Memory-Model-Aware Analysis of Parallel Programs

PhD Thesis Proposal

Vladimír Štill

Masaryk University

Brno, Czech Republic

16th January 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

- I work on the DIVINE model checker
- we focus on C and C++

Verification of Parallel Programs II

- I work on the DIVINE model checker
- we focus on 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

- I work on the DIVINE model checker
- we focus on 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
- 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;
}
```

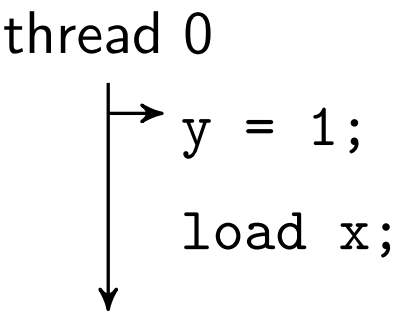
Relaxed Memory Example

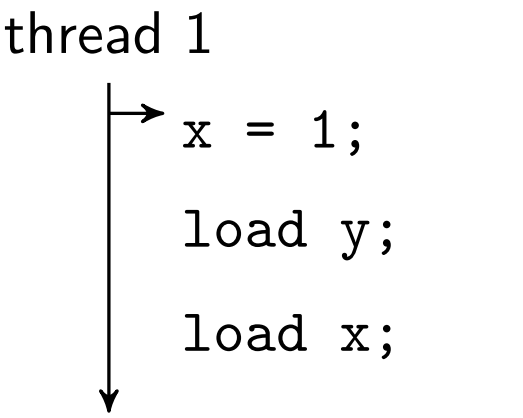
```
int x, y = 0;
void thread0() {
    y = 1;
    int a = x;
}
```

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

memory

x	0
y	0





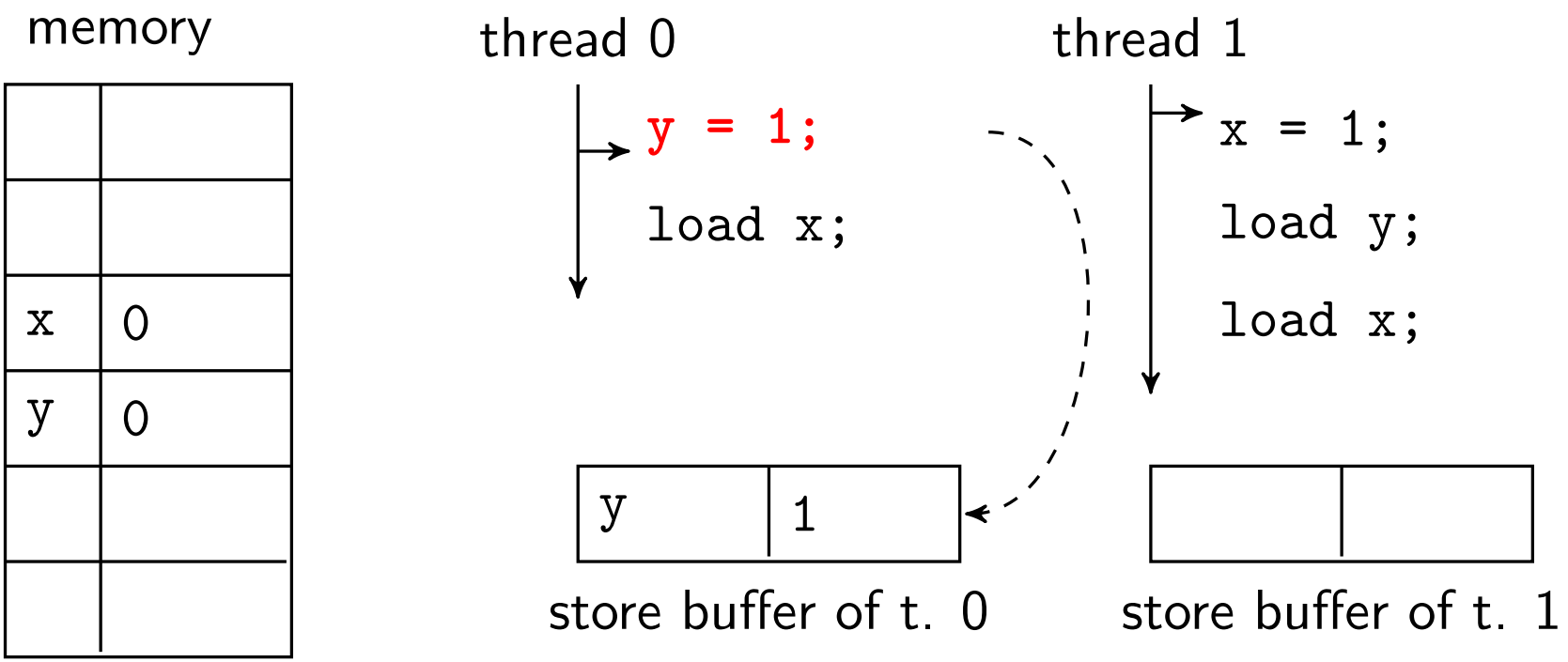
store buffer of t. 0

store buffer of t. 1

Relaxed Memory Example

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

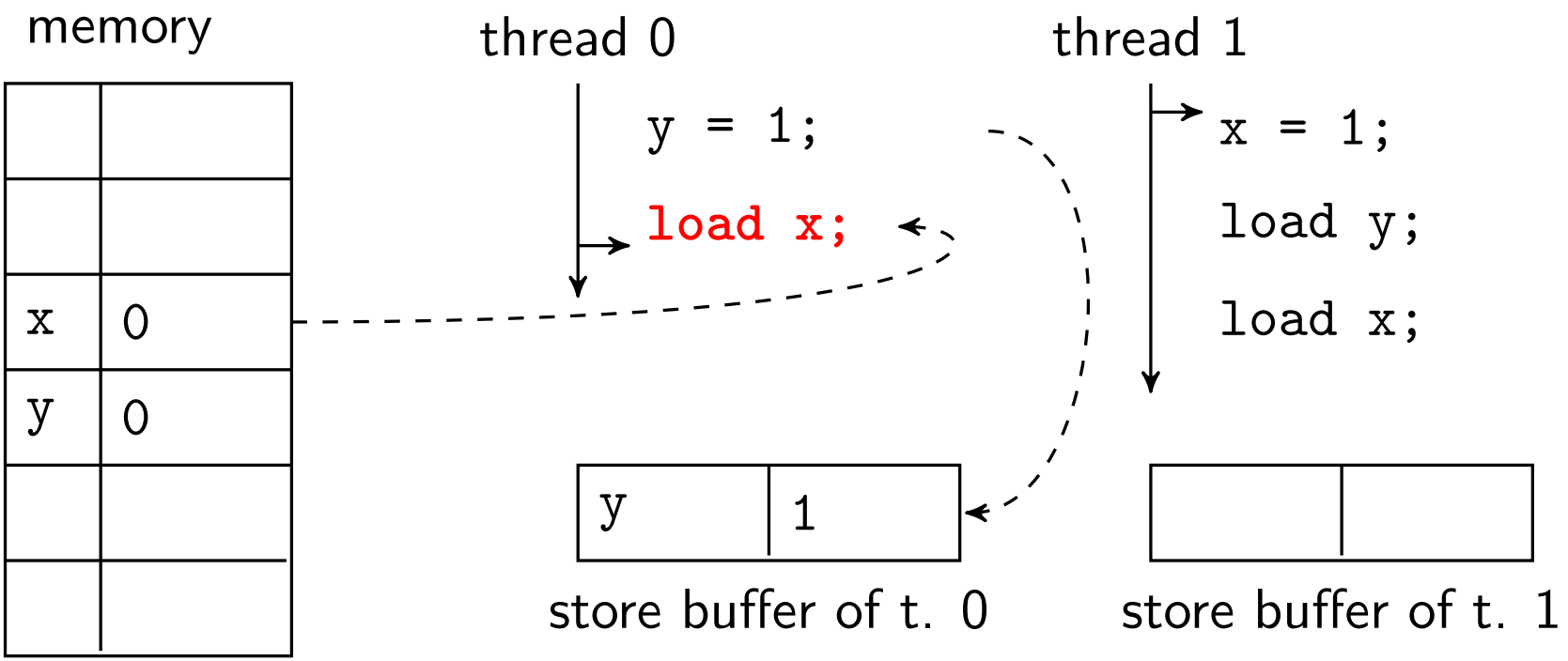
```
void thread1() {
    x = 1;
    int b = y;
    int c = x;
}
```



Relaxed Memory Example

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

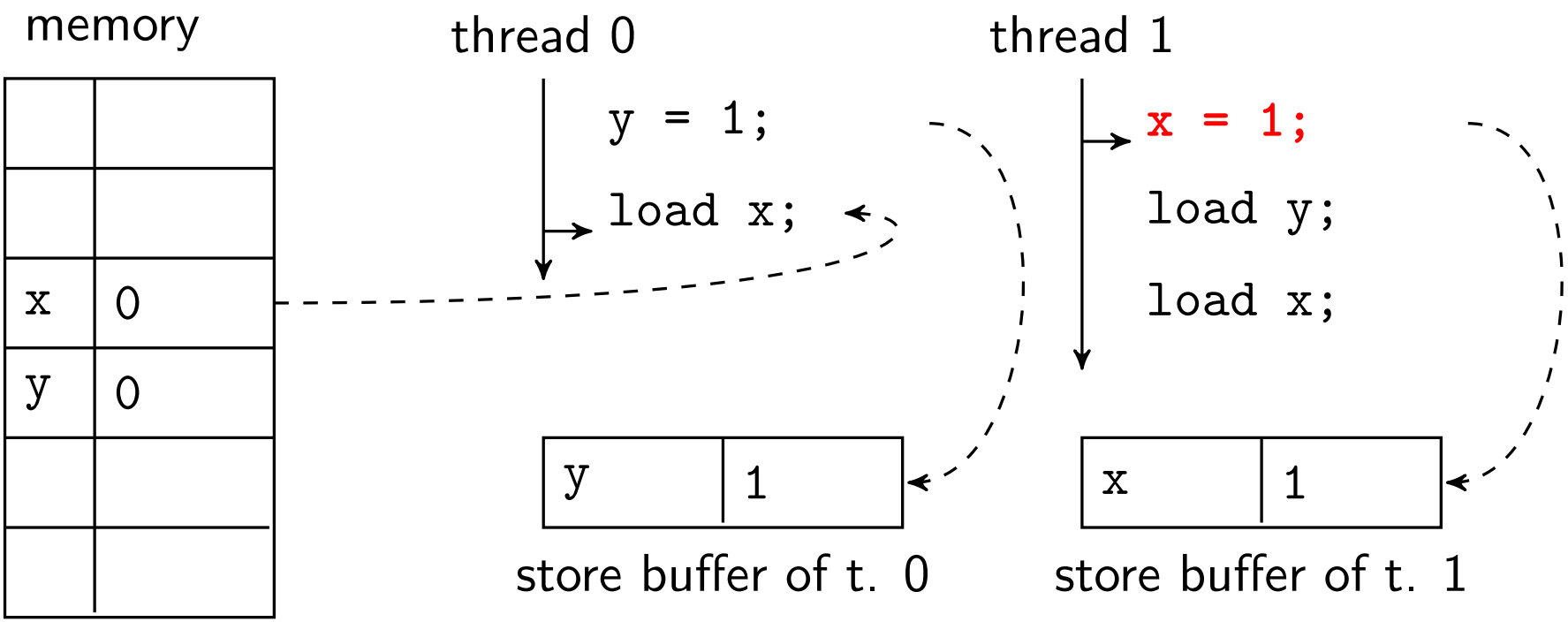
```
void thread1() {
    x = 1;
    int b = y;
    int c = x;
}
```



Relaxed Memory Example

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

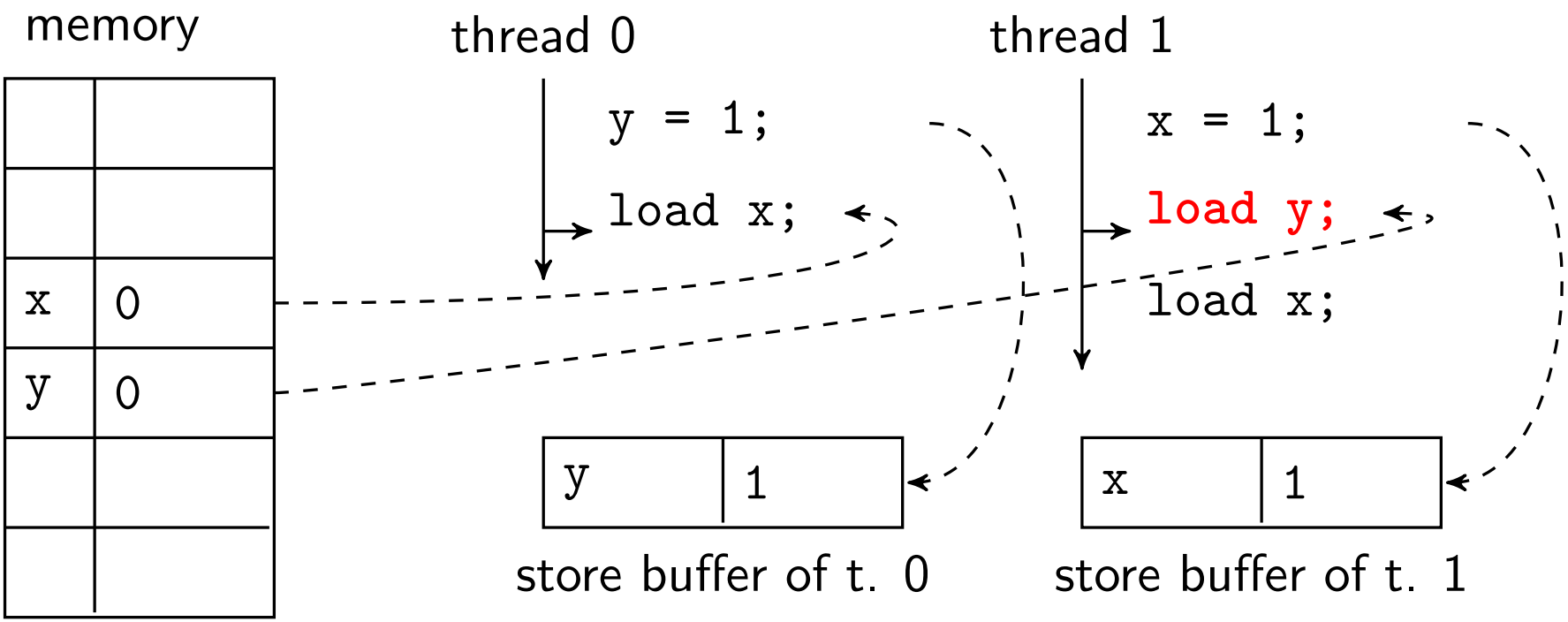
```
void thread1() {
    x = 1;
    int b = y;
    int c = x;
}
```



Relaxed Memory Example

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

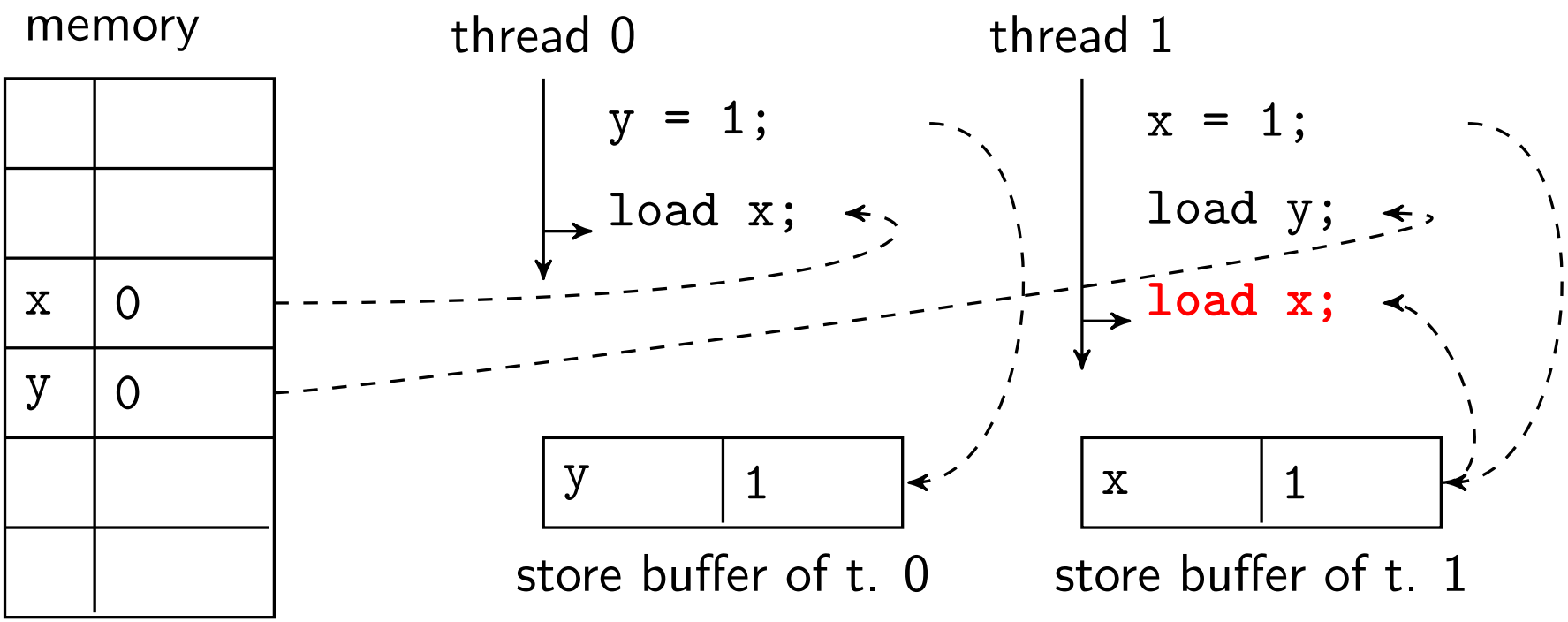
```
void thread1() {
    x = 1;
    int b = y;
    int c = x;
}
```



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

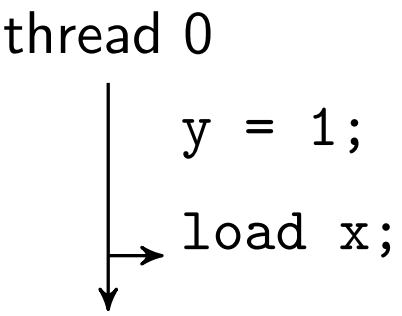
Relaxed Memory Example

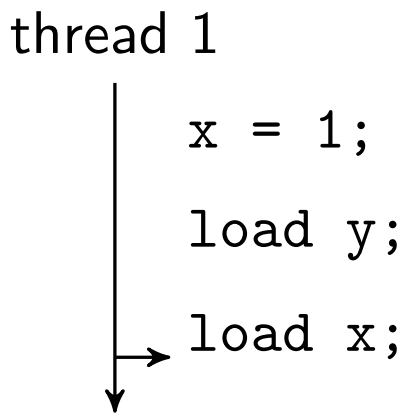
```
int x, y = 0;
void thread0() {
    y = 1;
    int a = x;
}
```

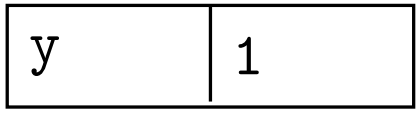
```
void thread1() {
    x = 1;
    int b = y;
    int c = x;
}
```

memory

x	1
y	0







store buffer of t. 0

store buffer of t. 1

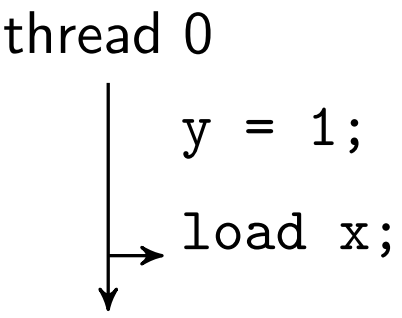
Relaxed Memory Example

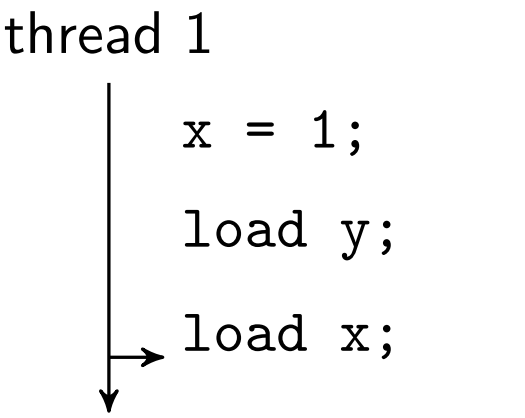
```
int x, y = 0;
void thread0() {
    y = 1;
    int a = x;
}
```

```
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

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
- overall behaviour described by a **(relaxed) memory model**

Memory-Model-Aware Analysis – My Approach 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 – My Approach 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
- can be more robust
- same program transformation for multiple verifiers

Memory-Model-Aware Analysis – My Approach 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
- can be more robust
- same program transformation for multiple verifiers

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

$\rightsquigarrow$

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

- `_load`, `_store` simulate the memory model

Memory-Model-Aware Analysis – My Approach II

program transformation

- relatively straightforward
- can be improved with static analysis
- memory model independent

Memory-Model-Aware Analysis – My Approach II

program transformation

- relatively straightforward
- can be improved with static analysis
- memory model independent

memory operations

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

Aims of the Work

- implement an LLVM-based program transformation for relaxed memory models
  - done and quite stable, unlikely to need significant changes

Aims of the Work

- implement an LLVM-based program transformation for relaxed memory models
  - done and quite stable, unlikely to need significant changes
- provide efficient support for memory models
  - for `x86-TSO` and Non-Speculative Writes
  - evaluate using DIVINE, other LLVM-based parallel verifiers

Aims of the Work

- implement an LLVM-based program transformation for relaxed memory models
  - done and quite stable, unlikely to need significant changes
- provide efficient support for memory models
  - for `x86-TSO` and Non-Speculative Writes
  - evaluate using DIVINE, other LLVM-based parallel verifiers
- explore heuristically-directed exploration to increase efficiency
  - tighter integration with the analysis tool

Aims of the Work

- implement an LLVM-based program transformation for relaxed memory models
  - done and quite stable, unlikely to need significant changes
- provide efficient support for memory models
  - for `x86-TSO` and Non-Speculative Writes
  - evaluate using DIVINE, other LLVM-based parallel verifiers
- explore heuristically-directed exploration to increase efficiency
  - tighter integration with the analysis tool
- analysis of very weak memory models
  - e.g. ARM and POWER, which can feature write reordering

Aims of the Work

- implement an LLVM-based program transformation for relaxed memory models
  - done and quite stable, unlikely to need significant changes
- provide efficient support for memory models
  - for `x86-TSO` and Non-Speculative Writes
  - evaluate using DIVINE, other LLVM-based parallel verifiers
- explore heuristically-directed exploration to increase efficiency
  - tighter integration with the analysis tool
- analysis of very weak memory models
  - e.g. ARM and POWER, which can feature write reordering
- techniques for unbounded analysis

Publications I

Techniques for Memory-Efficient Model Checking of C and C++ Code

Petr Ročkai, Vladimír Štill, and Jiří Barnat. "Techniques for Memory-Efficient Model Checking of C and C++ Code". In: *Software Engineering and Formal Methods*. Vol. 9276. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 268–282

- tree-based compression scheme, custom allocation scheme
- I have made part of the implementation (compression), whole evaluation, and part of the text
- I have presented this at SEFM 2015 (rank B)

Publications II

Weak Memory Models as LLVM-to-LLVM Transformation

Vladimír Štill, Petr Ročkai, and Jiří Barnat. "Weak Memory Models as LLVM-to-LLVM Transformations". In: *Mathematical and Engineering Methods in Computer Science, Revised Selected Papers*. Vol. 9548. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 144–155

- analysis of programs under the `x86-TSO` memory model using LLVM transformation
- I am the main author of this paper
- I have also presented this at MEMICS 2015

Publications III

Using Off-the-Shelf Exception Support Components in C++ Verification

Vladimír Štill, Petr Ročkai, and Jiří Barnat. "Using Off-the-Shelf Exception Support Components in C++ Verification". In: *Software Quality, Reliability and Security (QRS)*. IEEE, July 2017, pp. 54–64

- verification of C++ code with exceptions in DIVINE 4
- with maximal reuse of existing libraries
- I am the main author of this paper
- I have presented this at QRS 2017 (rank B)
- awarded best paper award

Publications IV

Model Checking of C and C++ with DIVINE 4

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: *Automated Technology for Verification and Analysis*. Vol. 10482. Lecture Notes in Computer Science. 2017

- tool paper describing architecture of DIVINE 4 and new features of this version
- I have written most of the text for this paper
- ATVA 2017 (rank A)

Reader's Questions

Otázky: Tomáš Vojnar I

V práci je několikrát uvedeno, že testování není pro paralelní programy užitečné. V souvislosti s tím bych se rád zeptal, kam přesně klade doktorand hranici pojmu testování a zda za neužitečné považuje i přístupy, jako je systematické testování či dynamická analýza s využitím algoritmů jako např. FastTrack.

- primárně mi jde o klasické postupy testování jako jednotkové testování nebo stress testing, případně s použitím nástrojů jako thread sanitizer nebo helgrind
- FastTrack umí detekovat data race, ale nevidím, jak by bylo možné jej adaptovat pro hledání chyb v lock-free programech v C/C++: pokud jsou všechny sdílené proměnné označeny jako atomické, pak podle C/C++ v programu nejsou data races, ale mohou se v něm snadno projevit chyby plynoucí z relaxované paměti

Otázky: Tomáš Vojnar II

Dále bych se doktoranda rád zeptal, jak chápe pojmy „soundness“ a „completeness“ v případě analyzátorů programů? Považuje za „sound“ přístup, který nemusí odhalit v programu chyby?

- *soundness*: analyzátor může neskončit nebo odpovědět NEVÍM, pokud však odpoví ANO/NE, odpověď musí být správná
  - např. bounded model checker, který při detekci překročení omezení odpoví NEVÍM, může být „sound“, pokud opoví ANO, není „sound“
- *completeness*: analyzátor musí skončit pro programy, které mají konečný stavový prostor pod modelem sekvenční konzistence, a když skončí, dát výsledek ANO/NE

Otázky: Tomáš Vojnar III

Konečně bych rád přesnější objasnění, proč je zapotřebí zvláštní podpora paměťových operací v modelu NSW (viz cíle v sekci 4.1.2) a není ji možno pokrýt přístupy definovanými v sekci 4.1.1.

- část 4.1.1 se zabývá primárně transformací/instrumentací LLVM, ale neřeší konkrétní implementace paměťového modelu
  - tato část musí být dostatečně generická pro zapojení různých paměťových modelů
- v části 4.1.2 jde primárně o efektivní implementaci konkrétního paměťového modelu nad základy vybudovanými v 4.1.1

Otázky: Petr Švenda I

Jaká jsou praktická omezení akutálního nástroje z pohledu velikosti a komplexnosti analyzovaného kódu? Na jaké databázi zdrojových kódů je funknčnost nástroje testována?

- u paralelních programů záleží především na množství vláken a míře jejich komunikace
  - lock-free je typicky náročnější než program korektně pracující se zámky
  - jednodušší lock-free datové struktury na 2–4 vláknech
  - složitější na 2 vláknech
- testy: cca 700 testů na veškerou funkcionalitu DIVINE
  - vlastní testy, testy C, C++ knihovny

Otázky: Petr Švenda II

Jaké je strovnání vyvíjeného nástroje DIVINE (a speciálně jeho části řešené studentem v rámci předložených tezí) s dalšími výzkumnými i komerčně dostupnými nástroji?

- co se týče výkonu, porovnání v oblasti relaxované paměti zatím není
- komerční nástroje často založeny na statické analýze
  - riziko false alarms – v DIVINE by být neměly
- výzkumné nástroje
  - DIVINE má typicky lepší podporu jazyka (především u C++)
  - typicky jiný princip (bounded MC, stateless MC, symbolická exekuce) $\rightarrow$ jiné kompromisy
  - přístup k relaxované paměti také může být různý: omezení přeuspořádání, omezení množství přepnutí kontextu, ...

Otázky: Petr Švenda III

Dále bych požádal o bližší vysvětlení pozice nástroje DIVINE z pohledu formální verifikace vs. hledání chyb. Pro jaké typy kontrolovaných programů lze očekávat realizovatelnost kompletní formální verifikace a ve kterých oblastech bude hlavním přínosem hledání chyb?

- snaha o verifikaci, jsou tu ale jistá omezení
  - **relaxovaná paměť**: limit na množství přeuspořádaných instrukcí
  - otevřené programy (verifikace pro všechny vstupy): nyní jen omezená podpora
  - vždy: riziko chyb v překladu LLVM na binární kód, v DIVINE
    - také v překladači/knihovně při překladu výsledného spustitelného programu jiným překladačem
- omezením je velikost analyzovaného programu
  - pro velké programy plánovaný bugfinding mód v DIVINE

Otázky: Petr Švenda IV

Je možné vyvíjený nástroj rozšířit o automatické vkládání synchronizačních bariér tak, aby se nástrojem detekovaný problém ihned odstranil?

- není vždy zřejmé, co má být správné chování, či jestli je chyba způsobená relaxovaným paměťovým modelem
- mířím na programátora, který „ví, co dělá“, ale potřebuje kontrolu
- v principu by mělo být možné detekovat „relaxované chování“, ale přijde mi smysluplnější soustředit se spíše na efektivitu hledání chyb