution
- optimizations of cache coherency protocols
  $\rightarrow$ observable effects

Why Relaxed Memory?

- memory is significantly slower than processor cores
- processor has caches to speed up execution
- optimizations of cache coherency protocols
  $\rightarrow$ observable effects
- reordering of instructions might be also observable (not on x86)

Why Relaxed Memory?

- memory is significantly slower than processor cores
- processor has caches to speed up execution
- optimizations of cache coherency protocols
  $\rightarrow$ observable effects
- reordering of instructions might be also observable (not on x86)
- overall behaviour described by a **(relaxed) memory model**

Why Relaxed Memory?

- memory is significantly slower than processor cores
- processor has caches to speed up execution
- optimizations of cache coherency protocols
  $\rightarrow$ observable effects
- reordering of instructions might be also observable (not on x86)
- overall behaviour described by a **(relaxed) memory model**

- now: x86-TSO memory model

Why Relaxed Memory?

- memory is significantly slower than processor cores
- processor has caches to speed up execution
- optimizations of cache coherency protocols
  $\rightarrow$ observable effects
- reordering of instructions might be also observable (not on x86)
- overall behaviour described by a **(relaxed) memory model**

- now: x86-TSO memory model
  - stores are performed to store buffer
  - core-local FIFO buffers
  - entries flushed eventually to the memory

Memory-Model-Aware Analysis I

- encode the memory model into the program
- verify it using a verifier without memory model support
  - e.g. DIVINE, a lot of other verifiers
  - program transformation instead of modification of the verifier

Memory-Model-Aware Analysis I

- encode the memory model into the program
- verify it using a verifier without memory model support
  - e.g. DIVINE, a lot of other verifiers
  - program transformation instead of modification of the verifier

```
x = 1;
int a = y;
```

$\rightsquigarrow$

```
_store( &x, 1 );
int a = _load( &y );
```

- `_load`, `_store` simulate the memory model
- (more complex in practice)

Memory-Model-Aware Analysis II

program transformation

- can be improved with static analysis
- memory model independent
- most complexity is technical

Memory-Model-Aware Analysis II

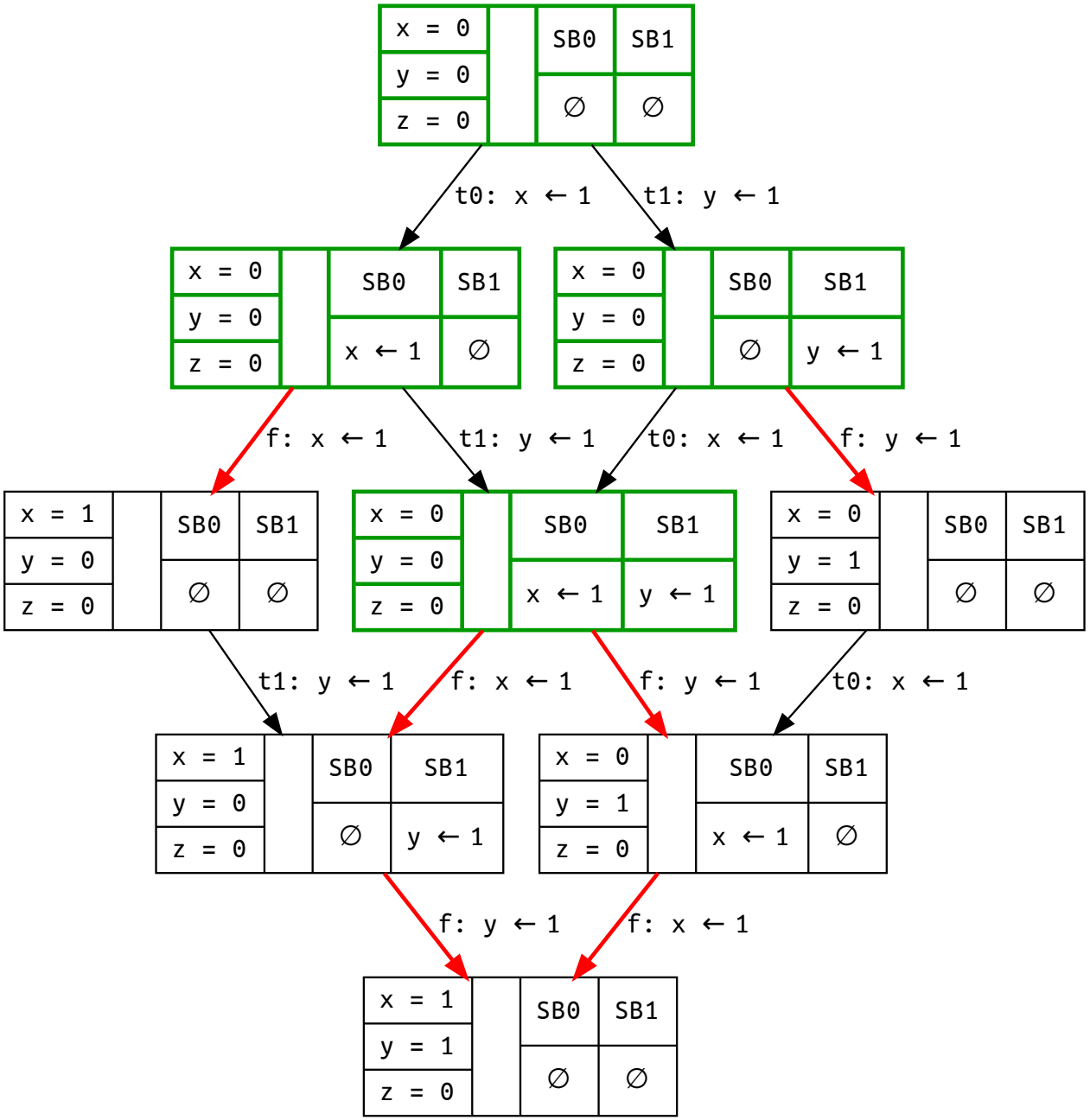
program transformation

- can be improved with static analysis
- memory model independent
- most complexity is technical

memory operations

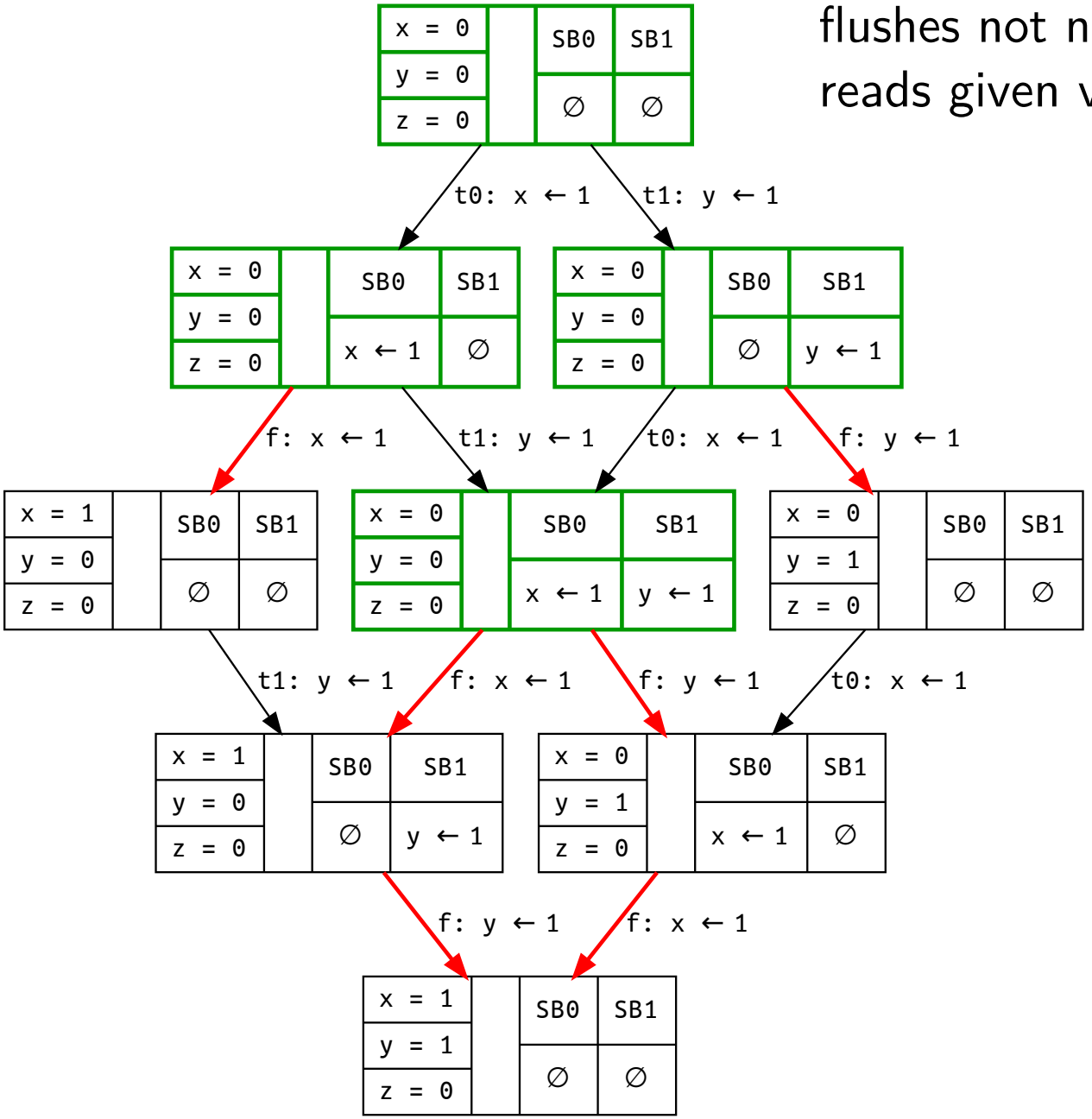
- memory model dependent
- rather complex (theoretically & technically)
- impact efficiency a lot $\rightarrow$ the main aim of my work
  - efficient data structures (time & memory)
  - amount of nondeterminism
- bounded reordering of (effects of) instructions

State Space Explosion



State Space Explosion

flushes not needed if noone reads given value



Critical Observations

1. not all memory is actually accessible by more than one thread (*shared*)

Critical Observations

1. not all memory is actually accessible by more than one thread (*shared*)
2. not all shared memory is actually *accessed* by more that one thread

Critical Observations

1. not all memory is actually accessible by more than one thread (*shared*)
2. not all shared memory is actually *accessed* by more that one thread
3. even memory accessed by more than one threads is usually not accessed by *all of them all the time*

Critical Observations

1. not all memory is actually accessible by more than one thread (*shared*)
2. not all shared memory is actually *accessed* by more that one thread
3. even memory accessed by more than one threads is usually not accessed by *all of them all the time*

- DIVINE's state space reduction uses these observations

Critical Observations

1. not all memory is actually accessible by more than one thread (*shared*)
2. not all shared memory is actually *accessed* by more that one thread
3. even memory accessed by more than one threads is usually not accessed by *all of them all the time*

- DIVINE's state space reduction uses these observations
- but relaxed memory simulation has to be adapted to support this

Lazy x86-TSO

instead of flushing store buffers nondeterministically, flush them only when needed

- i.e. when someone tries to load given address
- need to simulate all outcomes $\rightarrow$ nondeterminism in load

Lazy x86-TSO

instead of flushing store buffers nondeterministically, flush them only when needed

- i.e. when someone tries to load given address
- need to simulate all outcomes $\rightarrow$ nondeterminism in load
- how to handle other entries in store buffer?

Lazy x86-TSO

instead of flushing store buffers nondeterministically, flush them only when needed

- i.e. when someone tries to load given address
- need to simulate all outcomes $\rightarrow$ nondeterminism in load
- how to handle other entries in store buffer?
- memory barriers and compare-and-swap/read-modify-write not fully lazy

Lazy x86-TSO

instead of flushing store buffers nondeterministically, flush them only when needed

- i.e. when someone tries to load given address
- need to simulate all outcomes $\rightarrow$ nondeterminism in load
- how to handle other entries in store buffer?
- memory barriers and compare-and-swap/read-modify-write not fully lazy
  - flushing of local store buffer can nondeterministically flush entries from other buffers
  - fully lazy barriers would show down DiOS

Lazy x86-TSO – Current Status I

the lazy simulation of x86-TSO store buffers mostly done

- one known missing corner case
  - in sequence of stores to unrelated addresses
  - solution will probably increase laziness and therefore performance
- probably some space for speed improvement

Lazy x86-TSO – Current Status II

current delays caused by interaction with state space reductions

Lazy x86-TSO – Current Status II

current delays caused by interaction with state space reductions

- store buffers look like shared memory for reduction
- ensure reduction does not see every operation as visible

Lazy x86-TSO – Current Status II

current delays caused by interaction with state space reductions

- store buffers look like shared memory for reduction
- ensure reduction does not see every operation as visible
- not everything in memory model implementation can be hidden
  – “real” loads/stores

Lazy x86-TSO – Future

1. finish implementation
2. add more tests and benchmarks
3. compare with other tools
4. publish

Lazy x86-TSO – Future

1. finish implementation
2. add more tests and benchmarks
3. compare with other tools
4. publish
5. SV-COMP demo category?
6. (optimize & improve & publish)$^+$

Lazy x86-TSO – Future

1. finish implementation
2. add more tests and benchmarks
3. compare with other tools
4. publish
5. SV-COMP demo category?
6. (optimize & improve & publish)$^+$

That is all... Thank You!