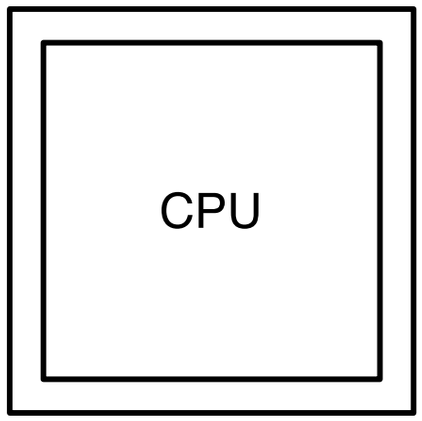
Transactional Memory Under the Hood

Aleksandar Dragojević

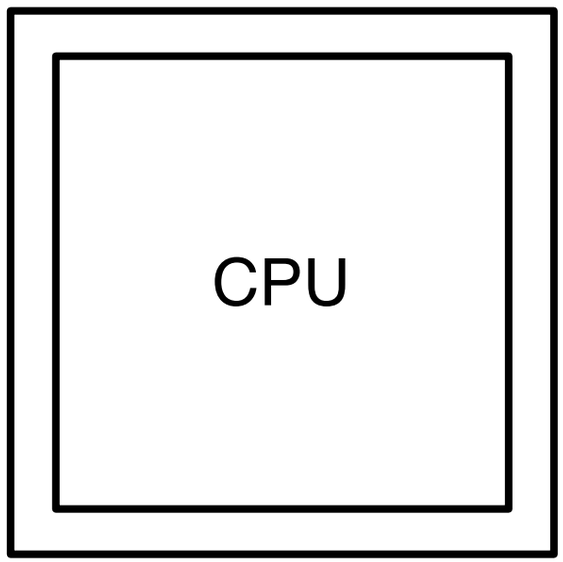
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Hardware Trends

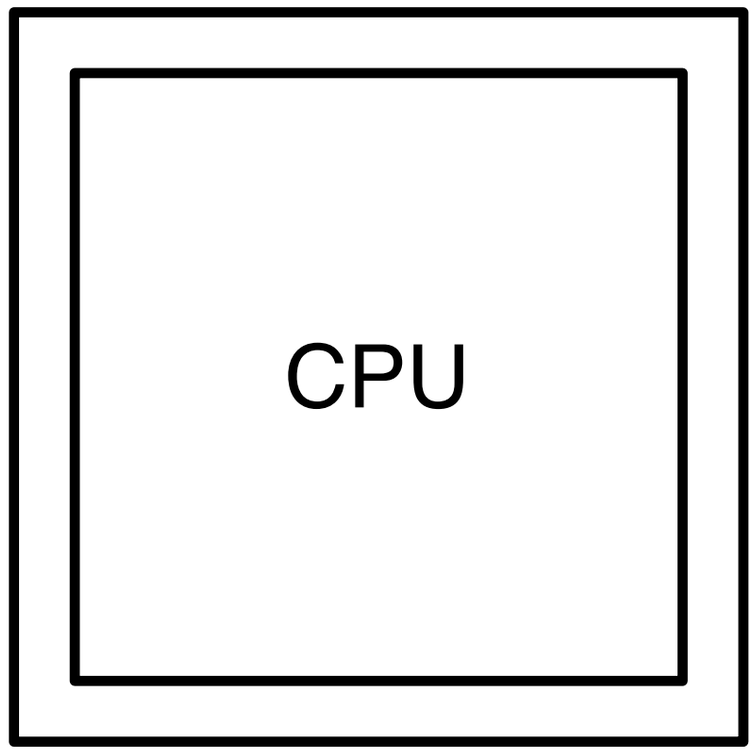
Hardware Trends



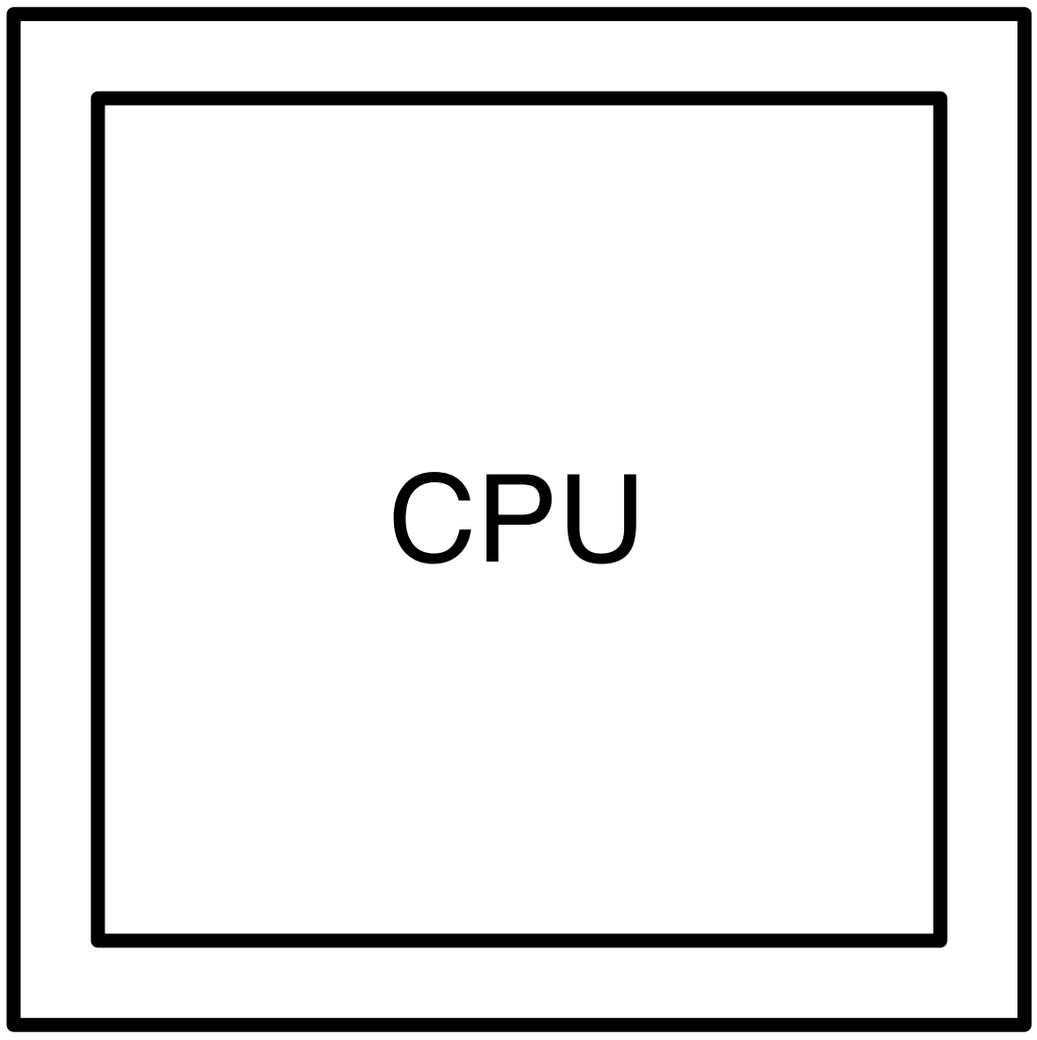
Hardware Trends



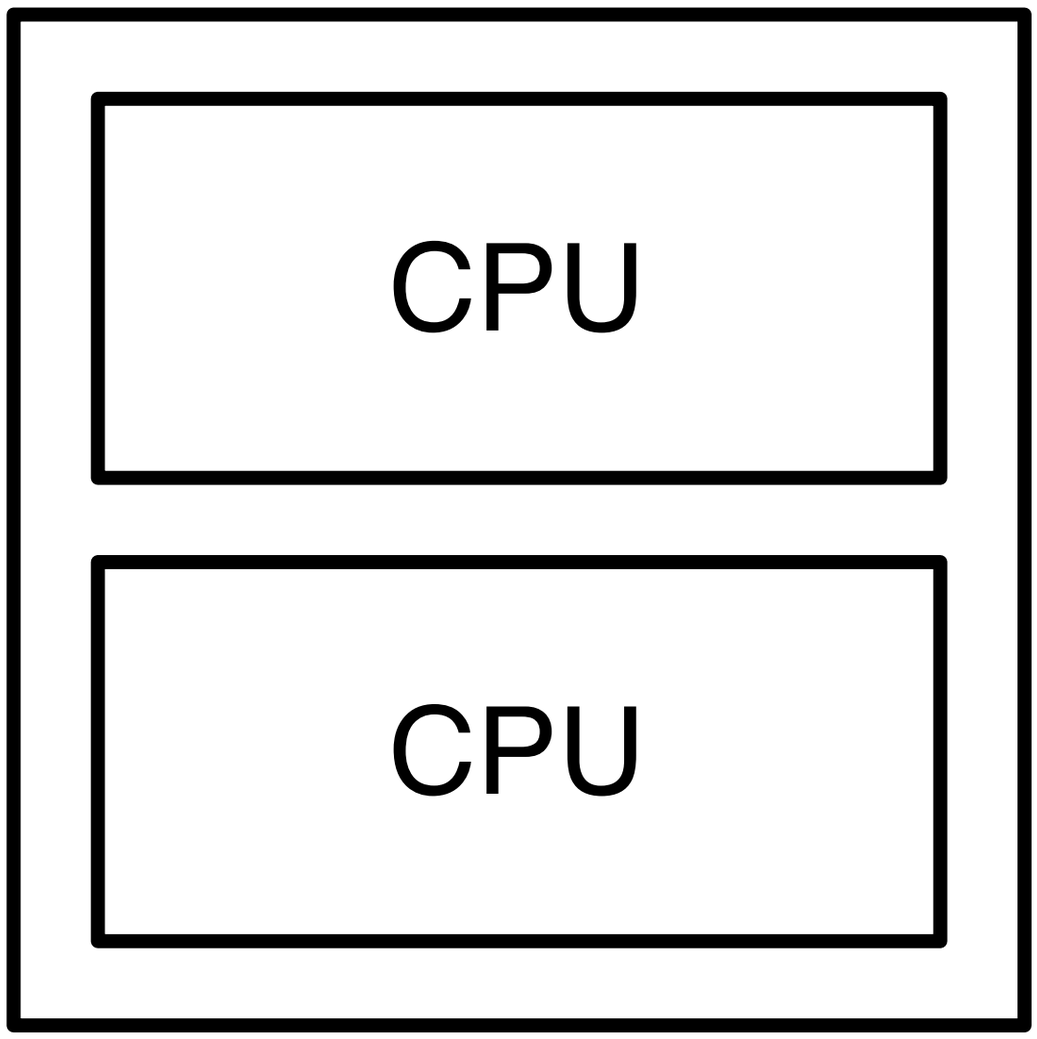
Hardware Trends



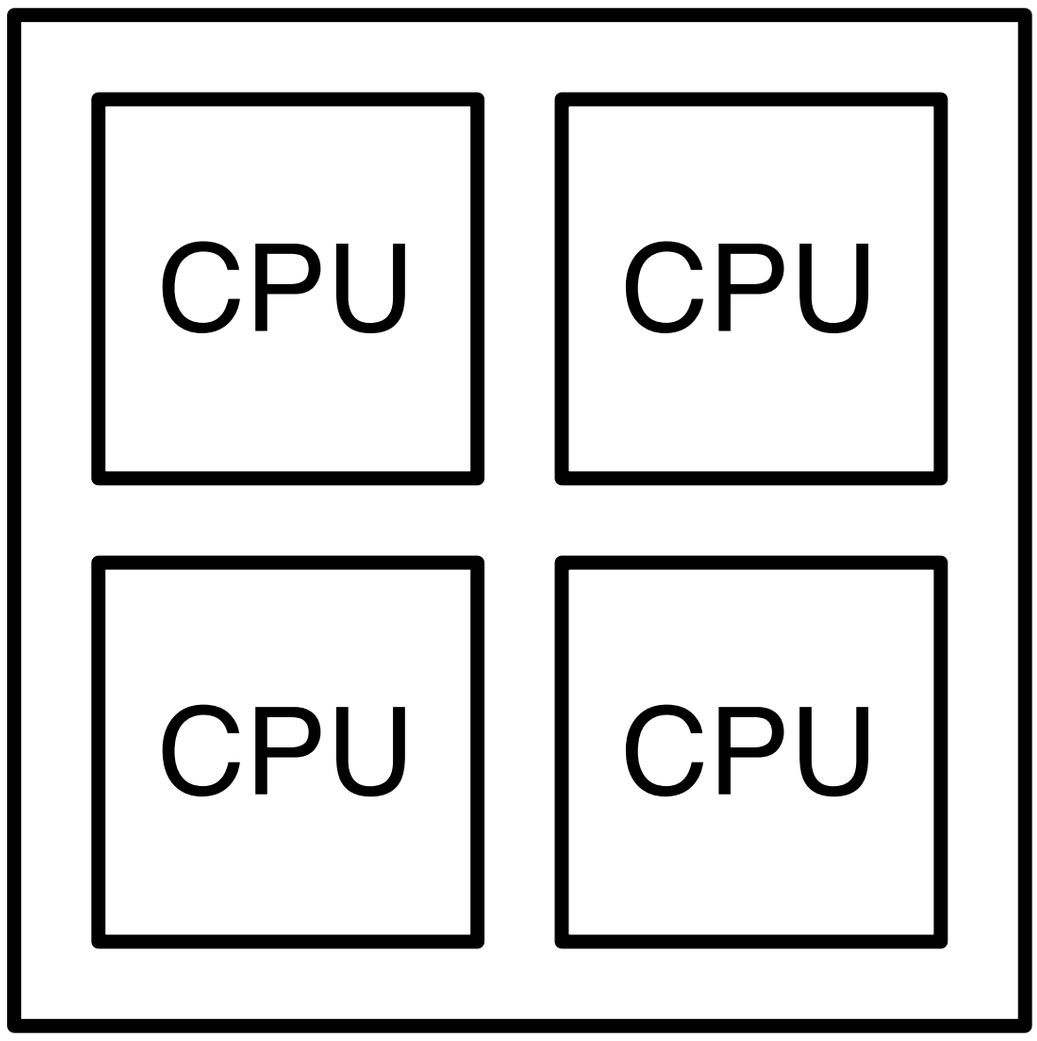
Hardware Trends



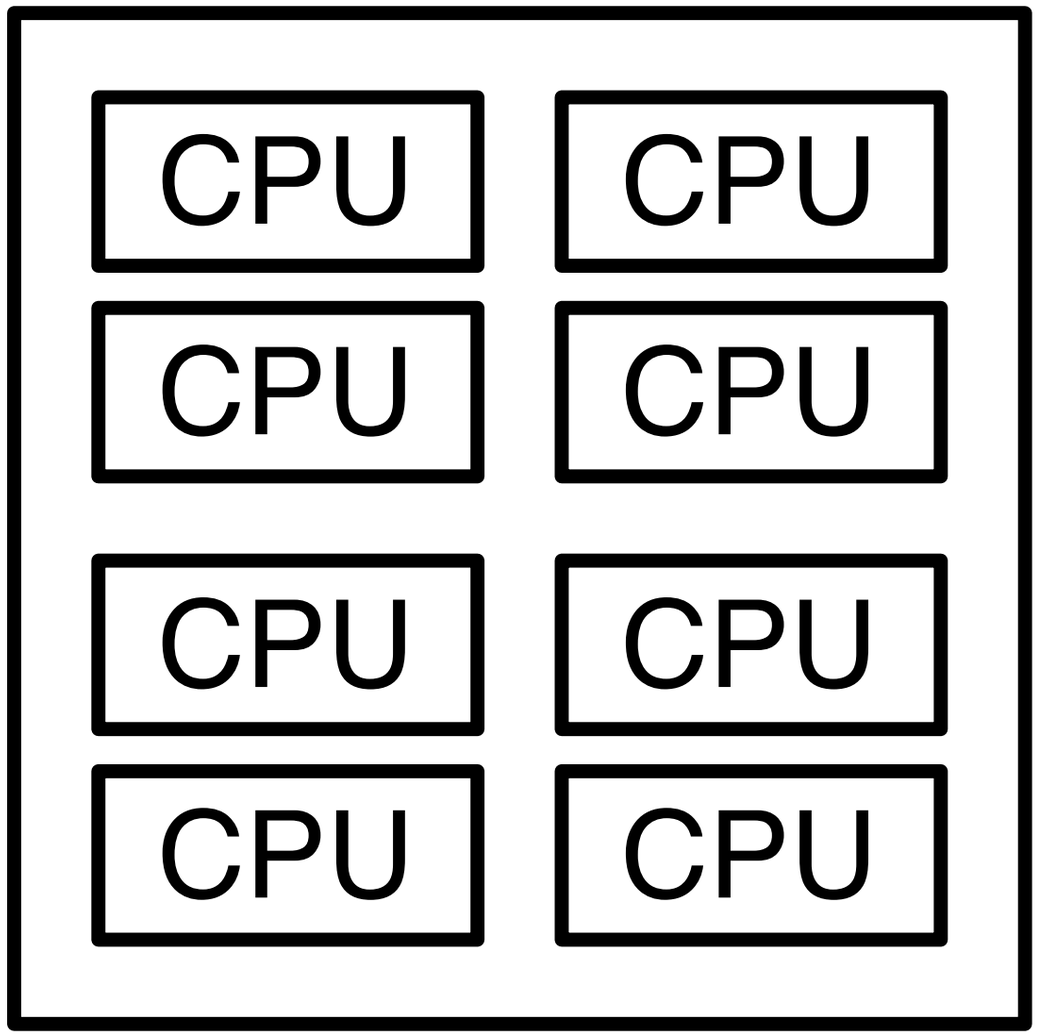
Hardware Trends



Hardware Trends



Hardware Trends



Concurrent Programming

- Domain of experts
  - Fine-grained locking
  - Lock-free
- Average programmers
  - New abstractions needed

Transactional Memory

Transactional Memory

```
1: int a = acc_a;
2: acc_a = a - 20;
3: int b = acc_b;
4: acc_b = b + 20;
```

Transactional Memory

```
atomic { // t1
1: int a = acc_a;
2: acc_a = a - 20;
3: int b = acc_b;
4: acc_b = b + 20;
}
```

Transactional Memory

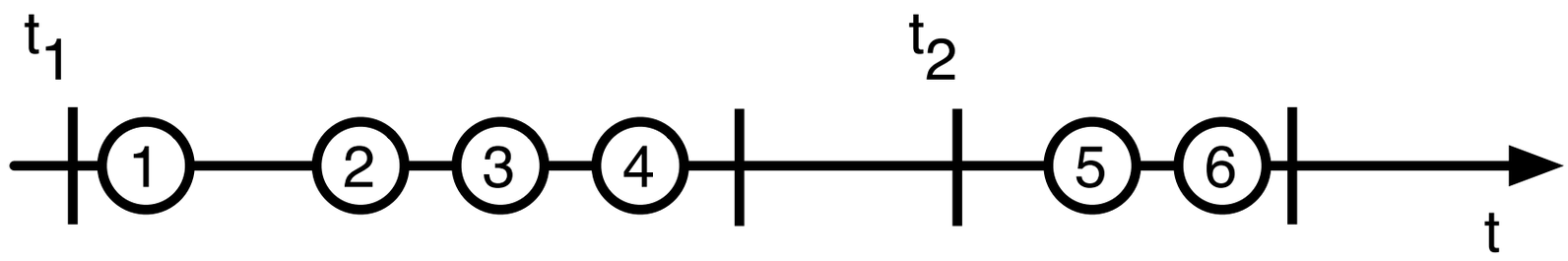
```
atomic { // t1
1: int a = acc_a;
2: acc_a = a - 20;
3: int b = acc_b;
4: acc_b = b + 20;
}
```

```
atomic { // t2
5: int a = acc_a;
6: acc_a = a + 10;
}
```

Transactional Memory

```
atomic { // t1
1: int a = acc_a;
2: acc_a = a - 20;
3: int b = acc_b;
4: acc_b = b + 20;
}
```

```
atomic { // t2
5: int a = acc_a;
6: acc_a = a + 10;
}
```

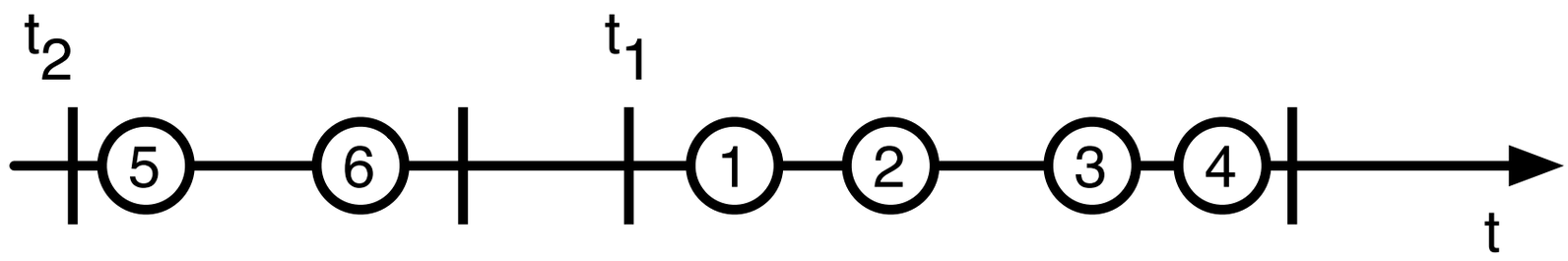


correct

Transactional Memory

```
atomic { // t1
1: int a = acc_a;
2: acc_a = a - 20;
3: int b = acc_b;
4: acc_b = b + 20;
}
```

```
atomic { // t2
5: int a = acc_a;
6: acc_a = a + 10;
}
```

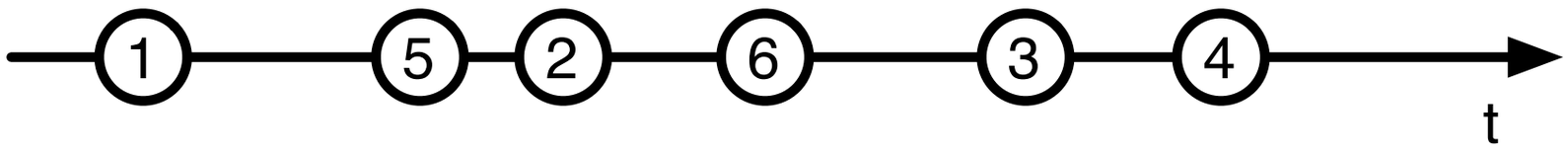


correct

Transactional Memory

```
atomic { // t1
1: int a = acc_a;
2: acc_a = a - 20;
3: int b = acc_b;
4: acc_b = b + 20;
}
```

```
atomic { // t2
5: int a = acc_a;
6: acc_a = a + 10;
}
```

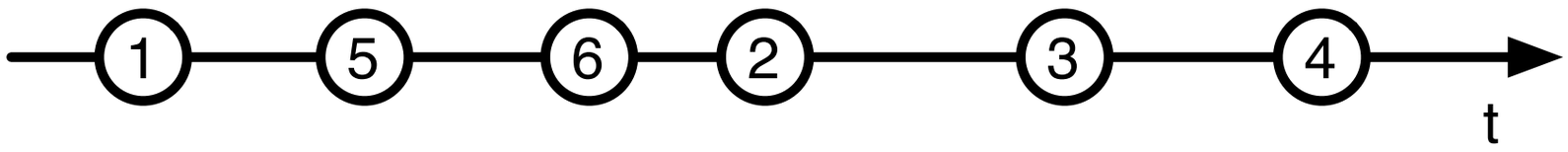


client ☺

Transactional Memory

```
atomic { // t1
1: int a = acc_a;
2: acc_a = a - 20;
3: int b = acc_b;
4: acc_b = b + 20;
}
```

```
atomic { // t2
5: int a = acc_a;
6: acc_a = a + 10;
}
```



bank ☺

Opacity reminder

- Serializability
  - there exists an equivalent serial (one thread) execution
- Consistent memory view
  - no transaction can e.g. divide by zero because of non-consistent reads

Outline

- STM How To
- SwissTM
- Evaluating Performance

STM How To

STM How To

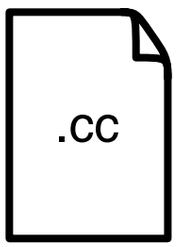
How does it all fit?

Software TM

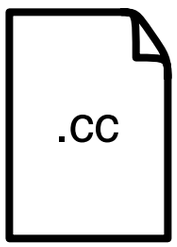
- Available now
- Component of HyTM
- Backwards compatibility

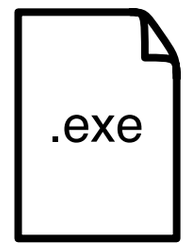
From .cc To .exe

From .cc To .exe



From .cc To .exe

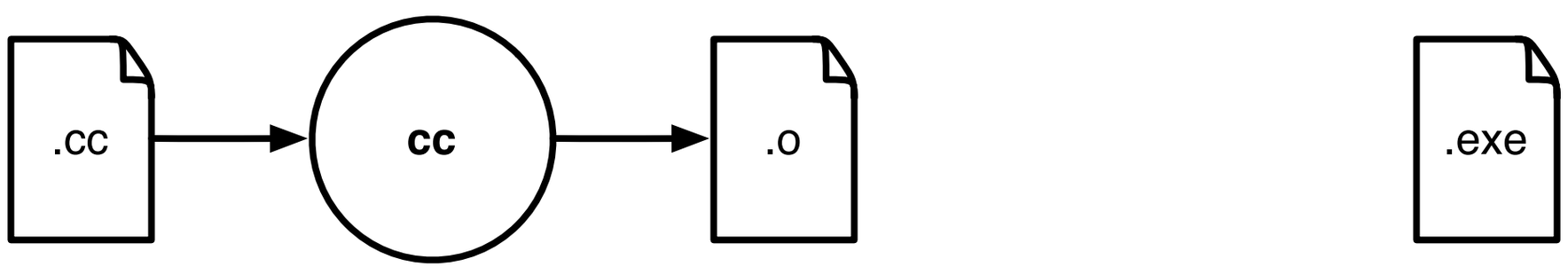




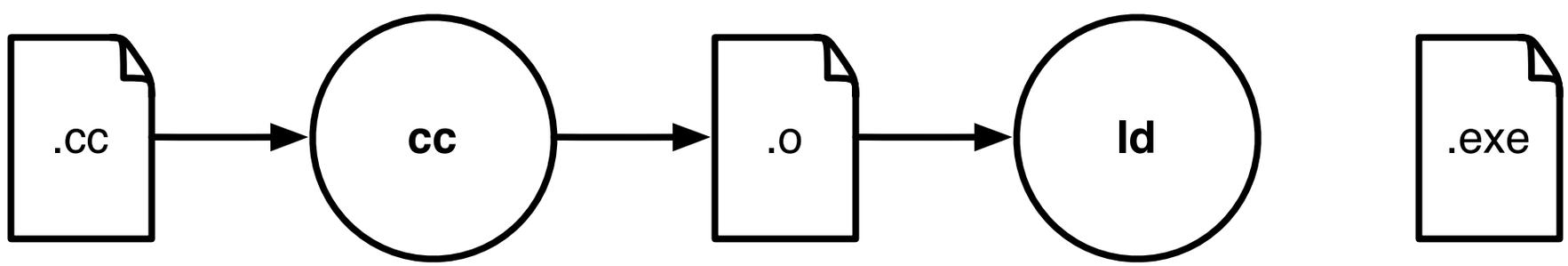
From .cc To .exe

From .cc To .exe

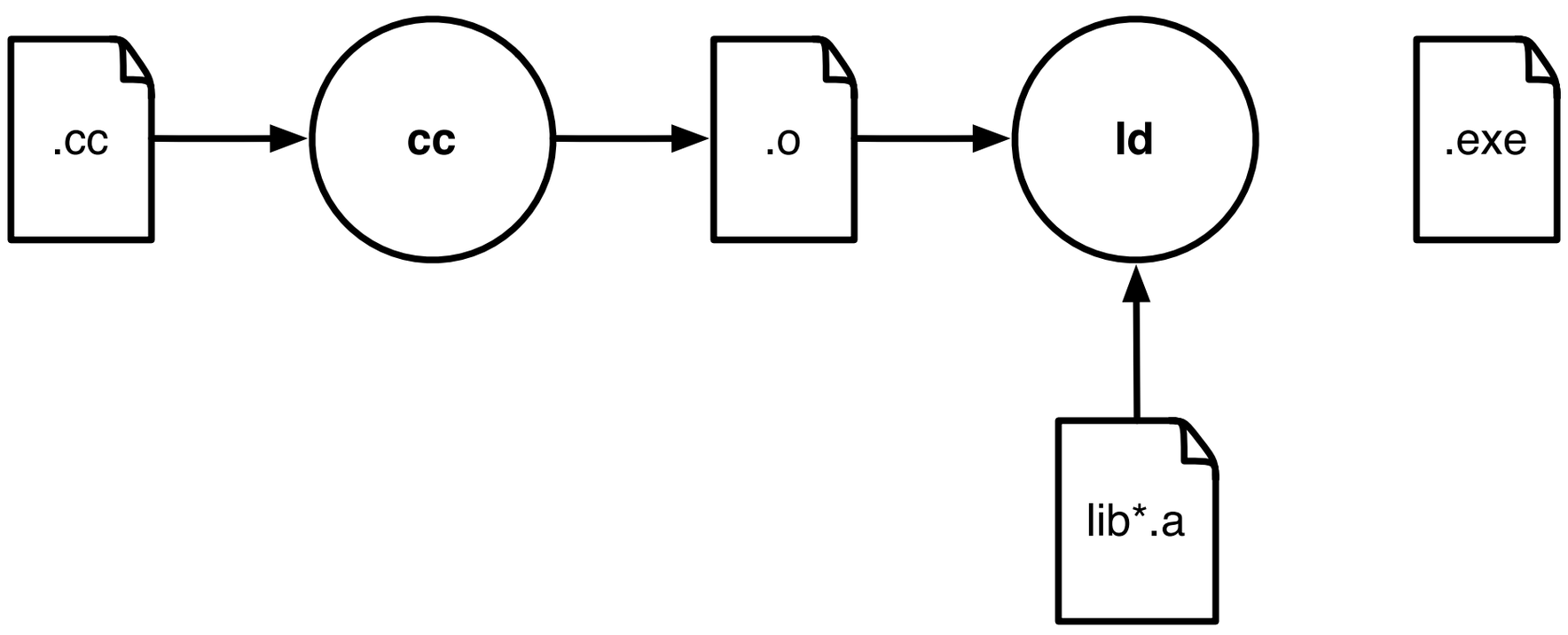
From .cc To .exe



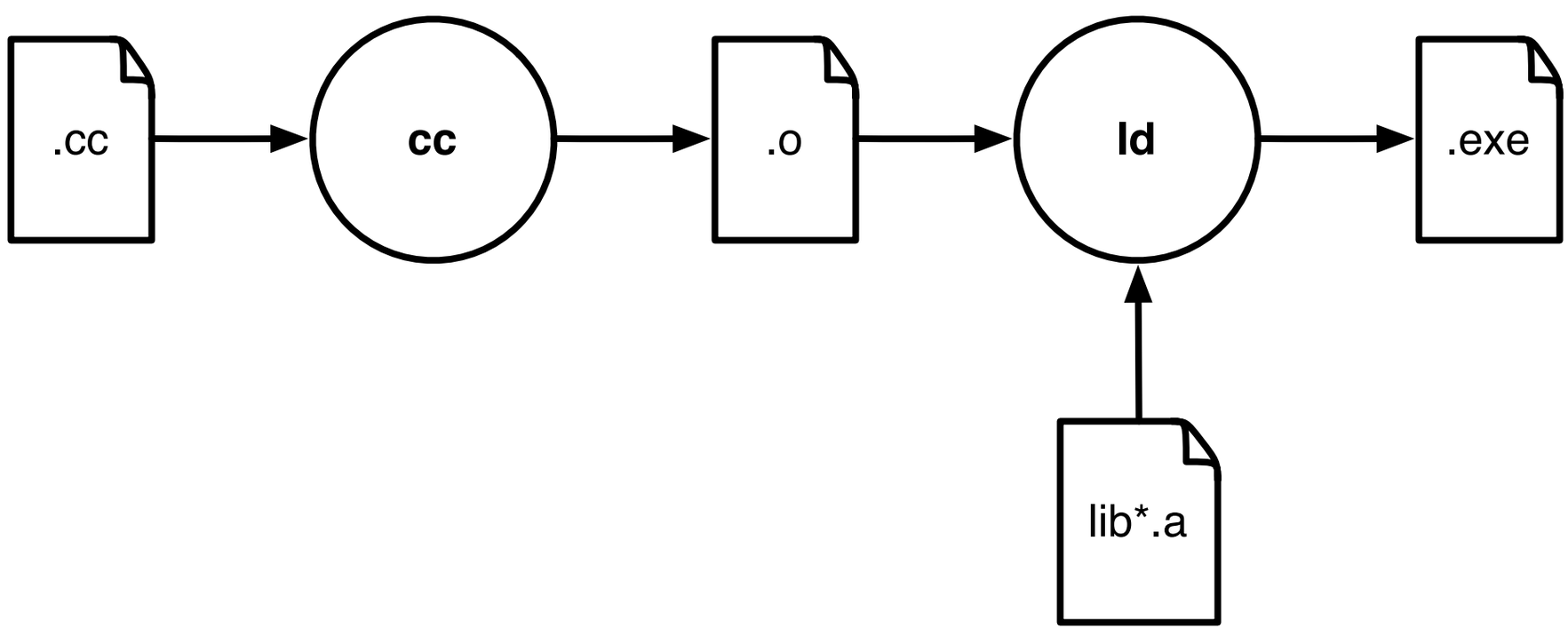
From .cc To .exe



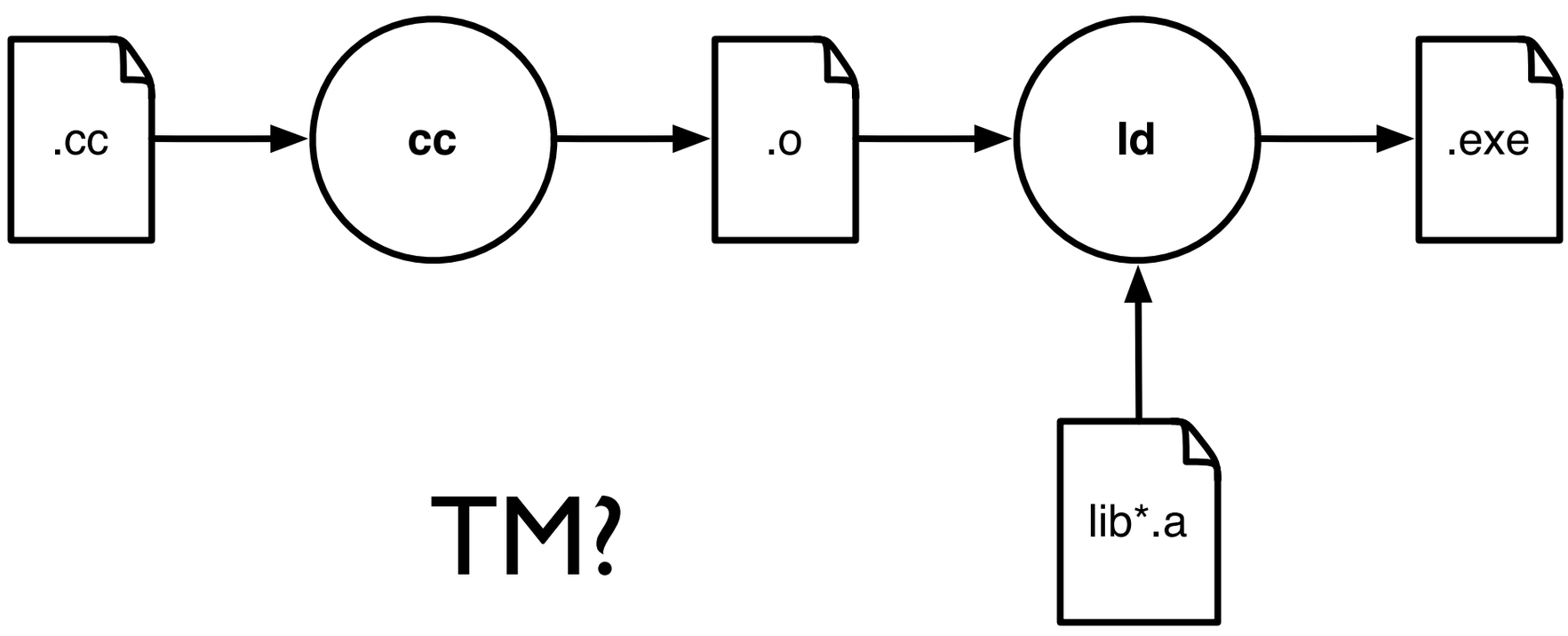
From .cc To .exe



From .cc To .exe



From .cc To .exe

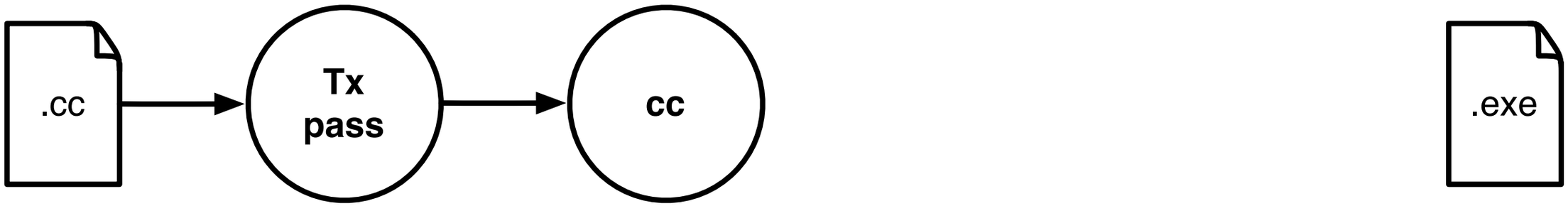


TM?

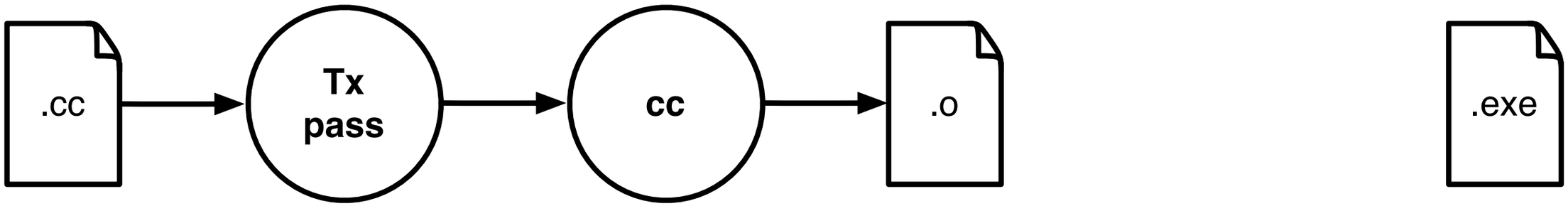
From .cc To .exe

From .cc To .exe

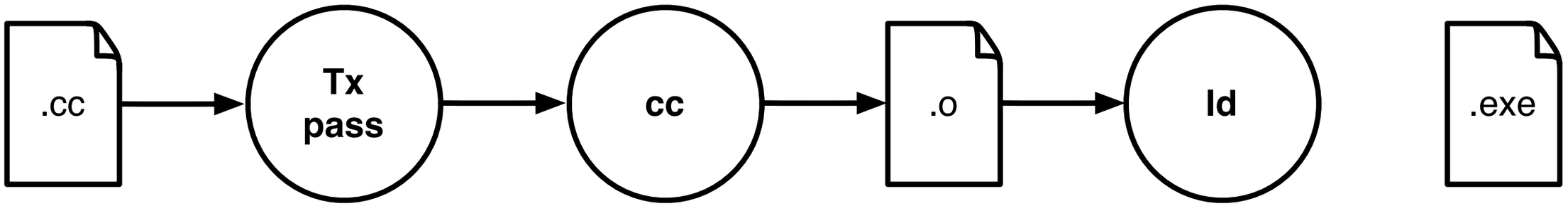
From .cc To .exe



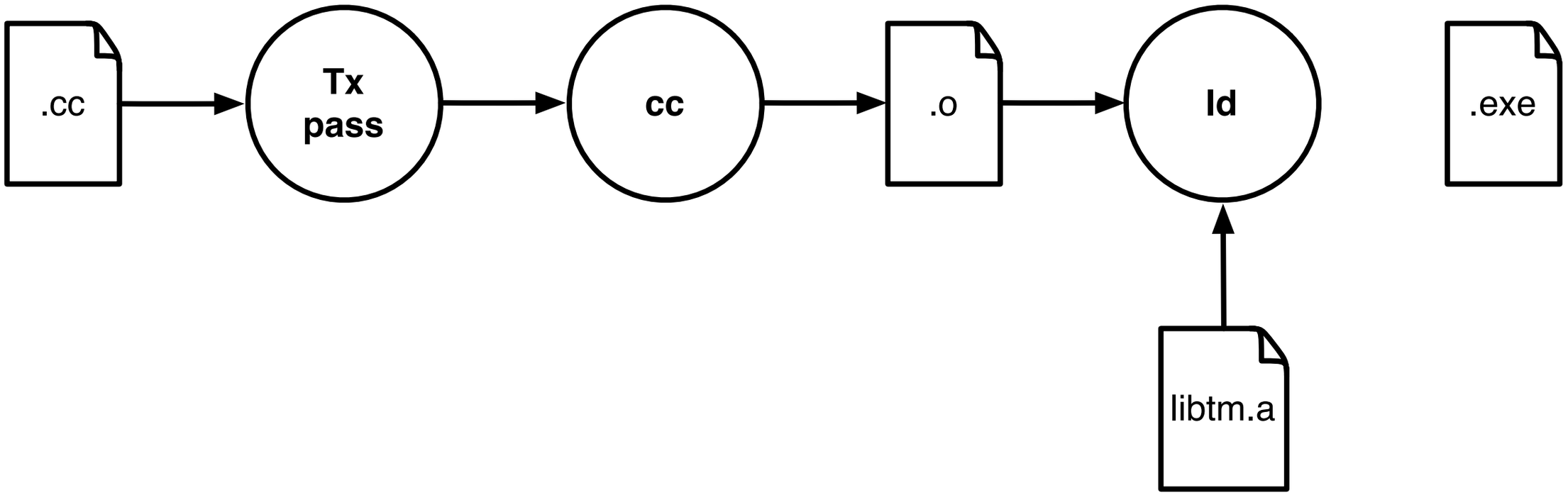
From .cc To .exe



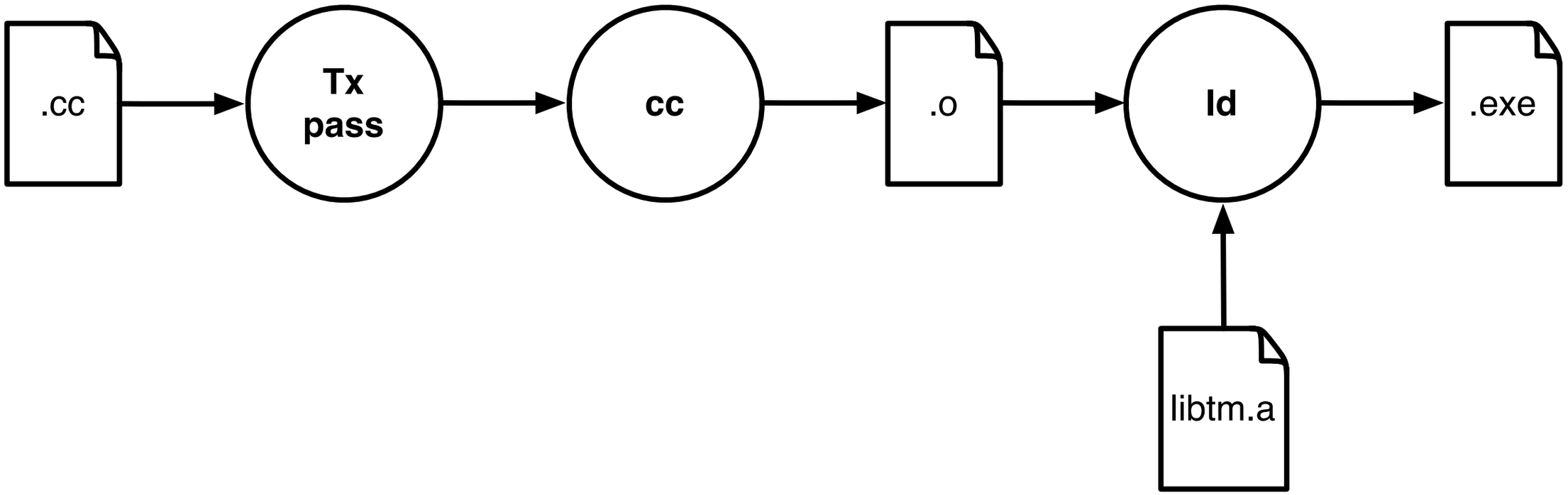
From .cc To .exe



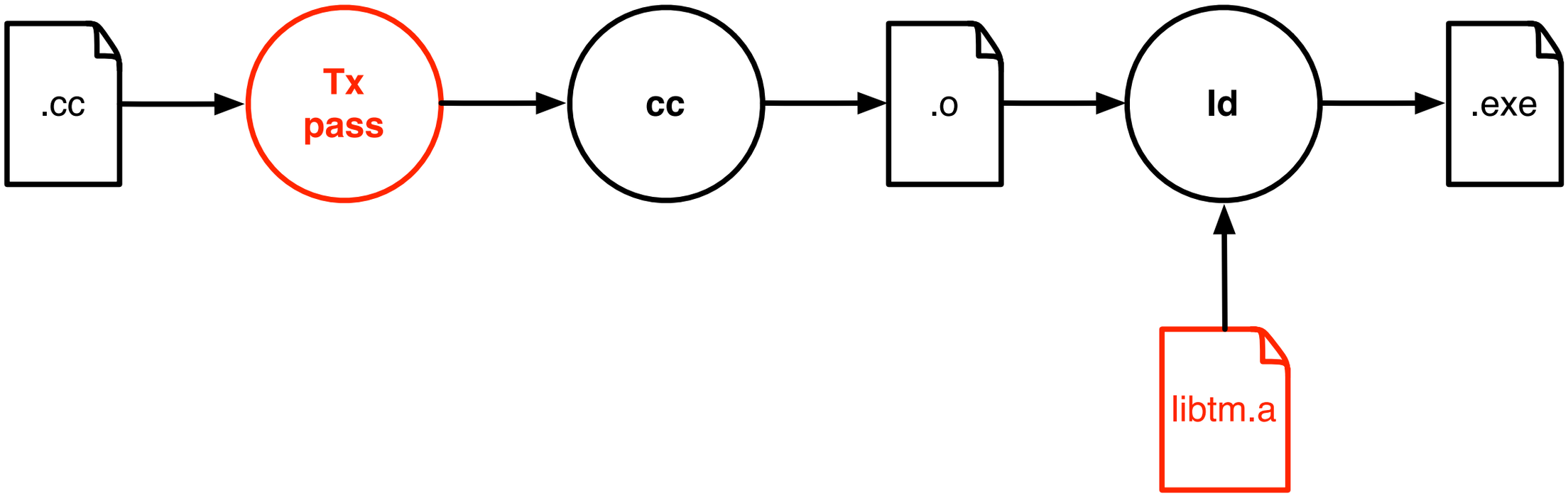
From .cc To .exe



From .cc To .exe



From .cc To .exe



TM library

- Implements TM
- Different algorithms
  - different performance
- Similar (same) API

TM library API

- tx_start()
- tx_read(addr) : val
- tx_write(addr, val)
- tx_commit()
- tx_abort()

Tx Pass

Tx Pass

```
atomic { // t1
1: int a = acc_a;
2: acc_a = a - 20;
3: int b = acc_b;
4: acc_b = b + 20;
}
```

Tx Pass

```
atomic { // t1
1: int a = acc_a;
2: acc_a = a - 20;
3: int b = acc_b;
4: acc_b = b + 20;
}
```

```
tx_start();
1: int a = tx_read(acc_a);
2: tx_write(acc_a, a-20);
3: int b = tx_read(acc_b);
4: tx_write(acc_b, b+20);
tx_commit();
```

Implementing Tx Pass

- Manual
- Compiler
- Other

Manual Tx Pass

- Manually insert TM API calls
- Highly optimized
  - no unnecessary TM calls
- Error-prone
  - missing TM calls
- Tedious
  - need to rewrite a lot of code

Compiler Tx Pass

- Integrated with the compiler
- Simple to use
- Lower performance
  - unnecessary TM calls
- Better support for optimizations
  - lower the overheads

Other Tx Pass

- Source to source compiler
  - separate, simpler compiler
- Bytecode instrumentation
  - for managed languages (Java, C#)

TM libraries

- SwissTM (EPFL)
- DSTM, TL2, TLRW, SkyTM (Sun)
- McRT (Intel)
- SXM, Bartok (Microsoft)
- ...

Tx Passes

- C/C++
  - Intel
  - DTMC (LLVM)
  - Sun
  - gcc
- Java
  - Deuce

SwissTM

SwissTM

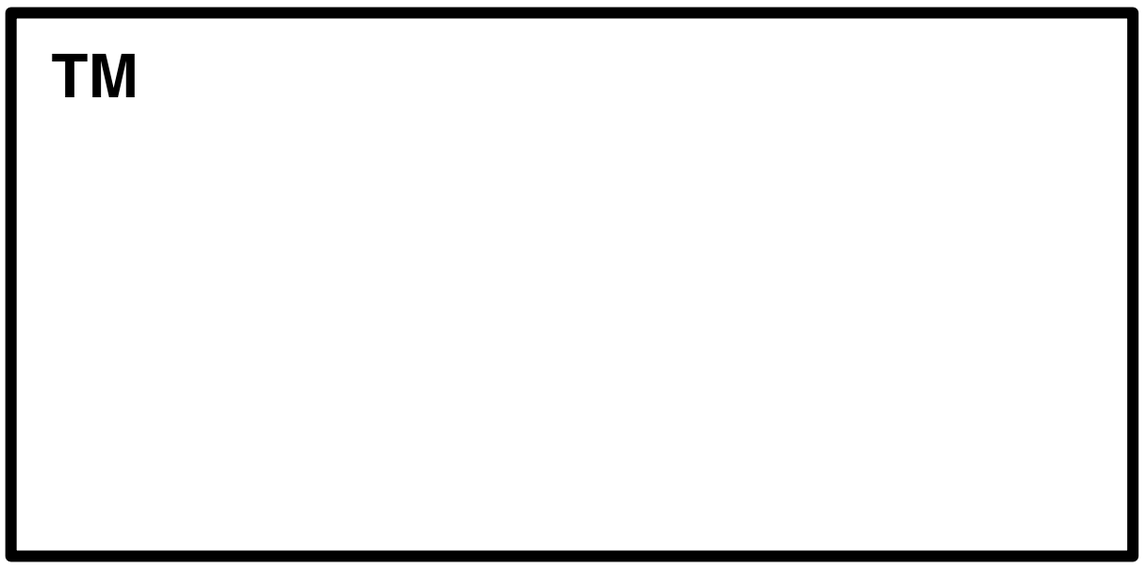
How to implement an STM library?

SwissTM Design

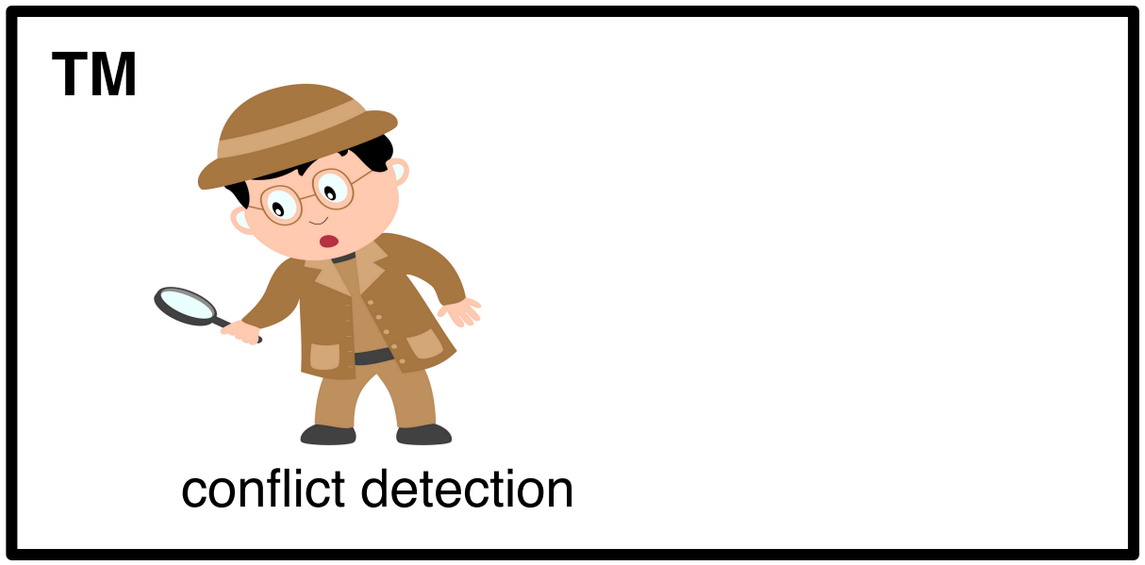
SwissTM Design

What is the high level view of SwissTM?

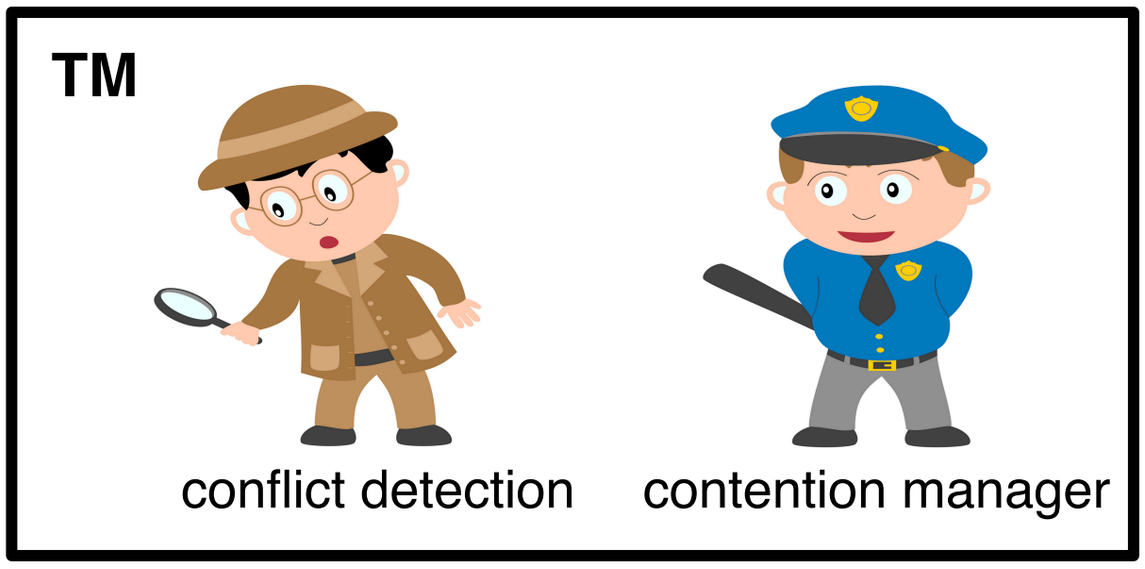
Transactional Memory



Transactional Memory

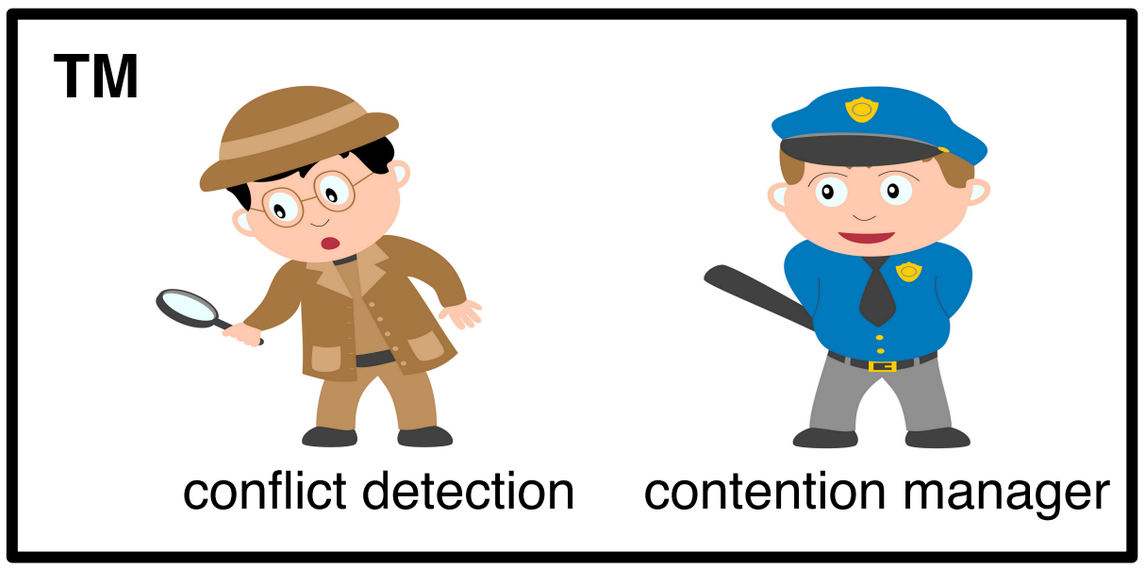


Transactional Memory



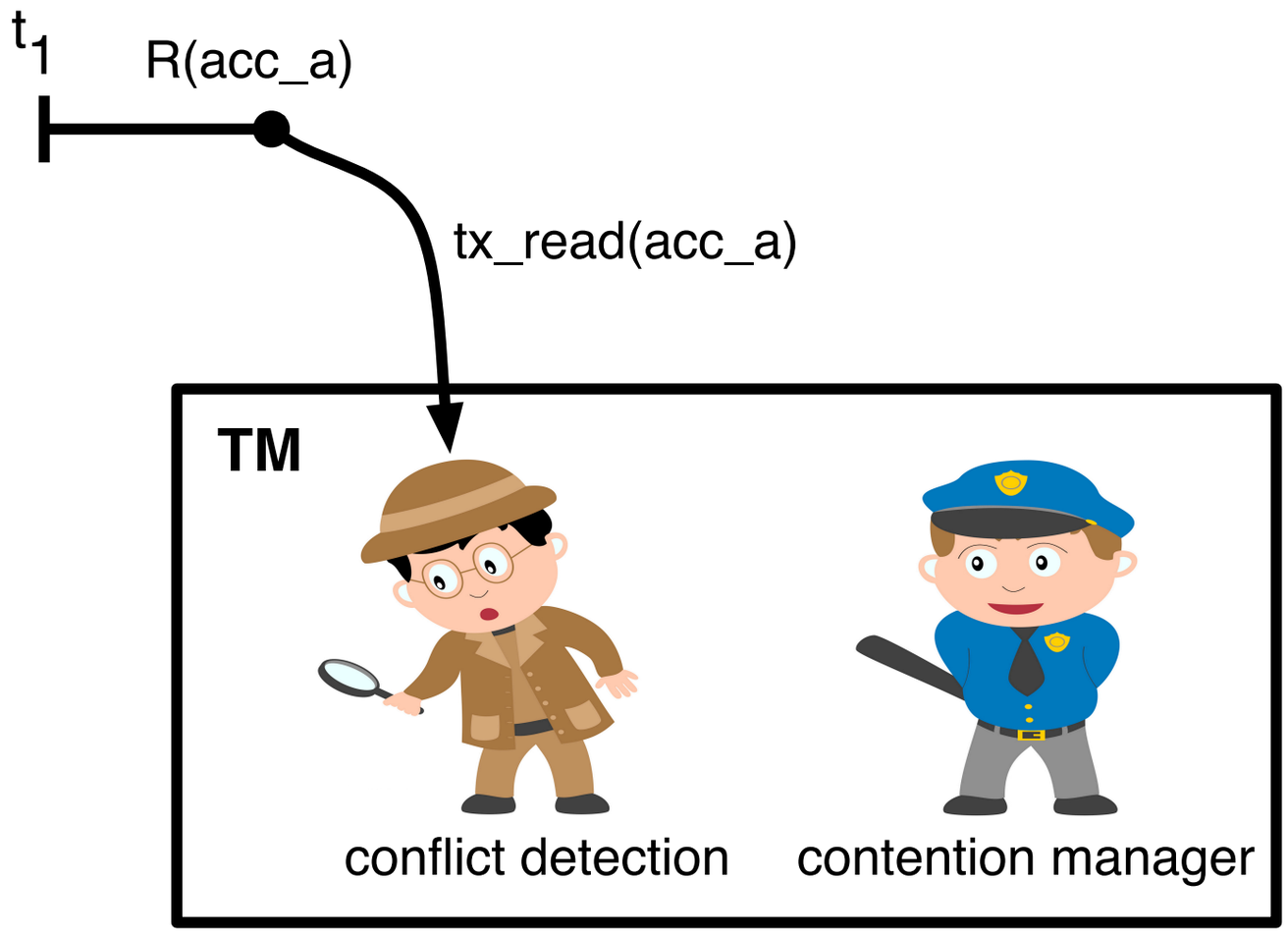
Transactional Memory

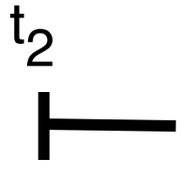
t_1 ⊢



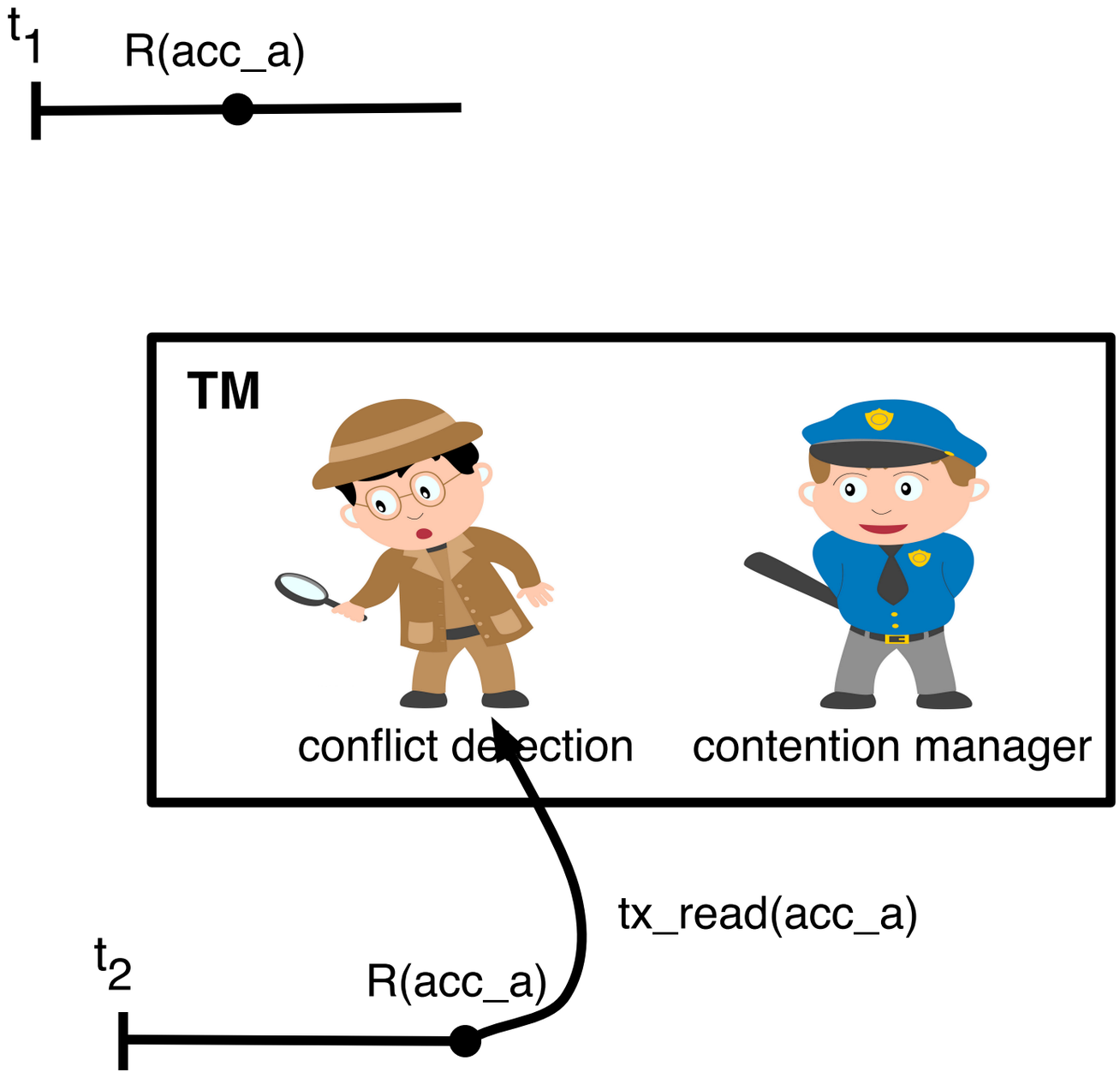
t_2 ⊢

Transactional Memory

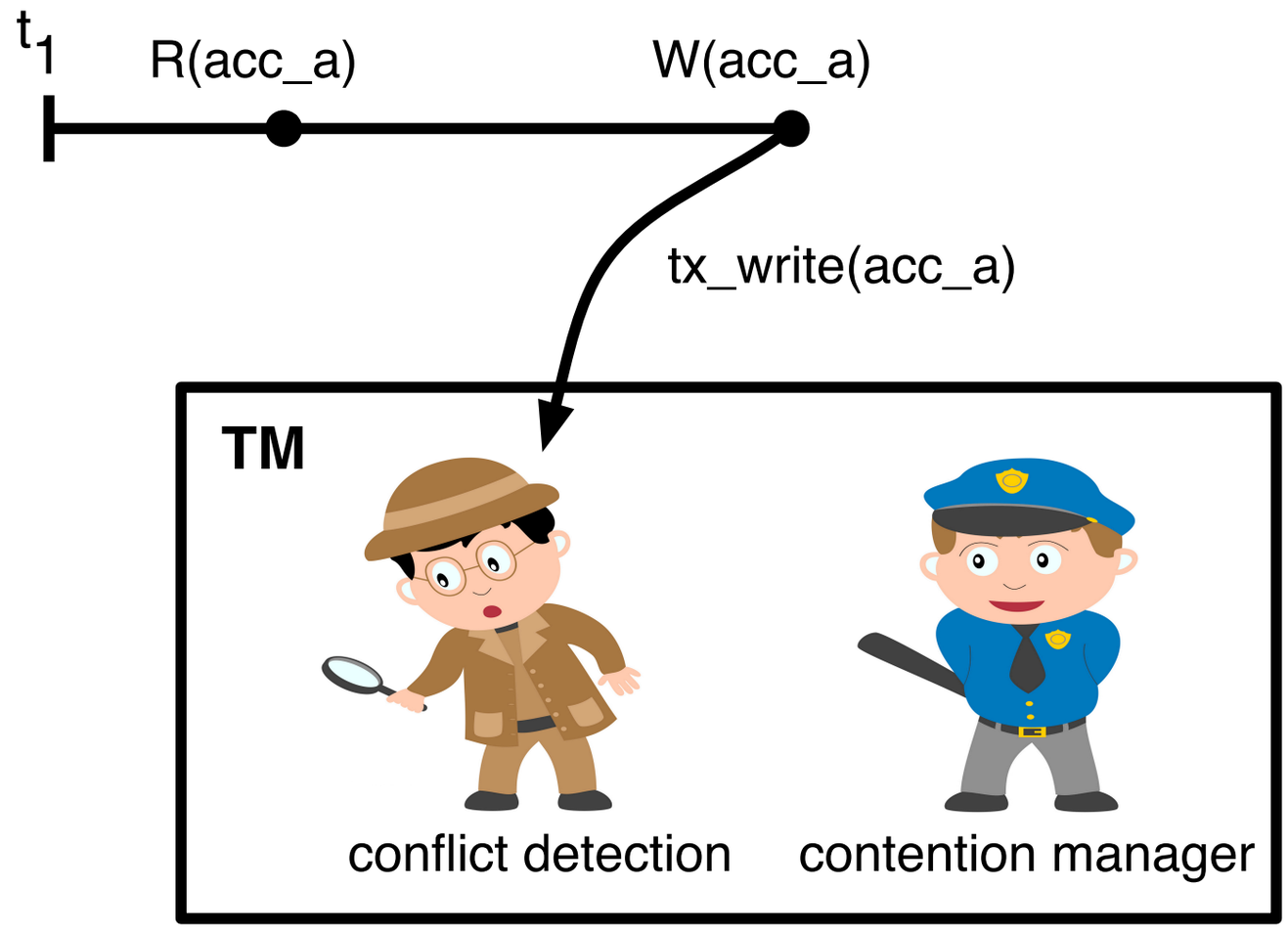


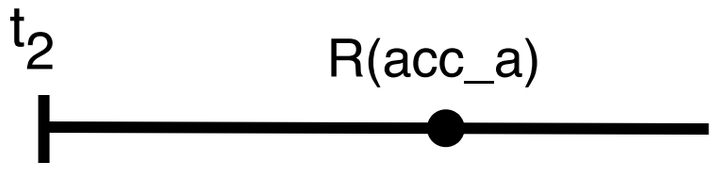


Transactional Memory

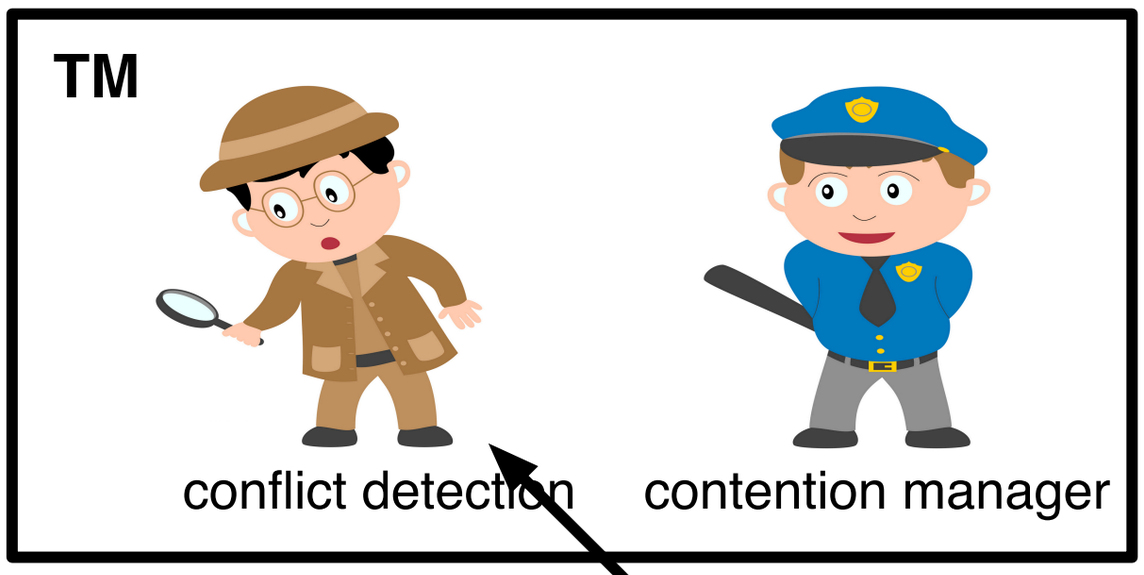


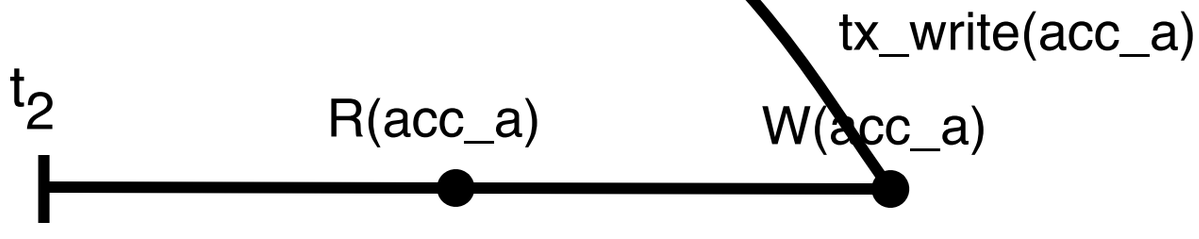
Transactional Memory



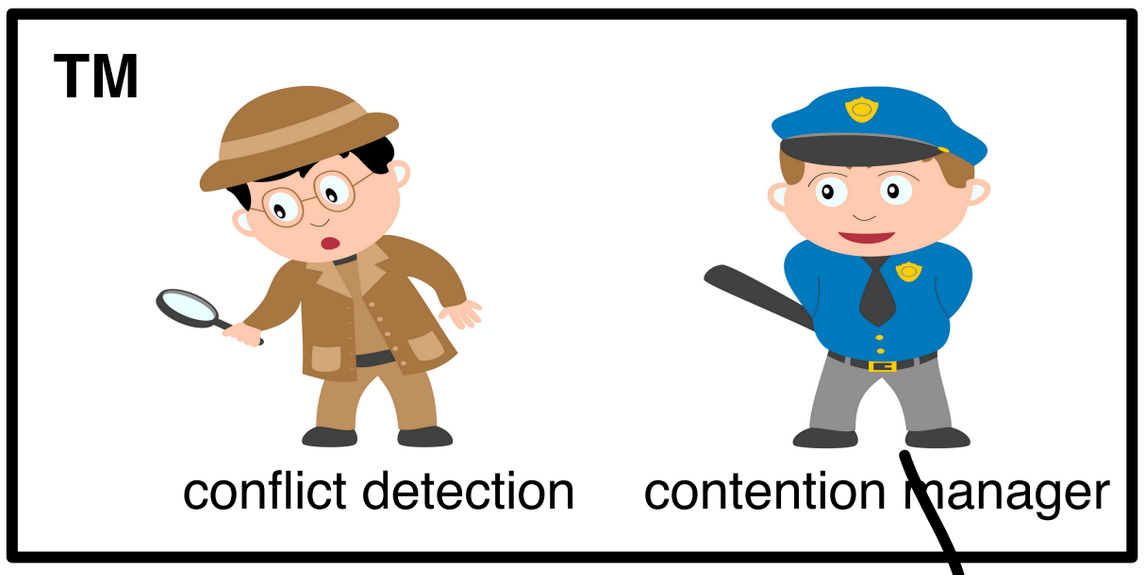


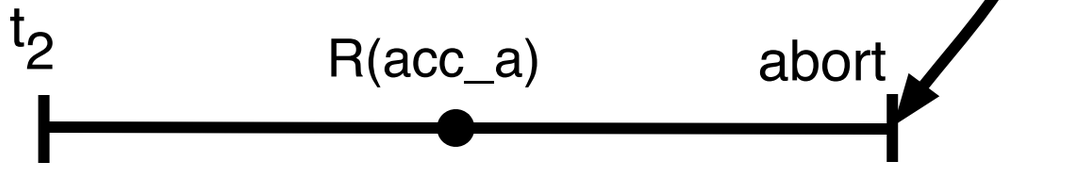
Transactional Memory





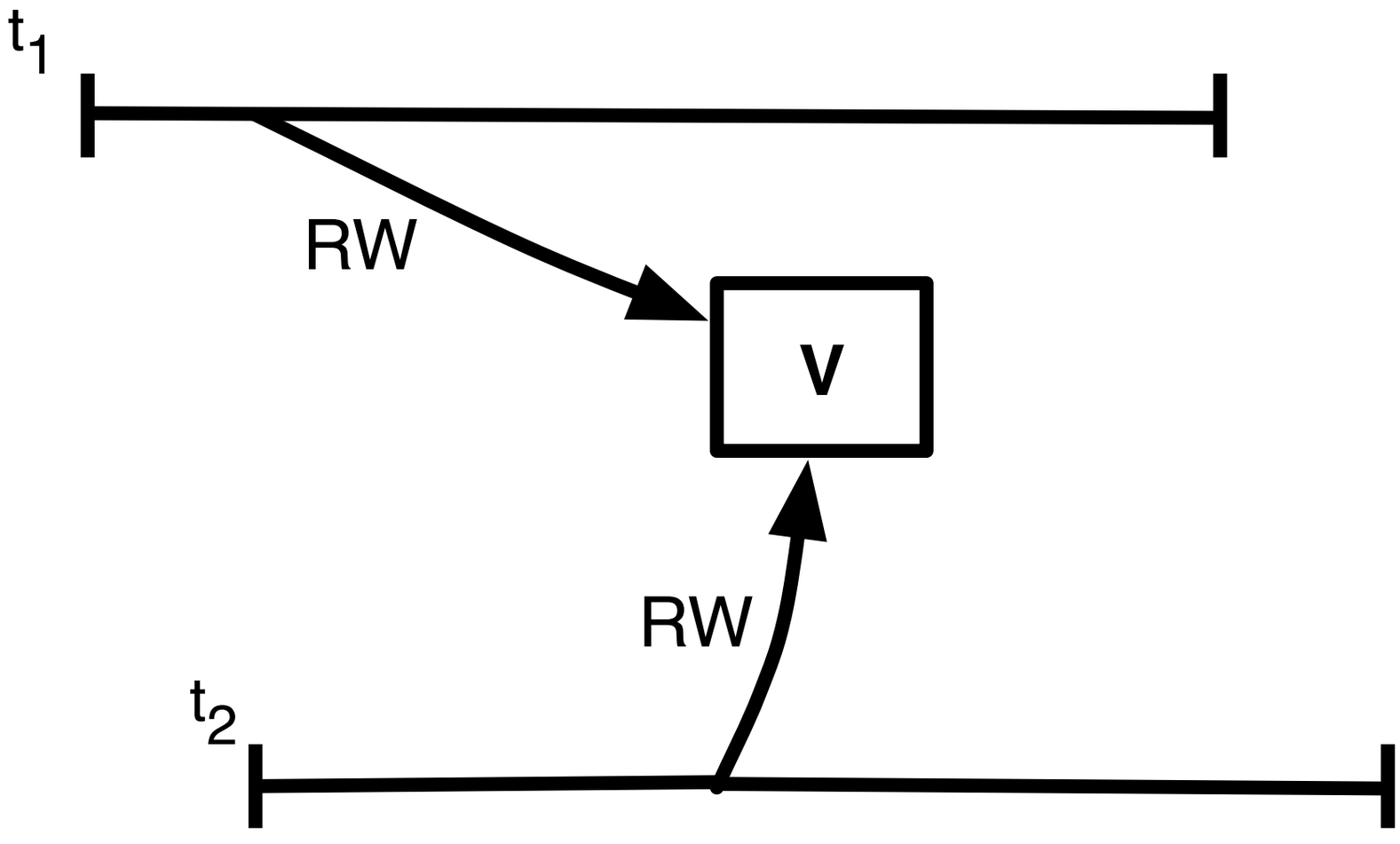
Transactional Memory





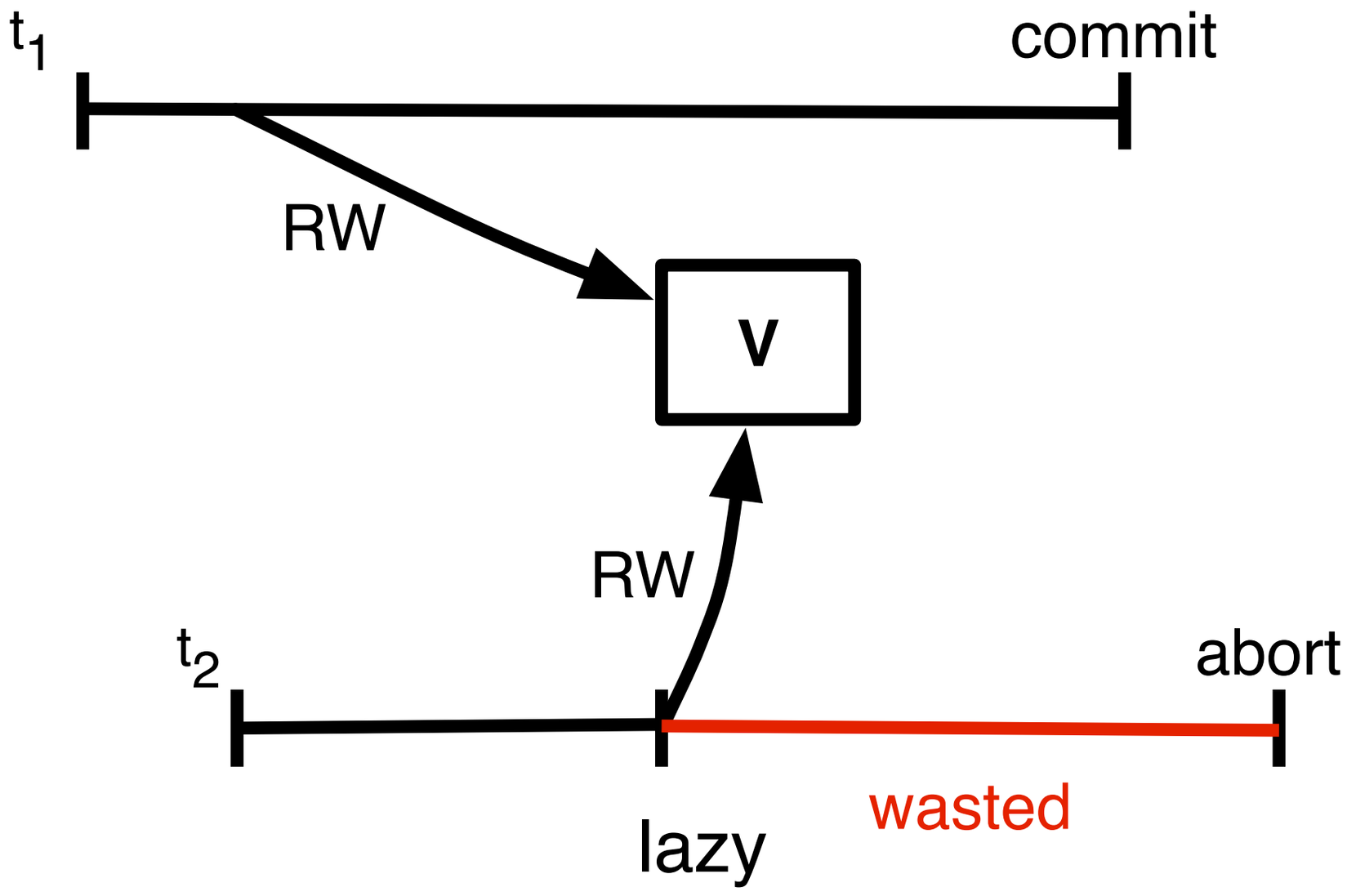
Conflict Detection

eager > lazy



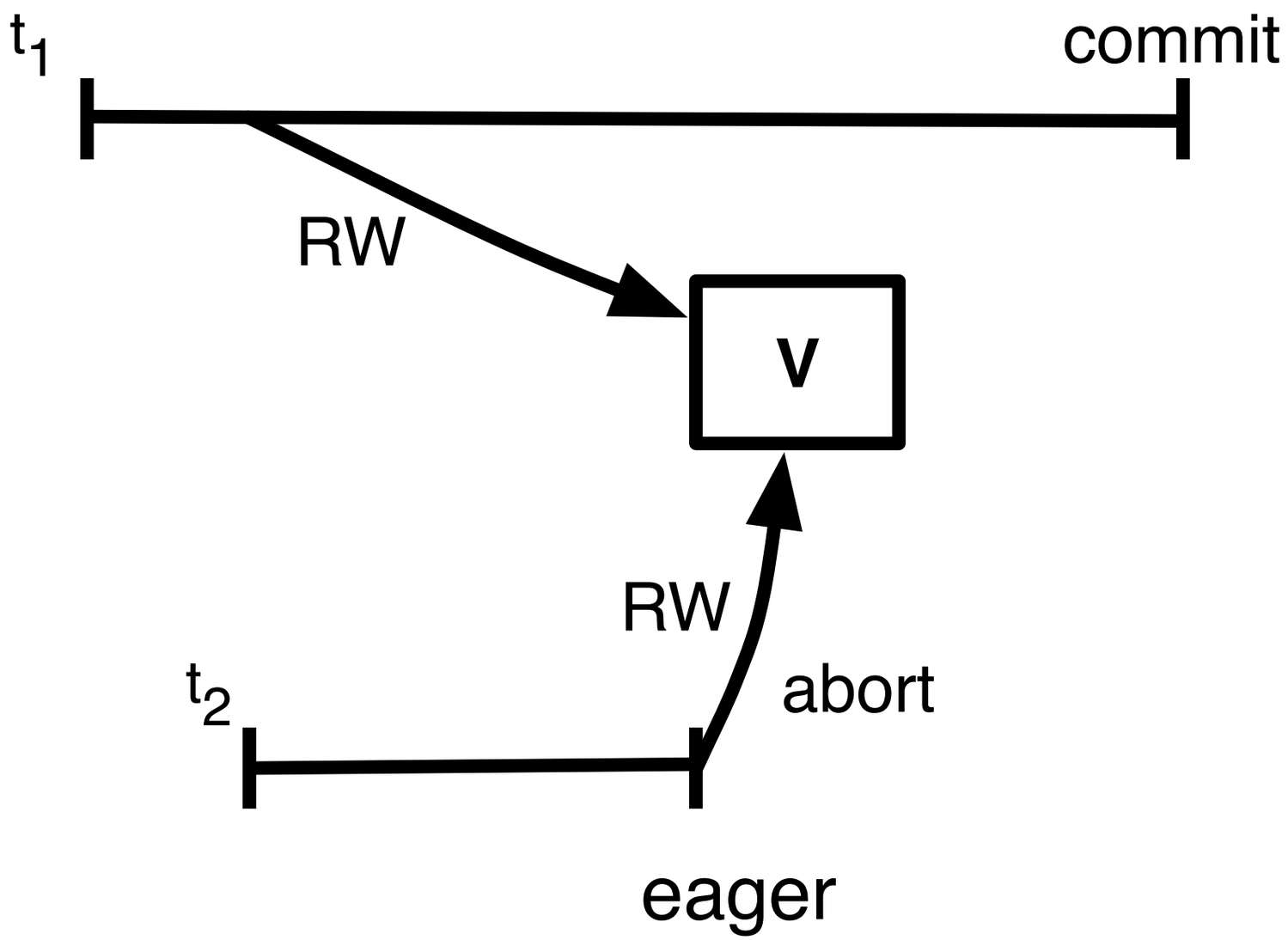
Conflict Detection

eager > lazy



Conflict Detection

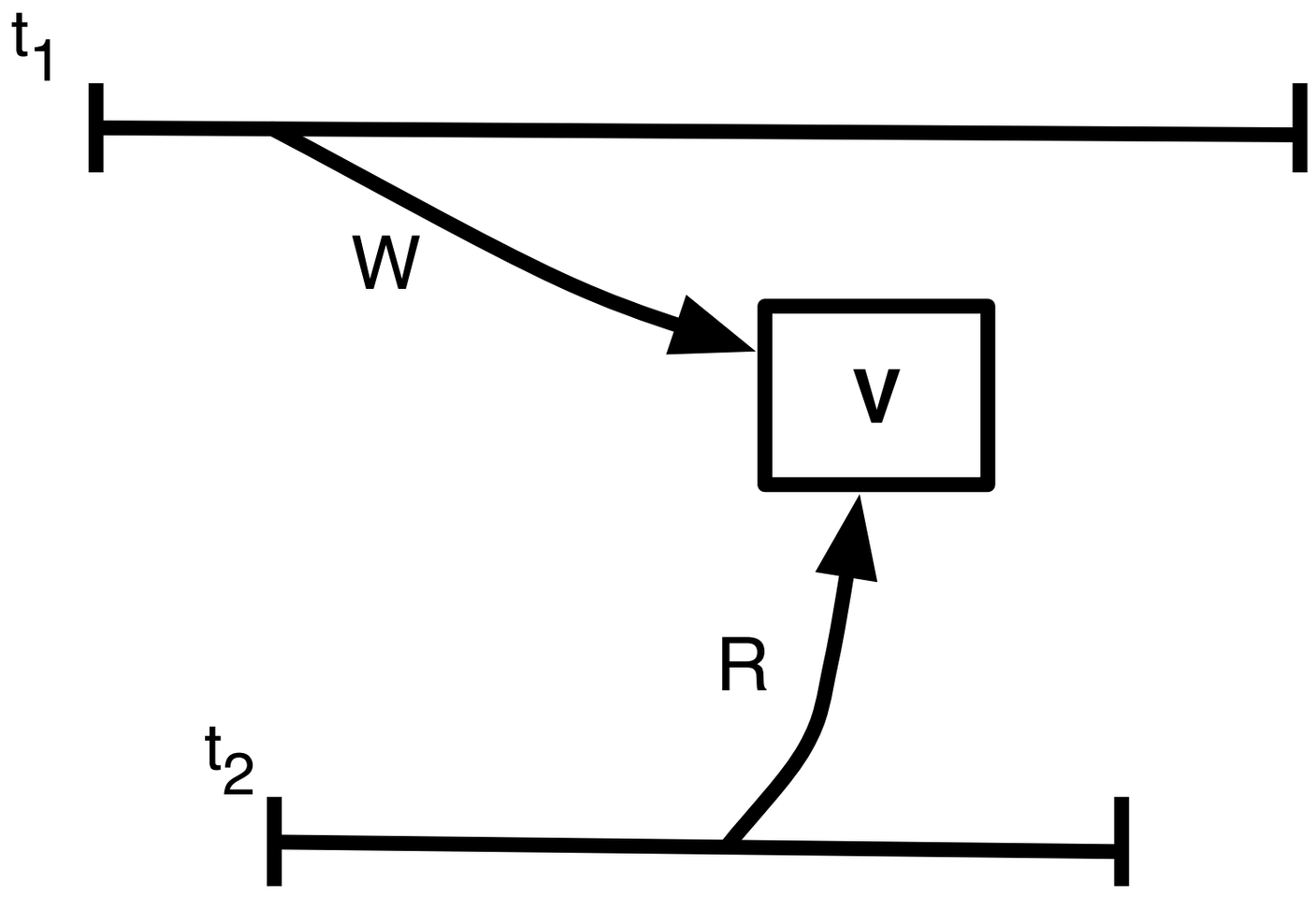
eager > lazy



Conflict Detection

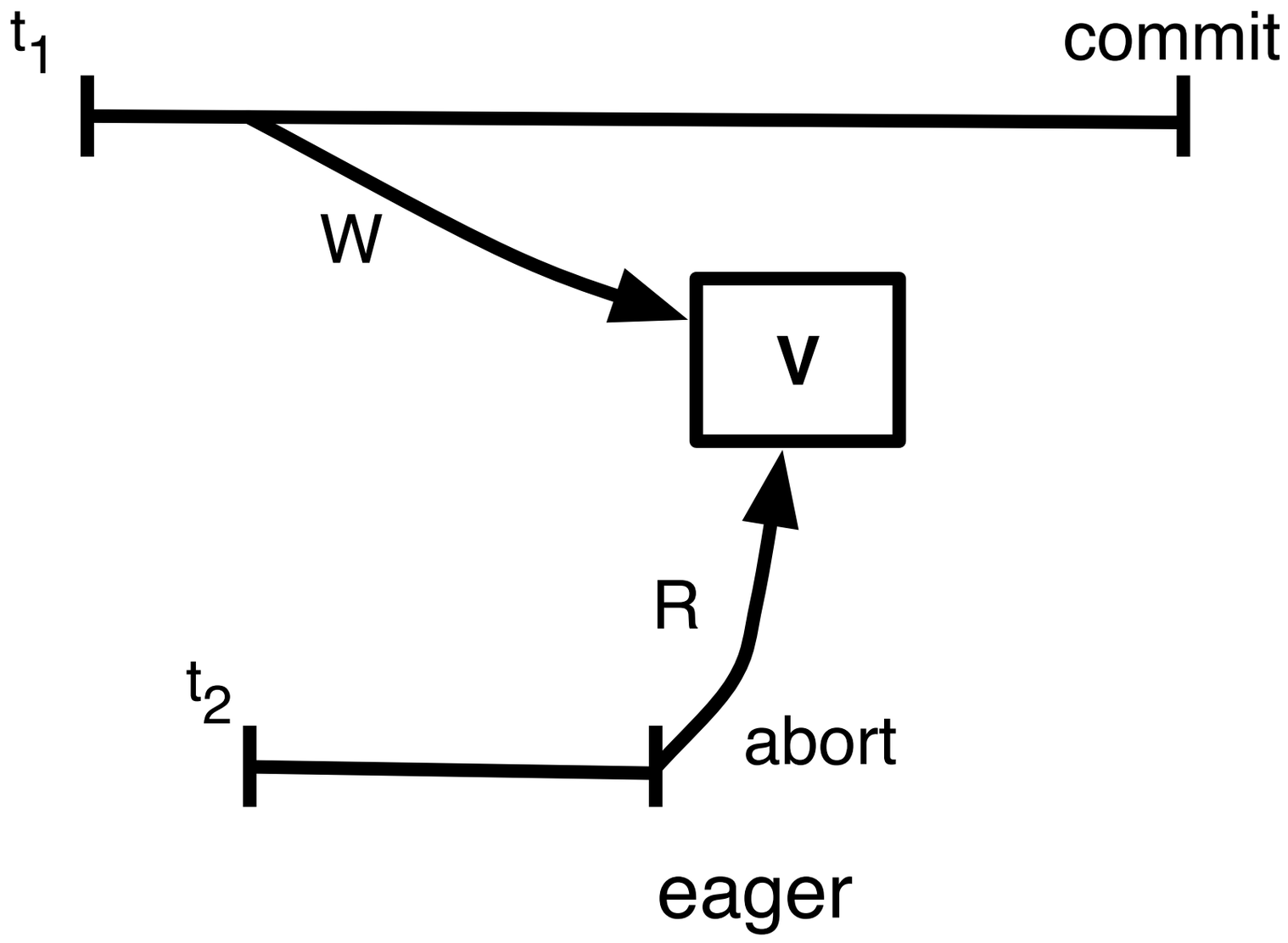
Conflict Detection

lazy > eager



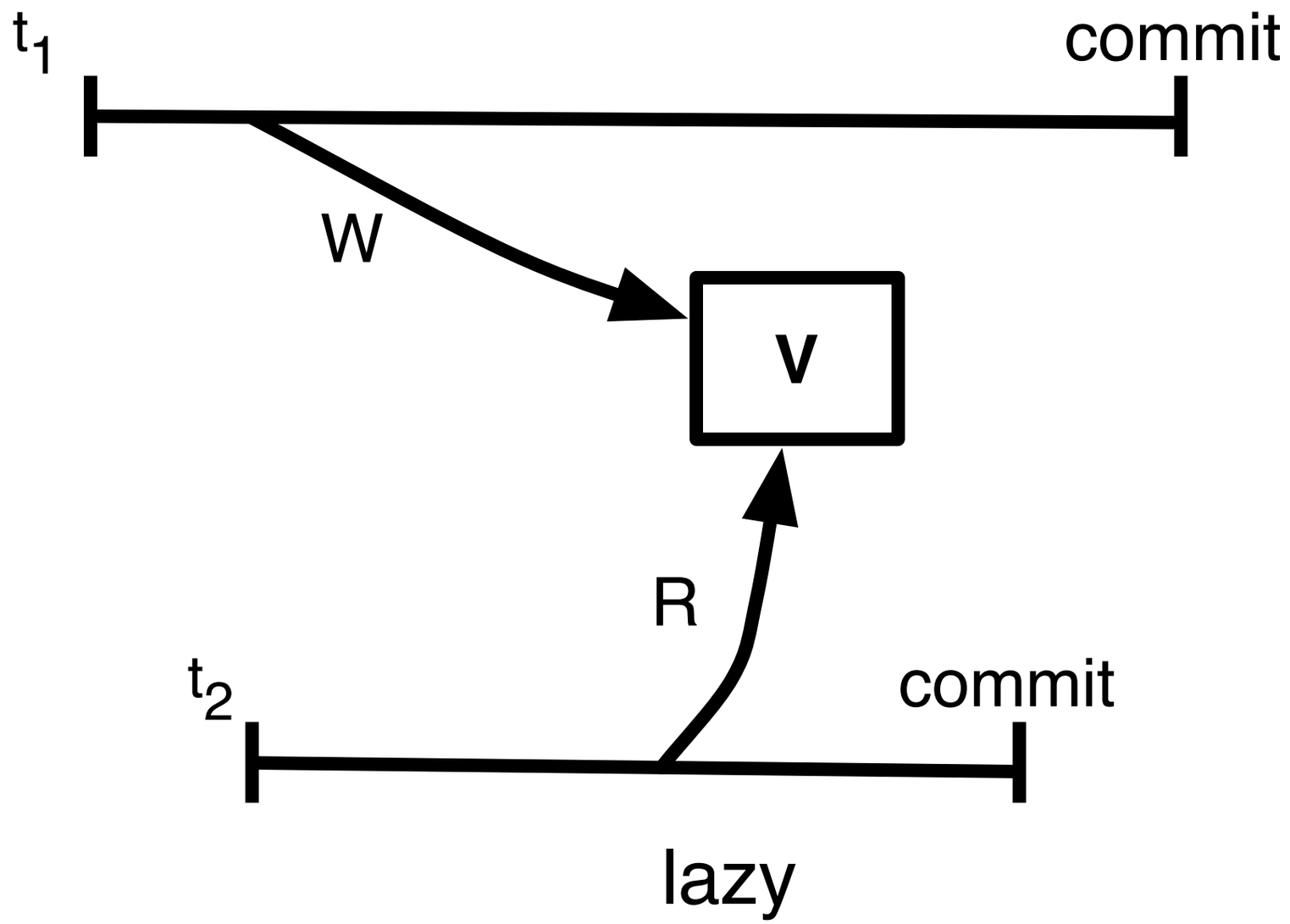
Conflict Detection

lazy > eager

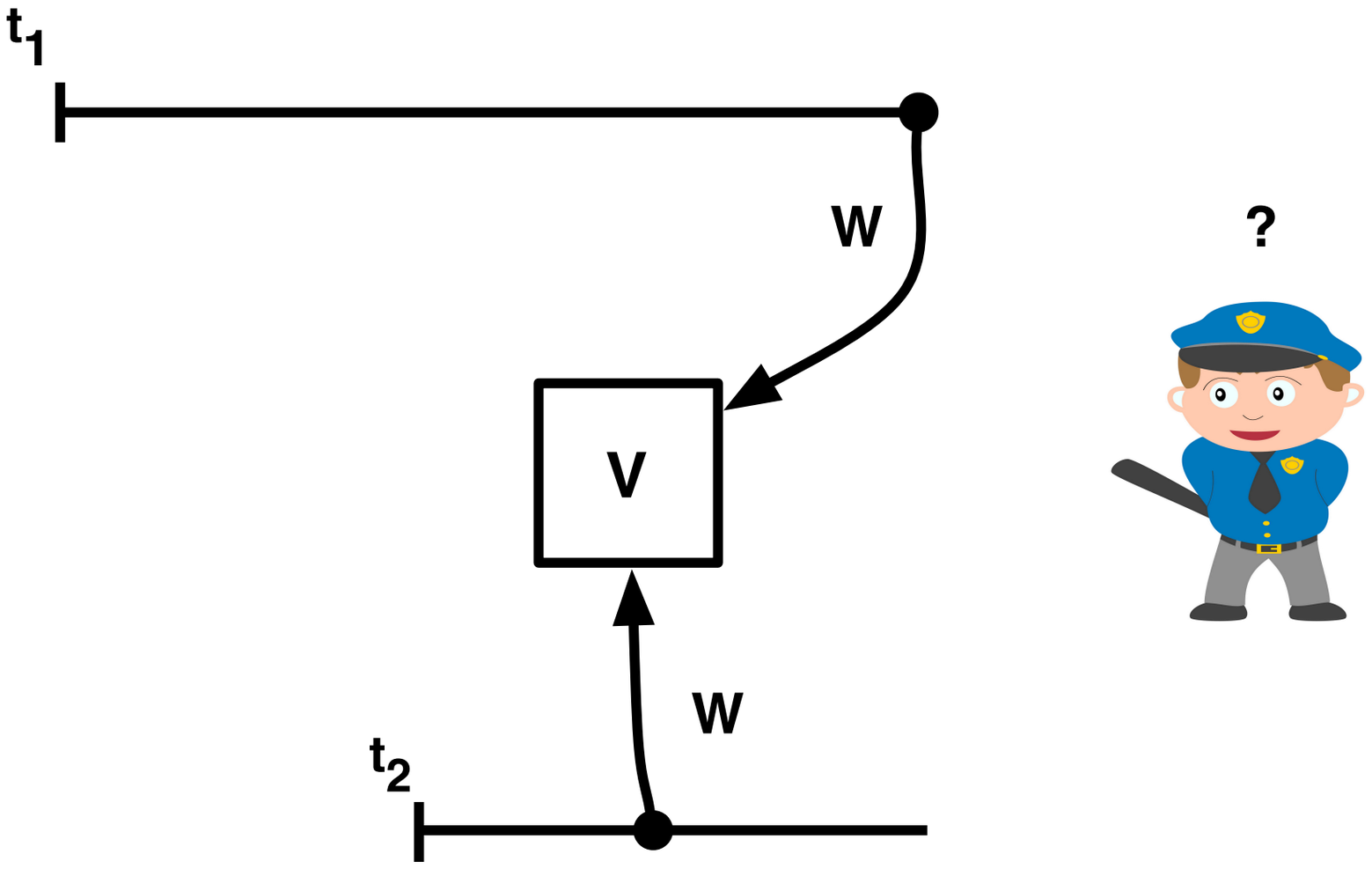


Conflict Detection

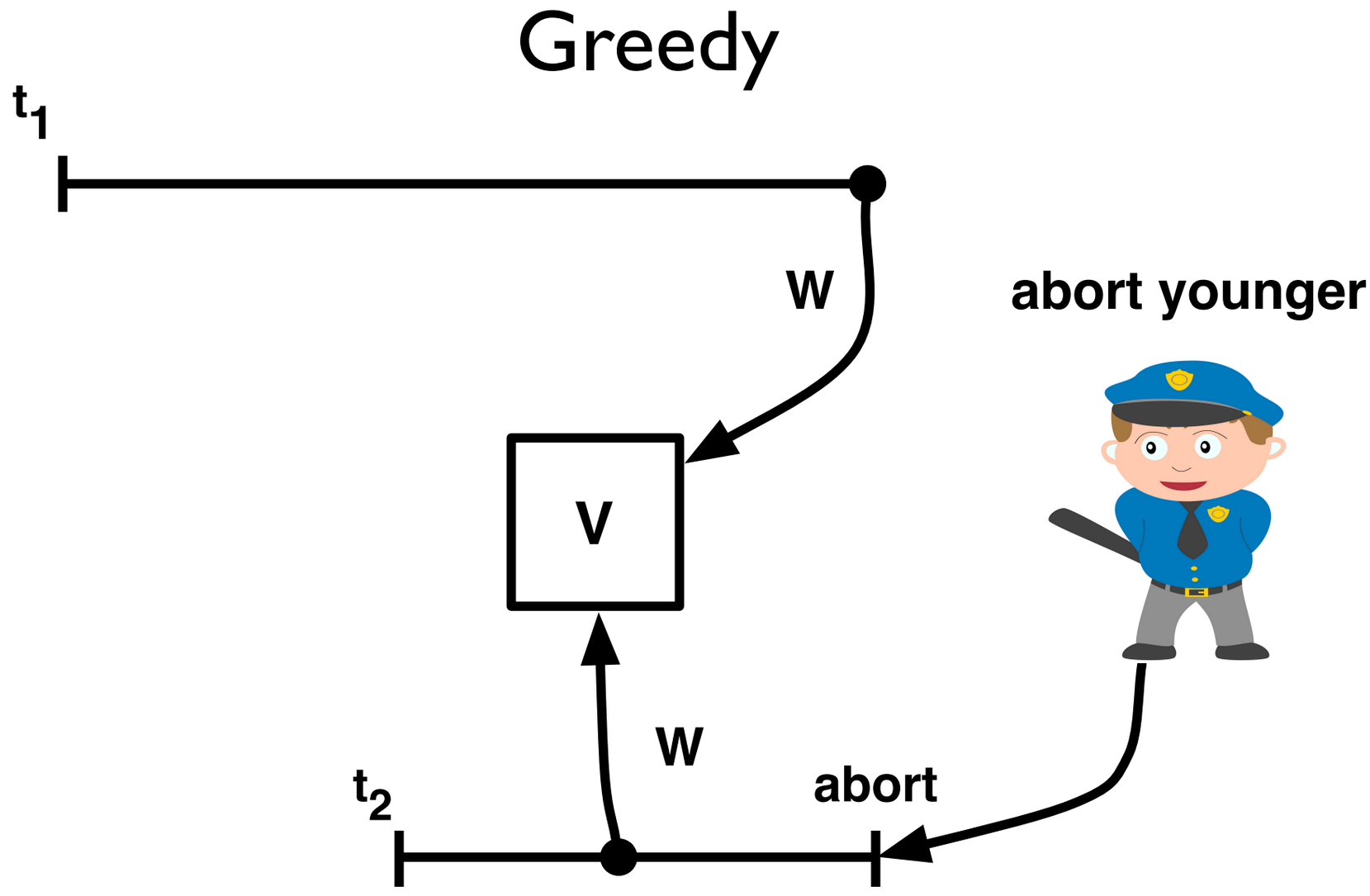
lazy > eager



Contention Manager

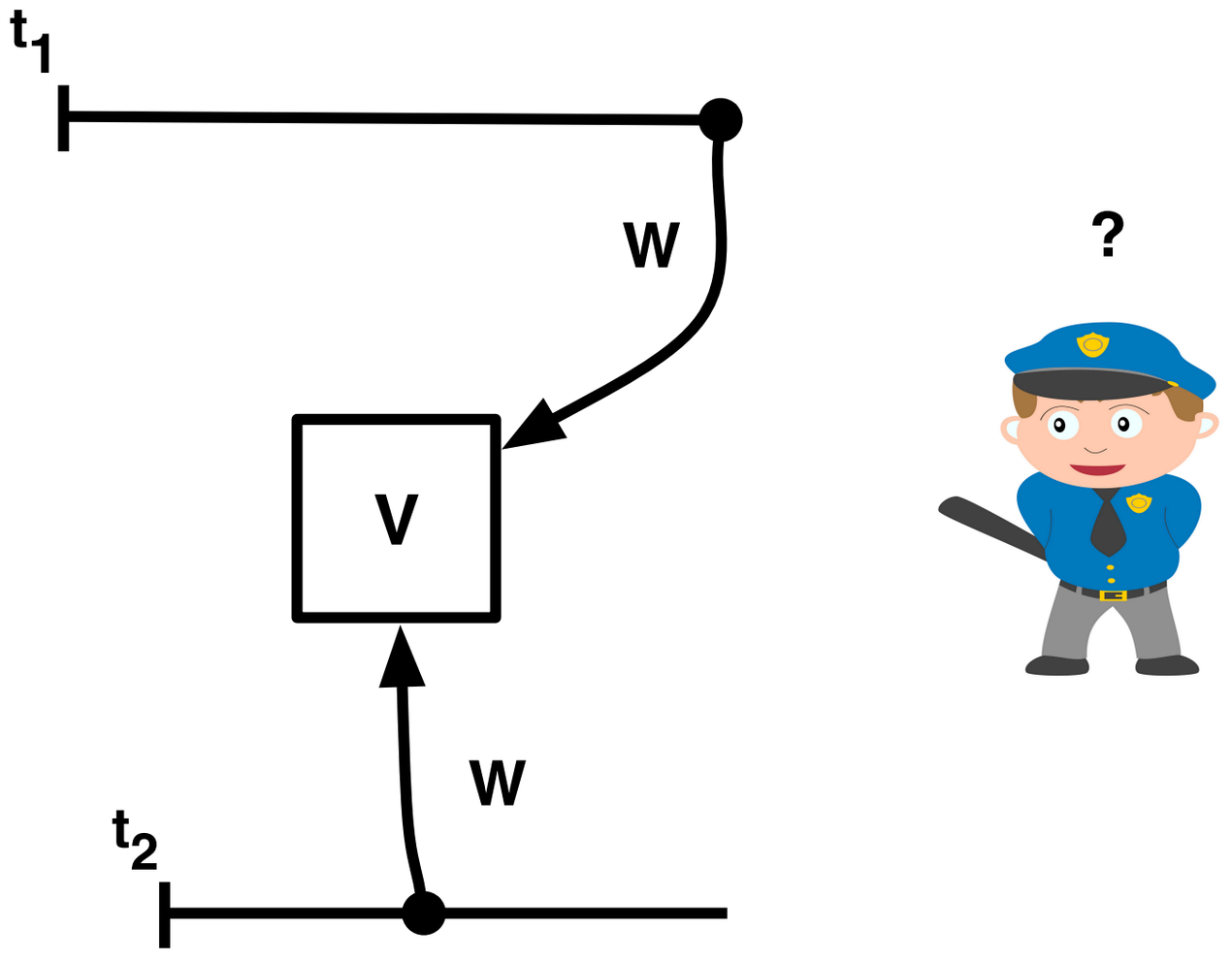


Contention Manager

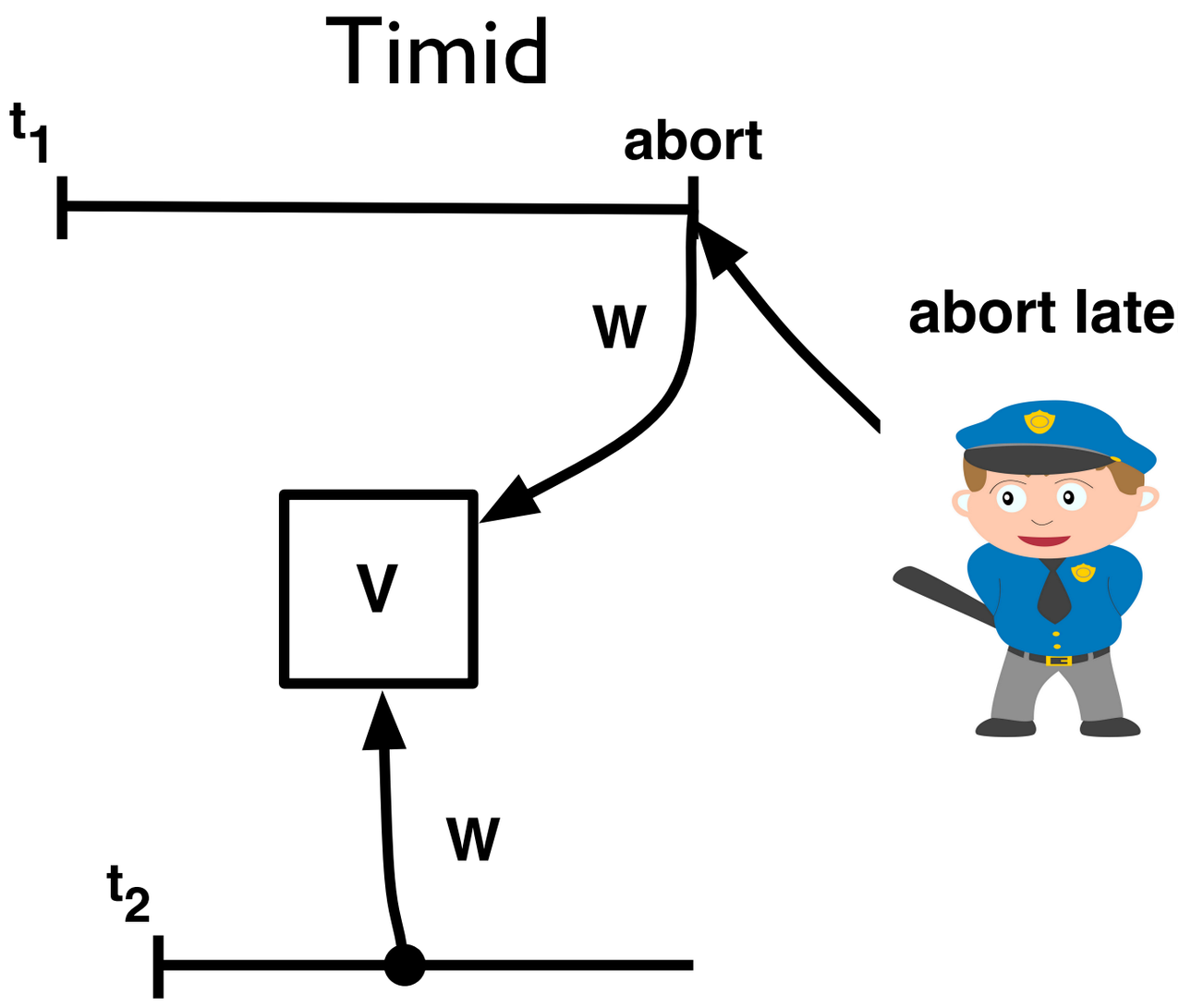


Contention Manager

Contention Manager



Contention Manager



SwissTM

- Mixed invalidation
  - eager write/write
  - lazy read/write

SwissTM

- Mixed invalidation
  - eager write/write
  - lazy read/write
- Two-phase contention manager
  - Timid for short transactions
  - Greedy for long transactions
  - Backoff on abort

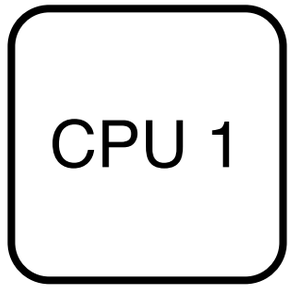
SwissTM Implementation

SwissTM Implementation

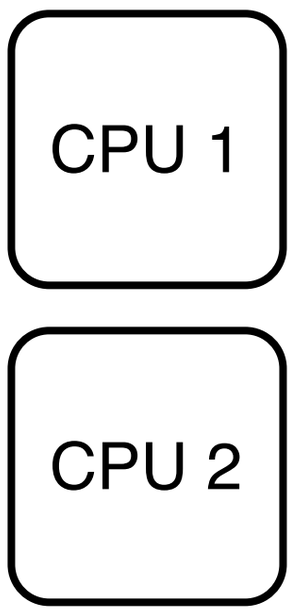
How is the high level view implemented?

Model

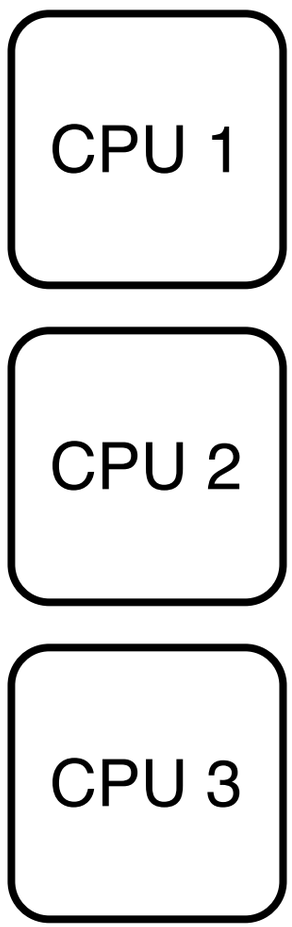
Model



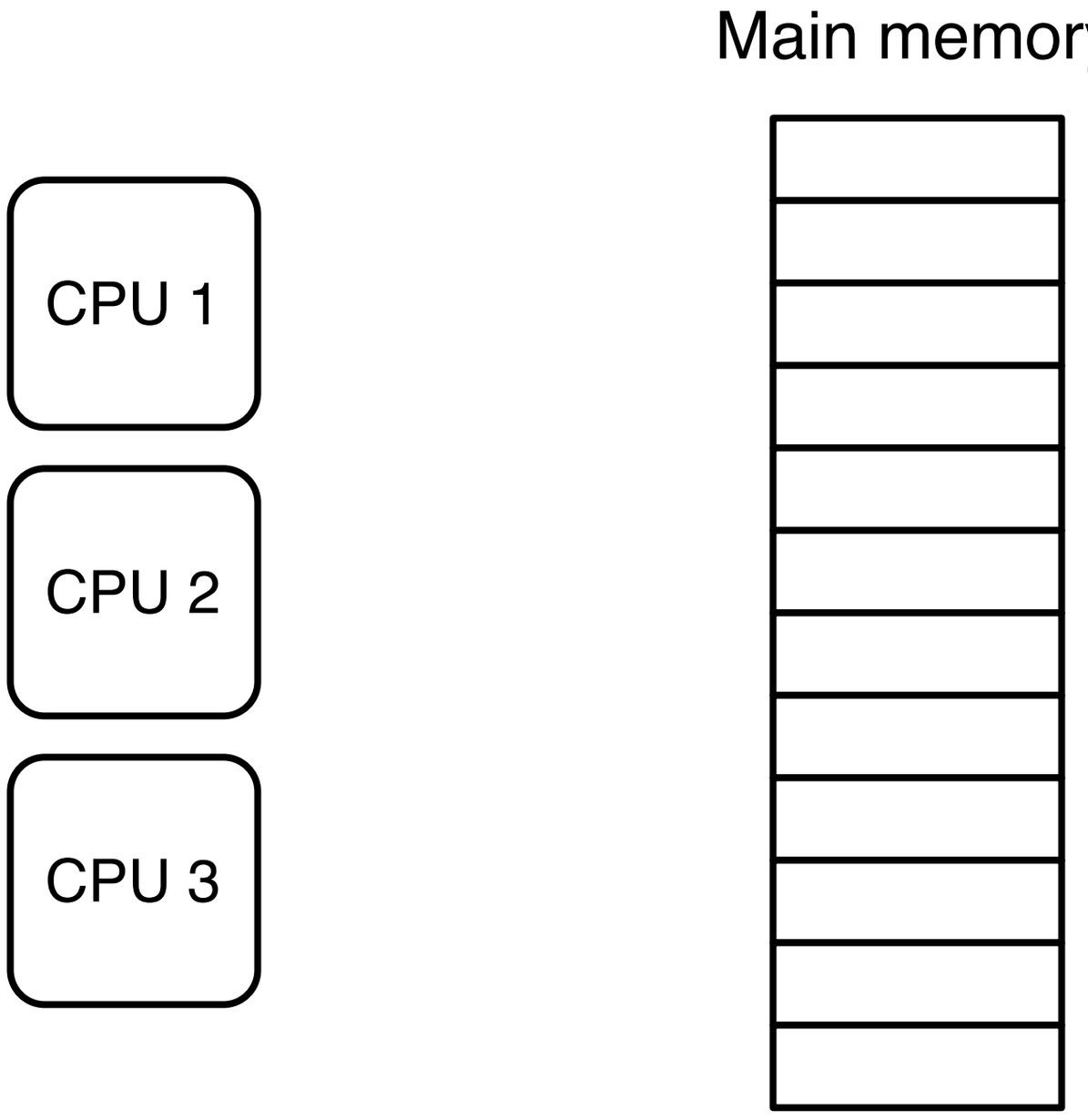
Model



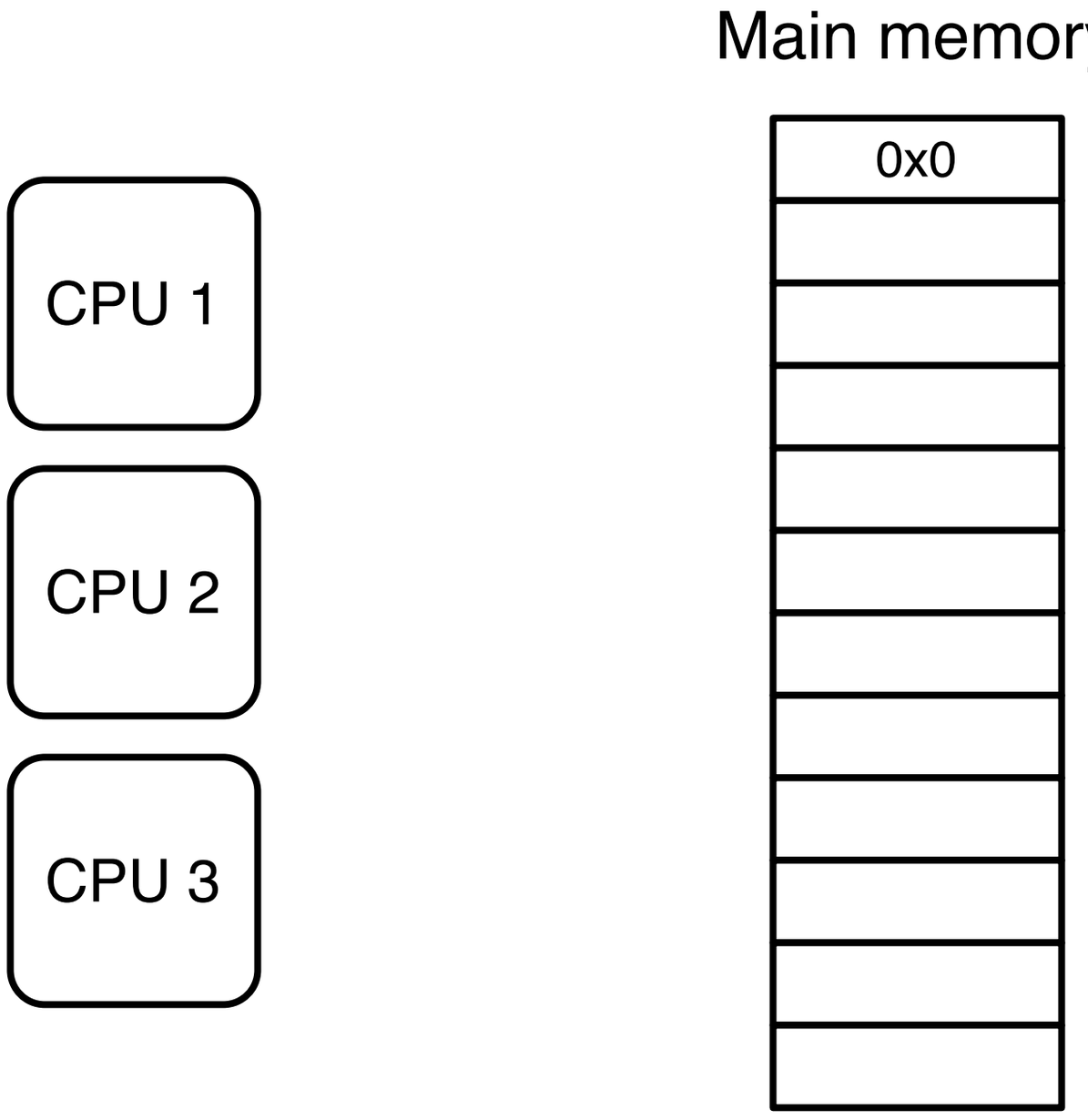
Model



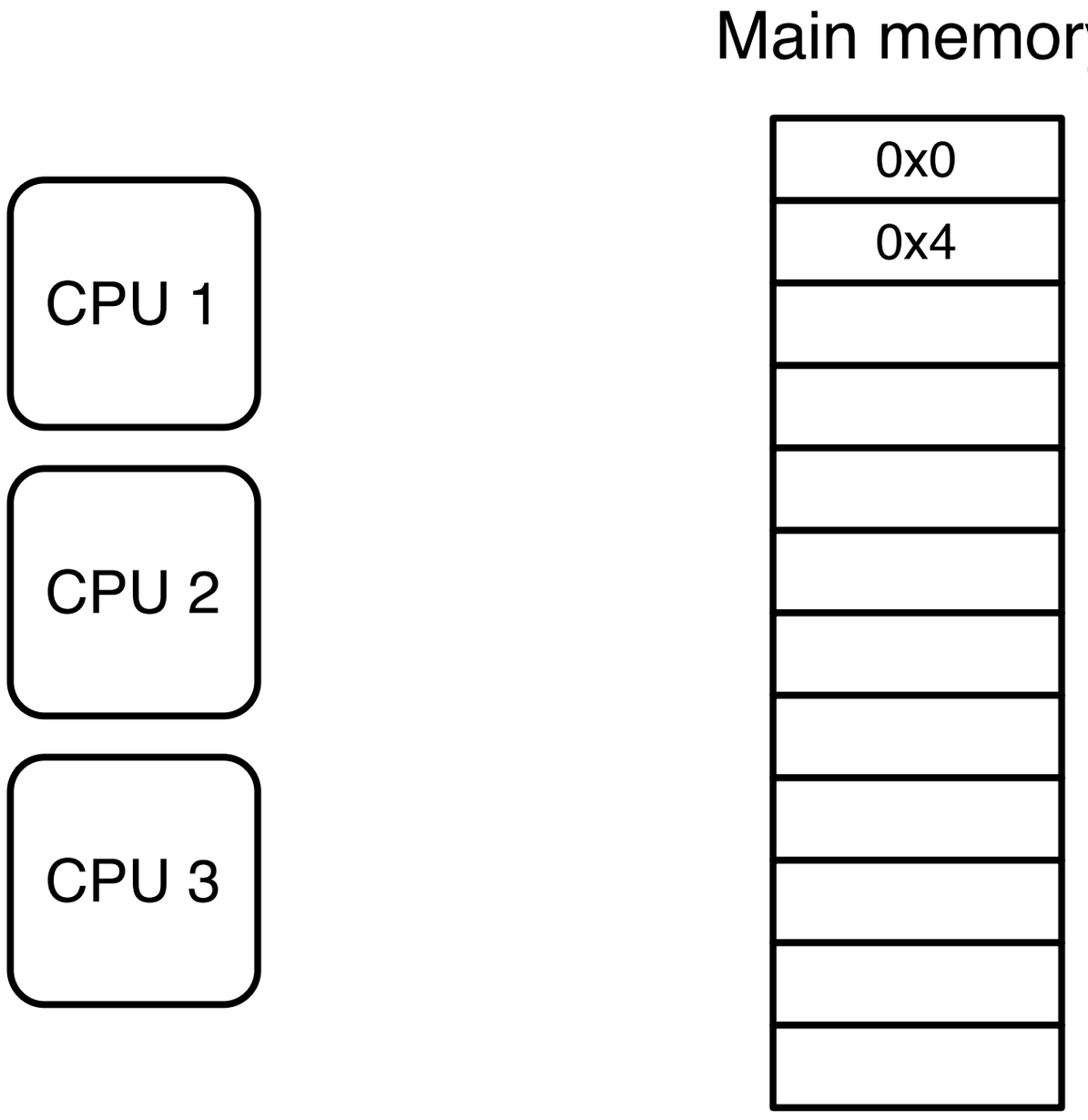
Model



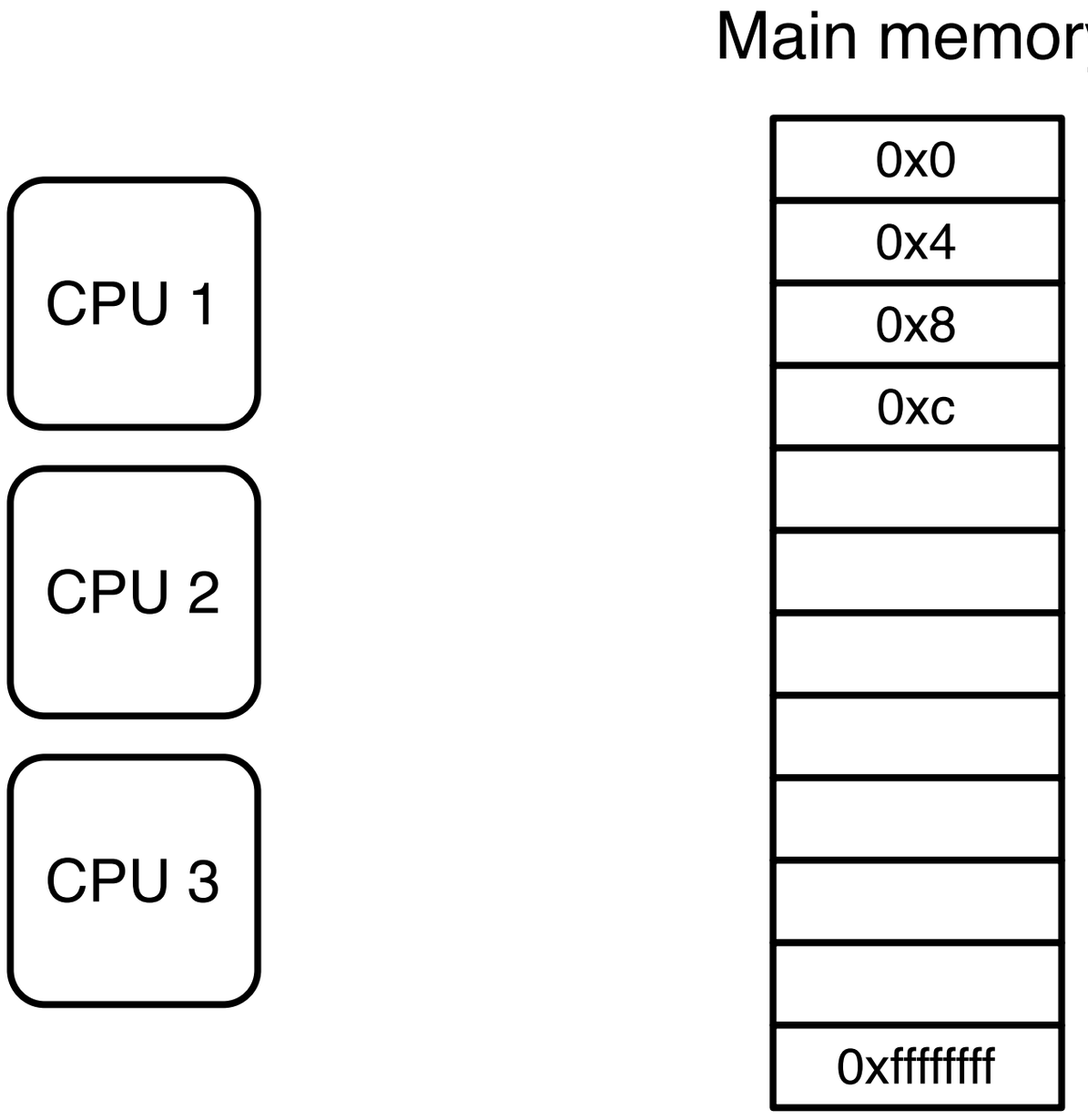
Model



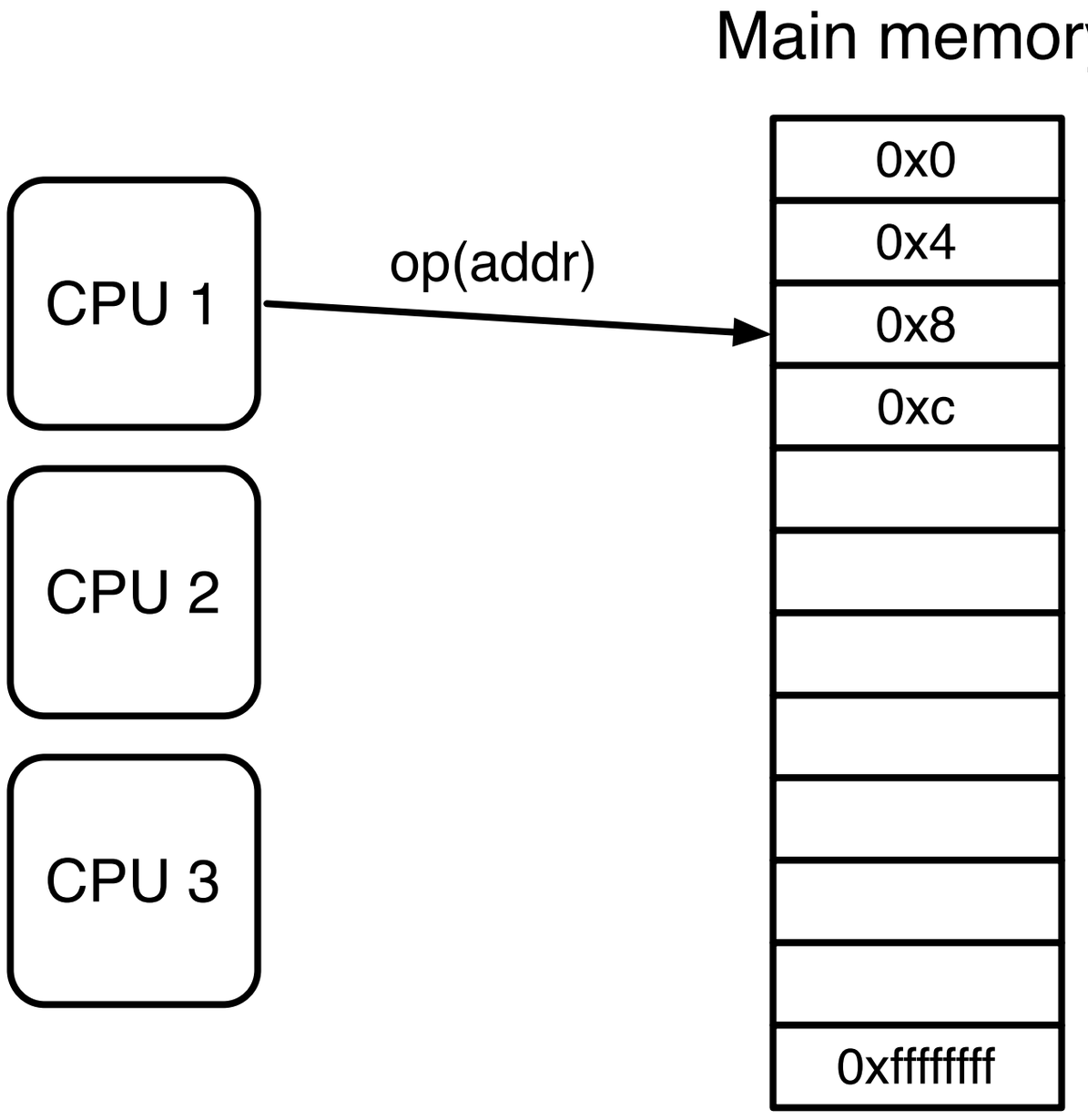
Model



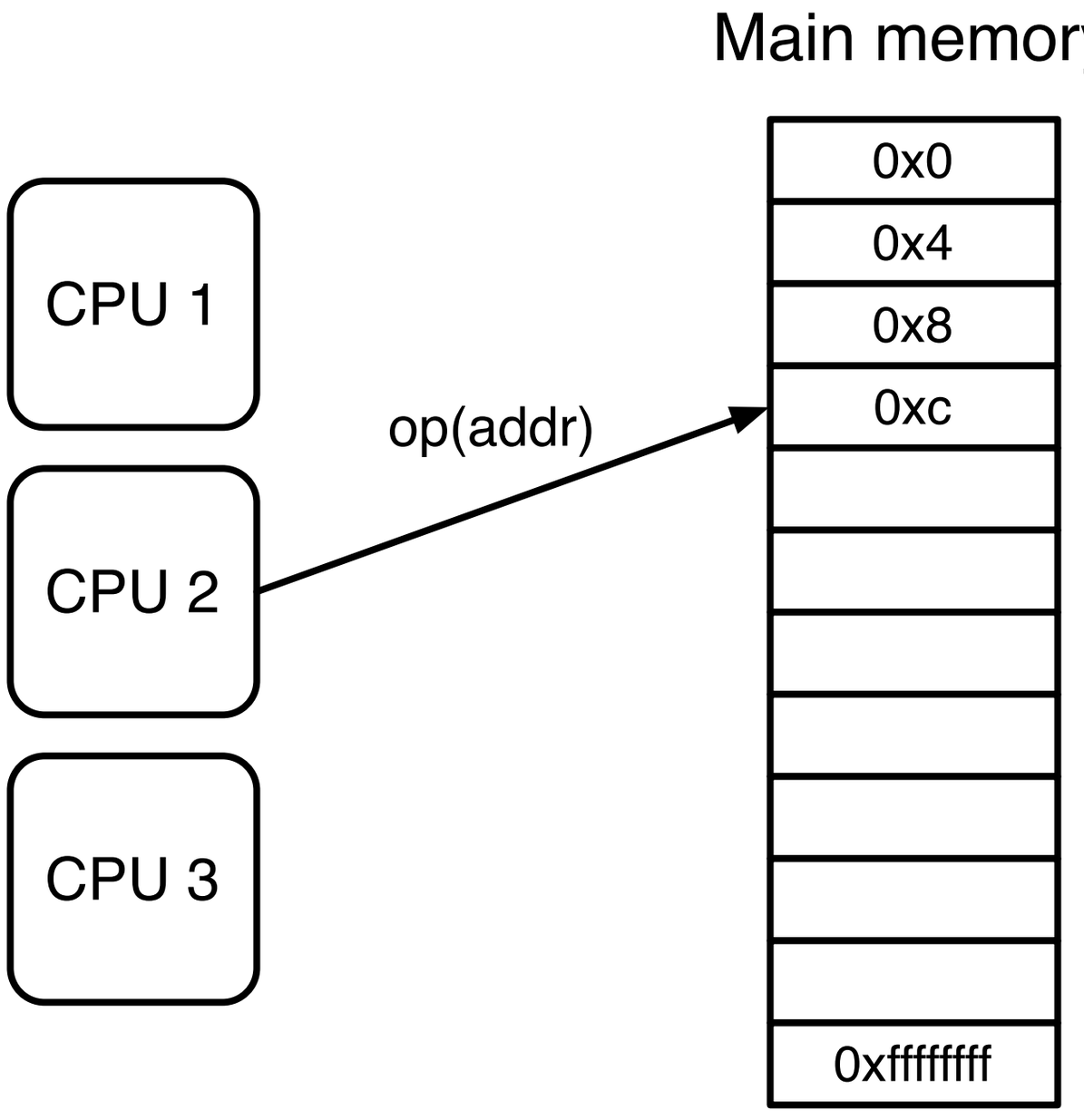
Model



Model


Main memory
CPU 1
CPU 2
CPU 3
op(addr)
0x0
0x4
0x8
0xc
0xffffffff

Model


Main memory
CPU 1
CPU 2
CPU 3
op(addr)
0x0
0x4
0x8
0xc
0xffffffff

Model (2)

- Memory consists of locations
  - word sized (32 bits)
- Each location has address
- CPUs execute operations on locations
  - read, write, c&s, t&s, ...

Model (3)

- Operations have different costs
- Try to avoid expensive operations
  - c&s
  - writing shared data
  - reading shared data

Starting point

```
void tx_start()

word_t tx_read(word_t *addr)

void tx_write(word_t *addr, word_t val)

void tx_commit()
```

Link to previous lecture

Link to previous lecture

Every object *O*, with state *s(O)* (a register),
is protected by a lock *l(O)* (a c&s)

Link to previous lecture

Every object *O*, with state *s(O)* (a register), is protected by a lock *l(O)* (a c&s)

Every memory location *addr*, with state *value(addr)*, is protected by a lock *l*

Link to previous lecture

Every object *O*, with state *s(O)* (a register), is protected by a lock *l(O)* (a c&s)

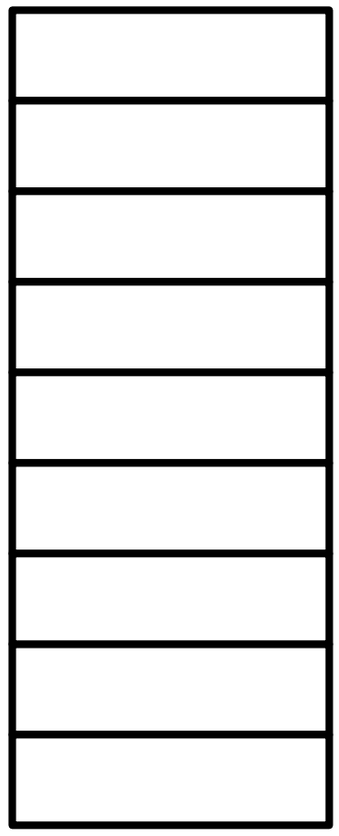
Every memory location *addr*, with state *value(addr)*, is protected by a lock *l*

Where is the lock?

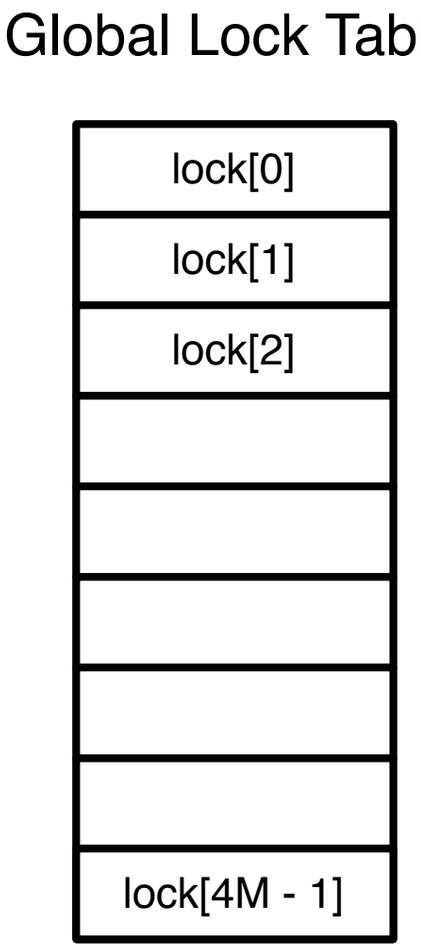
Where is the lock?

Where is the lock?

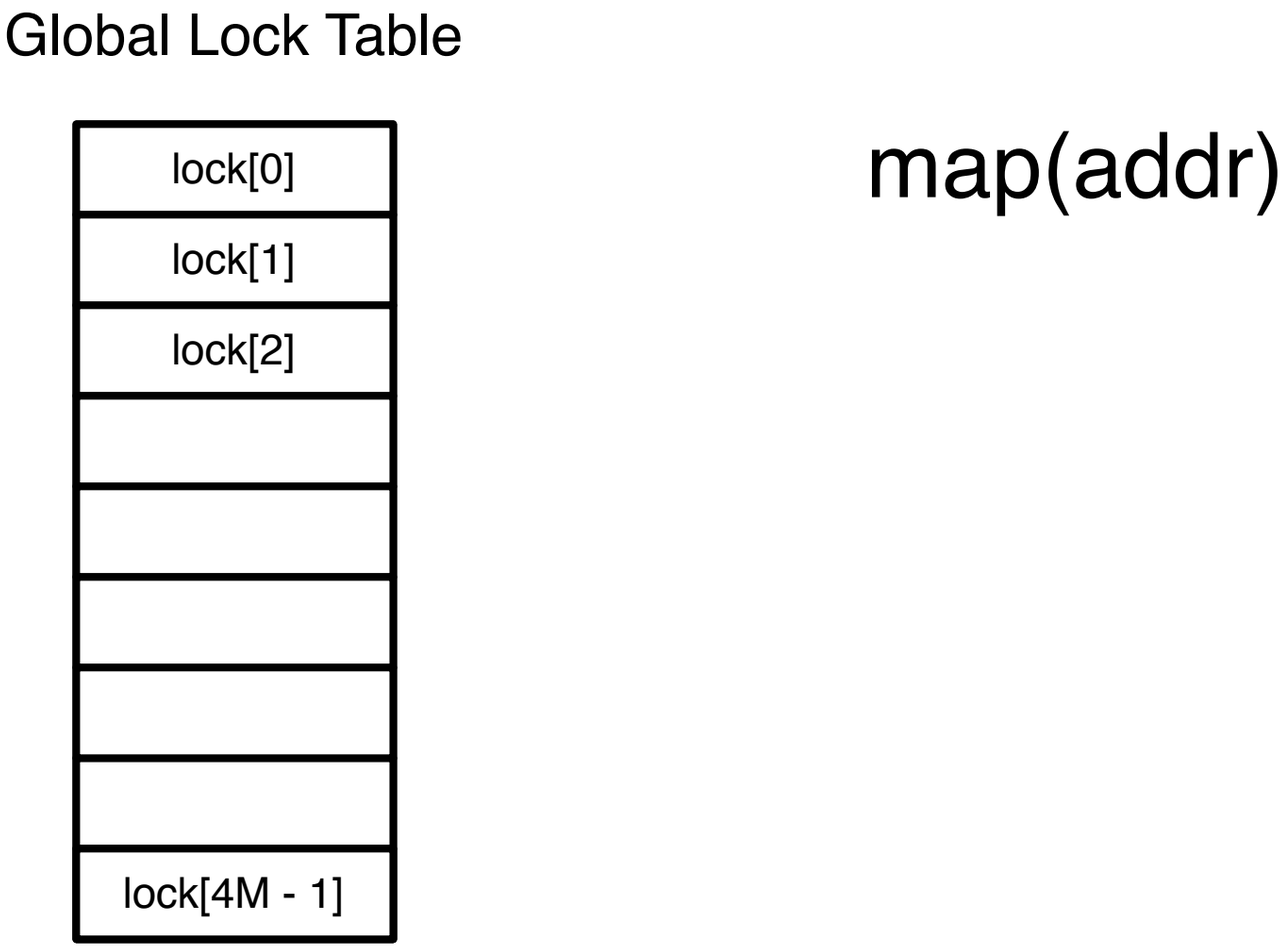
Global Lock Table



Where is the lock?

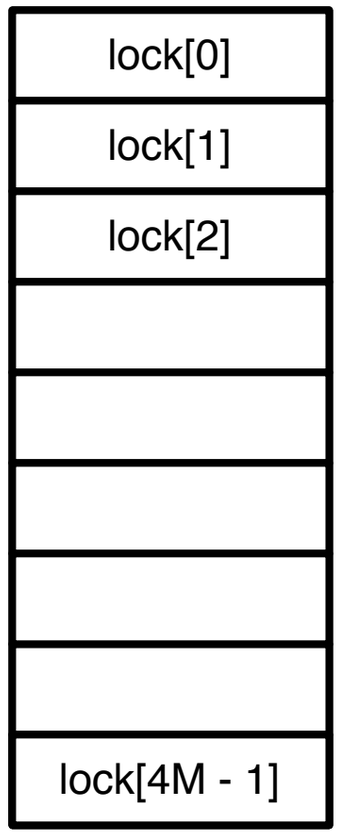


Where is the lock?



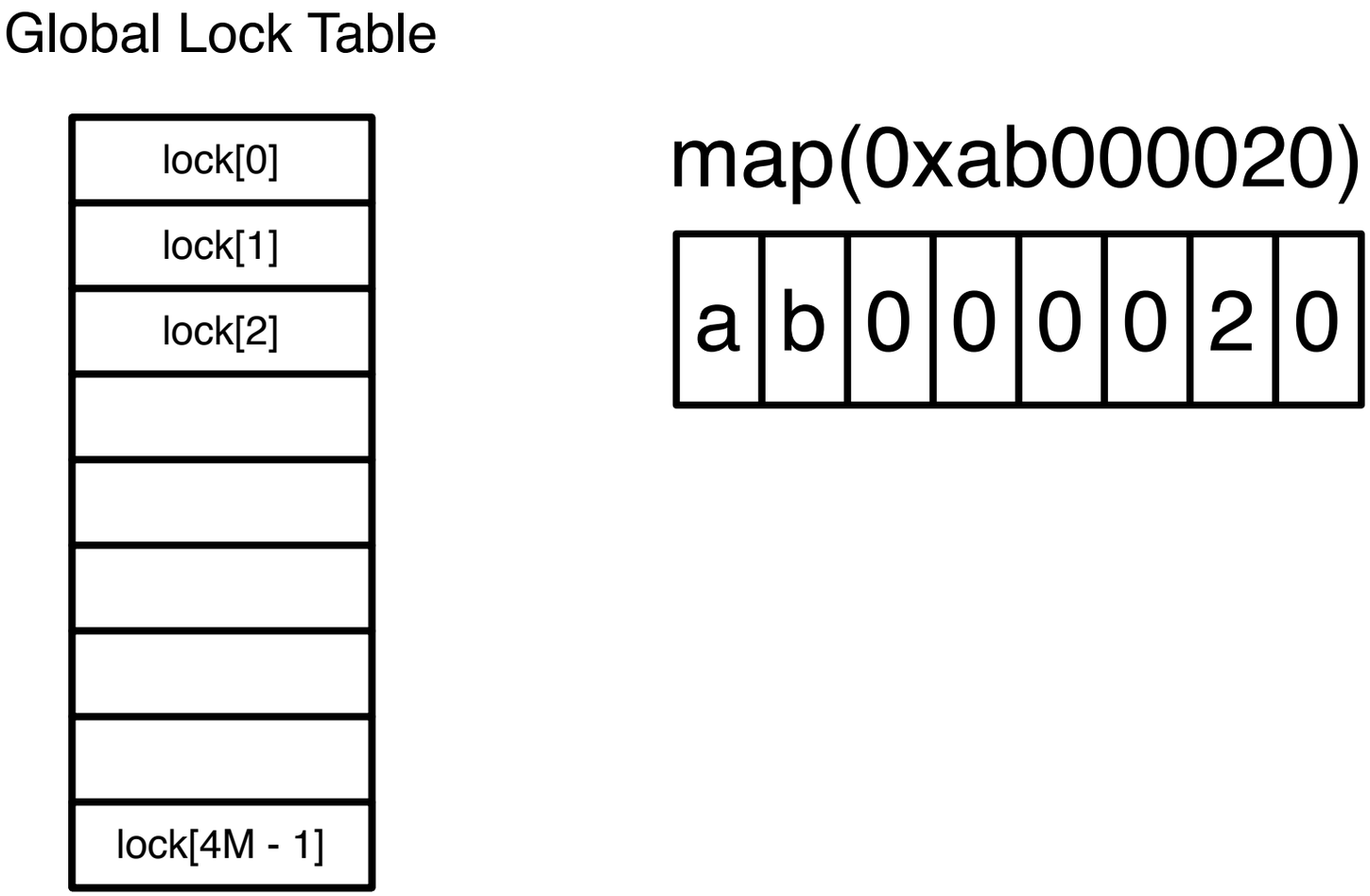
Where is the lock?

Global Lock Table



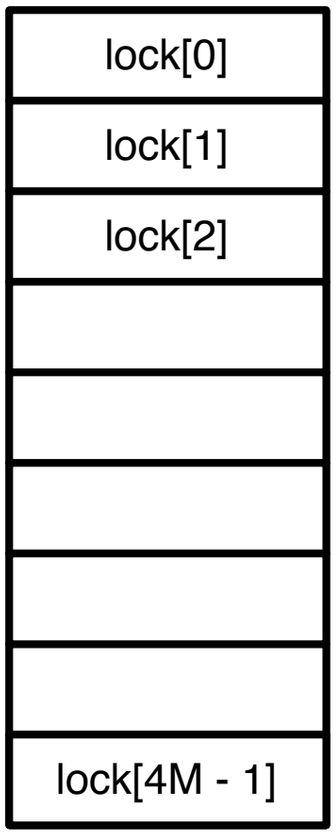
map(0xab000020)

Where is the lock?

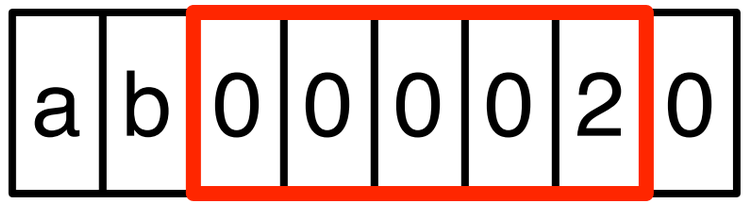


Where is the lock?

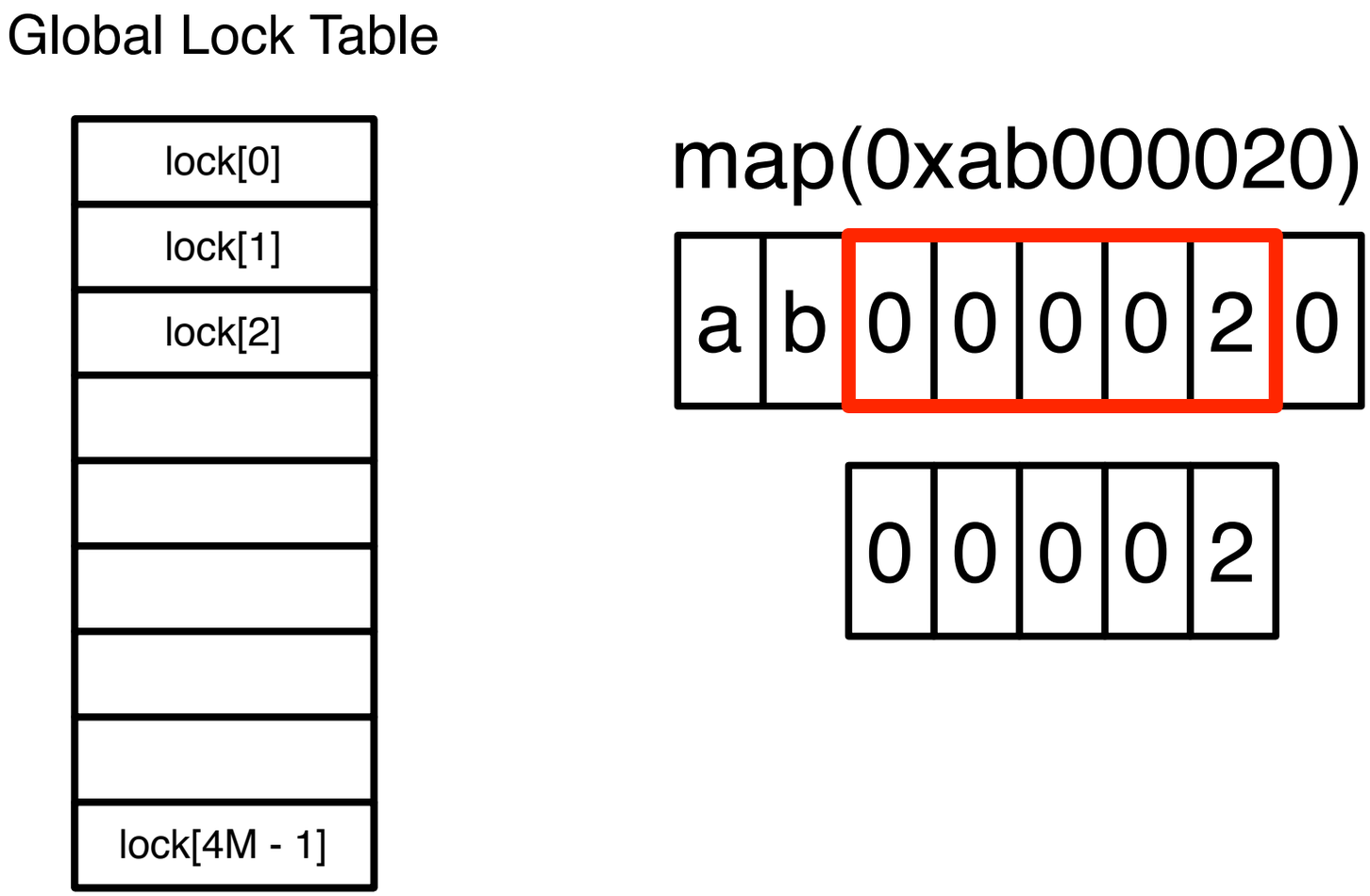
Global Lock Table



map(0xab000020)

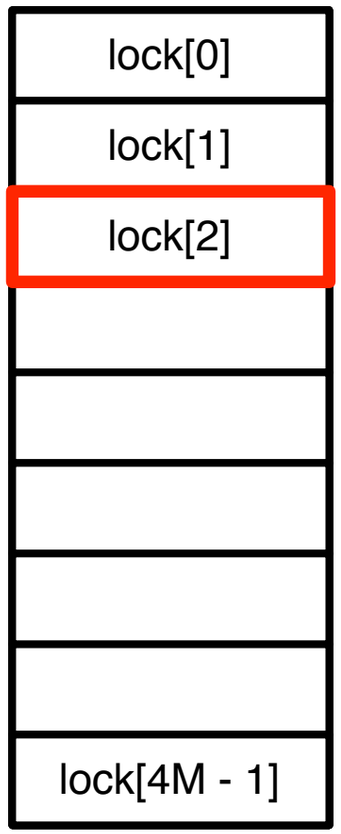


Where is the lock?

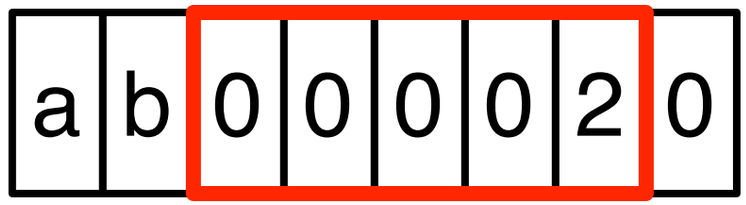


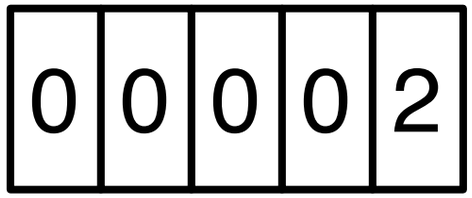
Where is the lock?

Global Lock Table



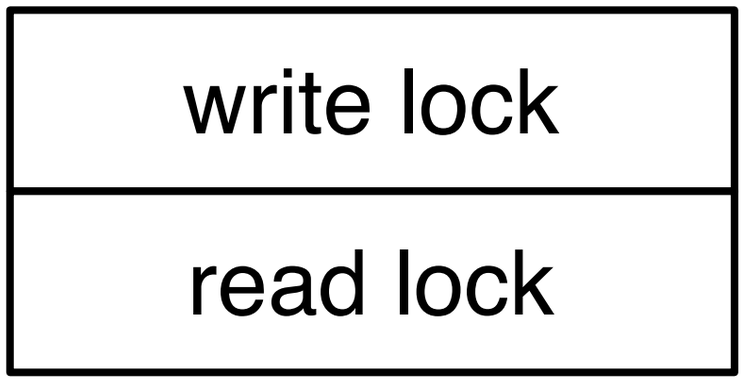
map(0xab000020)





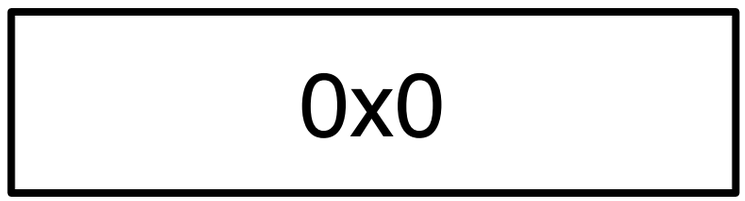
Lock

Lock



Lock

unlocked write lock



Lock

unlocked write lock

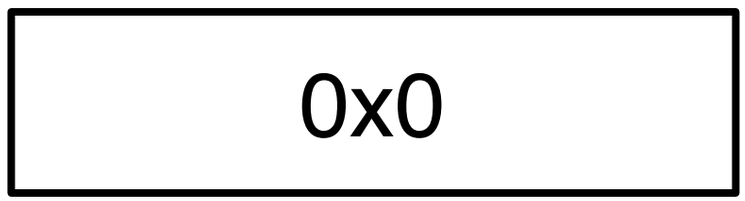
0x0

locked write lock

owner log entry ptr

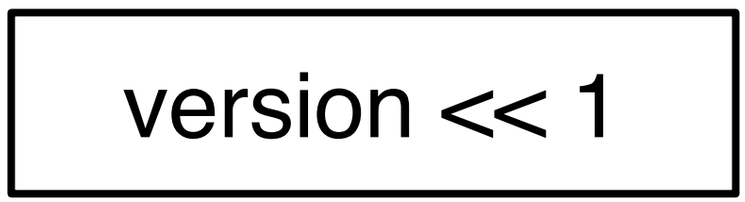
Lock

unlocked write lock



locked write lock

unlocked read lock



Lock

unlocked write lock

0x0

locked write lock

owner log entry ptr

unlocked read lock

version << 1

locked read lock

0x1

Lock (2)

- Two memory words
- Write lock
  - detect write/write conflicts
- Read lock
  - detect read/write conflicts

Versions

- Each location has a version
- Use shared version counter
  - speeds up validation
- Every transaction that writes, updates the counter on commit

Versions (2)

- Transactions read version counter at start
- If version counter has not changed ⇒
  no transaction updated anything ⇒
  read set is consistent
- If it has, validate
- Remember current version counter

Start

```
1: tx_start():
2:    _start := create_jump_target()
3:    _valid_ts := read(Commit_ts)
```

Read

```
1: tx_read(word_t *addr)
2:    (r_lock, w_lock) := map(addr)
3:    if locked_by_me(w_lock) return get_val(w_lock)
4:    version := read(r_lock)
5:    while true
6:        if version = 1
7:            version := read(r_lock)
8:            continue
9:        value := read(addr)
10:       version2 := read(r_lock)
11:       if version = version2 break
12:   read_log_add(r_lock, version)
13:   if version > valid_ts and not extend():
14:       rollback()
15:   return value
```

Extend

```
1: extend()
2:     ts := read(Commits_ts)
3:     if validate()
4:       _valid_ts := ts
5:       return true
6:   return false
```

Validate

```
1: validate()
2:    for entry in _read_log
3:       if entry.version = read(entry.r_lock)
4:           continue
5:       if locked_by_me(entry.w_lock)
6:           continue
7:       return false
8:    return true
```

Write

```
1: tx_write(word_t *addr, word_t val)
2:    (r_lock, w_lock) := map(addr)
3:    if locked_by_me(w_lock)
4:       update(w_lock, val)
5:       return
6:    while true
7:       if w_lock != 0x0
8:          if cm_should_abort(w_lock) rollback()
9:          else continue
10:      entry := write_log_add(w_lock,addr,val)
11:      if c&s(w_lock, 0, entry)
12:         break
13:   if read(r_lock) > valid_ts and not extend():
14:      rollback()
```

Commit

```
1:  tx_commit()
2:    if empty(_write_log) return
3:    for entry in _write_log
4:      write(entry.r_lock,1)
4:    ts := increment(Commits ts)
5:    if ts > _valid_ts + 1 and not validate
6:      for entry in _write_log
7:        write(eentry.r_lock, entry.version)
8:      rollback()
9:    for entry in _write_log
10:     write(entry.addr, entry.value)
11:     write(entry.r_lock, ts << 1)
12:     write(entry.w_lock, 0)
```

Rollback

```
1: rollback()
2:    for entry in _write_log
3:       write(entry.w_lock, 0)
4:    long_jump(_start)
```

Contention manager

```
1: on_start()
2:    _cm_ts := ∞

3: on_write()
4:    if _cm_ts = ∞ and size(_write_log) > 10
5:       _cm_ts := increment(Greedy_ts)

6: on_rollback()
7:    wait_random()

8: cm_should_abort(w_lock)
9:    if _cm_ts = ∞ return true
10:   owner = owner(w_lock)
11:   if owner._cm_ts < _cm_ts return true
12:   abort(owner)
13:   return false
```

Next steps

- Translate into code
  - some additional details
    - http://lpd.epfl.ch/site/research/tmeval
- Compile
- Run
- ☺

Additional details

- Memory management
  - allocations inside transactions that abort?
- Actions that cannot be rolled back
  - input / output
- Same data used by non-tx code
  - privatization / publication

Evaluating performance

Evaluating performance

How fast is our implementation?

Measuring performance

- Compare different approaches
  - different STMs
    - STM vs locking vs lock-free
- How to express performance?
  - metric
- Common approaches work

Approach 1

Approach 1

Approach 1 (2)

- Take some (long) task
  - e.g. genome sequencing
- Run with different STMs
- Faster STM needs less time

Approach II

Approach II (2)

- Take some repetitive task
  - e.g. insert element into red-black tree
- Keep executing it for some (fixed) time
- Run with different STMs
- Faster STM executes it more times

Approach III

Avoiding pitfalls

- STM1 sequences genome faster than STM2
  - does not mean STM1 solves optimization problem A faster than STM2
- No complete solution
  - run as many workloads as possible
- Be careful

STMBench7

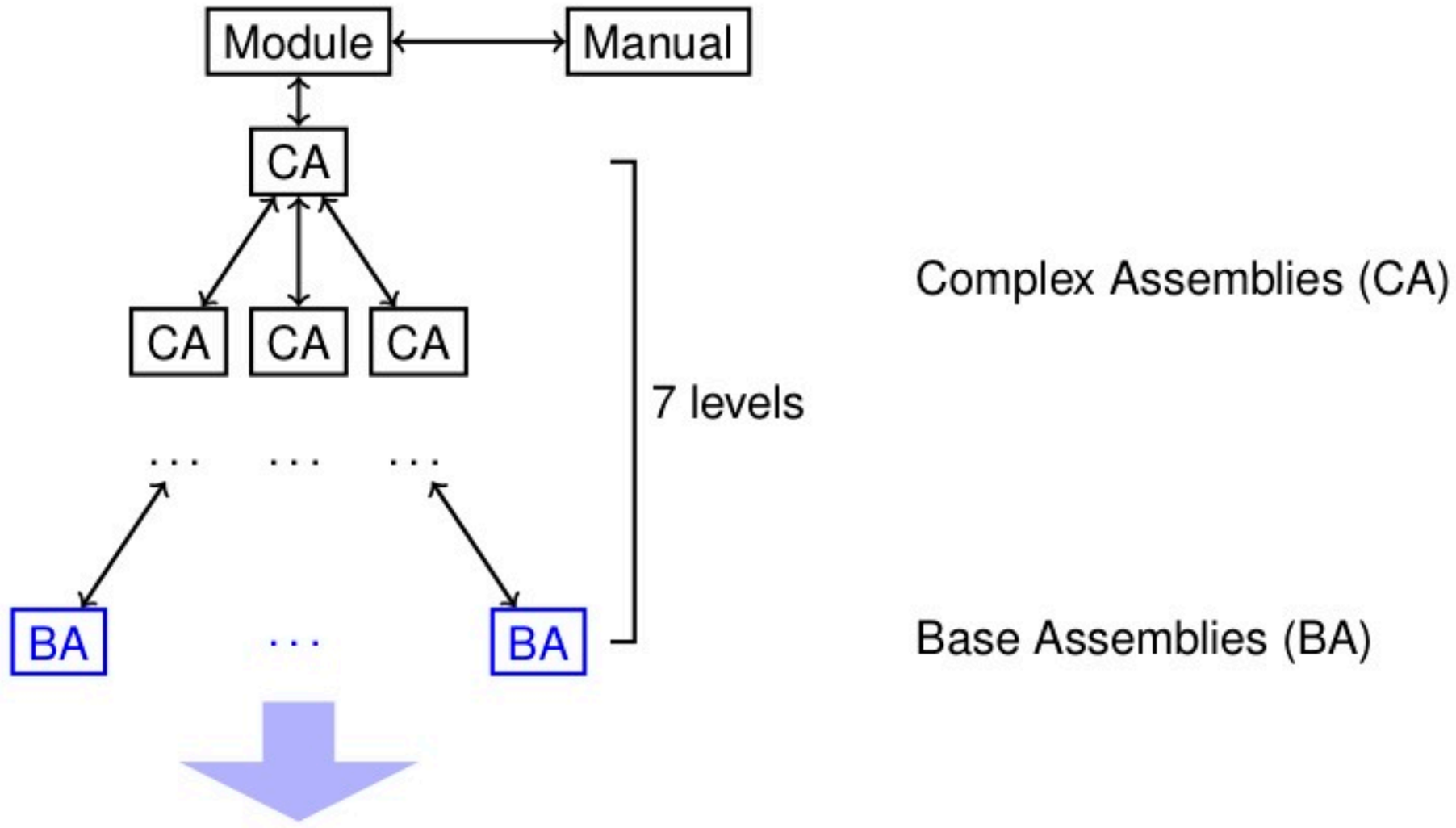
STMBench7

What does a benchmark look like?

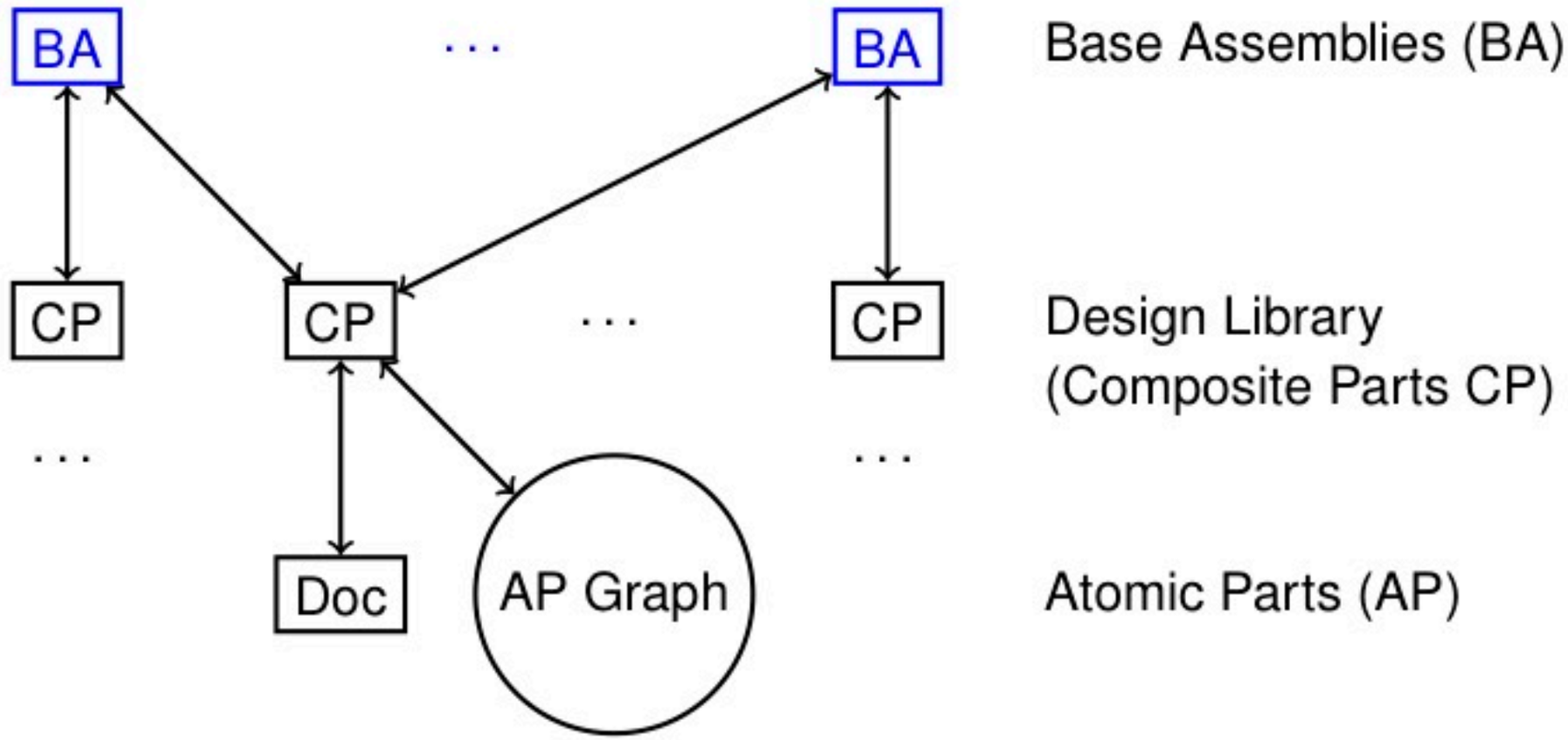
STMBench7

- Uses approach II
- Large data structure
- Modeled on OO7
  - CAD / CAM / CASE workloads
- Two locking, several STM implementations
- Crash test

Data structure



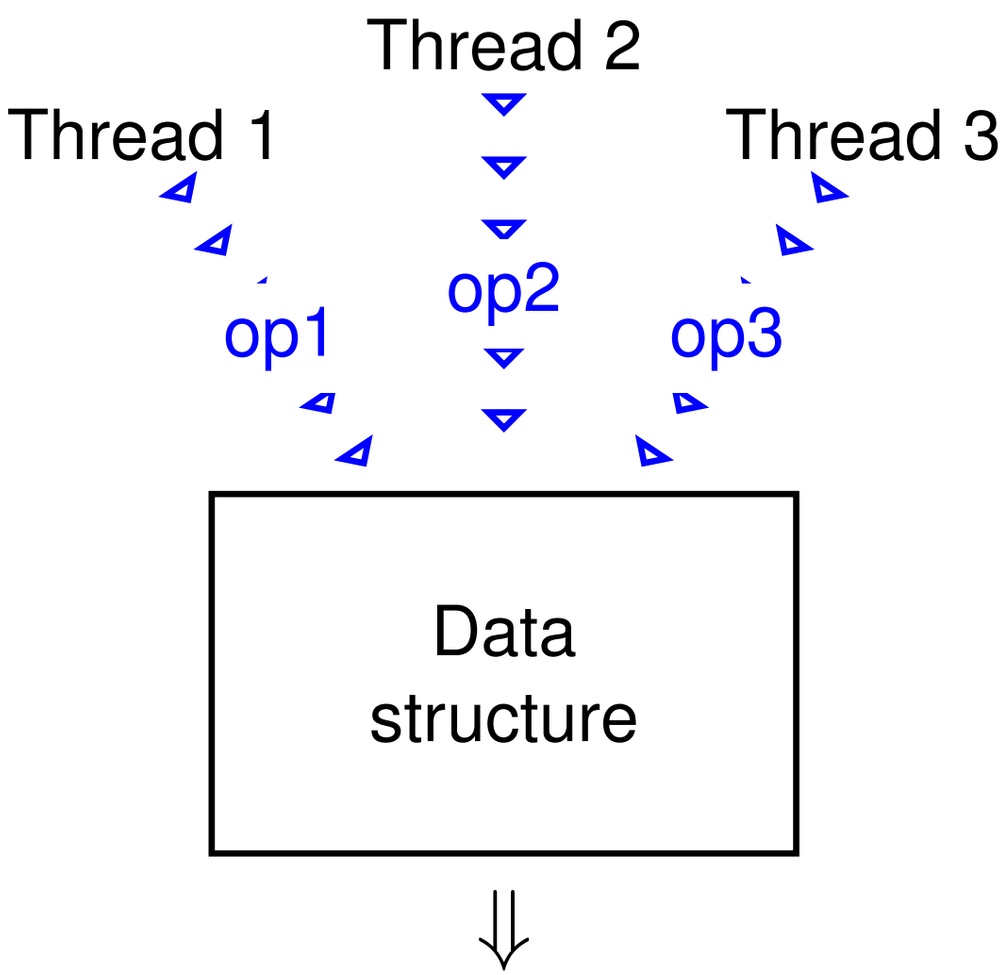
Data structure (2)



Operations

- 45 operations
- Read only and Update
- Short and long
- Three different workloads
  - read, read-write, write

Execution



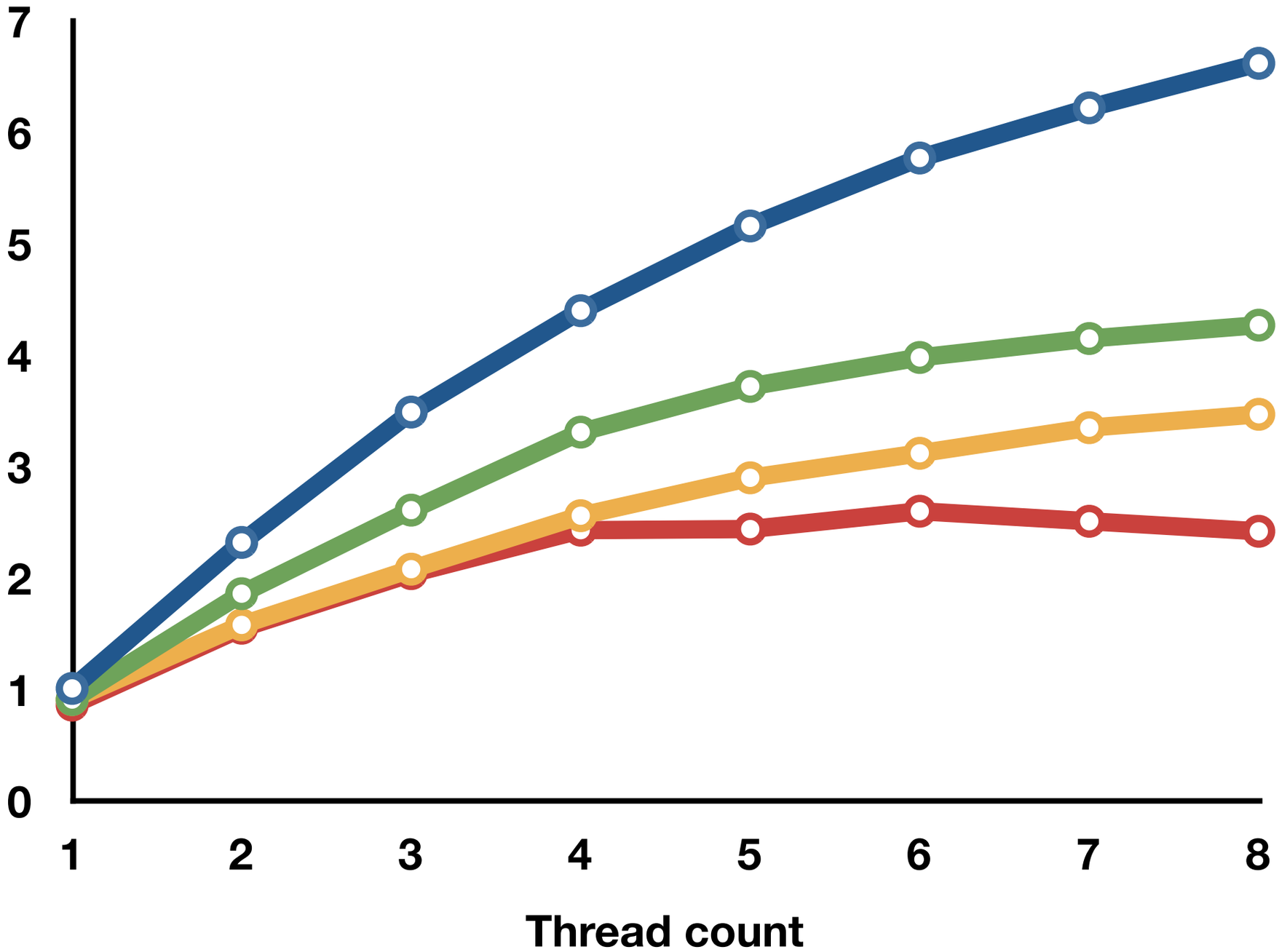
⇓

throughput

Execution (2)

- Threads execute a mix of operations
- Experiment lasts for 10s
- Measure of performance is the number of executed operations per second

STMBench7 results

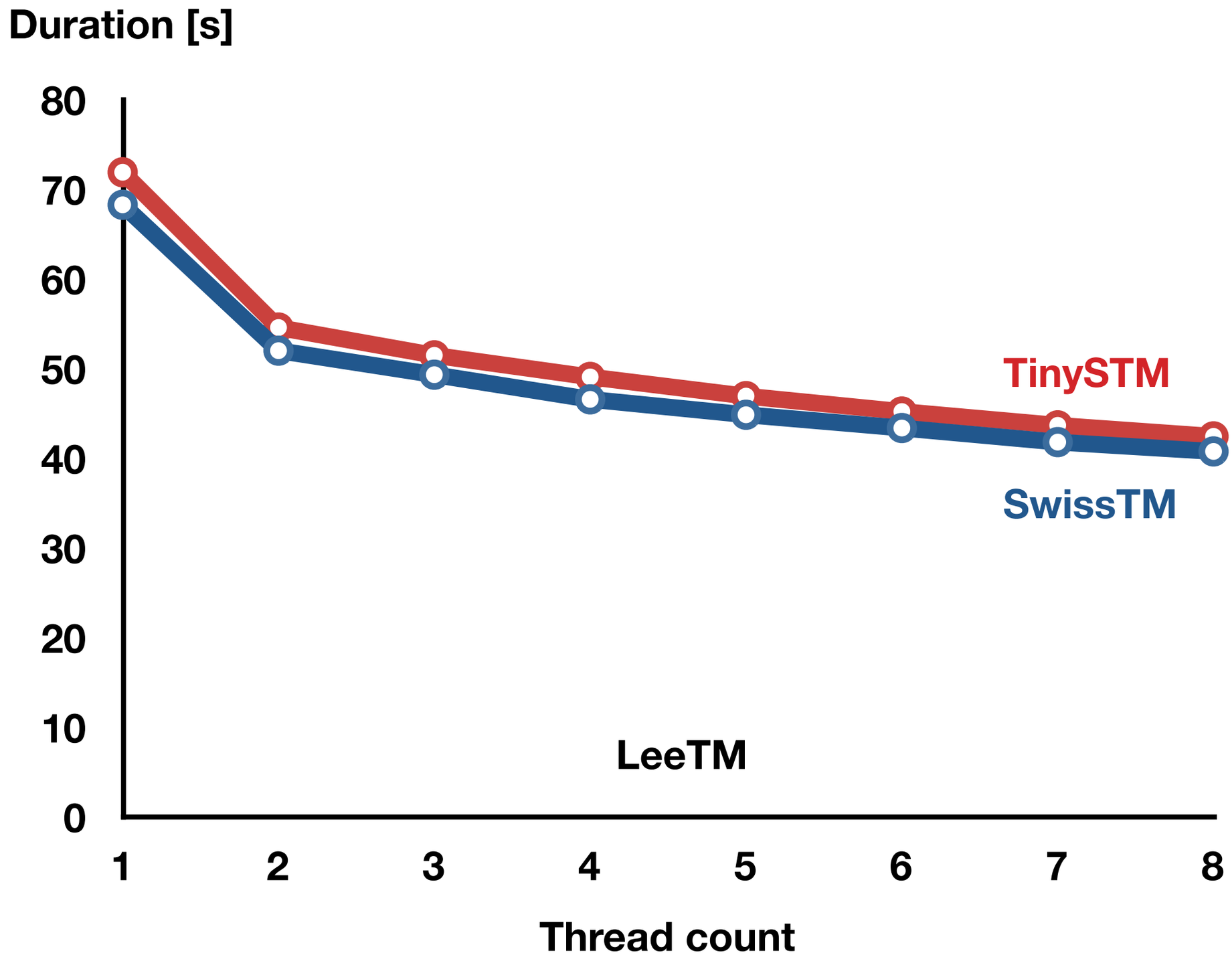


Other results

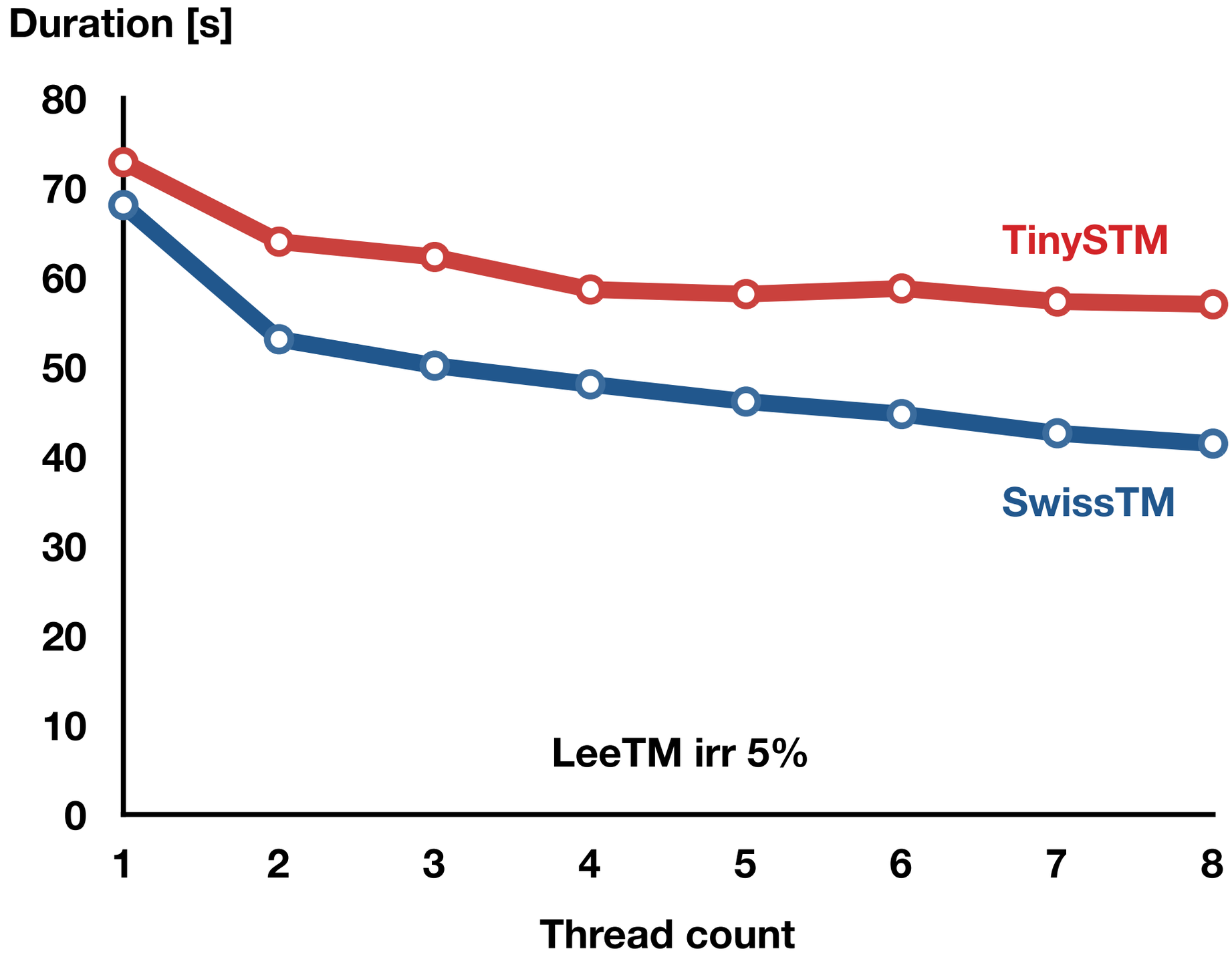
Other results

How does SwissTM perform?

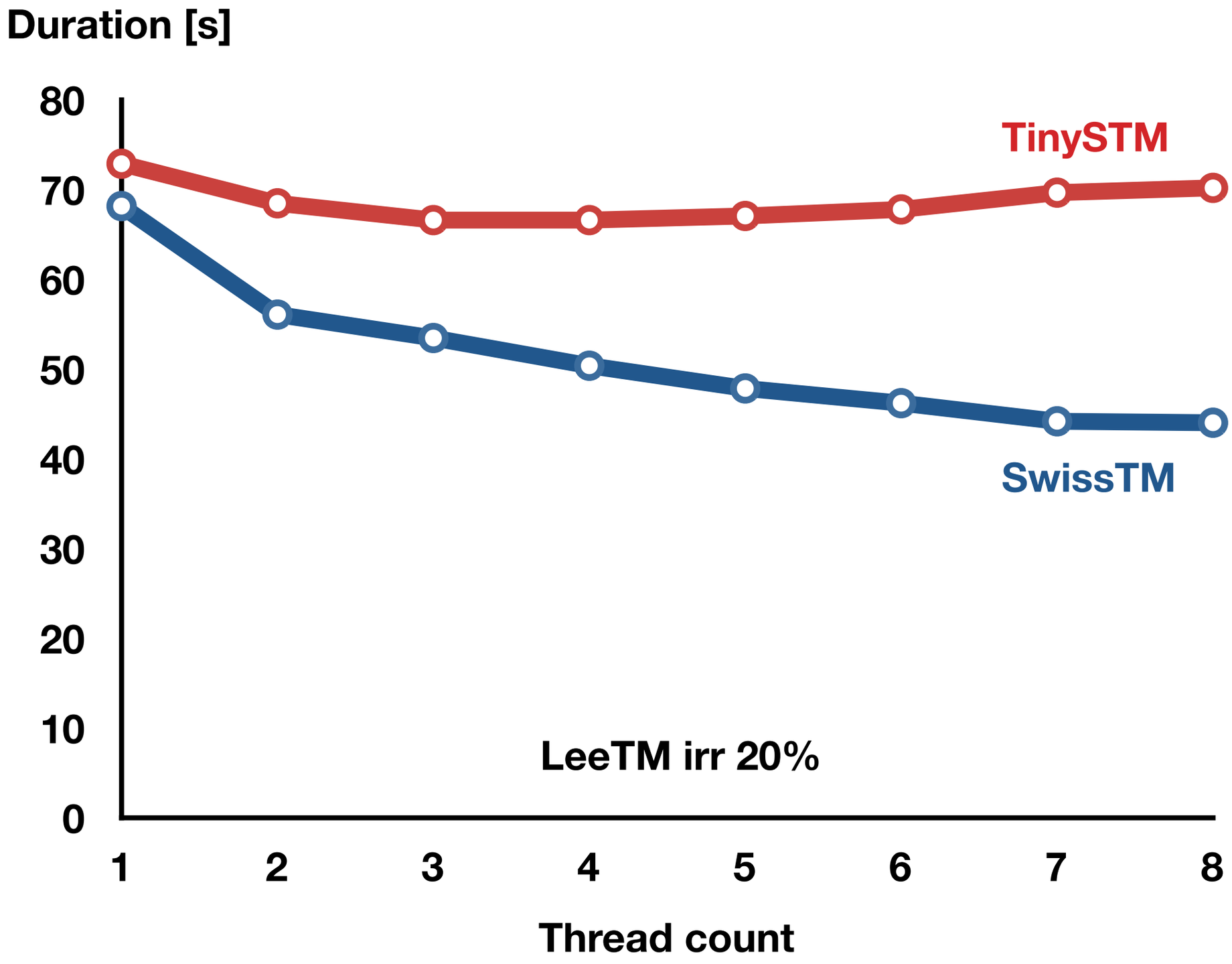
Mixed invalidation



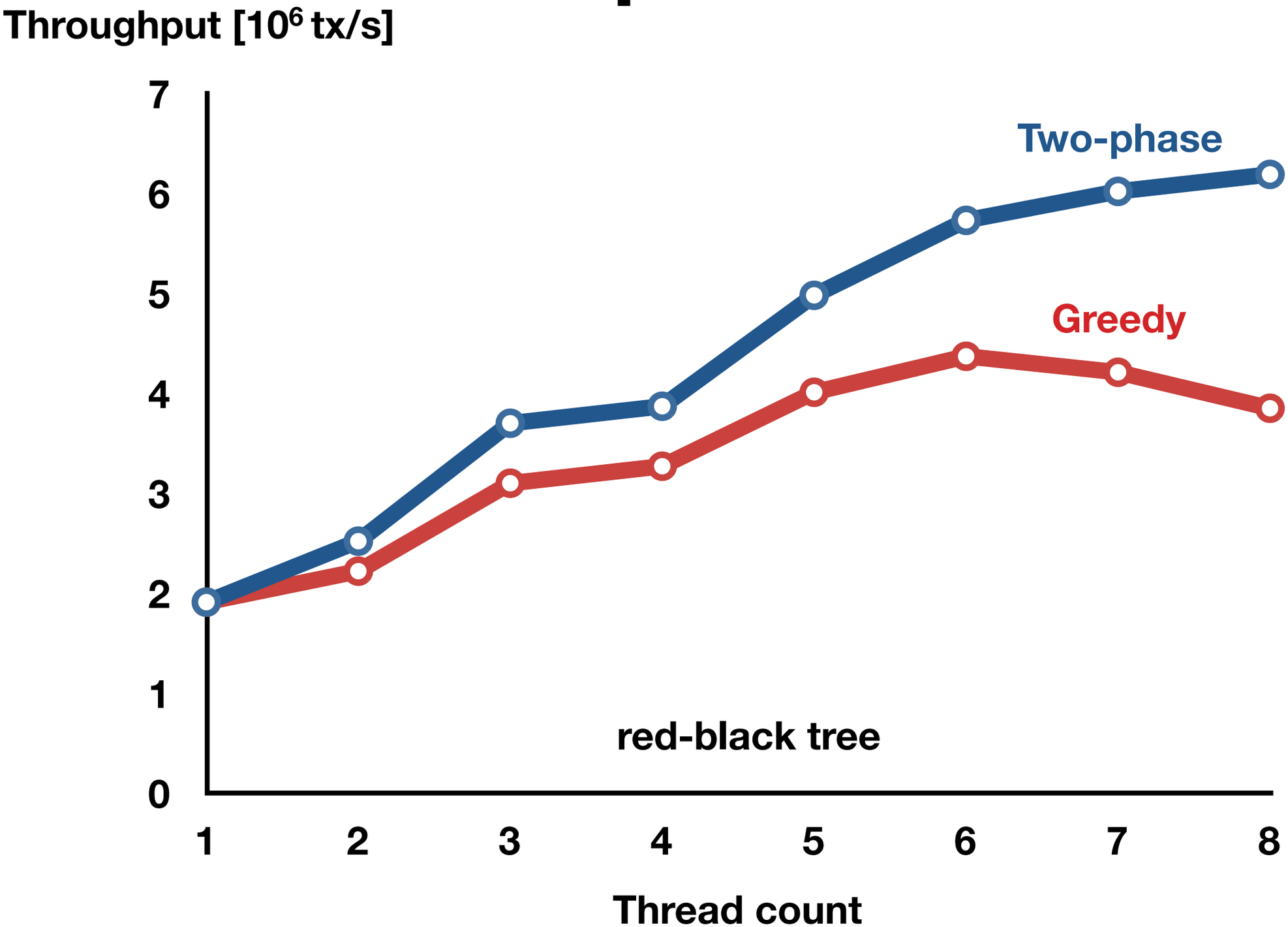
Mixed invalidation



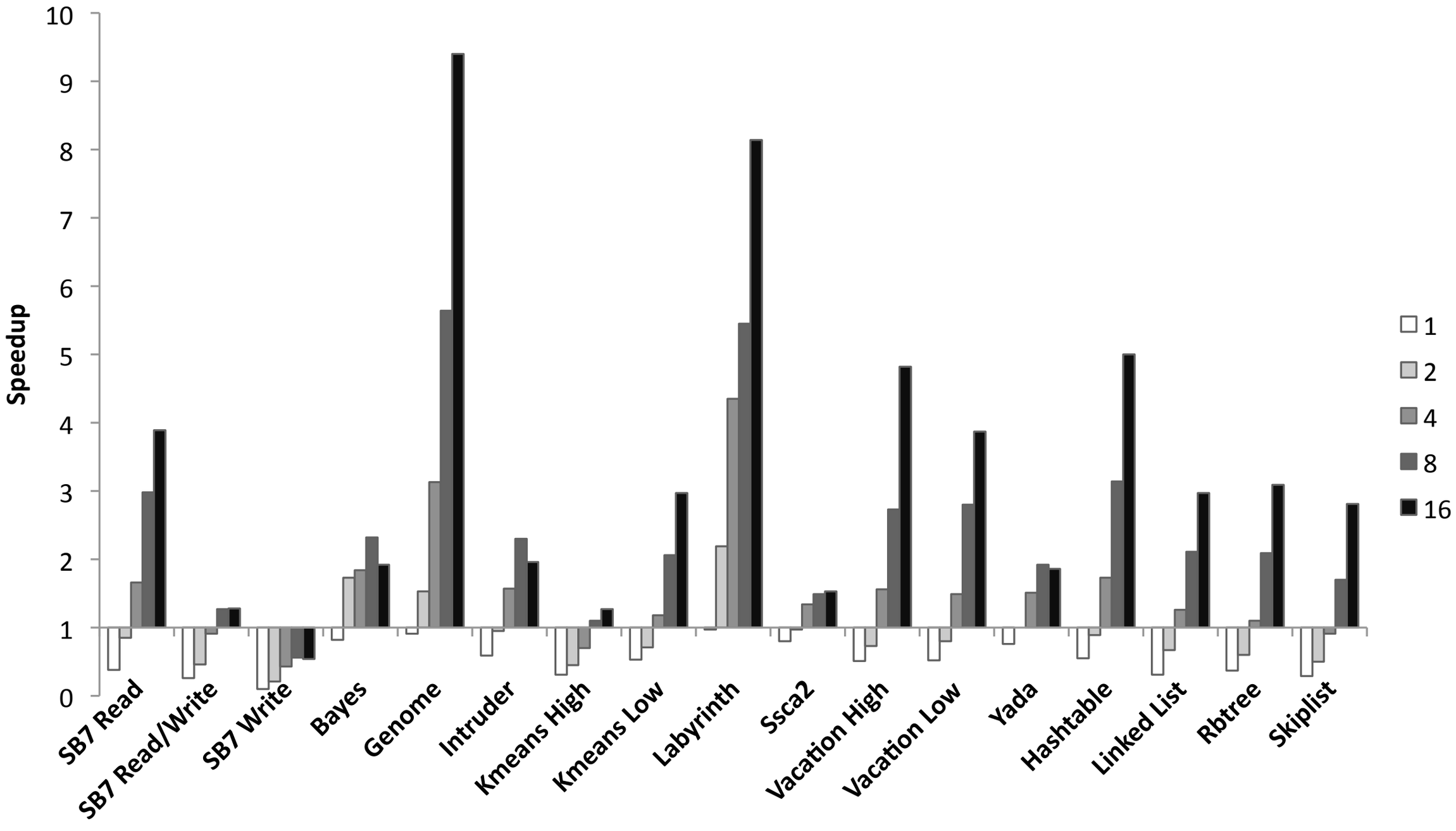
Mixed invalidation



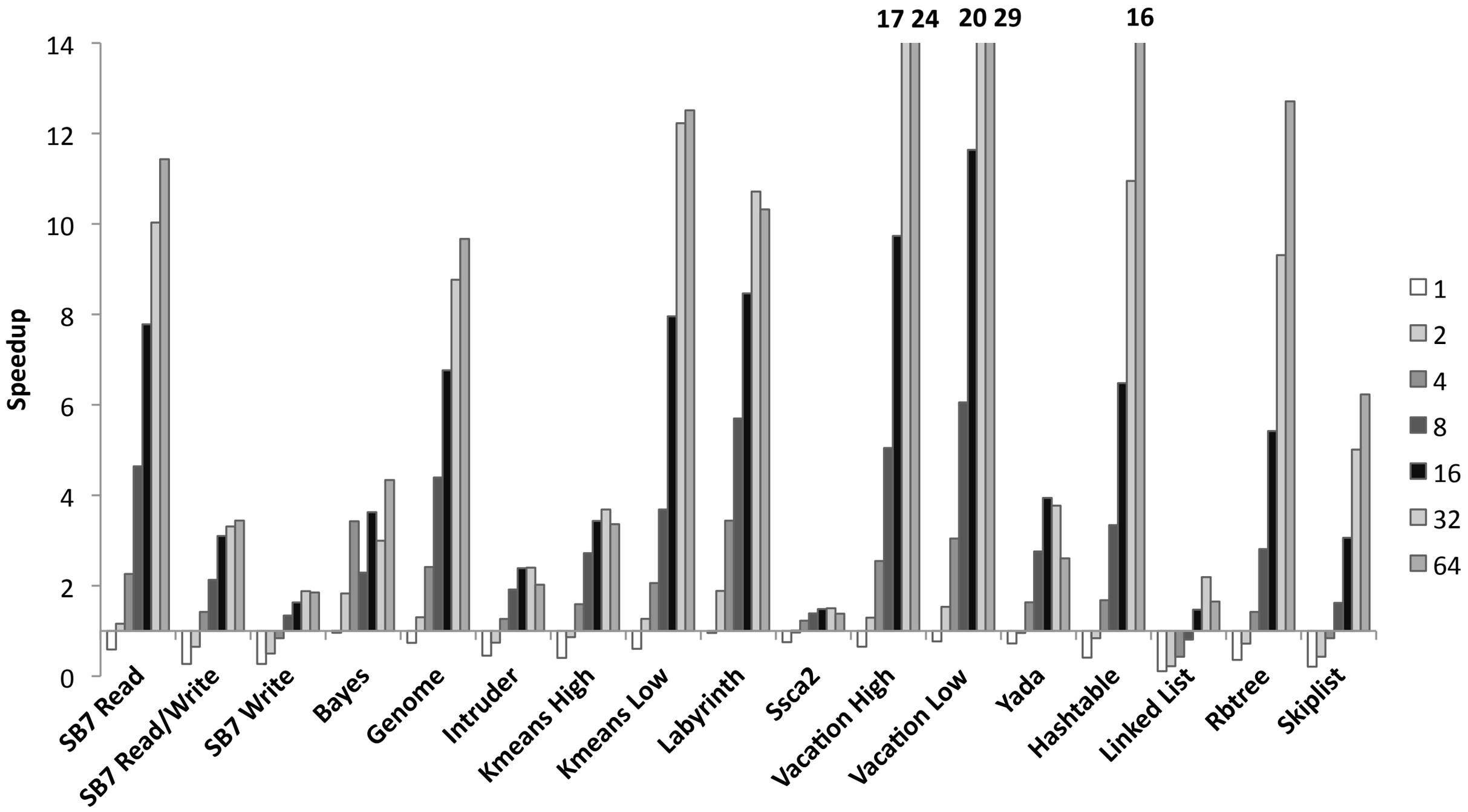
Two-phase CM


Throughput [10^6 tx/s]
7
6
5
4
3
2
1
0
Two-phase
Greedy
red-black tree
1
2
3
4
5
6
7
8
Thread count

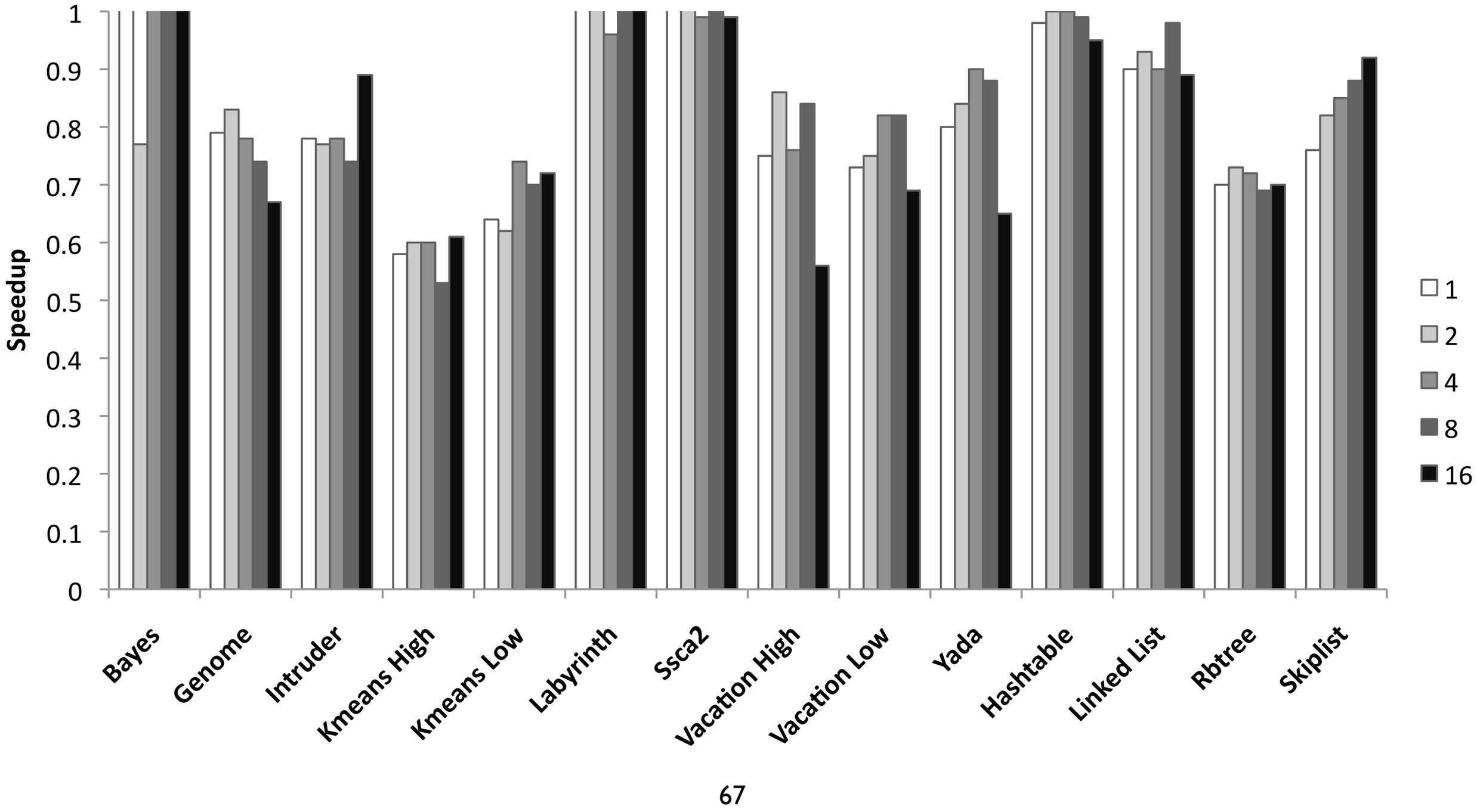
SwissTM vs Sequential



SwissTM vs Sequential (2)



Compiler cost



Links

- SwissTM
  - http://lpd.epfl.ch/site/research/tmeval
- Intel C/C++
  - http://software.intel.com/en-us/whatif/
- DTMC C/C++ (LLVM)
  - http://www.velox-project.eu/software/dtmc