Bounding Data Races in *Space* and *Time*

KC Sivaramakrishnan

University of Cambridge

OCaml Labs

Darwin College, Cambridge

1851 Royal Commission

Multicore OCaml

Multicore OCaml

- OCaml is an industrial-strength, functional programming language
  - ★ Projects: MirageOS unikernel, Coq proof assistant, F* programming language
  - ★ Companies: Facebook (Hack, Flow, Infer, Reason), Microsoft (Everest, F*), JaneStreet (all trading & support systems), Docker (Docker for Mac & Windows), Citrix (XenStore)

Multicore OCaml

- OCaml is an industrial-strength, functional programming language
  - ★ Projects: MirageOS unikernel, Coq proof assistant, F* programming language
  - ★ Companies: Facebook (Hack, Flow, Infer, Reason), Microsoft (Everest, F*), JaneStreet (all trading & support systems), Docker (Docker for Mac & Windows), Citrix (XenStore)
- *No multicore support!*

Multicore OCaml

- OCaml is an industrial-strength, functional programming language
  - ★ Projects: MirageOS unikernel, Coq proof assistant, F* programming language
  - ★ Companies: Facebook (Hack, Flow, Infer, Reason), Microsoft (Everest, F*), JaneStreet (all trading & support systems), Docker (Docker for Mac & Windows), Citrix (XenStore)
- *No multicore support!*
- Multicore OCaml
  - ★ Native support for *concurrency* and *parallelism* in OCaml
  - ★ Lead from OCaml Labs + (JaneStreet, Microsoft Research, INRIA).

Modelling Memory

Modelling Memory

- How do you reason about access to memory?

Modelling Memory

- How do you reason about access to memory?
  - ★ *Spoiler: No single global sequentially consistent memory*

Modelling Memory

- How do you reason about access to memory?
  - ★ *Spoiler: No single global sequentially consistent memory*
- Modern multicore processors reorder instructions for performance

Modelling Memory

- How do you reason about access to memory?
  - ★ *Spoiler: No single global sequentially consistent memory*
- Modern multicore processors reorder instructions for performance

Initially a = 0 && b =0

Thread 1	**Thread 2**
`a = 1`	`b = 1`
`r1 = b`	`r2 = a`

r1 == 0 && r2 ==0 ???

Modelling Memory

- How do you reason about access to memory?
  - ★ *Spoiler: No single global sequentially consistent memory*
- Modern multicore processors reorder instructions for performance

Initially a = 0 && b =0

Thread 1	**Thread 2**
`a = 1`	`b = 1`
`r1 = b`	`r2 = a`

r1 == 0 && r2 ==0 ???

Allowed under x86, ARM, POWER

Modelling Memory

- How do you reason about access to memory?
  - ★ *Spoiler: No single global sequentially consistent memory*
- Modern multicore processors reorder instructions for performance

Initially a = 0 && b =0

Thread 1	**Thread 2**
`a = 1`	`b = 1`
`r1 = b`	`r2 = a`

r1 == 0 && r2 ==0 ???

Allowed under x86, ARM, POWER

Write buffering

Modelling Memory

- How do you reason about access to memory?
  - ★ *Spoiler: No single global sequentially consistent memory*
- Modern multicore processors reorder instructions for performance

Initially a = 0 && b =0

Thread 1	**Thread 2**
`r1 = b`	`r2 = a`
`a = 1`	`b = 1`

r1 == 0 && r2 ==0 ???

Write buffering

Allowed under x86, ARM, POWER

Modelling Memory

- Compilers optimisations also reorder memory access instructions

Modelling Memory

- Compilers optimisations also reorder memory access instructions

Thread 1

```
r1 = a * 2
r2 = b + 1
r3 = a * 2
```

CSE ⟶

Thread 1

```
r1 = a * 2
r2 = b + 1
r3 = r1
```

Modelling Memory

- Compilers optimisations also reorder memory access instructions

Thread 1

```
r1 = a * 2
r2 = b + 1
r3 = a * 2
```

CSE ⟶

Thread 1

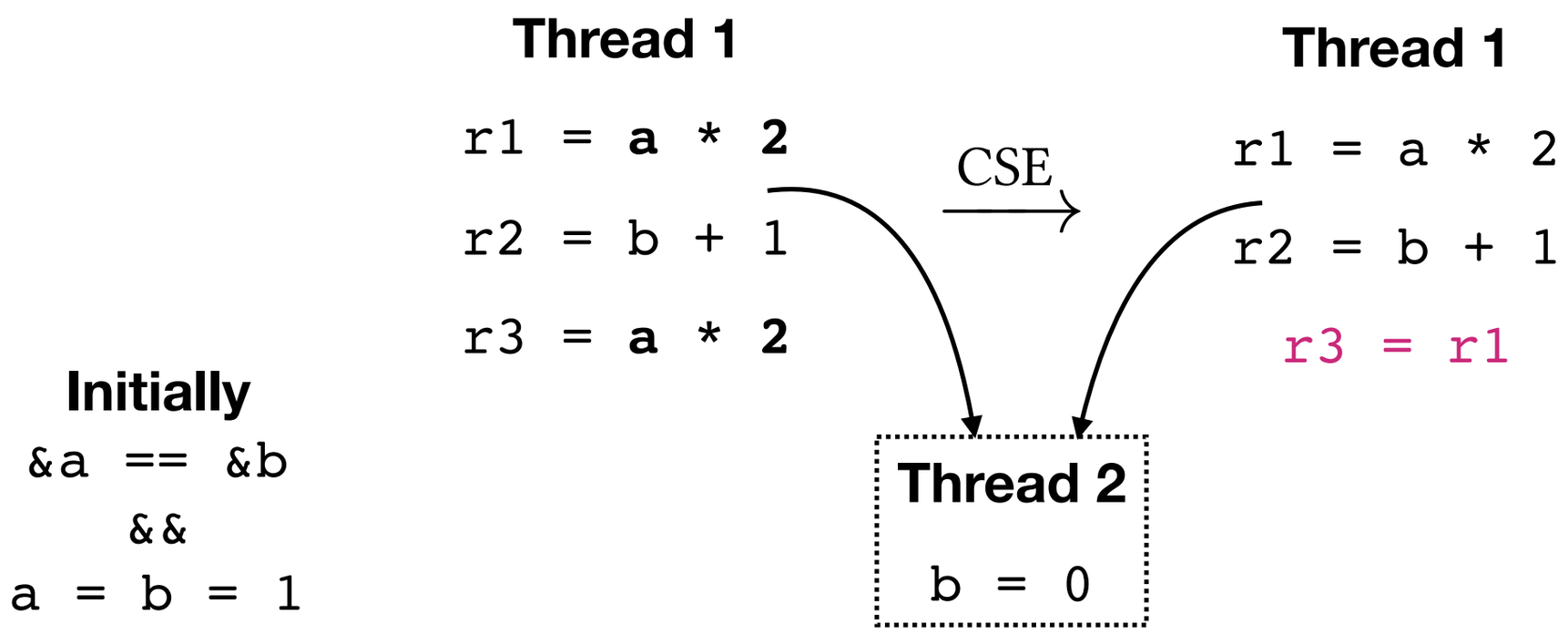
```
r1 = a * 2
r2 = b + 1
r3 = r1
```

Initially

```
&a == &b
   &&
a = b = 1
```

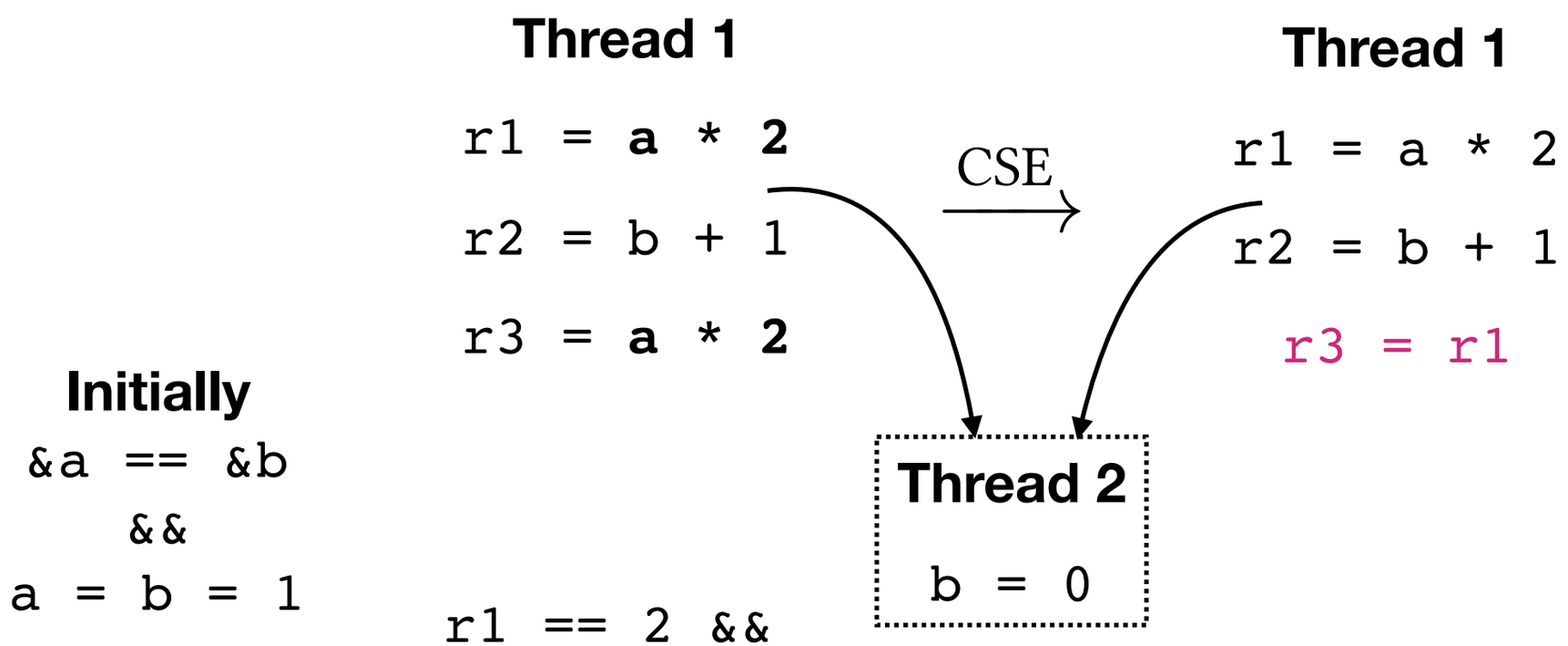
Modelling Memory

- Compilers optimisations also reorder memory access instructions



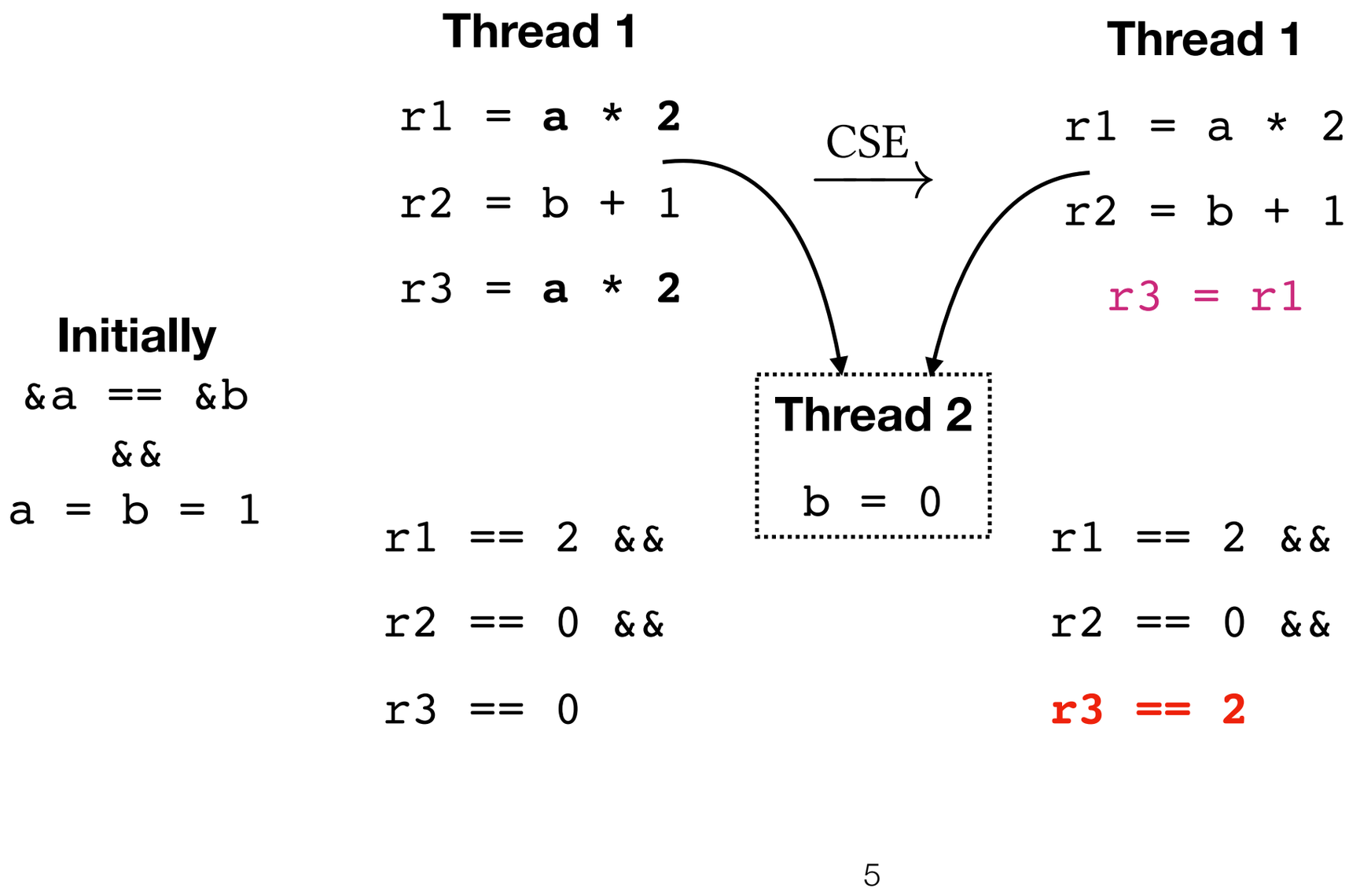
Modelling Memory

- Compilers optimisations also reorder memory access instructions



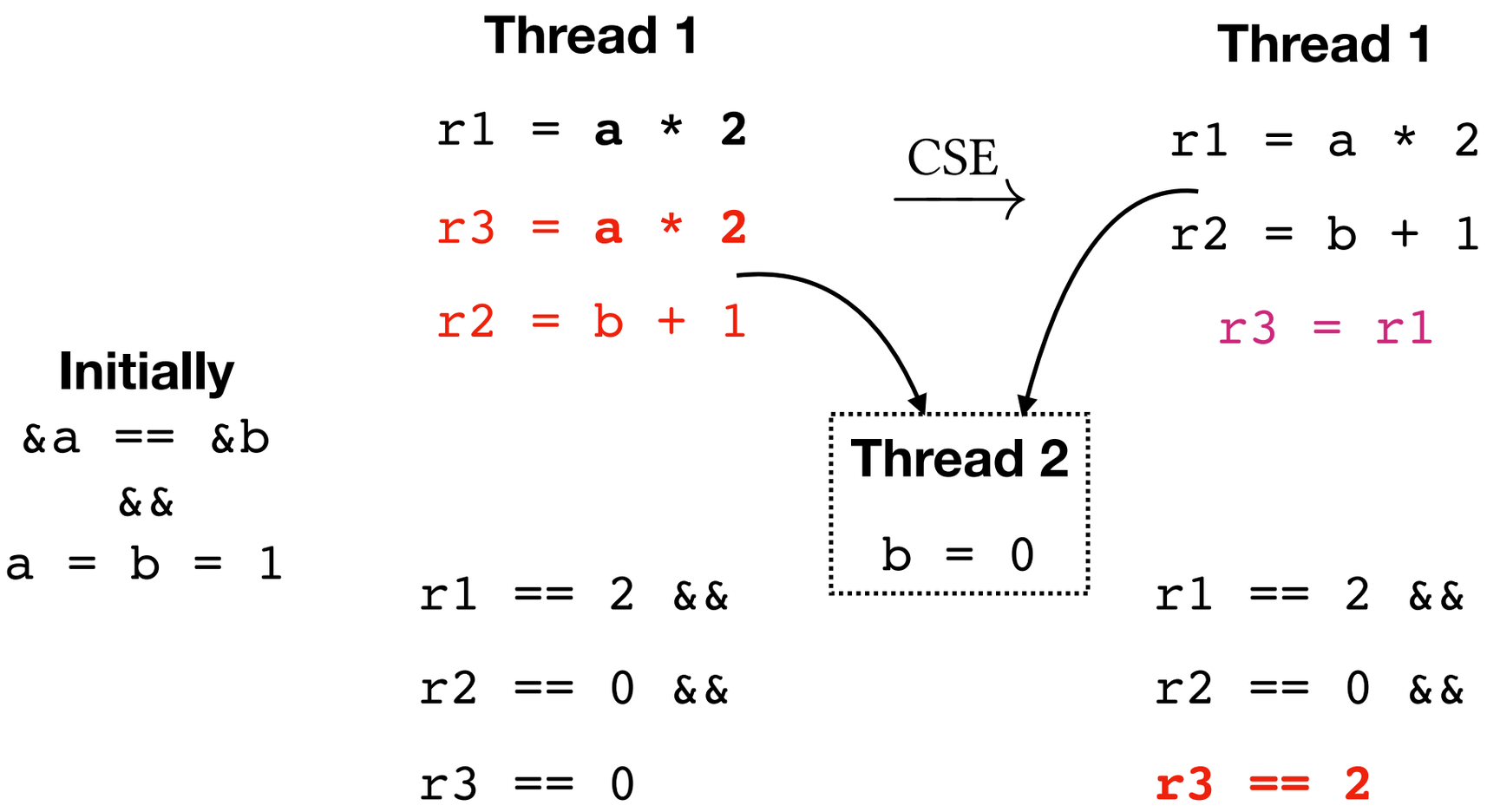
Modelling Memory

- Compilers optimisations also reorder memory access instructions



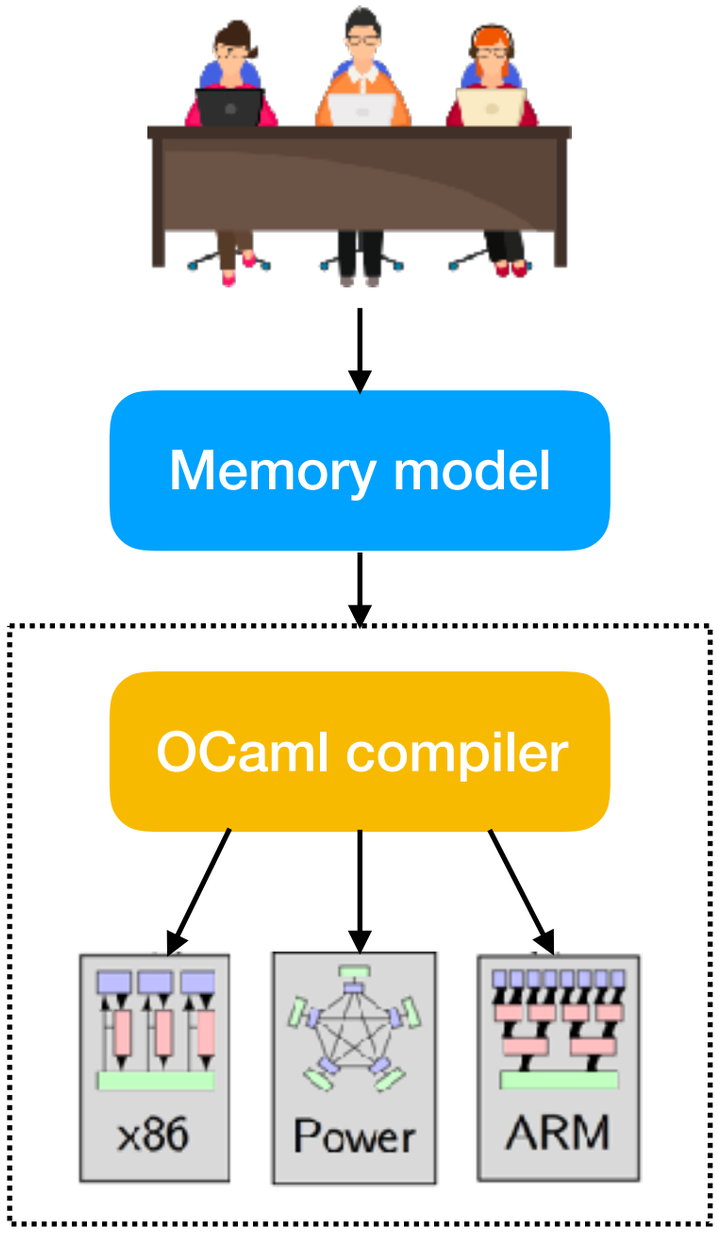
Modelling Memory

- Compilers optimisations also reorder memory access instructions

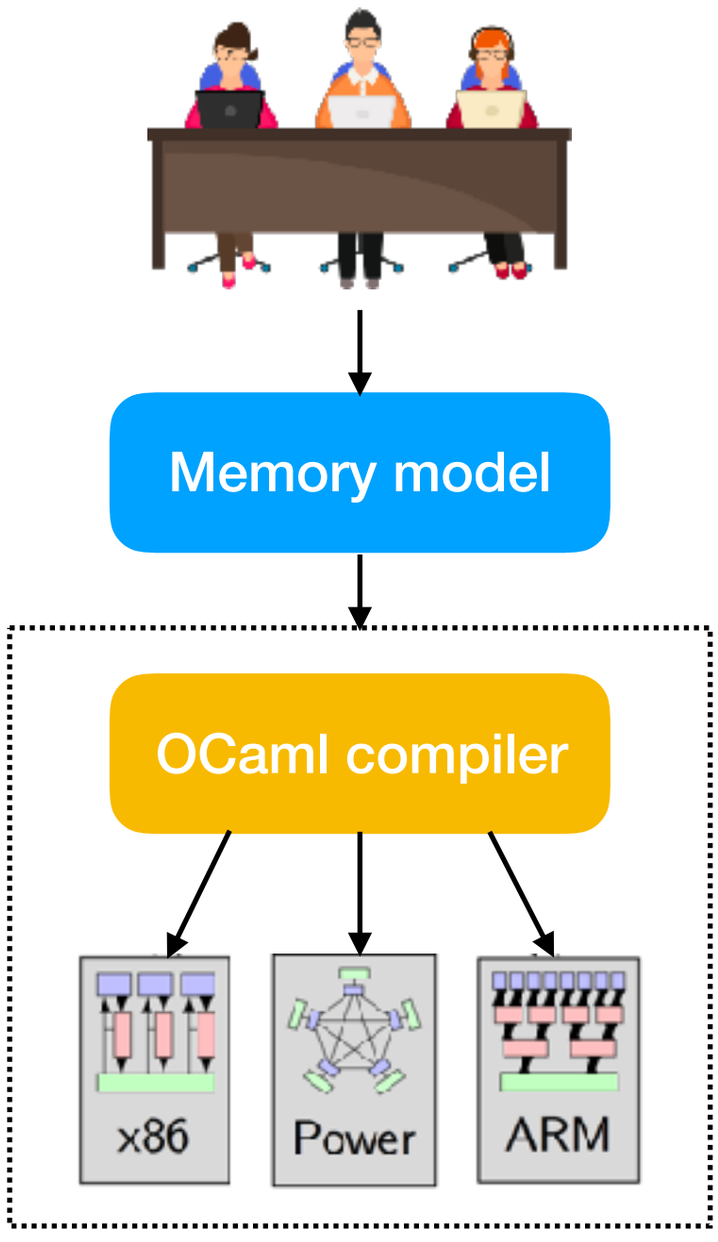


Memory Model

- Unambiguous specification of program outcomes
  - ★ More than just thread interleavings

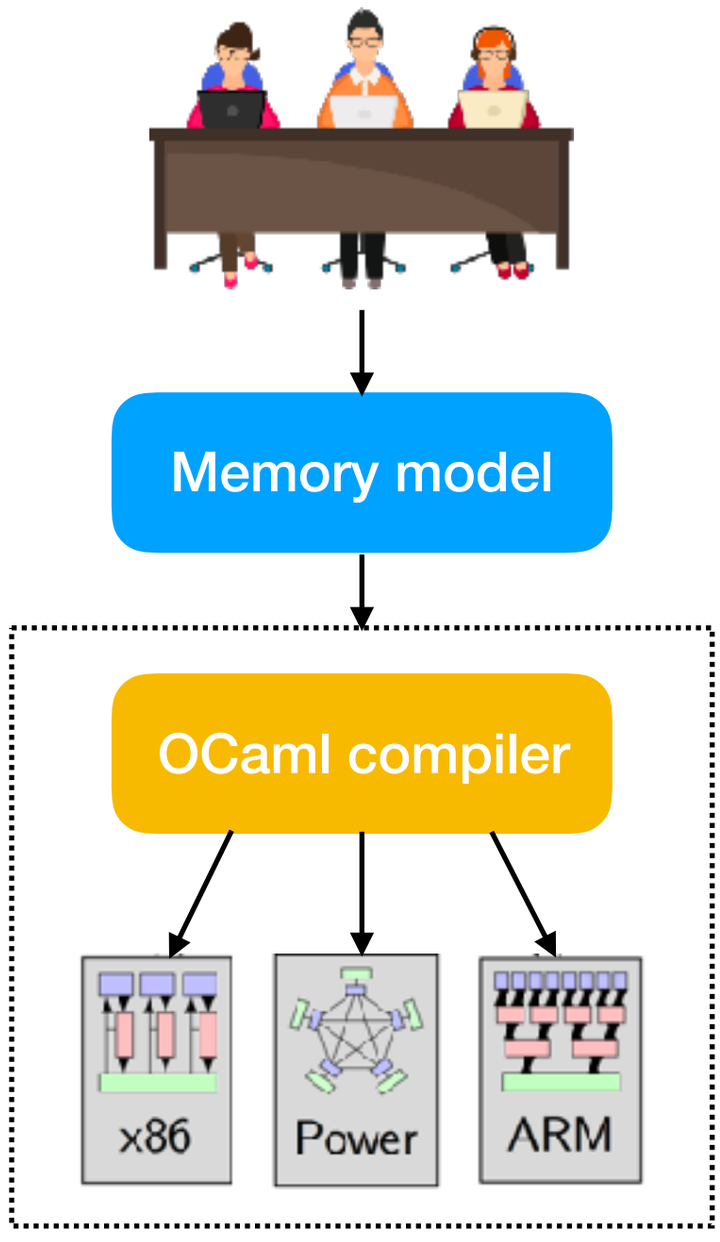


Memory Model



- Unambiguous specification of program outcomes
  - ★ More than just thread interleavings
- Memory Model Desiderata
  - ★ Not too weak (good for programmers)
  - ★ Not too strong (good for hardware)
  - ★ Admits optimisations (good for compilers)
  - ★ Mathematically rigorous (good for verification)

Memory Model



- Unambiguous specification of program outcomes
  - ★ More than just thread interleavings
- Memory Model Desiderata
  - ★ Not too weak (good for programmers)
  - ★ Not too strong (good for hardware)
  - ★ Admits optimisations (good for compilers)
  - ★ Mathematically rigorous (good for verification)
- Difficult to get right
  - ★ C/C++11 memory model is flawed
  - ★ Java memory model is flawed
  - ★ Several papers every year in top PL conferences proposing / fixing models

Memory Model: Programmer's view

Memory Model: Programmer's view

- Data race
  - ★ Concurrent access to memory location, one of which is a write

Memory Model: Programmer's view

- Data race
  - ★ Concurrent access to memory location, one of which is a write
- Sequential consistency (SC)
  - ★ No intra-thread *reordering*, only inter-thread *interleaving*

Memory Model: Programmer's view

- Data race
  - ★ Concurrent access to memory location, one of which is a write
- Sequential consistency (SC)
  - ★ No intra-thread *reordering*, only inter-thread *interleaving*
- **DRF-SC**: primary tool in concurrent programmers arsenal
  - ★ *If a program has no races (under SC semantics), then the program has SC semantics*
  - ★ Well-synchronised programs do not have surprising behaviours

Memory Model: Programmer's view

- Data race
  - ★ Concurrent access to memory location, one of which is a write
- Sequential consistency (SC)
  - ★ No intra-thread *reordering*, only inter-thread *interleaving*
- **DRF-SC**: primary tool in concurrent programmers arsenal
  - ★ *If a program has no races (under SC semantics), then the program has SC semantics*
  - ★ Well-synchronised programs do not have surprising behaviours
- *Our observation: DRF-SC is too weak for programmers*

C/C++ Memory Model

- C/C++ (C11) memory model offers DRF-SC, but..

C/C++ Memory Model

- C/C++ (C11) memory model offers DRF-SC, but..
  - ★ *If a program has races (even benign), then the behaviour is undefined!*

C/C++ Memory Model

- C/C++ (C11) memory model offers DRF-SC, but..
  - ★ *If a program has races (even benign), then the behaviour is undefined!*
  - ★ Most C/C++ programs have races => most C/C++ programs are allowed to *crash and burn*

C/C++ Memory Model

- C/C++ (C11) memory model offers DRF-SC, but..
  - ★ *If a program has races (even benign), then the behaviour is undefined!*
  - ★ Most C/C++ programs have races => most C/C++ programs are allowed to *crash and burn*
- Races on unrelated locations can affect behaviour

C/C++ Memory Model

- C/C++ (C11) memory model offers DRF-SC, but..
  - ★ *If a program has races (even benign), then the behaviour is undefined!*
  - ★ Most C/C++ programs have races => most C/C++ programs are allowed to *crash and burn*
- Races on unrelated locations can affect behaviour
  - ★ We would like a memory model where data races are ***bounded in space***

Java Memory Model

- Java also offers DRF-SC
  - ★ Unlike C++, *type safety* necessitates *defined* behaviour under races

Java Memory Model

- Java also offers DRF-SC
  - ★ Unlike C++, *type safety* necessitates *defined* behaviour under races
  - ★ No data races in ***space***, but allows races in ***time***…

Java Memory Model

- Java also offers DRF-SC
  - ★ Unlike C++, *type safety* necessitates *defined* behaviour under races
  - ★ No data races in ***space***, but allows races in ***time***…

```
int a;
volatile bool flag;
```

Java Memory Model

- Java also offers DRF-SC
  - ★ Unlike C++, *type safety* necessitates *defined* behaviour under races
  - ★ No data races in ***space***, but allows races in ***time***…

```
int a;
volatile bool flag;
```

Thread 1

```
a = 1;
flag = true;
```

Java Memory Model

- Java also offers DRF-SC
  - ★ Unlike C++, *type safety* necessitates *defined* behaviour under races
  - ★ No data races in ***space***, but allows races in ***time***…

```
int a;
volatile bool flag;
```

Thread 1

```
a = 1;
flag = true;
```

Thread 2

```
a = 2;
if (flag) {
  // no race here
  r1 = a;
  r2 = a;
}
```

Java Memory Model

- Java also offers DRF-SC
  - ★ Unlike C++, *type safety* necessitates *defined* behaviour under races
  - ★ No data races in ***space***, but allows races in ***time***…

```
int a;
volatile bool flag;
```

Thread 1

```
a = 1;
flag = true;
```

Thread 2

```
a = 2;
if (flag) {
  // no race here
  r1 = a;
  r2 = a;
}
```

r1 == 1 && r2 == 2 is allowed

Java Memory Model

- Java also offers DRF-SC
  - ★ Unlike C++, *type safety* necessitates *defined* behaviour under races
  - ★ No data races in ***space***, but allows races in ***time***…

```
int a;
volatile bool flag;
```

Thread 1

```
a = 1;
flag = true;
```

Thread 2

```
a = 2;
if (flag) {
  // no race here
  r1 = a;
  r2 = a;
}
```

Races in the past affects future

r1 == 1 && r2 == 2 is allowed

Java Memory Model

- Future data races can affect the past

Java Memory Model

- Future data races can affect the past

```
Class C { int x; }
```

Java Memory Model

- Future data races can affect the past

```
Class C { int x; }
```

Thread 1

```
C c = new C();
c.x = 42;
r1 = c.x;
```

Can assert (r1 == 42) fail?

Java Memory Model

- Future data races can affect the past

```
Class C { int x; }
C g;
```

Thread 1

```
C c = new C();
c.x = 42;
r1 = c.x;
g = c;
```

Thread 2

```
g.x = 7;
```

Can assert (r1 == 42) fail?

Java Memory Model

- Future data races can affect the past

```
Class C { int x; }
C g;
```

Thread 1

```
C c = new C();
c.x = 42;
g = c;
r1 = c.x;
```

Thread 2

```
g.x = 7;
```

Java Memory Model

- Future data races can affect the past

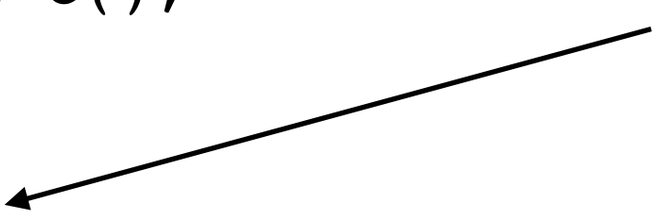
```
Class C { int x; }
C g;
```

Thread 1

```
C c = new C();
c.x = 42;
g = c;
r1 = c.x;
```

Thread 2

```
g.x = 7;
```



assert (r1 == 42) fails

Java Memory Model

- Future data races can affect the past

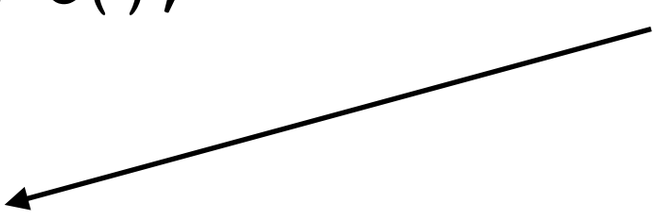
```
Class C { int x; }
C g;
```

Thread 1

```
C c = new C();
c.x = 42;
g = c;
r1 = c.x;
```

Thread 2

```
g.x = 7;
```



assert (r1 == 42) fails

- We would like a memory model that bounds data races in ***time***

OCaml Memory Model: Goal

OCaml Memory Model: Goal

- Language memory models should specify behaviours under data races

OCaml Memory Model: Goal

- Language memory models should specify behaviours under data races
  - ★ Not because they are useful

OCaml Memory Model: Goal

- Language memory models should specify behaviours under data races
  - ★ Not because they are useful
  - ★ But to limit their damage

OCaml Memory Model: Goal

- Language memory models should specify behaviours under data races
  - ★ Not because they are useful
  - ★ But to limit their damage

If I read a variable twice and there are no concurrent writes, then both reads return the same value

OCaml MM: Contributions

- Memory Model Desiderata
  - ★ Not too weak (good for programmers)
  - ★ Not too strong (good for hardware)
  - ★ Admits optimisations (good for compilers)
  - ★ Mathematically rigorous (good for verification)
- OCaml Memory model
  - ★ *Local version of DRF-SC — key discovery*
  - ★ Free on x86, 0.6% overhead on ARM, 2.6% overhead on POWER
  - ★ Allows most common compiler optimisations
  - ★ Simple operational and axiomatic semantics + proved soundness (optimization + to-hardware)

Local DRF

Local DRF

- If there are no data races,

Local DRF

- If there are no data races,
  - ★ *on some variables (space)*

Local DRF

- If there are no data races,
  - ★ *on some variables (space)*
  - ★ *in some interval (time)*

Local DRF

- If there are no data races,
  - ★ *on some variables (space)*
  - ★ *in some interval (time)*
  - ★ then the program has SC behaviour *on those variables in that time interval*

Local DRF

- If there are no data races,
  - ★ *on some variables (space)*
  - ★ *in some interval (time)*
  - ★ then the program has SC behaviour *on those variables in that time interval*
- Space = {all variables} && Time = whole execution => DRF-SC

Local DRF

- If there are no data races,
  - ★ *on some variables (space)*
  - ★ *in some interval (time)*
  - ★ then the program has SC behaviour *on those variables in that time interval*
- Space = {all variables} && Time = whole execution => DRF-SC

Flag is atomic

Thread 1

```
msg = 1;
b = 0;
Flag = 1;
```

Thread 2

```
b = 1;
if (Flag) {
  r = msg;
}
```

Local DRF

- If there are no data races,
  - ★ *on some variables (space)*
  - ★ *in some interval (time)*
  - ★ then the program has SC behaviour *on those variables in that time interval*
- Space = {all variables} && Time = whole execution => DRF-SC

Flag is atomic

Thread 1

```
msg = 1;
b = 0;
Flag = 1;
```

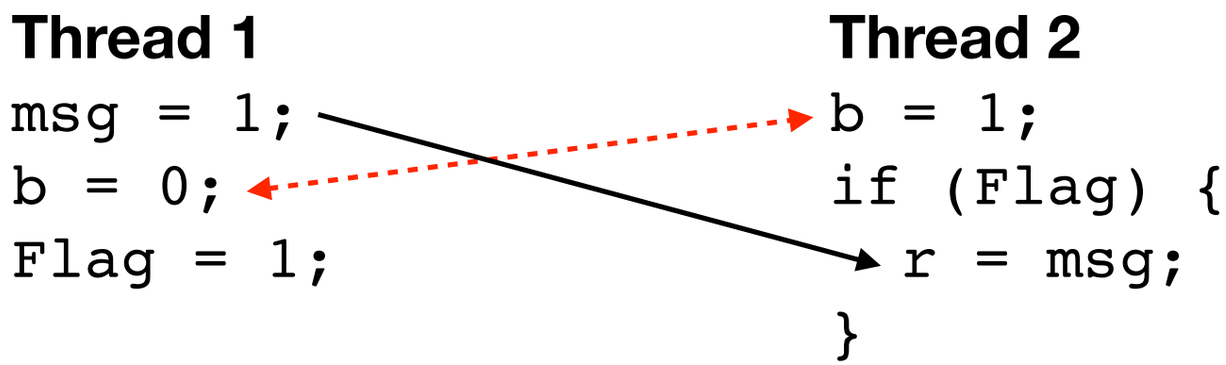
Thread 2

```
b = 1;
if (Flag) {
  r = msg;
}
```

Local DRF

- If there are no data races,
  - ★ *on some variables (space)*
  - ★ *in some interval (time)*
  - ★ then the program has SC behaviour *on those variables in that time interval*
- Space = {all variables} && Time = whole execution => DRF-SC

Flag is atomic



Due to local DRF, despite the race on **b**, message-passing idiom still works!

Formal Memory Model

Formal Memory Model

- Most programmers can live with local DRF
  - ★ Experts demand more (concurrency libraries, high-performance code, etc.)

Formal Memory Model

- Most programmers can live with local DRF
  - ★ Experts demand more (concurrency libraries, high-performance code, etc.)
- Simple *operational semantics* that captures all of the allowed behaviours

Formal Memory Model

- Most programmers can live with local DRF
  - ★ Experts demand more (concurrency libraries, high-performance code, etc.)
- Simple *operational semantics* that captures all of the allowed behaviours

(SILENT) $$\frac{e \overset{\epsilon}{\rightsquigarrow} e'}{\langle S, P[i \mapsto (F, e)]\rangle \rightarrow \langle S, P[i \mapsto (F, e')]\rangle}$$

(MEMORY) $$\frac{e \overset{\ell:\phi}{\rightsquigarrow} e' \qquad S(\ell); F \overset{\ell:\phi}{\longrightarrow\!\!\!\!\!\rightarrow} C'; F'}{\langle S, P[i \mapsto (F, e)]\rangle \rightarrow \langle S[\ell \mapsto C'], P[i \mapsto (F', e')]\rangle}$$

(b) Machine steps

(READ-NA) $H; F \xrightarrow{\text{a: read } H(t)} H; F$ if $F(\mathsf{a}) \leq t, t \in \mathrm{dom}(H)$

(WRITE-NA) $H; F \xrightarrow{\text{a: write } x} H[t \mapsto x]; F[\mathsf{a} \mapsto t]$ if $F(\mathsf{a}) < t, t \notin \mathrm{dom}(H)$

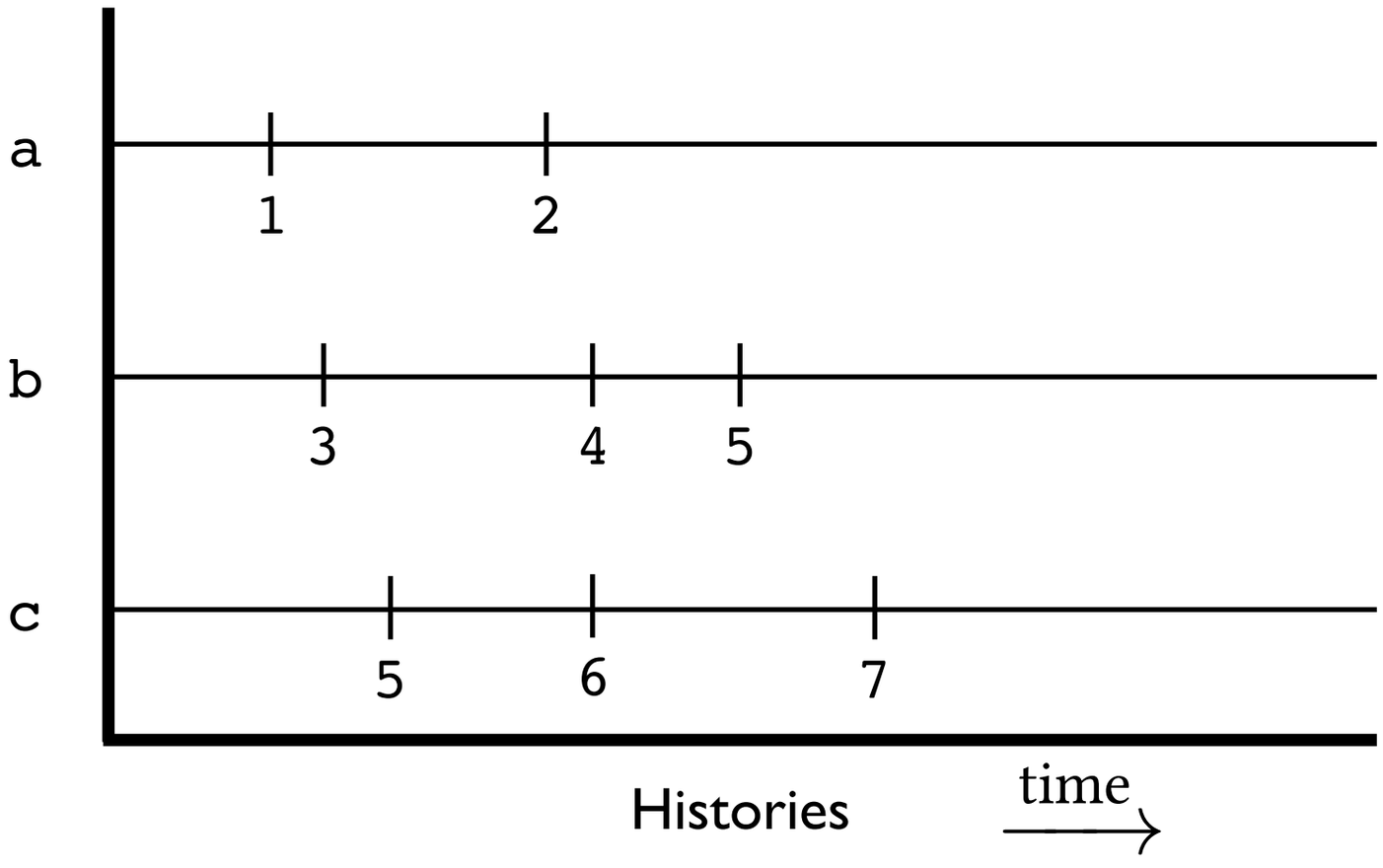
(READ-AT) $(F_{\mathsf{A}}, x); F \xrightarrow{\text{A: read } x} (F_{\mathsf{A}}, x); F_{\mathsf{A}} \sqcup F$

(WRITE-AT) $(F_{\mathsf{A}}, y); F \xrightarrow{\text{A: write } x} (F_{\mathsf{A}} \sqcup F, x); F_{\mathsf{A}} \sqcup F$

(c) Memory operations

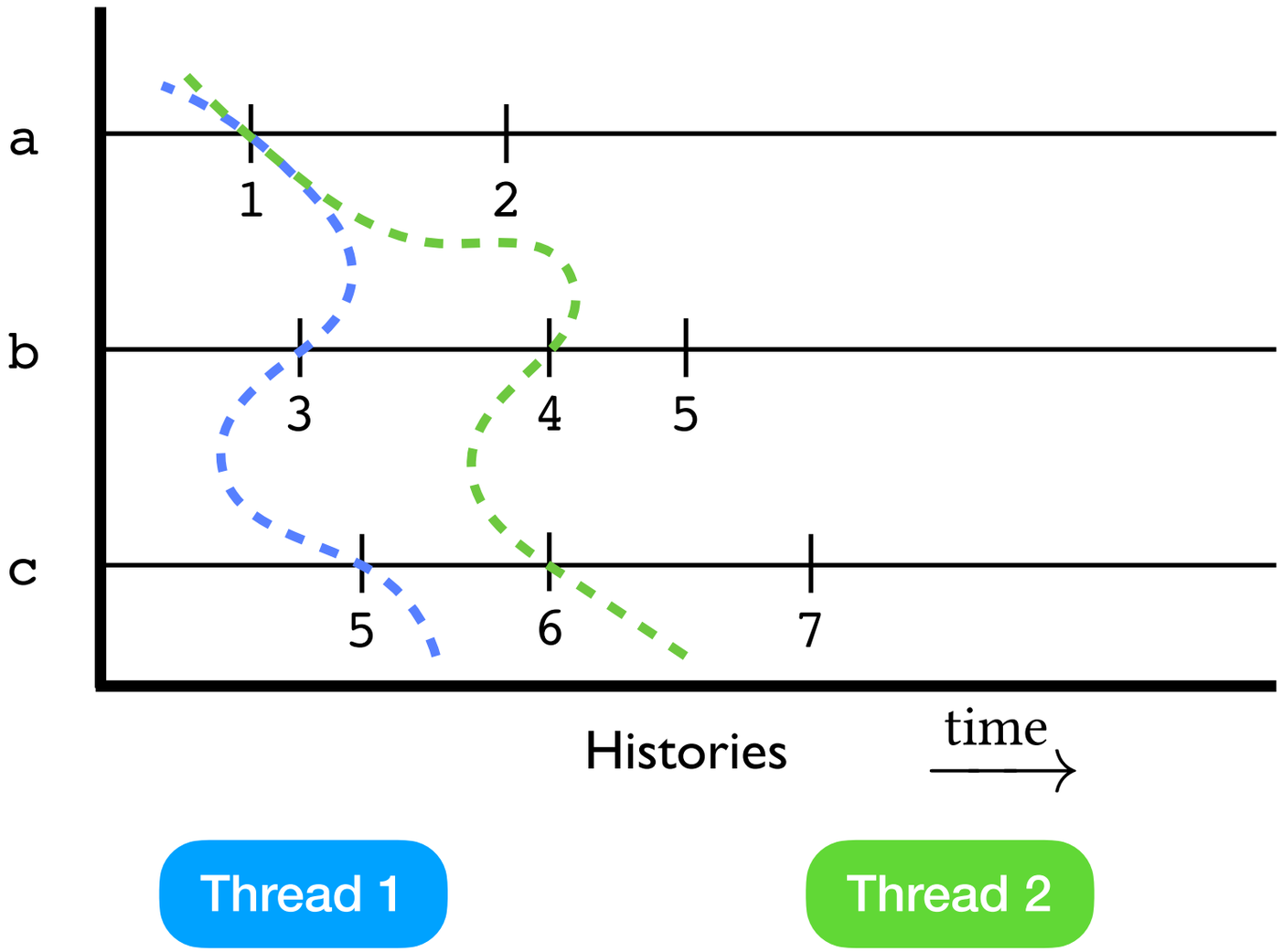
Visualising operational semantics

Non atomic



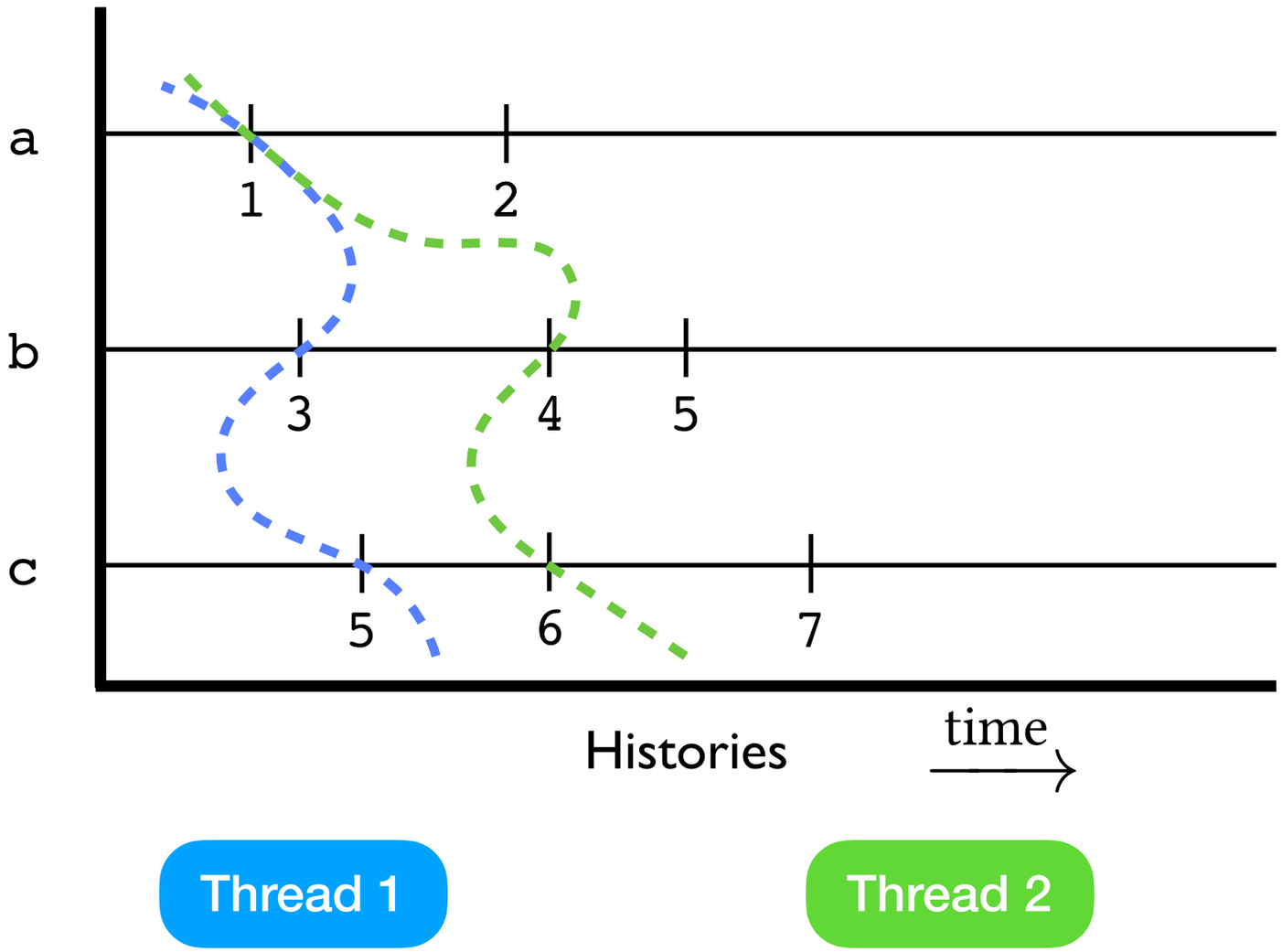
Visualising operational semantics

Non atomic



Visualising operational semantics

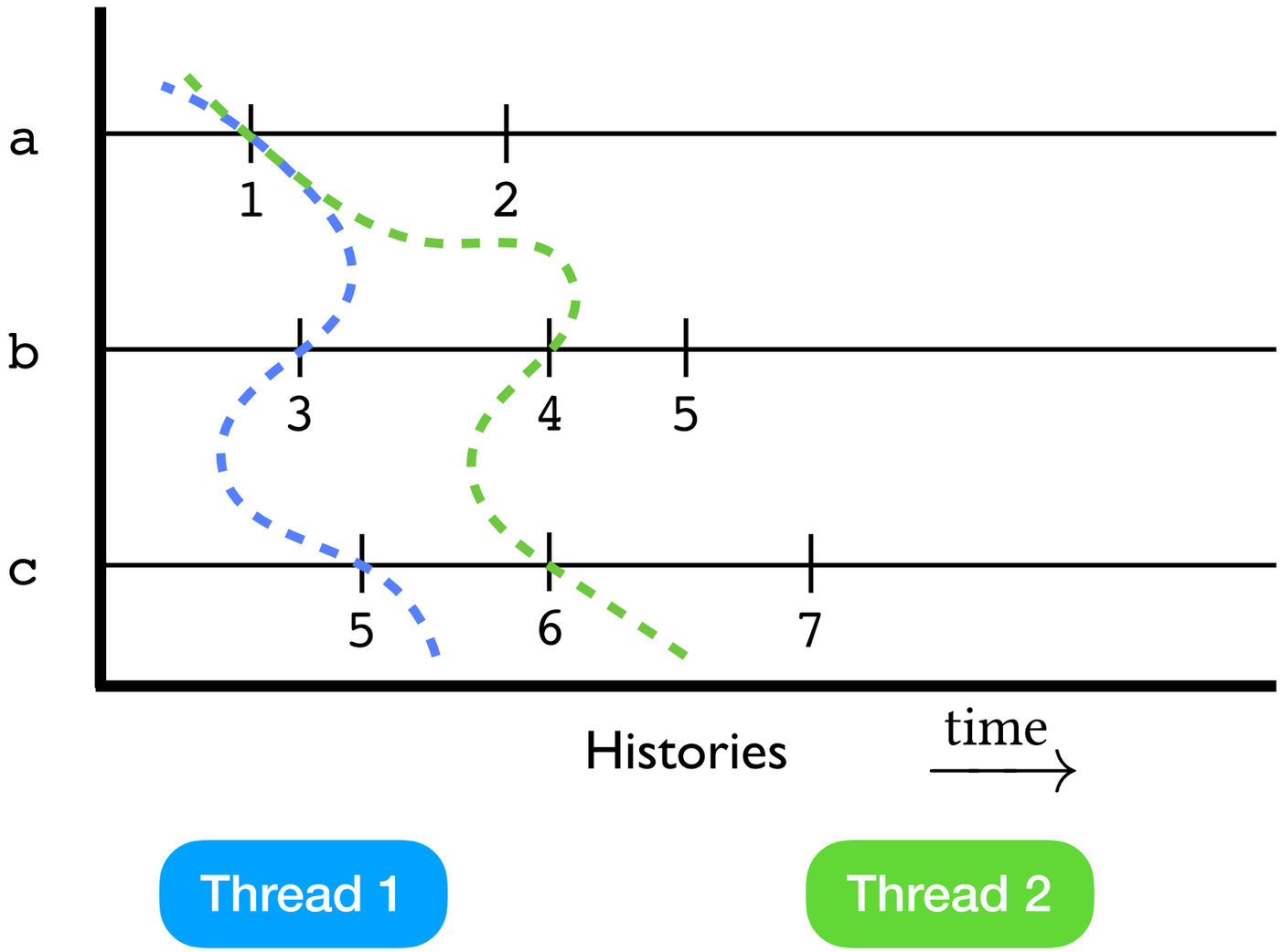
Non atomic




```
read(b)
```

Visualising operational semantics

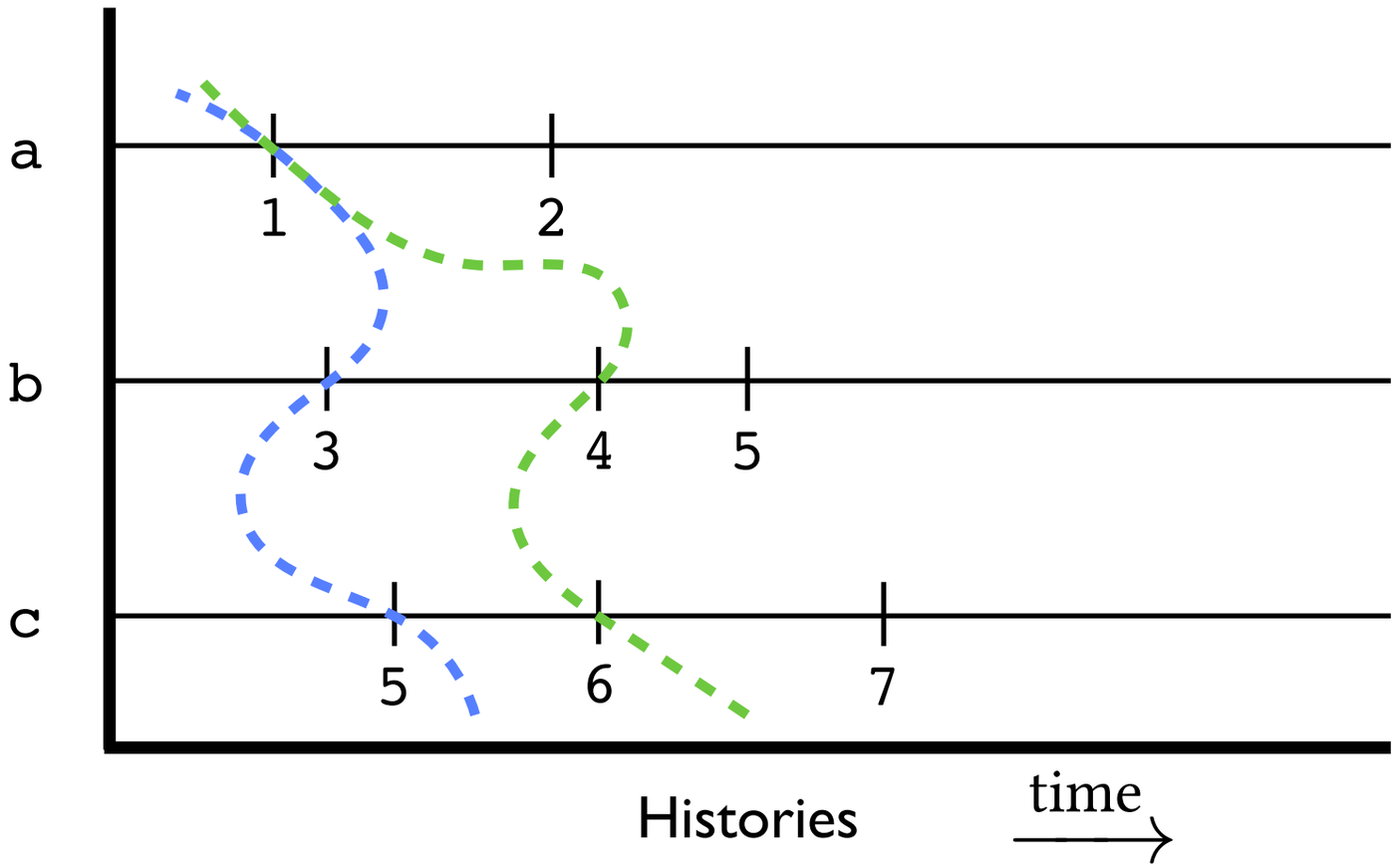
Non atomic




```
read(b) -> 3/4/5
```

Visualising operational semantics

Non atomic



Thread 1

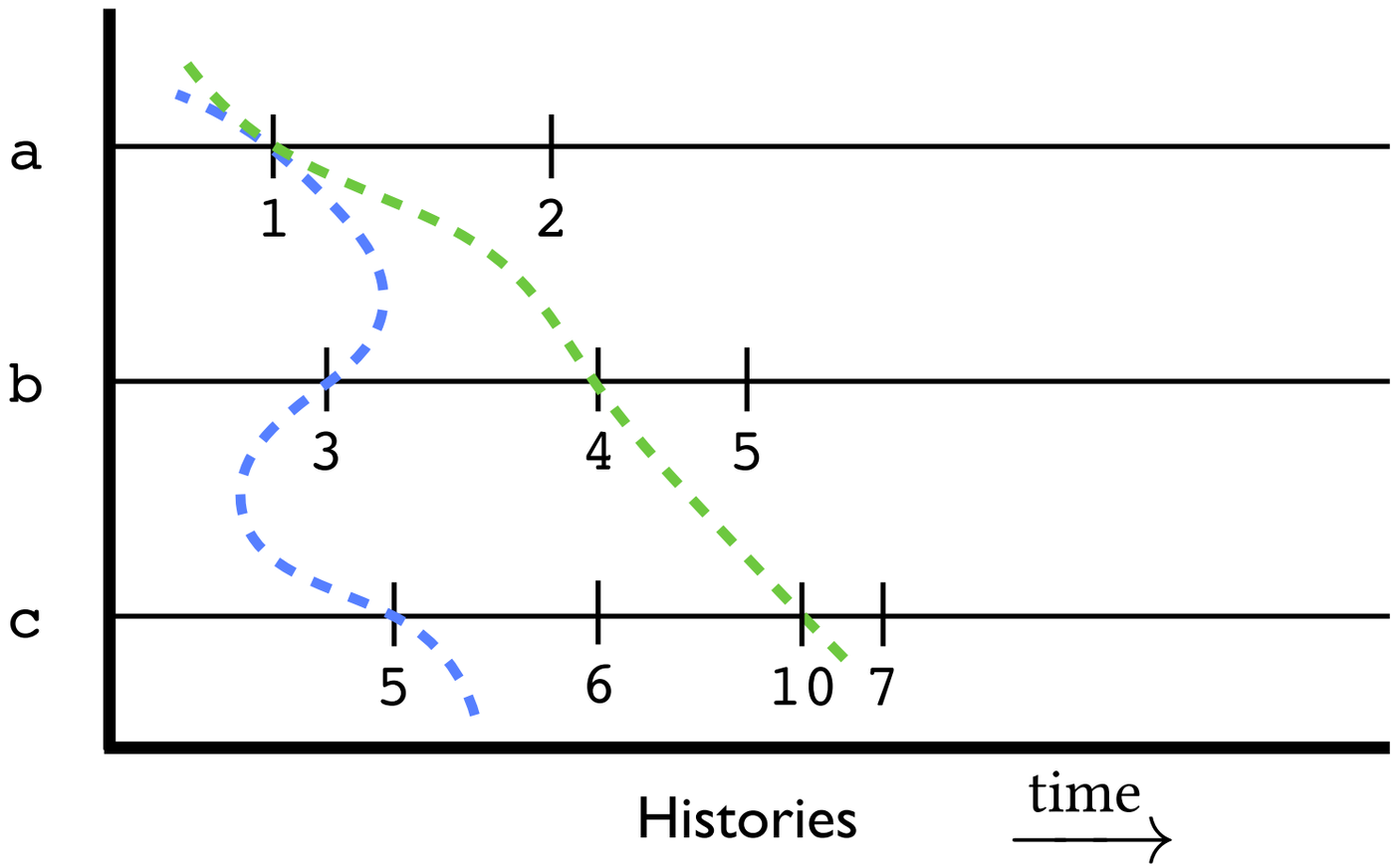
Thread 2

```
read(b) -> 3/4/5
```

```
write(c,10)
```

Visualising operational semantics

Non atomic



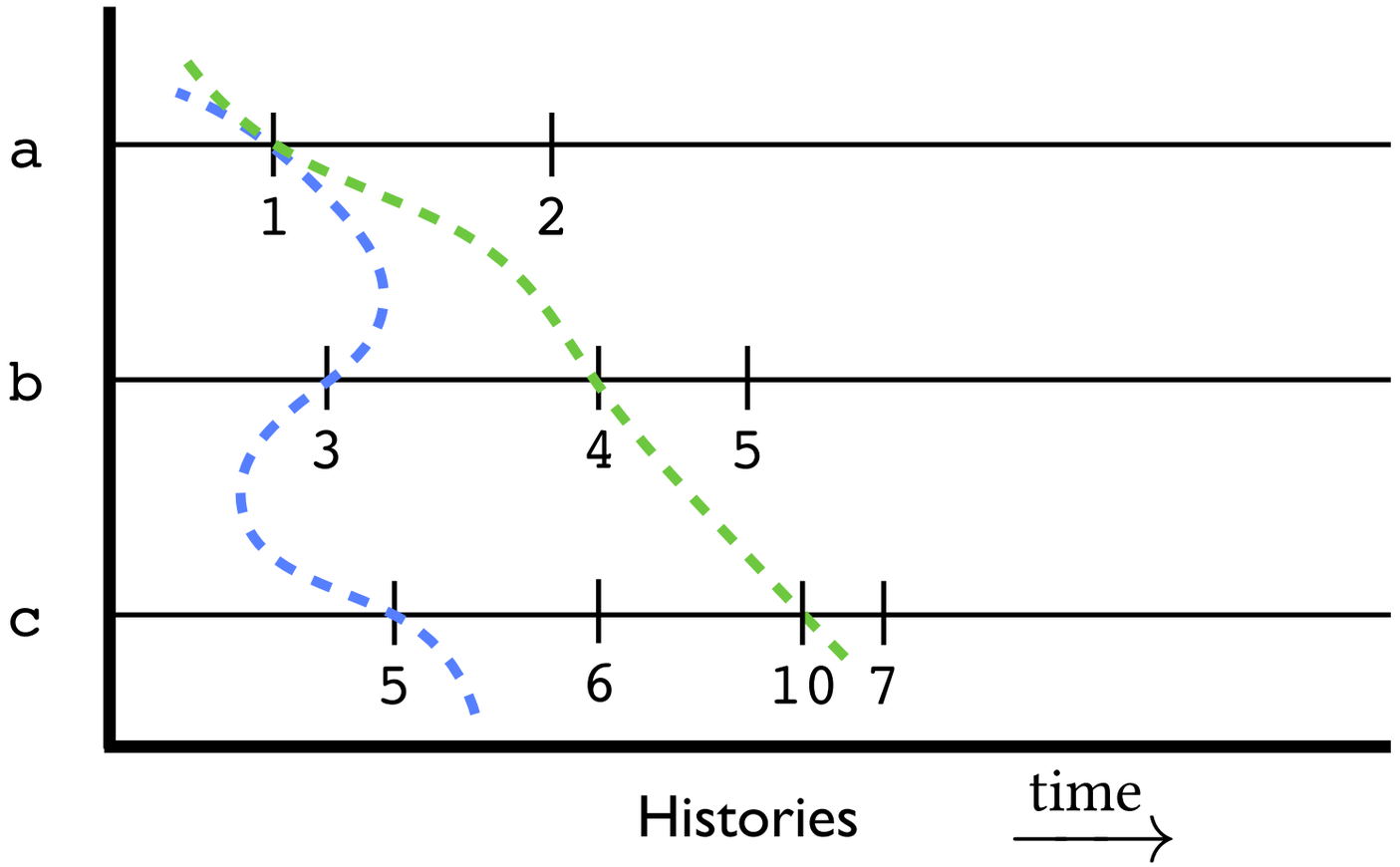
Thread 1

Thread 2

```
read(b) -> 3/4/5        write(c,10)
```

Visualising operational semantics

Non atomic



Atomic

A 10

B 5

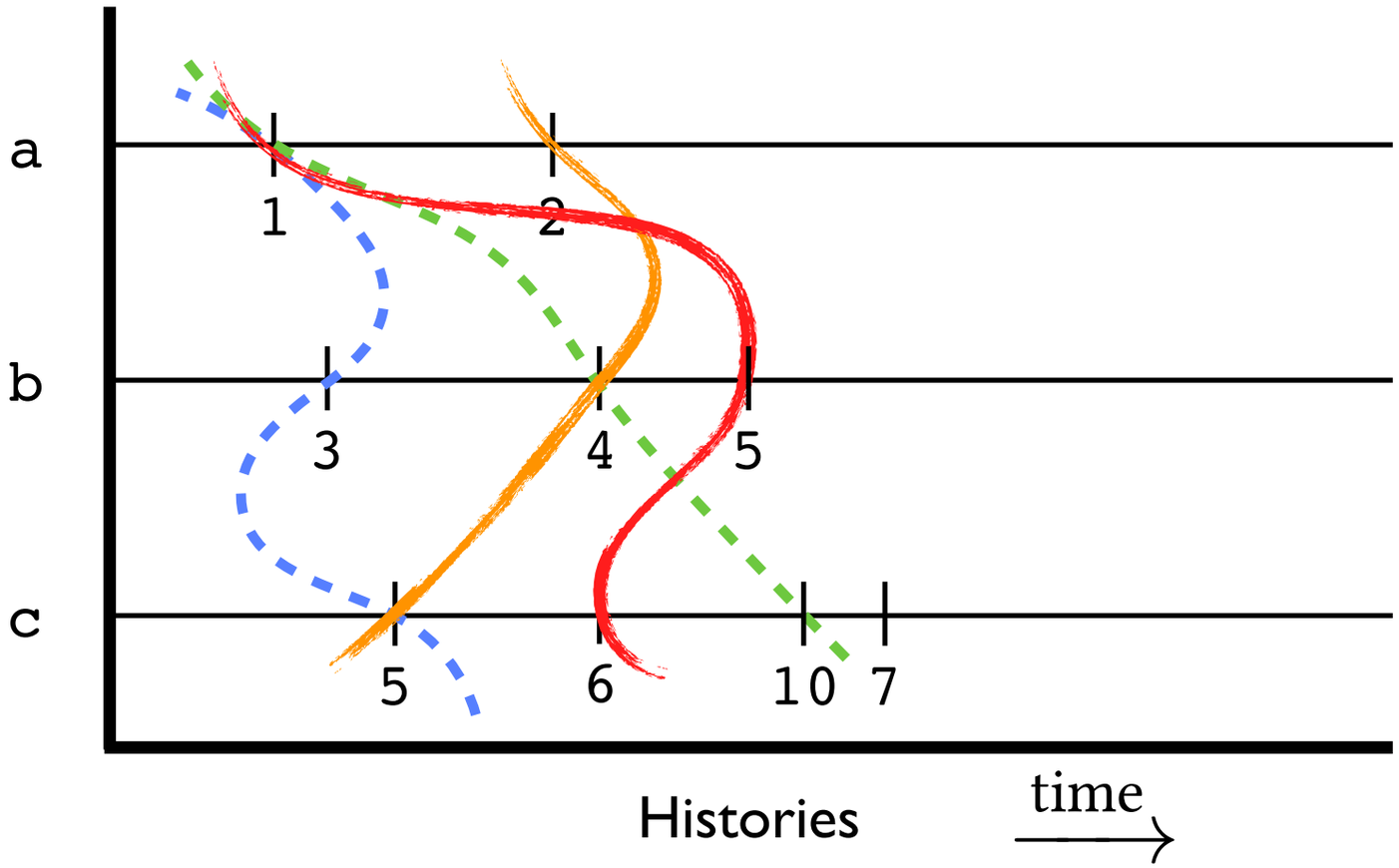
Thread 1

Thread 2

```
read(b) -> 3/4/5        write(c,10)
```

Visualising operational semantics

Non atomic



Atomic

A 10

B 5

Thread 1

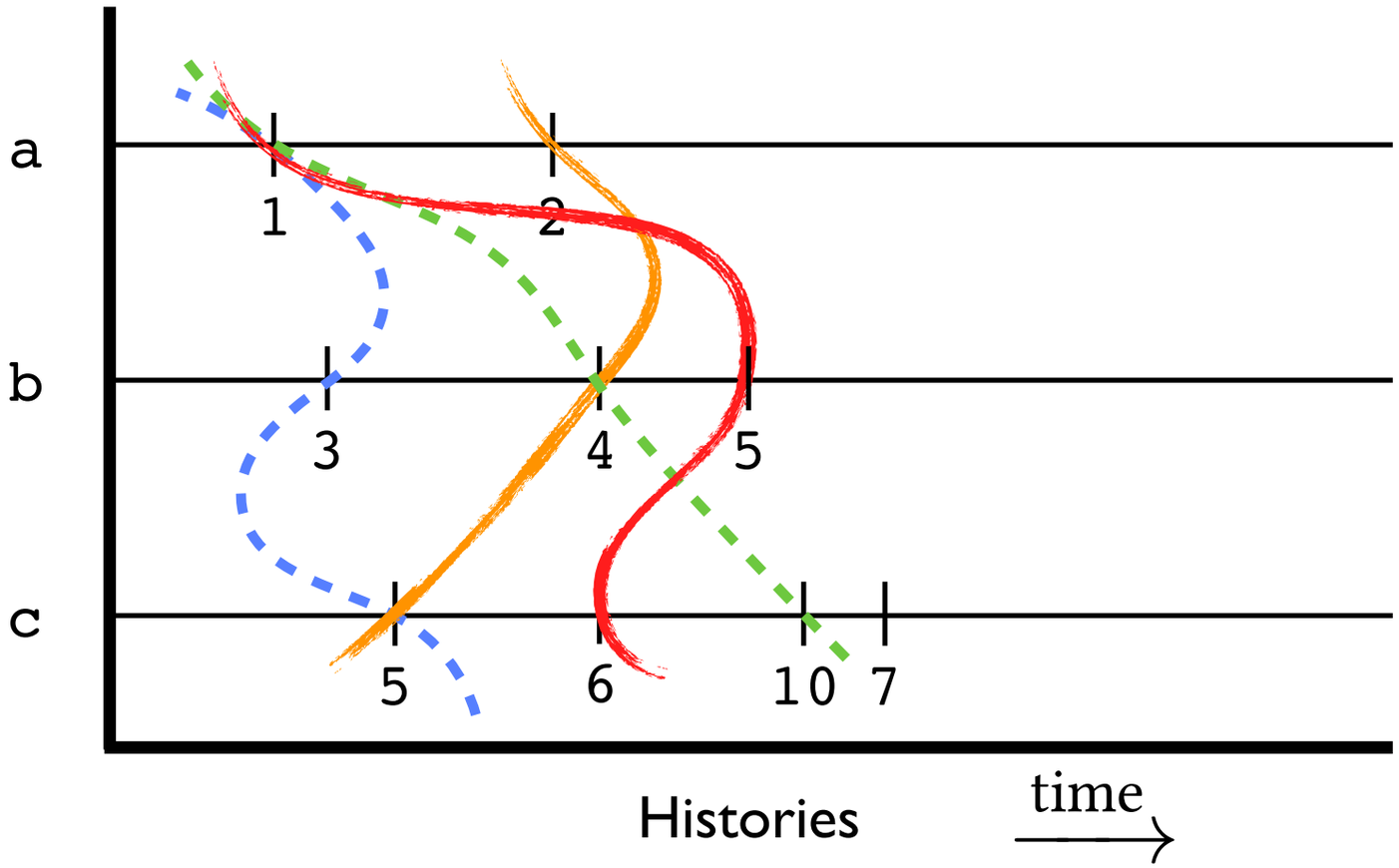
Thread 2

```
read(b) -> 3/4/5
```

```
write(c,10)
```

Visualising operational semantics

Non atomic



Thread 1

Thread 2

```
read(B)
```

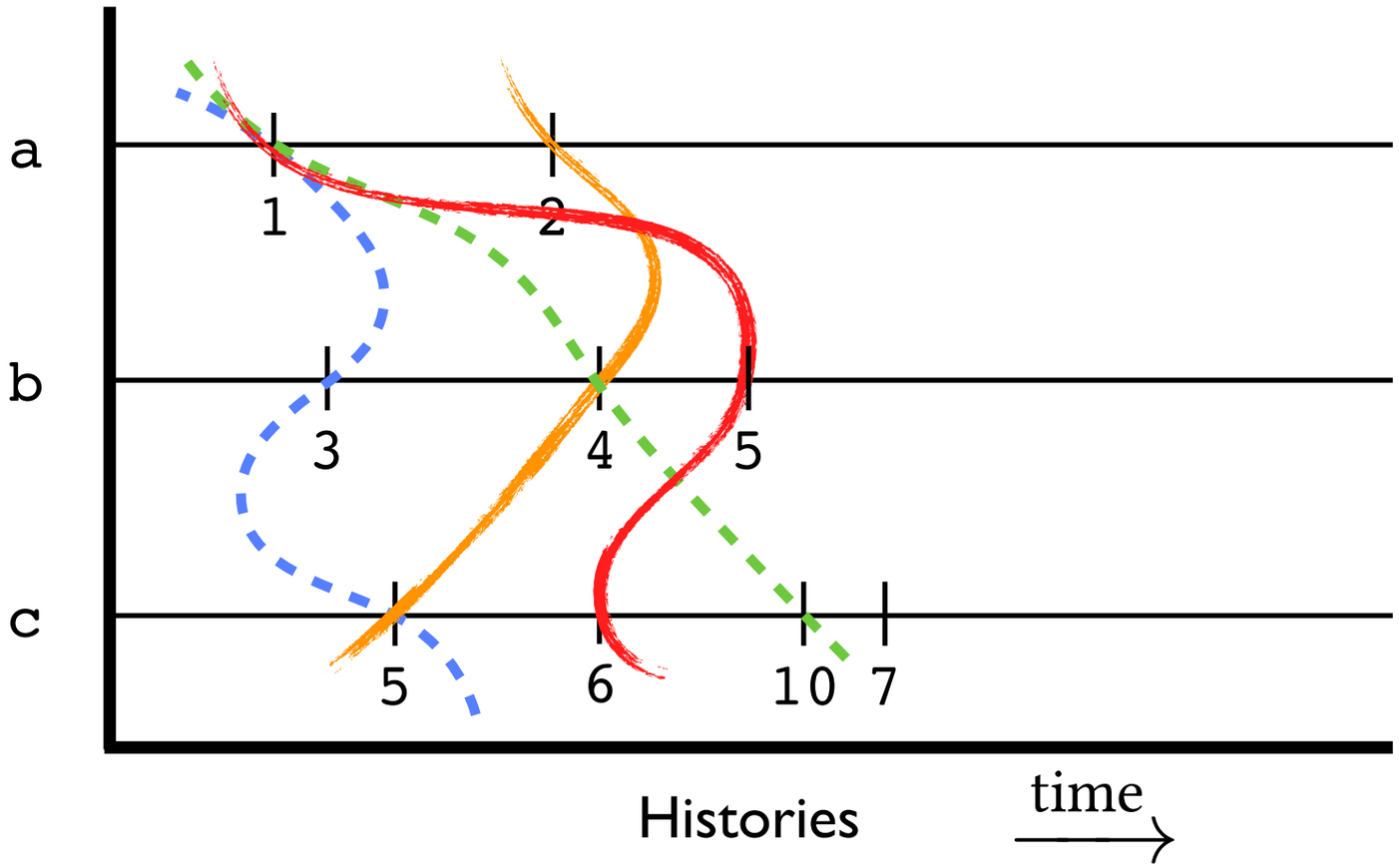
Atomic

A 10

B 5

Visualising operational semantics

Non atomic



Atomic

A 10

B 5

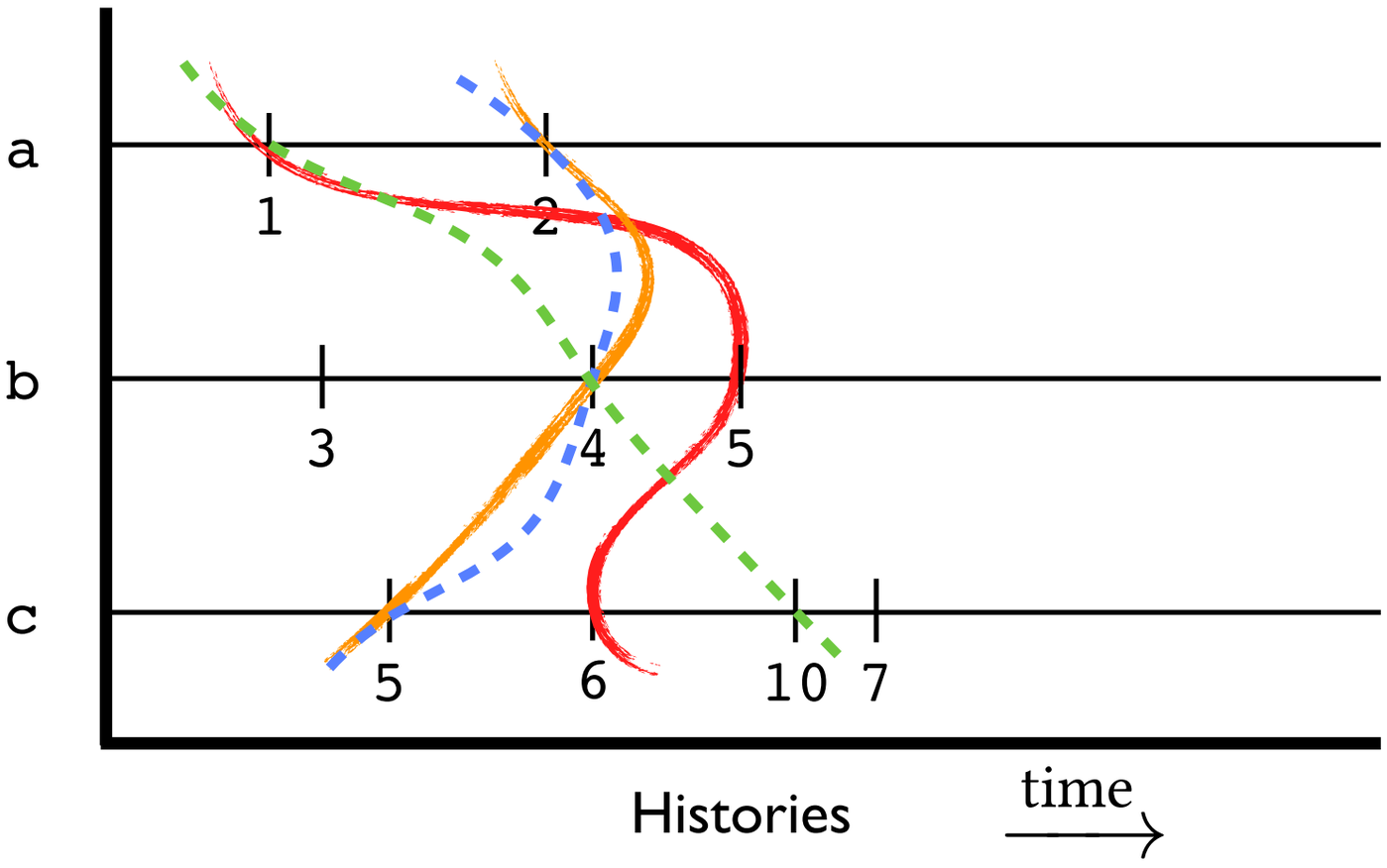
Thread 1

Thread 2

```
read(B) -> 5
```

Visualising operational semantics

Non atomic



Thread 1

Thread 2

```
read(B) -> 5
```

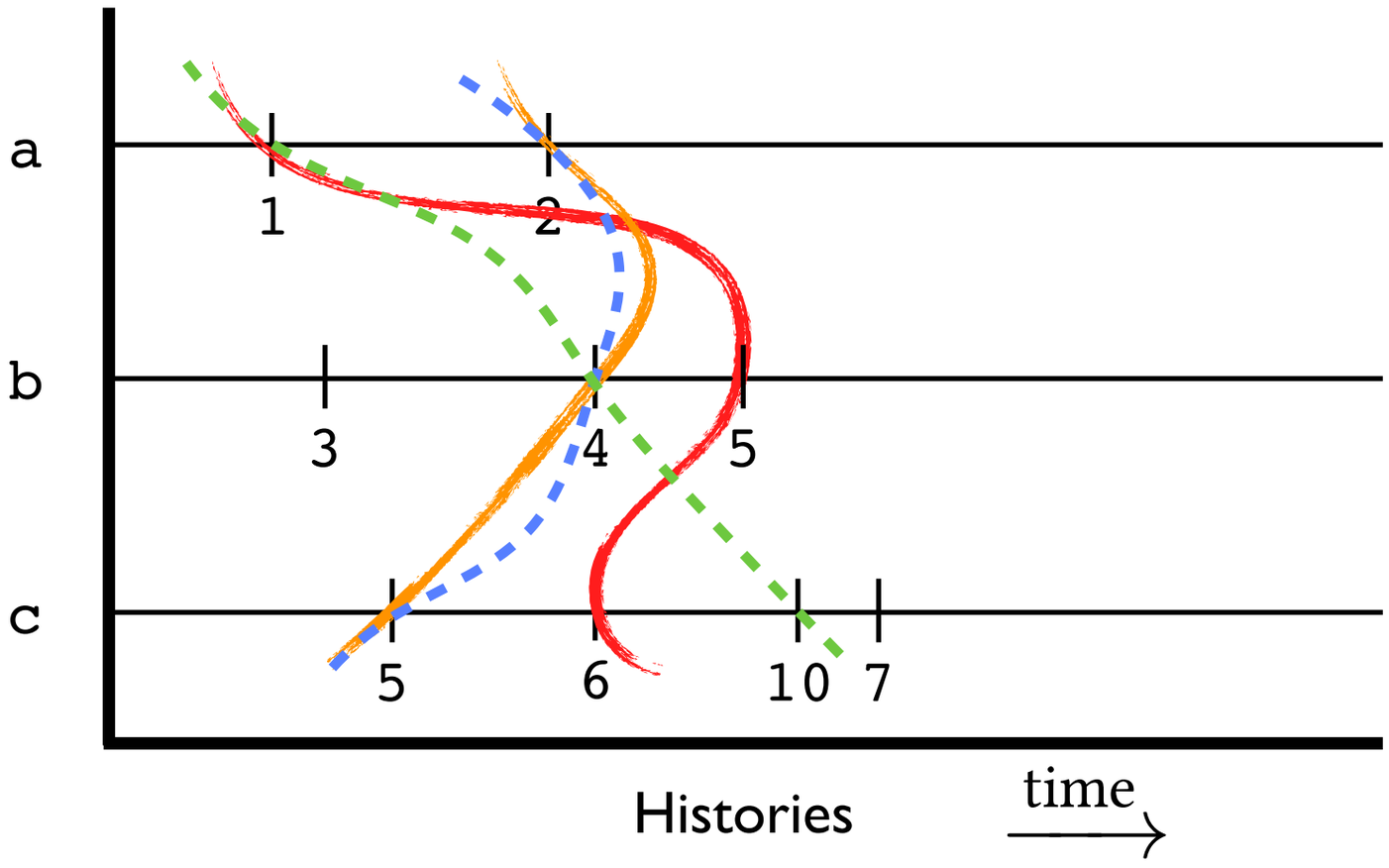
Atomic

A 10

B 5

Visualising operational semantics

Non atomic



Atomic

A 10

B 5

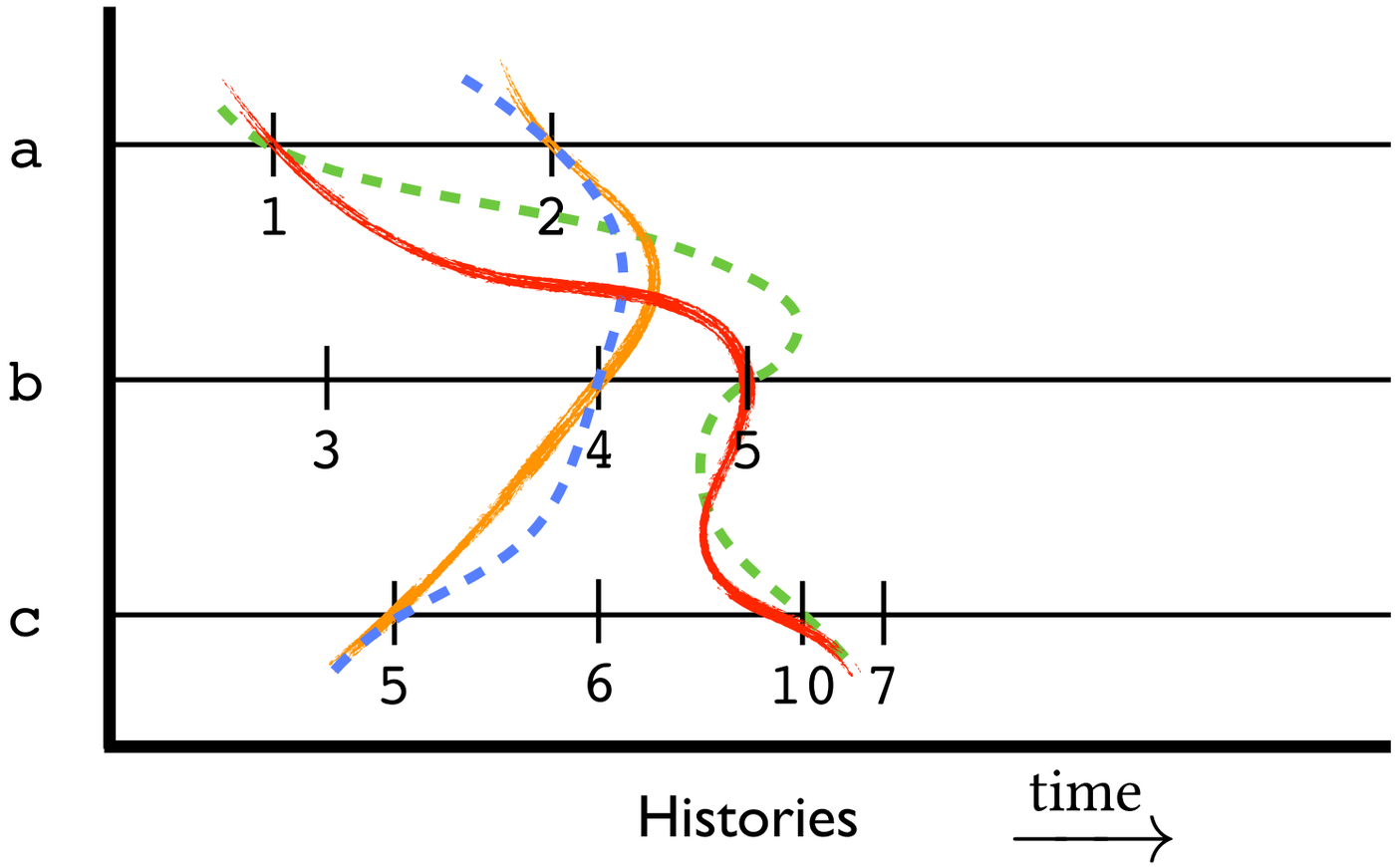
Thread 1

Thread 2

```
read(B) -> 5          write (A,20)
```

Visualising operational semantics

Non atomic



Atomic

A 20

B 5

Thread 1

```
read(B) -> 5
```

Thread 2

```
write (A,20)
```

Formalizing Local DRF

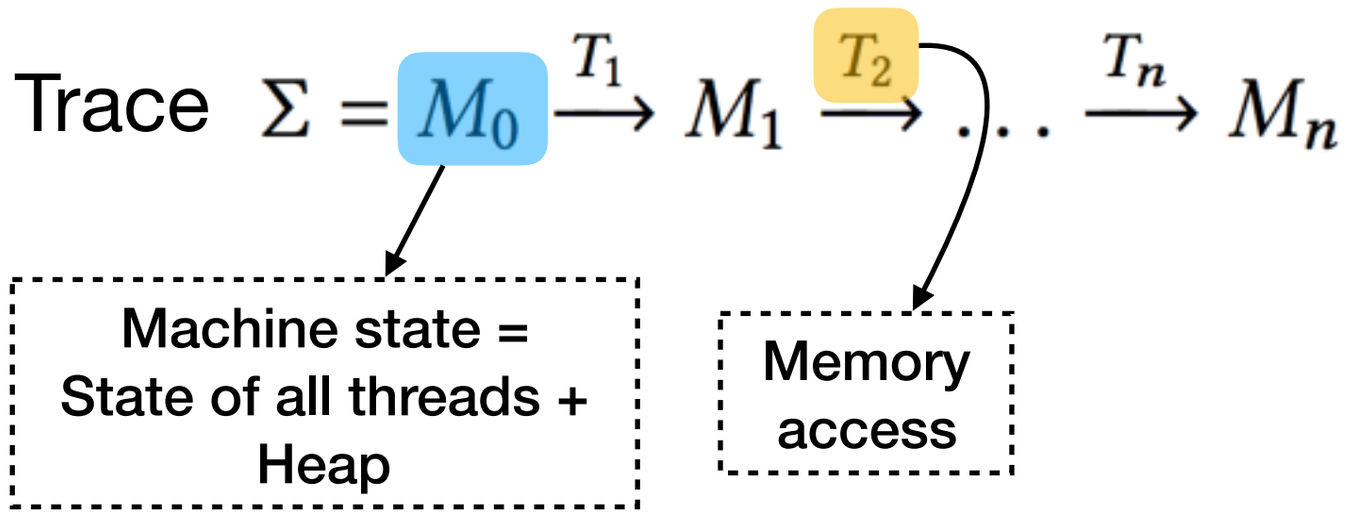
Trace $\Sigma = M_0 \xrightarrow{T_1} M_1 \xrightarrow{T_2} \ldots \xrightarrow{T_n} M_n$

Formalizing Local DRF

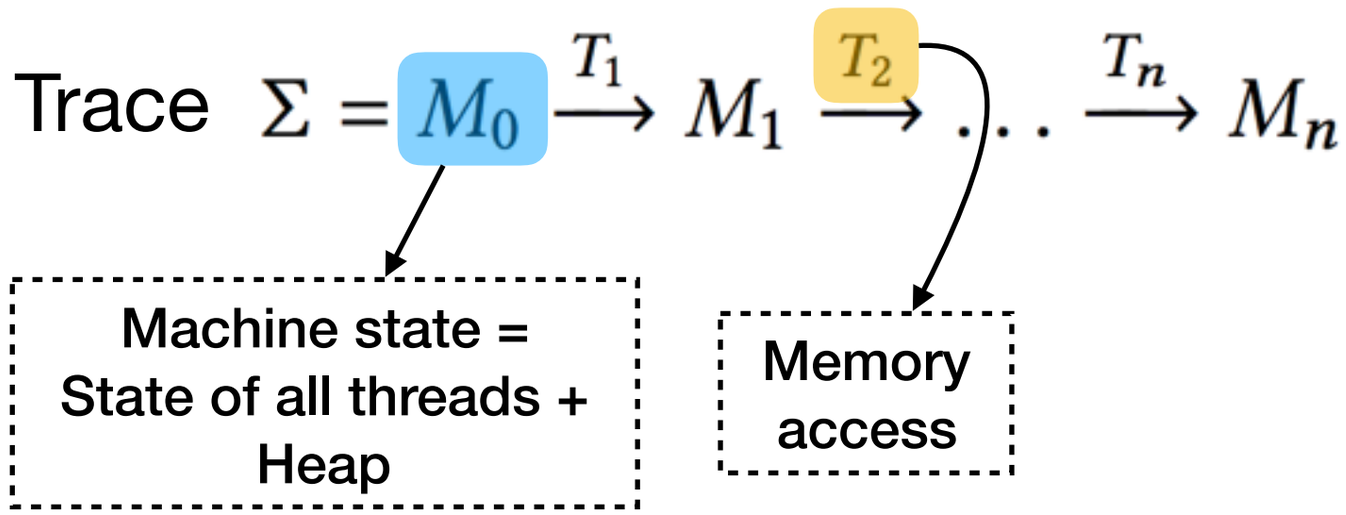
Trace $\Sigma = M_0 \xrightarrow{T_1} M_1 \xrightarrow{T_2} \ldots \xrightarrow{T_n} M_n$

Machine state =
State of all threads +
Heap

Formalizing Local DRF

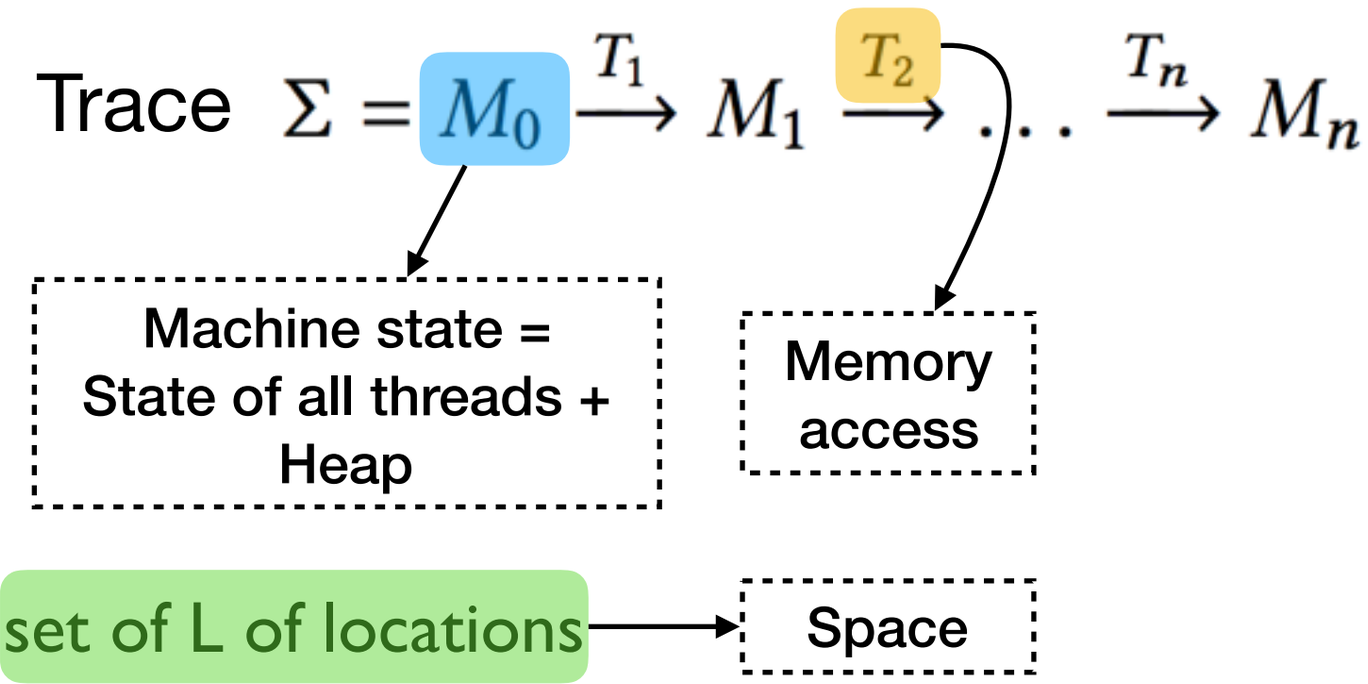


Formalizing Local DRF



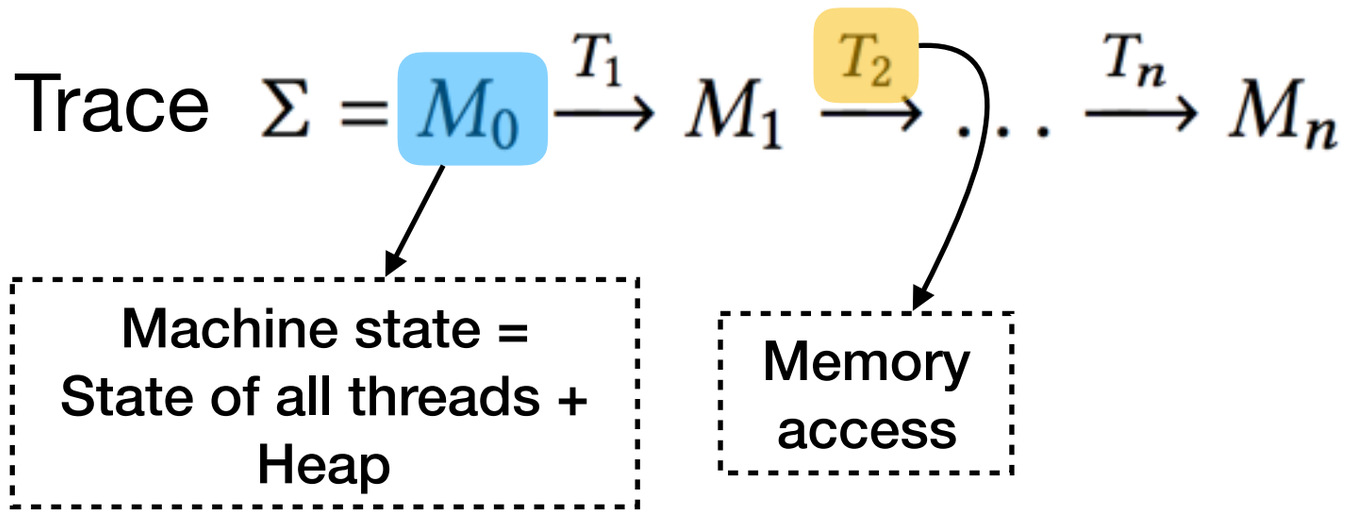
- Pick a set of L of locations

Formalizing Local DRF



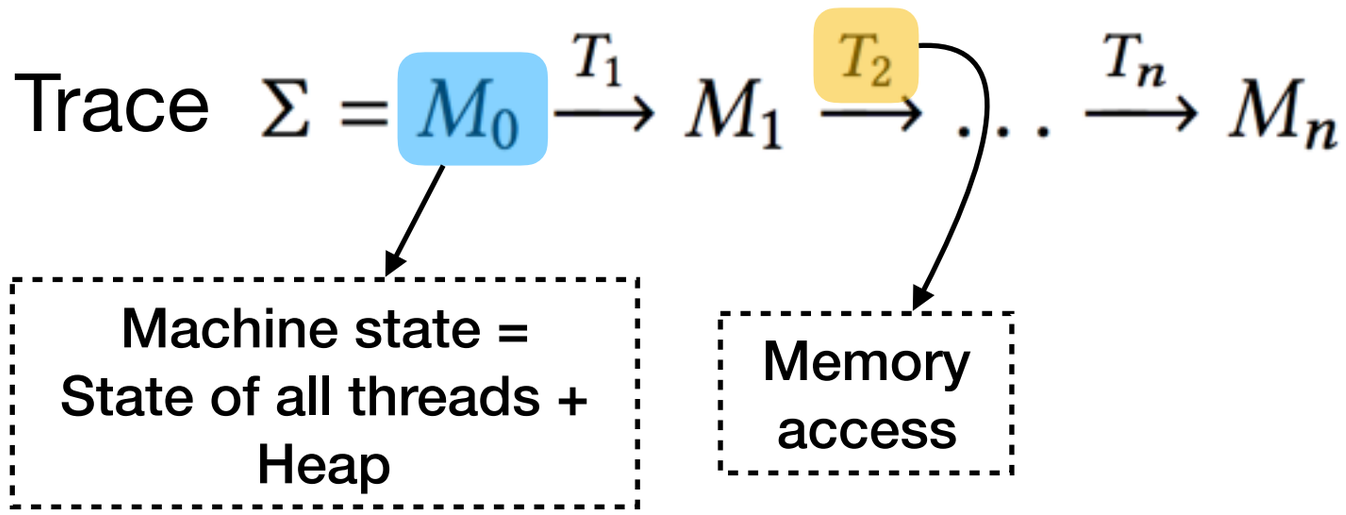
- Pick a set of L of locations → Space

Formalizing Local DRF



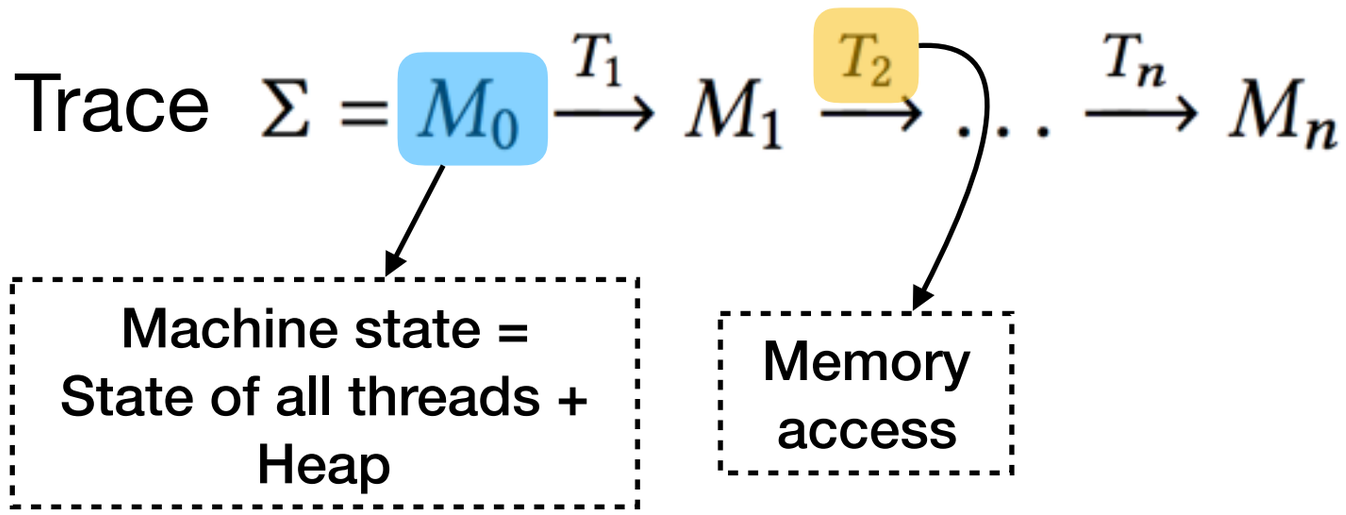
- Pick a set of L of locations → Space
- Pick a machine state M where there are no ongoing races in L
  - ★ M is said to be L-stable

Formalizing Local DRF



- Pick a set of L of locations → Space
- Pick a machine state M where there are no ongoing races in L
  - ★ M is said to be L-stable
- Local DRF Theorem
  - ★ *Starting from an L-stable state M, until the next race on any location in L under SC semantics, the program has SC semantics*

Formalizing Local DRF



- Pick a set of L of locations → Space
- Pick a machine state M where there are no ongoing races in L
  - ★ M is said to be L-stable
- Local DRF Theorem
  - ★ *Starting from an L-stable state M, until the next race on any location in L under SC semantics, the program has SC semantics*

Time

Performance Implication

- Local DRF prohibits certain hardware and software optimisations
  - ★ Preserve load-to-store ordering

Performance Implication

- Local DRF prohibits certain hardware and software optimisations
  - ★ Preserve load-to-store ordering
- No compiler optimisation that reorders load-to-store ordering is allowed

Performance Implication

- Local DRF prohibits certain hardware and software optimisations
  - ★ Preserve load-to-store ordering
- No compiler optimisation that reorders load-to-store ordering is allowed

```
r1 = a;
b = c;      ——Redundant store elimination——→
a = r1;
```

```
r1 = a;
b = c;
;
```

Performance Implication

- Local DRF prohibits certain hardware and software optimisations
  - ★ Preserve load-to-store ordering
- No compiler optimisation that reorders load-to-store ordering is allowed

Performance Implication

- Local DRF prohibits certain hardware and software optimisations
  - ★ Preserve load-to-store ordering
- No compiler optimisation that reorders load-to-store ordering is allowed

Performance Implication

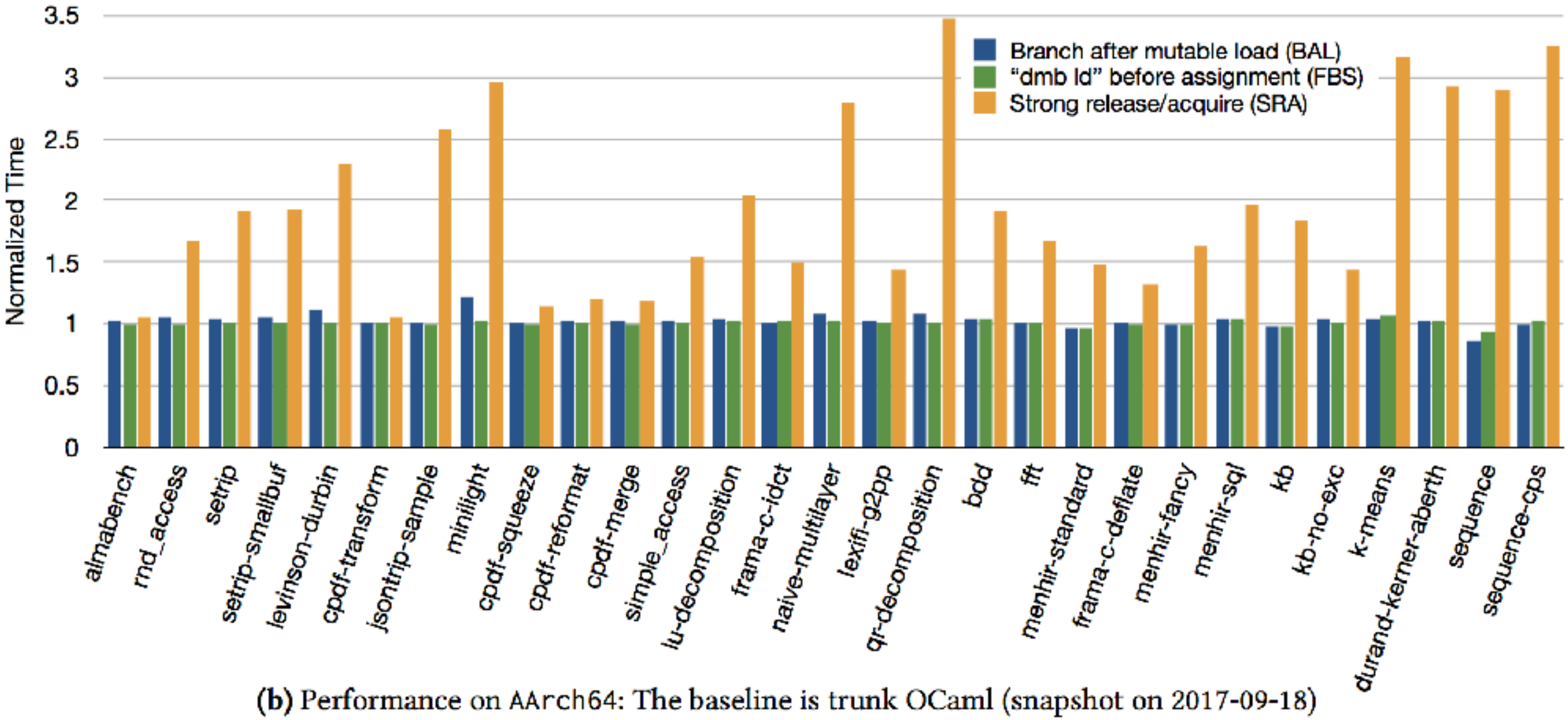
- Local DRF prohibits certain hardware and software optimisations
  - ★ Preserve load-to-store ordering
- No compiler optimisation that reorders load-to-store ordering is allowed

Performance Implication

- Local DRF prohibits certain hardware and software optimisations
  - ★ Preserve load-to-store ordering
- No compiler optimisation that reorders load-to-store ordering is allowed

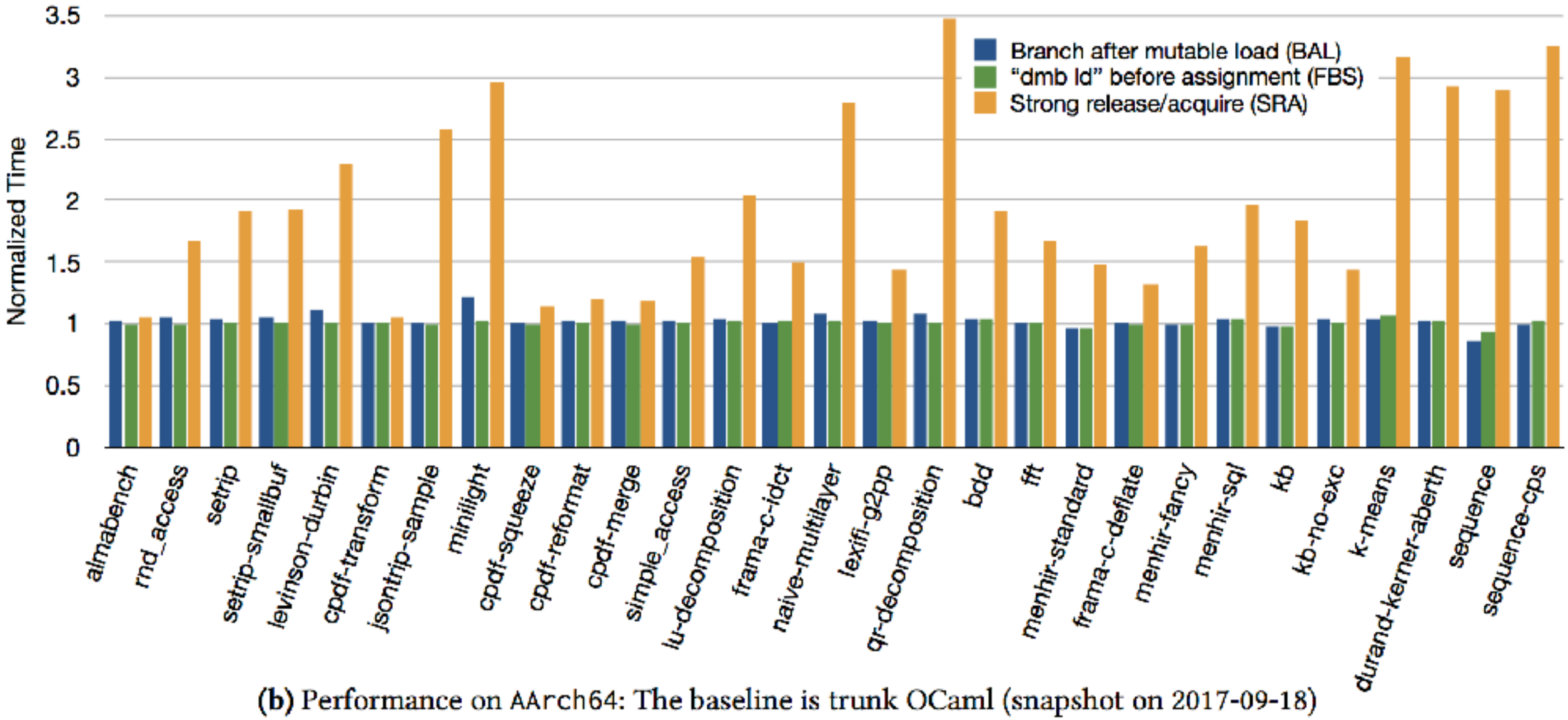
- ARM & POWER do not preserve load-to-store ordering
  - ★ Insert necessary synchronisation between every mutable load and store
  - ★ *What is the performance cost?*

Performance



(b) Performance on AArch64: The baseline is trunk OCaml (snapshot on 2017-09-18)

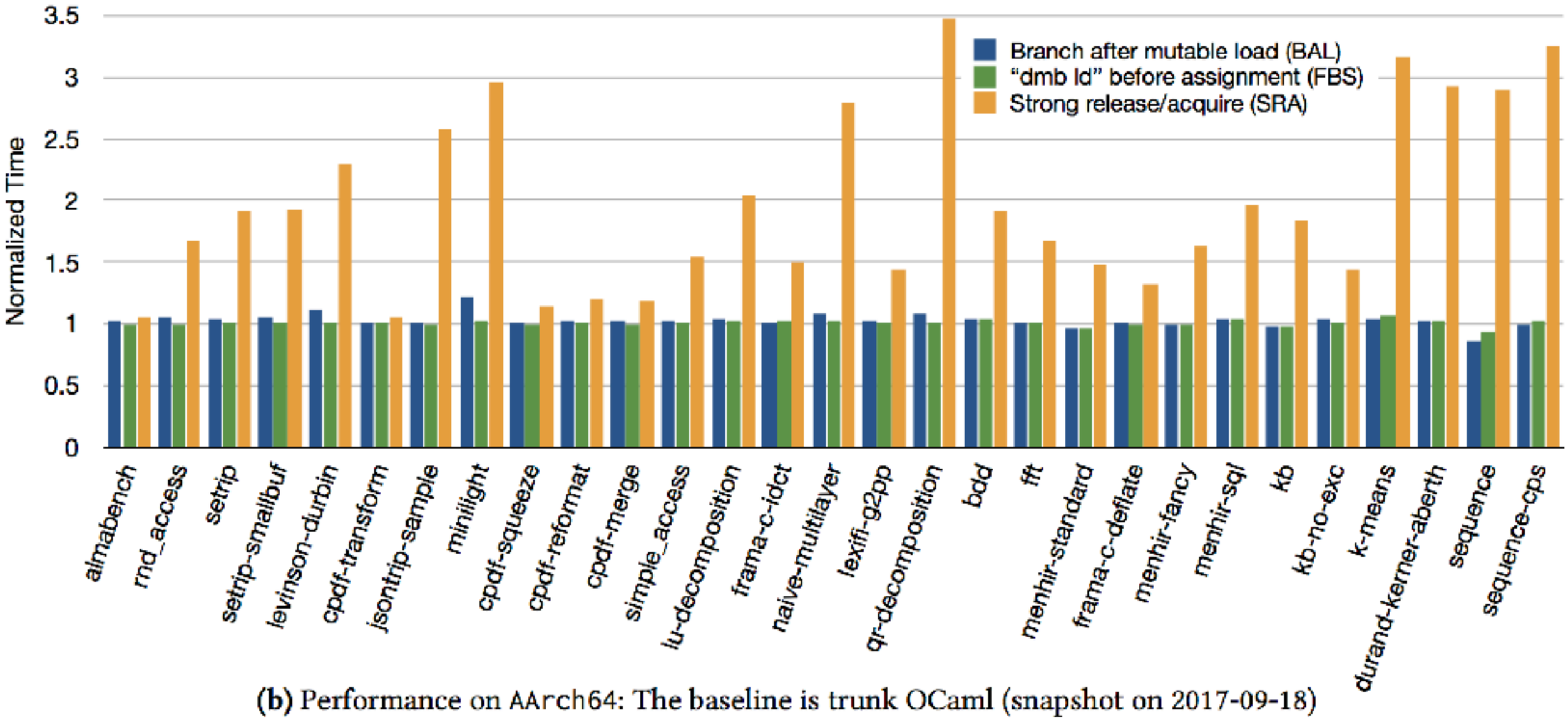
Performance



(b) Performance on AArch64: The baseline is trunk OCaml (snapshot on 2017-09-18)

0.6% overhead on AArch64 (ARMv8)

Performance



(b) Performance on AArch64: The baseline is trunk OCaml (snapshot on 2017-09-18)

0.6% overhead on AArch64 (ARMv8)

Free on x86, 2.6% on POWER

Summary

- OCaml memory model
  - ★ Balances *comprehensibility* (Local DRF theorem) and *Performance* (free on x86, 0.6% on ARMv8, 2.6% on POWER)
  - ★ Allows common compiler optimisations
  - ★ Compilation + Optimisations proved sound

Summary

- OCaml memory model
  - ★ Balances *comprehensibility* (Local DRF theorem) and *Performance* (free on x86, 0.6% on ARMv8, 2.6% on POWER)
  - ★ Allows common compiler optimisations
  - ★ Compilation + Optimisations proved sound
- Proposed as the memory model for OCaml
  - ★ Also suitable for other safe languages (Swift, WebAssembly, JavaScript)