A Runtime System for Software Lock Elision

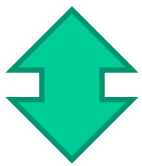
Amitabha Roy (U. Cambridge)
Steven Hand (U. Cambridge)
Tim Harris (MSR Cambridge)

Motivation

- Multicores mean application scalability is key to good performance
- Scaling programs synchronising with locks
  - Existing software systems use locks
  - Locks are very popular with programmers
- Start with data race free correctly synchronised lock based program
- Use transactional memory opportunistically while retaining the locks

Critical Sections & Speculation

Thread 1:
Lock(L)
Do stuff …
Unlock(L)

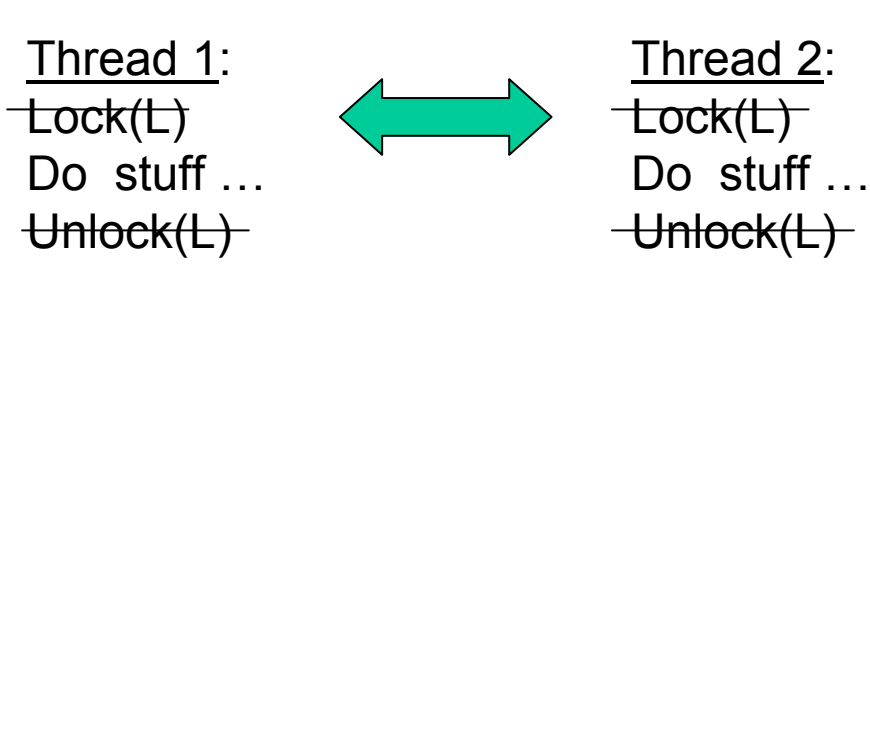


Serialize

Thread 2:
Lock(L)
Do stuff …
Unlock(L)

Critical Sections & Speculation

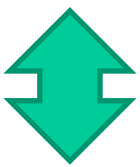
Rajwar et al: *Speculative Lock Elision* … Micro 2001



- Relies on Hardware Transactional Memory (TM) support to enable optimistic concurrency control
- Exploits disjoint-access parallelism (red-black trees, hash tables, etc)

Critical Sections & Speculation

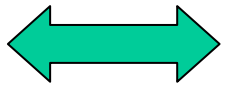
Thread 1:
Lock(L)
Do stuff …
Unlock(L)



Serialize

Thread 2:
Lock(L)
Do stuff …
Unlock(L)

Thread 1:
~~Lock(L)~~
Do stuff …
~~Unlock(L)~~



Thread 2:
~~Lock(L)~~
Do stuff …
~~Unlock(L)~~

- Can coexist (excessive conflicts, I/O, wait conditions, ...)
- No need for new semantics – start from lock-based programs
- This paper: ***Software Lock Elision*** **(SLE)**; no special h/w required

Coming Up ...

- Speculation in software
- Retaining lock semantics & behaviour
- Implementation and evaluation
- Interfacing to the runtime

Speculation

- Speculating threads and memory
  - Isolate using thread private copies
  - Write back changes atomically
- Well developed ideas in the Software Transactional Memory (STM) field
- We use a design similar to TL2
  - Dice et al: *Transactional Locking II* … DISC 2006

Speculation: Shadowing

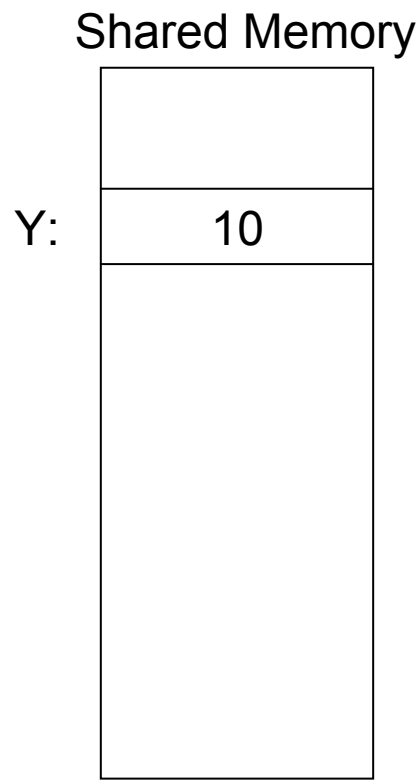
Lock(L) ← elided

…

X = Y + 1

…

Unlock(L)



Speculation: Shadowing

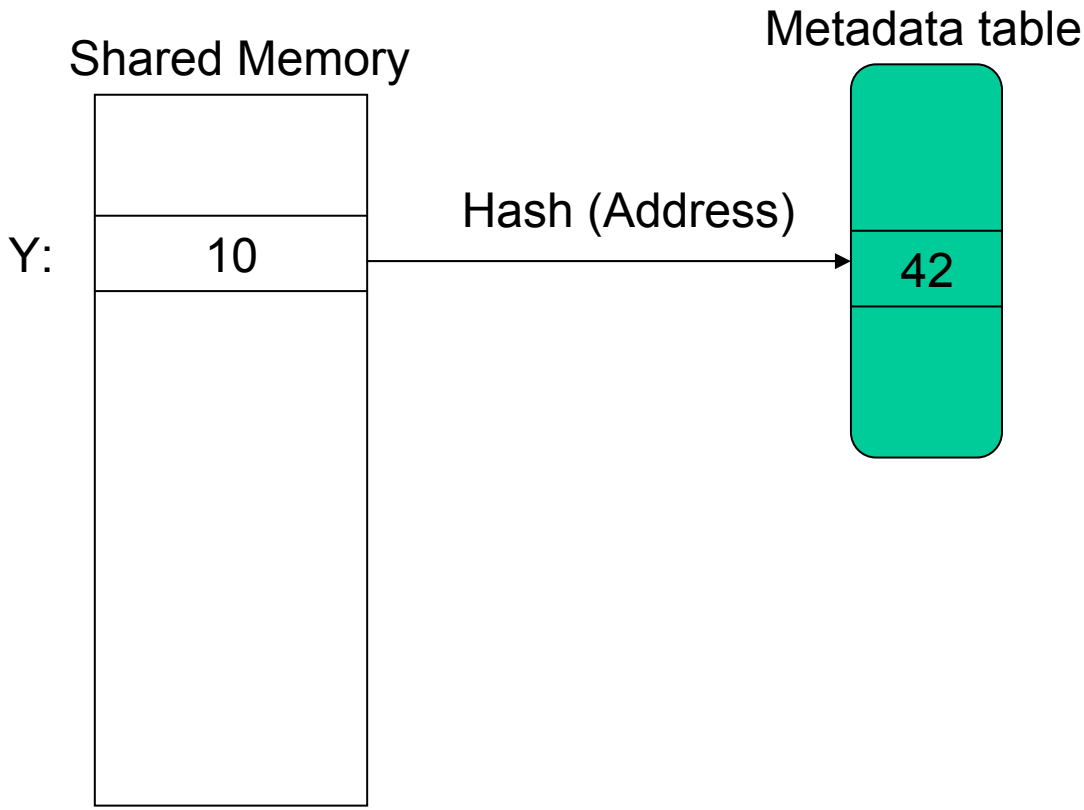
Lock(L) ← elided

…

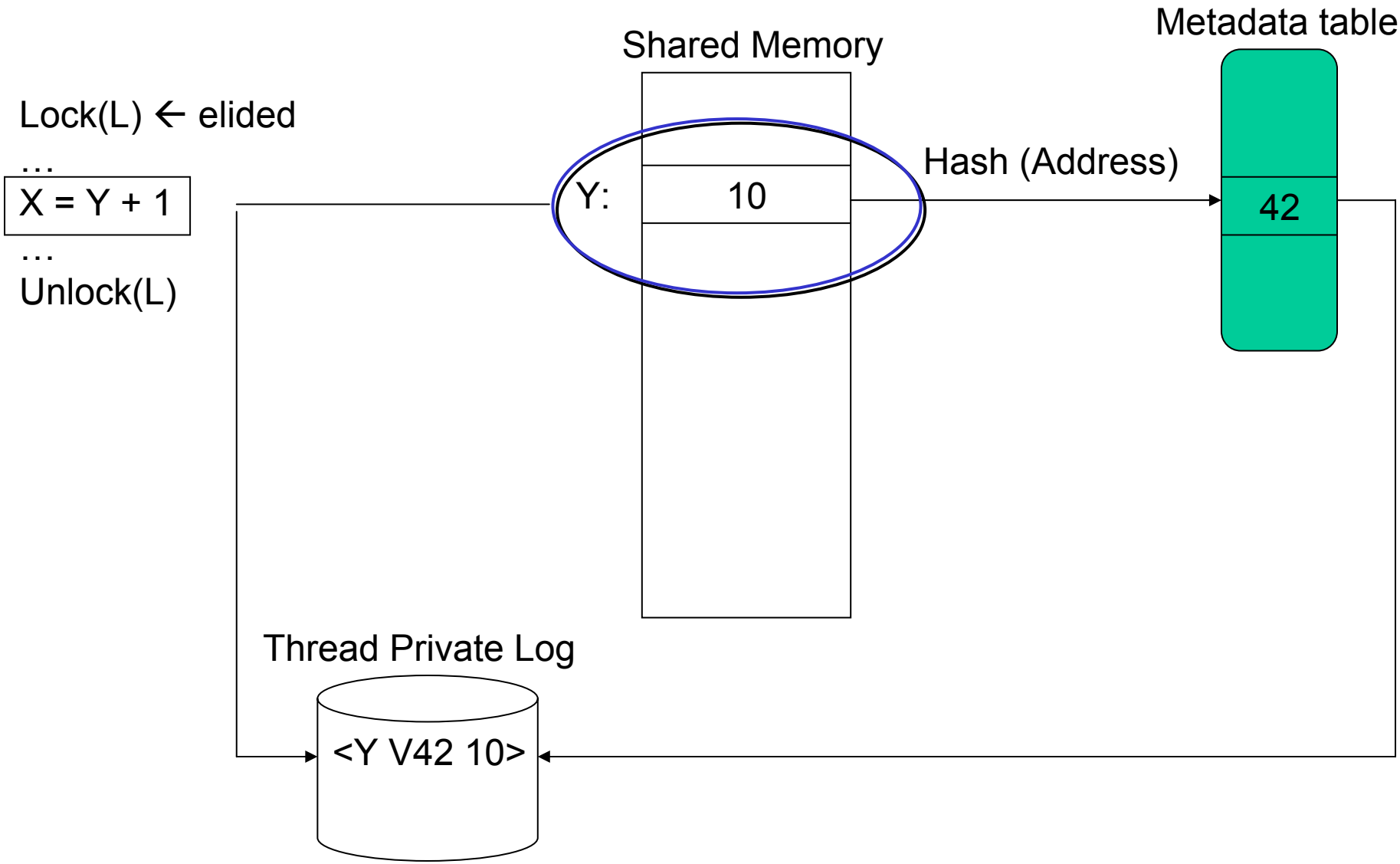
X = Y + 1

…

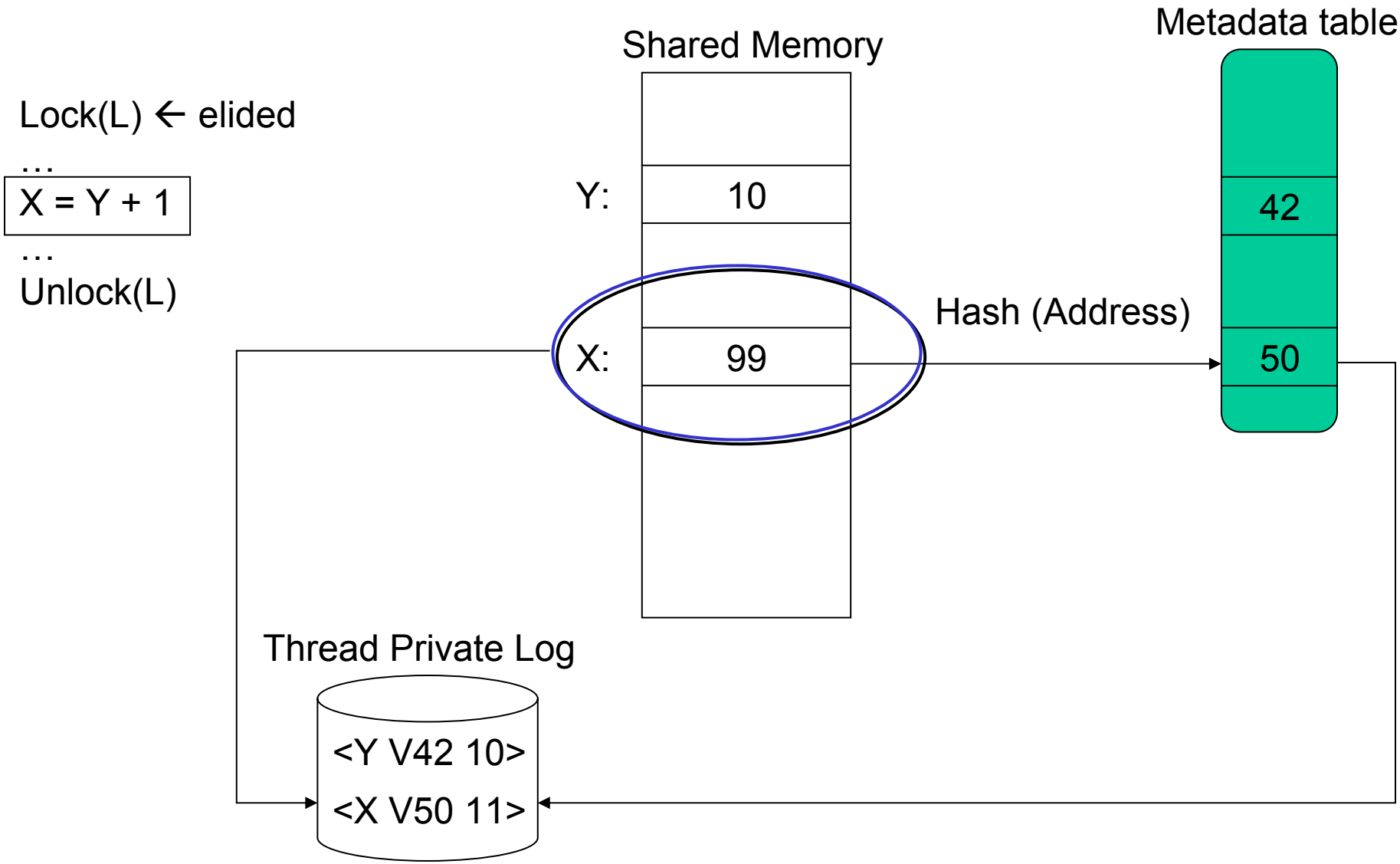
Unlock(L)



Speculation: Shadowing



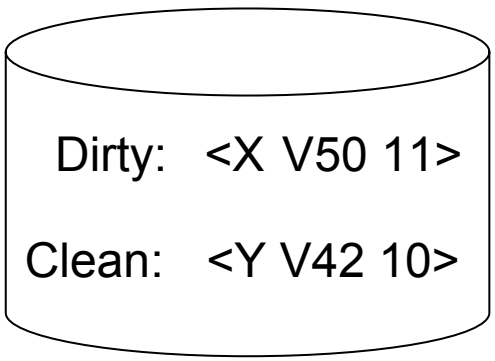
Speculation: Shadowing



Speculation: Commit

- Commit (2PL): Lock, Verify, Write, Unlock

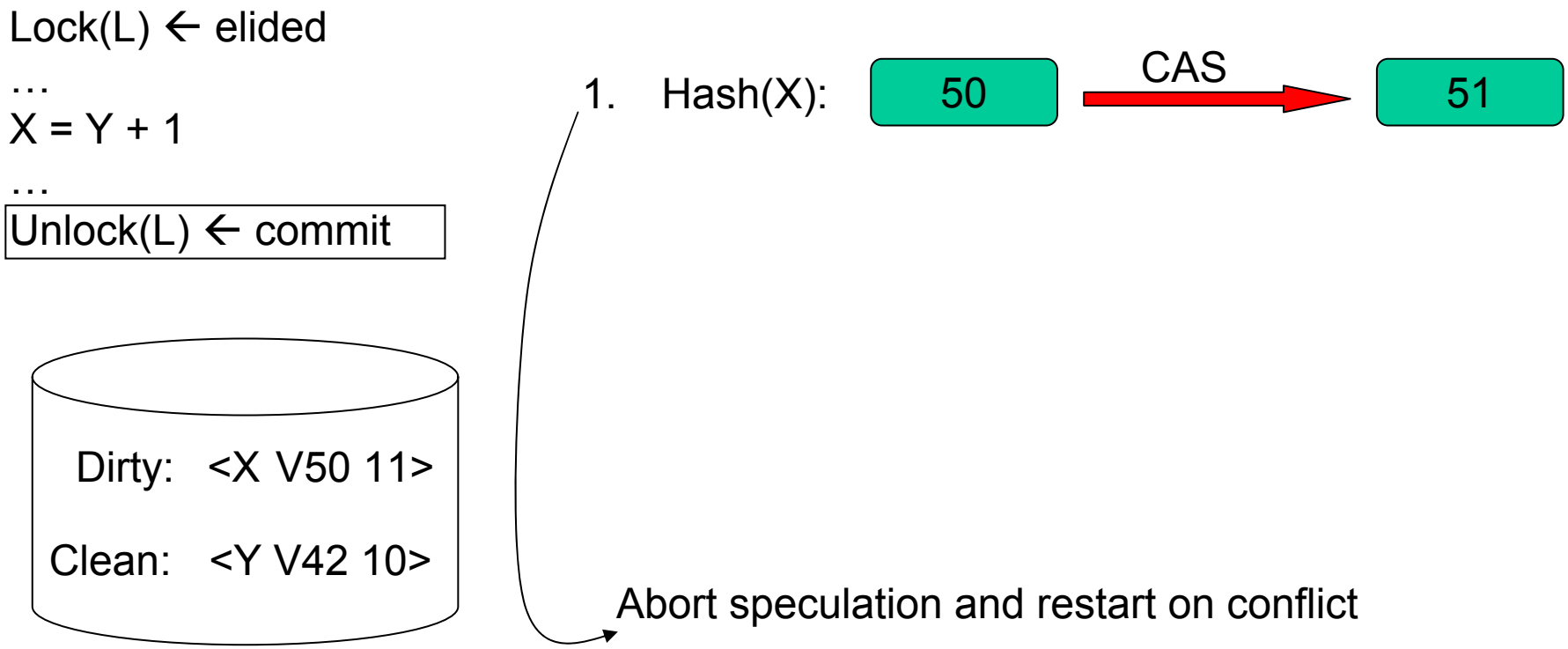
Lock(L) ← elided
…
X = Y + 1
…
Unlock(L) ← commit



- Odd version numbers used to represent locked objects
- Manipulate with Compare and Swap (CAS) for atomicity

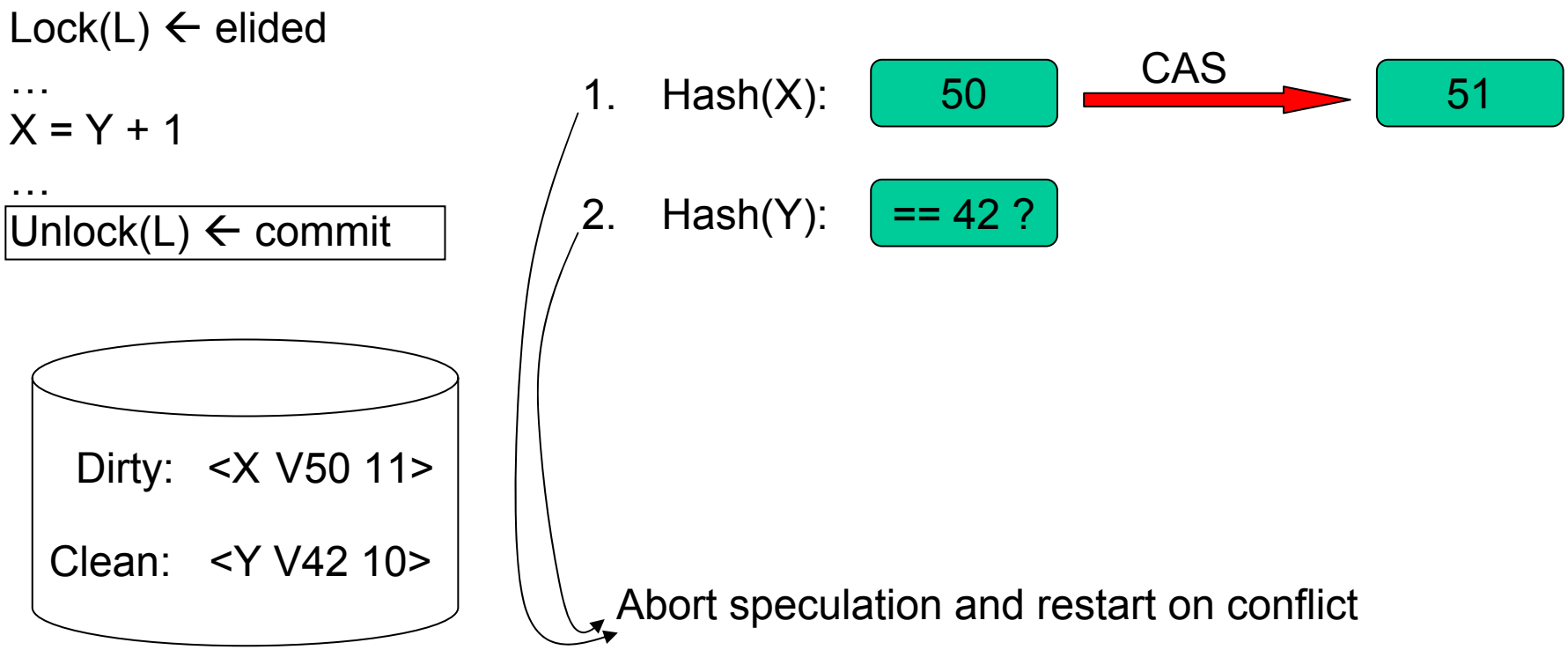
Speculation: Commit

- Commit (2PL): *Lock*, Verify, Write, Unlock



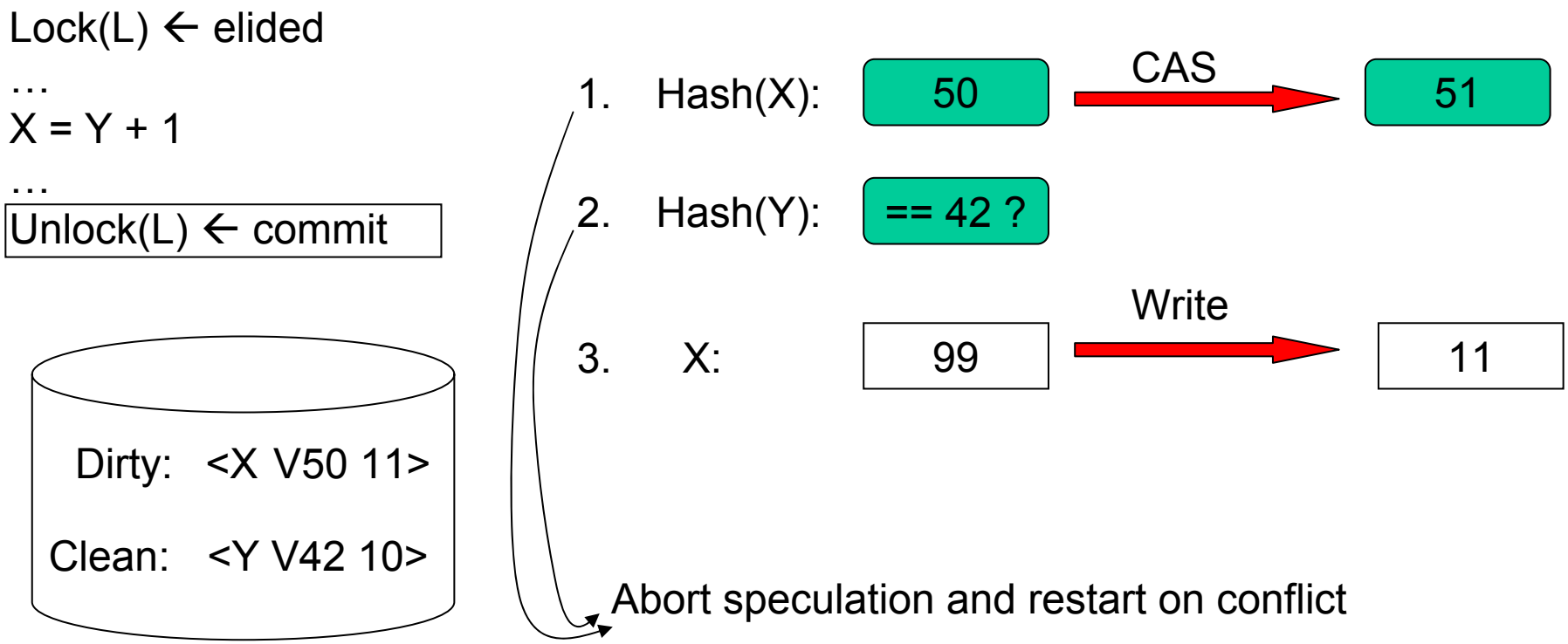
Speculation: Commit

- Commit (2PL): Lock, *Verify*, Write, Unlock



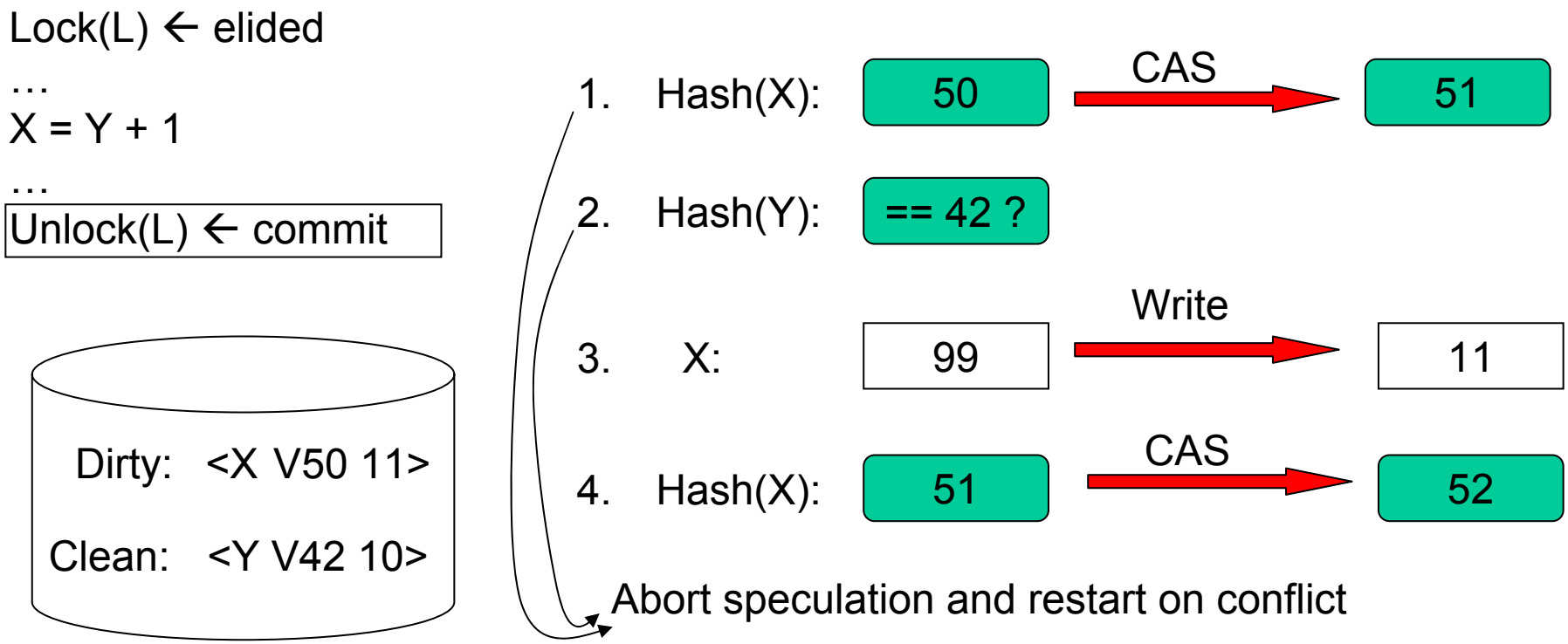
Speculation: Commit

- Commit (2PL): Lock, Verify, *Write*, Unlock



Speculation: Commit

- Commit (2PL): Lock, Verify, Write, *Unlock*



Coming Up ...

- Speculation in software
- **Retaining lock semantics & behaviour**
- Implementation and evaluation
- Interfacing to the run-time

Semantics

- Programmers should see the same semantics with SLE as when using locks
- This means:
  - Lock acquisition must be allowed
  - No constraints on memory recycling
- Solve this via insertion of Safe() calls:

  Safe(O): **while(metadata(O) is locked) wait;**
- We also want to ensure there's no unexpected (i.e. additional) blocking on other threads
  - Safe(O) must not wait for any other thread

Semantics – Application Locks

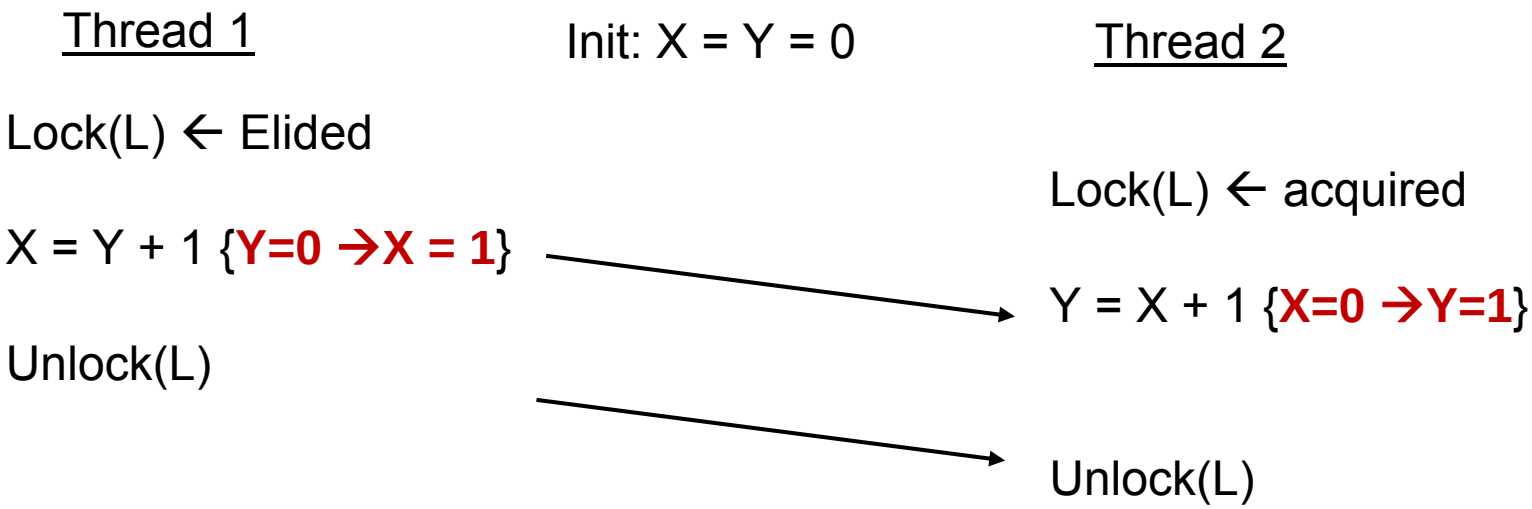
- Acquisition of critical section locks
- Need to reconcile with speculating threads

Thread 1	Init: X = Y = 0	Thread 2
Lock(L) ← Elided X = Y + 1 Unlock(L)		Lock(L) ← Acquired Y = X + 1 Unlock(L)

Can X == Y ?

Semantics – Application Locks

- Acquisition of critical section locks
- Need to reconcile with speculating threads



X == Y == 1 !!!

Semantics – Application Locks

Roy et al: *Brief Announcement: A Transactional Approach to Lock Scalability*… SPAA'08

- Basic idea: add a version number to locks
- Lock is a shared memory object

  Lock(L) → Lock(L) ; version(L)++

  Unlock(L) → Version(L)++; Unlock(L)

  Elide (L) → L.version even: Log (L.version)
- Check for non speculative access
  - Use Safe(O) as defined before
- Additional complexity to handle reader locks
- No information required about other threads

Semantics – Privatisation

- Memory no longer protected by a lock

Thread 1

```
Lock(L) ← Elided
node = List_head(list)
node.value = 42
Unlock(L)
```

Thread 2

```
Lock(L) ← Elided
node = List_head(list)
List_delete(node)
Unlock(L)
free (node)
```

Semantics – Privatisation

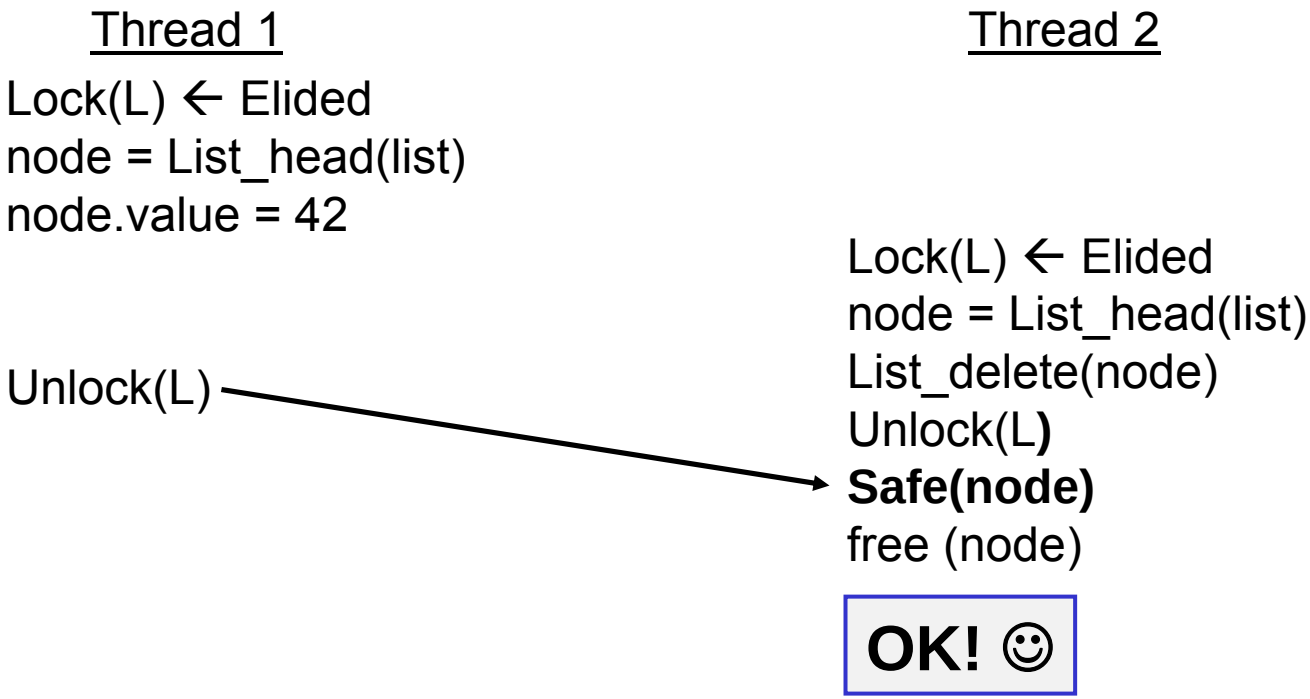
- Memory no longer protected by a lock

Thread 1	Thread 2
Lock(L) ← Elided	
node = List_head(list)	
node.value = 42	
	Lock(L) ← Elided
	node = List_head(list)
	List_delete(node)
	Unlock(L)
	free (node)
Unlock(L) →	 **Memory corruption**

- Unmanaged environment→ no Garbage Collector

Semantics – Privatisation

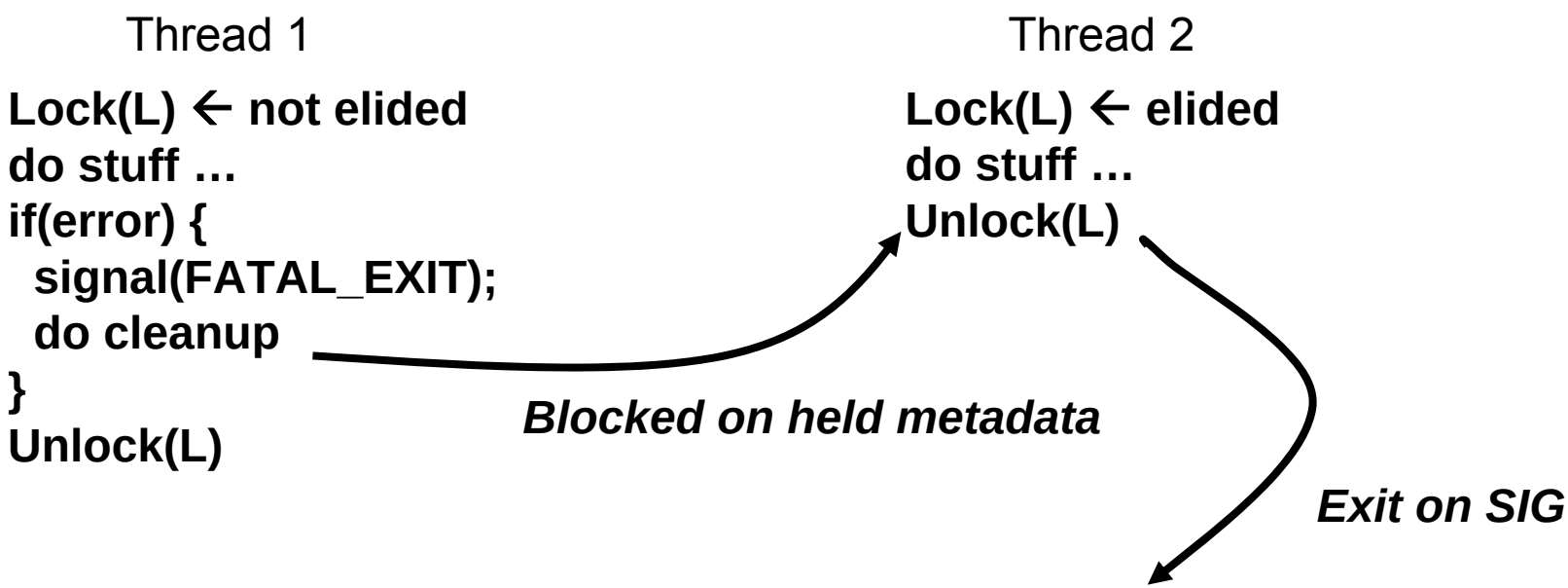
- Memory no longer protected by a lock



Semantics – Avoiding Blocking

- Locked metadata blocks non-speculative threads
- Execution behaviour changes:
  - Can block on other threads even if not at Lock(L)

Example from Apache webserver



Semantics – Avoiding Blocking

Harris et al: *Revocable Locks for Non-Blocking Programming* … PPoPP'05

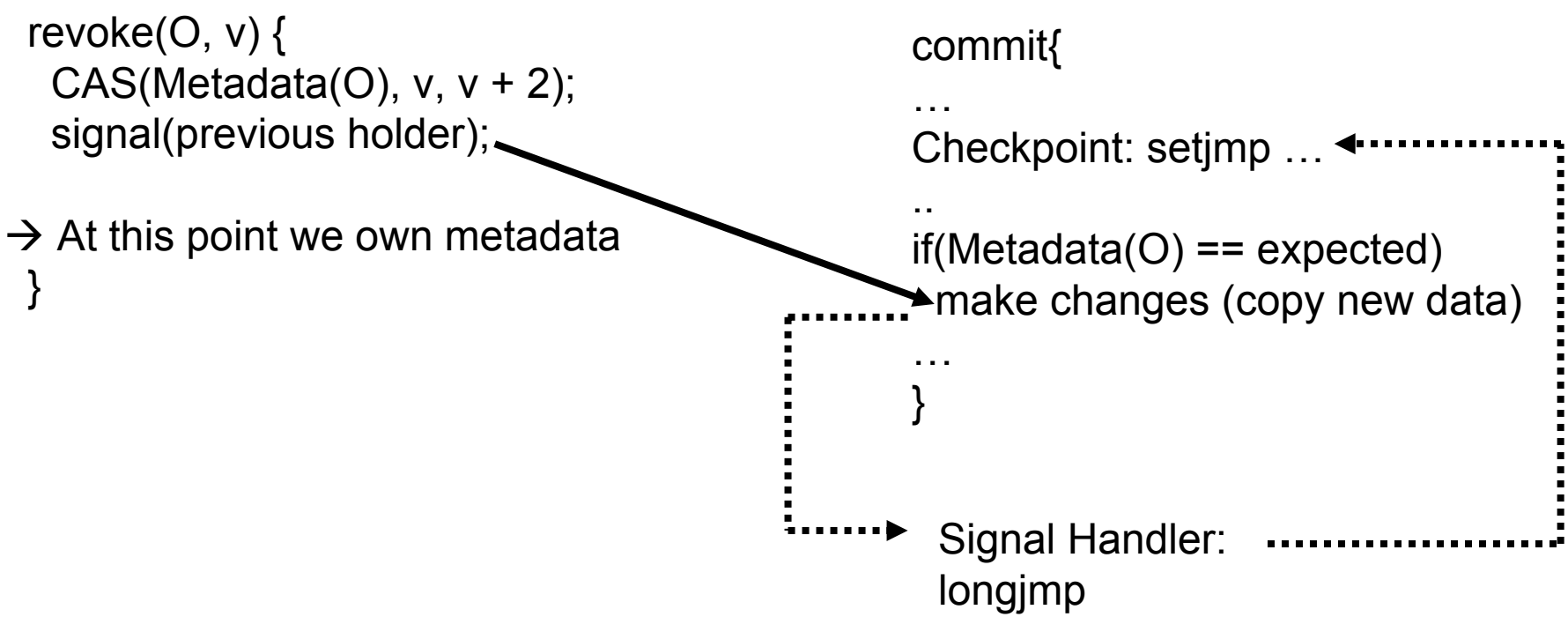
- We use **revocable locks**:
  - Allow lock to be revoked, displacing lock holder's execution to a special cleanup path
  - Call revoke(O, v) if Safe(O) finds O locked at version v

```
revoke(O, v) {
  CAS(Metadata(O), v, v + 2);
  signal(previous holder);

→ At this point we own the metadata
}
```

```
commit{
…
Checkpoint: setjmp …
..
if(Metadata(O) == expected)
  make changes (copy new data)
…
}
```

Semantics – Avoiding Blocking



Semantics – Avoiding Blocking

```
revoke(O, v) {
  CAS(Metadata(O), v, v + 2);
  signal(previous holder);

  → At this point we own the lock
}
```

```
commit{
…
Checkpoint: setjmp …
..
if(Metadata(O) == expected)
make changes (copy new data)
…
}
```

Signal Handler:
longjmp

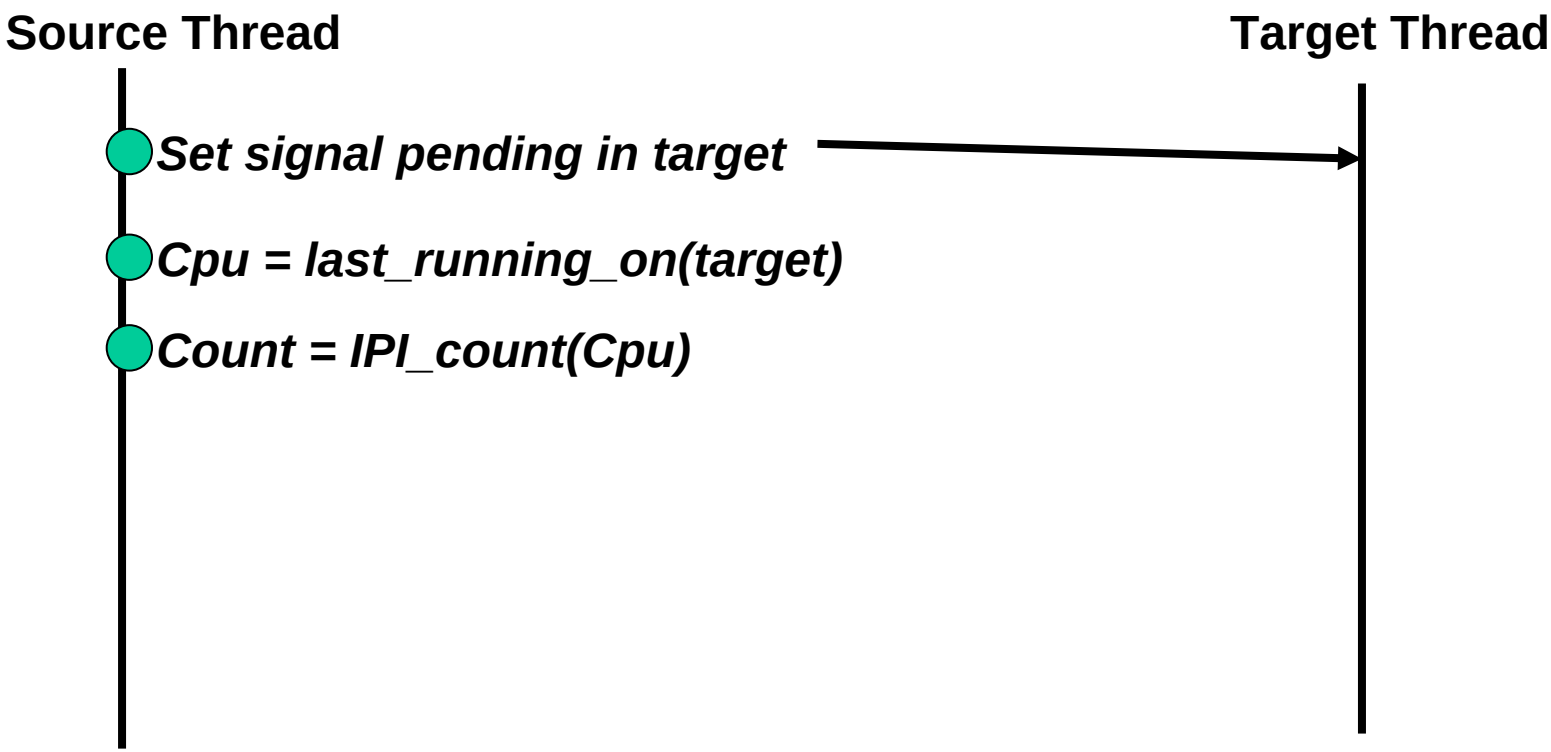
How to synchronously signal ?
We use a custom signalling service implemented as a kernel module

Semantics – Avoiding Blocking

- Problem: we know nothing of target thread state
  - Can send an inter-processor interrupt (IPI)
  - Signal delivery on return to userspace

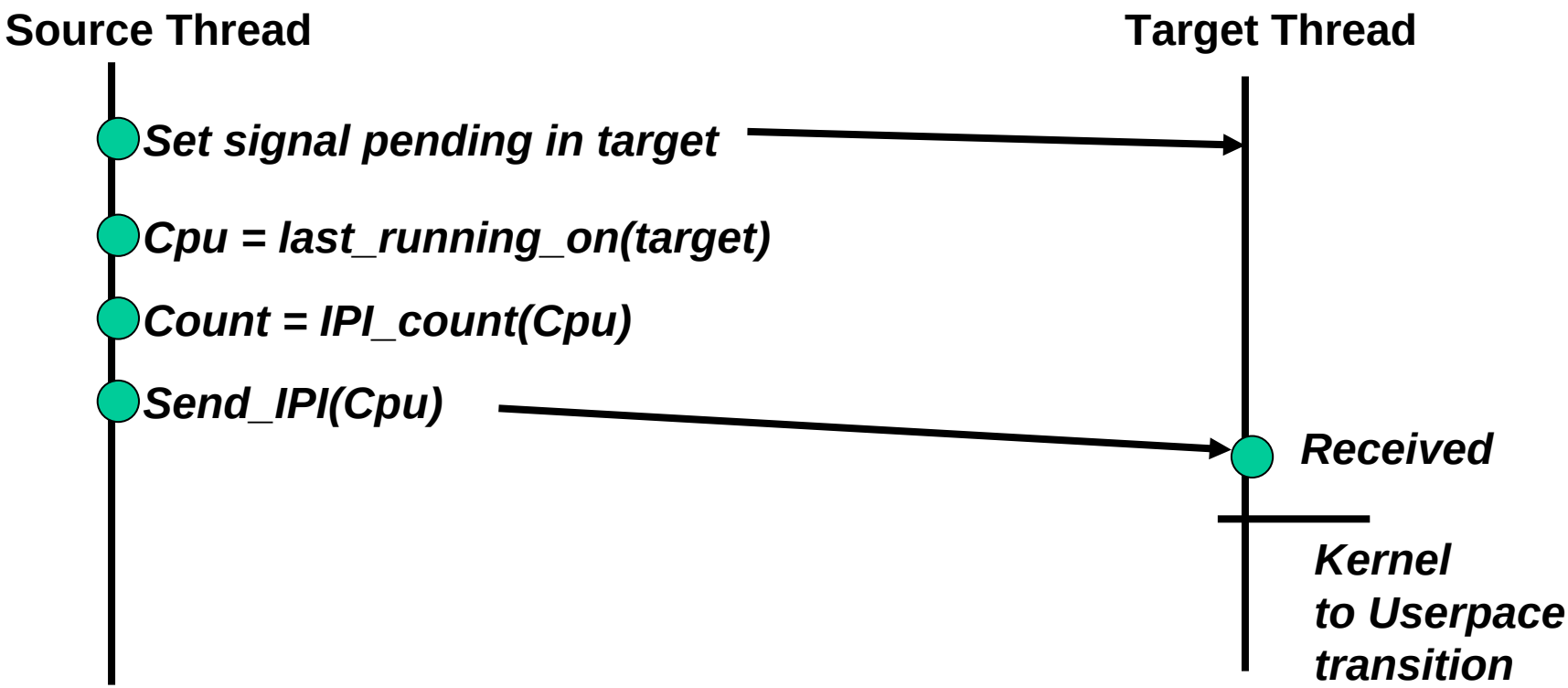
Semantics – Avoiding Blocking

- Problem: we know nothing of target thread state
  - Can send an inter-processor interrupt (IPI)
  - Signal delivery on return to userspace



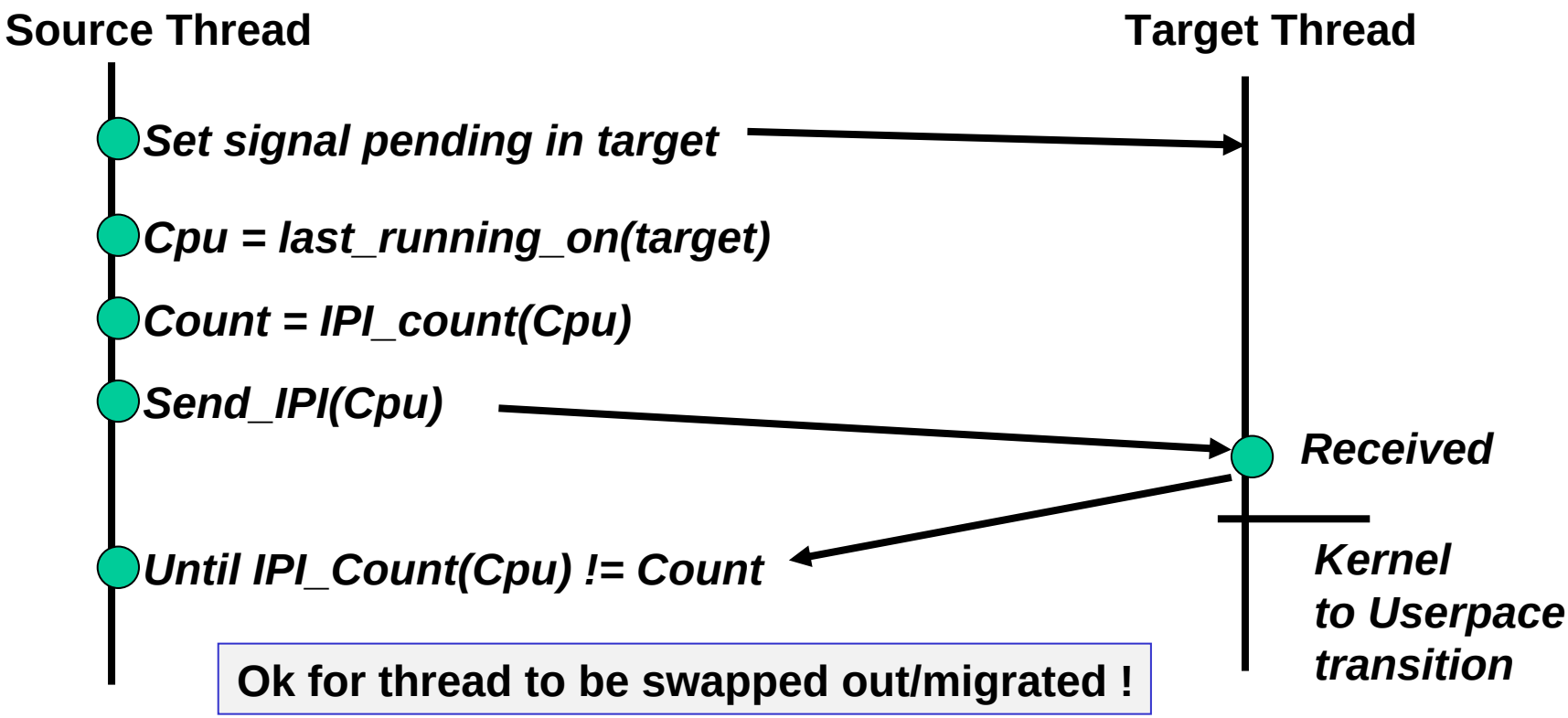
Semantics – Avoiding Blocking

- Problem: we know nothing of target thread state
  - Can send an inter-processor interrupt (IPI)
  - Signal delivery on return to userspace



Semantics – Avoiding Blocking

- Problem: we know nothing of target thread state
  - Can send an inter-processor interrupt (IPI)
  - Signal delivery on return to userspace



Coming Up ...

- Speculation in software
- Retaining lock semantics & behaviour
- **Implementation and evaluation**
- Interfacing to the run-time

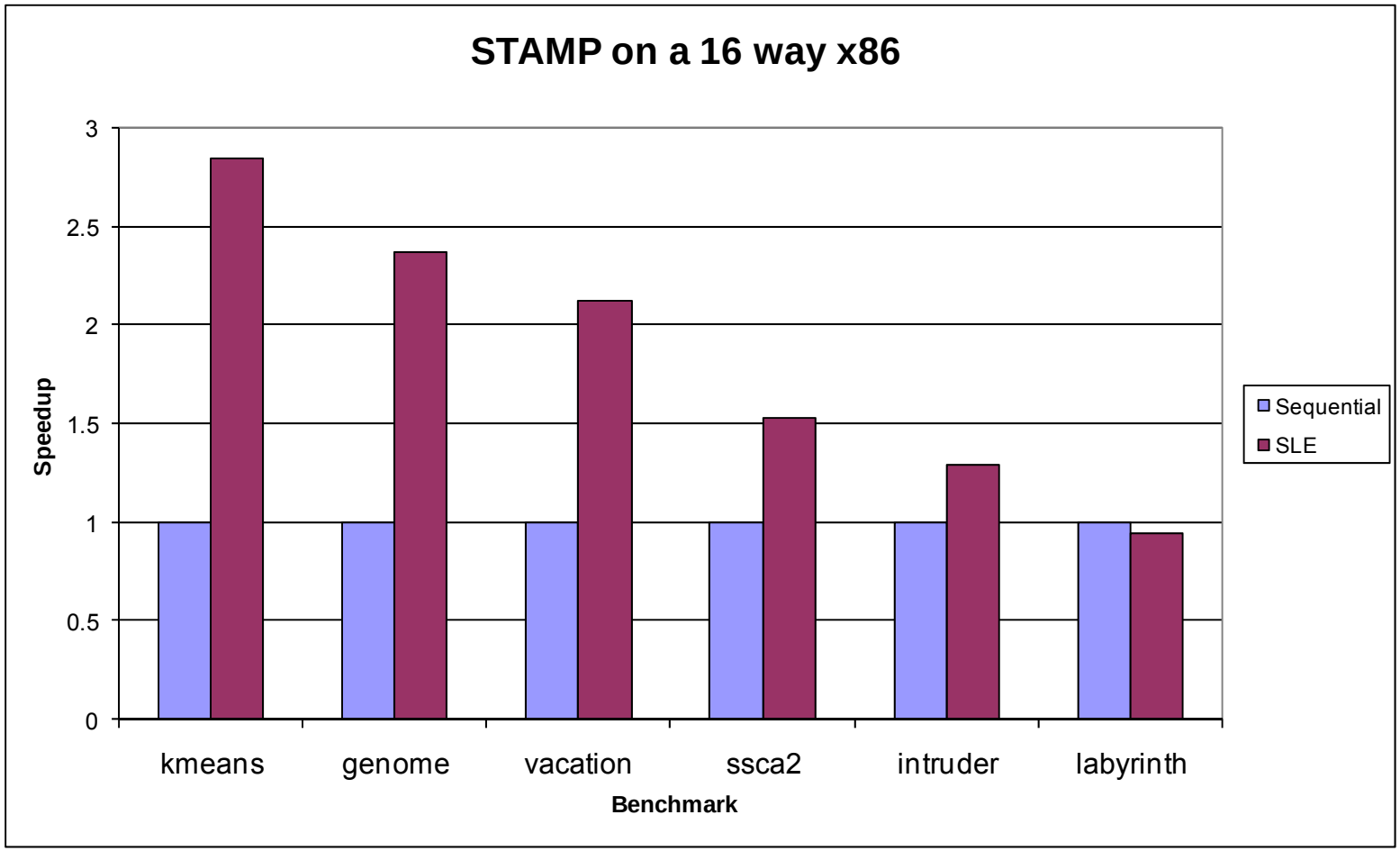
Implementation

- Runtime ~ 2000 lines of C code
  - x86 and Itanium
  - Targets C/C++ Applications
- Extra features
  - Variable sized objects
  - Version number embedded in objects
  - Hash index
- Per lock tuning parameters
  - Control cost of hash indexing
  - Control optimism

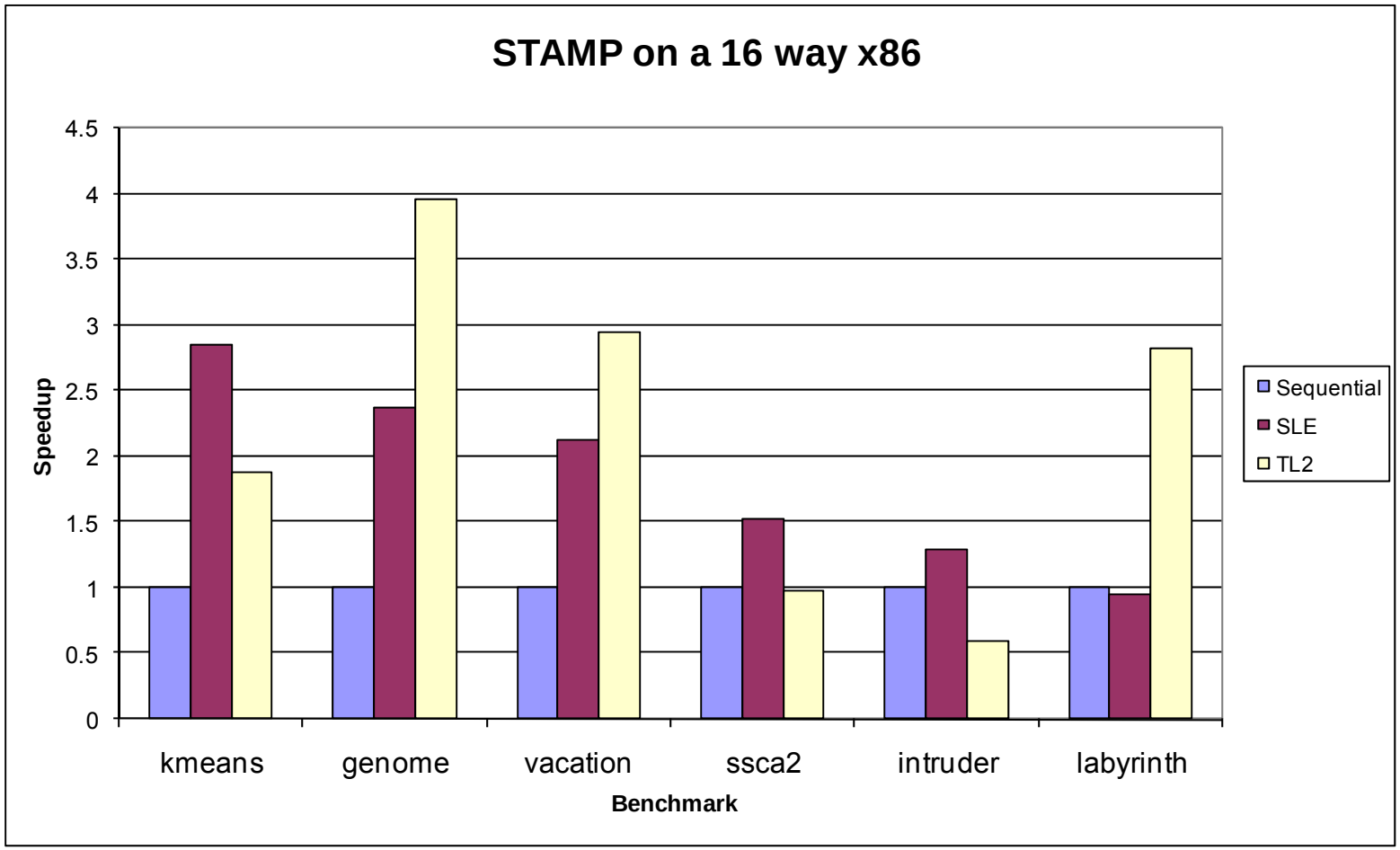
Evaluation

- Performance
  - SLE removes synchronisation bottlenecks
- Design goals
  - Preserve blocking behavior

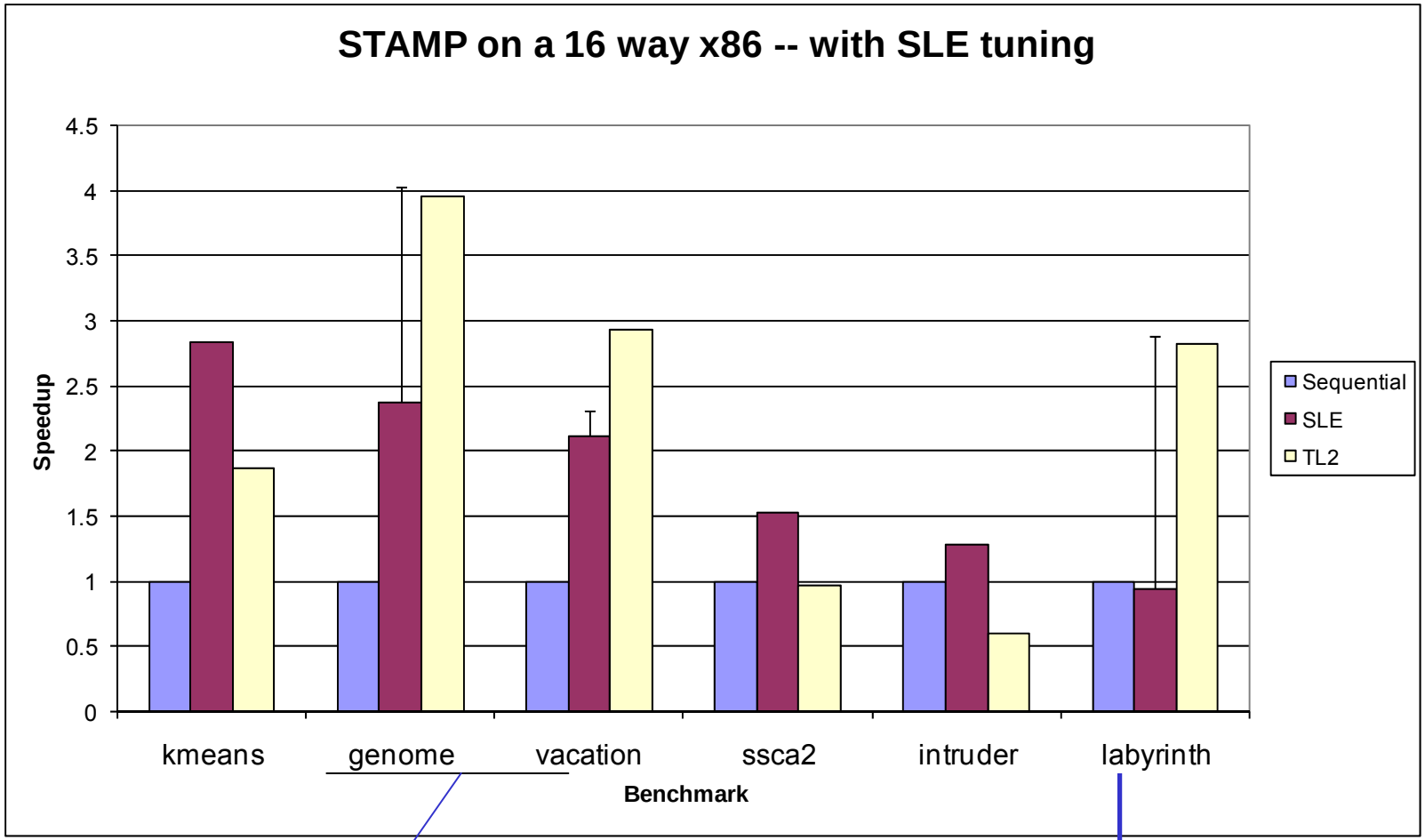
STAMP


STAMP on a 16 way x86
Speedup
3
2.5
2
1.5
1
0.5
0
kmeans
genome
vacation
ssca2
intruder
labyrinth
Benchmark
Sequential
SLE

STAMP


STAMP on a 16 way x86
Speedup
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
kmeans
genome
vacation
ssca2
intruder
labyrinth
Benchmark
Sequential
SLE
TL2

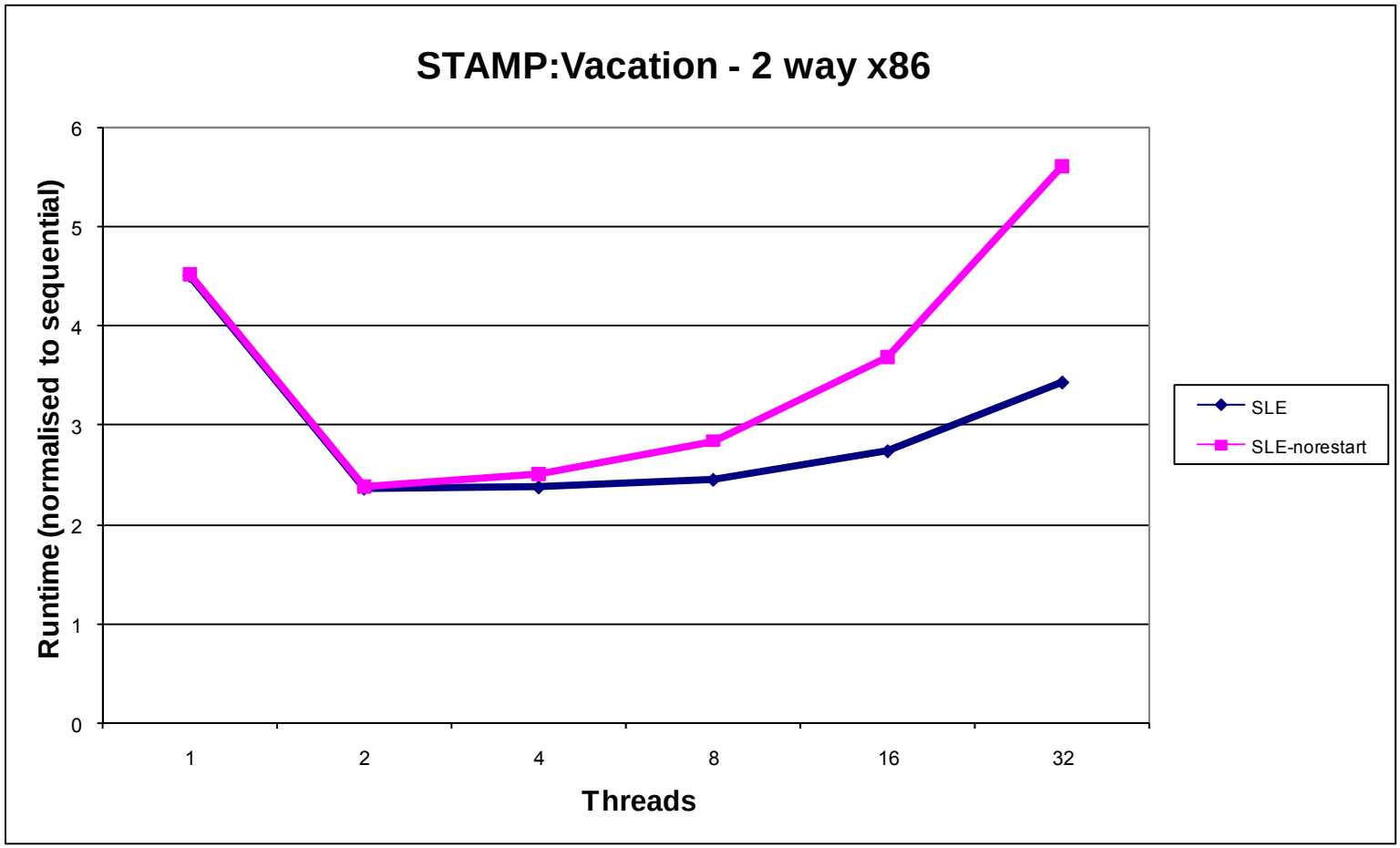
STAMP



With larger hash

Fix hash function to be alignment agnostic

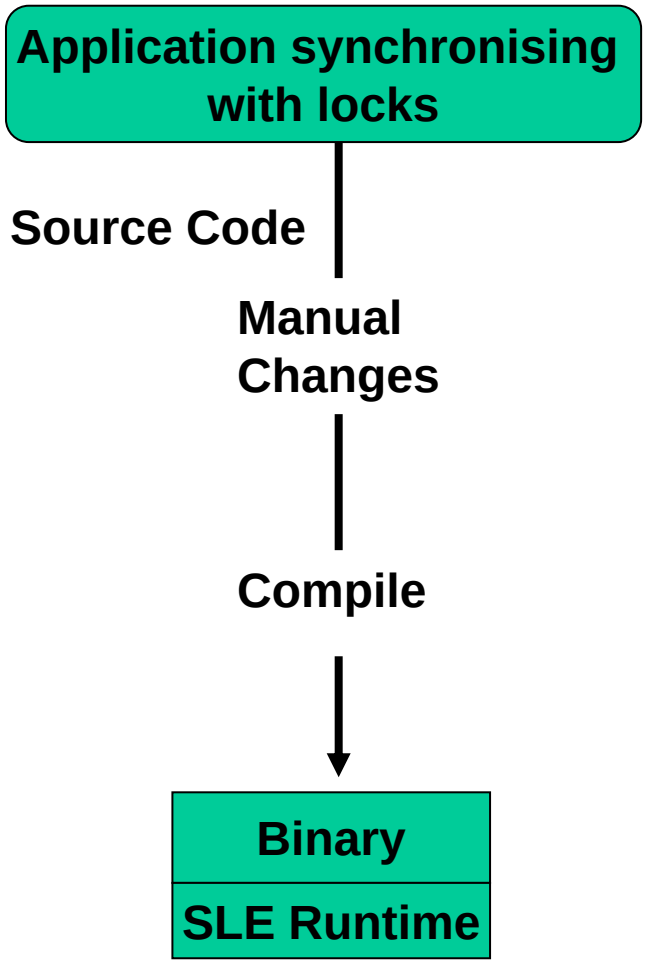
Multiprogramming



Coming Up ...

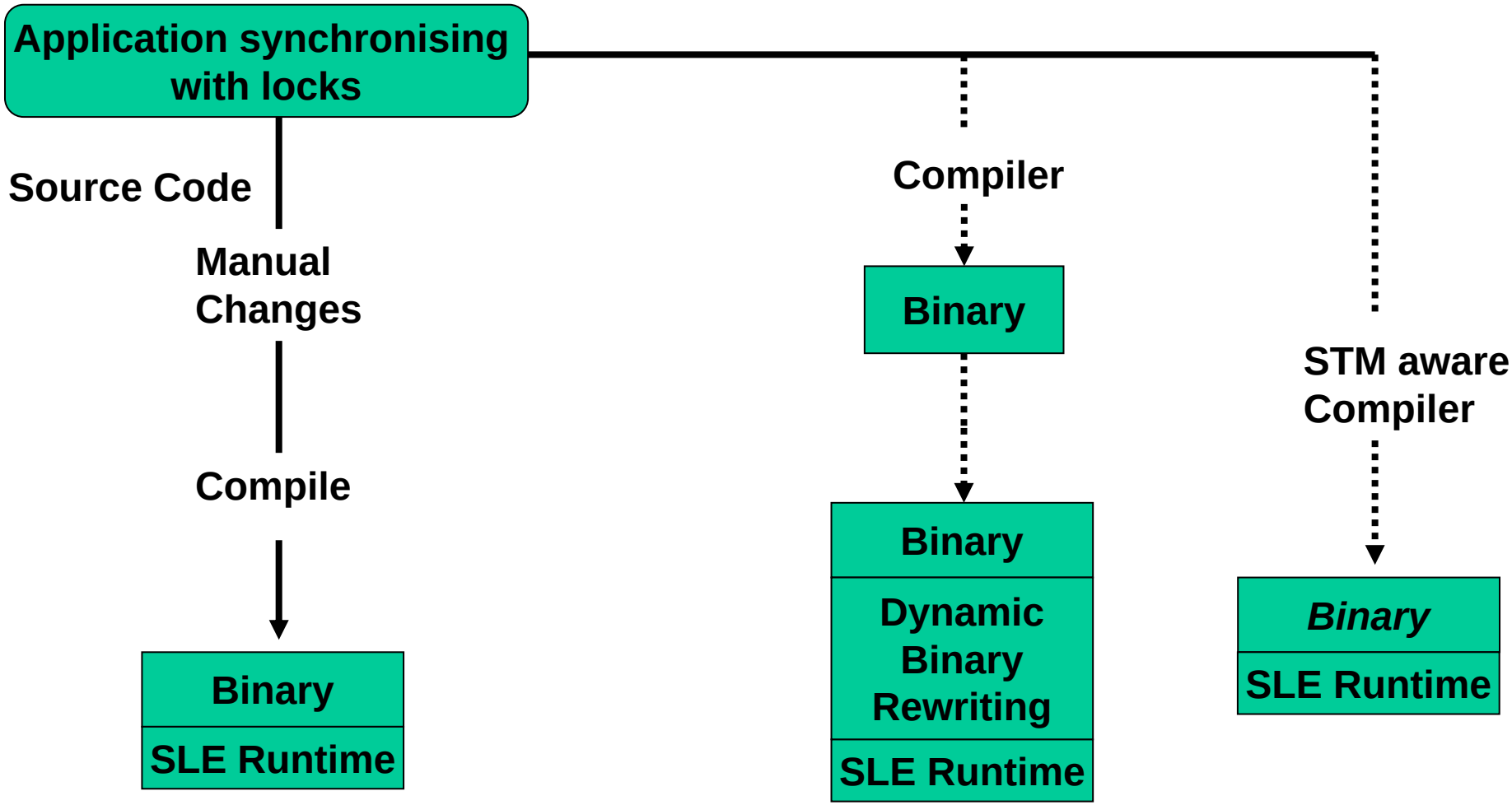
- Speculation in software
- Retaining lock semantics & behaviour
- Implementation / evaluation
- **Interfacing to the run-time**

Programmer Interface

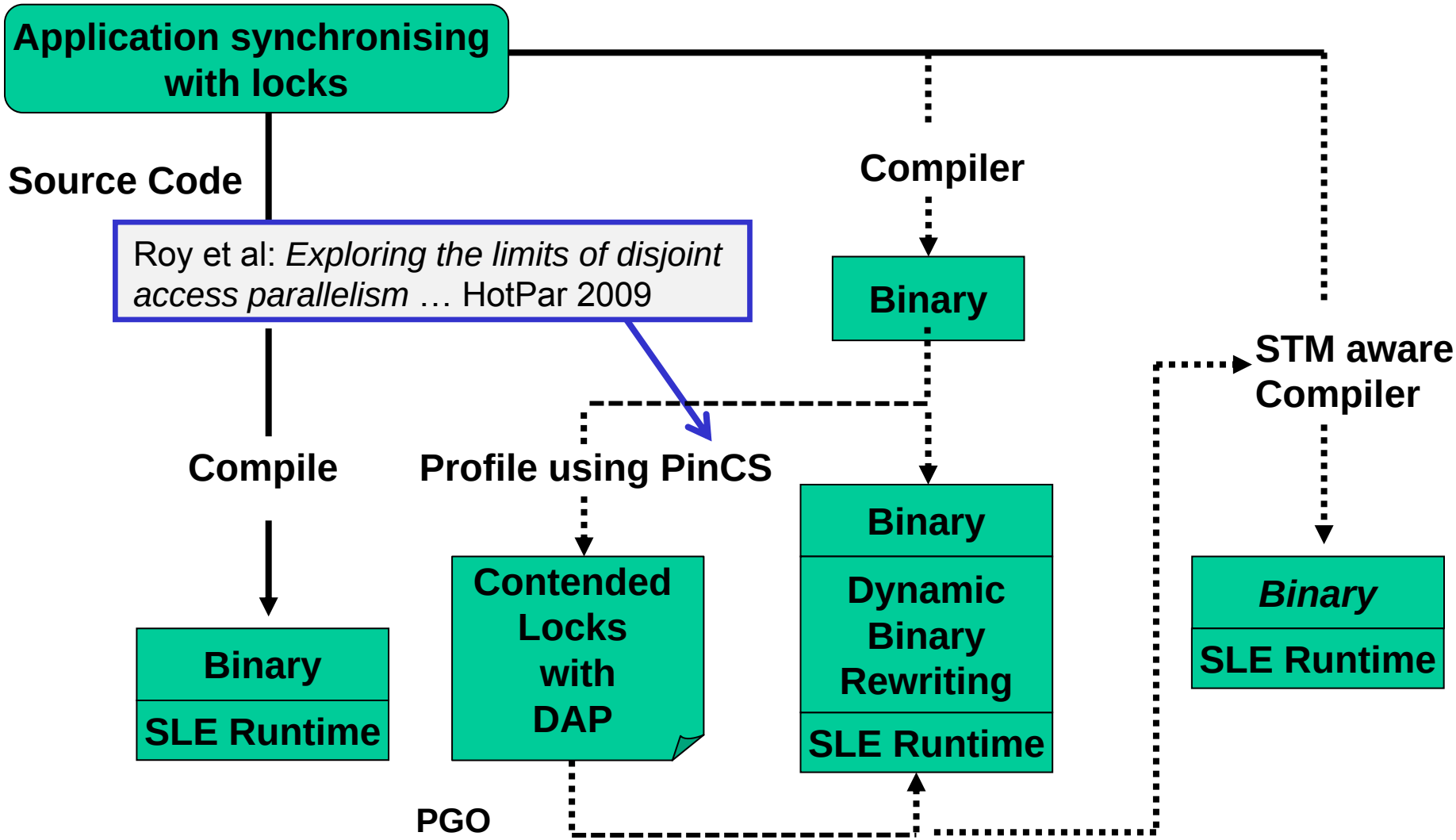


- Manual placement of calls into SLE runtime
  - For declaring and acquiring locks
  - For thread private copies
  - For privatisation

Future Work: Automation



Future Work: Profiling



Conclusion

- Software Lock Elision
  - Off the shelf microprocessors
  - STM to manage speculation
- Retain semantics of locks
  - STM reconciles with locks
  - Block only when lock is held
- Revocable locks in software

Backups

Atomic Blocks

- Atomic blocks ≠ transactional memory

  Just one of the (very popular) ways to expose transactions to the programmer

- Lock Elision subsumes atomic blocks

  Atomic{ } ==

  Lock(big global lock) { } Unlock(big global lock)

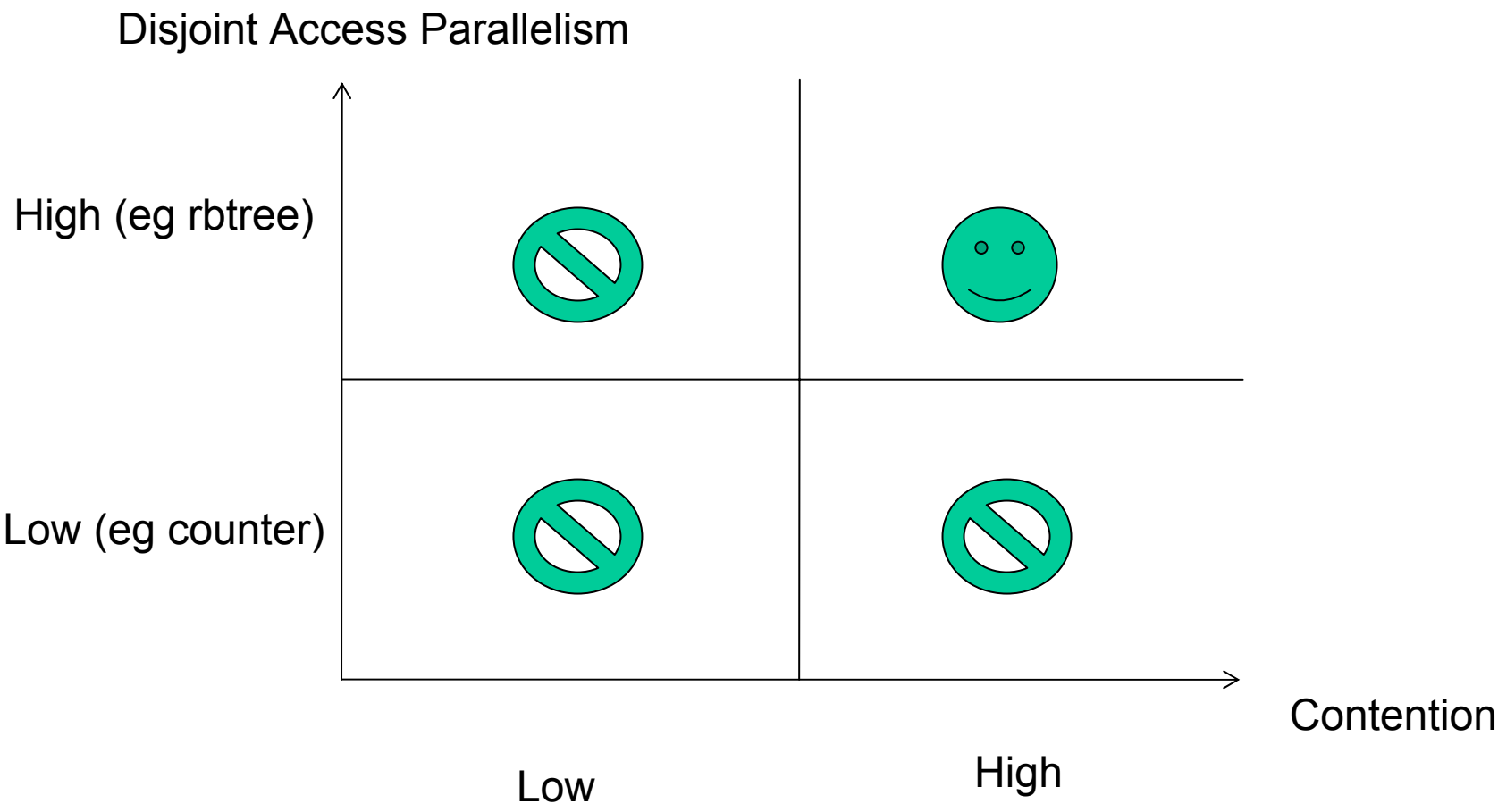
  Could easily build atomic blocks over SLE

  Approach followed for evaluations with STAMP

Related Work

- Welc et al ECOOP 2008
  - Combine monitors and transactions in Java
  - Use the GC in the Java runtime to get around privatisation problems
  - Do not optimise for reader locks
  - Do not retain blocking semantics
- Rossbach et al SOSP 2007
  - Cxspinlock in the linux kernel
  - Lock elision, depends on HTM but declarative
  - Does not need to solve software specific problems but would only run on a simulator ☺

Suitability for Lock Elision



Pending Signals

- SIGHUP < … < SIGALRM < … SIGUSR1

Semantics – Avoiding Blocking

- Problem: we know nothing of target thread state
  - Can send an inter-processor interrupt (IPI)
  - Signal delivery on return to userspace