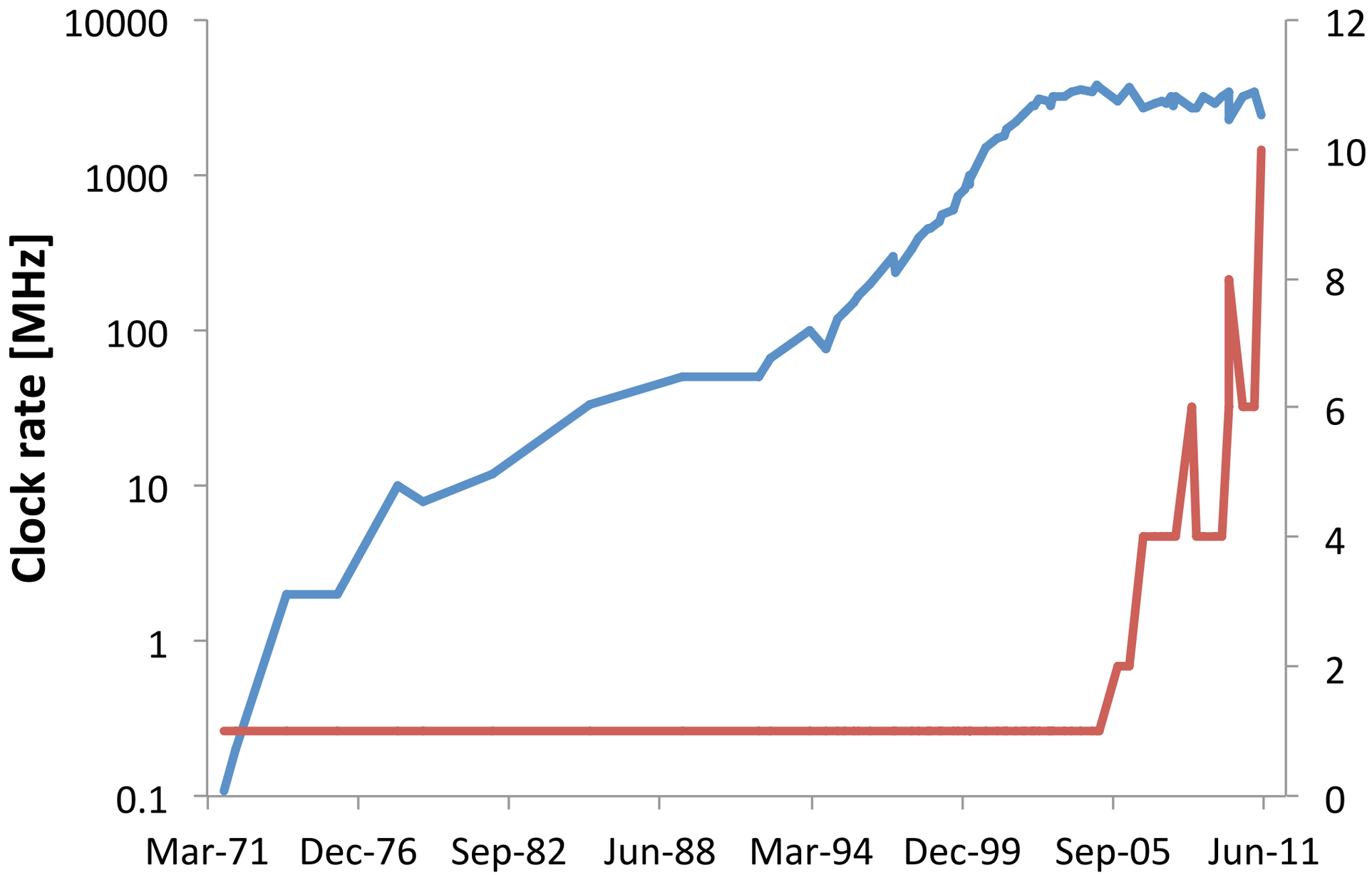
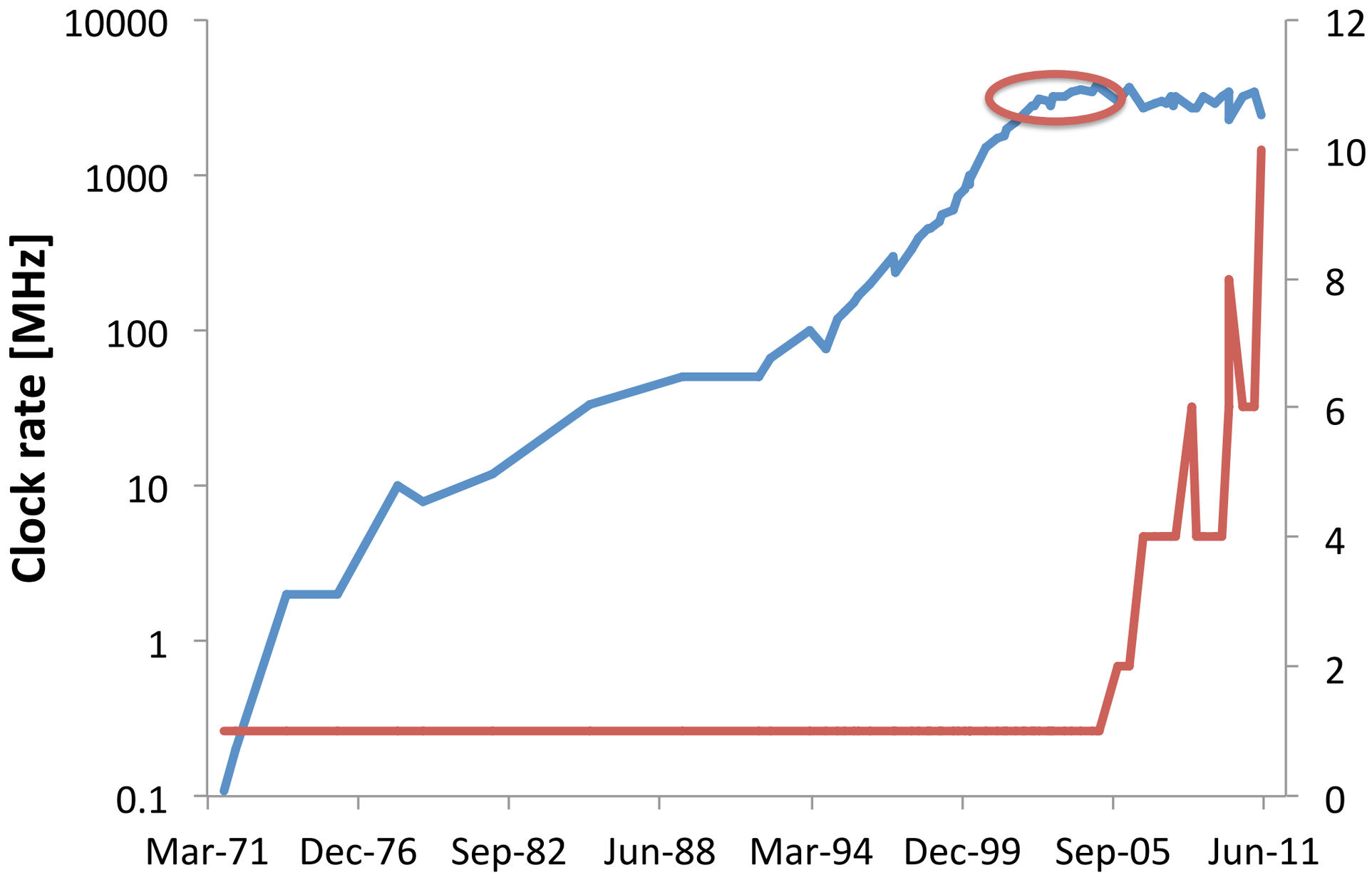
Transactional Memory Under the Hood

Aleksandar Dragojević

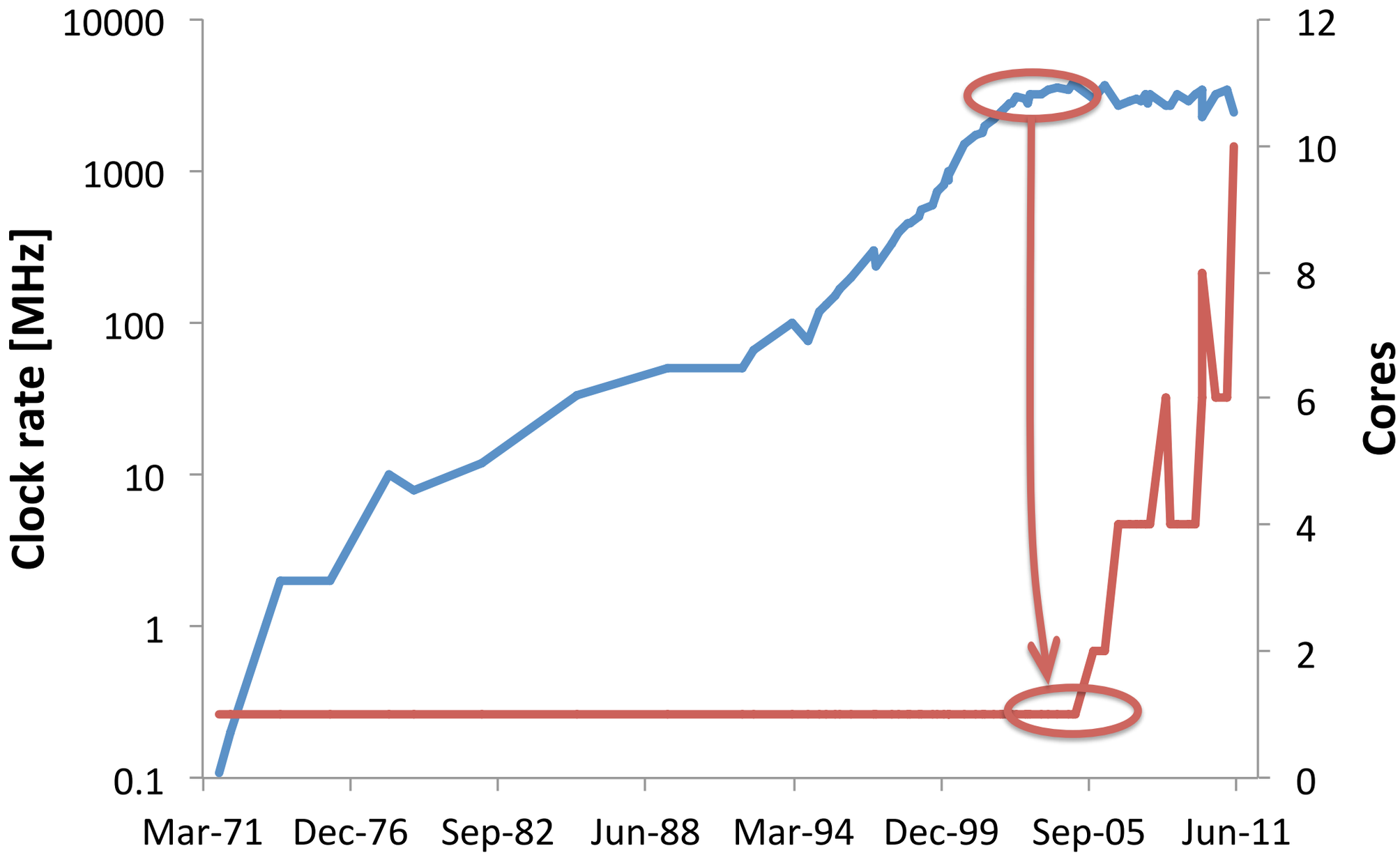
Hardware Trends



Hardware Trends



Hardware Trends



Concurrent Programming

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

Transactional Memory

Transactional Memory

```
// O1: move 20 a->b
1: int a = acc_a;
2: acc_a = a - 20;
3: int b = acc_b;
4: acc_b = b + 20;
// O2: add 10 to a
5: int a = acc_a;
6: acc_a = a + 10;
```

Transactional Memory

```
// O1: move 20 a->b
atomic {
1: int a = acc_a;
2: acc_a = a - 20;
3: int b = acc_b;
4: acc_b = b + 20;
}
// O2: add 10 to a
atomic {
5: int a = acc_a;
6: acc_a = a + 10;
}
```

Transactional Memory

```
// O1: move 20 a->b
atomic {
1: int a = acc_a;
2: acc_a = a - 20;
3: int b = acc_b;
4: acc_b = b + 20;
}
```

```
// O2: add 10 to a
atomic {
5: int a = acc_a;
6: acc_a = a + 10;
                }
```

Transactional Memory

- Sequential code
- Add transactions
  - *atomic* key word
- System ensures correctness
  - equivalent to sequential

Is it really simpler?

Is it really simpler?

```
void Enqueue(int value) {
  ptrver_t next, tail, tail2, newb;
  node_t *node = alloc_queue_node(value);
  while(true) {
    tail = Tail;
    next = tail.ptr->next;
    tail2 = Tail;
    if(tail == tail2)
      if(next.ptr == NULL) {
        newb.ptr = node;
        newb.ver = next.ver + 1;
        if(CAS(&tail.ptr->next, next, newb))
          break;
      } else {
        newb.ptr = next.ptr;
        newb.ver = tail.ver + 1;
        CAS(&Tail, tail, newb);
      }
  }
  newb.ptr = node;
  newb.ver = tail.ver + 1;
  CAS(&Tail, tail, newb);
}
```

Is it really simpler?

```
void Enqueue(int value) {
  node_t *node = alloc_queue_node(value);
  atomic {
    if(tail == NULL)
      head = node;
    else
      tail->next = node;
    tail = node;
  }
}
```

Disclaimer

What I might say

- TM is easy to use by non-experts
- TM shows great promise
- I hope we can make TM widely used

What I am not saying

Outline

- What is TM?
- How to implement an STM?
  - SwissTM
- STM performance
  - predicting the performance

What is TM?

What is TM?

What does TM guarantee?

Serializability

Serializability

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

Serializability

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

Serializability

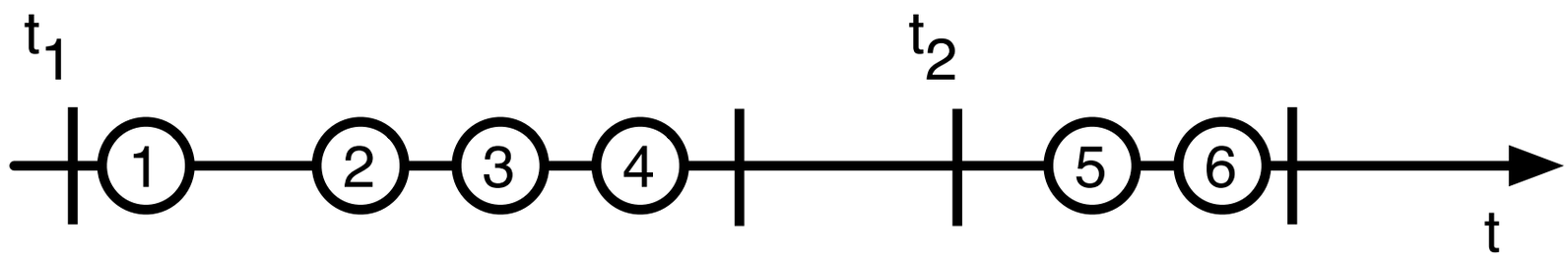
```
atomic { // t1
1: int a = acc_a;
2: acc_a = a - 20;
3: int b = acc_b;
4: acc_b = b + 20;
}
```

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

Serializability

```
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;
}
```

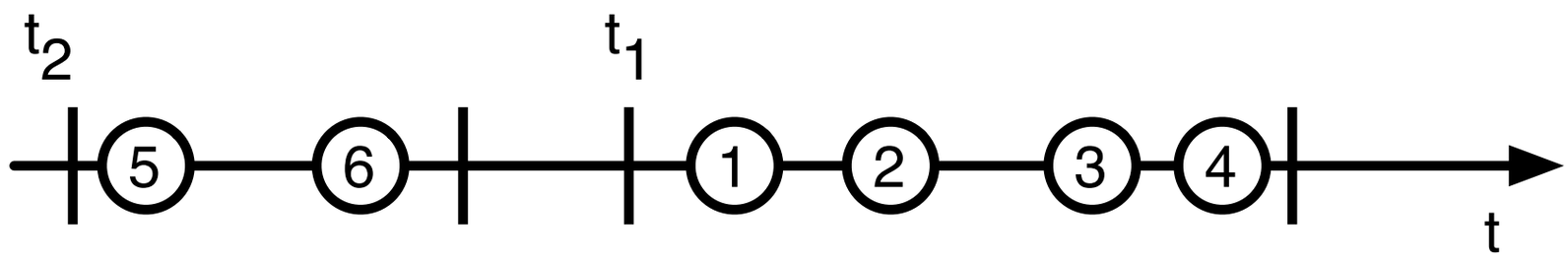


correct

Serializability

```
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;
}
```

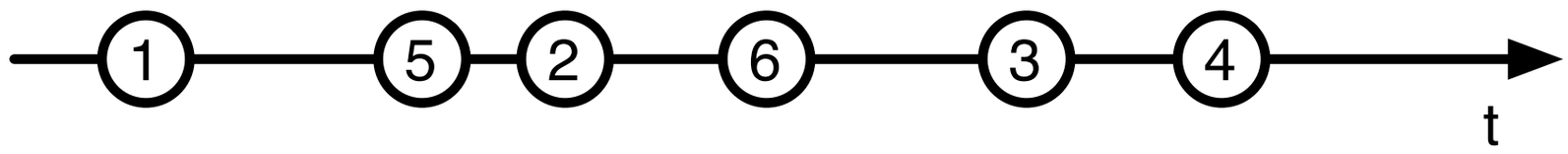


correct

Serializability

```
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;
}
```

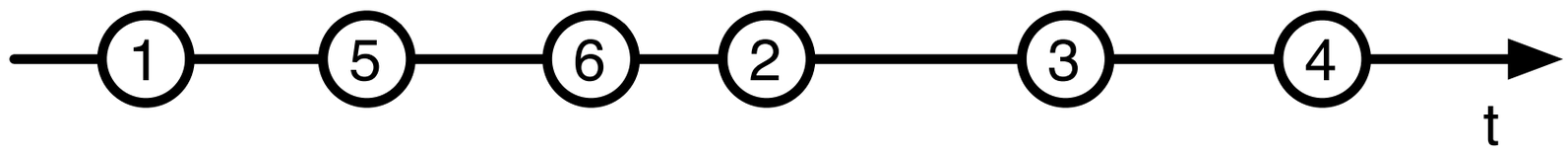


client ☺

Serializability

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

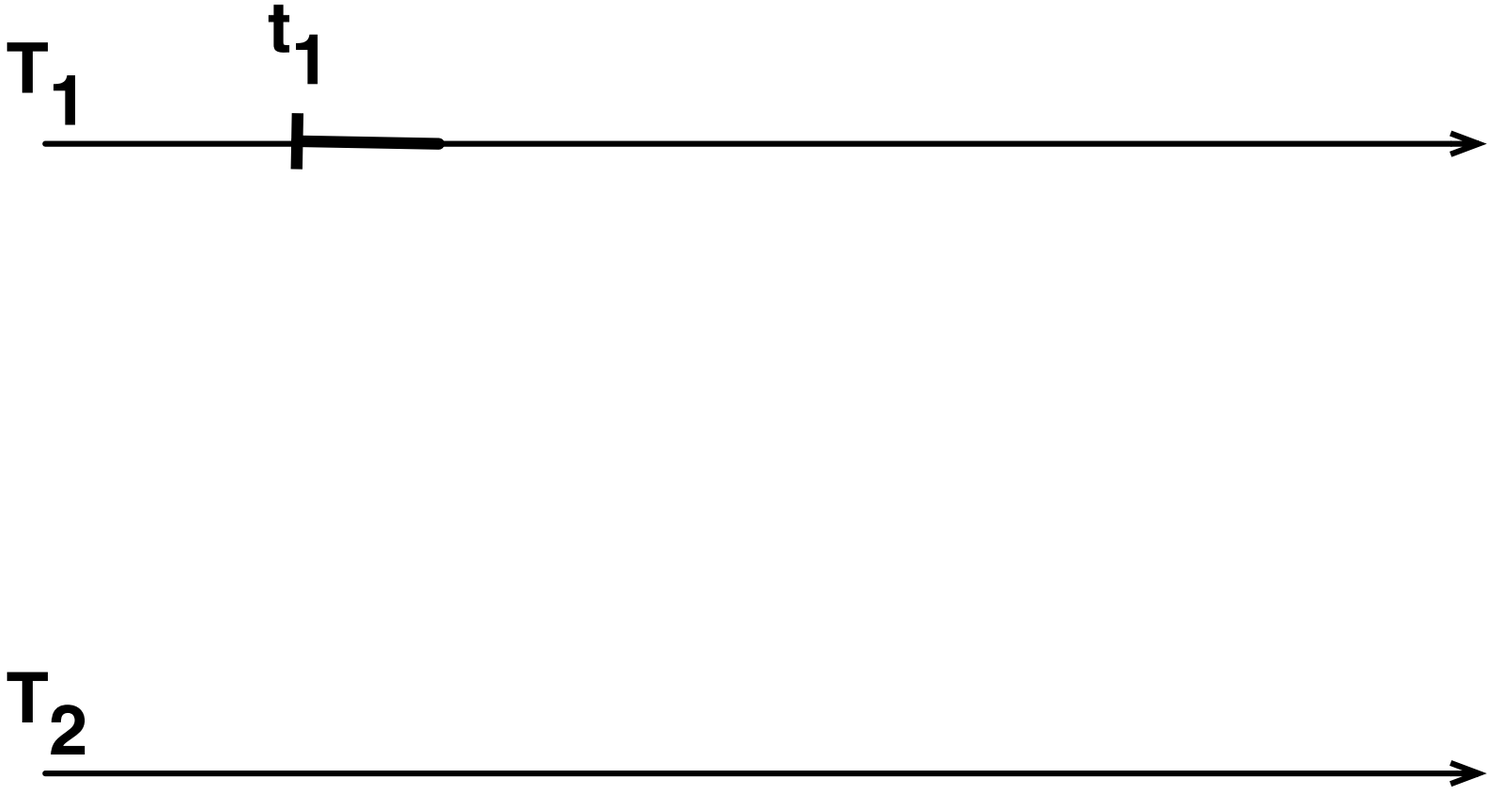
```
atomic { // t2
5: int a = acc_a;
6: acc_a = a + 10;
}
```



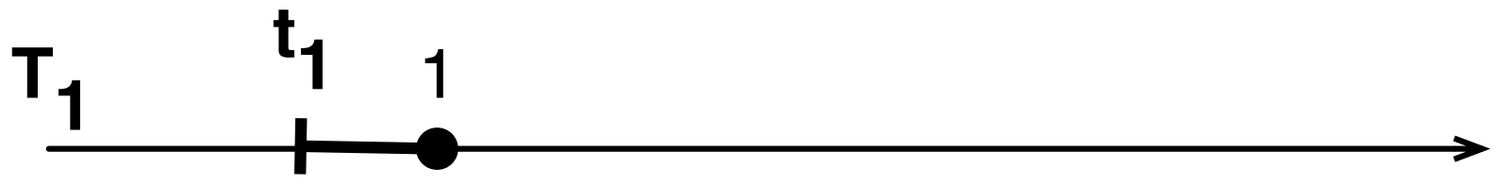
bank ☺

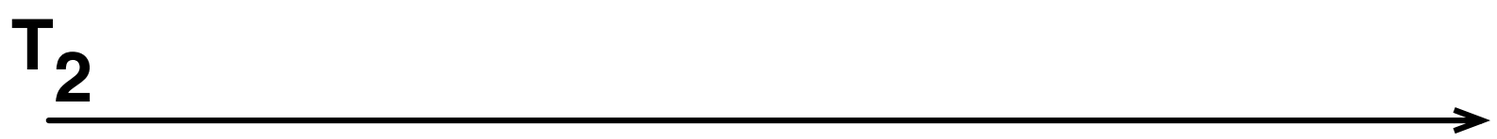
How is this achieved?

How is this achieved?

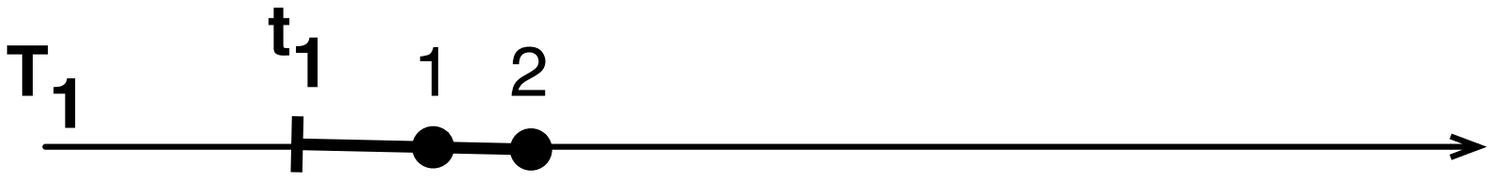


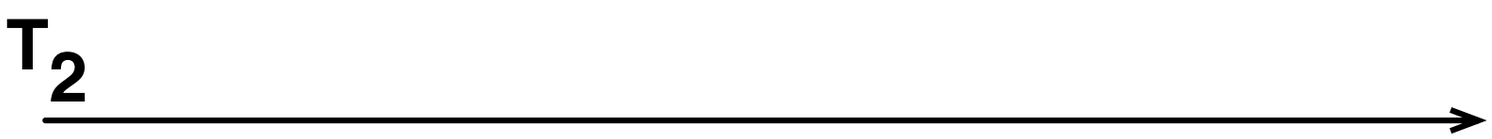
How is this achieved?



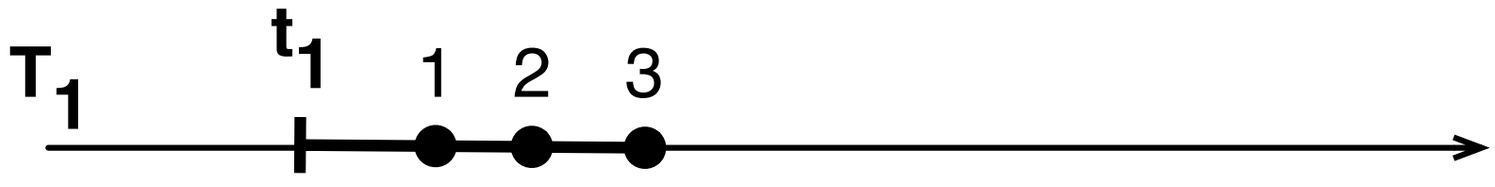


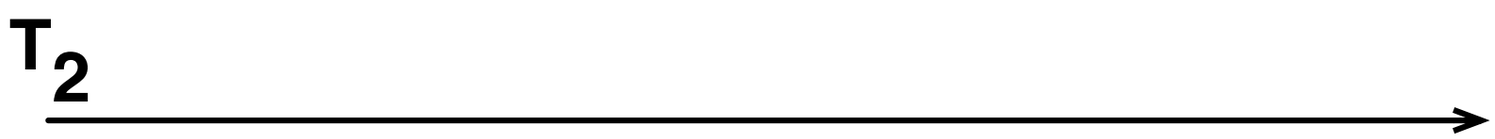
How is this achieved?



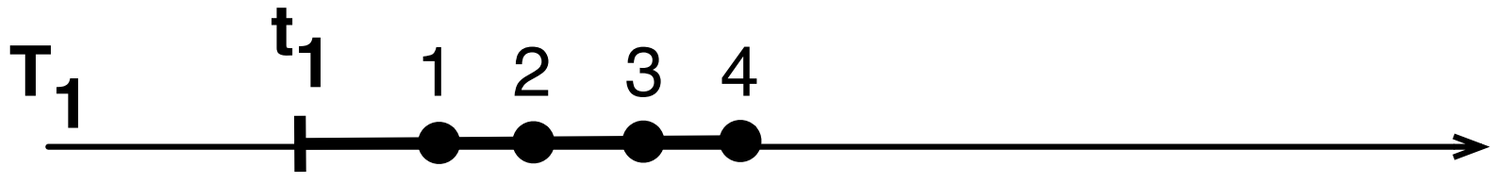


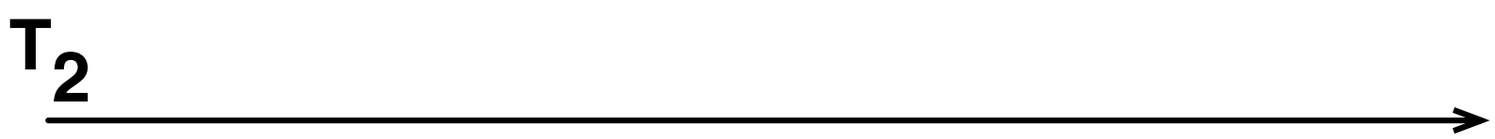
How is this achieved?



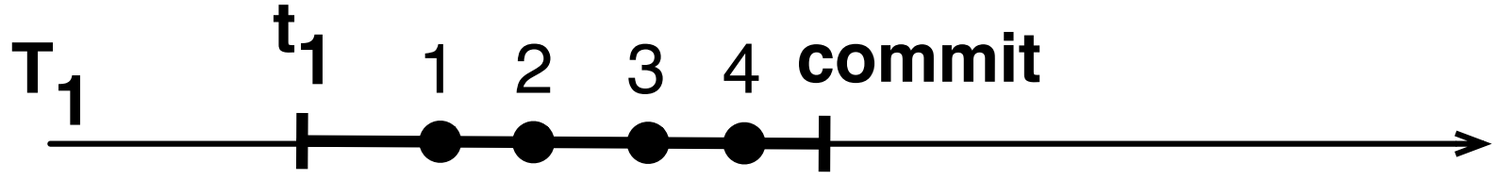


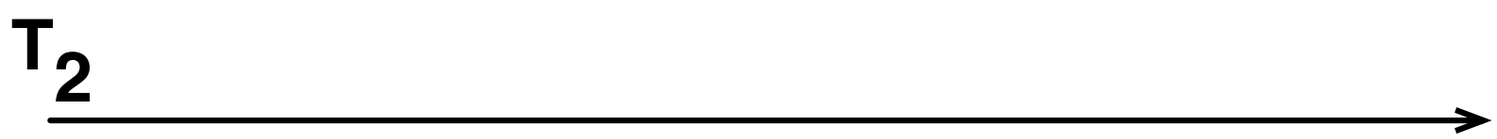
How is this achieved?



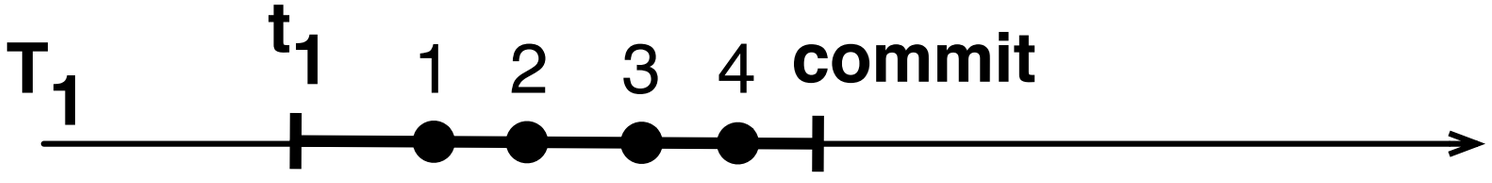


How is this achieved?

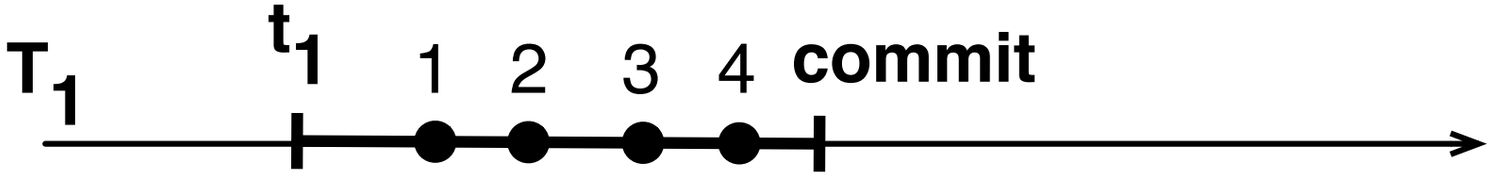




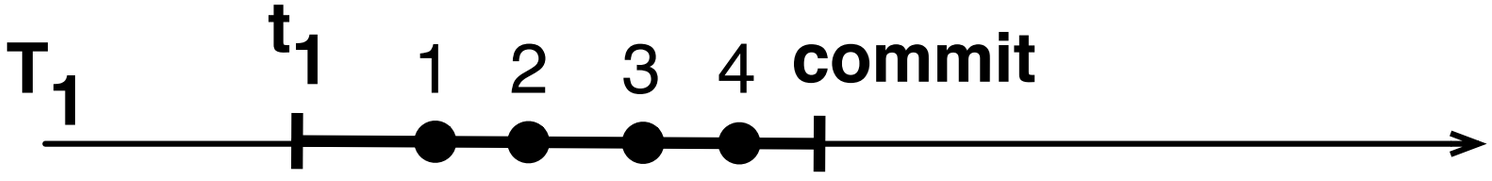
How is this achieved?



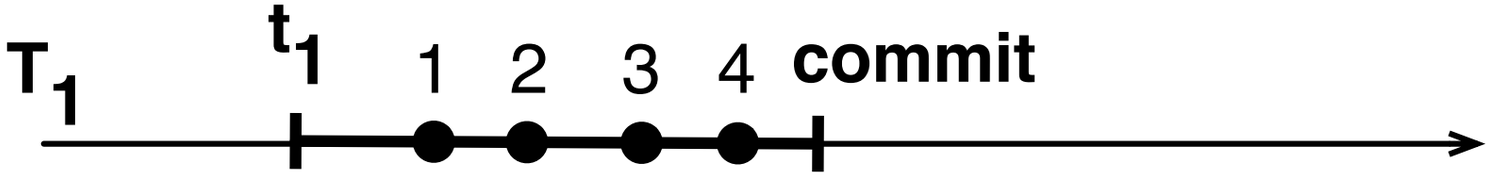
How is this achieved?



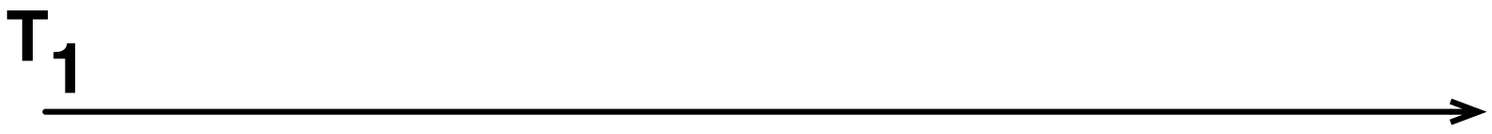
How is this achieved?



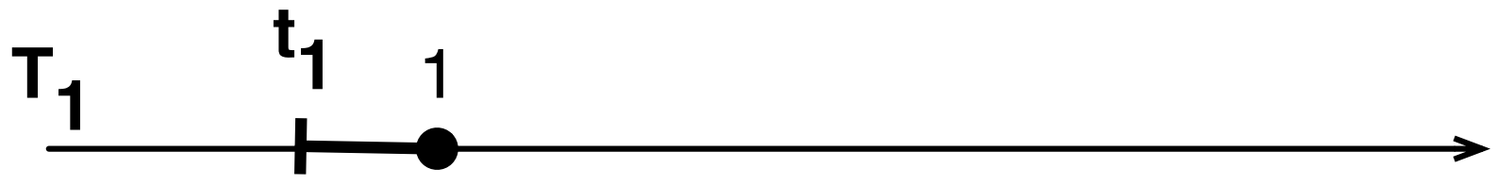
How is this achieved?

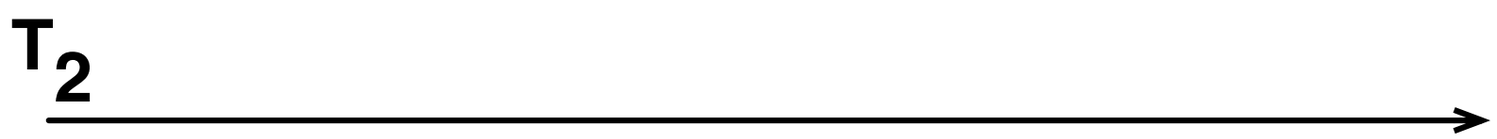


How is this achieved?

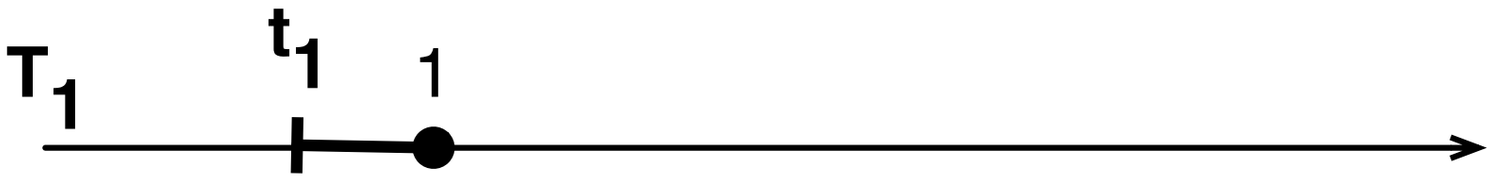


How is this achieved?

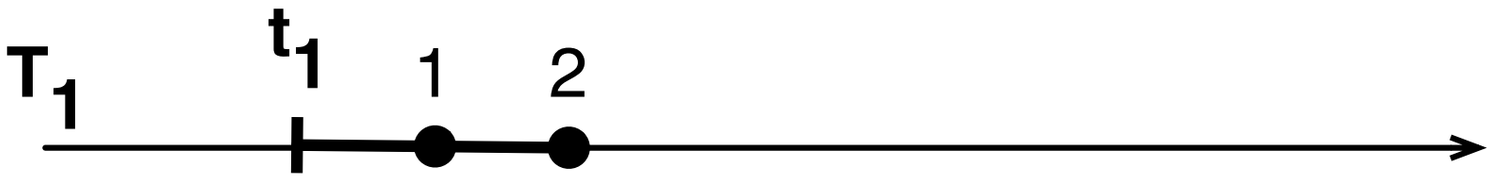


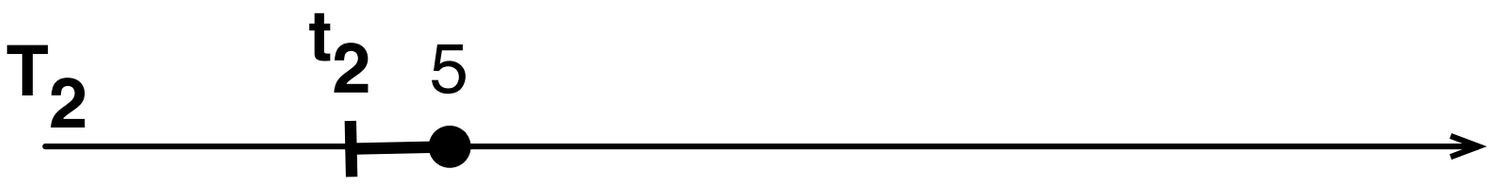


How is this achieved?

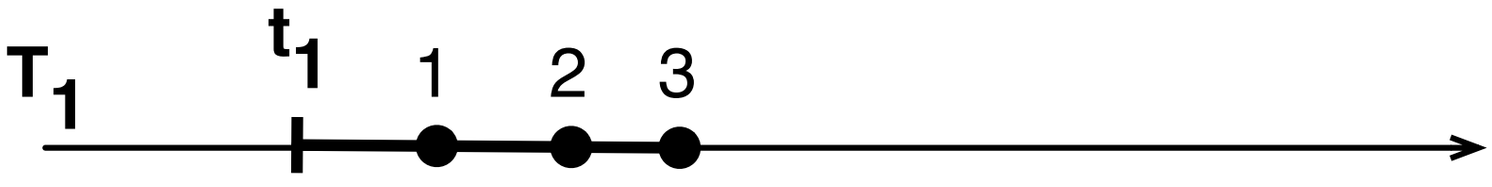


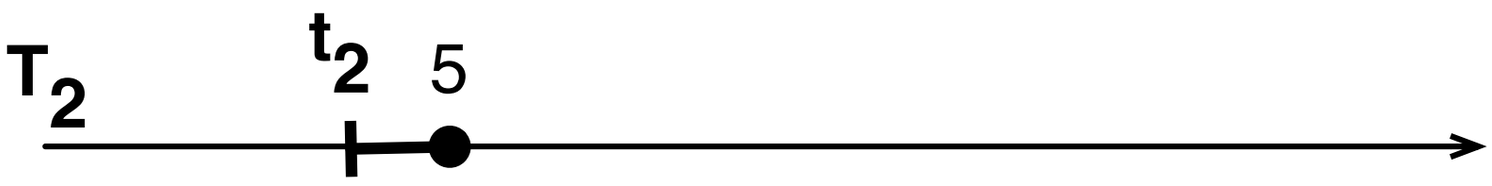
How is this achieved?



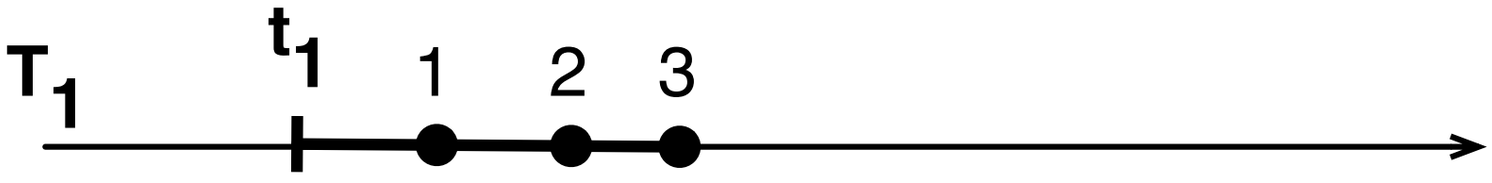


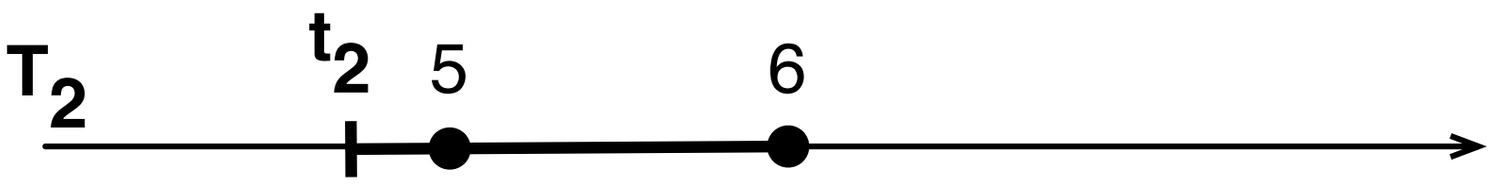
How is this achieved?



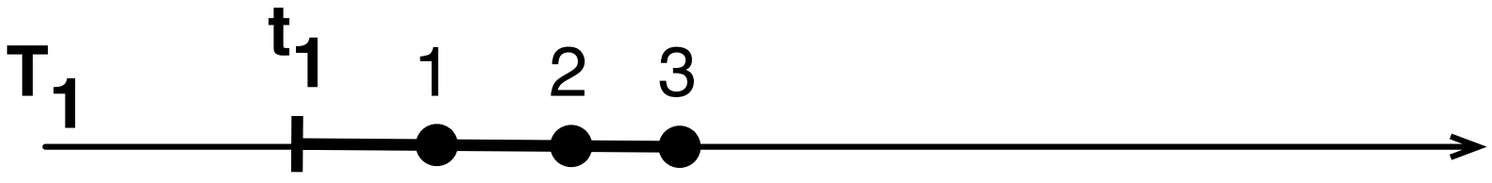


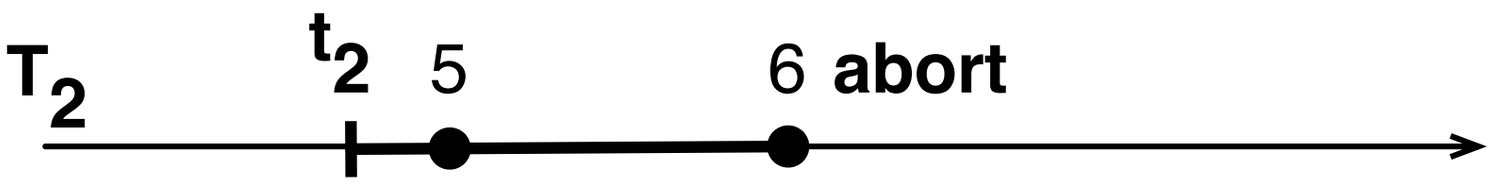
How is this achieved?



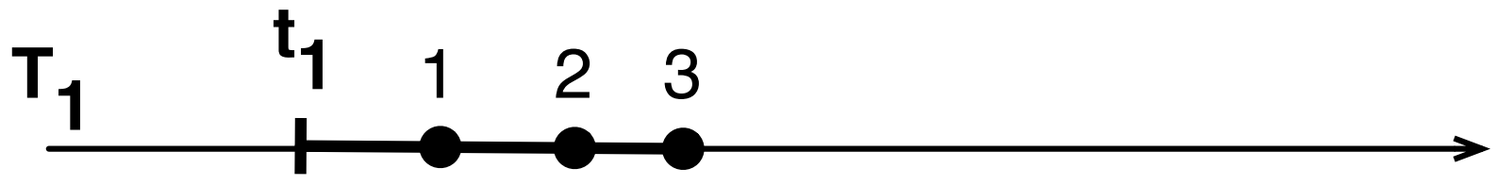


How is this achieved?

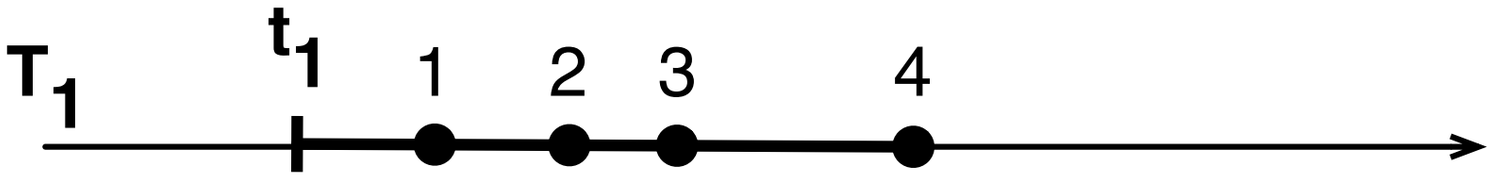




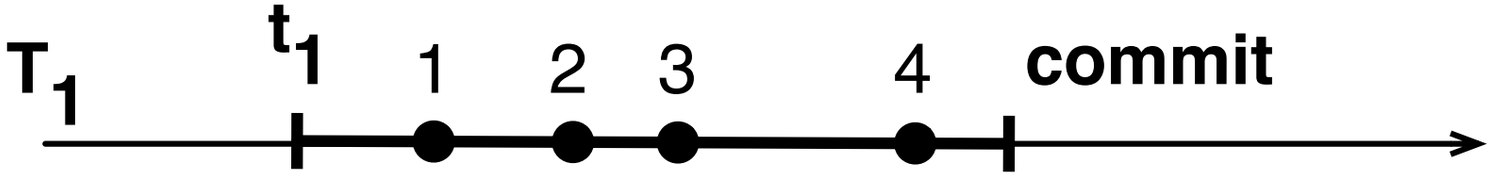
How is this achieved?



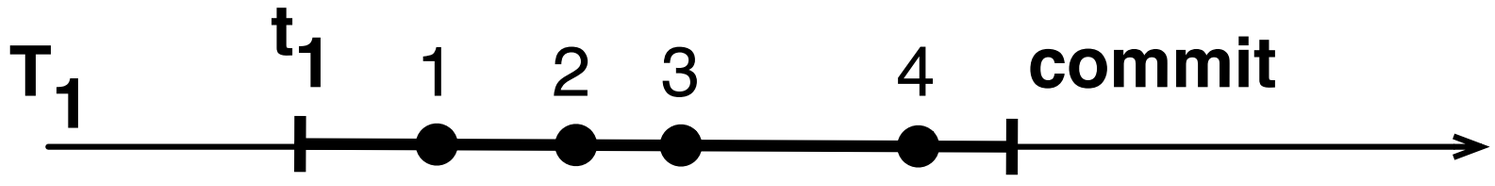
How is this achieved?



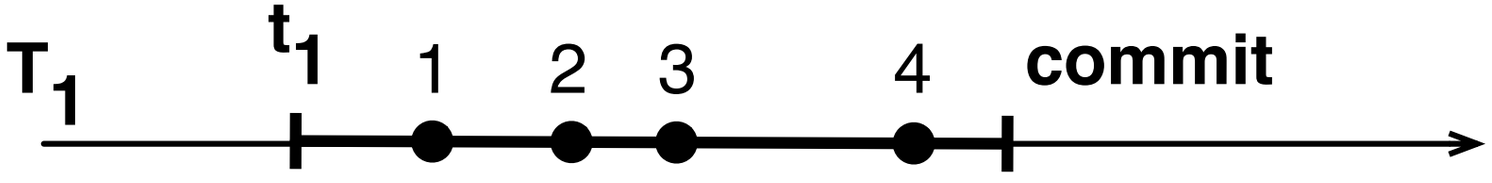
How is this achieved?



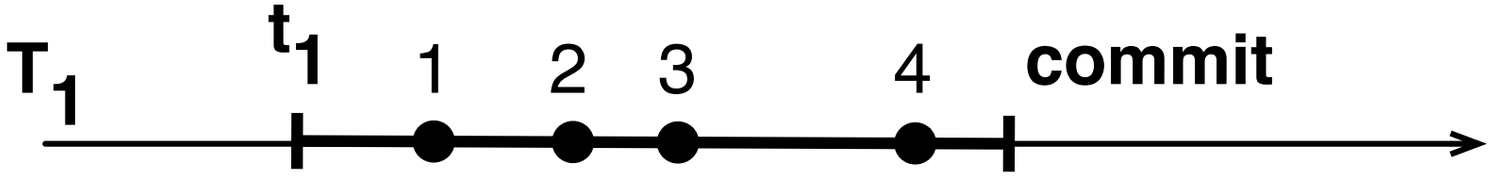
How is this achieved?



How is this achieved?



How is this achieved?



How is this achieved?

- TM monitors accesses to objects
- When it detects *conflicting* access
  - one transaction is *aborted*
  - it is *restarted*
- When all actions are not conflicting
  - transaction *commits*

Is serializability enough?

Is serializability enough?

```
Variables:
    int x=0, y=1;
Invariant:
    x < y
```

Is serializability enough?

```
Variables:
    int x=0, y=1;
Invariant:
    x < y

atomic {
1: int xl = x;
2: int yl = y;
3: x = yl;
4: y = yl * 2;
}
```

Is serializability enough?

```
Variables:
    int x=0, y=1;
Invariant:
    x < y
```

```
atomic {
1: int xl = x;
2: int yl = y;
3: x = yl;
4: y = yl * 2;
}
```

```
atomic {
5: int xl = x;
6: int yl = y;
7: int zl = 1/(yl-xl);
8: z = zl;
}
```

Consistent view

- All transactions must observe consistent views of memory at all times
- even the aborted ones

Opacity

- 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

TM semantics

- Committed: instantaneous
- Aborted: never visible
- All: observe consistent state

How to implement an STM?

How to implement an STM?

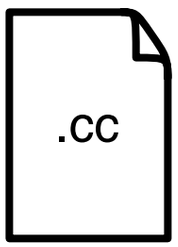
How does it all fit?

Software TM

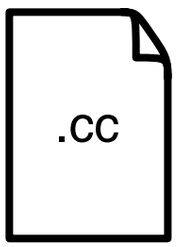
- Available now
- Component of HyTM
- Backwards compatibility

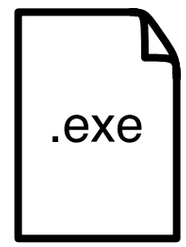
From .cc To .exe

From .cc To .exe

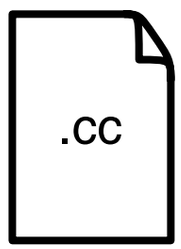


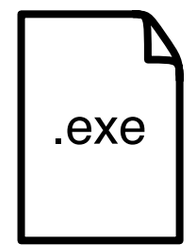
From .cc To .exe





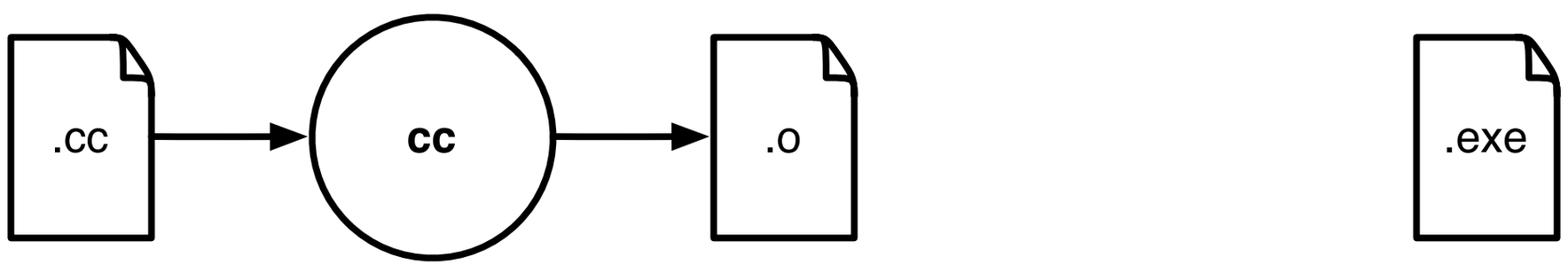
From .cc To .exe



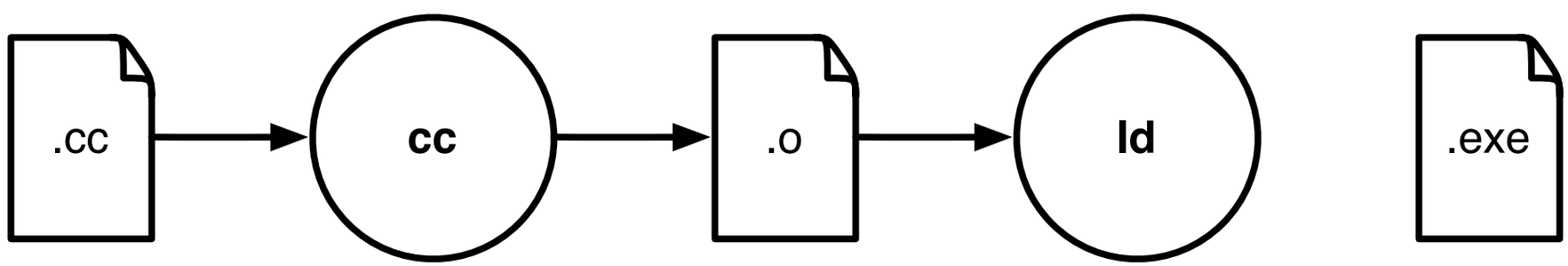


From .cc To .exe

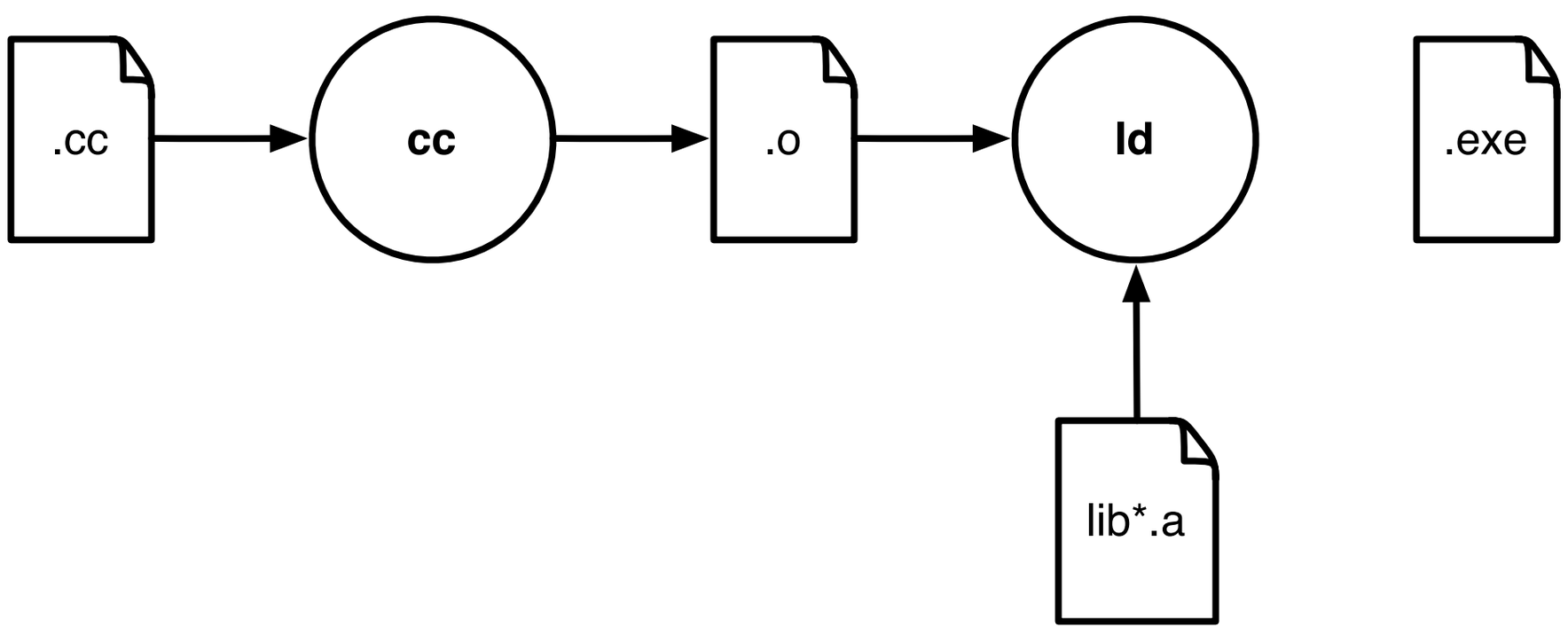
From .cc To .exe



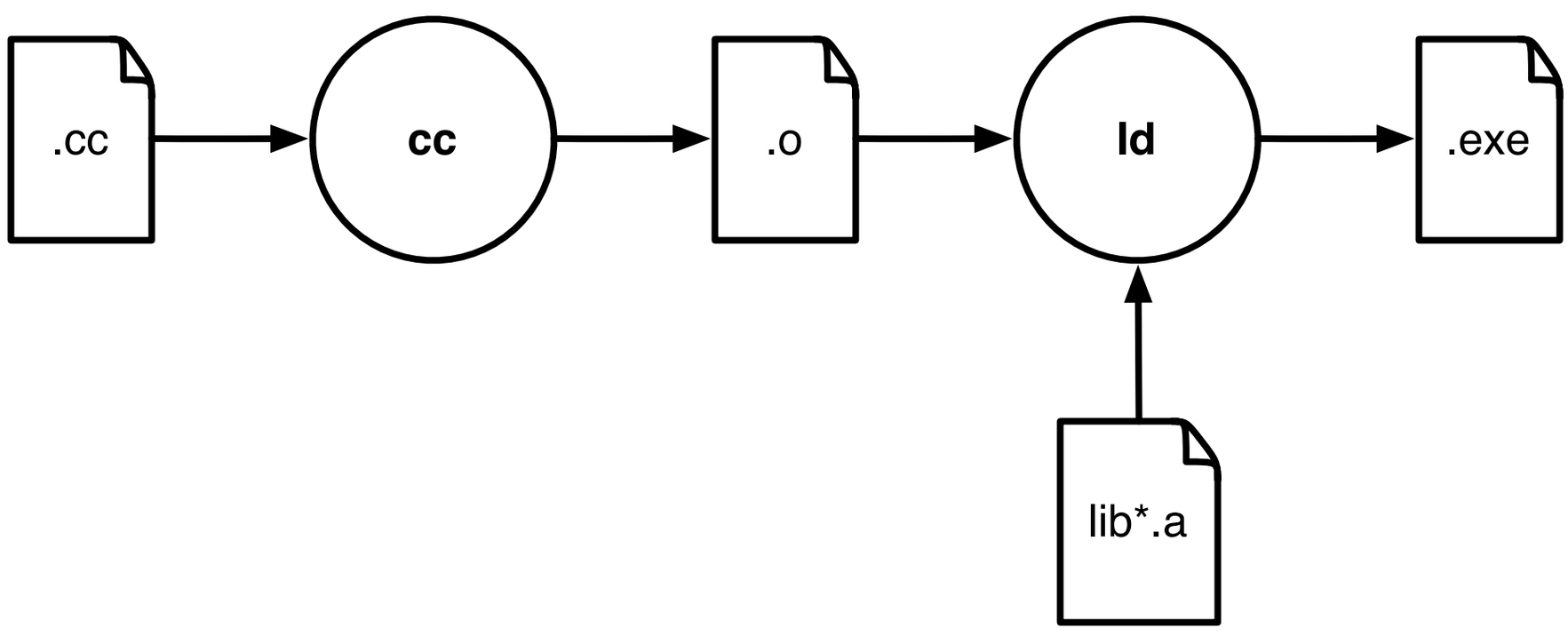
From .cc To .exe



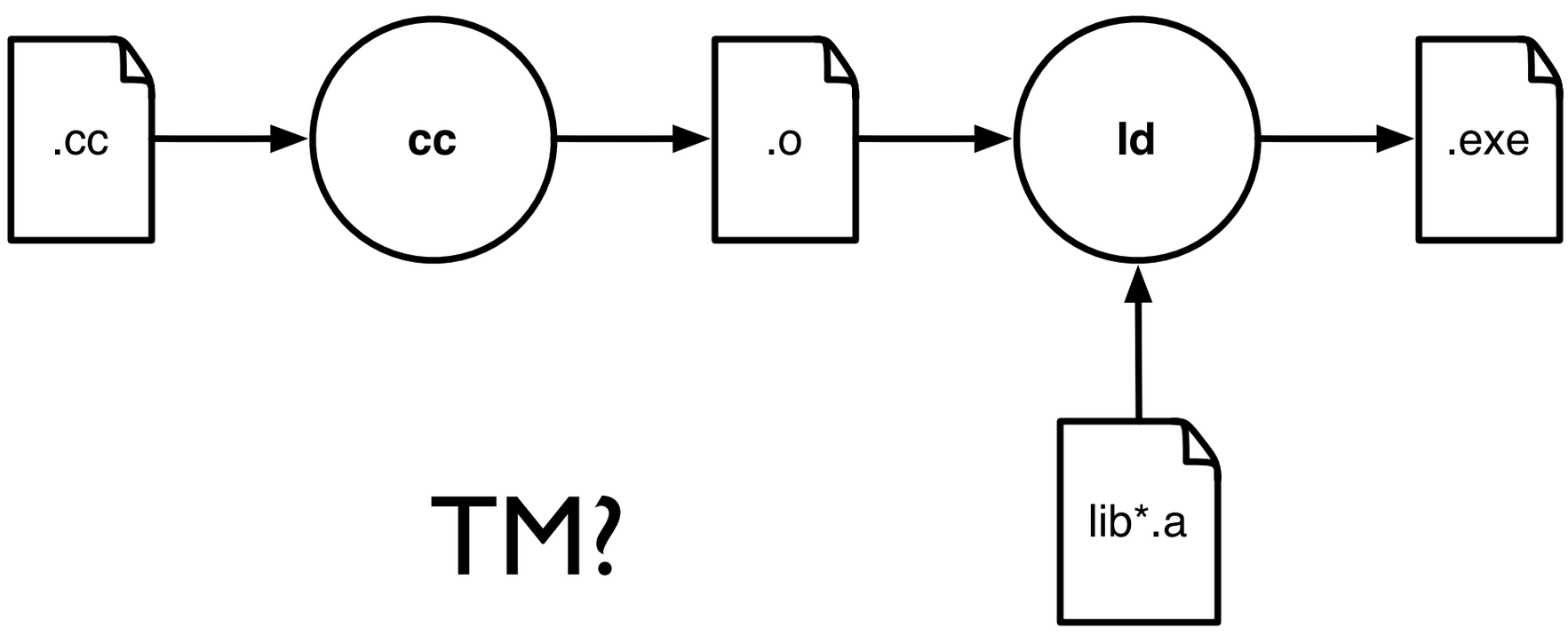
From .cc To .exe



From .cc To .exe



From .cc To .exe

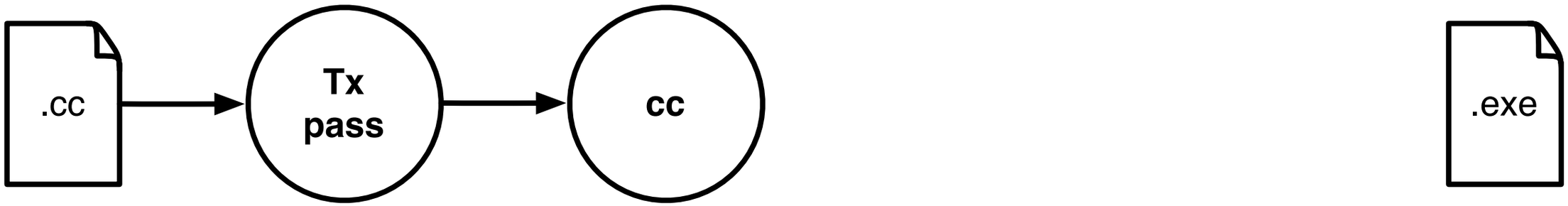


TM?

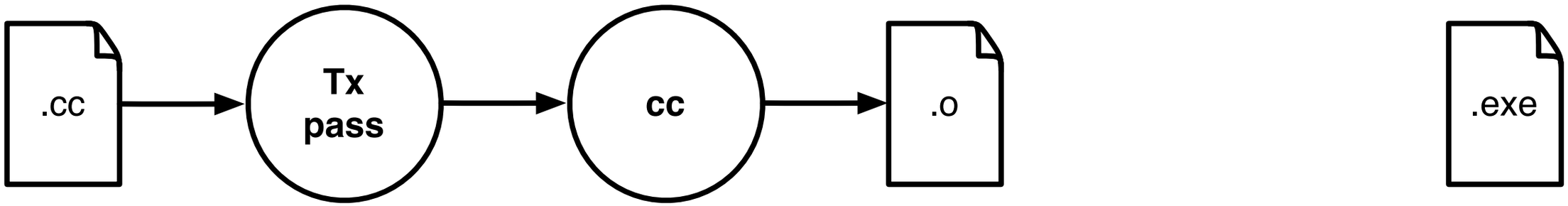
From .cc To .exe

From .cc To .exe

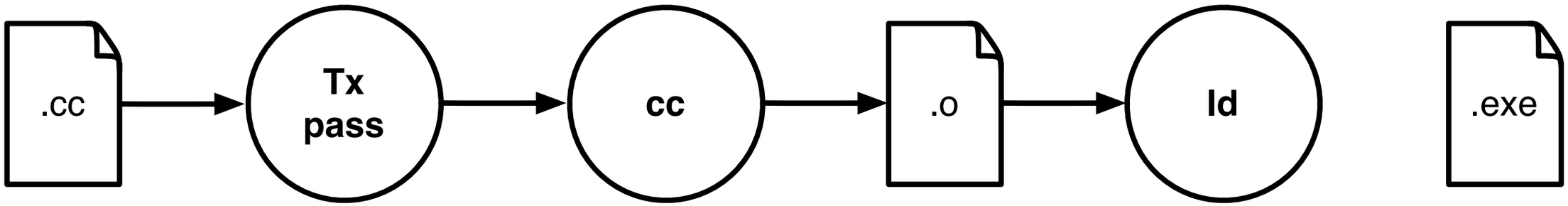
From .cc To .exe



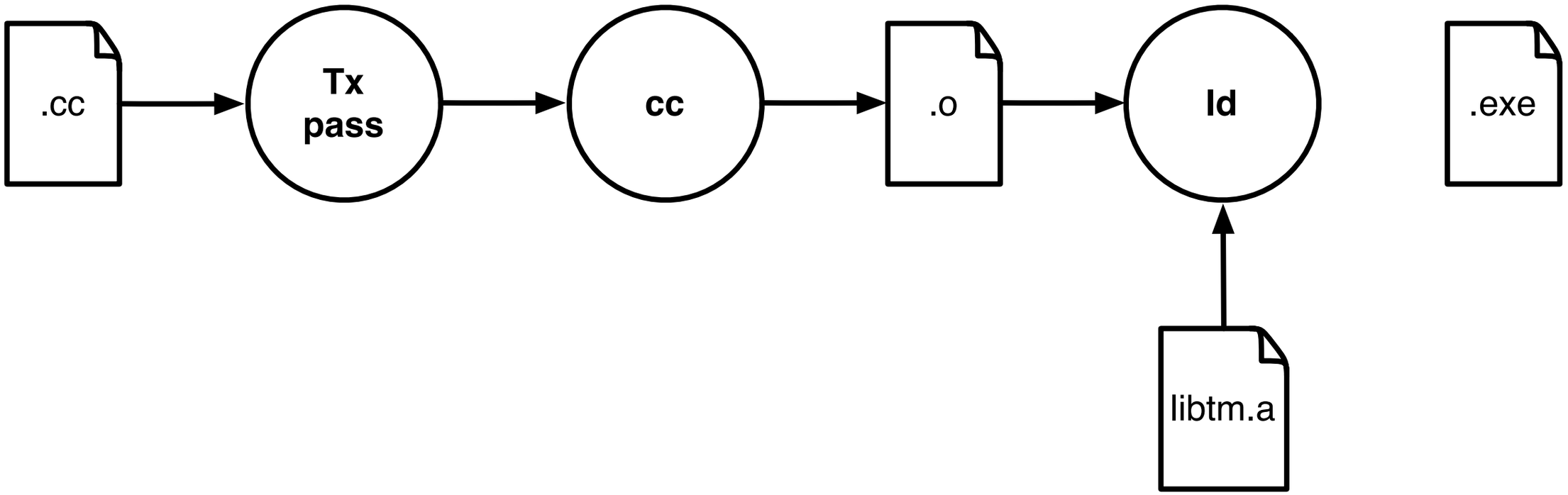
From .cc To .exe



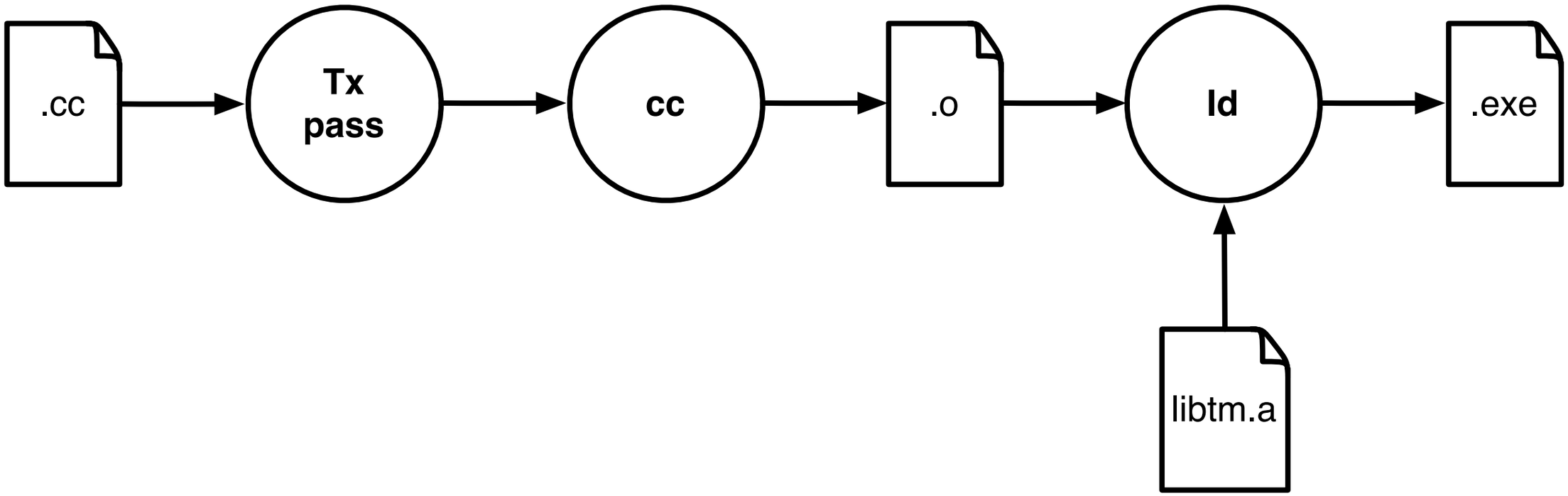
From .cc To .exe



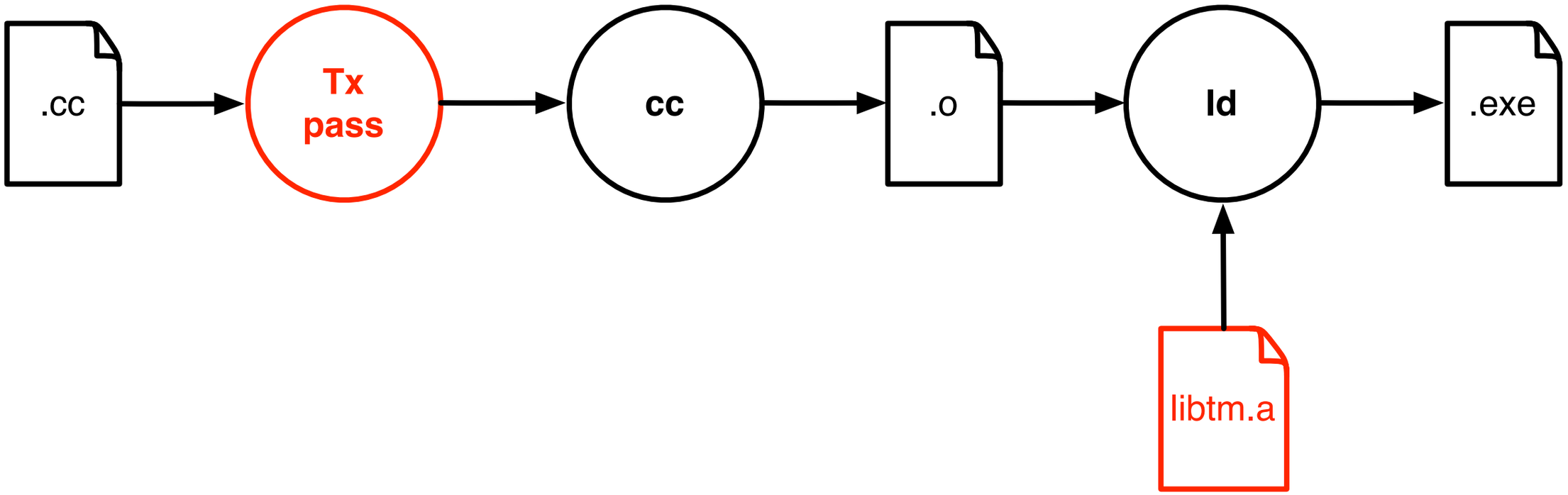
From .cc To .exe



From .cc To .exe



From .cc To .exe



TM library

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

TM library API

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

TM libraries

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

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

Implementing Tx pass

- Manual
- Compiler
- Other

Manual Tx pass

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

Compiler Tx pass

- 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#)

Tx Passes

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

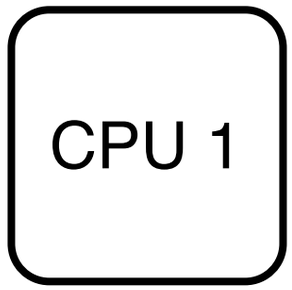
SwissTM

SwissTM

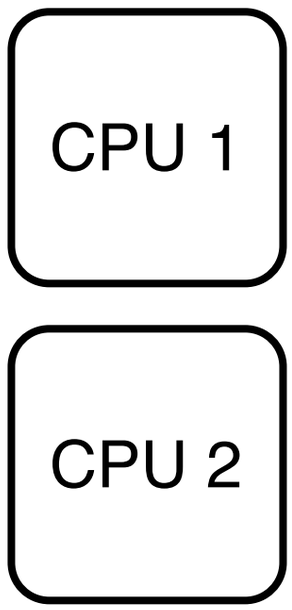
How to implement an STM library?

Setting

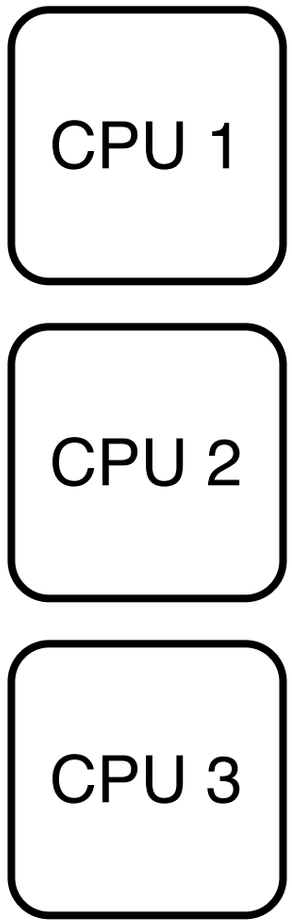
Setting



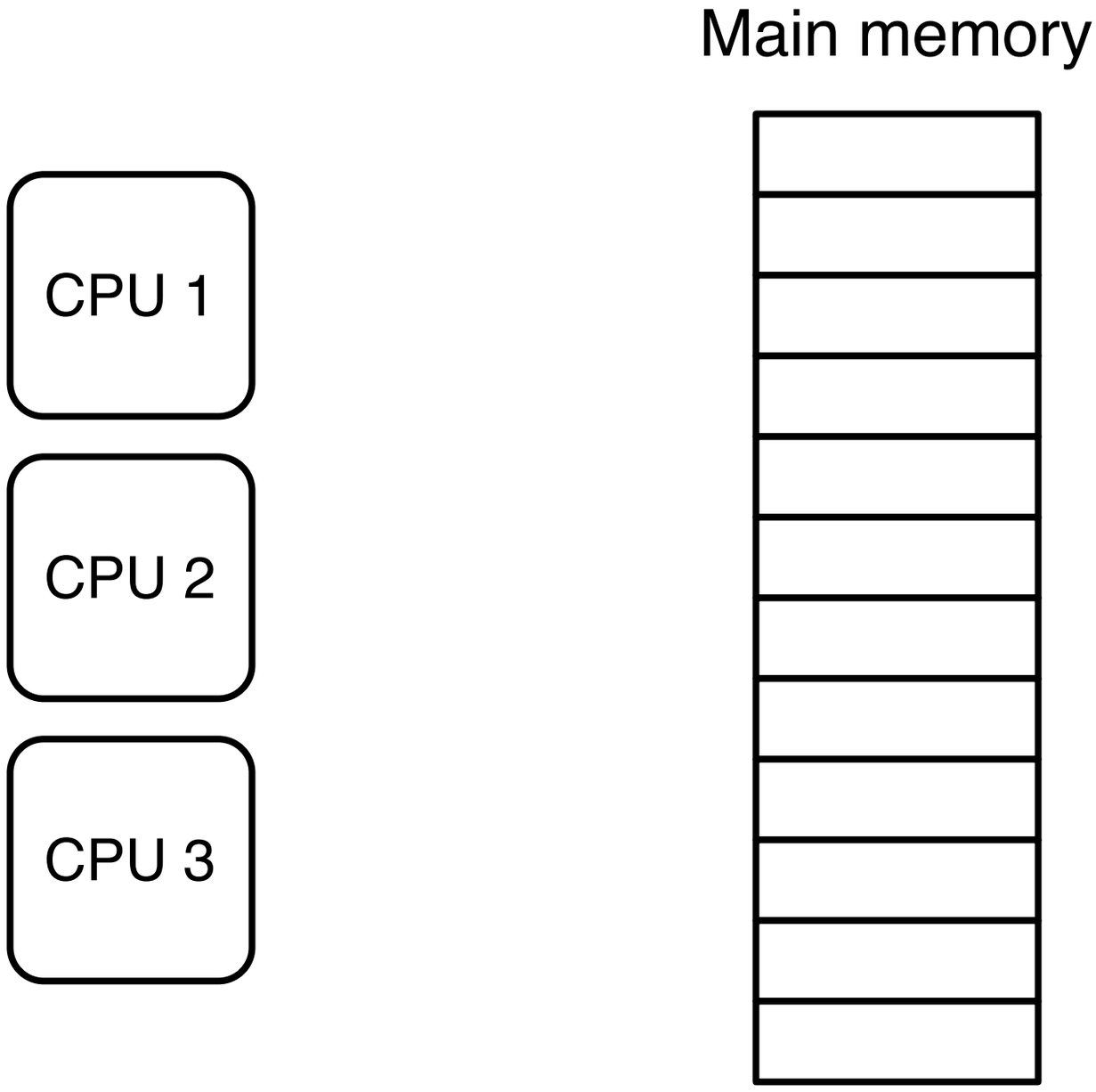
Setting



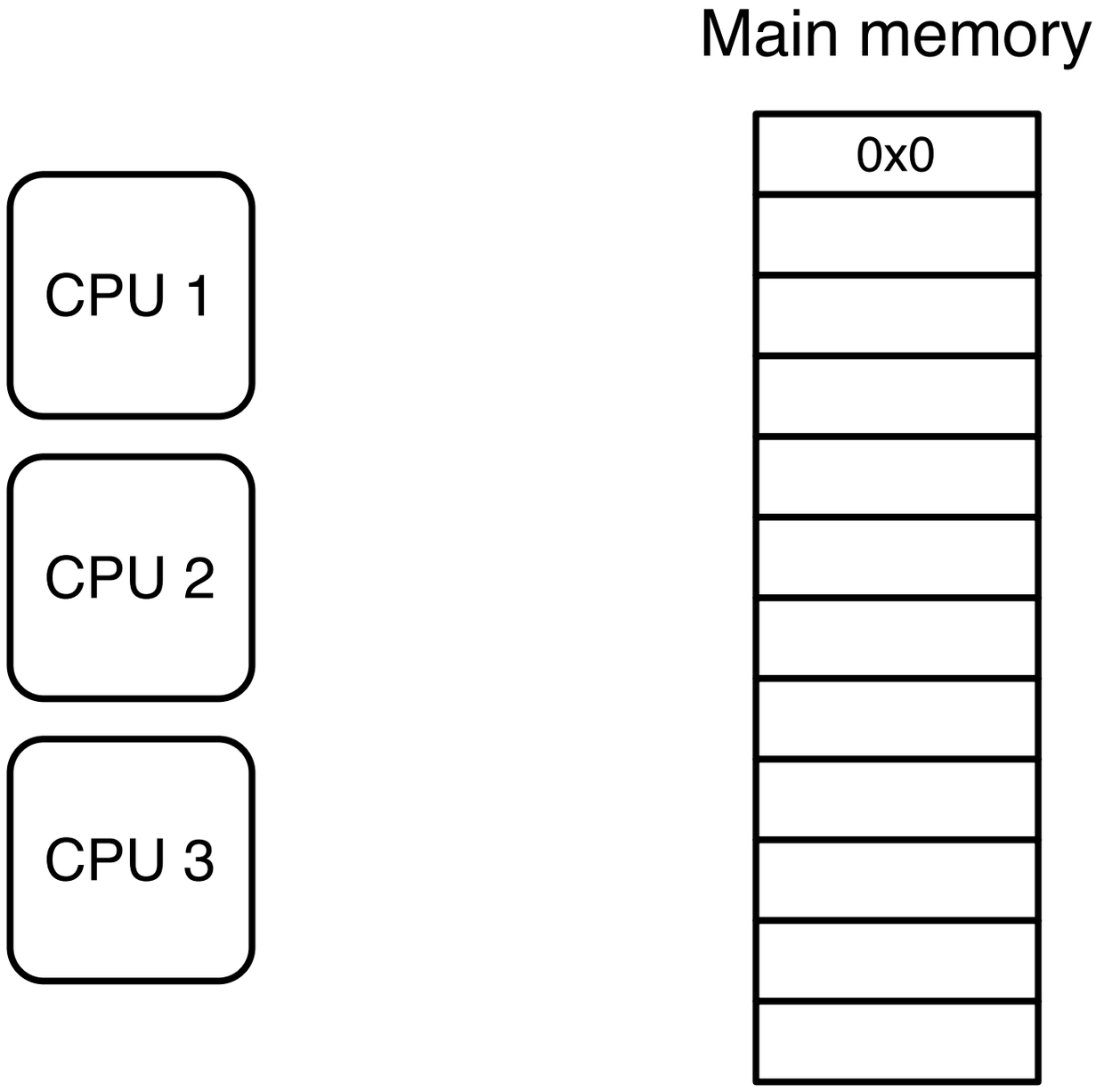
Setting



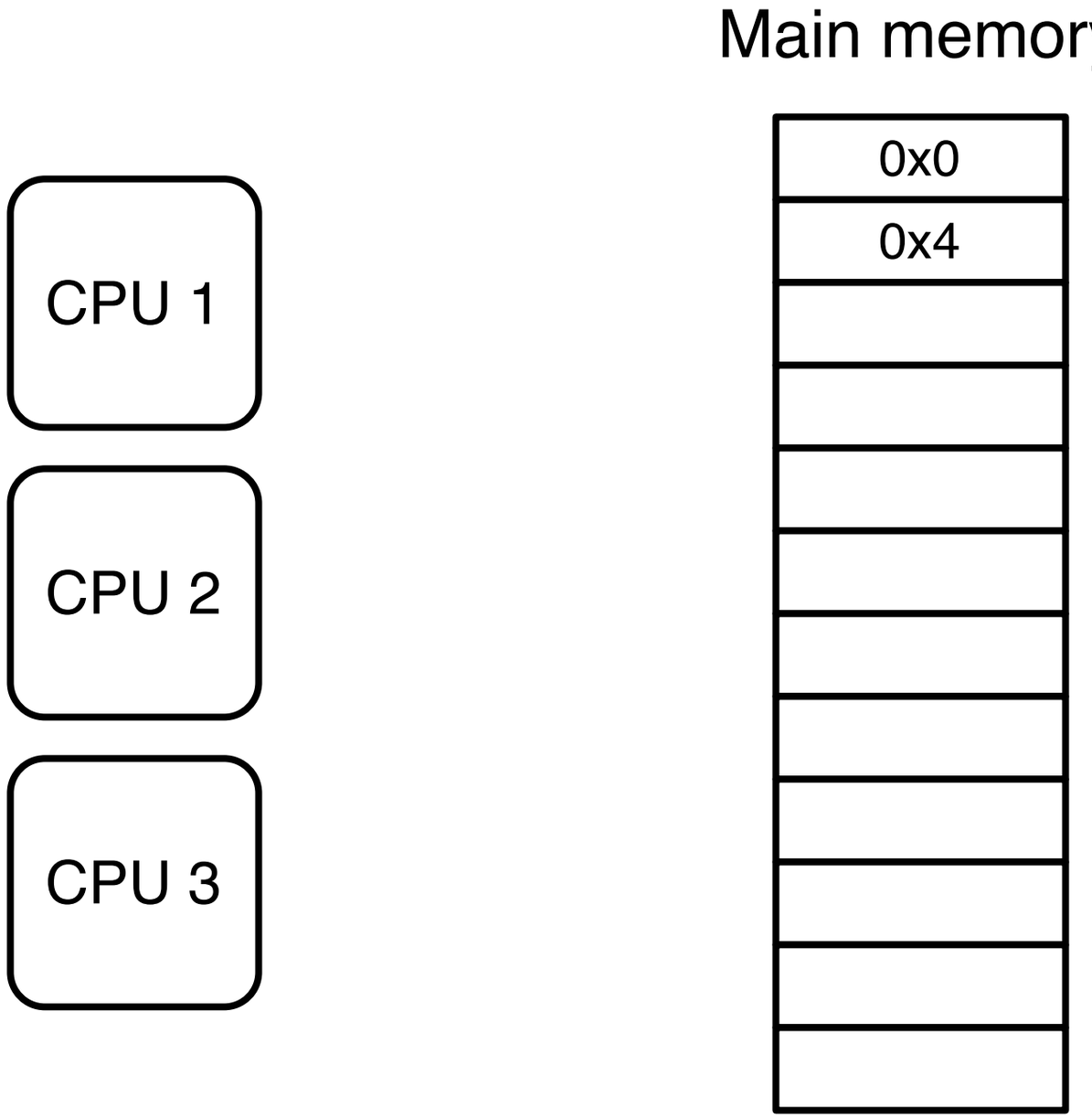
Setting



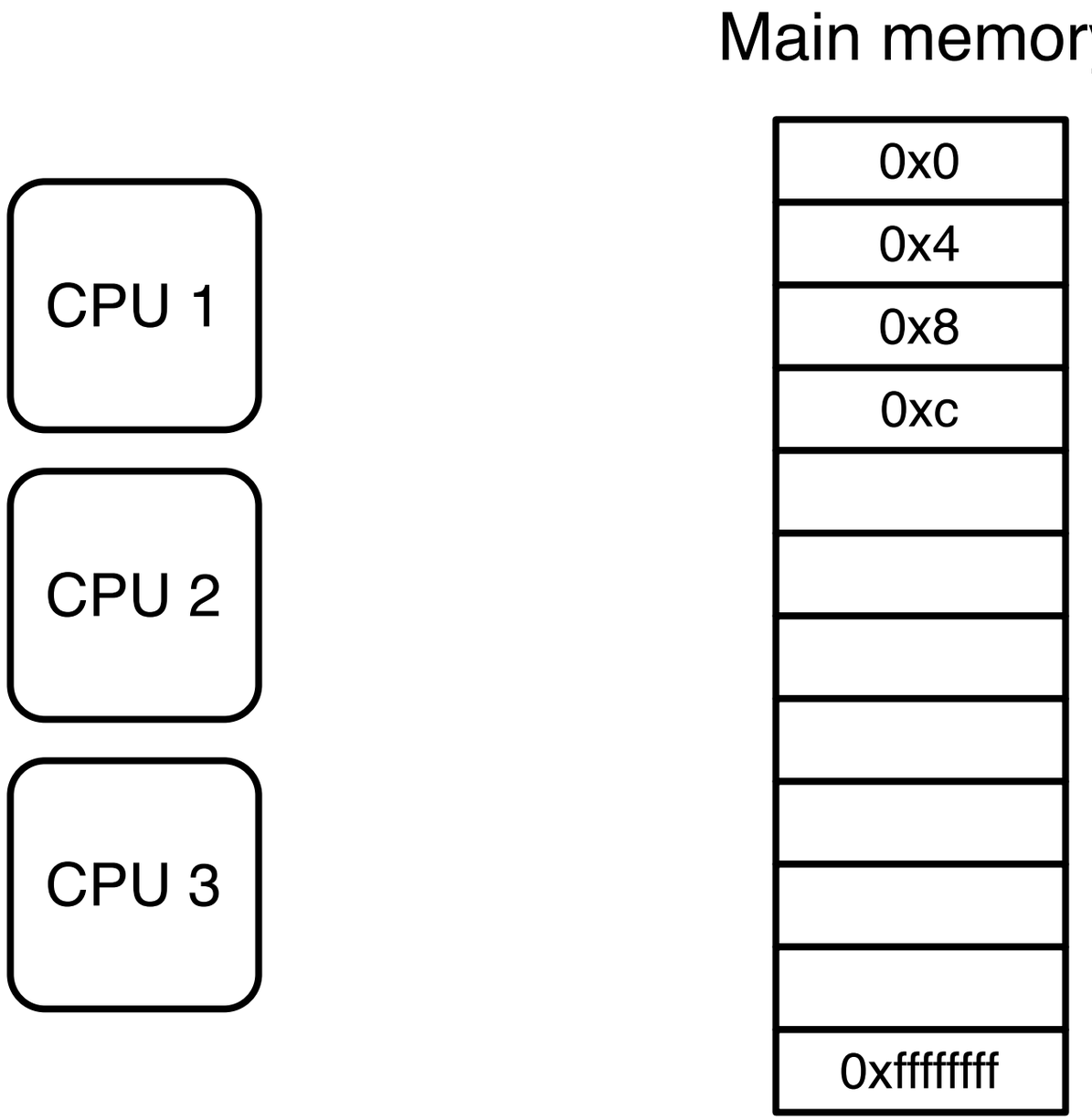
Setting



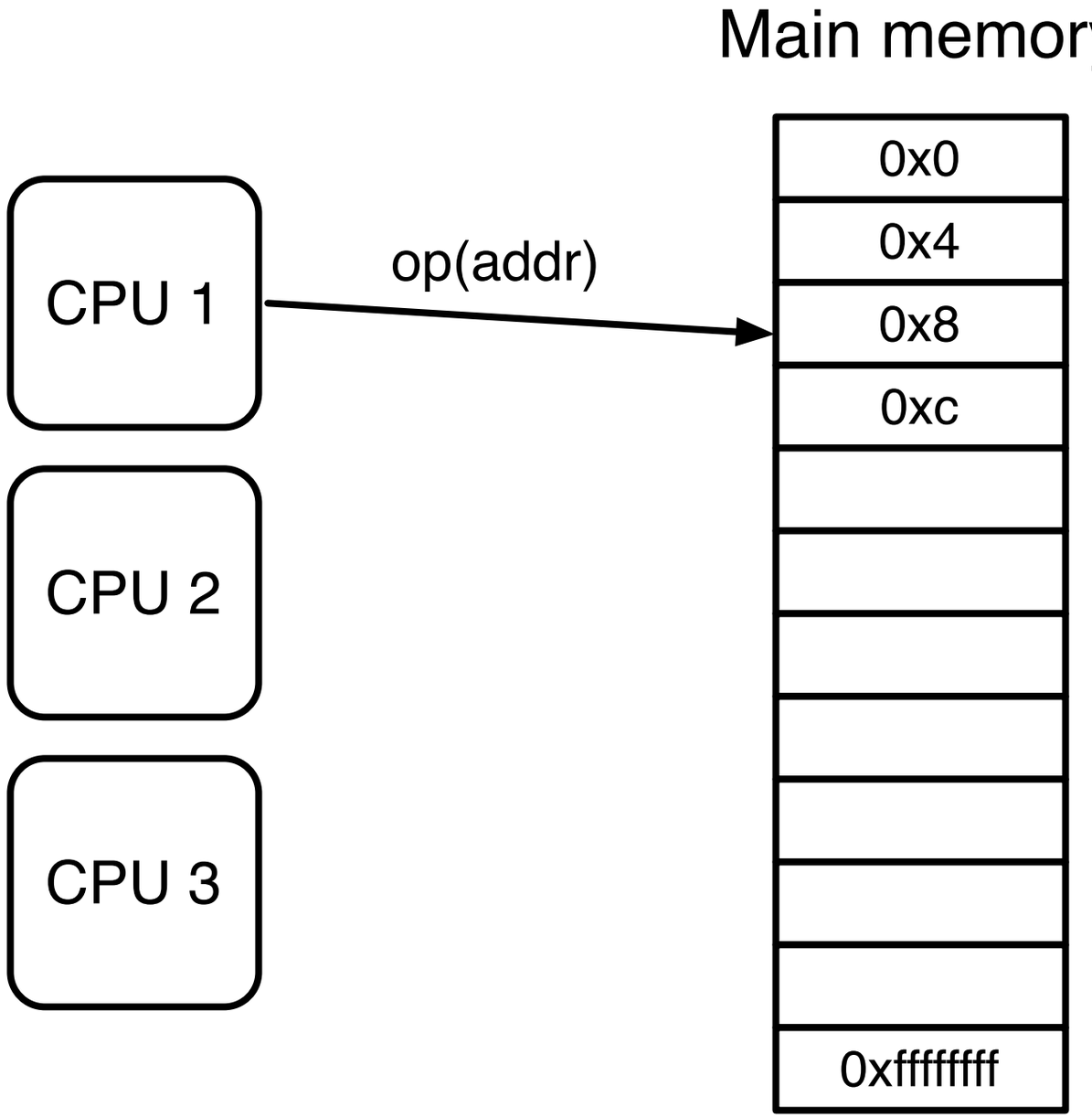
Setting



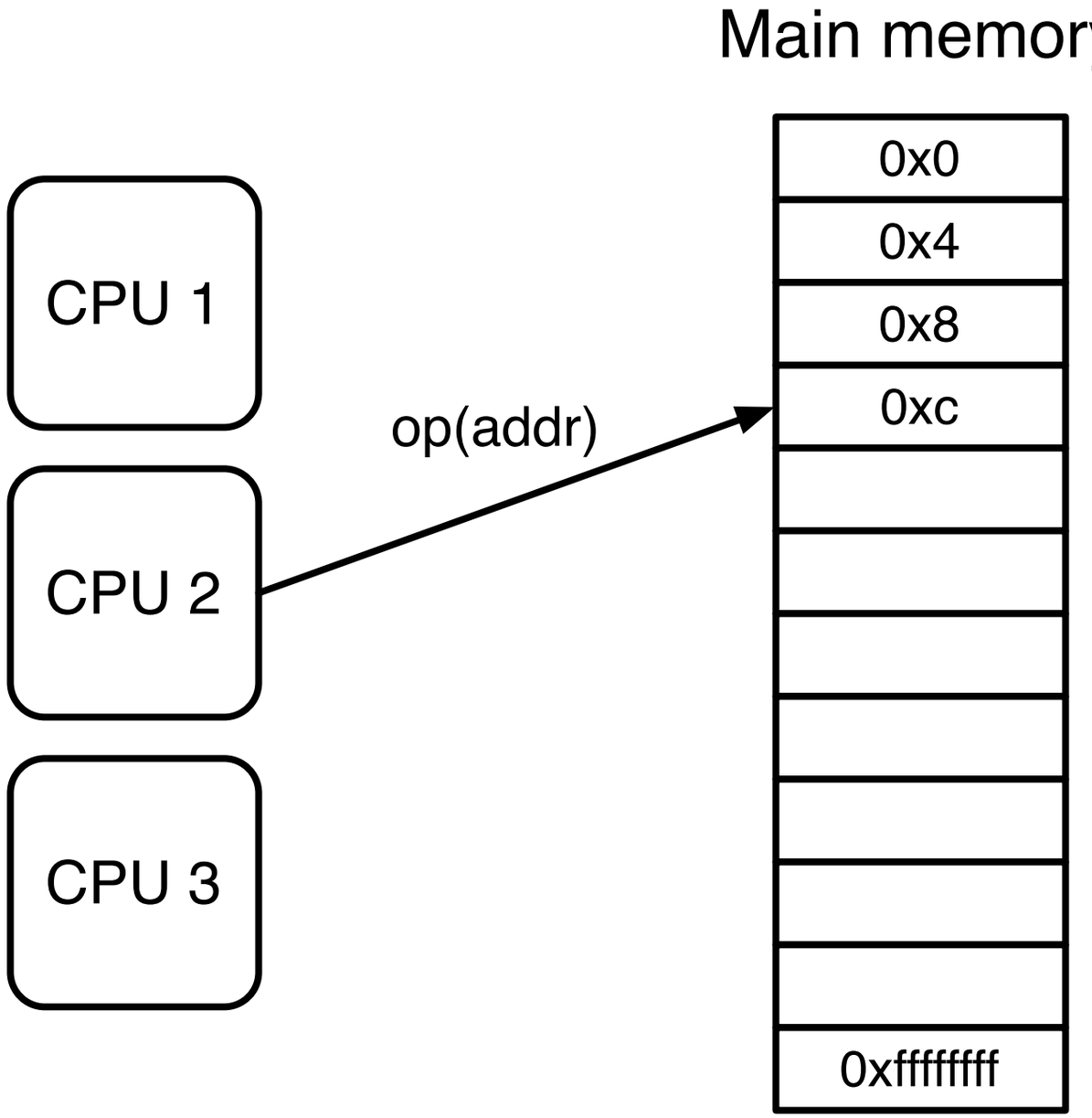
Setting



Setting


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

Setting


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

Setting (2)

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

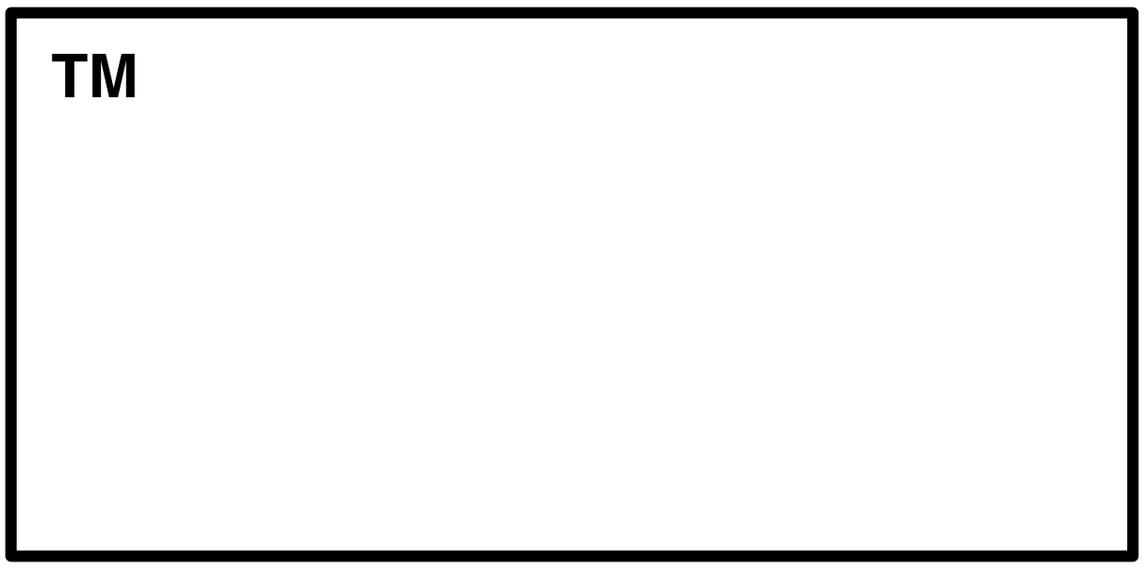
Setting (3)

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

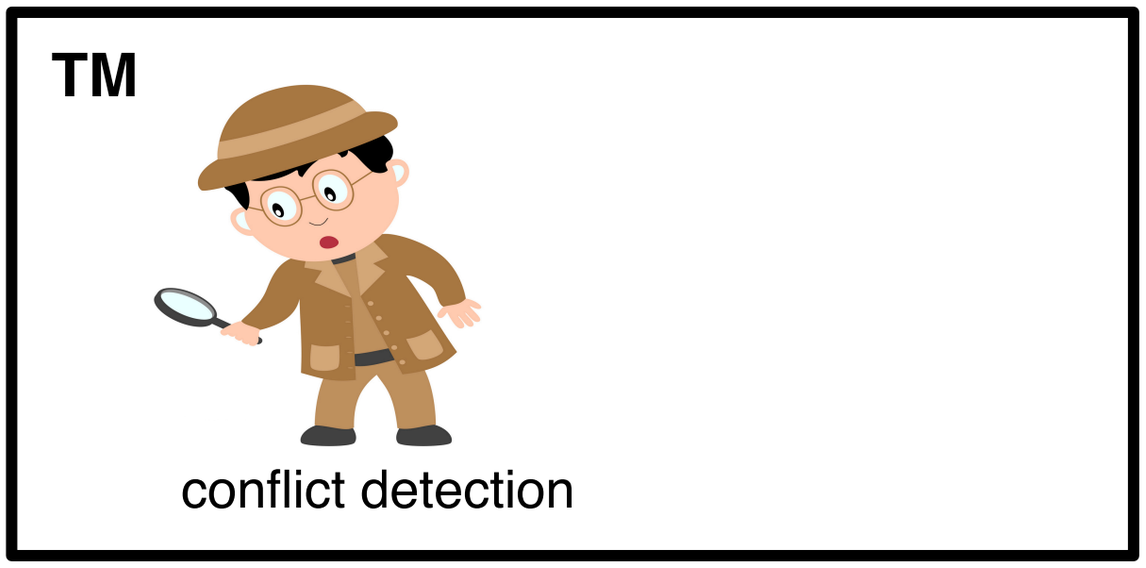
SwissTM

- Two phase locking
- Invisible reads
  - time-based validation
- Deferred updates
  - support rollback

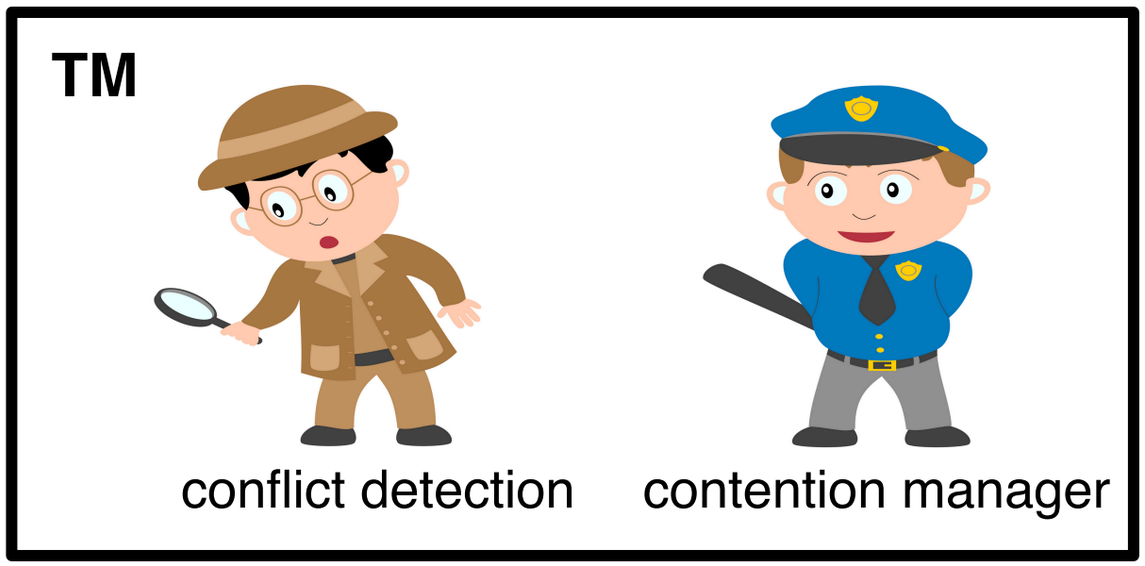
SwissTM design



SwissTM design

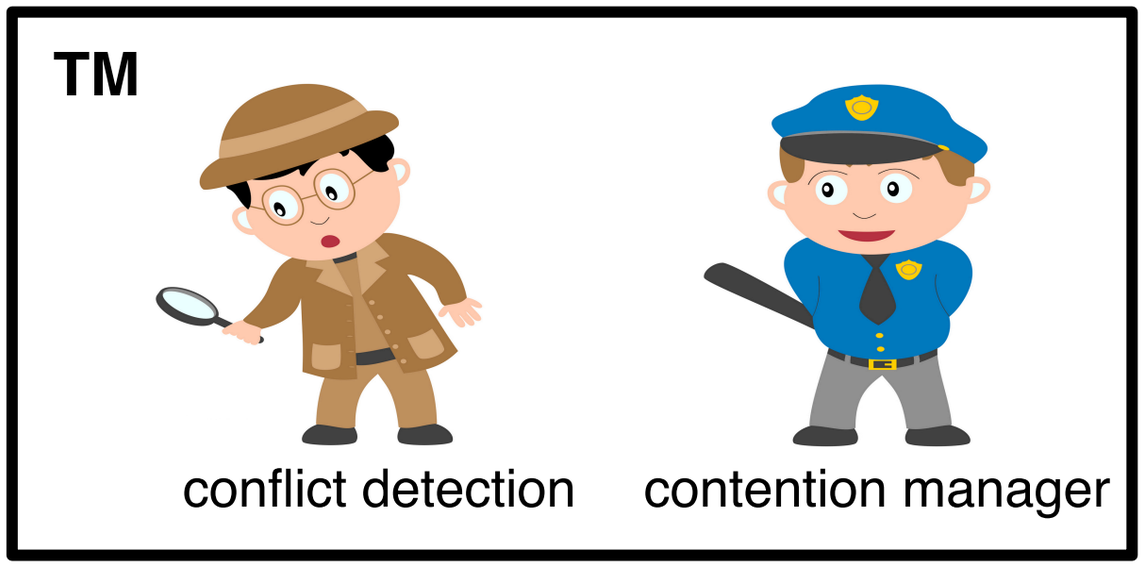


SwissTM design



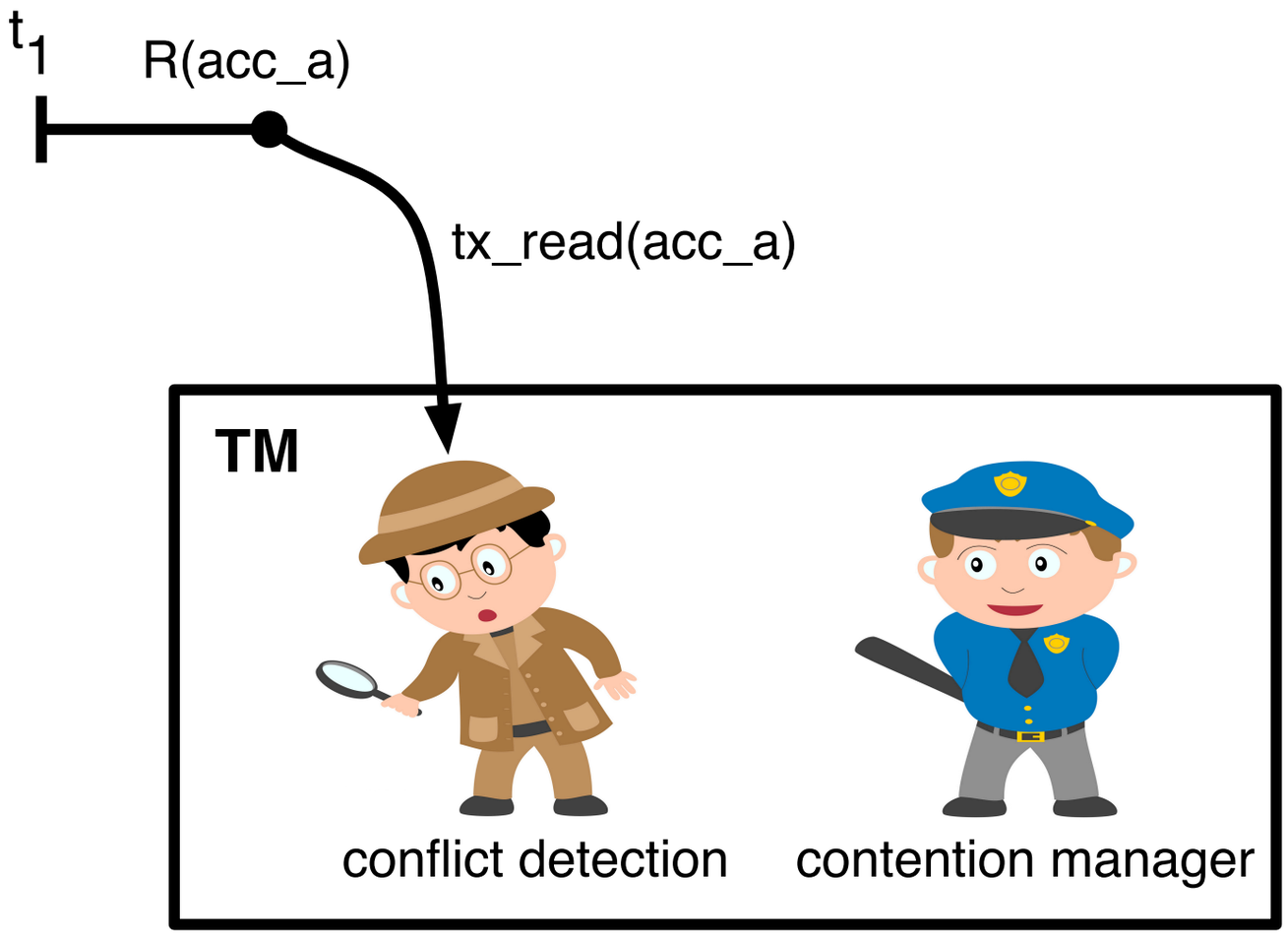
SwissTM design

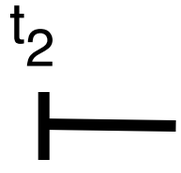
t_1



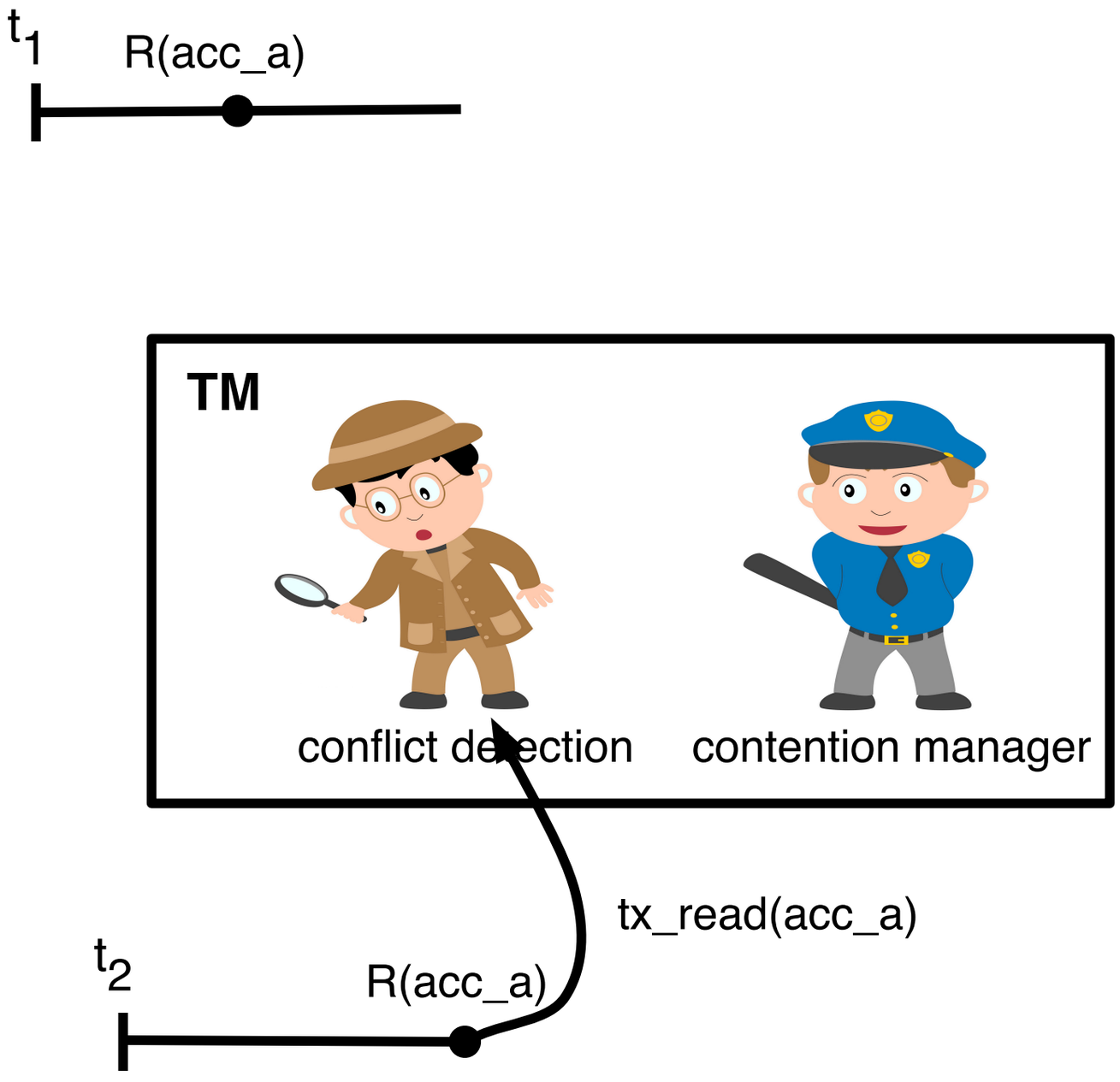
t_2

SwissTM design

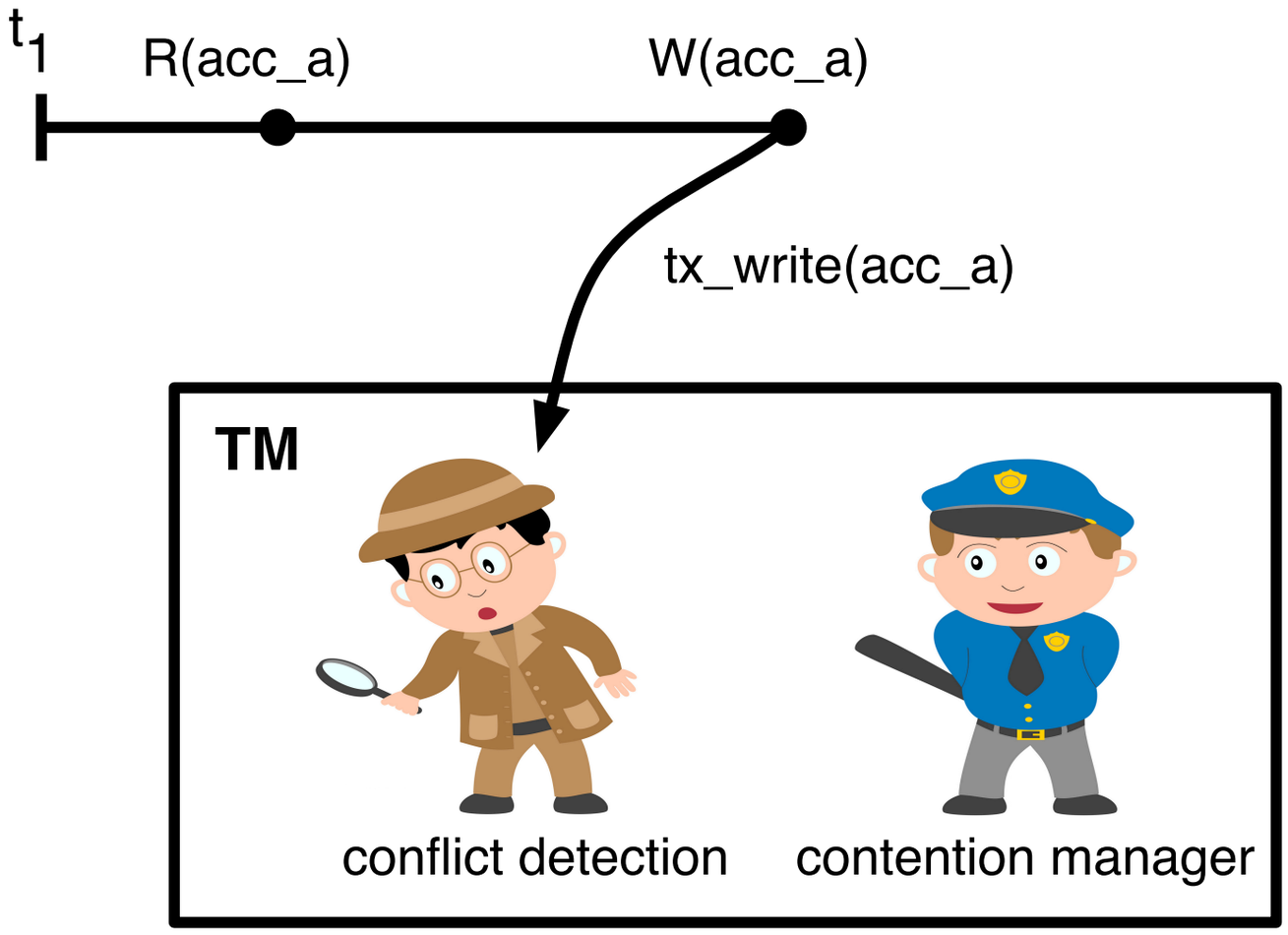


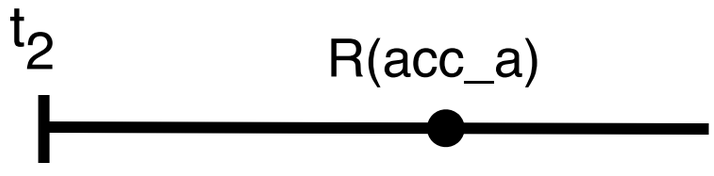


SwissTM design

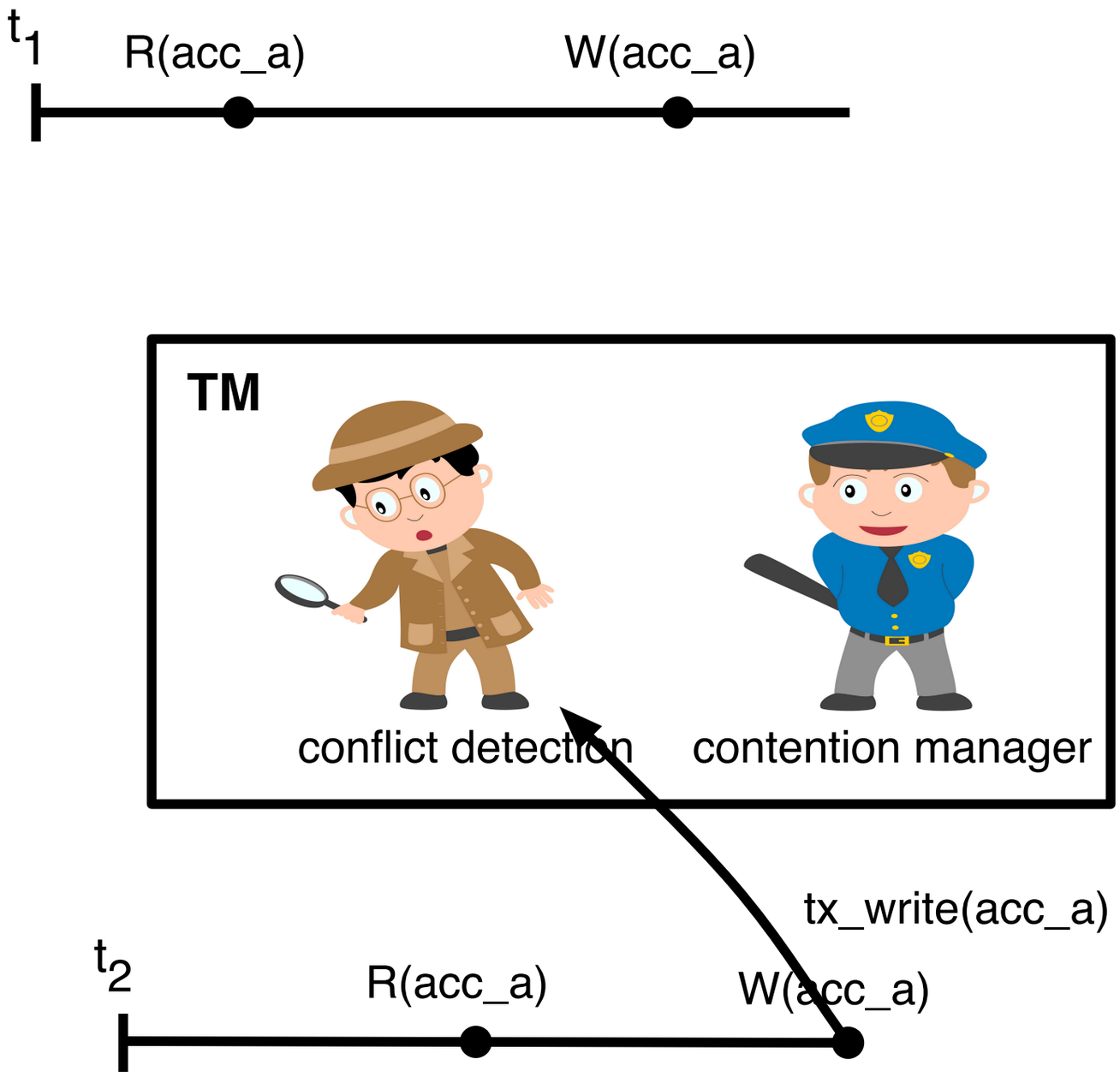


SwissTM design

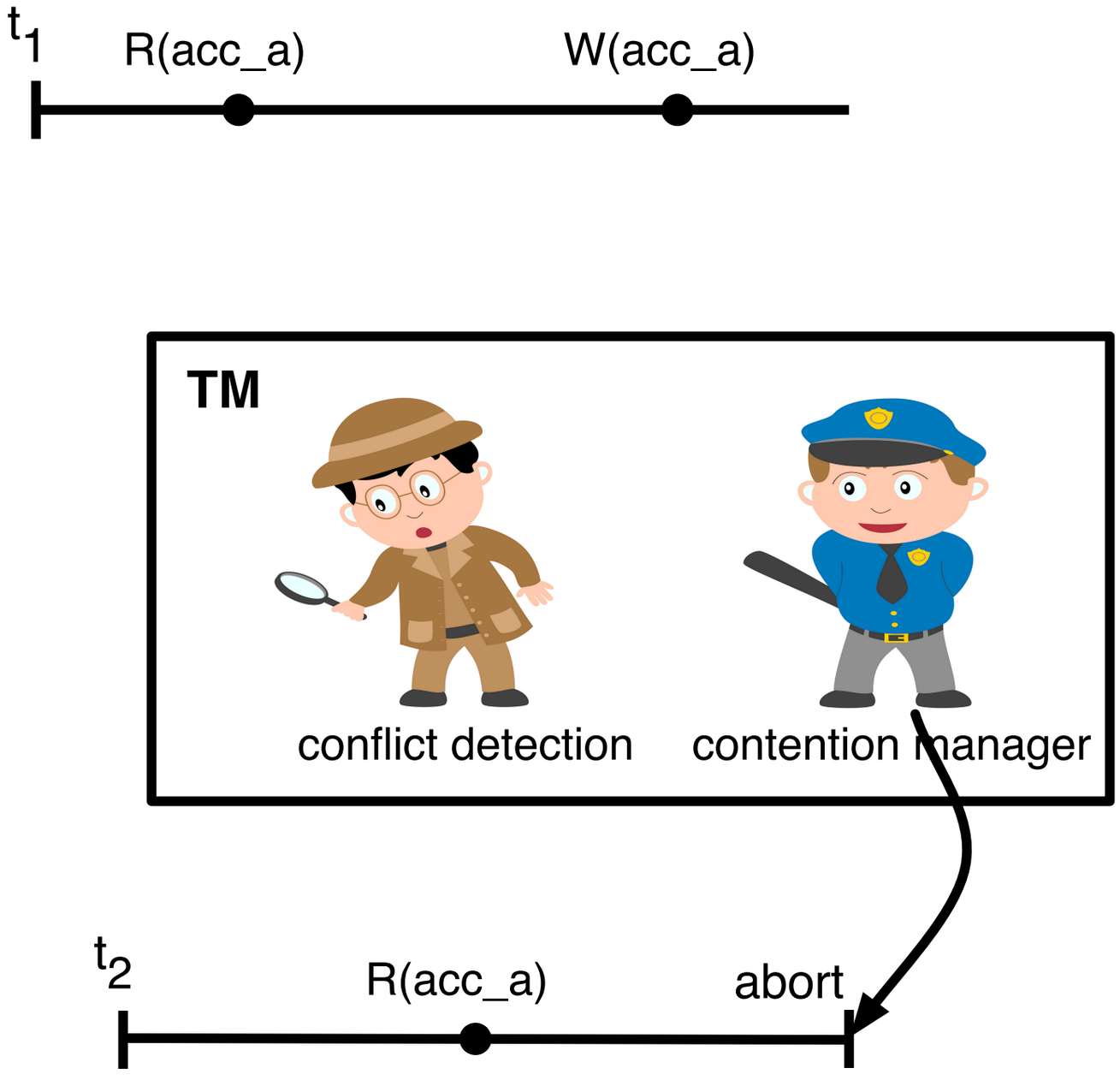




SwissTM design

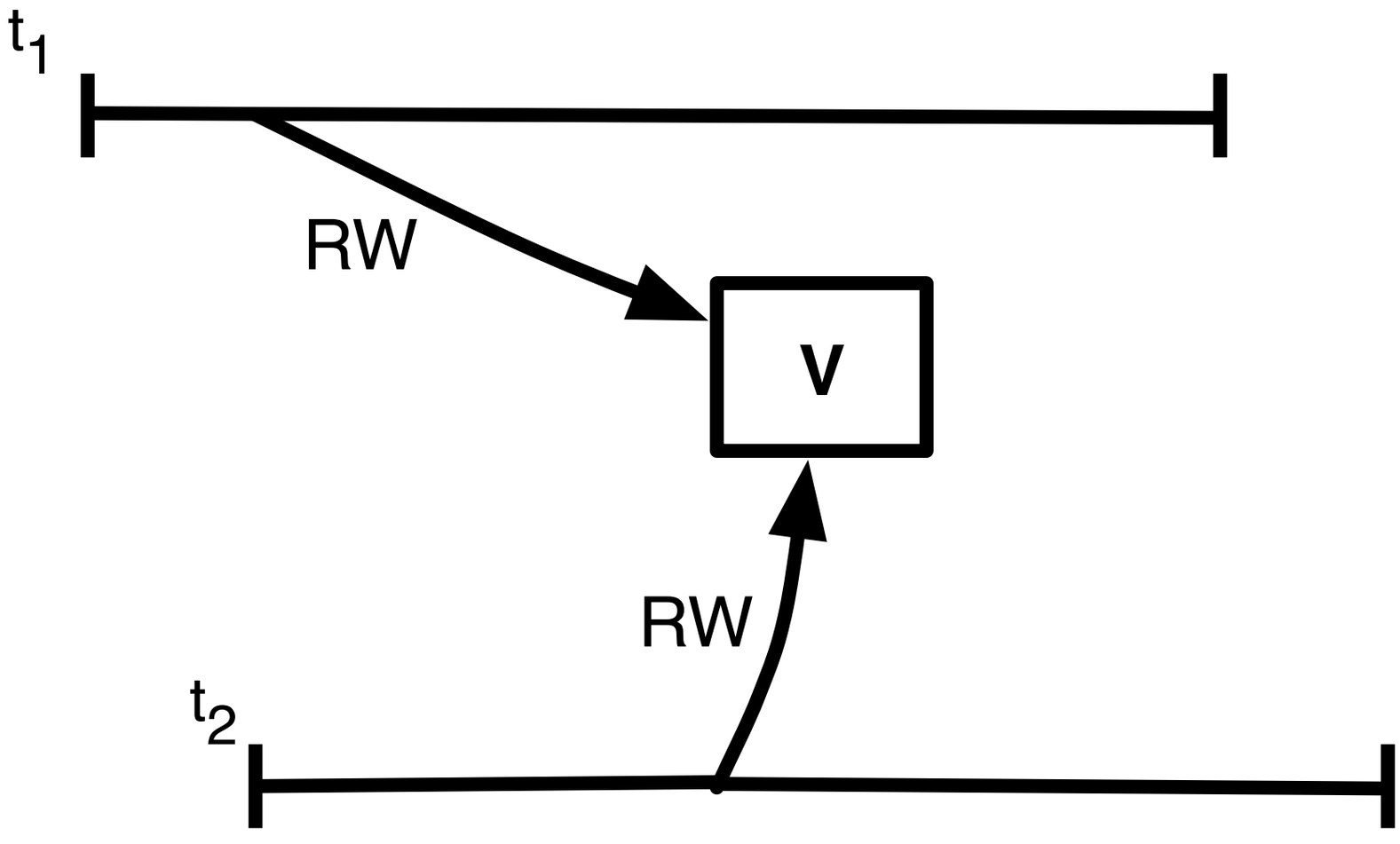


SwissTM design



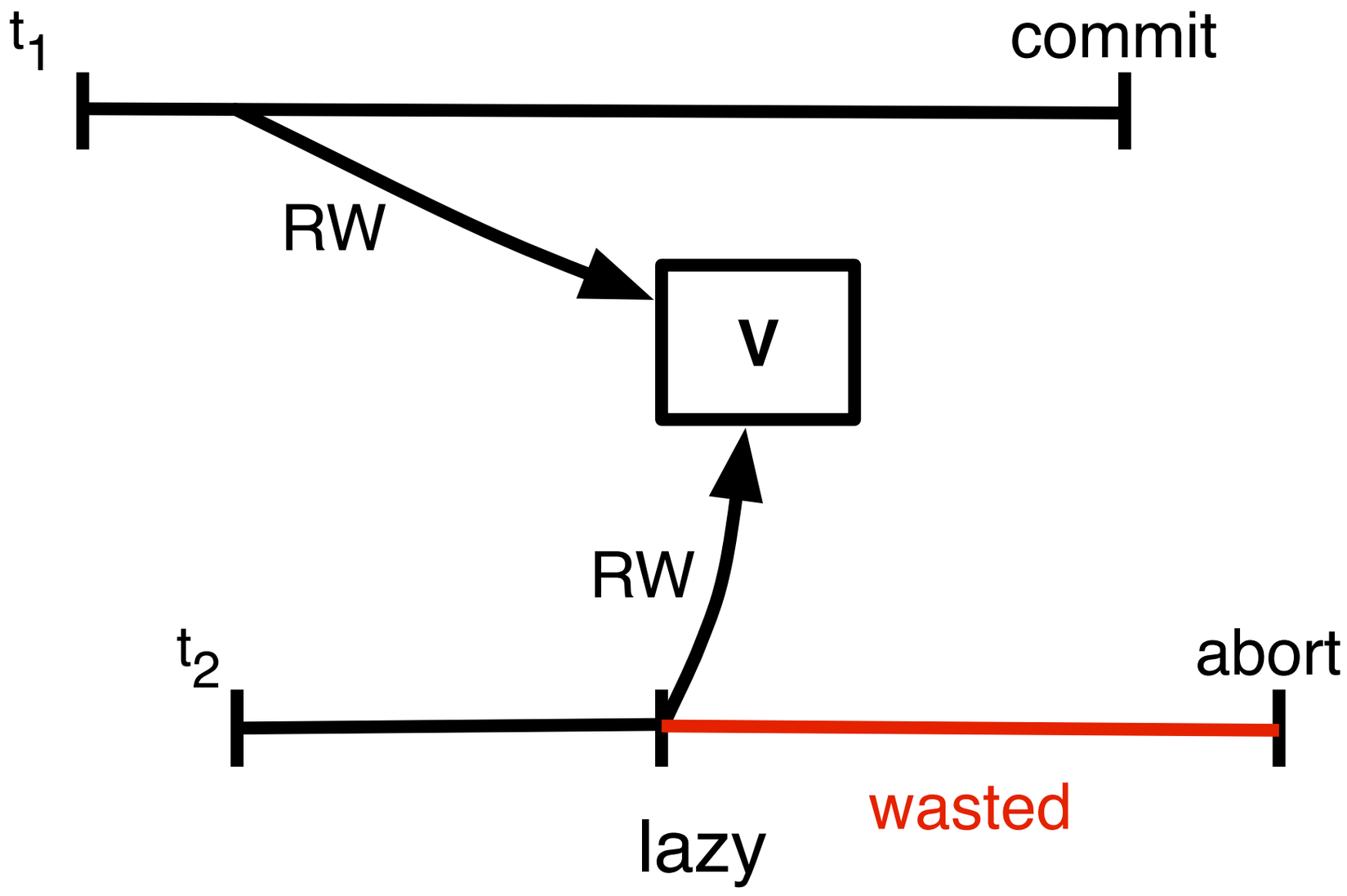
Conflict Detection

eager > lazy



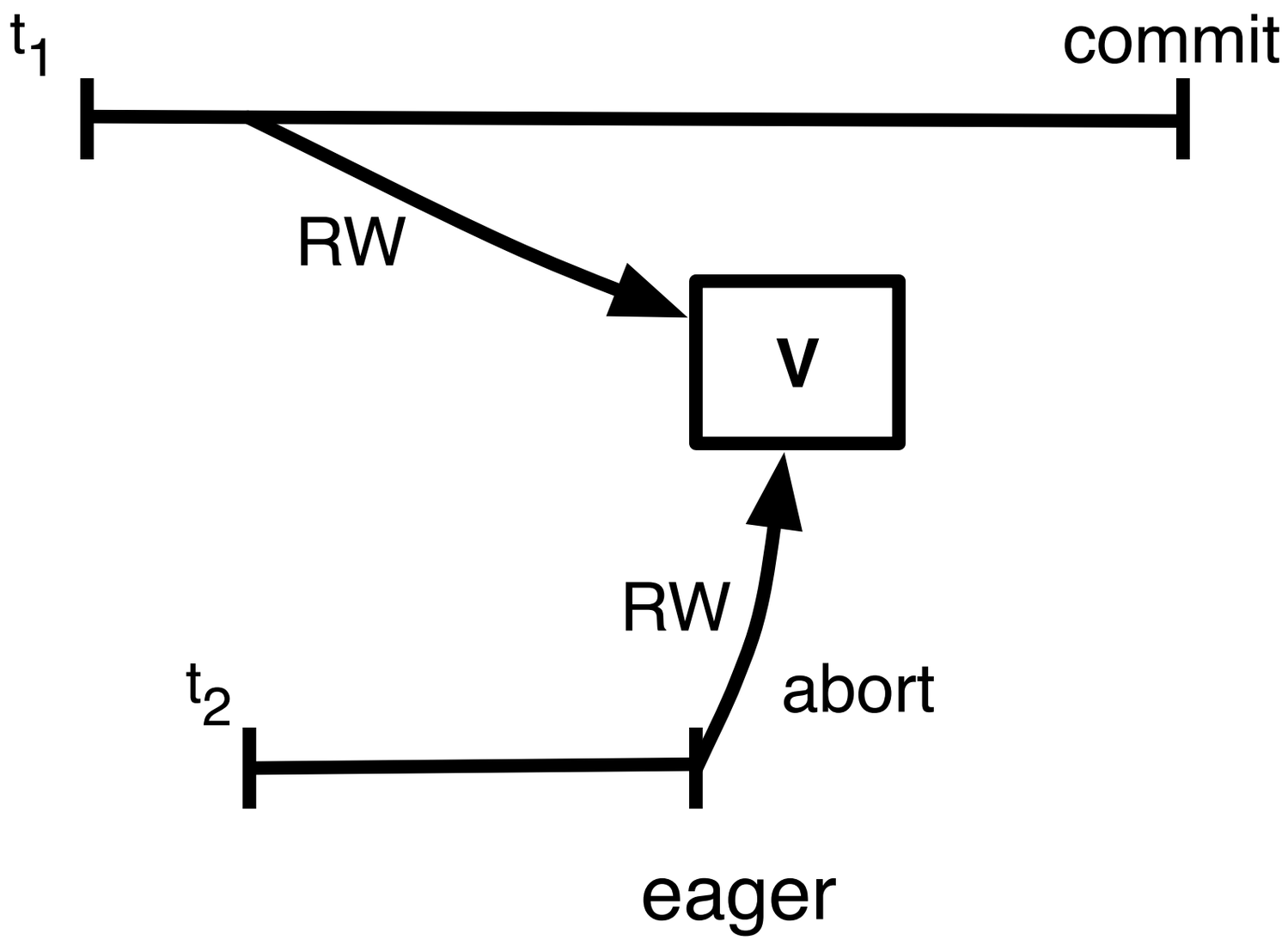
Conflict Detection

eager > lazy



Conflict Detection

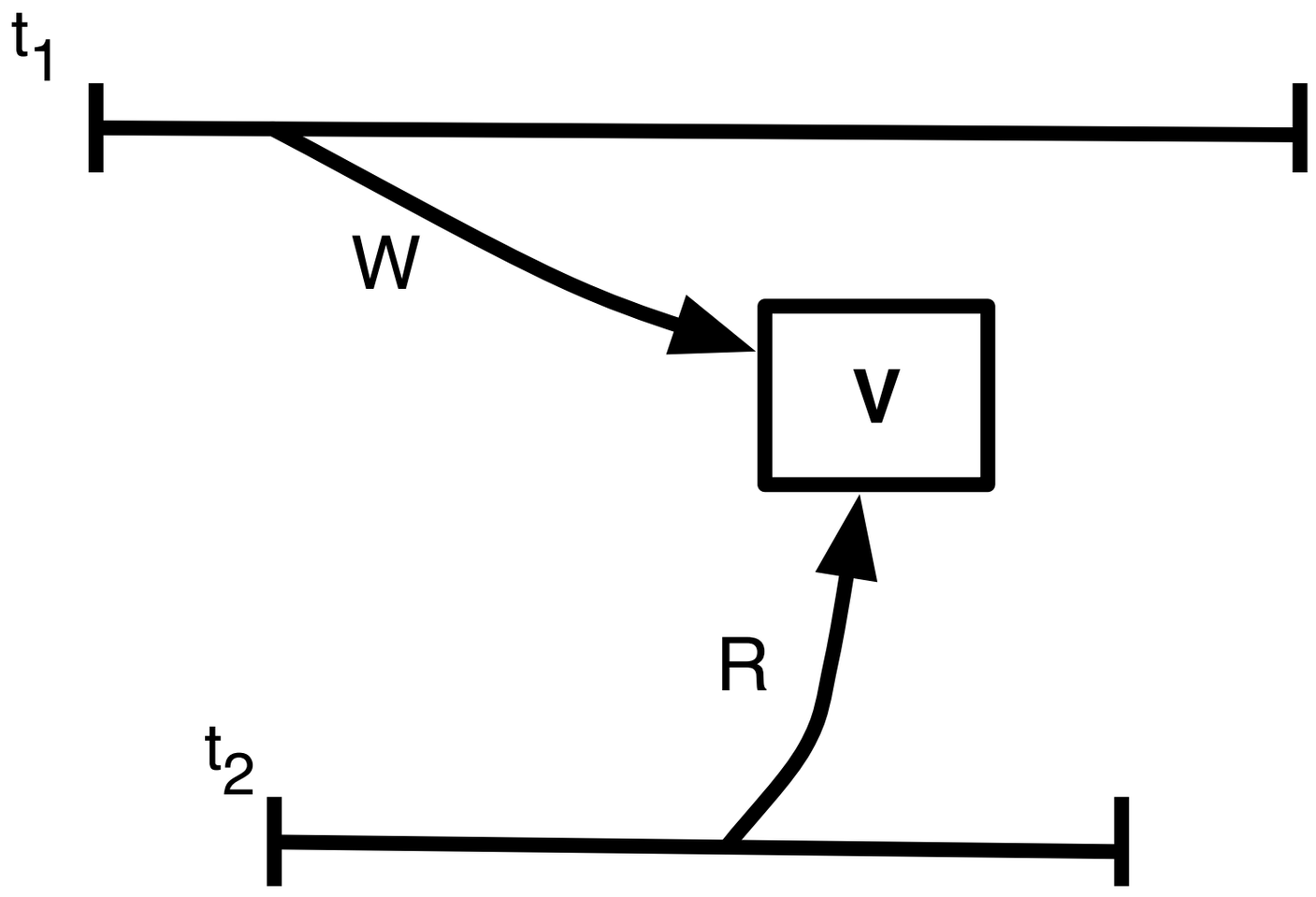
eager > lazy



Conflict Detection

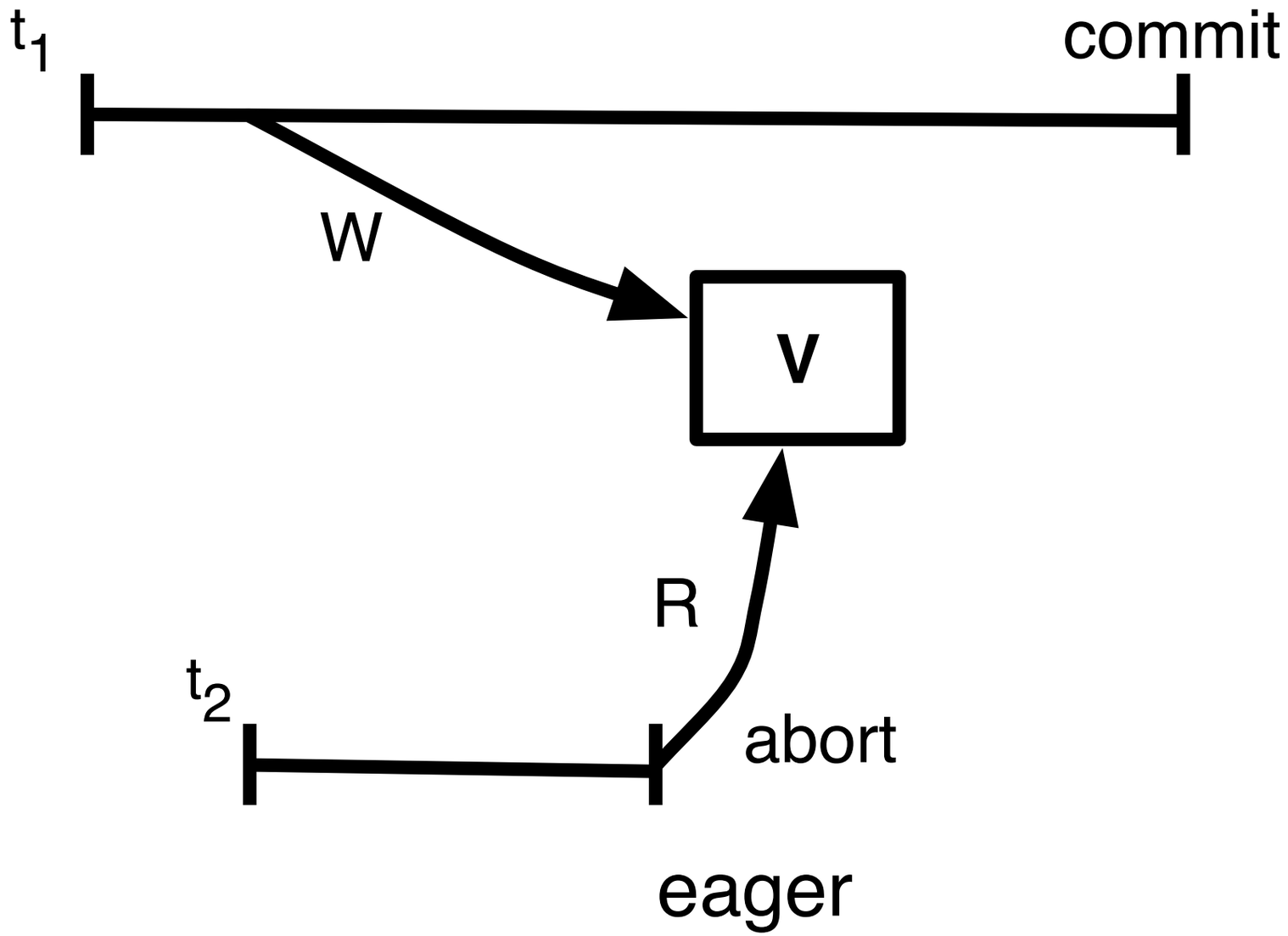
Conflict Detection

lazy > eager



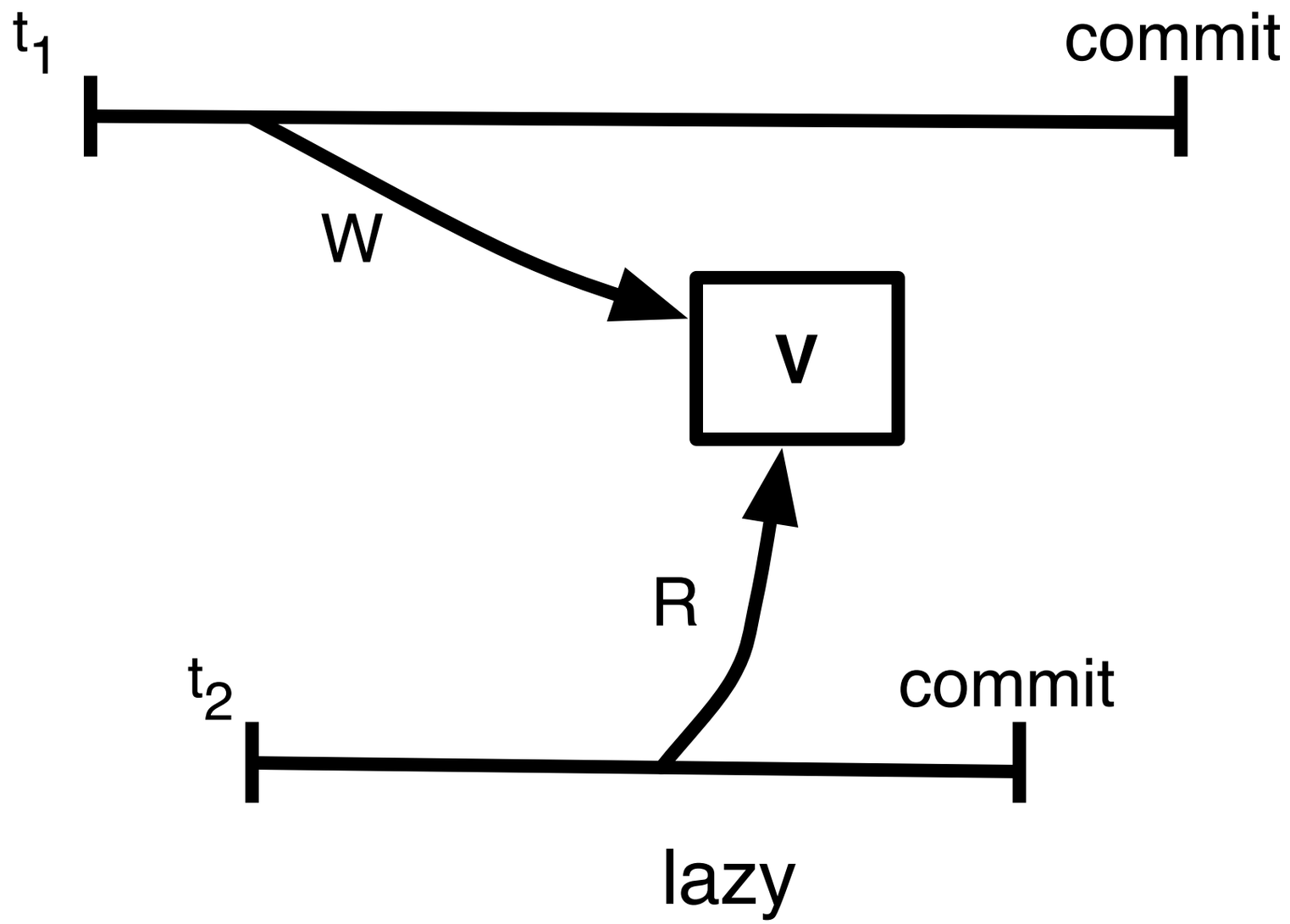
Conflict Detection

lazy > eager

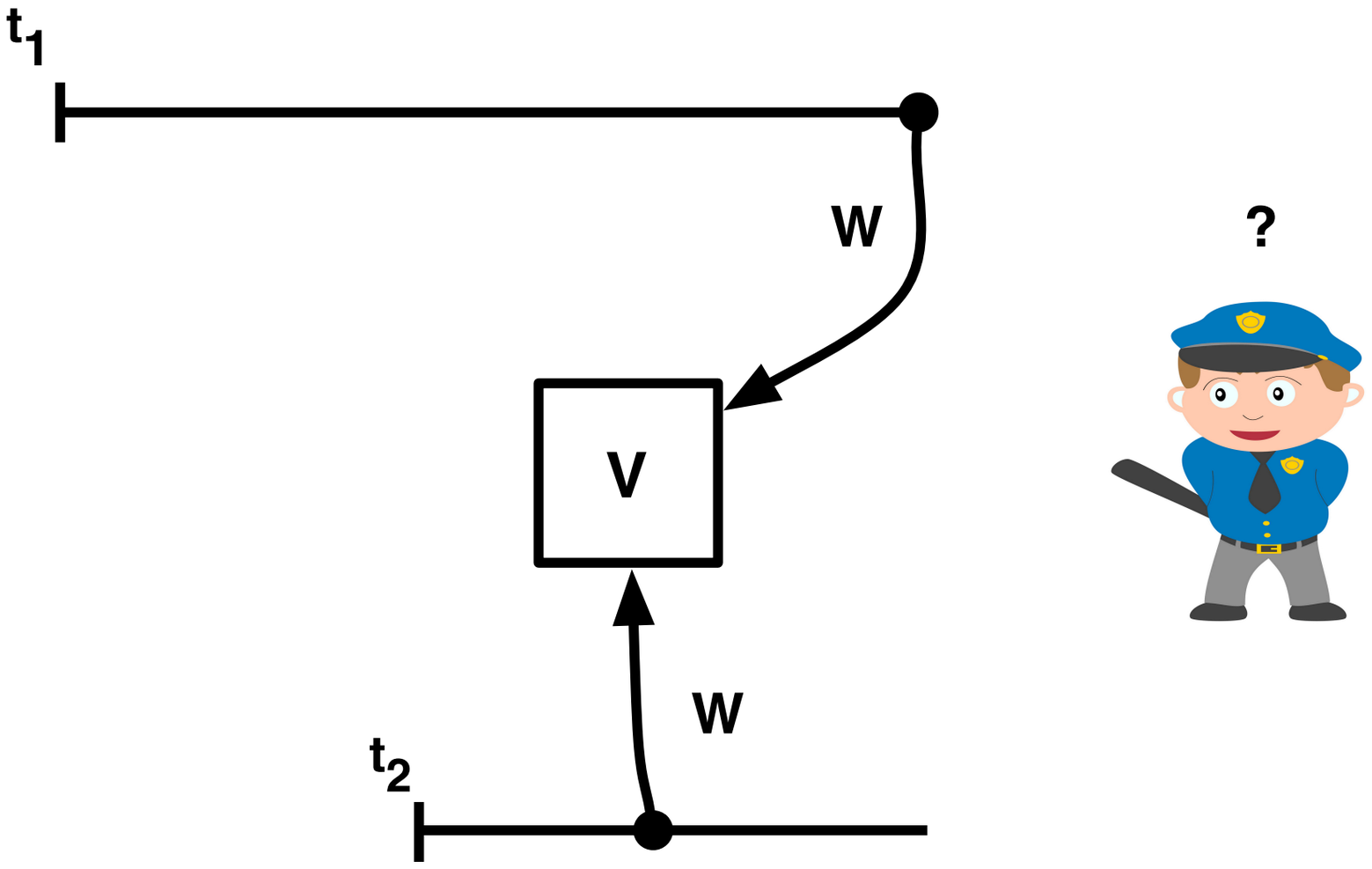


Conflict Detection

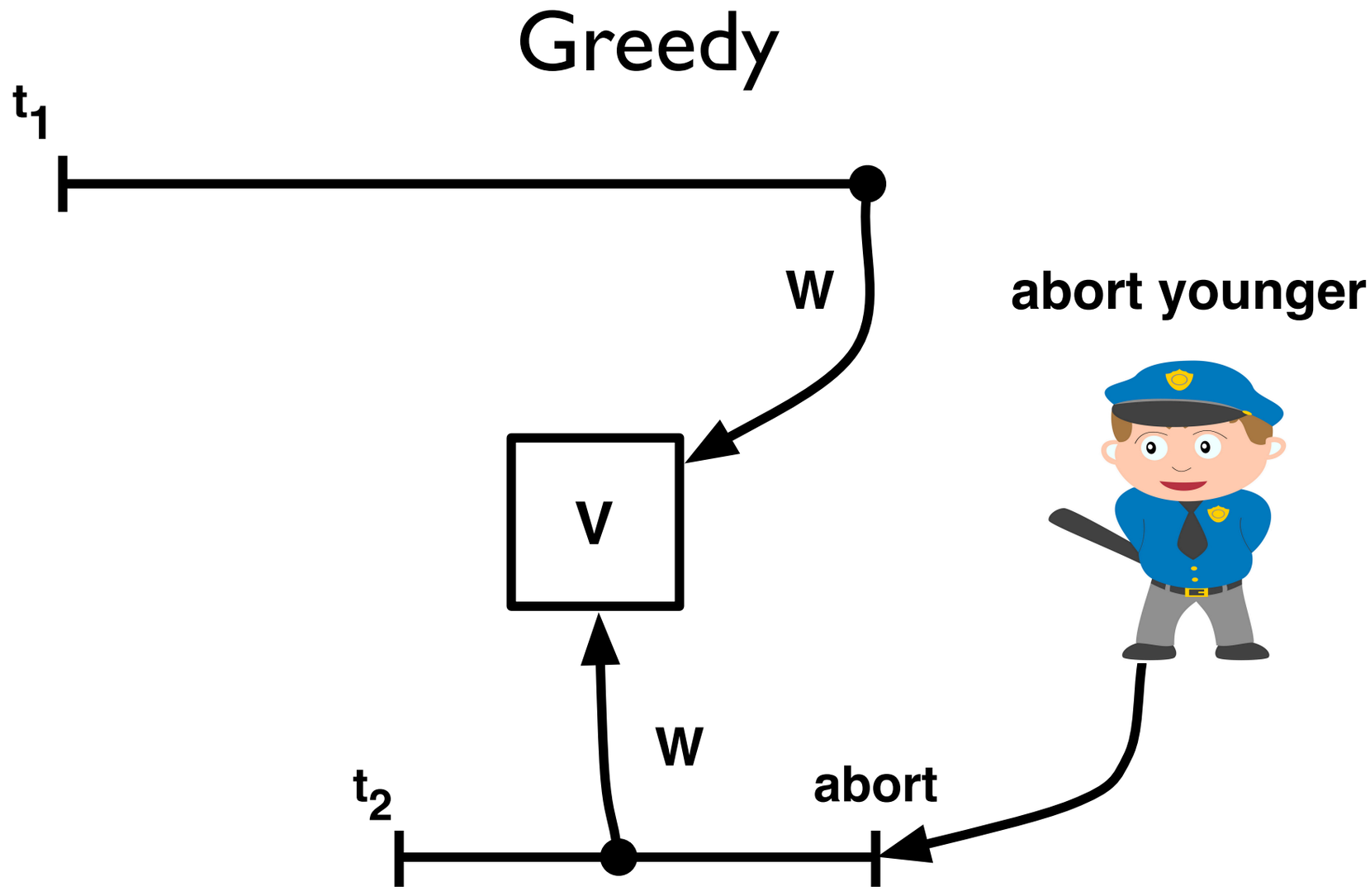
lazy > eager



Contention Manager

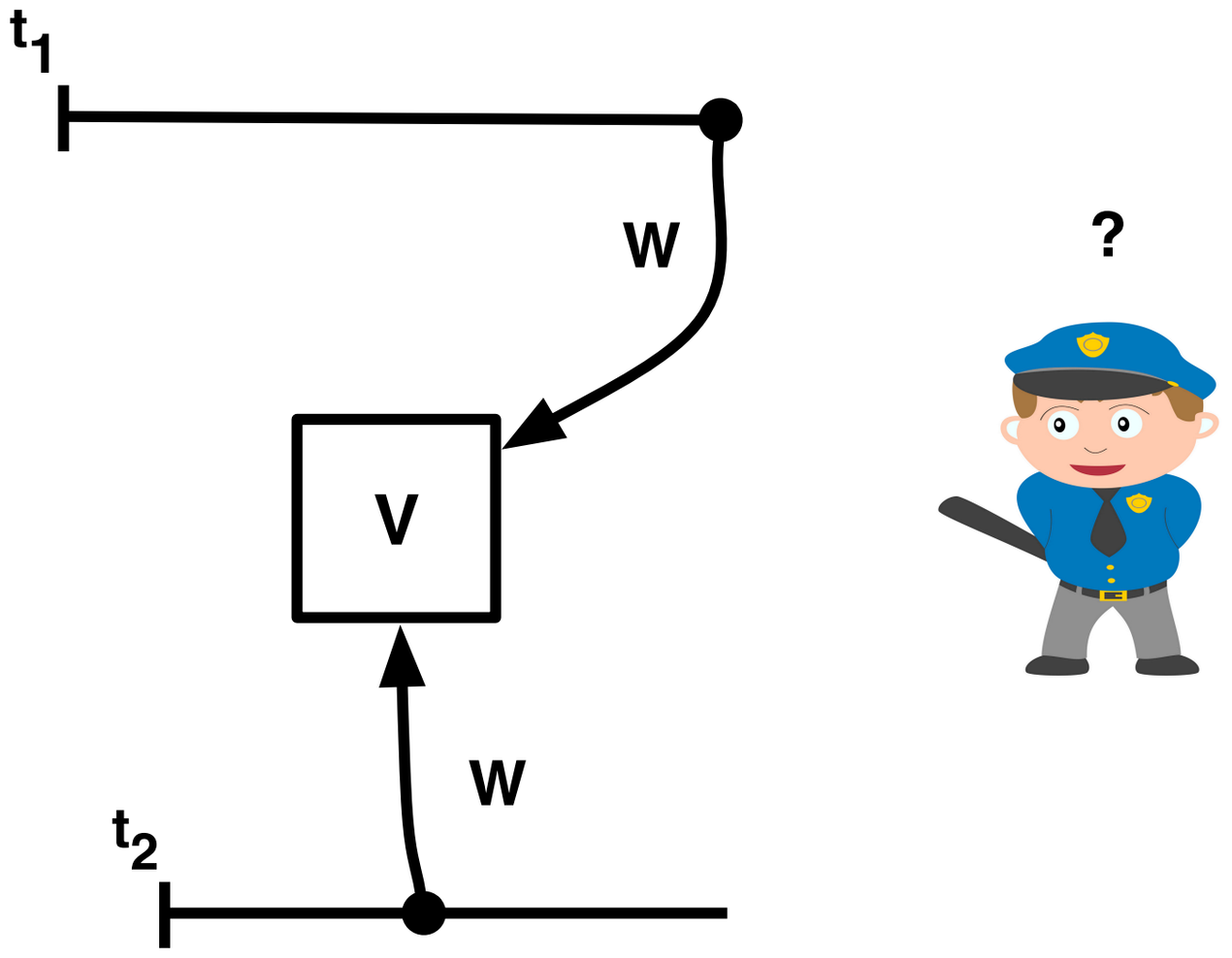


Contention Manager

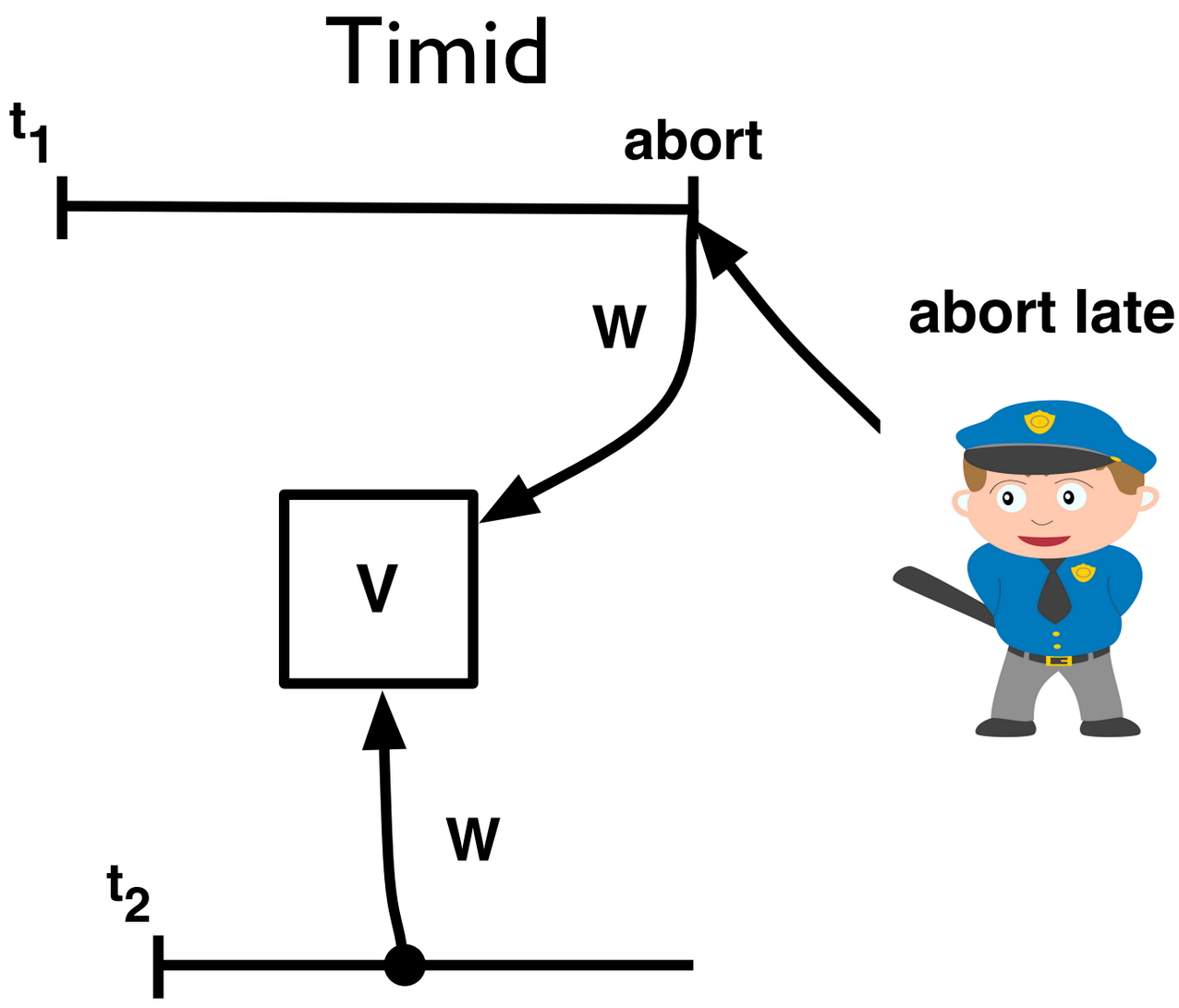


Contention Manager

Contention Manager



Contention Manager



SwissTM Design

- Mixed invalidation
  - lazy for read/write
  - eager for write/write
- Two-phase contention manager
  - timid for short
  - greedy for long

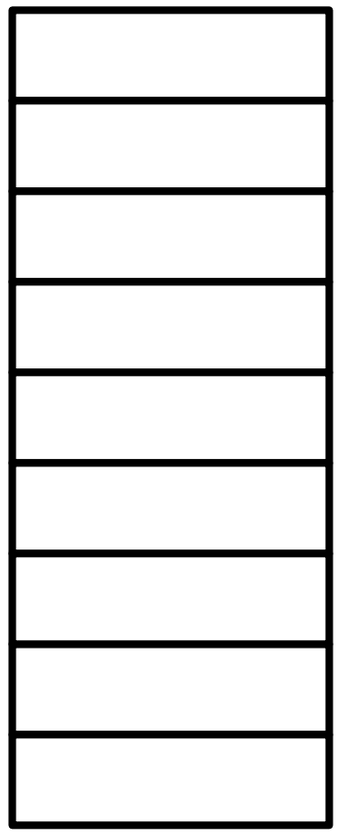
Starting point

```
void tx_start()

word_t tx_read(word_t *addr)

void tx_write(word_t *addr, word_t val)

void tx_commit()
```

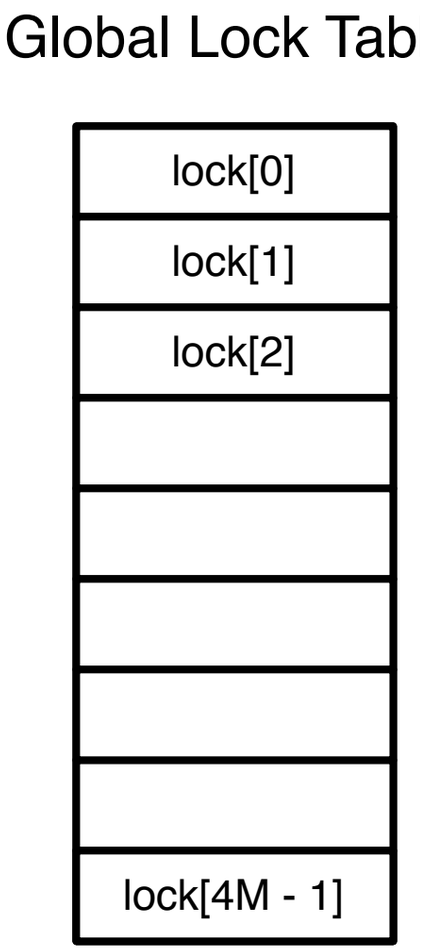
Where is the lock?

Where is the lock?

Global Lock Table

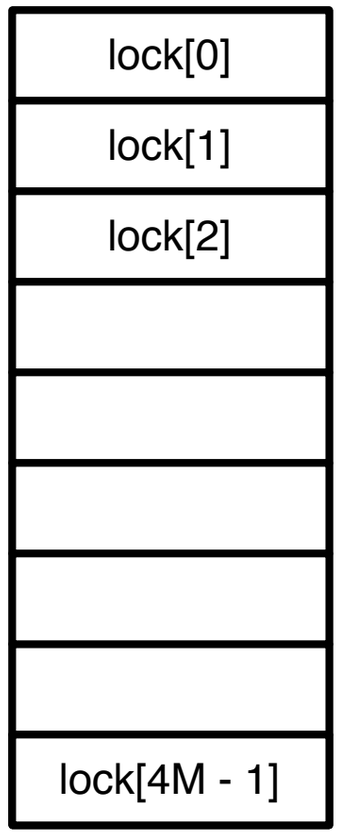


Where is the lock?



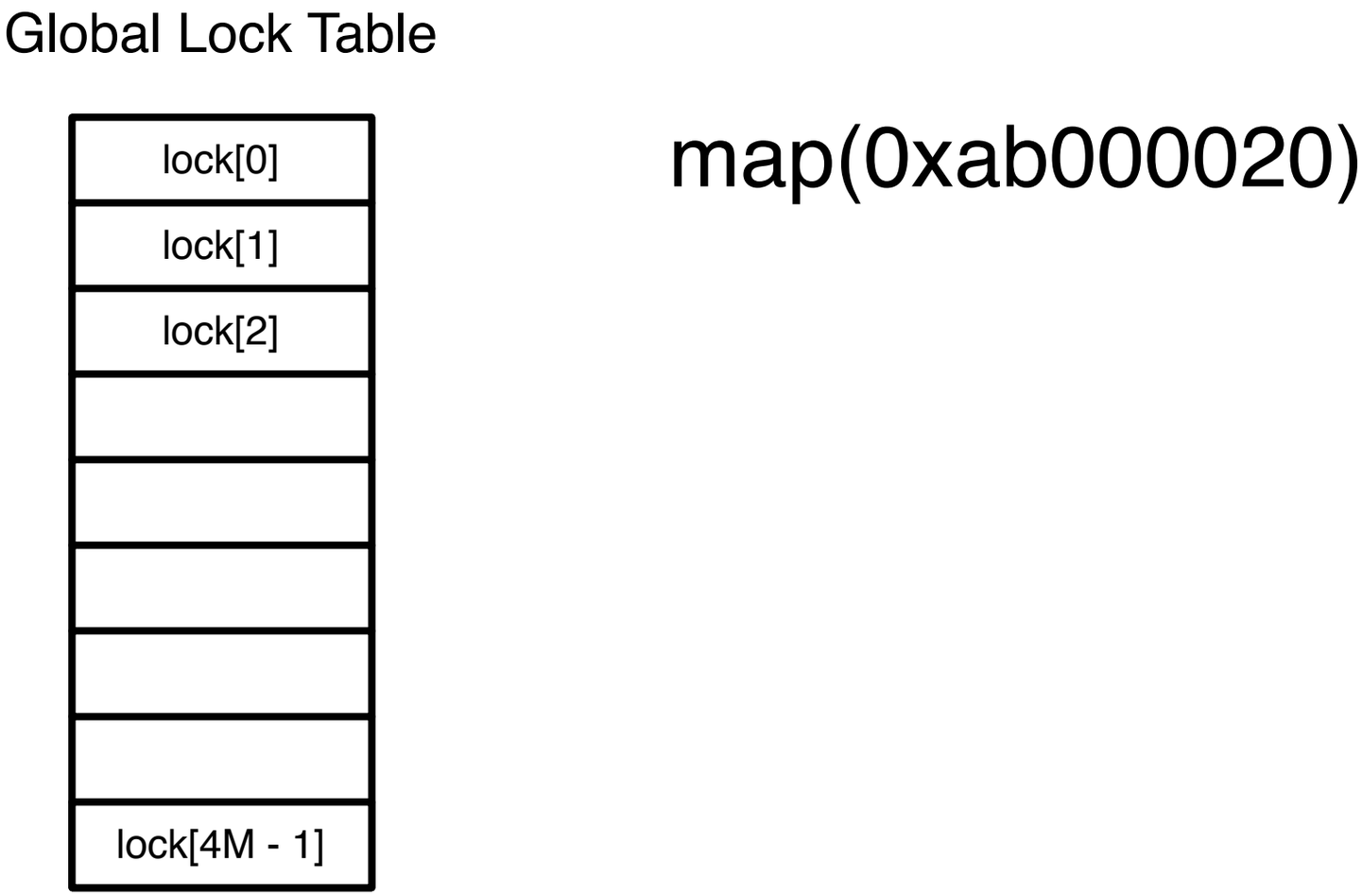
Where is the lock?

Global Lock Table



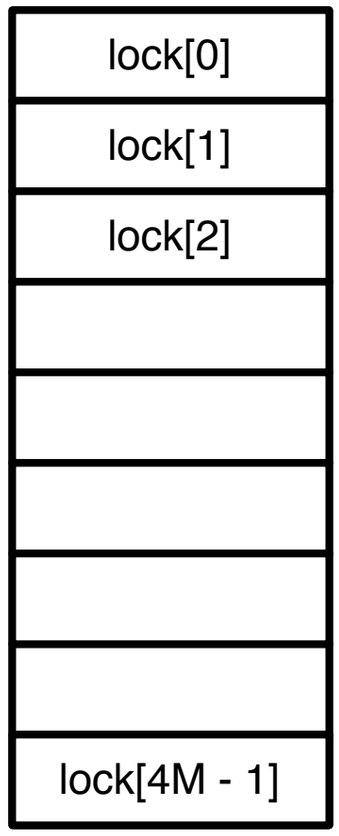
map(addr)

Where is the lock?

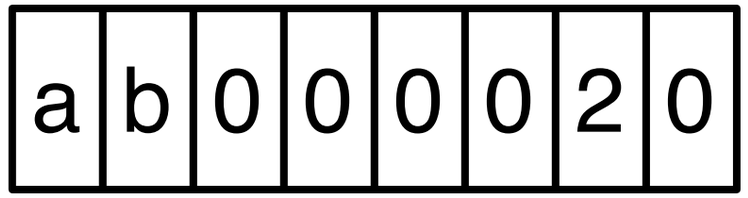


Where is the lock?

Global Lock Table

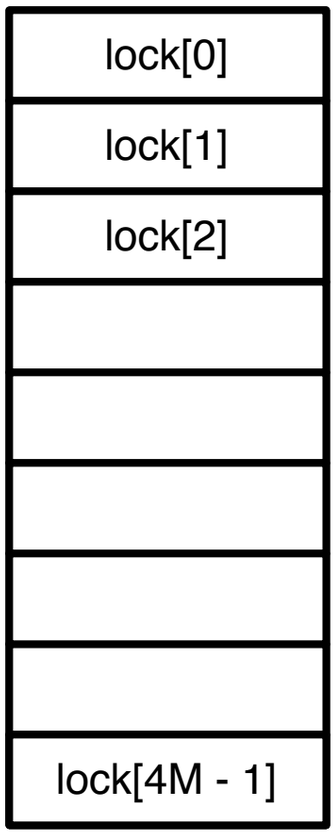


map(0xab000020)

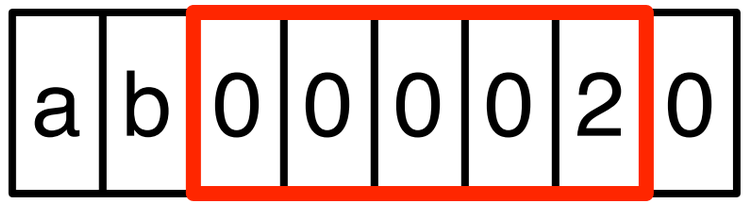


Where is the lock?

Global Lock Table

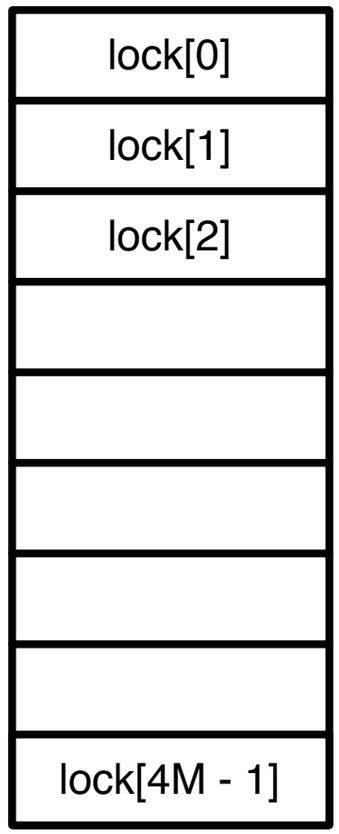


map(0xab000020)

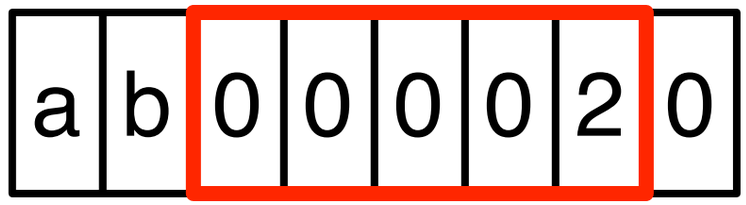


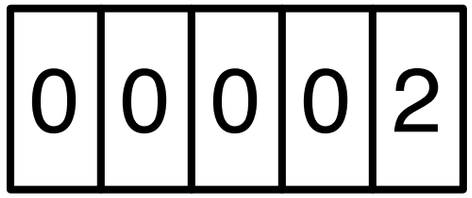
Where is the lock?

Global Lock Table

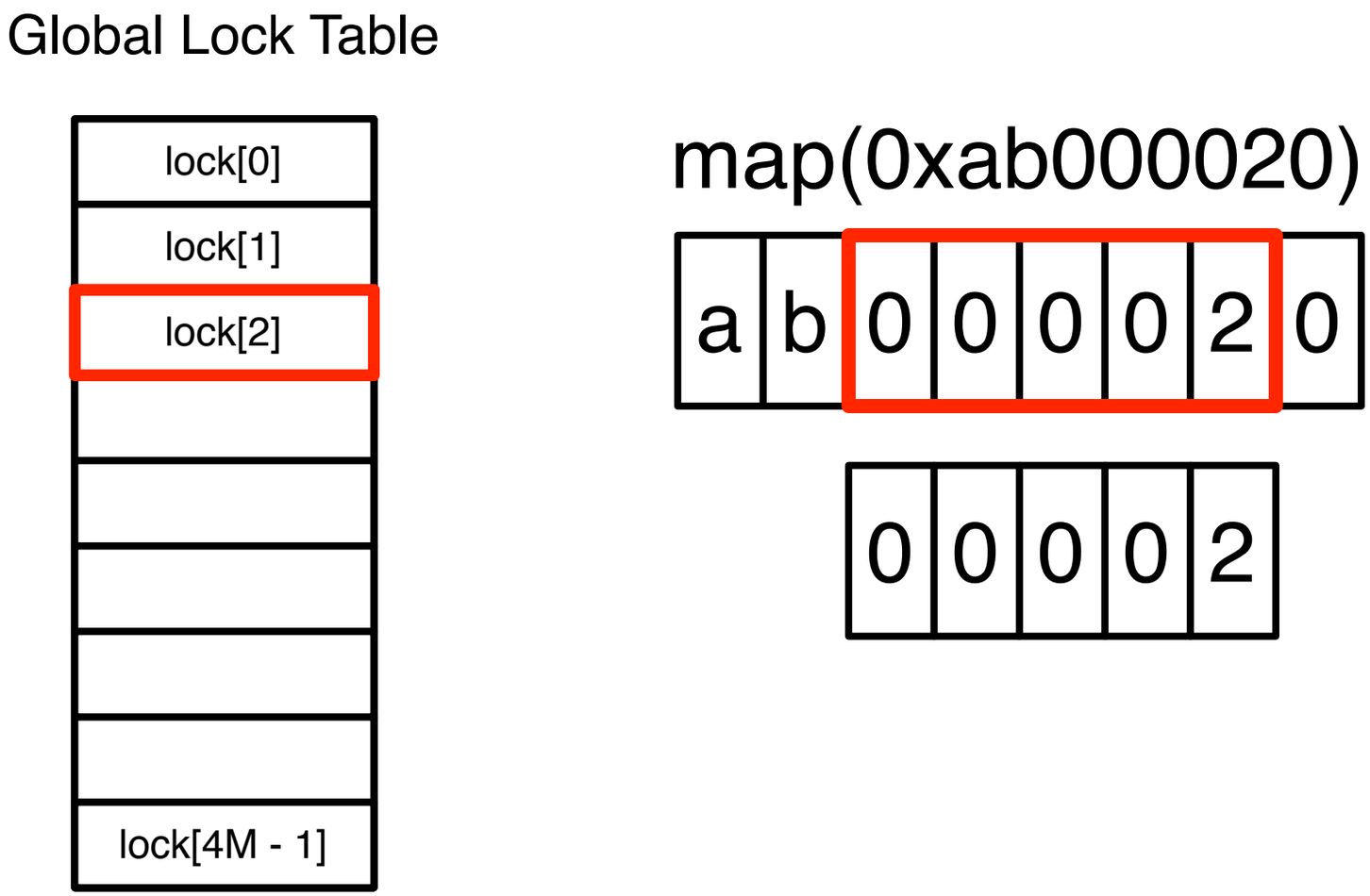


map(0xab000020)



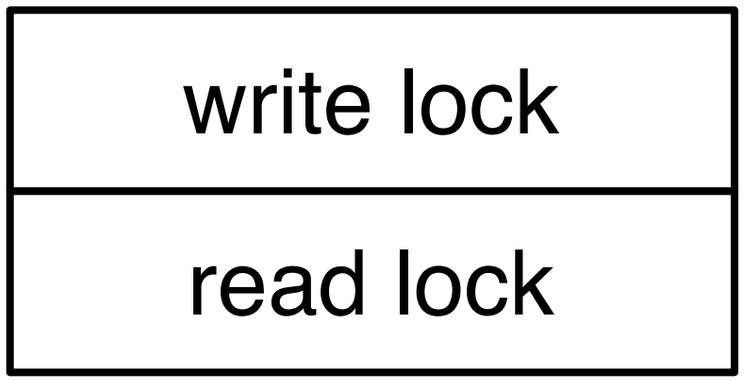


Where is the lock?



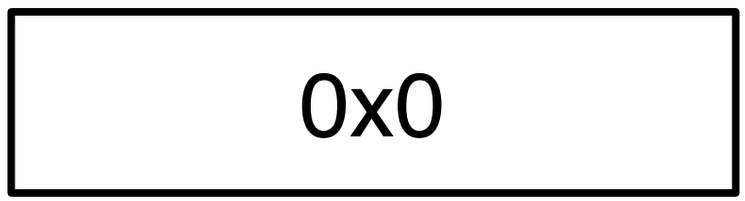
Lock

Lock



Lock

unlocked write lock



Lock

unlocked write lock

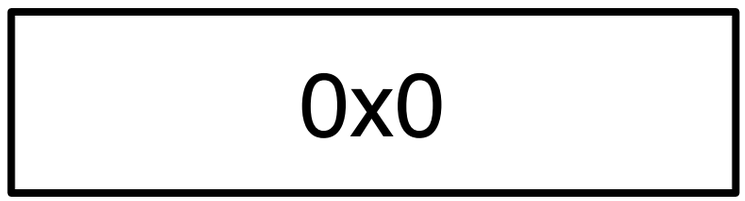
0x0

locked write lock

owner log entry ptr

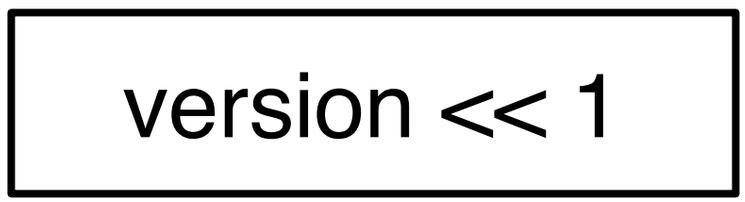
Lock

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)

- Write lock
  - detect write/write conflicts
    - encounter time
- Read lock
  - detect read/write conflicts
  - commit time

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 location version lower than counter ⇒
  no updates to the location since start ⇒
  read set is consistent
- Otherwise, validate
  - remember current version counter

Shared data

```
Commit_ts  // shared version counter
Greedy_ts  // shared CM counter
Lock_table // locks for all locations
```

Thread-local data

```
_start      // rollback jump target
_valid_ts   // read set version
_read_log   // what tx read
_write_log  // what tx wrote
_cm_ts      // CM timestamp
```

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 = 0x1
7:            version := read(r_lock)
8:            continue
9:        value := read(addr)
10:       version2 := read(r_lock)
11:       if version = version2 break
12:       version := version2
13:   read_log_add(r_lock, version)
14:   if version > _valid_ts and not extend()
15:       rollback()
16:   return value
```

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 = 0x1
7:              version := read(r_lock)
8:              continue
9:          value := read(addr)
10:         version2 := read(r_lock)
11:         if version = version2 break
12:         version := version2
13:     read_log_add(r_lock, version)
14:     if version > _valid_ts and not extend()
15:         rollback()
16:     return value
```

Map to lock

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 = 0x1
7:              version := read(r_lock)
8:              continue
9:          value := read(addr)
10:         version2 := read(r_lock)
11:         if version = version2 break
12:         version := version2
13:     read_log_add(r_lock, version)
14:     if version > _valid_ts and not extend()
15:         rollback()
16:     return value
```

Map to lock

Read after write

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 = 0x1
7:             version := read(r_lock)
8:             continue
9:         value := read(addr)
10:        version2 := read(r_lock)
11:        if version = version2 break
12:        version := version2
13:    read_log_add(r_lock, version)
14:    if version > _valid_ts and not extend()
15:        rollback()
16:    return value
```

Map to lock

Read after write

Read consistent version and value

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 = 0x1
7:              version := read(r_lock)
8:              continue
9:          value := read(addr)
10:         version2 := read(r_lock)
11:         if version = version2 break
12:         version := version2
13:     read_log_add(r_lock, version)
14:     if version > _valid_ts and not extend()
15:         rollback()
16:     return value
```

Map to lock

Read after write

Read consistent version and value

Read state consistent?

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) break
12:     write_log_remove(entry)
13:   if read(r_lock) > _valid_ts and not extend()
14:     rollback()
```

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) break
12:     write_log_remove(entry)
13:   if read(r_lock) > _valid_ts and not extend()
14:     rollback()
```

Map to lock

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) break
12:     write_log_remove(entry)
13:   if read(r_lock) > _valid_ts and not extend()
14:     rollback()
```

Map to lock

Write after write

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) break
12:     write_log_remove(entry)
13:   if read(r_lock) > _valid_ts and not extend()
14:     rollback()
```

Map to lock

Write after write

Acquire write lock

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) break
12:     write_log_remove(entry)
13:   if read(r_lock) > _valid_ts and not extend()
14:     rollback()
```

Map to lock

Write after write

Acquire write lock

Validate (for WaR)

Commit

```
1:  tx_commit()
2:    if is_empty(_write_log) return
3:    for entry in _write_log
4:      write(entry.r_lock,0x1)
5:    ts := increment(Commits ts)
6:    if ts > _valid_ts + 1 and not validate()
7:      for entry in _write_log
8:        write(entry.r_lock, entry.version)
9:      rollback()
10:   for entry in _write_log
11:     write(entry.addr, entry.value)
12:     write(entry.r_lock, ts << 1)
13:     write(entry.w_lock, 0)
```

Commit

```
1:  tx_commit()
2:    if is_empty(_write_log) return
3:    for entry in _write_log
4:      write(entry.r_lock,0x1)
5:    ts := increment(Commits ts)
6:    if ts > _valid_ts + 1 and not validate()
7:      for entry in _write_log
8:        write(entry.r_lock, entry.version)
9:      rollback()
10:   for entry in _write_log
11:     write(entry.addr, entry.value)
12:     write(entry.r_lock, ts << 1)
13:     write(entry.w_lock, 0)
```

Read-only

Commit

```
1:  tx_commit()
2:    if is_empty(_write_log) return
3:    for entry in _write_log
4:      write(entry.r_lock,0x1)
5:    ts := increment(Commits ts)
6:    if ts > _valid_ts + 1 and not validate()
7:      for entry in _write_log
8:        write(entry.r_lock, entry.version)
9:      rollback()
10:   for entry in _write_log
11:     write(entry.addr, entry.value)
12:     write(entry.r_lock, ts << 1)
13:     write(entry.w_lock, 0)
```

Read-only

Acquire read locks

Commit

```
1:  tx_commit()
2:    if is_empty(_write_log) return
3:    for entry in _write_log
4:      write(entry.r_lock,0x1)
5:    ts := increment(Commits ts)
6:    if ts > _valid_ts + 1 and not validate()
7:      for entry in _write_log
8:        write(entry.r_lock, entry.version)
9:      rollback()
10:   for entry in _write_log
11:     write(entry.addr, entry.value)
12:     write(entry.r_lock, ts << 1)
13:     write(entry.w_lock, 0)
```

Read-only

Acquire read locks

Get next version

Commit

```
 1: tx_commit()
 2:   if is_empty(_write_log) return
 3:   for entry in _write_log
 4:     write(entry.r_lock,0x1)
 5:   ts := increment(Commits ts)
 6:   if ts > _valid_ts + 1 and not validate()
 7:     for entry in _write_log
 8:       write(entry.r_lock, entry.version)
 9:     rollback()
10:   for entry in _write_log
11:     write(entry.addr, entry.value)
12:     write(entry.r_lock, ts << 1)
13:     write(entry.w_lock, 0)
```

Read-only

Acquire read locks

Get next version

Validate read set

Commit

```
1:  tx_commit()
2:    if is_empty(_write_log) return
3:    for entry in _write_log
4:      write(entry.r_lock,0x1)
5:    ts := increment(Commits ts)
6:    if ts > _valid_ts + 1 and not validate()
7:      for entry in _write_log
8:        write(entry.r_lock, entry.version)
9:      rollback()
10:   for entry in _write_log
11:     write(entry.addr, entry.value)
12:     write(entry.r_lock, ts << 1)
13:     write(entry.w_lock, 0)
```

Read-only

Acquire read locks

Get next version

Validate read set

Commit values to memory

Rollback

```
1: rollback()
2:    for entry in _write_log
3:      write(entry.w_lock, 0x0)
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
```

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
```

Start as Timid

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
```

Start as Timid

Switch to Greedy

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
```

Start as Timid

Switch to Greedy

Random backoff

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
```

Start as Timid

Switch to Greedy

Random backoff

Timid

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
```

Start as Timid

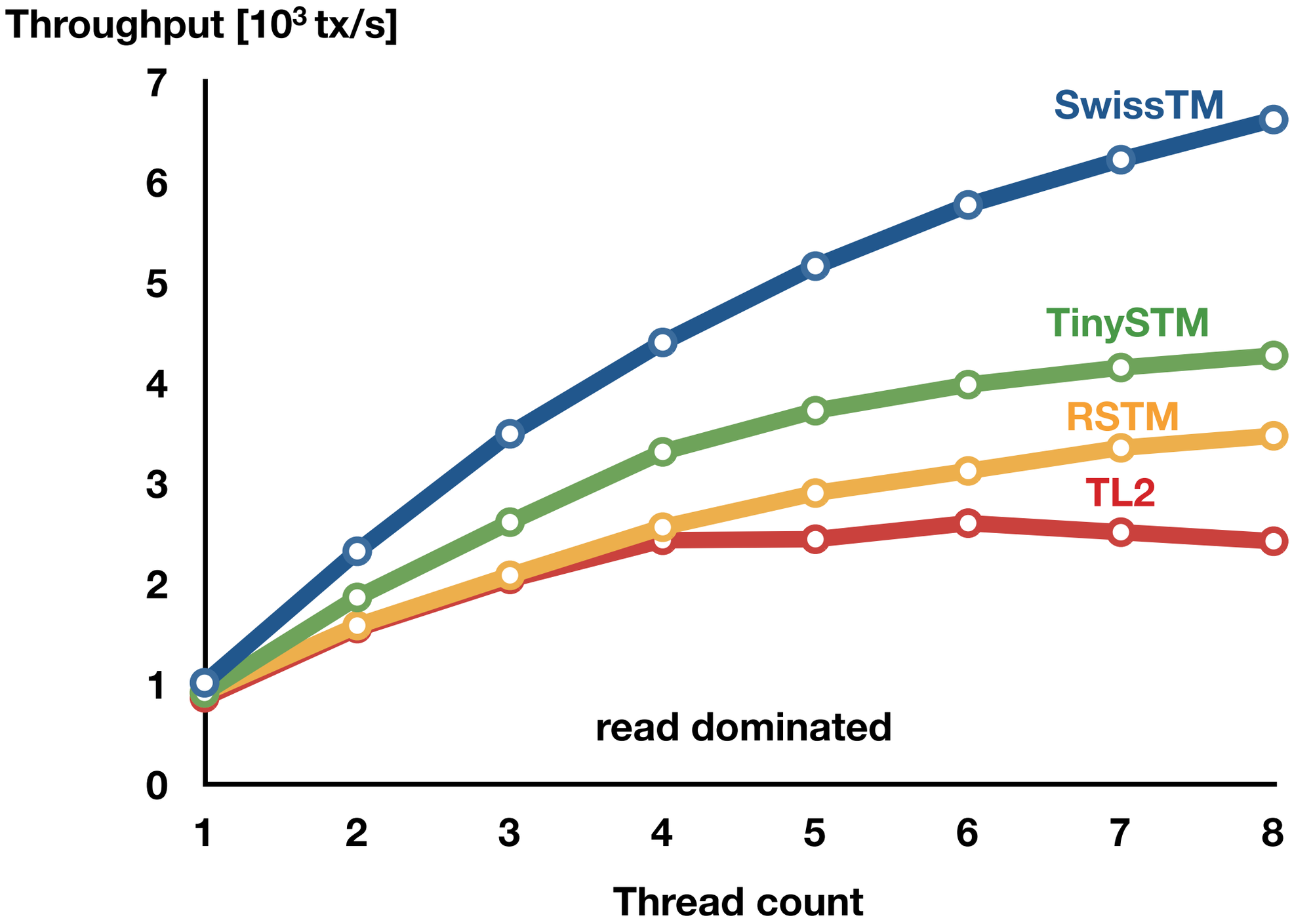
Switch to Greedy

Random backoff

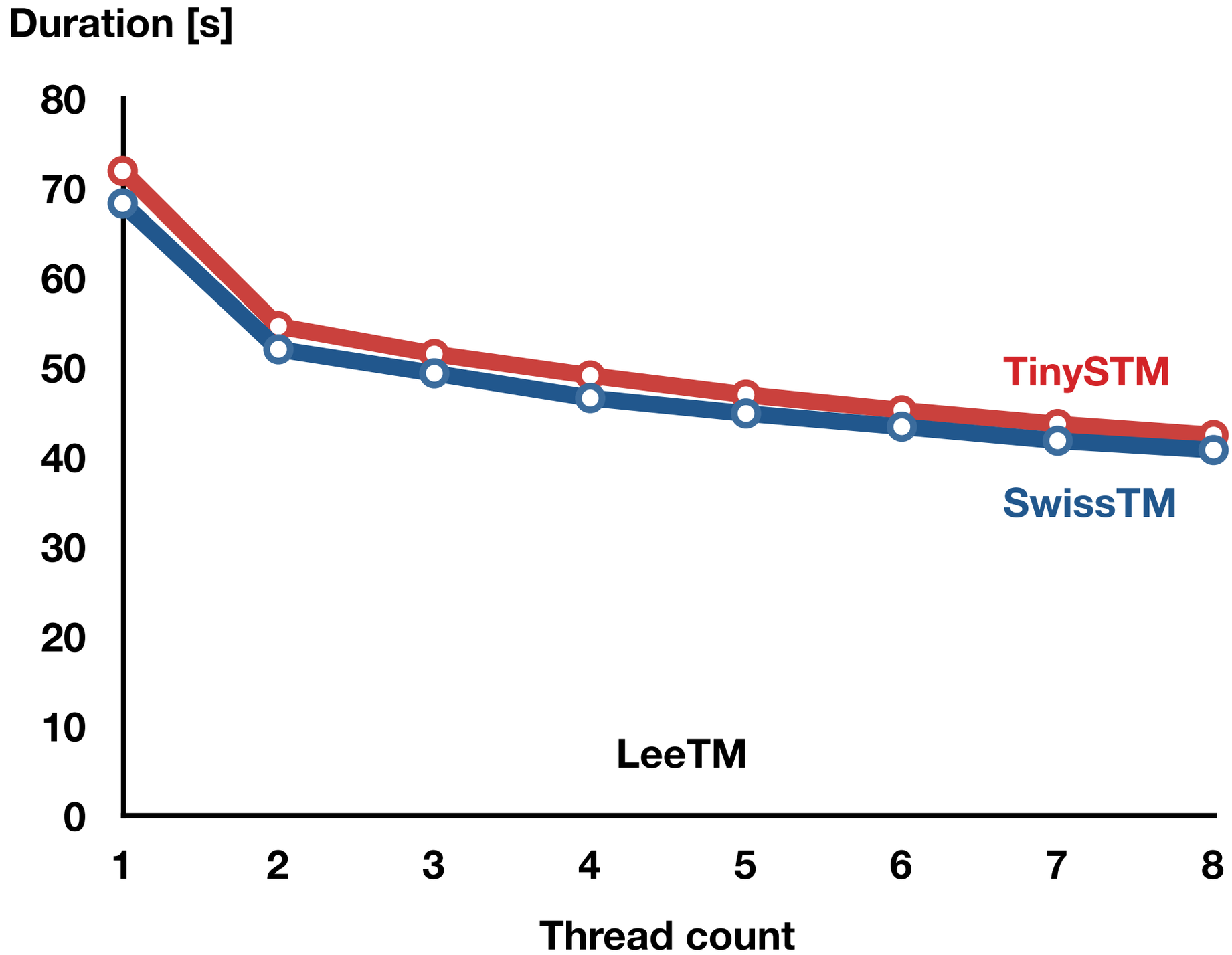
Timid

Greedy

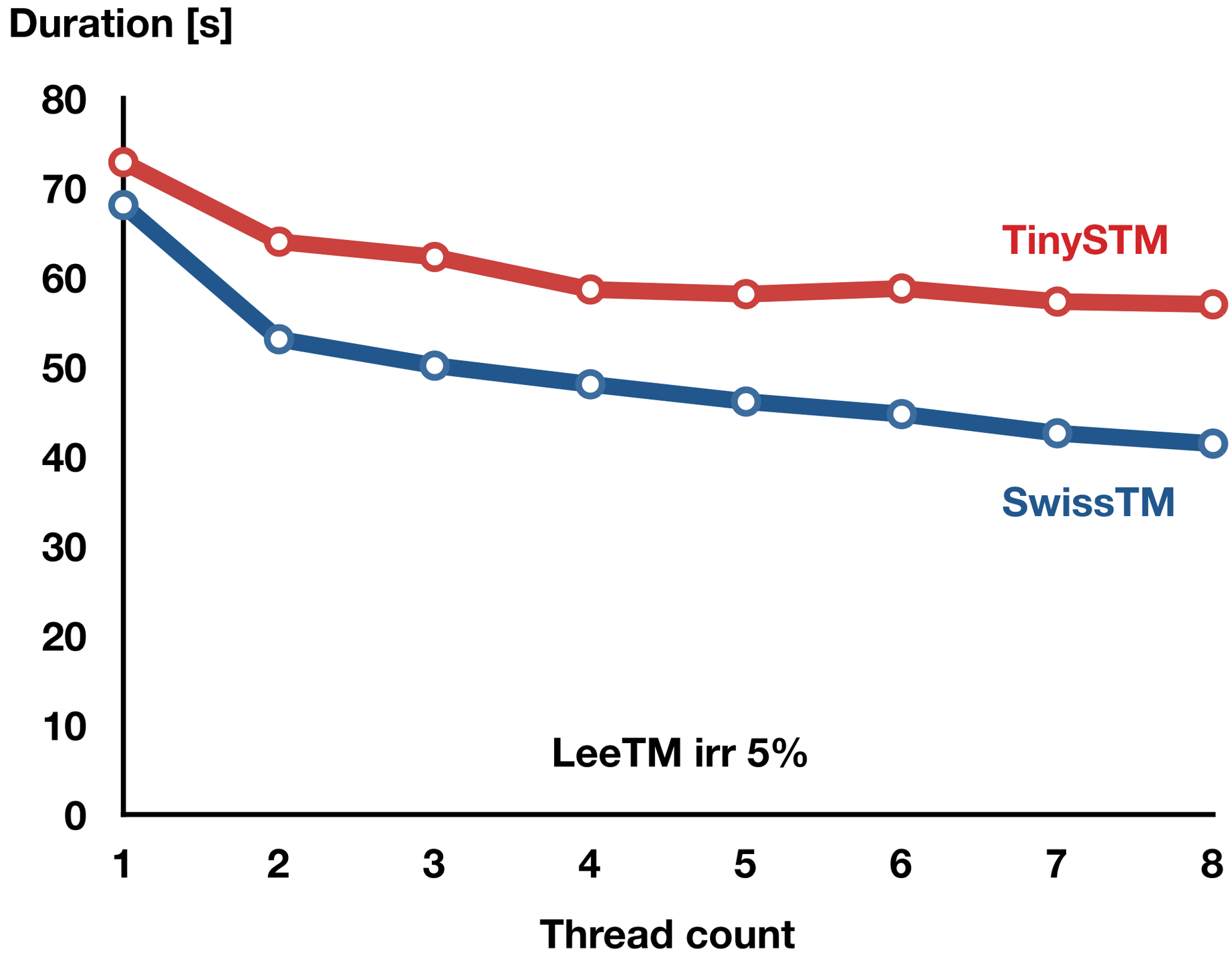
STMBench7


Throughput [10³ tx/s]
7
6
5
4
3
2
1
0
SwissTM
TinySTM
RSTM
TL2
read dominated
1
2
3
4
5
6
7
8
Thread count

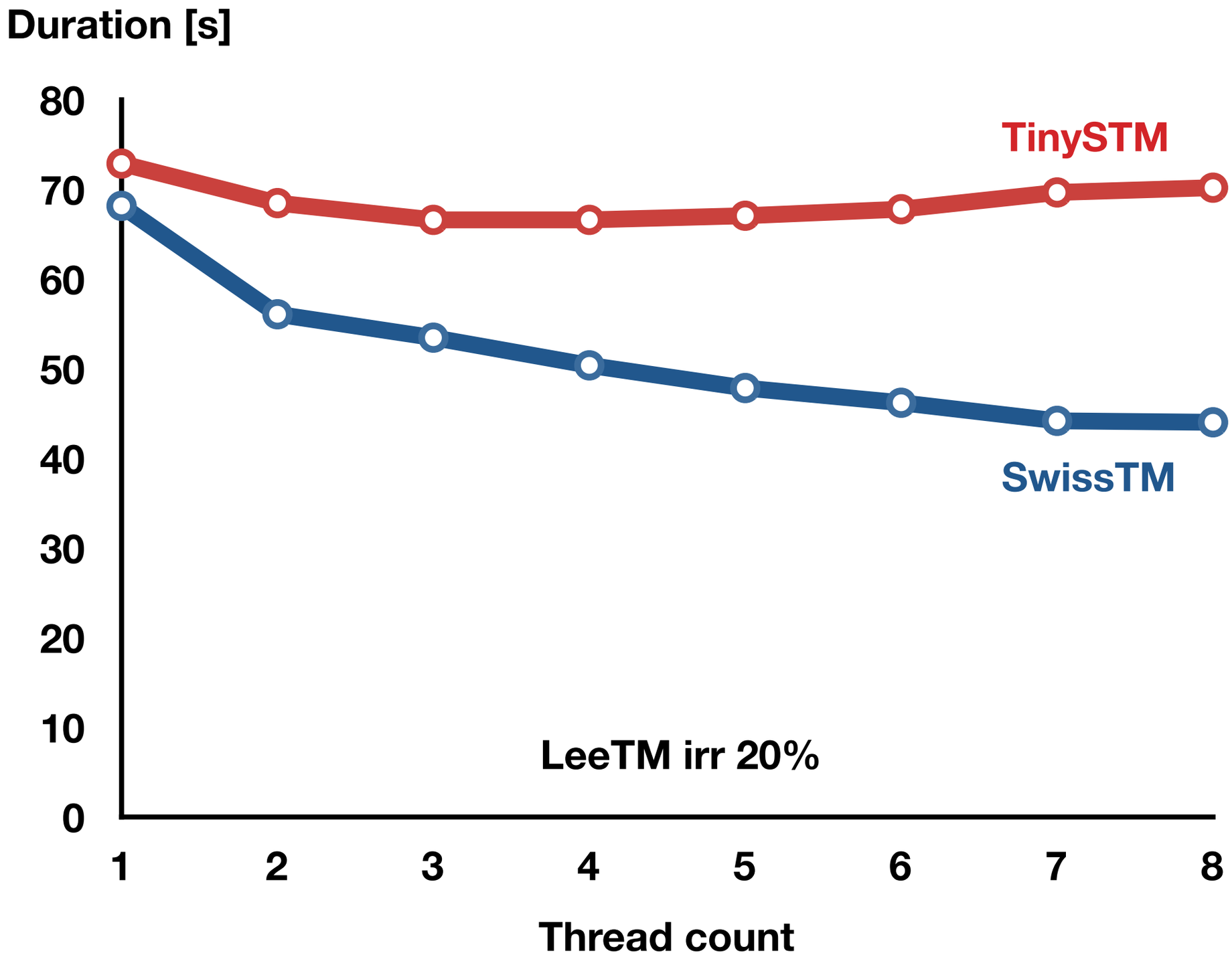
Mixed invalidation



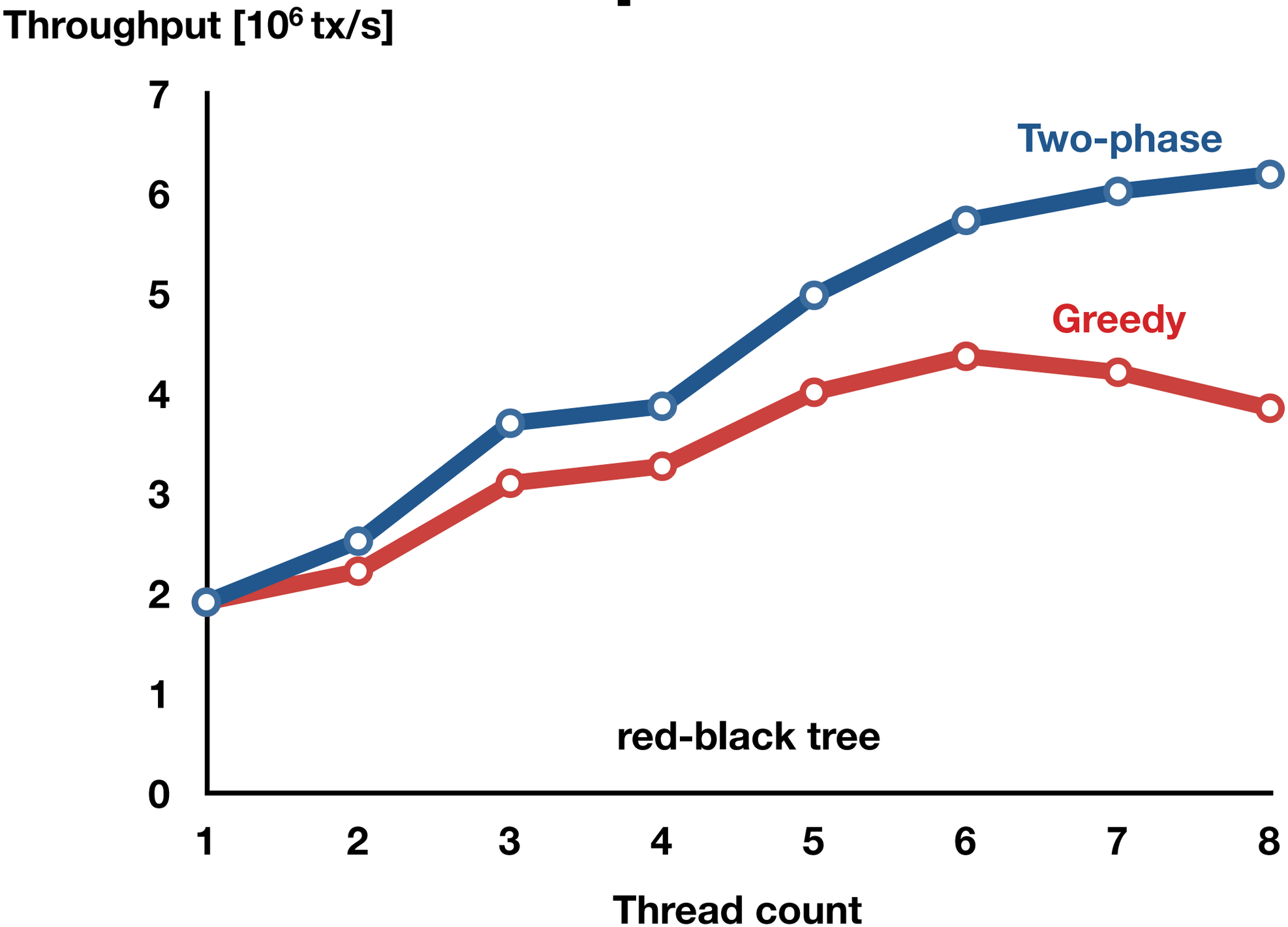
Mixed invalidation



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

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

STM performance

STM performance

How fast is an STM really?

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 the task more times

Approach III

Avoiding pitfalls

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

STMBench7

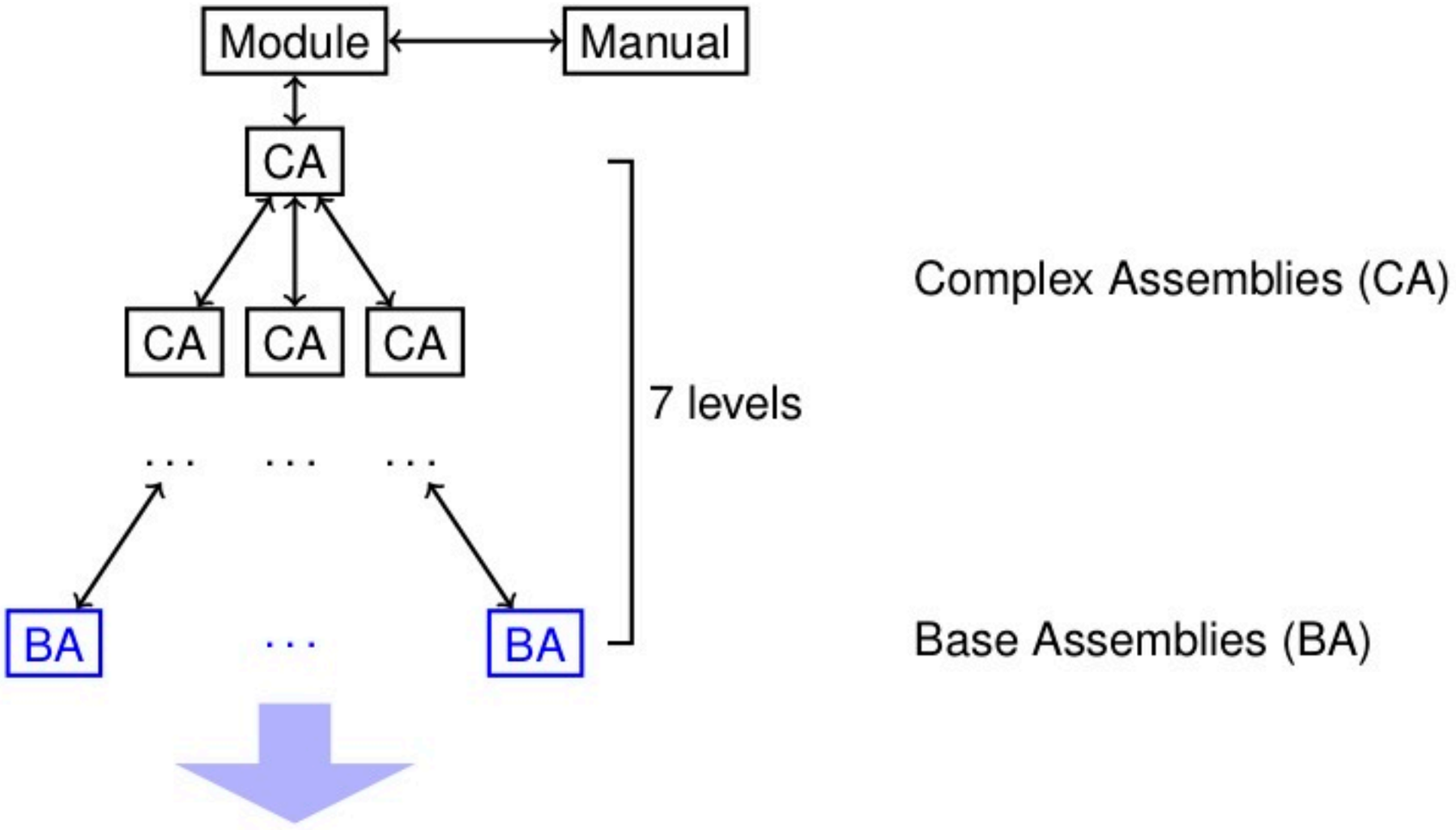
STMBench7

What does a benchmark look like?

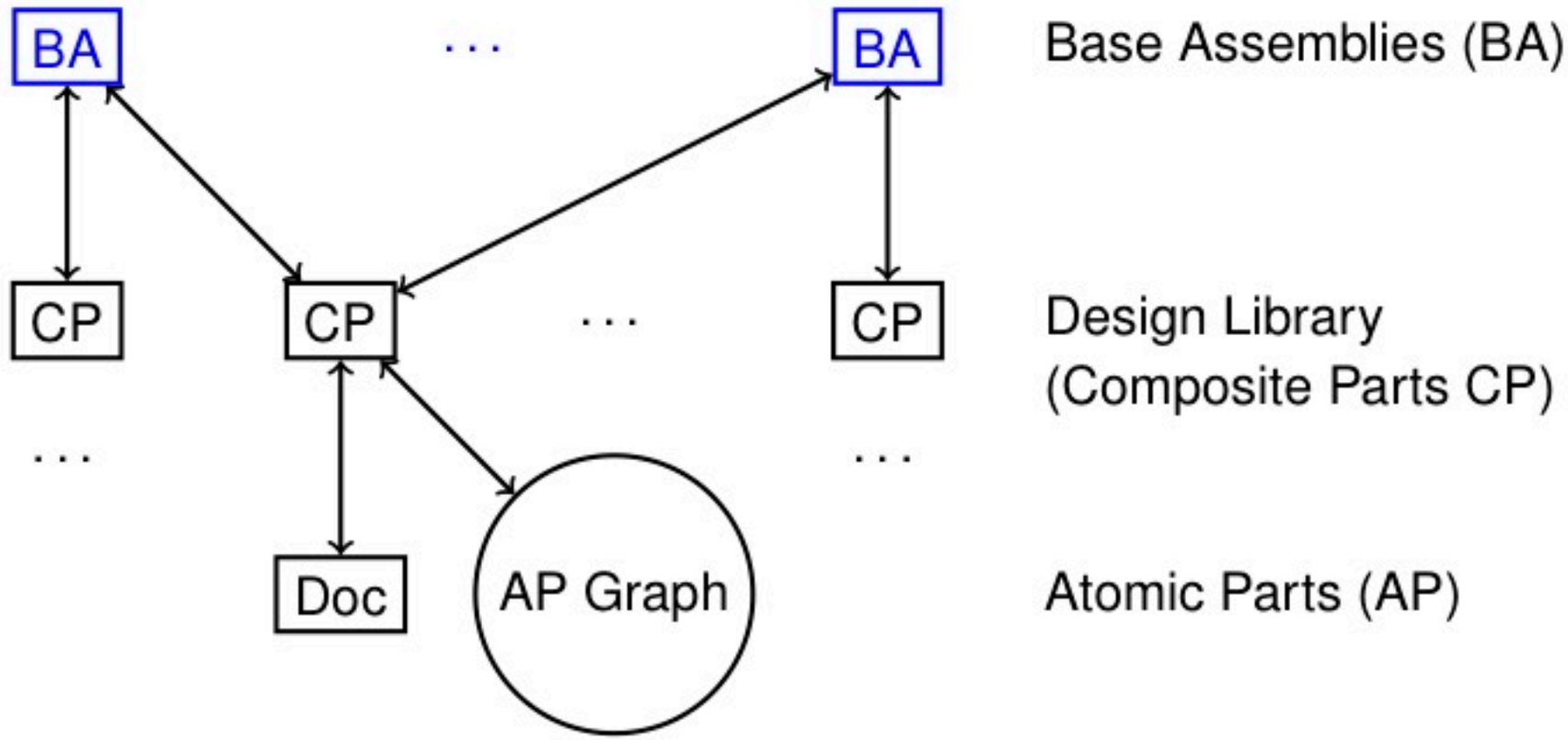
STMBench7

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

Data structure



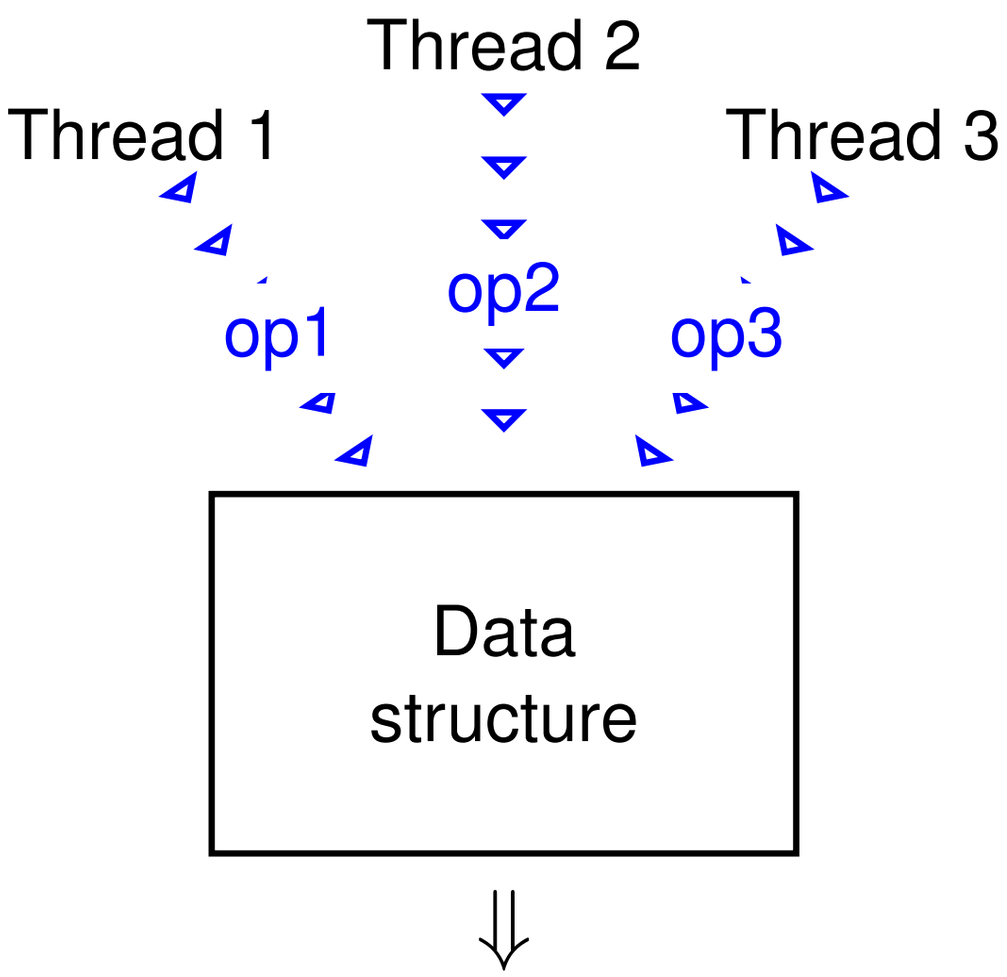
Data structure (2)



Operations

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

Execution



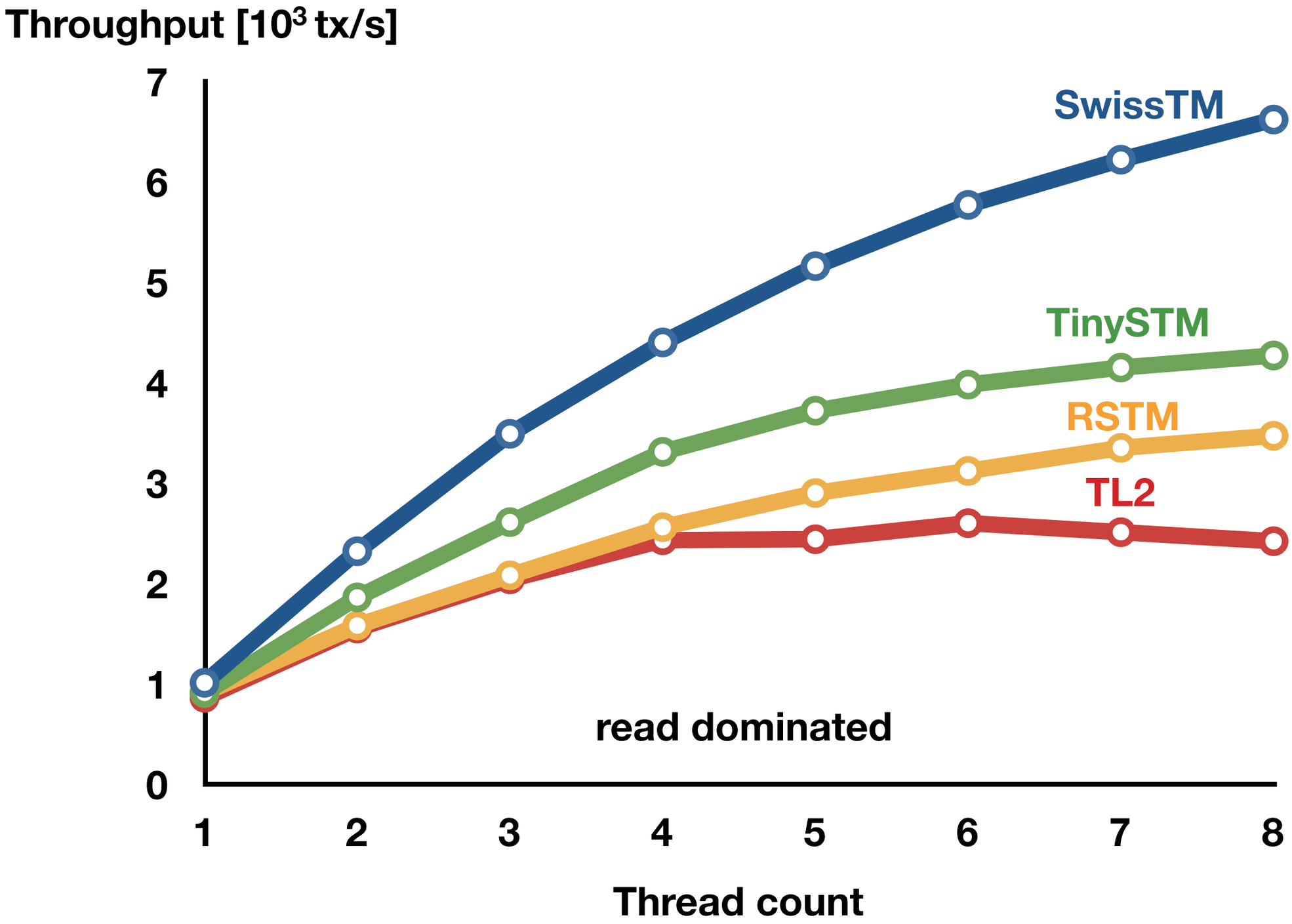
⇓

throughput

Execution (2)

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

STMBench7


Throughput [10³ tx/s]
7
6
5
4
3
2
1
0
SwissTM
TinySTM
RSTM
TL2
read dominated
1
2
3
4
5
6
7
8
Thread count

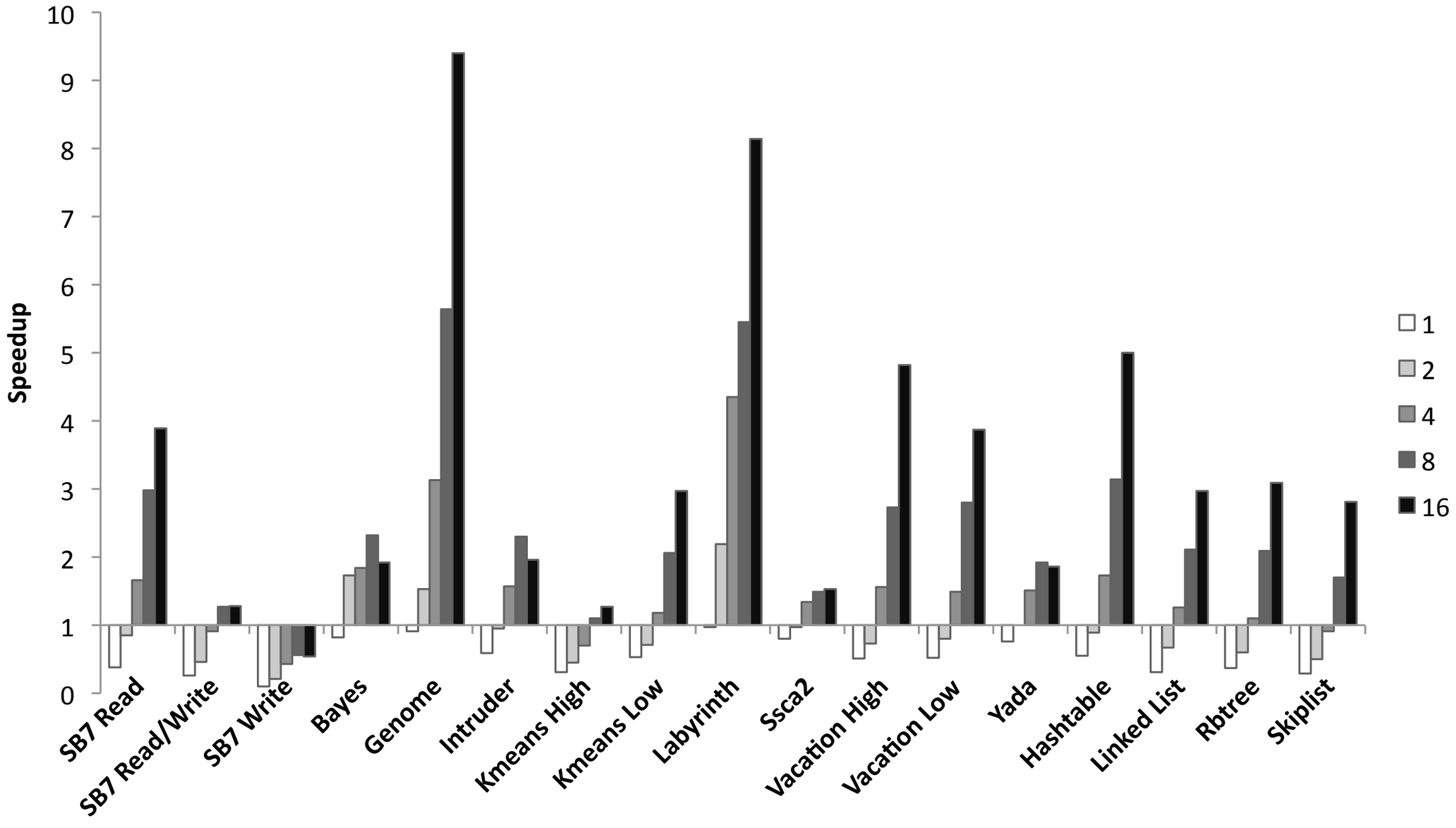
Important question

Important question

Can STM outperform sequential code?

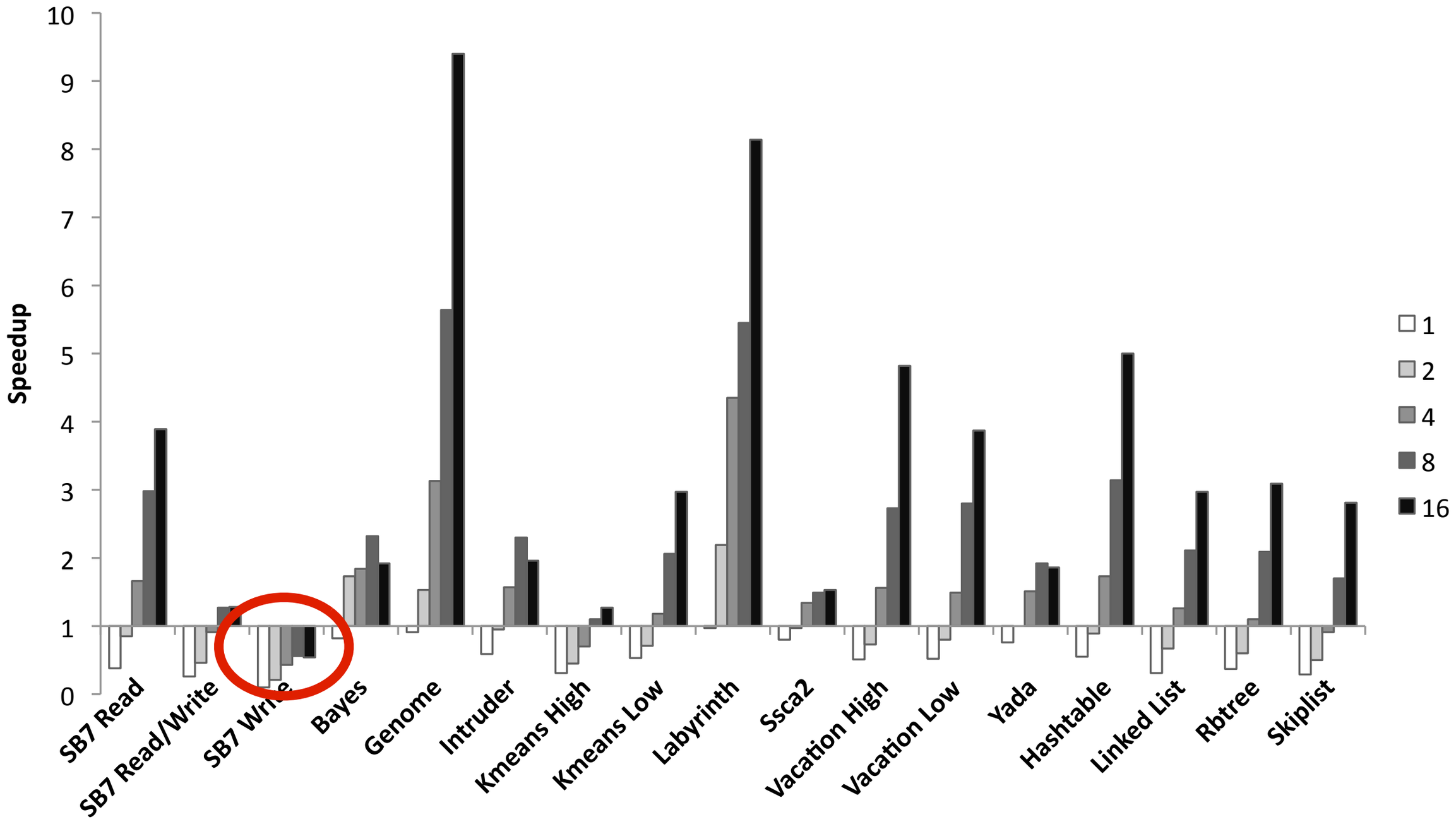
SwissTM vs Sequential

(x86 manual)



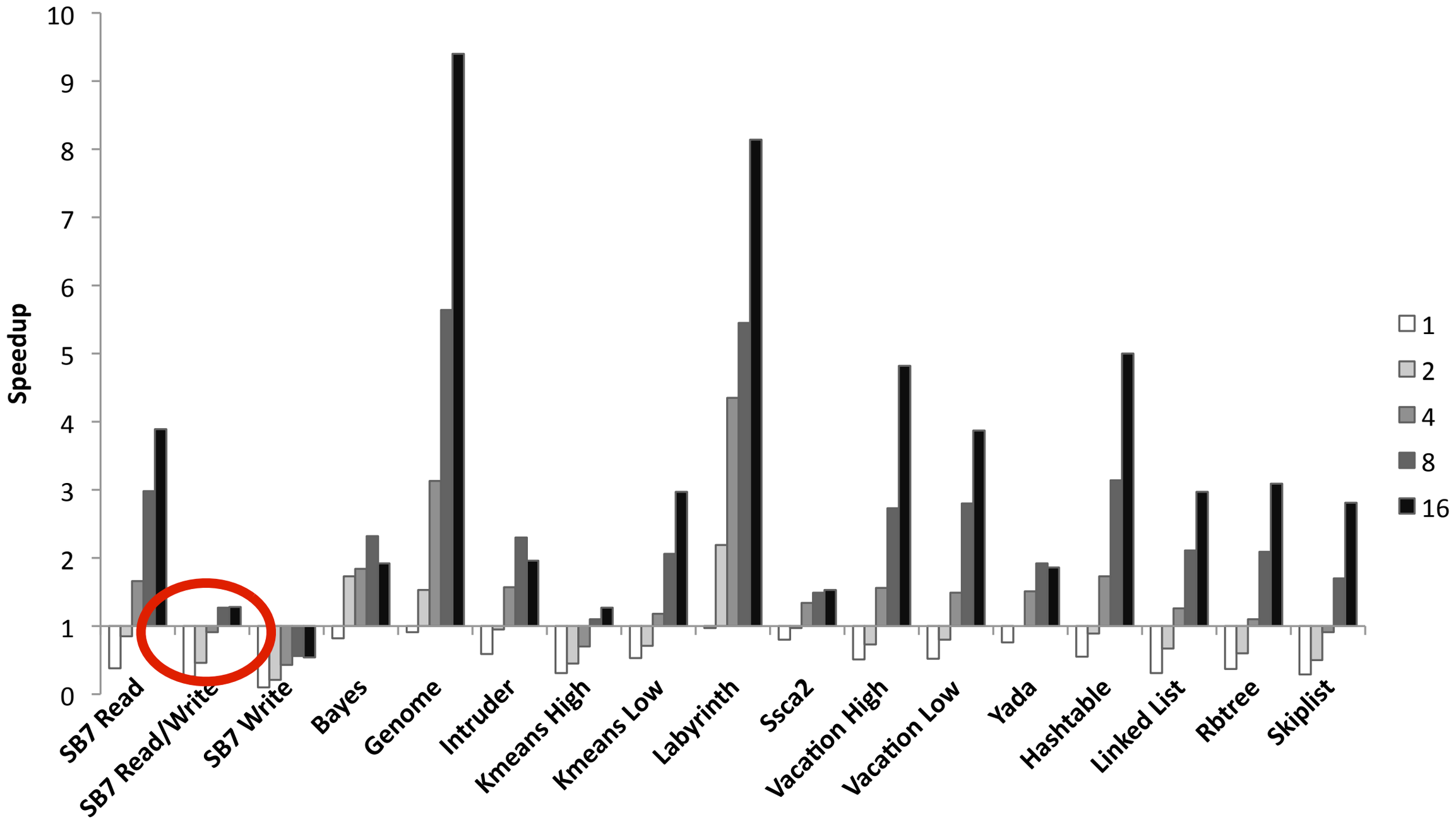
SwissTM vs Sequential

(x86 manual)



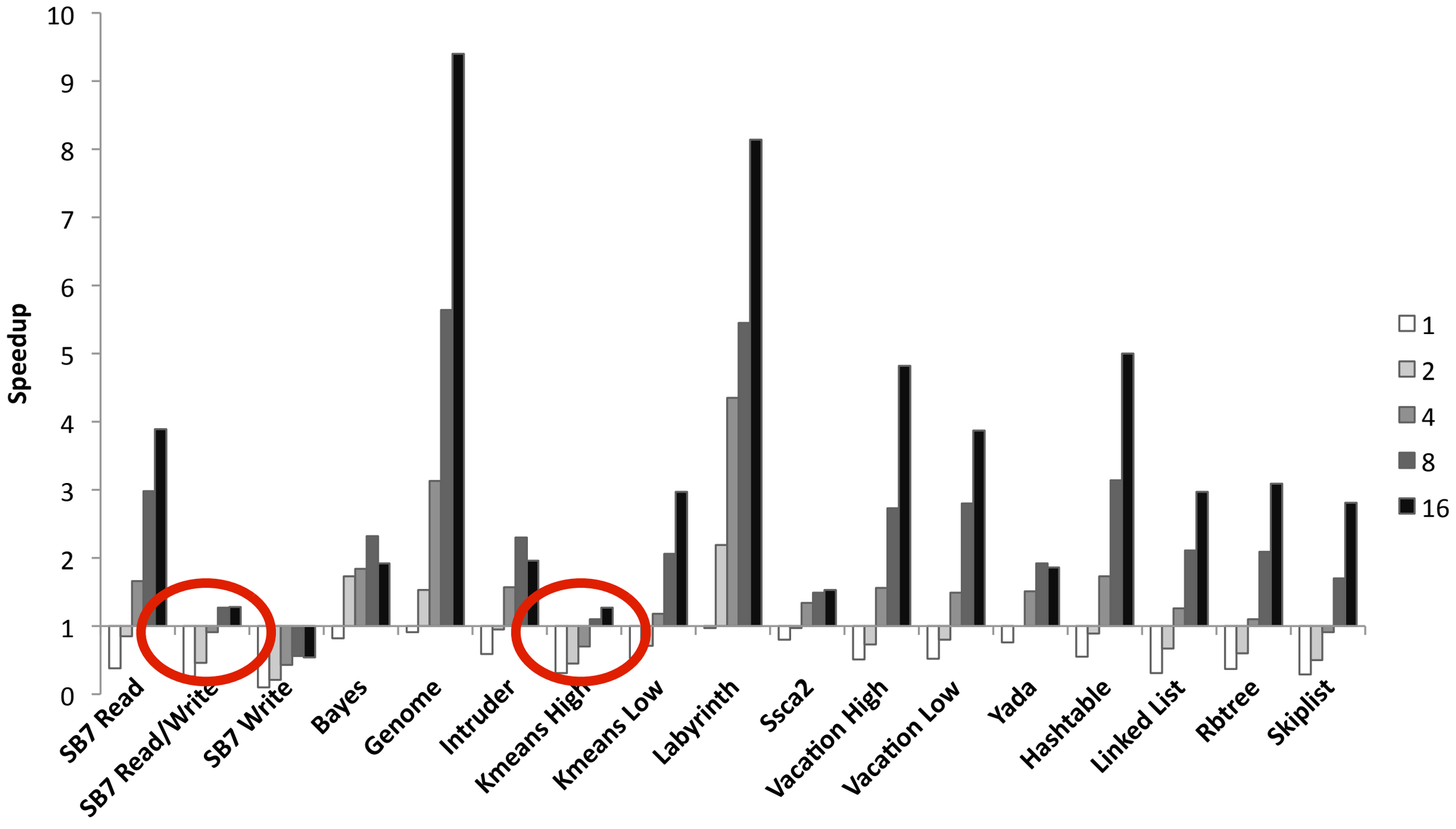
SwissTM vs Sequential

(x86 manual)



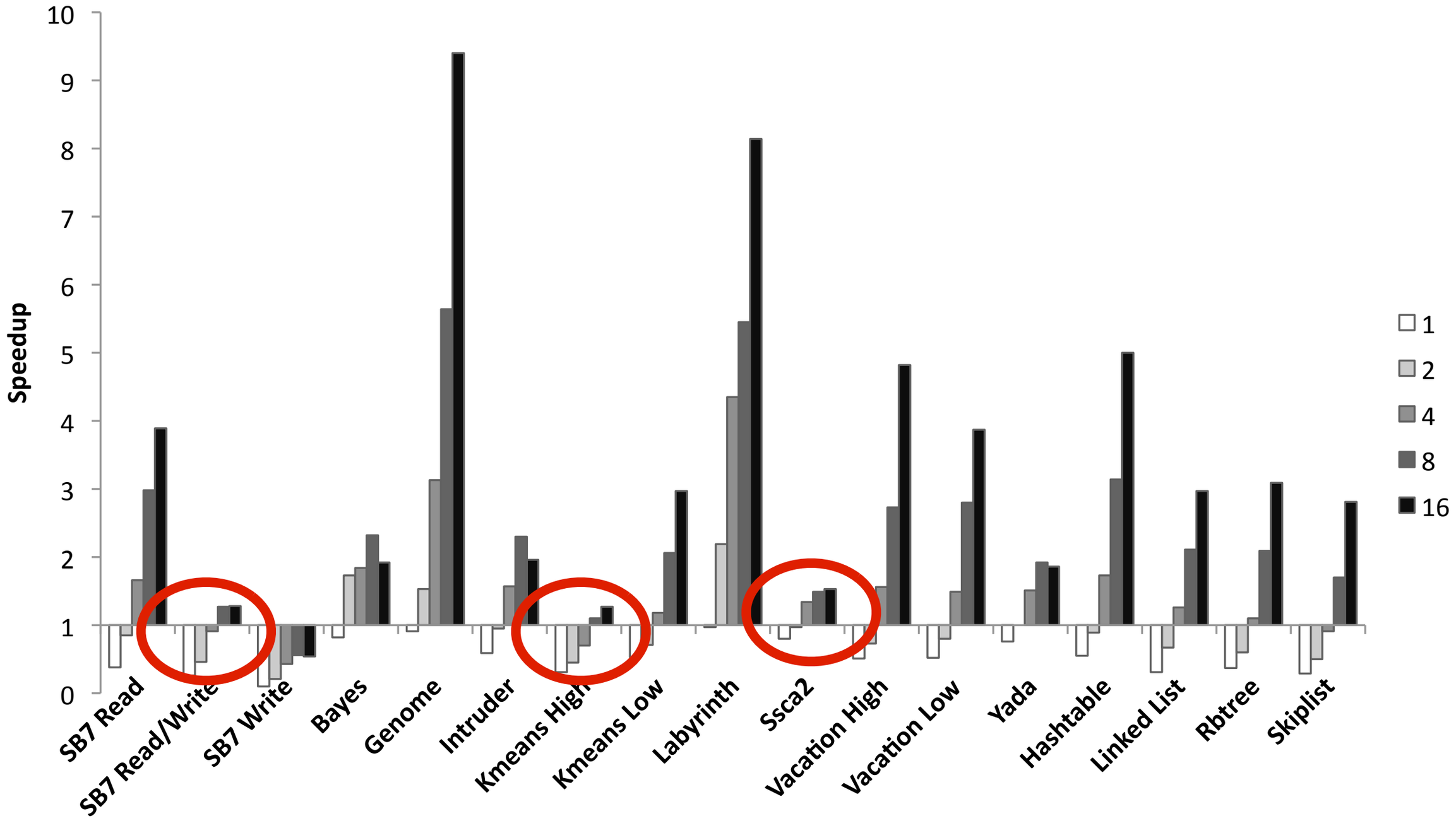
SwissTM vs Sequential

(x86 manual)



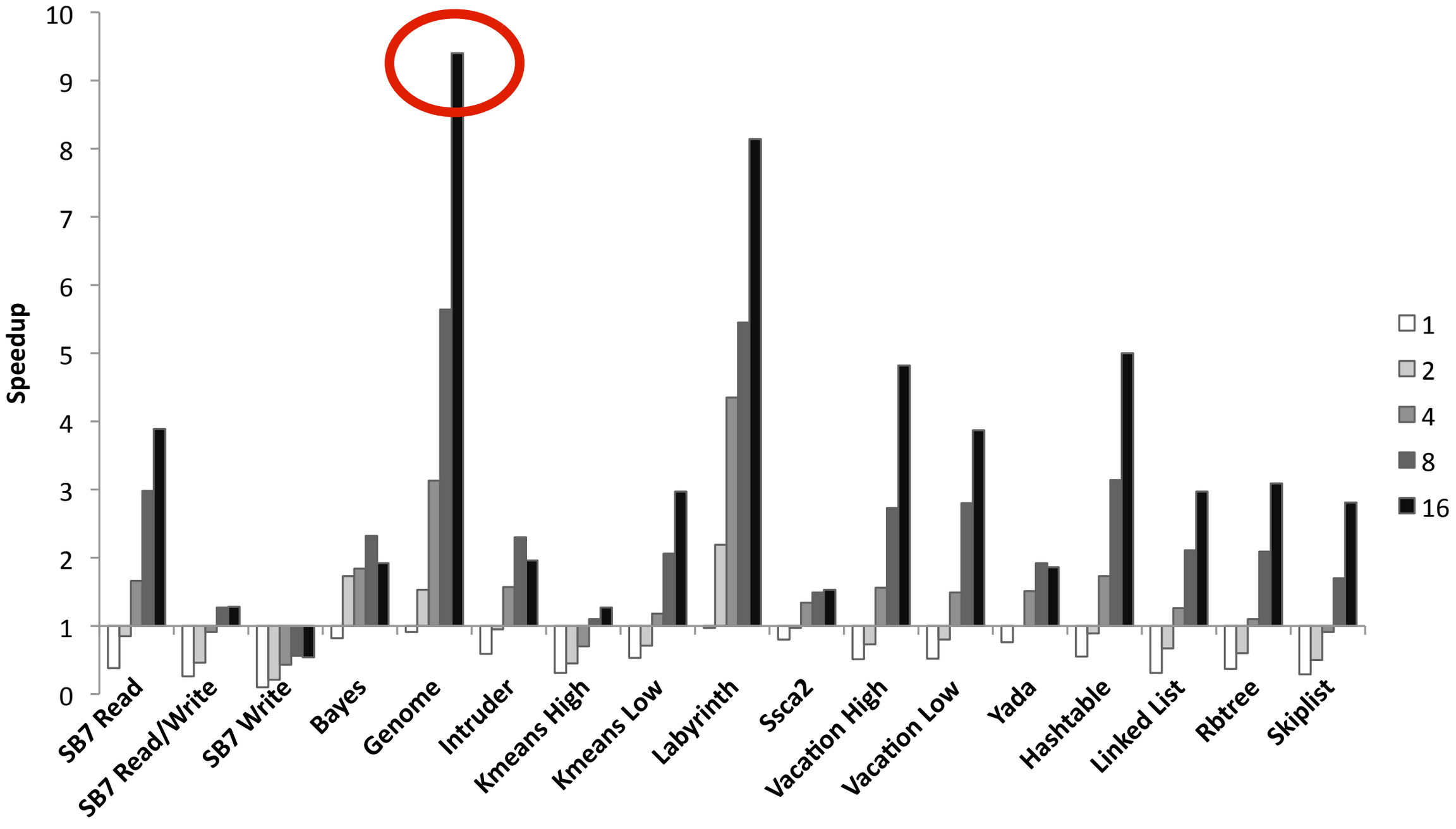
SwissTM vs Sequential

(x86 manual)



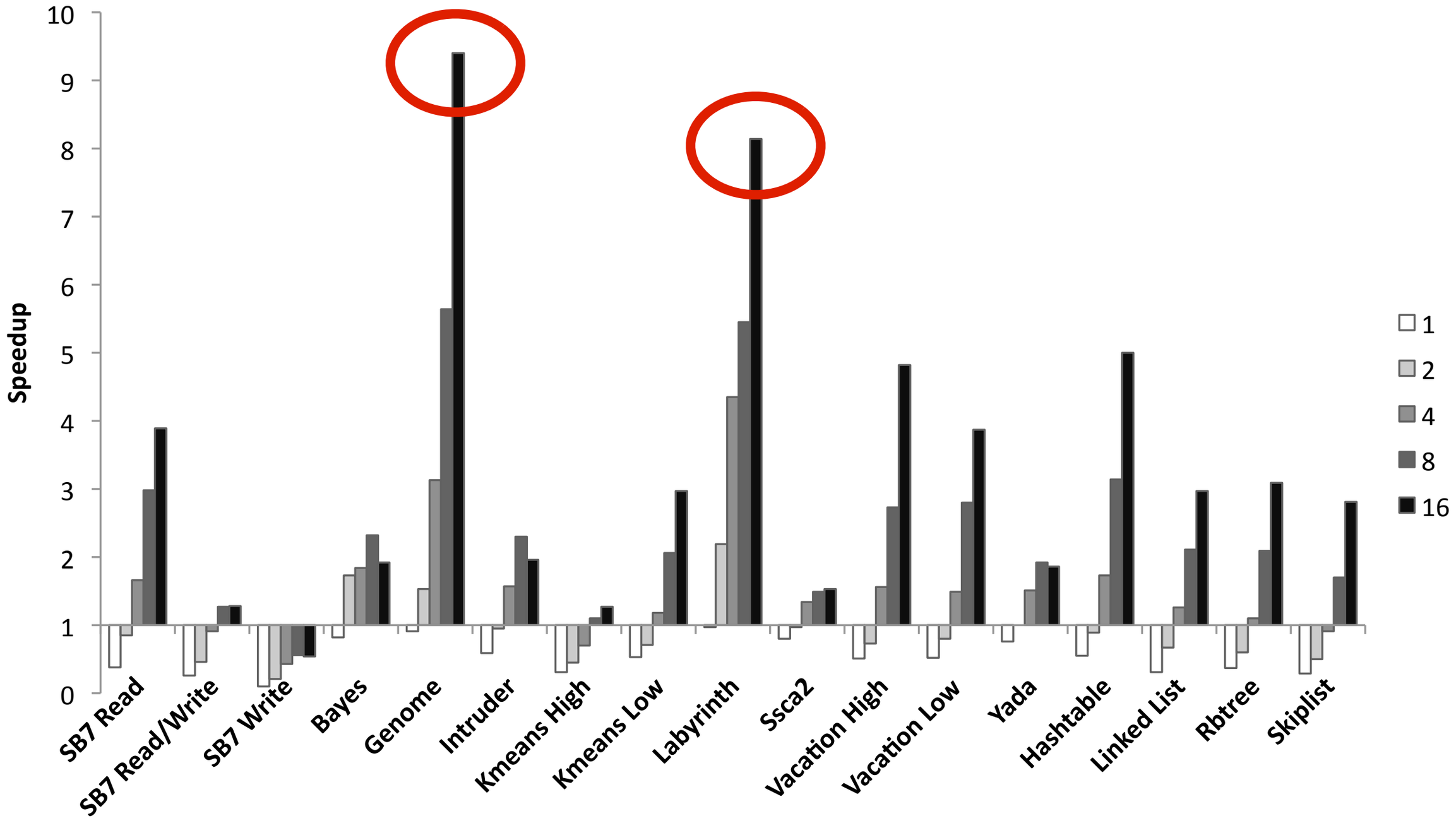
SwissTM vs Sequential

(x86 manual)



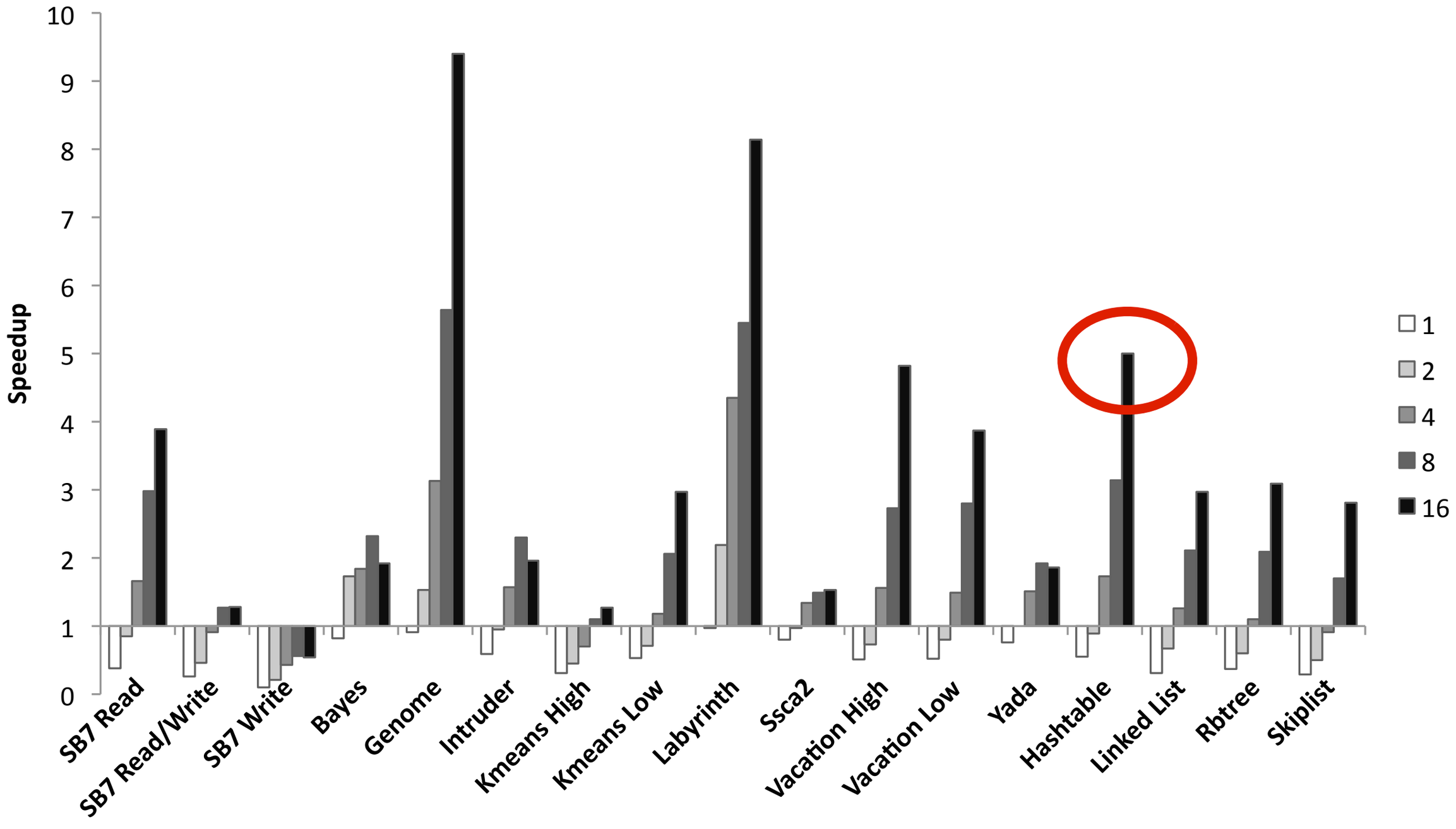
SwissTM vs Sequential

(x86 manual)



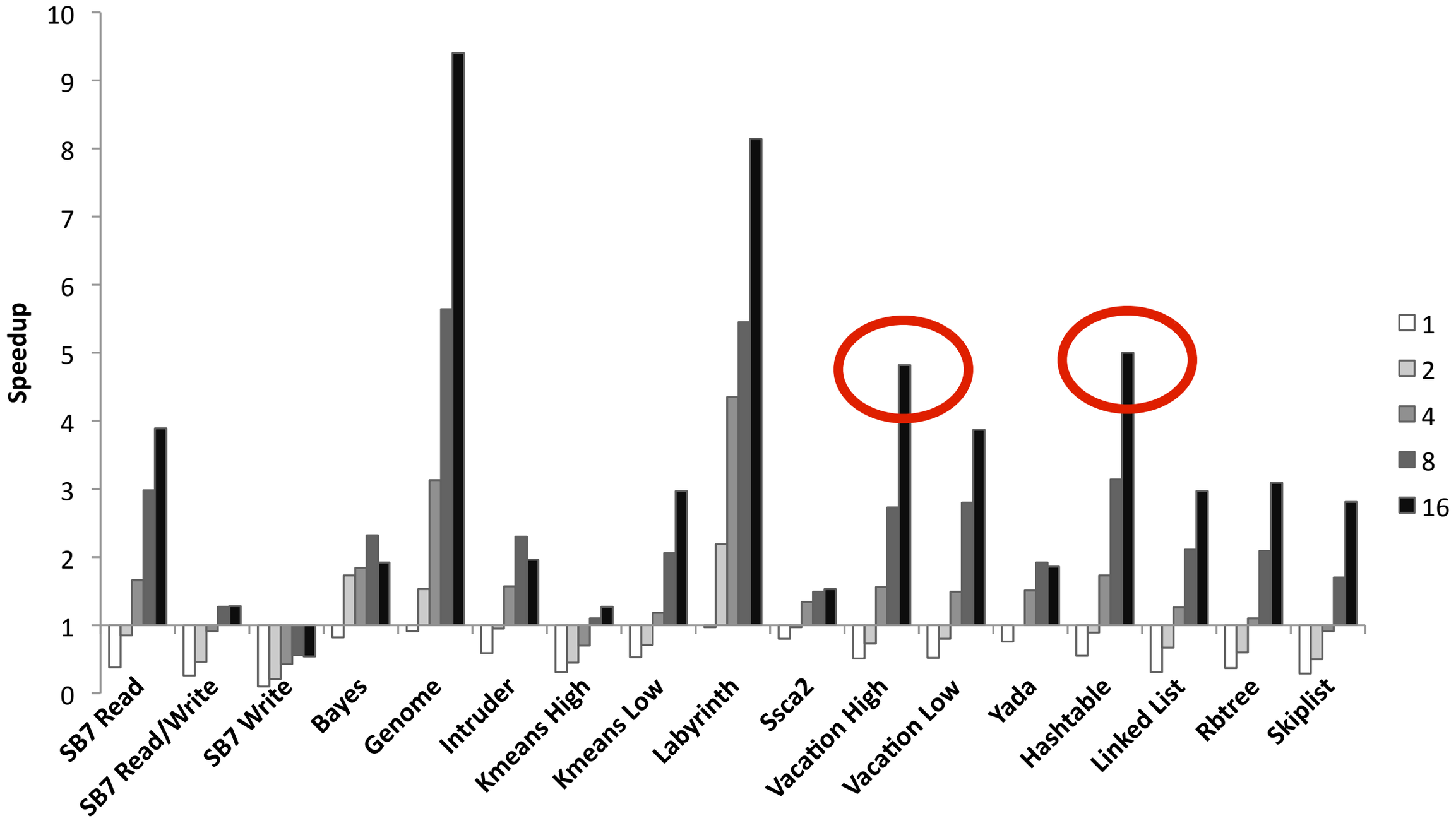
SwissTM vs Sequential

(x86 manual)



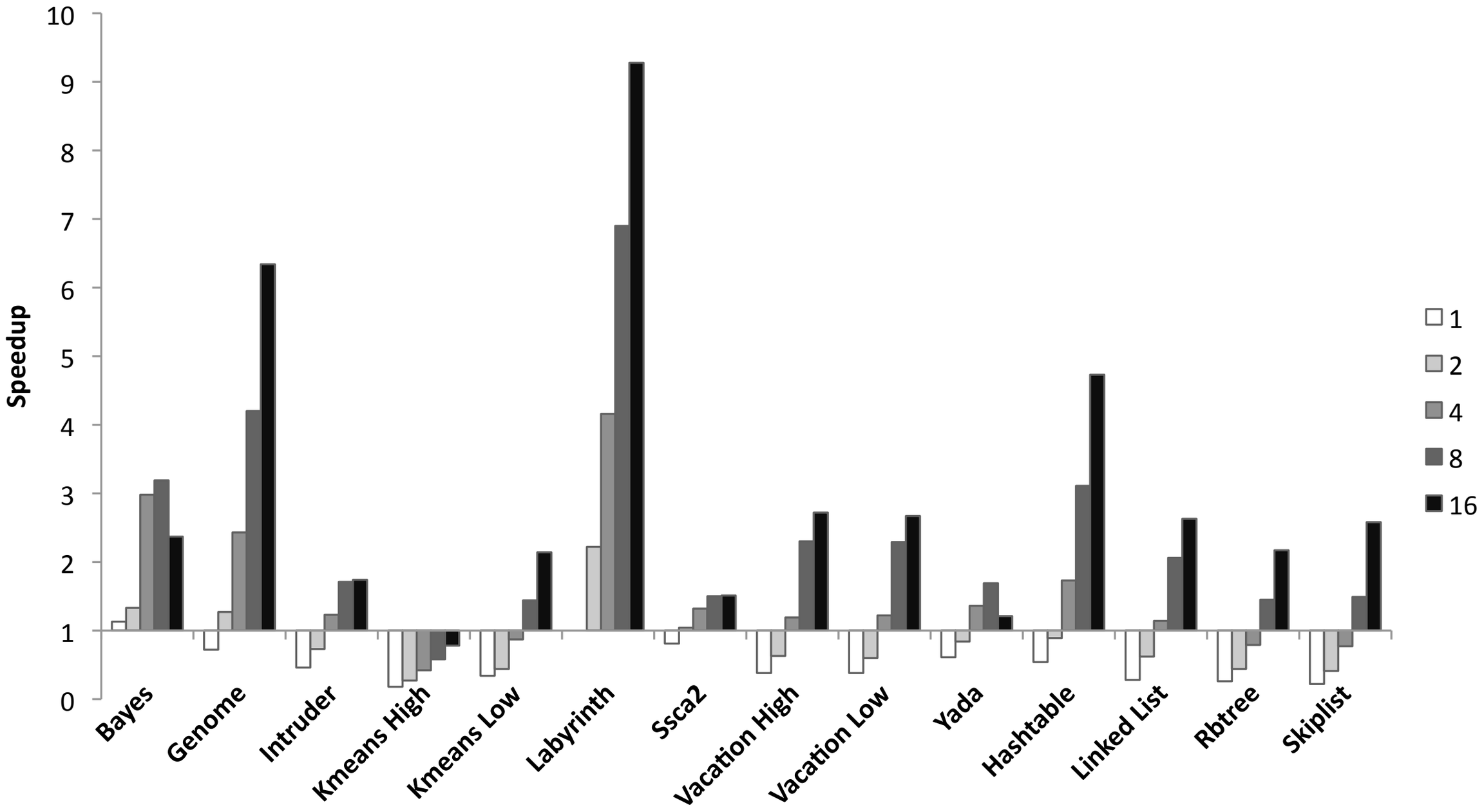
SwissTM vs Sequential

(x86 manual)



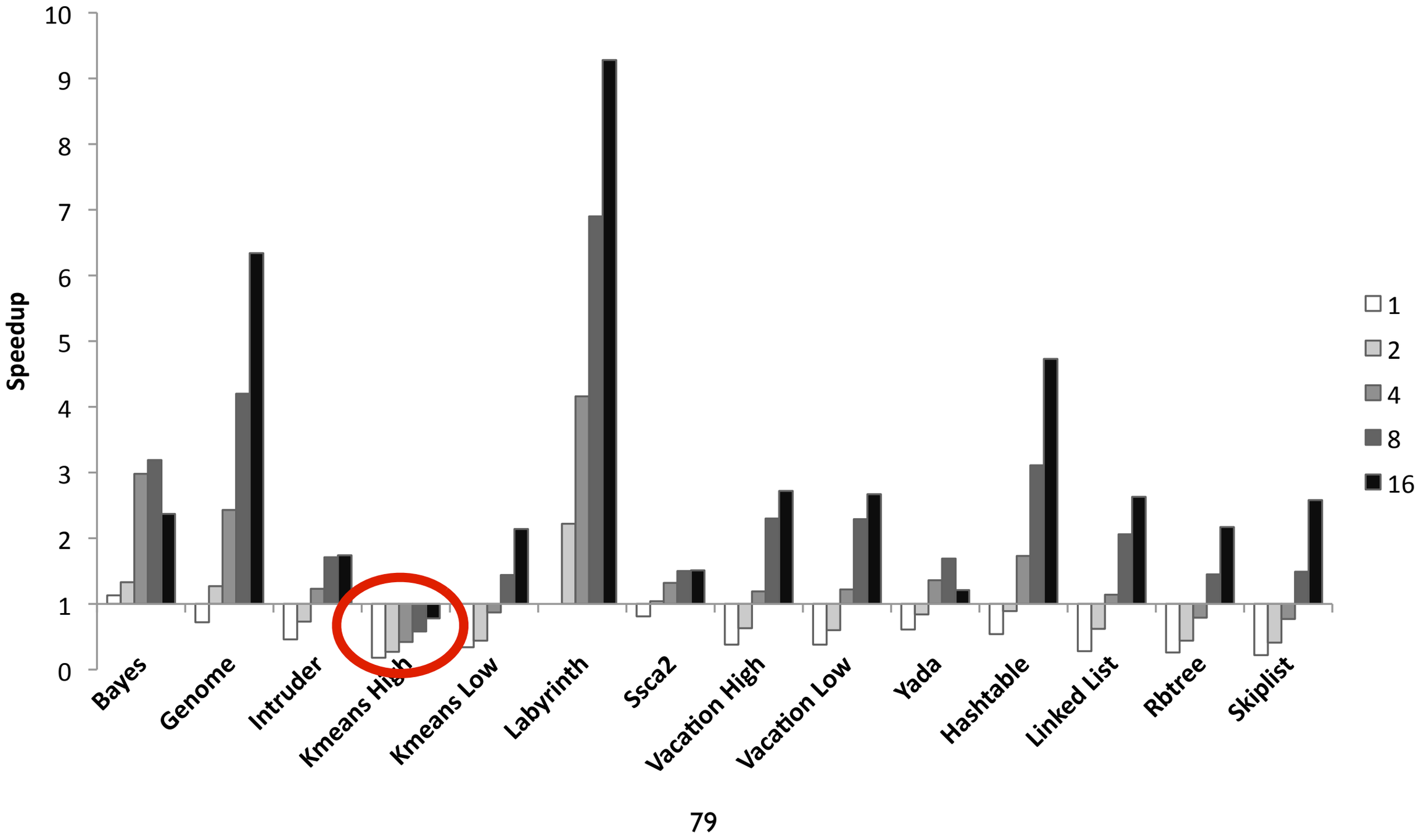
SwissTM vs Sequential

(x86 Intel compiler)



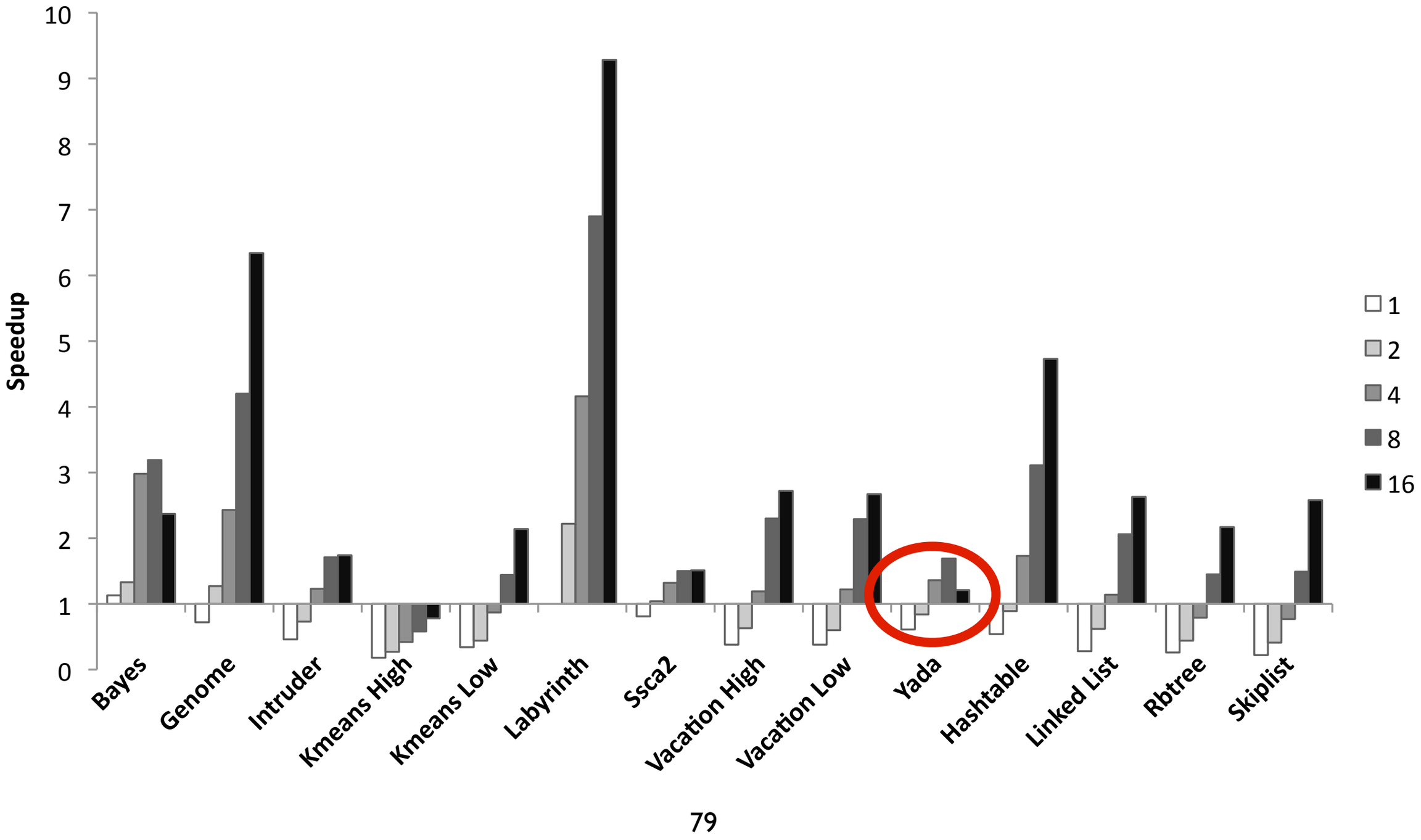
SwissTM vs Sequential

(x86 Intel compiler)



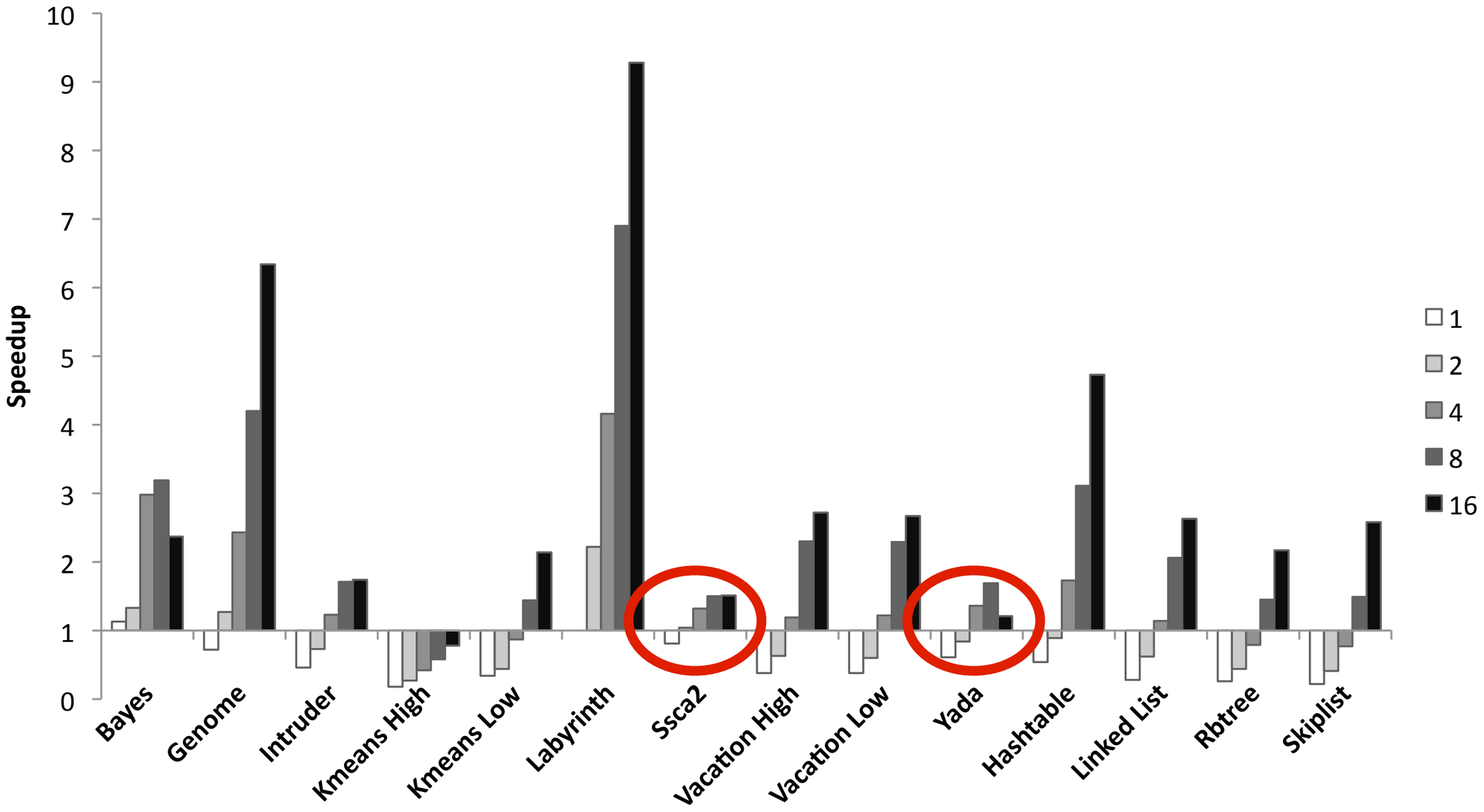
SwissTM vs Sequential

(x86 Intel compiler)



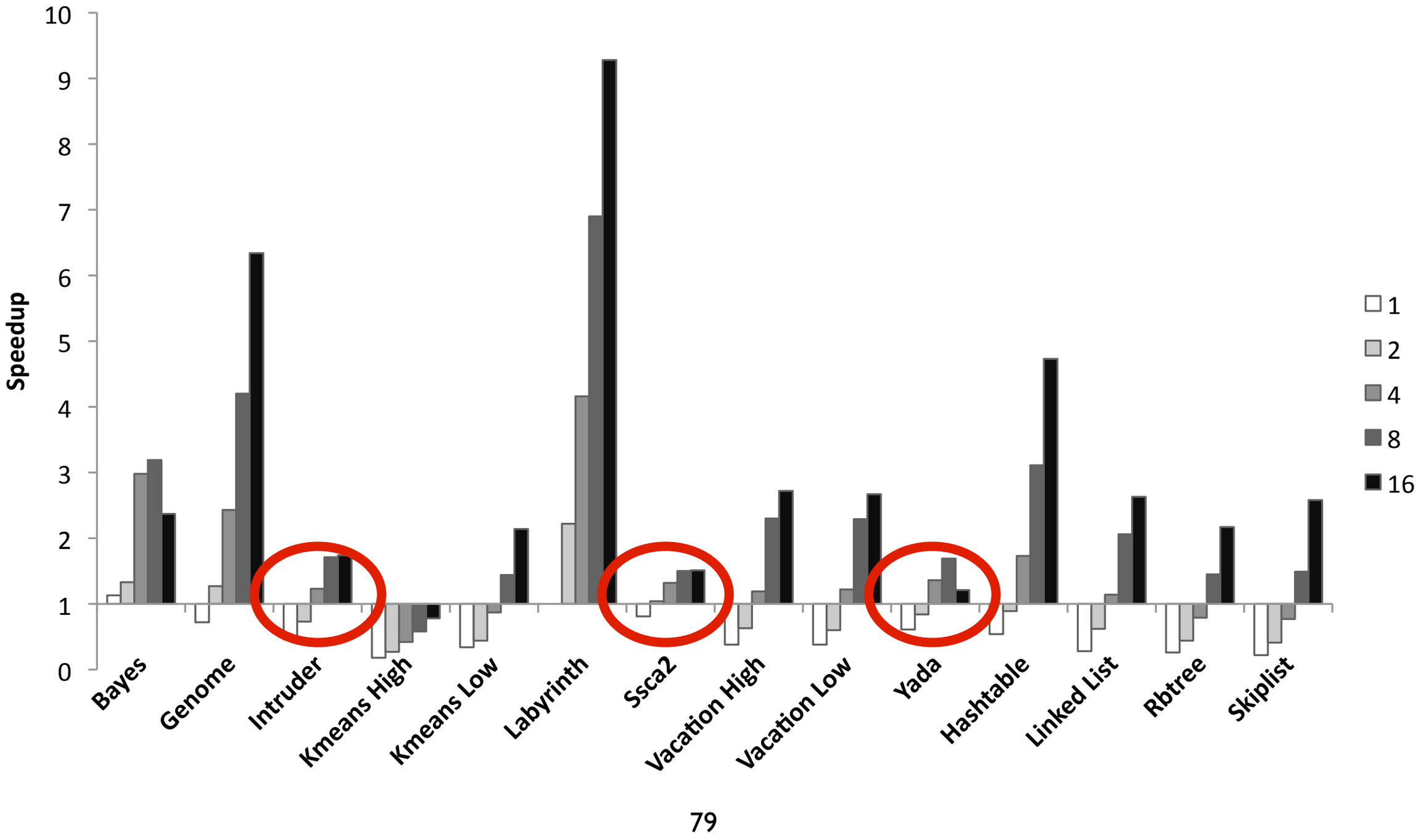
SwissTM vs Sequential

(x86 Intel compiler)



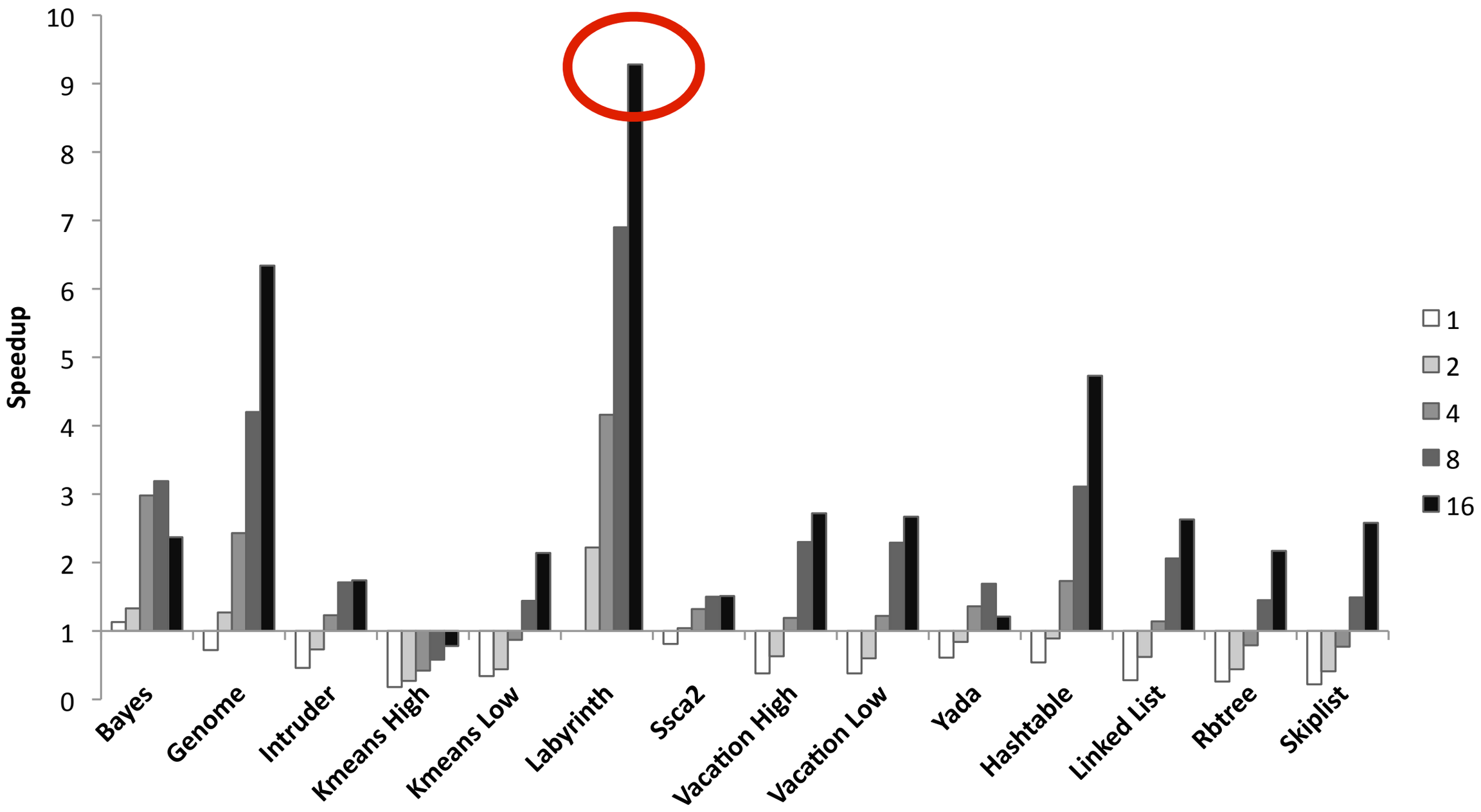
SwissTM vs Sequential

(x86 Intel compiler)



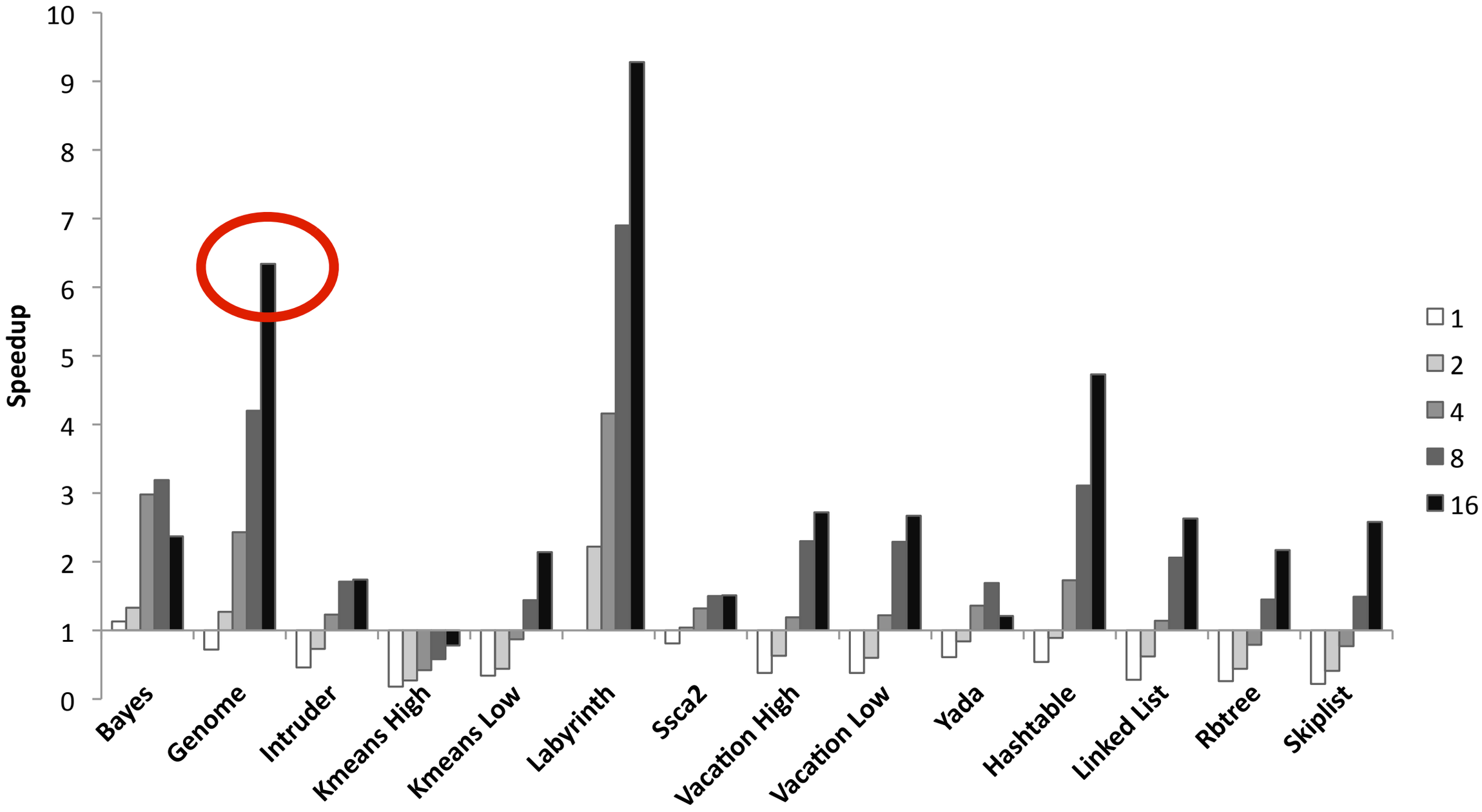
SwissTM vs Sequential

(x86 Intel compiler)



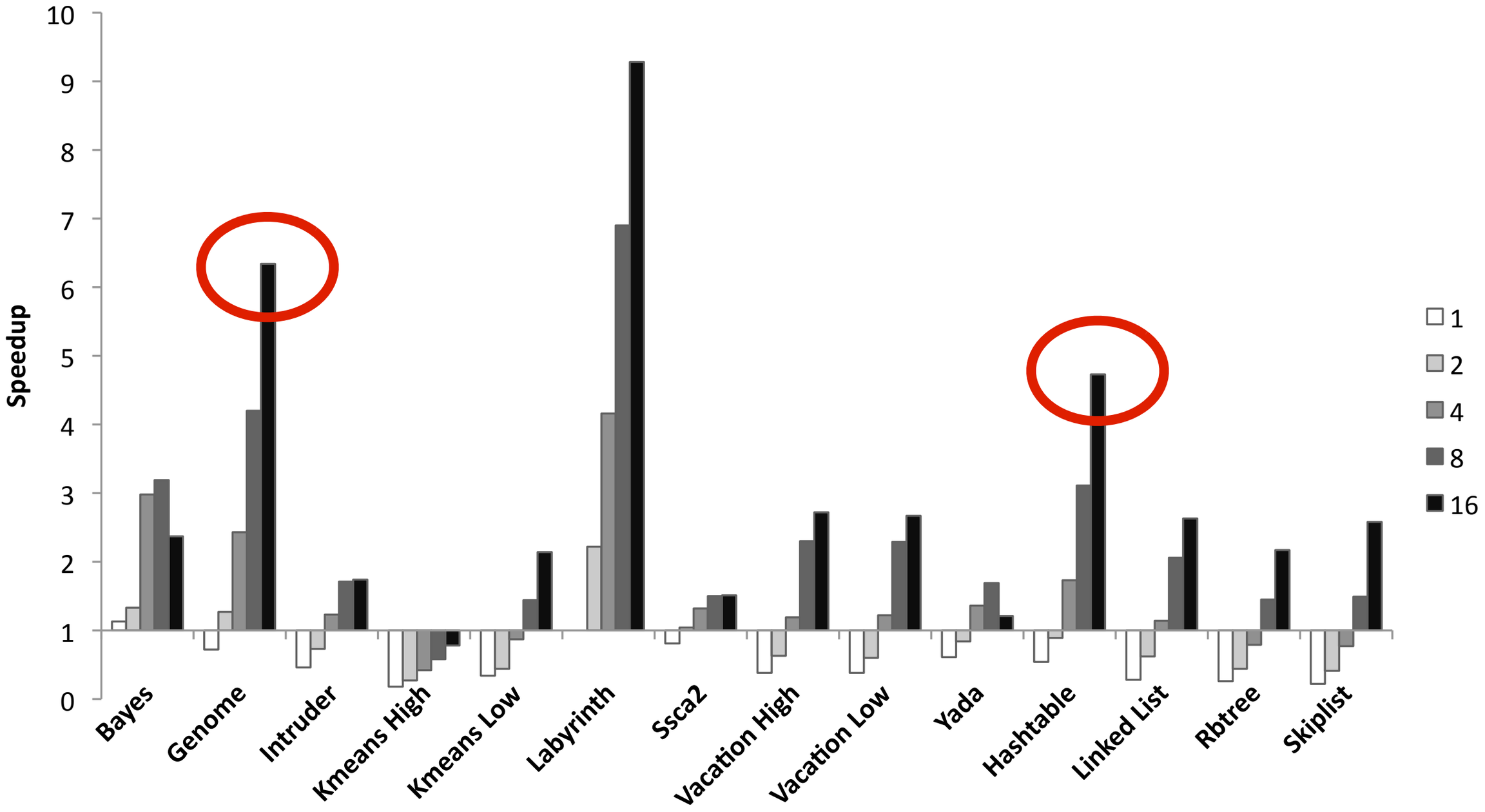
SwissTM vs Sequential

(x86 Intel compiler)

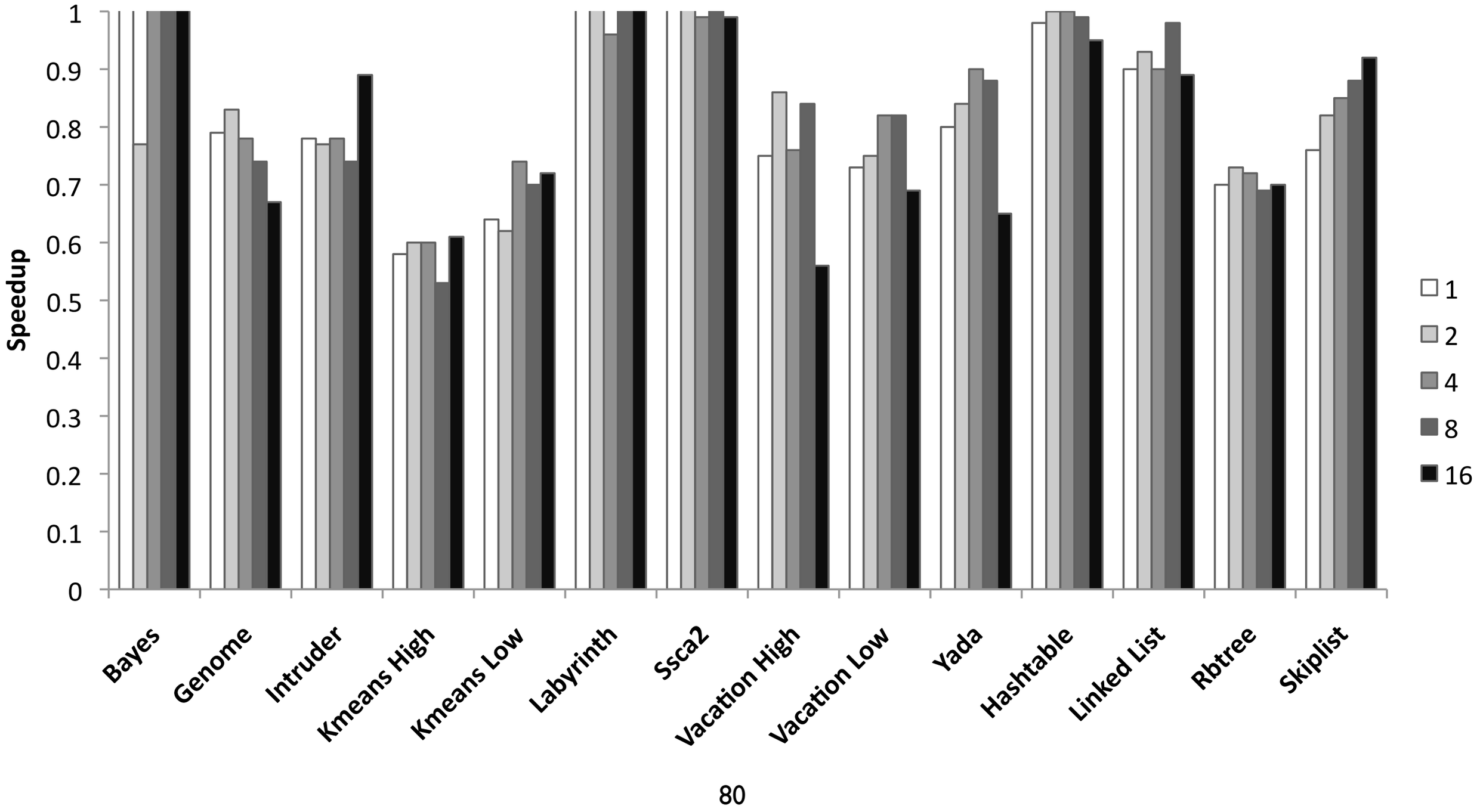


SwissTM vs Sequential

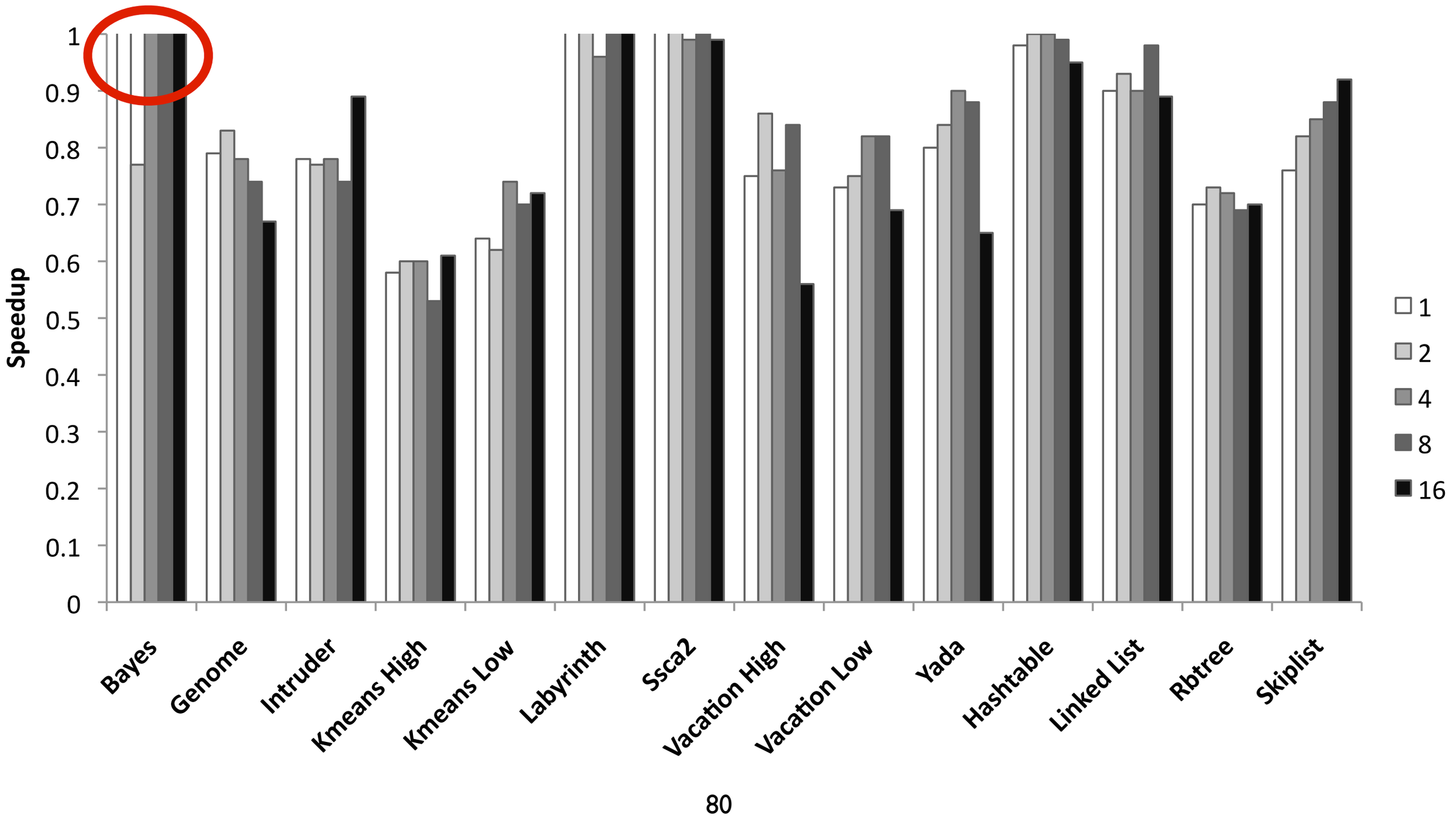
(x86 Intel compiler)



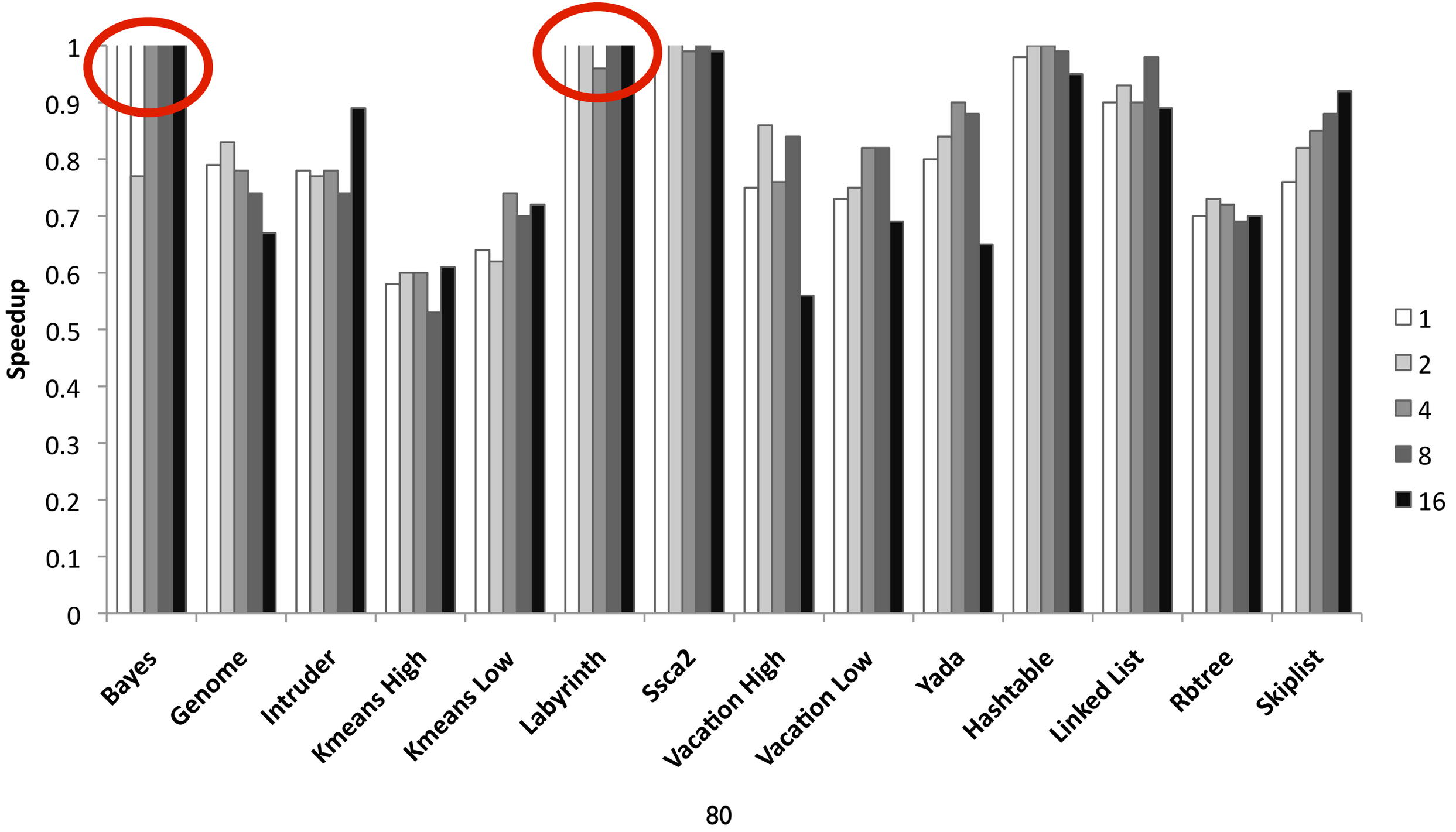
Compiler cost



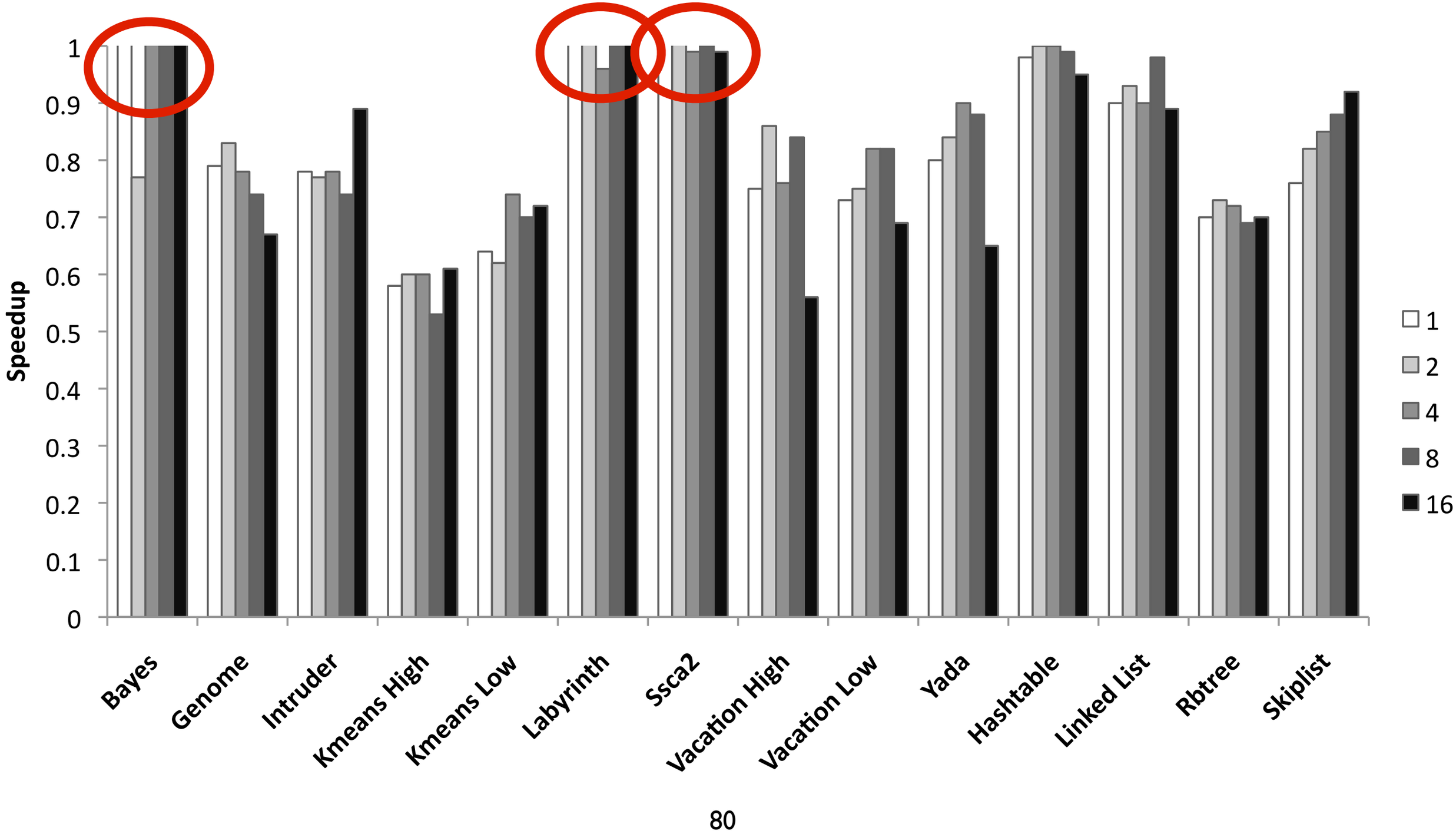
Compiler cost



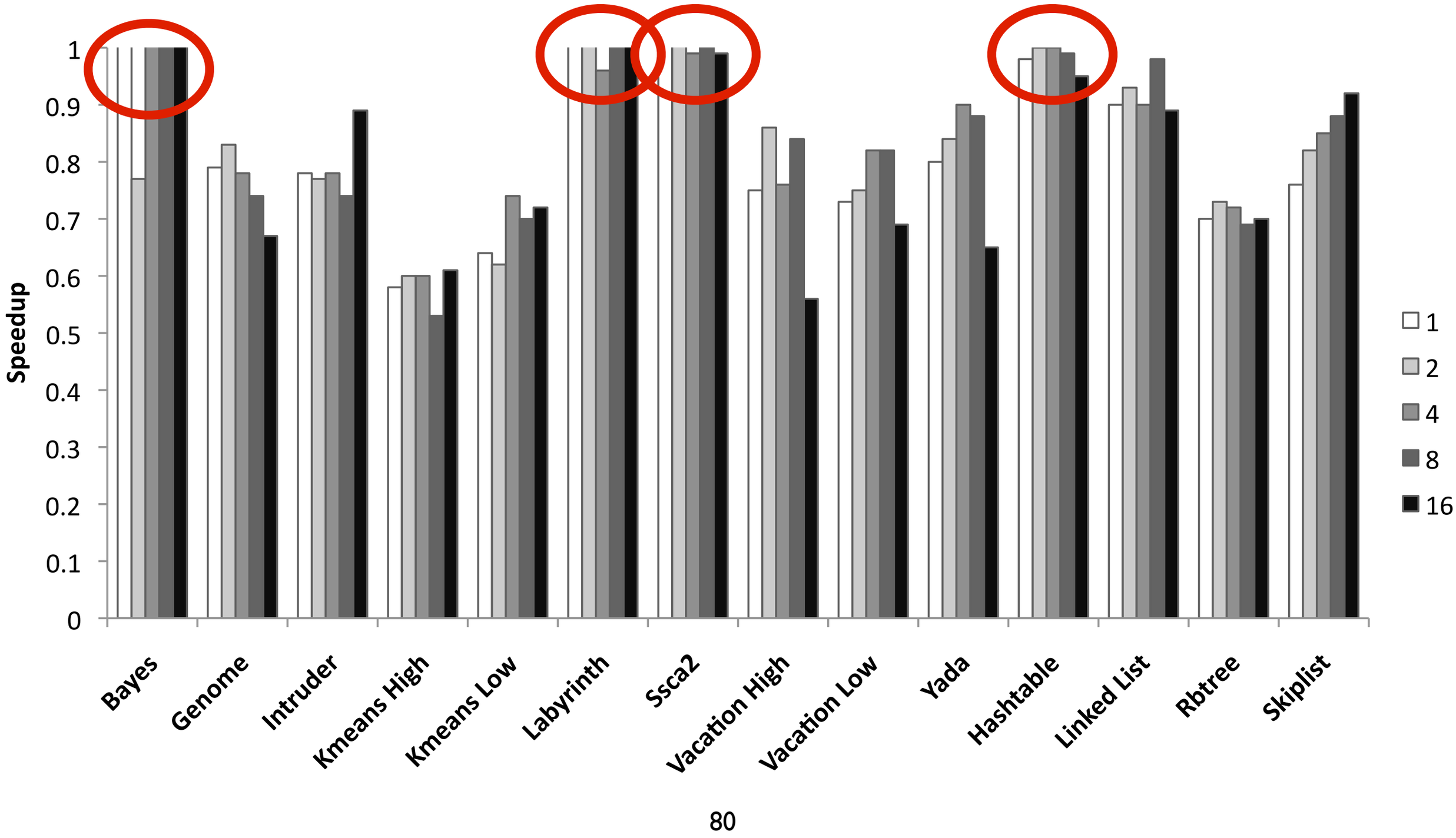
Compiler cost



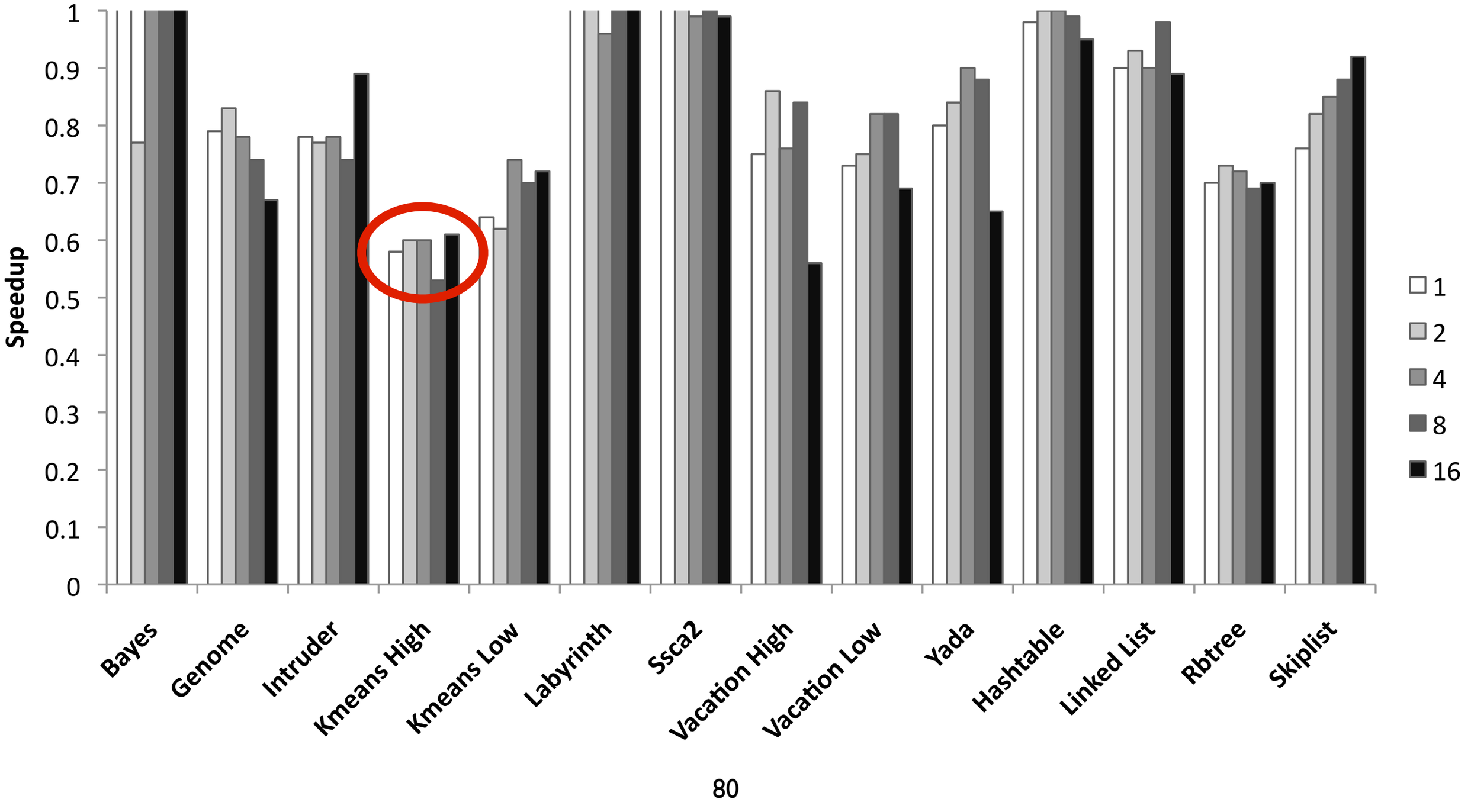
Compiler cost



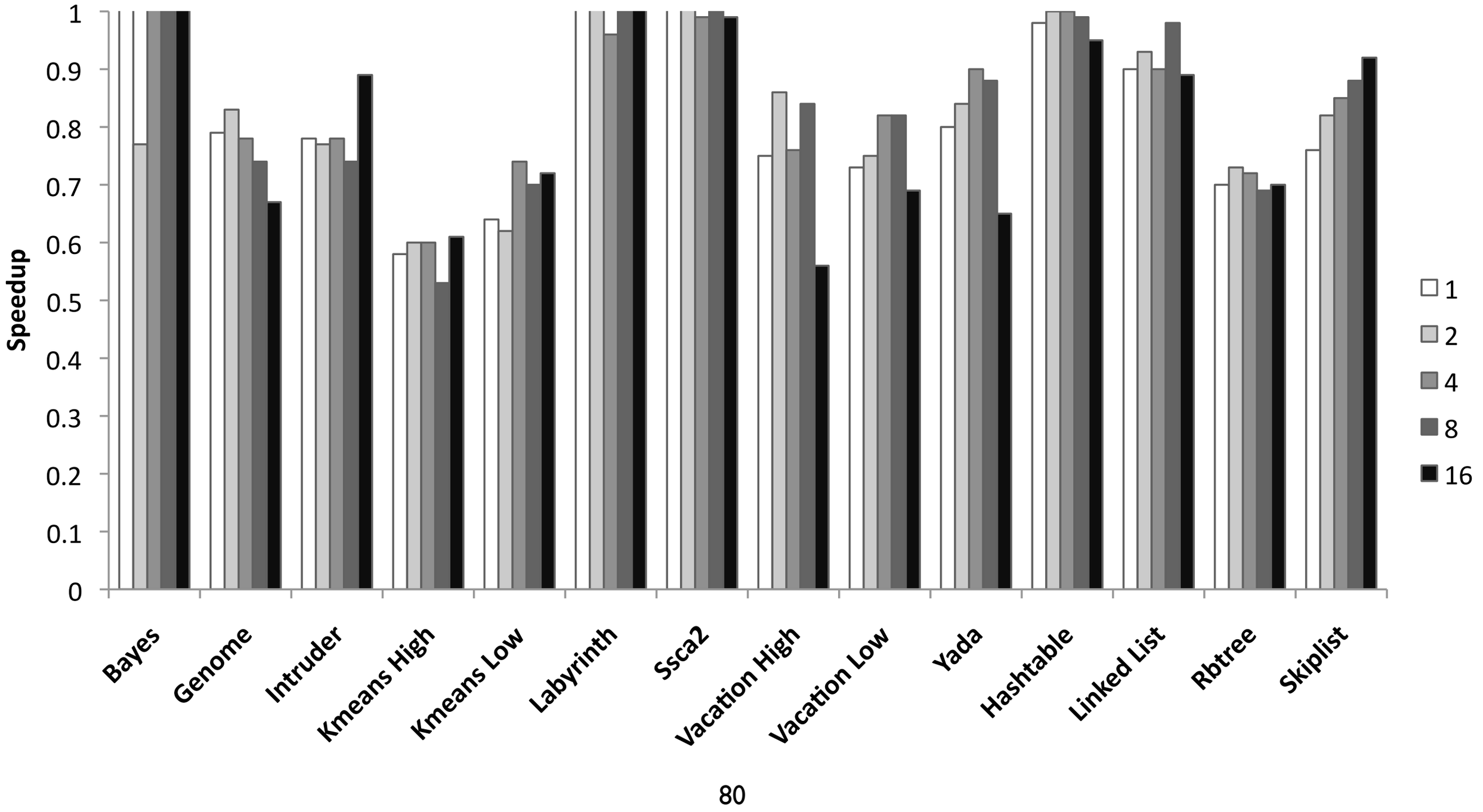
Compiler cost



Compiler cost

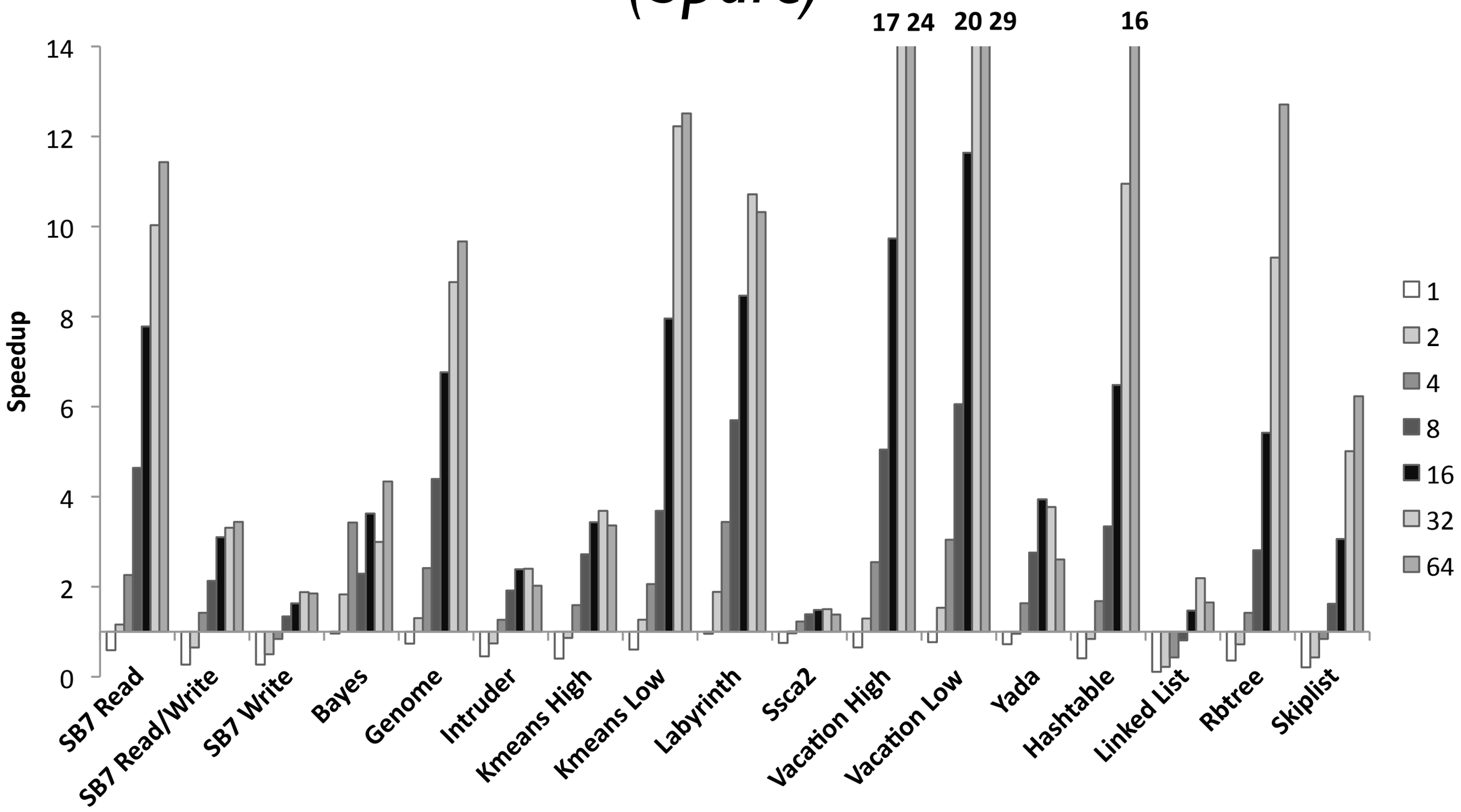


Compiler cost



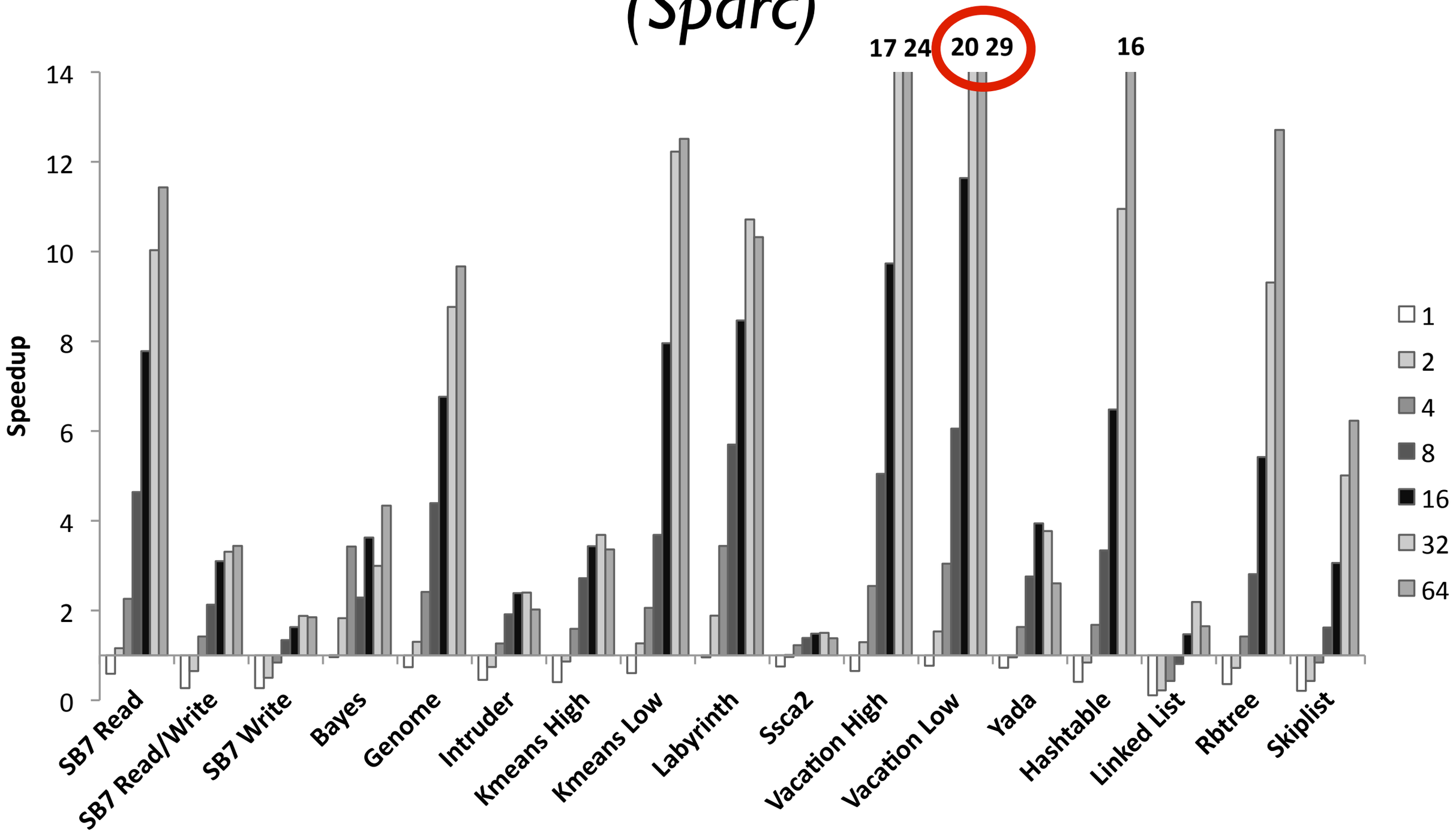
SwissTM vs Sequential

(Sparc)



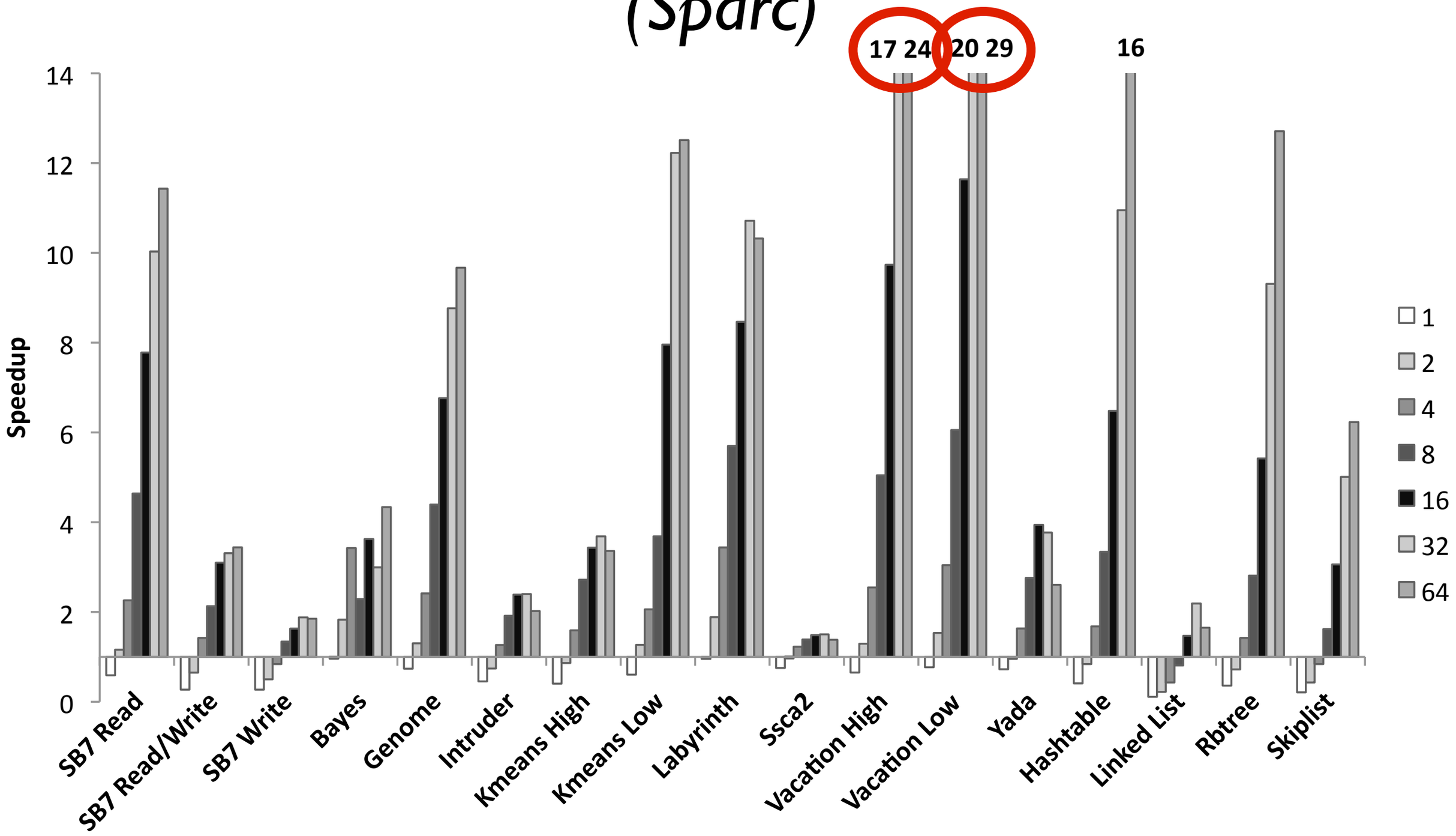
SwissTM vs Sequential

(Sparc)



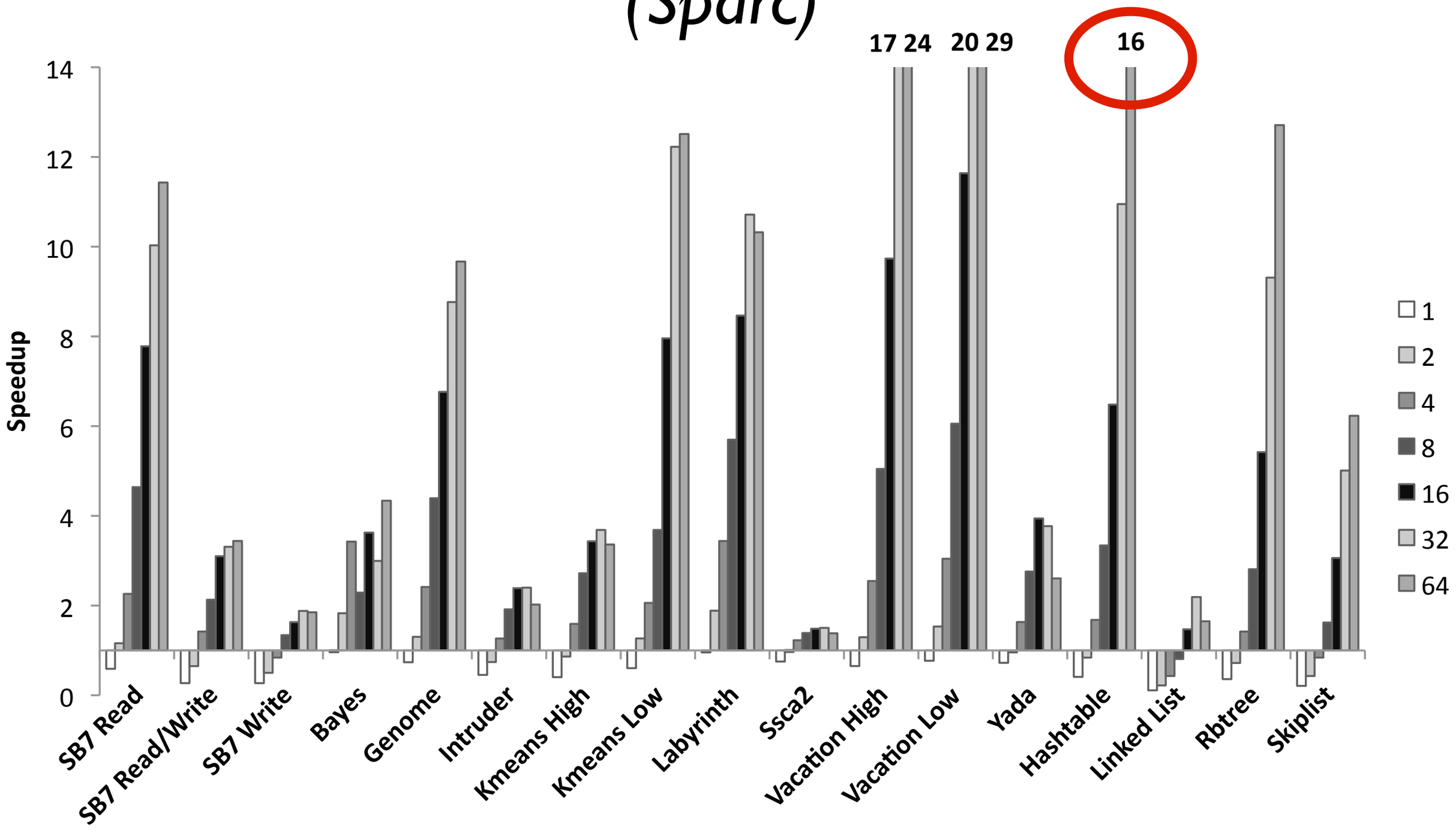
SwissTM vs Sequential

(Sparc)



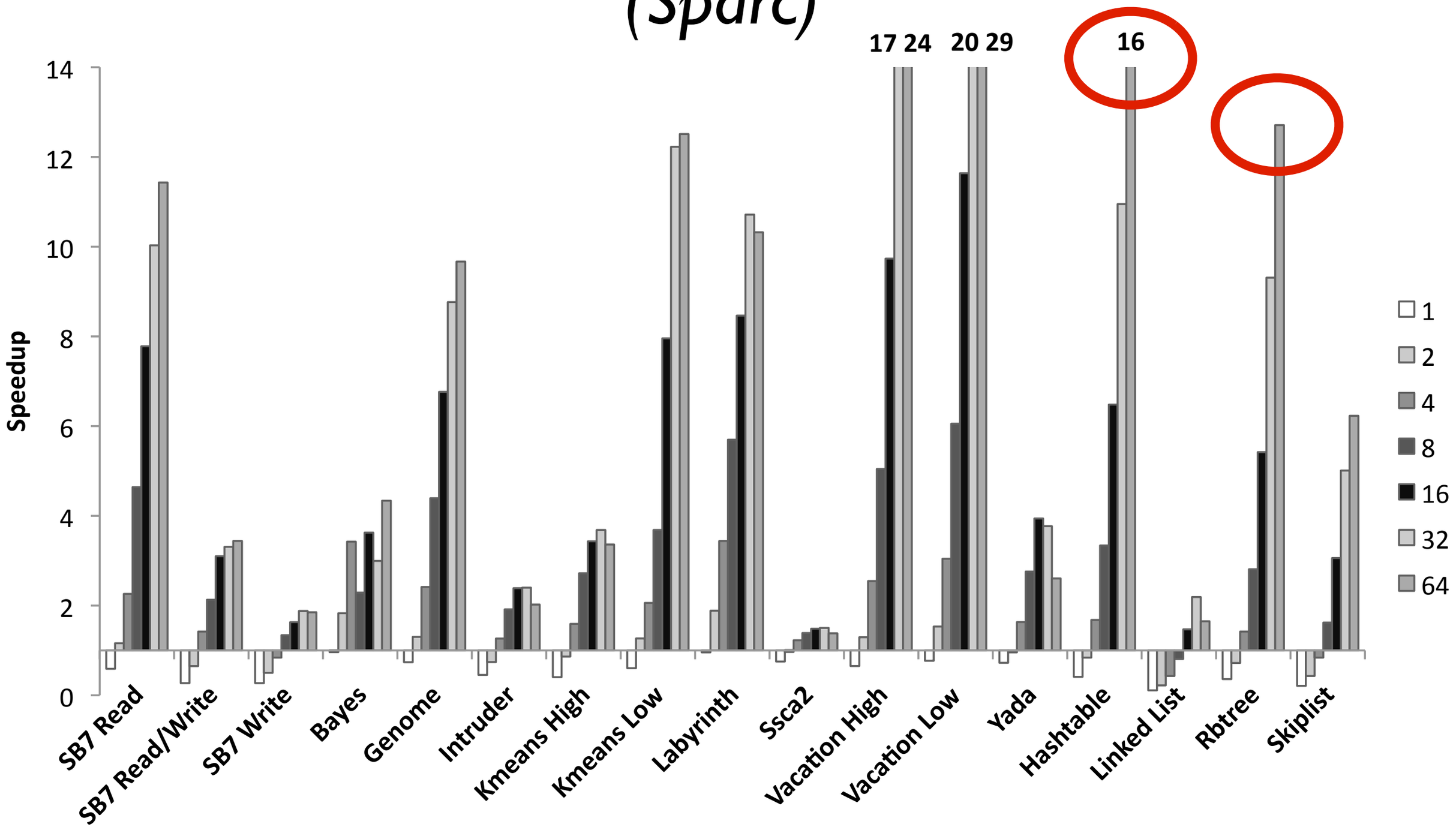
SwissTM vs Sequential

(Sparc)



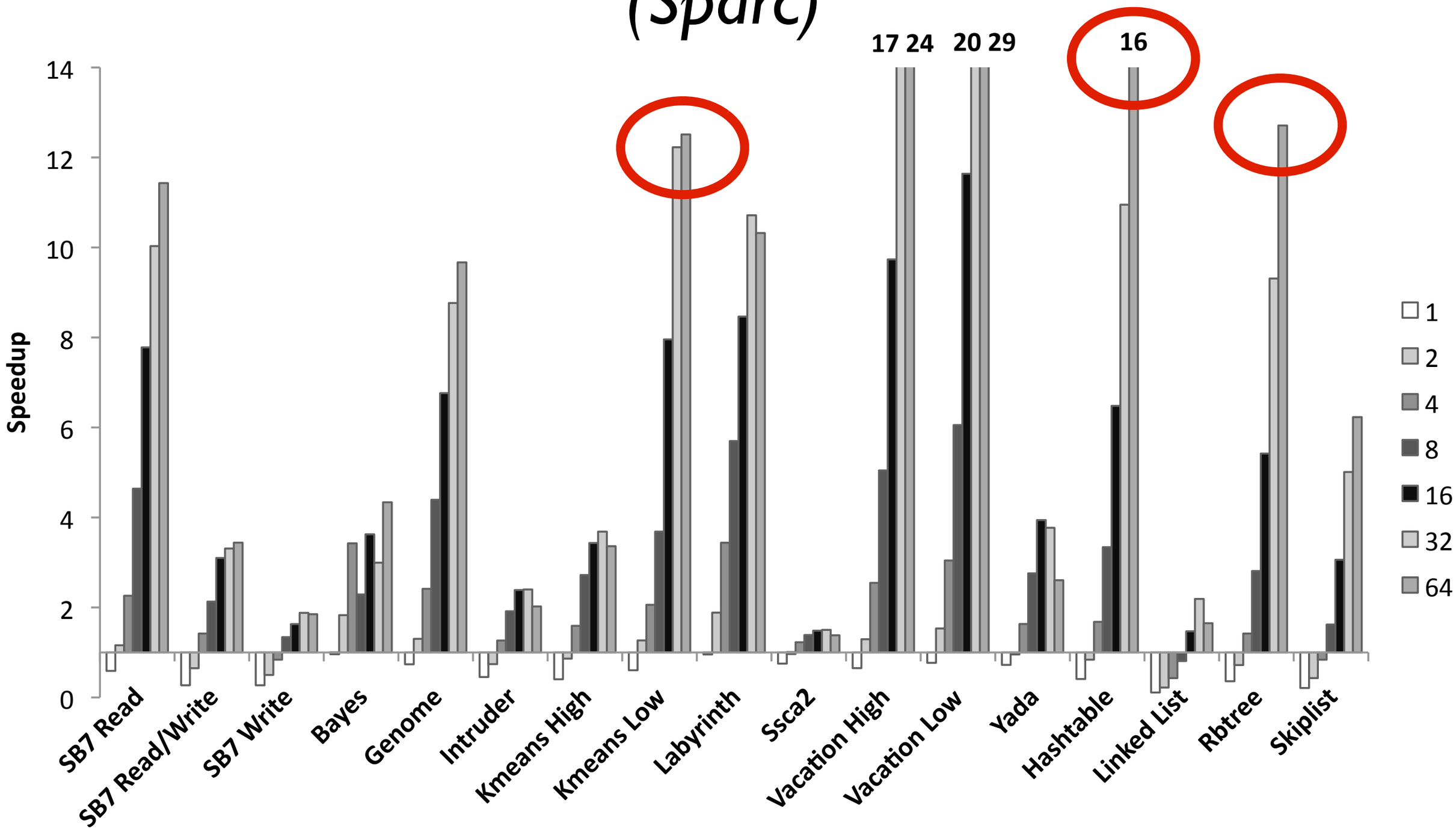
SwissTM vs Sequential

(Sparc)



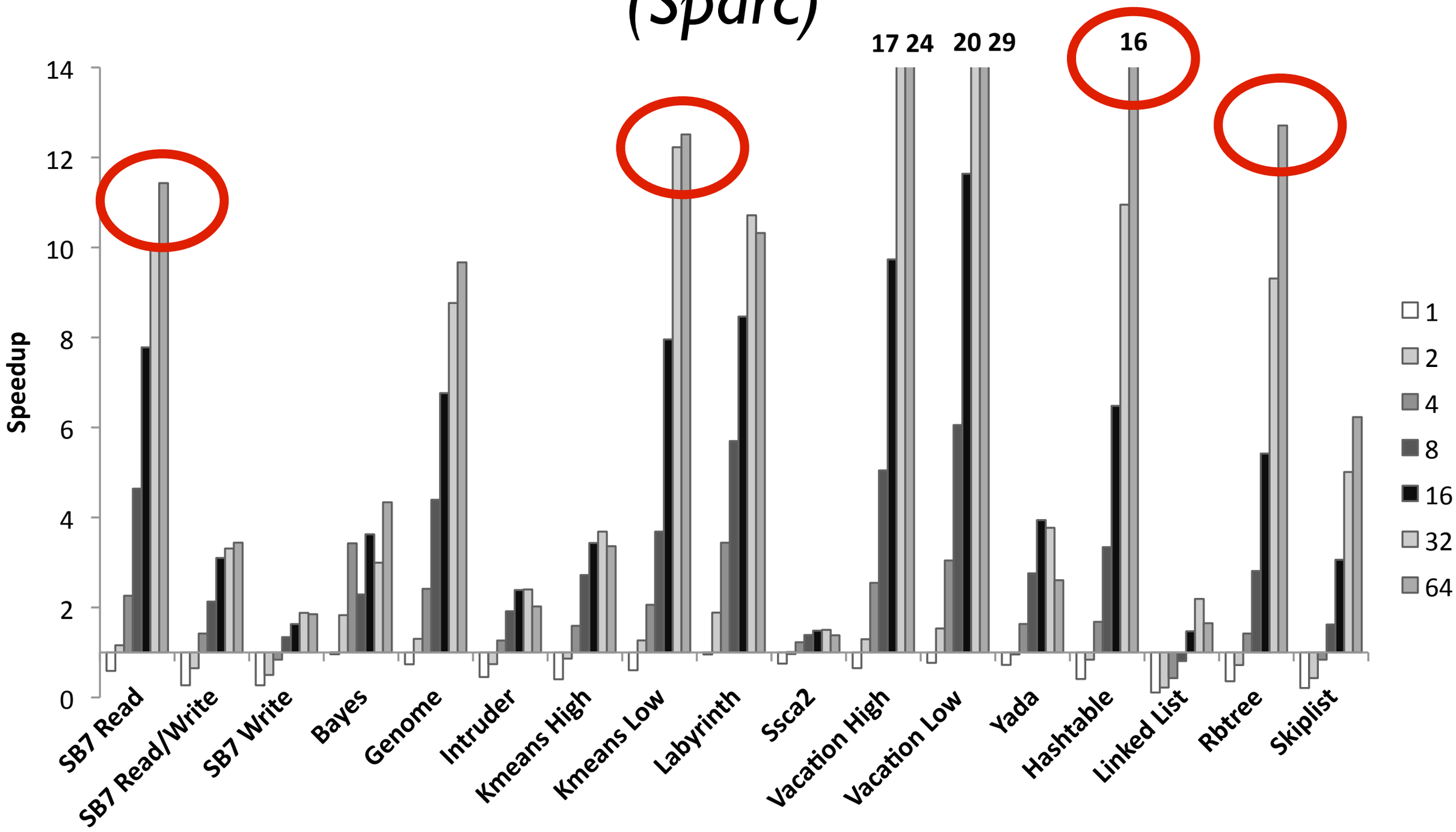
SwissTM vs Sequential

(Sparc)



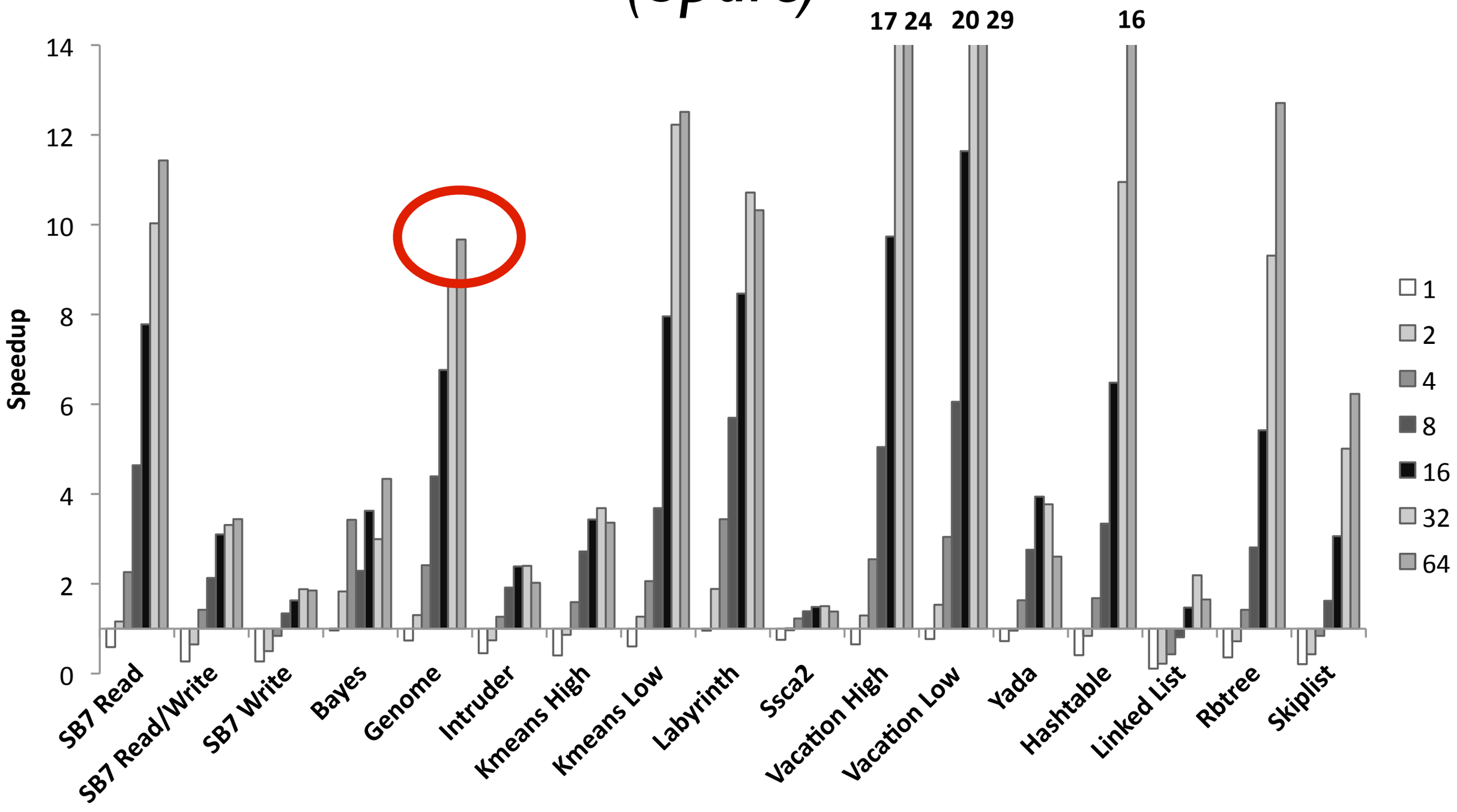
SwissTM vs Sequential

(Sparc)



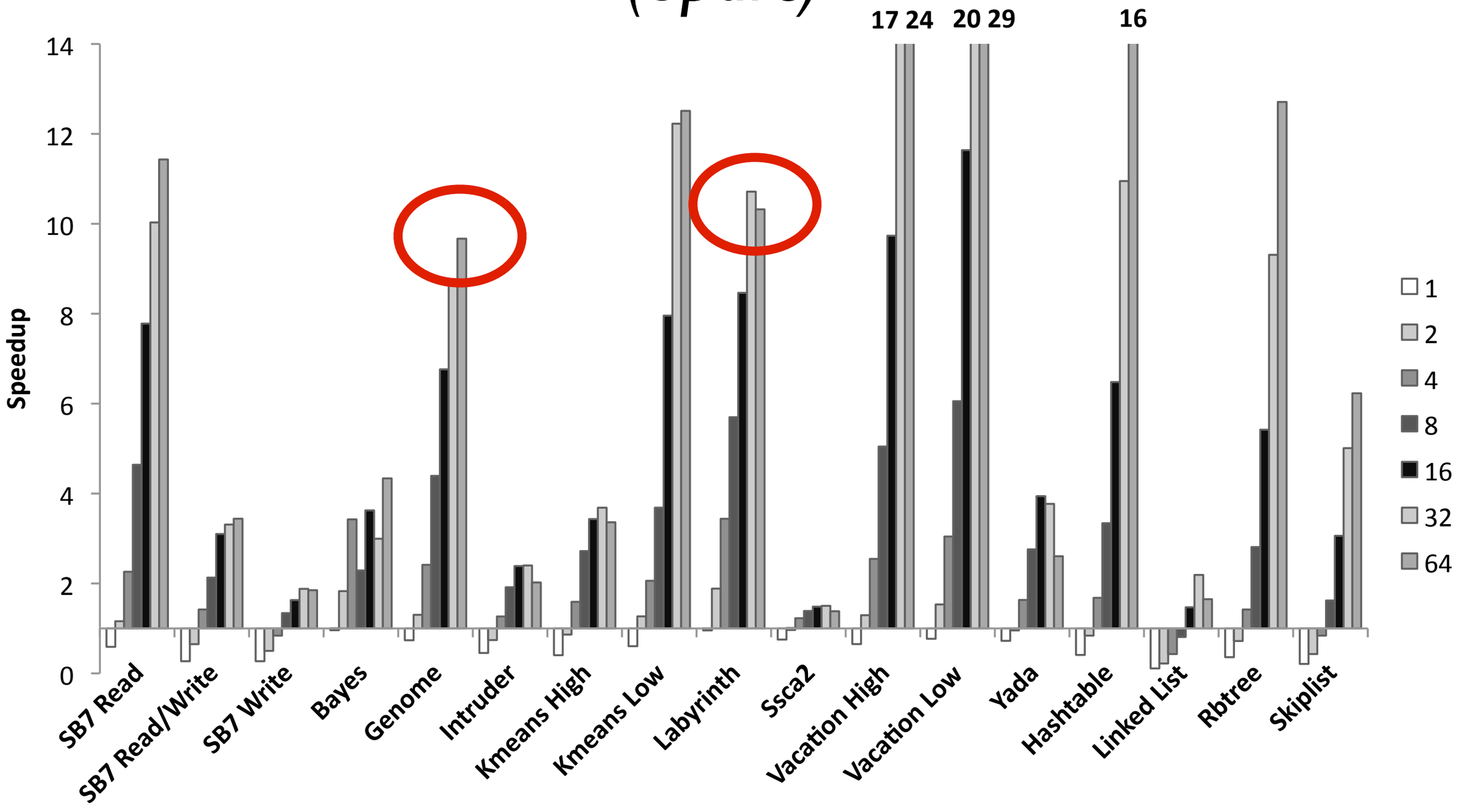
SwissTM vs Sequential

(Sparc)



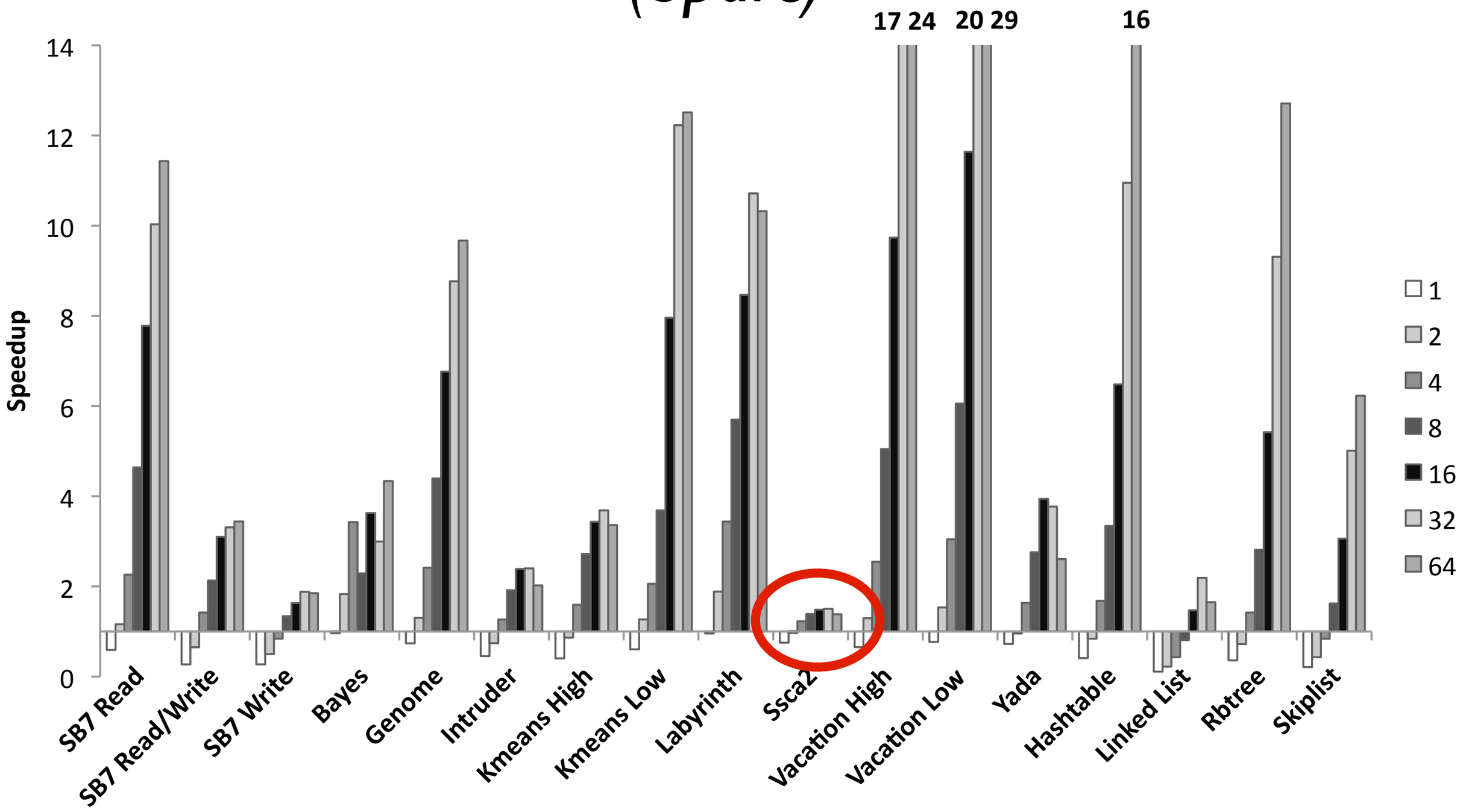
SwissTM vs Sequential

(Sparc)



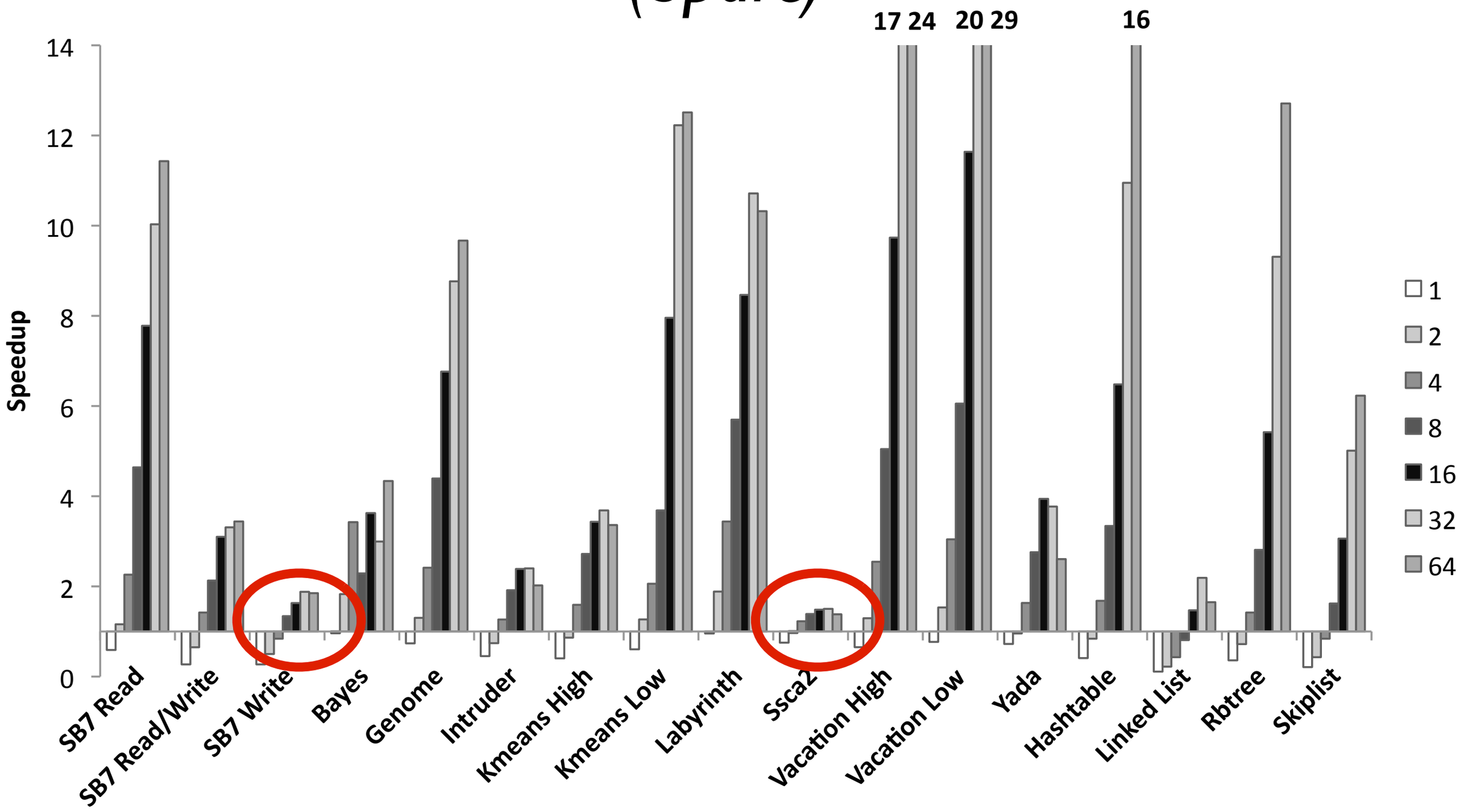
SwissTM vs Sequential

(Sparc)



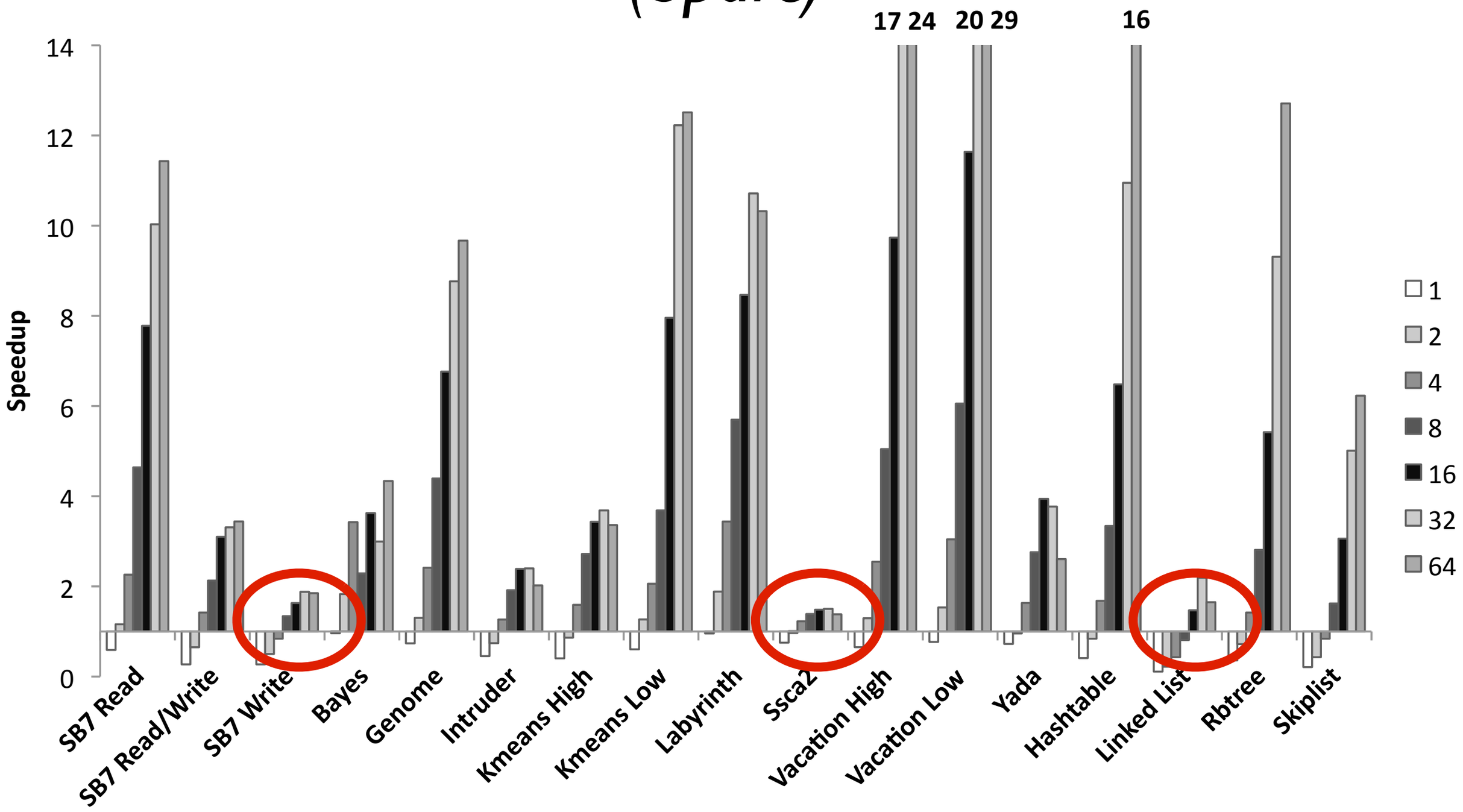
SwissTM vs Sequential

(Sparc)



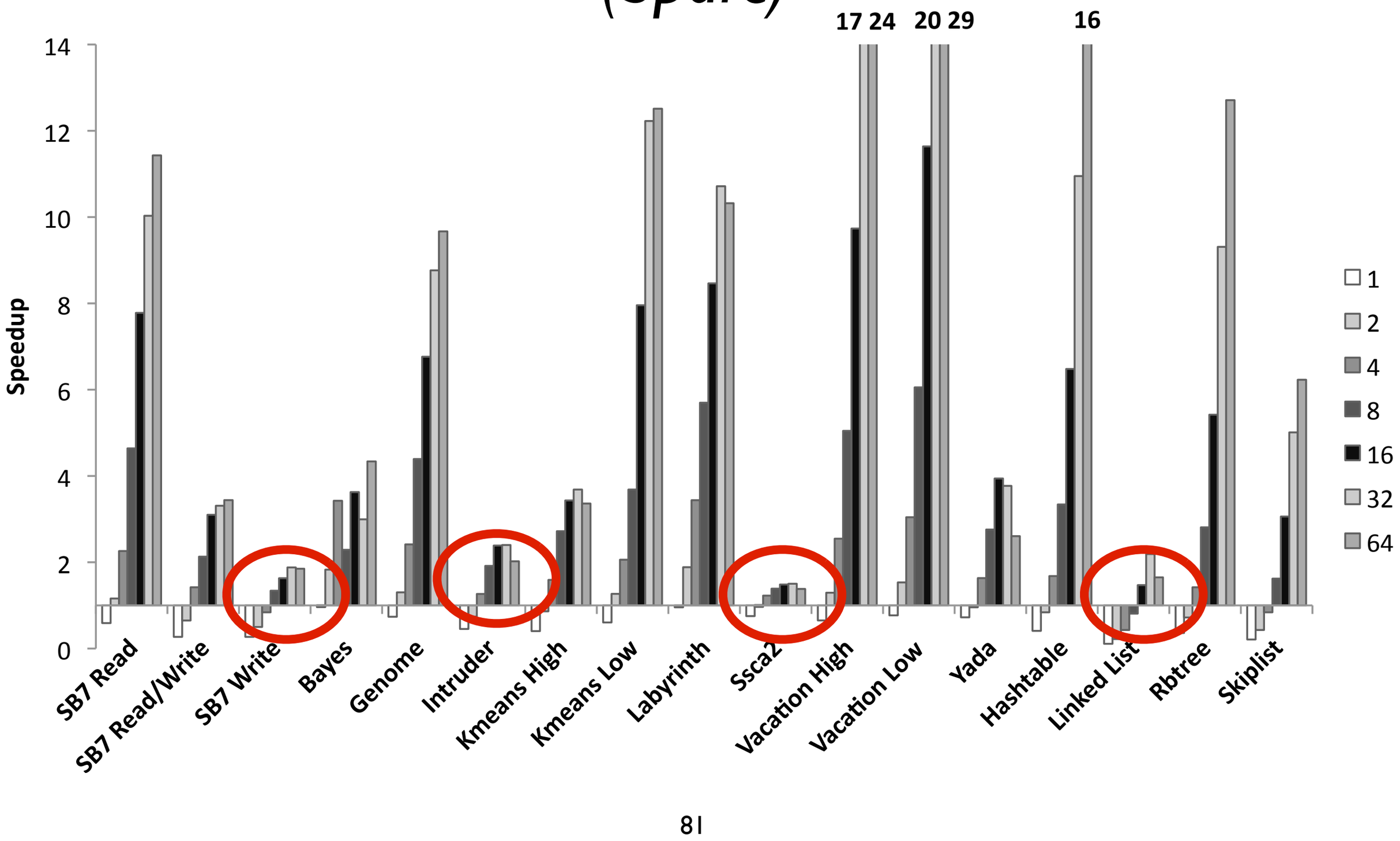
SwissTM vs Sequential

(Sparc)



SwissTM vs Sequential

(Sparc)



SwissTM vs Sequential

	x86 (16)	SPARC (64)
SwissTM (manual)	16/17	17/17
SwissTM (compiler)	13/14	-
Total	46/48 (95%)	

SwissTM vs Sequential

	x86 (16)			SPARC (64)		
	min	max	avg	min	max	avg
SwissTM (manual)	0.54	9.4	**3.37**	1.38	29.72	**9.08**
SwissTM (compiler)	0.78	9.28	**3.06**	-		

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

References

- Rachid Guerraoui, Michał Kapałka, and Jan Vitek. **STMBench7: A Benchmark for Software Transactional Memory.** *EuroSys 2007.*

- Rachid Guerraoui and Michał Kapałka. **On the Correctness of Transactional Memory.** *PPoPP 2008.*

- Aleksandar Dragojević, Rachid Guerraoui, and Michał Kapałka. **Dividing Transactional Memories by Zero.** *Transact 2008.*

References (2)

- Aleksandar Dragojević, Rachid Guerraoui, and Michał Kapałka. **Stretching Transactional Memory.** *PLDI 2009.*

- Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, Rachid Guerraoui. **Why STM can be more than a Research Toy.** *CACM April, 2011.*

- Aleksandar Dragojević, Rachid Guerraoui. **Predicting the Scalability of an STM: A Pragmatic Approach.** *Transact 2010.*

Thank you

Questions, comments?