Disentangling Software Thread-level Speculation

Clark Verbrugge
clump@cs.mcgill.ca

McGill

16th Workshop on Compiler-Driven Performance
November 8, 2017

Credits

- Christopher J.F. Pickett, PhD
  - Java (SableSpMT), MLS, RVP
- Zhen Cao, PhD
  - C/C++/Fortran (MUTLS), opts

- MSc: Alexander Krolik, Haiying Xu
- IBM: Allan Kielstra

- Dozens of other TLS & TLS-related projects around the world!!

Basic Idea

- Runtime optimization for sequential progs
- Use otherwise idle cores (threads)

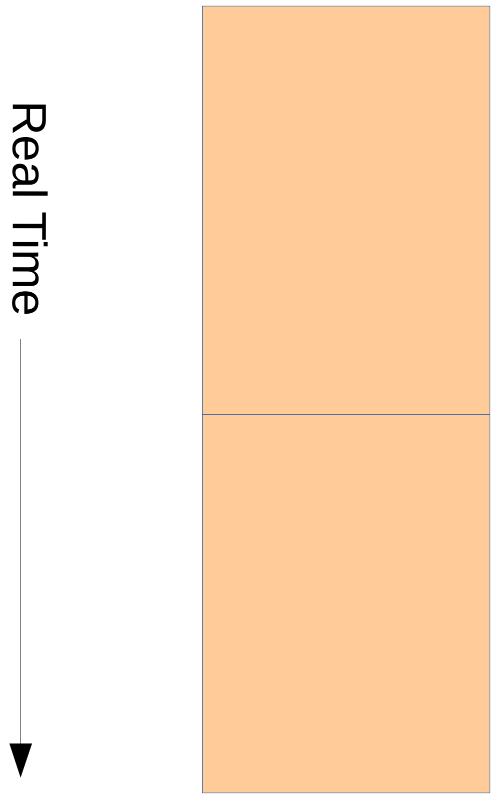
- ***Fork*** (spawn) threads to execute in the future
- New threads start at the ***Join*** point
  - Code between Fork-Join, and after Join executed in parallel

Basic Idea

- E.g.

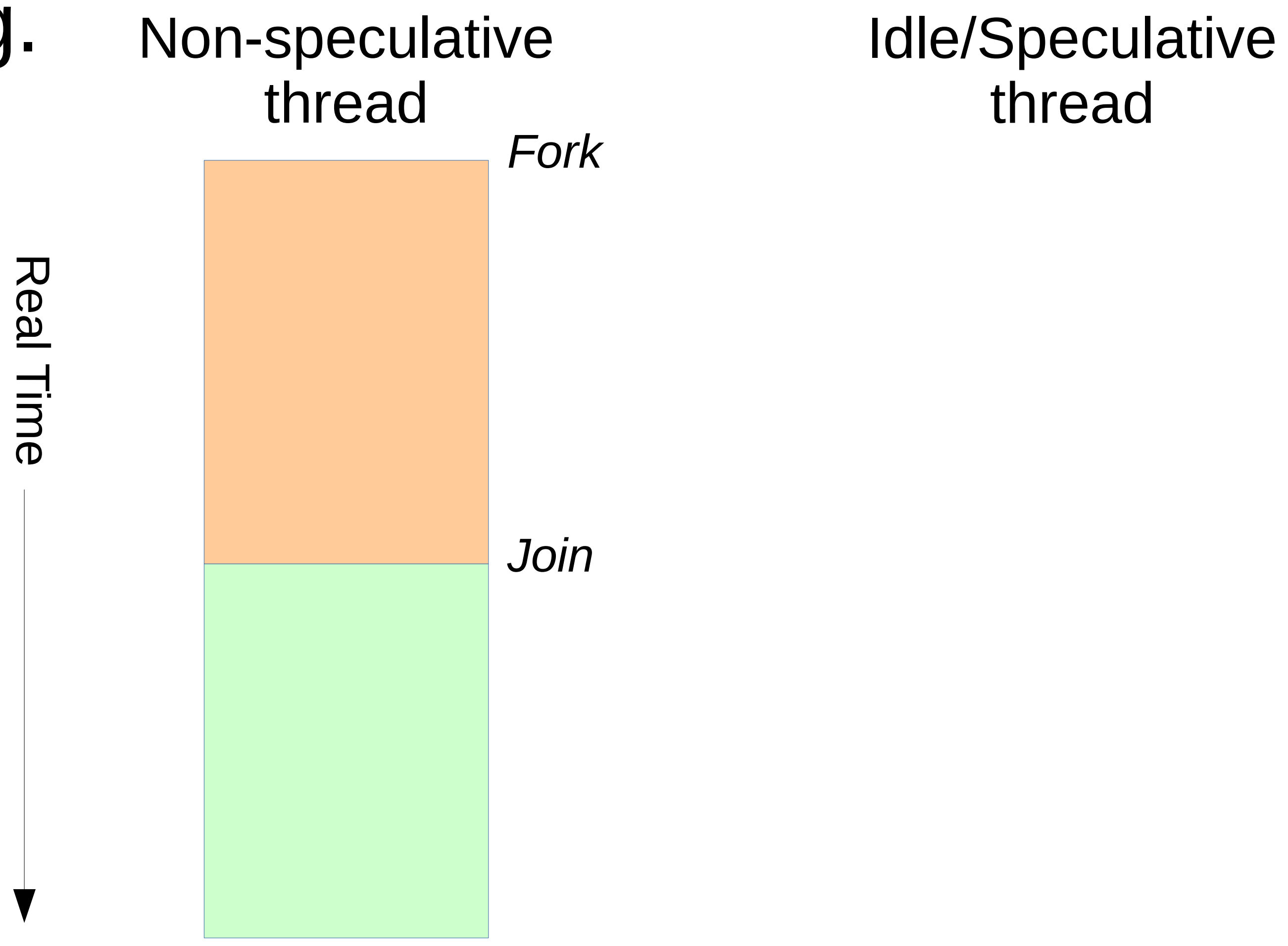
Non-speculative thread

Idle/Speculative thread



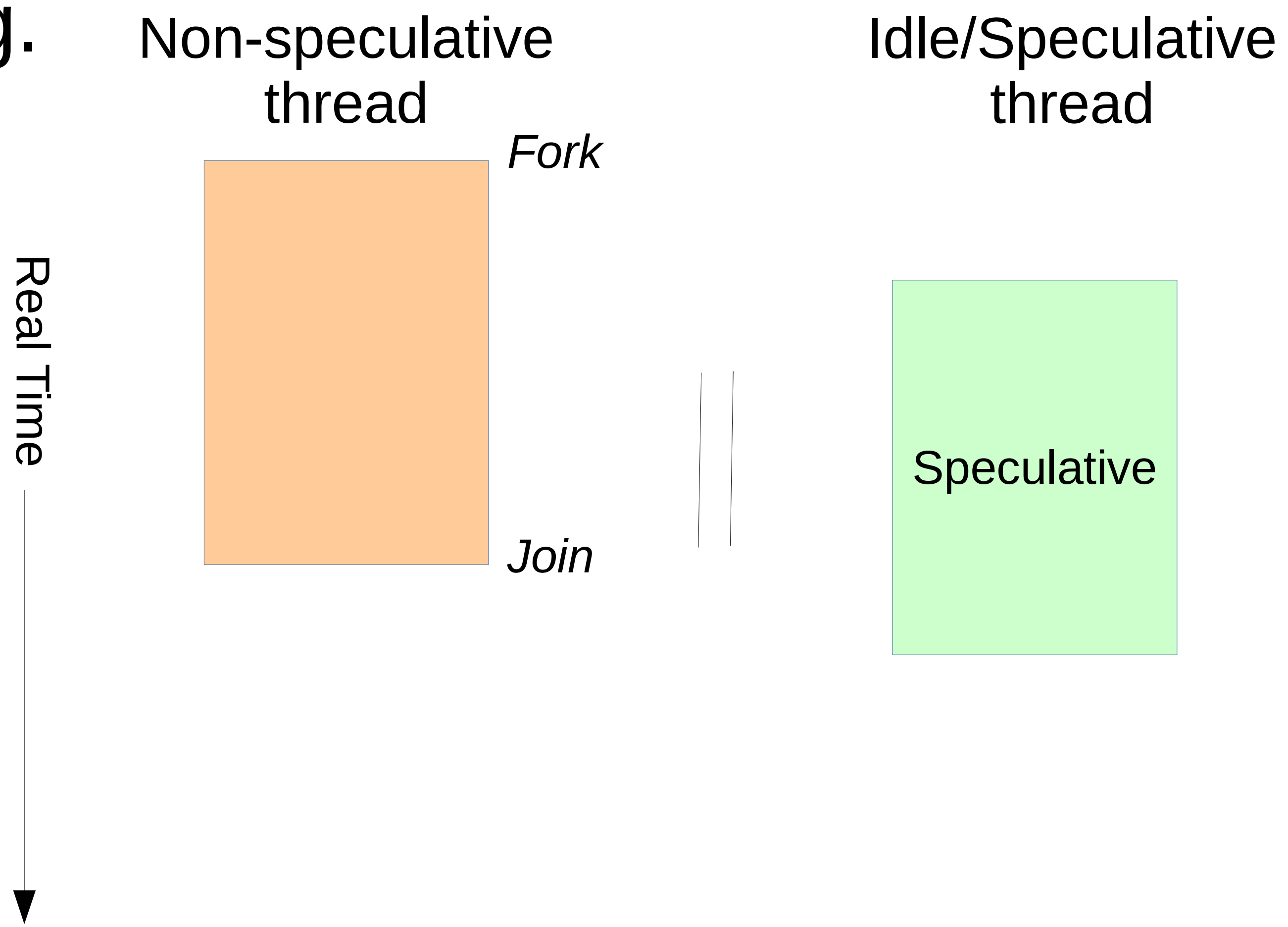
Basic Idea

- E.g.



Basic Idea

- E.g.

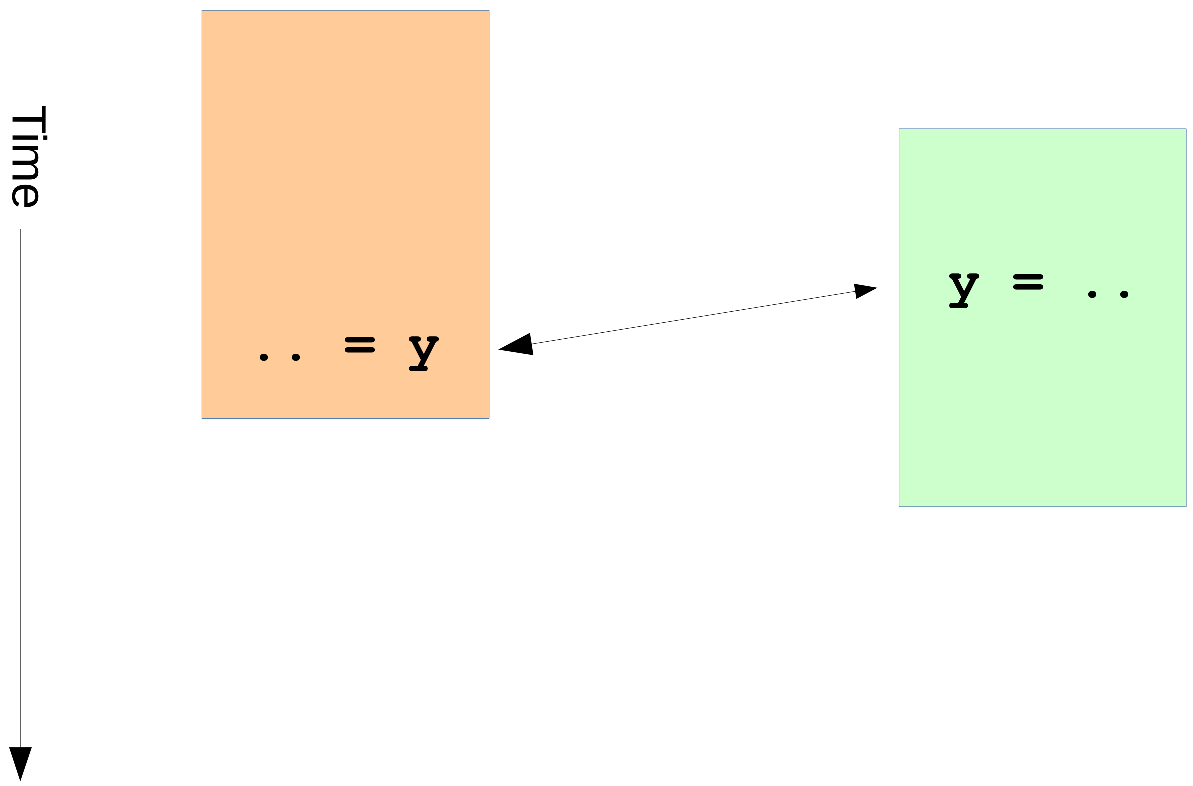


Basic Idea

- Speculation
  - Not sure of program state (data) in the future
  - Not sure of exact control flow in the future
- Safety required!
  - Guarantee equivalence to sequential execution

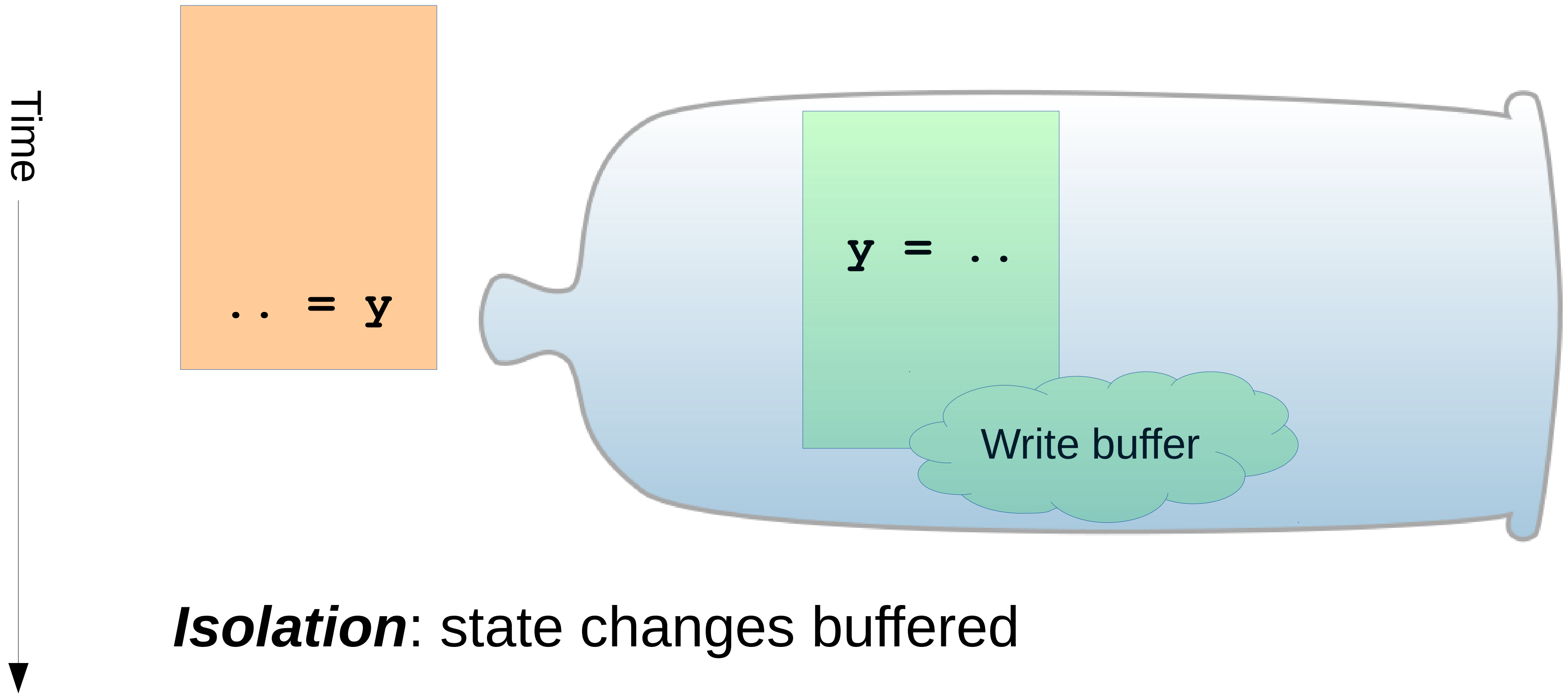
Basic Idea

- Safety: future should not affect the past



Basic Idea

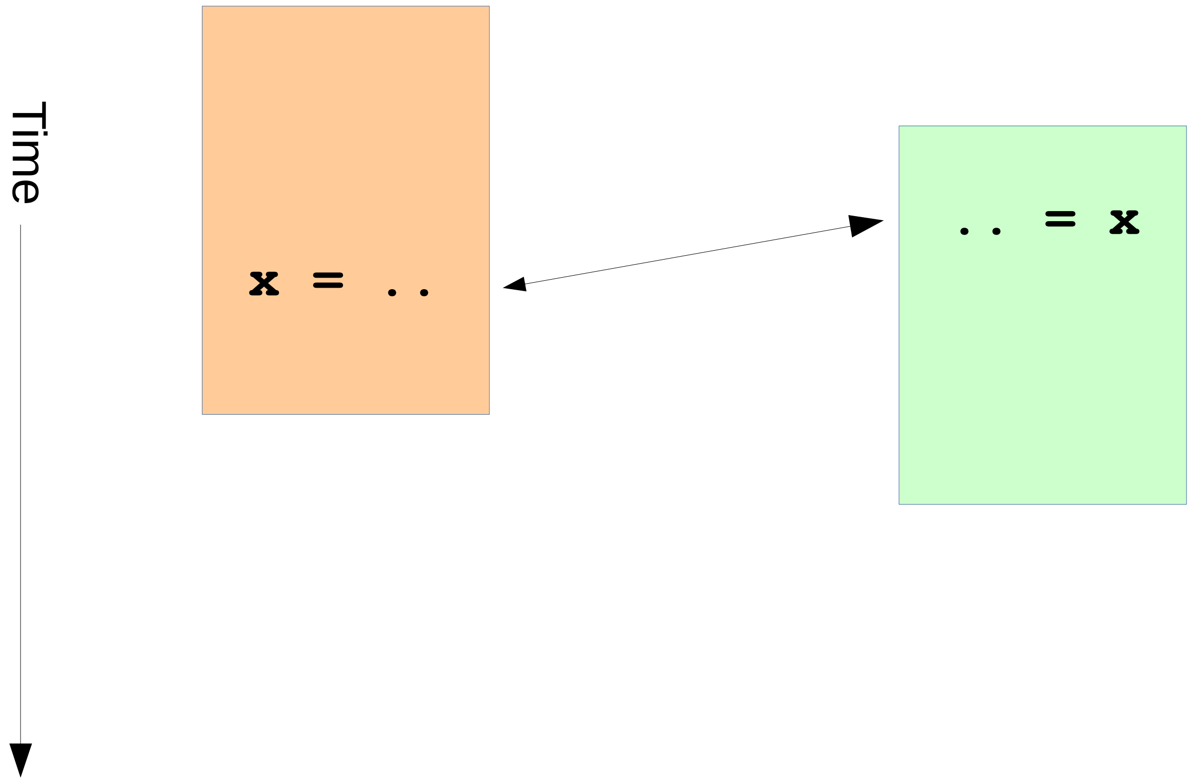
- Safety: future should not affect the past



Isolation: state changes buffered

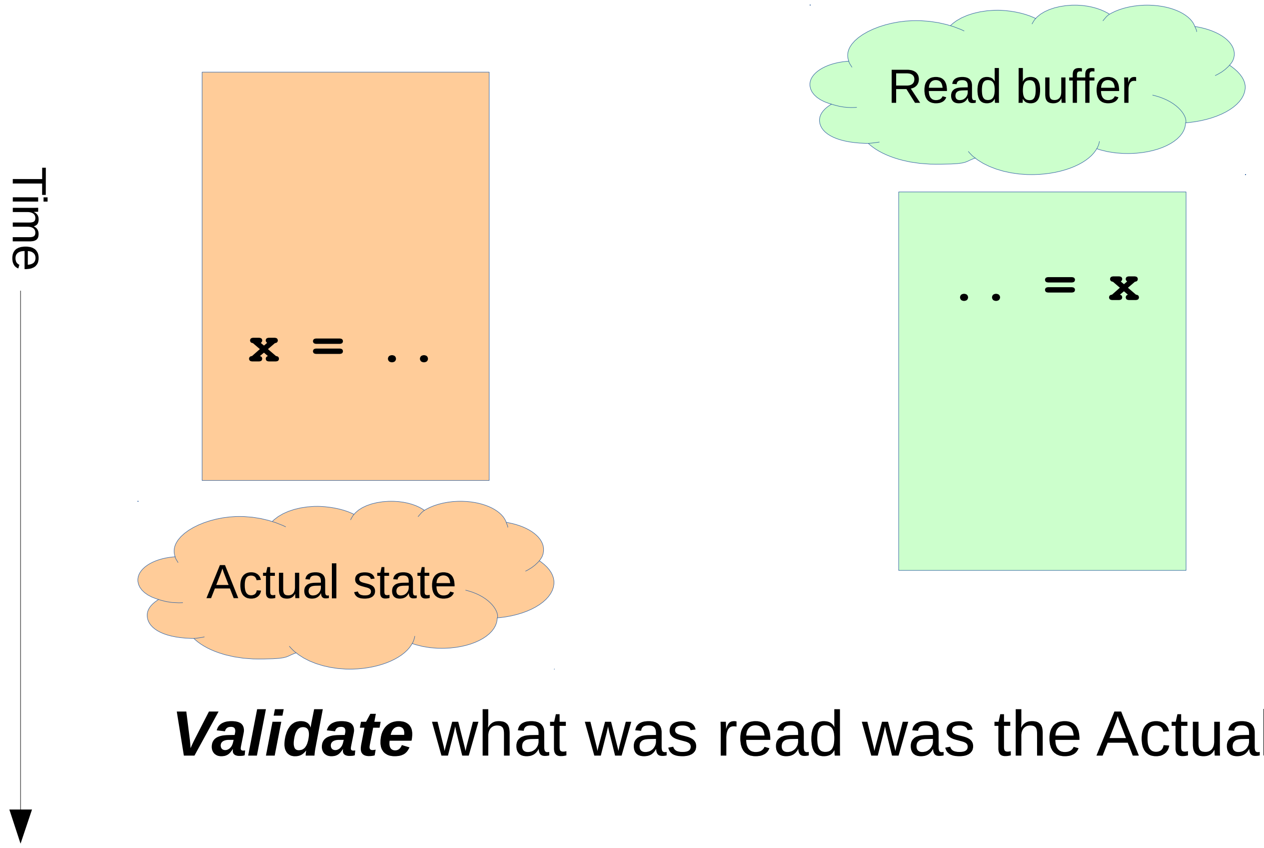
Basic Idea

- Safety: past should affect the future



Basic Idea

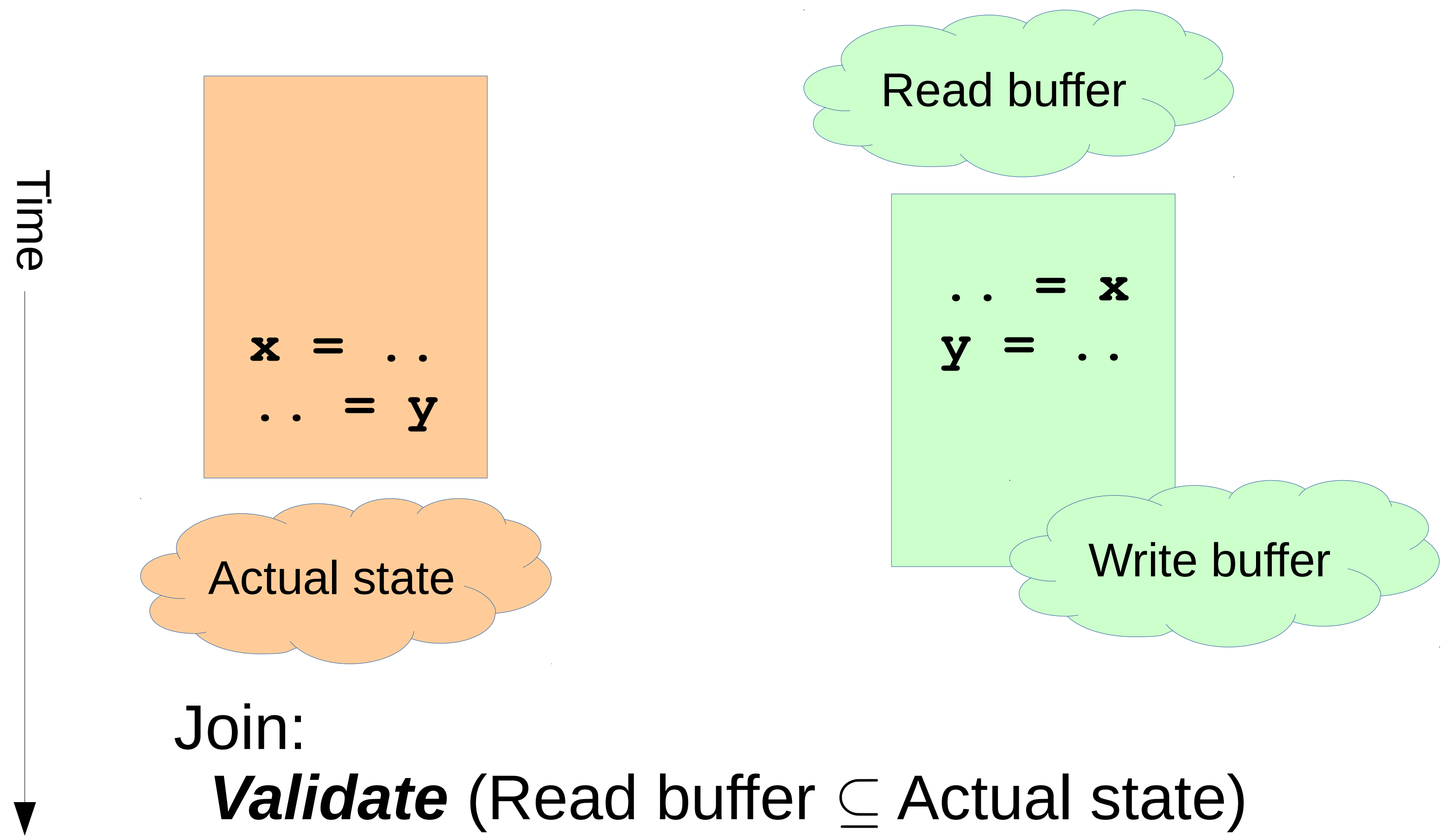
- Safety: past should affect the future



Validate what was read was the Actual State

Basic Idea

- Both



Join:
__Validate__ (Read buffer $\subseteq$ Actual state)
If so *__commit__* (Write buffer $\Rightarrow$ Actual state)
otherwise *__abort__* and discard speculation

Performance Factors

- Overhead in each step
  - Hardware support helps!
- *Misspeculation* (abort) is bad
  - Reduces parallelism
  - Wastes time

Design

- Various designs since 1990s [Franklin, “Multiscalar” 1993]
  - Mainly hardware [Jrpm, STAMPede, Mitosis, ...]
  - But some software
- Differ in opts, assumptions, benchmarks
  - Research focus on misspeculation, overhead
- Comparison obscured by design choices

Choices, Choices, Choices

- Speculative Targets
- Thread Model
- Versioning Model
- Performance Model

Speculative Targets

- Core constraint
  - Fork > join
- Natural program divisions
  - Methods
  - Loop bodies
- Arbitrary
- (All equivalent via code-transformation)

Method-level Speculation

- Suitable for method-rich contexts
  - OO programs
    - Java: [Chen & Olukuton, 1998]
- Fork: call-site
- Join: continuation

e.g., SableSpMT

Method-level Speculation

```
foo();
.
.
.
```

```
foo() {
  . . .
  . . .
}
```

Method-level Speculation

Fork
Join

```
foo();          foo() {
.                 ...
.                 ...
.               }
```

Method-level Speculation

Fork
V/C

```
foo();            foo() {
.                   ...
.                   ...
.                 }
```

Method-level Speculation

- Suitable for method-rich contexts
  - OO programs
    - Java: [Chen & Olukuton, 1998]
- Fork: call-site
- Join: continuation

- Drawbacks
  - Method returns?

Return Value Prediction

- Guess the return value!
- Consider history, simple patterns, partial state
  - Can be surprisingly accurate
- Predicting is key to MLS [Hu, Bhargava, John, 2003]

Loop-level Speculation

- Lots of work done in loops
  - Scientific programs, C/Fortran
- Fork: loop iteration entry
- Join: loop iteration end (start of next)

e.g., SoftSpec, SpLSC/SpLIP

Loop-level Speculation

```
for (...) {
    .
    .
}
```

```
for (...) {
    .
    .
}
```

```
for (...) {
    .
    .
}
```

Loop-level Speculation

Fork

Join

```
for (...) {
    .
    .
}
            for (...) {
                .
                .
            }
                        for (...) {
                            .
                            .
                        }
```

Loop-level Speculation

Fork

V/C

```
for (...) {
    .
    .
}
          for (...) {
              .
              .
          }
                      for (...) {
                          .
                          .
                      }
```

Loop-level Speculation

- Lots of work done in loops
  - Scientific programs
- Fork: loop iteration entry
- Join: loop iteration end

- Drawbacks
  - Loop-carried dependencies

Choices, Choices, Choices

- Speculative Targets
- Thread Model
- Versioning Model
- Performance Model

Thread Model

- More than 1 speculative thread
- Which thread(s) can speculate?
  - Speculative or non-speculative or both
- How many children?

Thread Model: Forking

- Out-of-order

```
foo() {
    ...
    bar()
    ...
}
```

```
bar() {
    ...
    ping()
    ...
}
```

```
ping() {
    ...
}
```

```
foo() {
```

Thread Model: Forking

- Out-of-order

```
foo() {
    ...
    bar()
S1  ...
}
```

```
bar() {
    ...
    ping()
    ...
}
```

```
ping() {
    ...
}
```

```
foo() {
```

Thread Model: Forking

- Out-of-order

```
foo() {
    ...
    bar()
S1  ...
}
```

```
bar() {
    ...
    ping()
    ...
}
```

```
ping() {
    ...
}
```

```
foo() {
   ↓
bar() {
```

Thread Model: Forking

- Out-of-order

```
foo() {
    ...
    bar()
S1  ...
}
```

```
bar() {
    ...
    ping()
S2  ...
}
```

```
ping() {
    ...
}
```

```
foo() {
  ↓
bar() {
```

Thread Model: Forking

- Out-of-order

```
foo() {
    ...
    bar()
S1  ...
}
```

```
bar() {
    ...
    ping()
S2  ...
}
```

```
ping() {
    ...
}
```

```
foo() {
   ↓
bar() {
   ↓
ping() {
```

Thread Model: Forking

- Out-of-order

```
foo() {
    ...
    bar()
S1  ...
}
```

```
bar() {
    ...
    ping()
S2  ...
}
```

```
ping() {
    ...
}
```

```
foo() {
  ↓
bar() {
  ↑
ping() {
```

Join S2

Thread Model: Forking

- Out-of-order

```
foo() {
    ...
    bar()
S1  ...
}
```

```
bar() {
    ...
    ping()
S2  ...
}
```

```
ping() {
    ...
}
```

```
foo() {          
   ↓          Join S1
bar() {          ↑
   ↑          Join S2
ping() {
```

Thread Model: Forking

- In-order

```
foo() {
    …
    bar()
    …
    ping()
    …
    woot()
    …
}
```

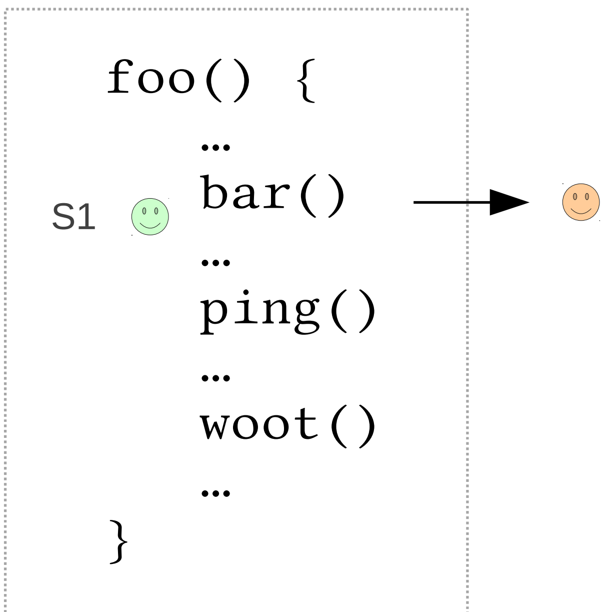
Thread Model: Forking

- In-order

```
foo() {
        …
        bar()
S1
        …
        ping()
        …
        woot()
        …
}
```

Thread Model: Forking

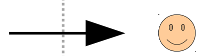
- In-order



Thread Model: Forking

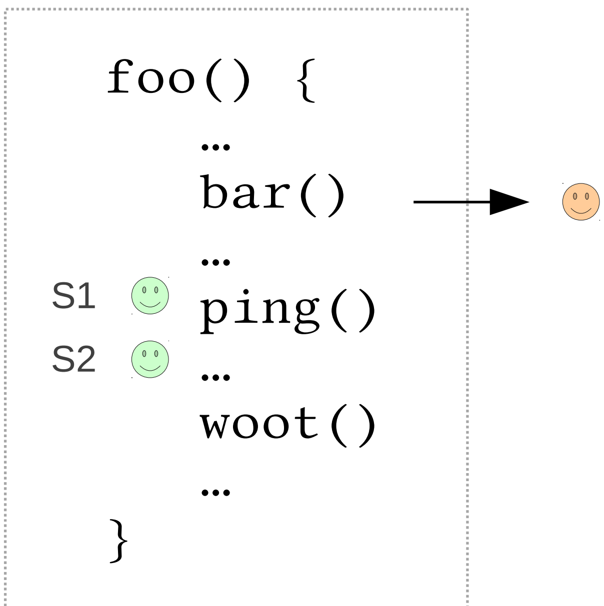
- In-order

```
   foo() {
       …
       bar()
       …
S1     ping()
       …
       woot()
       …
   }
```



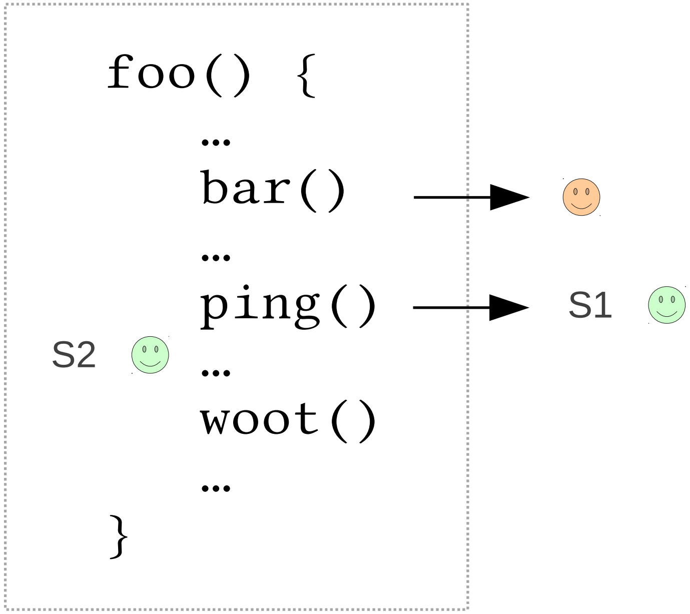
Thread Model: Forking

- In-order



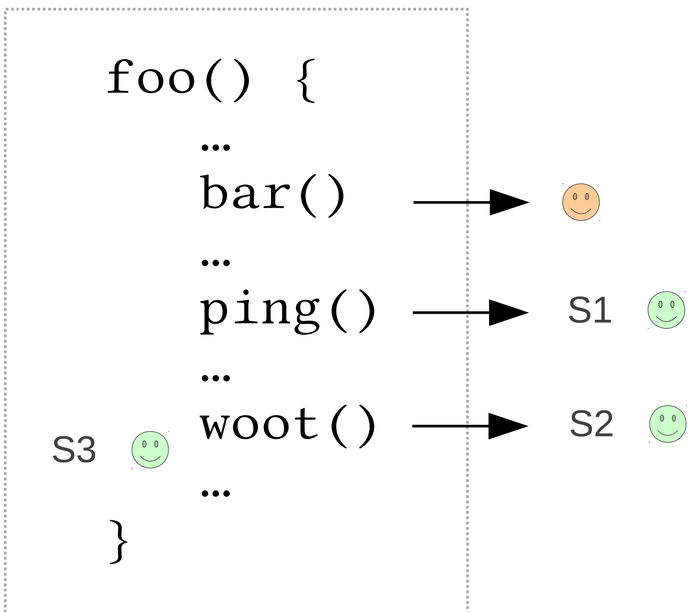
Thread Model: Forking

- In-order



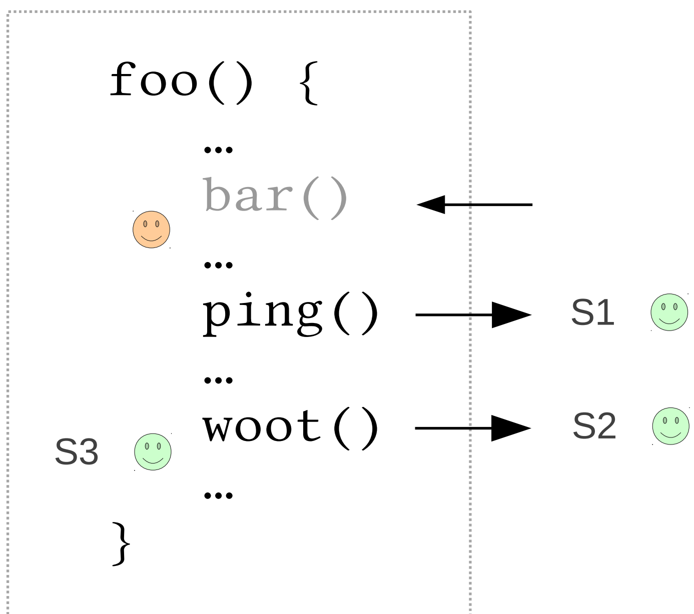
Thread Model: Forking

- In-order



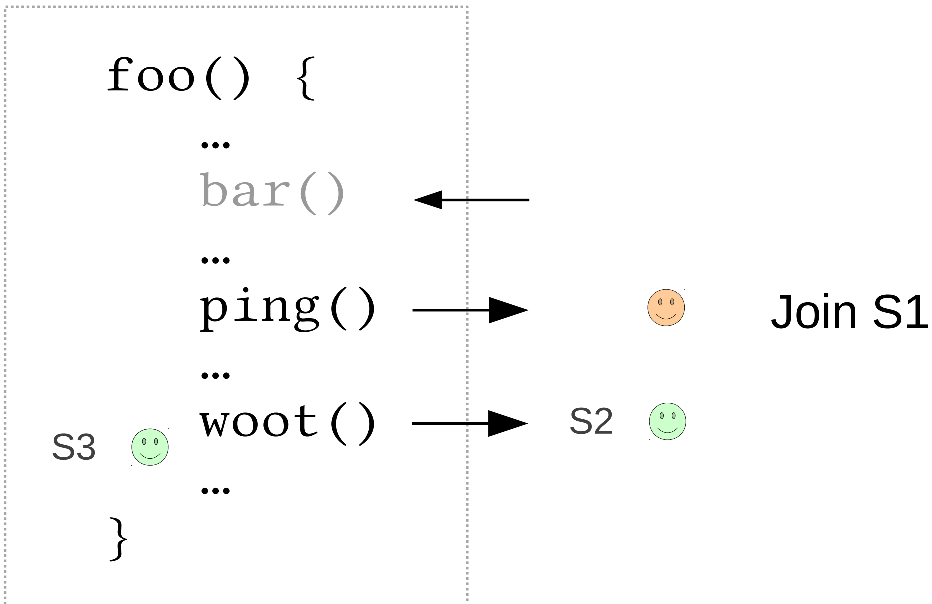
Thread Model: Forking

- In-order



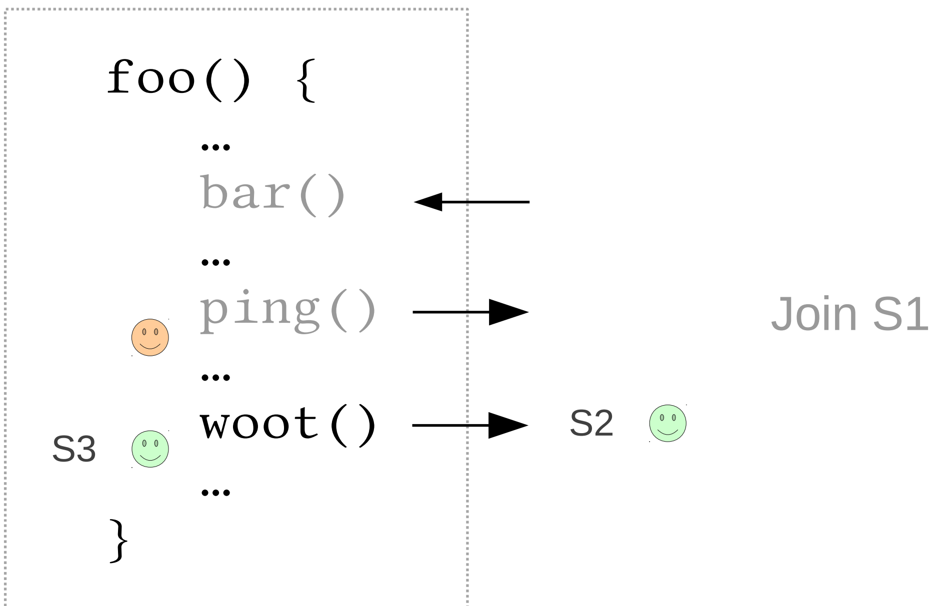
Thread Model: Forking

- In-order



Thread Model: Forking

- In-order



Thread Model: Forking

- In-order

```
foo() {
    …
    bar()
    …
    ping()
    …
    woot()
    …
}
```

S3

S2

Join S1

Join S2

Thread Model: Forking

- Mixed

NS

```
A();
D();
W7
```

```
A() {
  B();
  W3
}
```

```
B() {
  C();
  W2
}
```

```
C() {
  W1
}
```

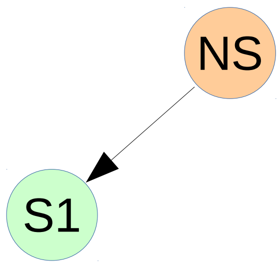
```
D() {
  E();
  W6
}
```

```
E() {
  F();
  W5
}
```

```
F() {
  W4
}
```

Thread Model: Forking

- Mixed




```
A();
D();
W7
```

```
A() {
 B();
 W3
}
```

```
B() {
 C();
 W2
}
```

```
C() {
 W1
}
```

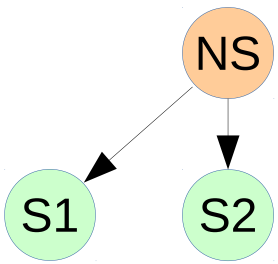
```
D() {
 E();
 W6
}
```

```
E() {
 F();
 W5
}
```

```
F() {
 W4
}
```

Thread Model: Forking

- Mixed




```
A();
D();
W7
```

```
A() {
  B();
  W3
}
```

```
B() {
  C();
  W2
}
```

```
C() {
  W1
}
```

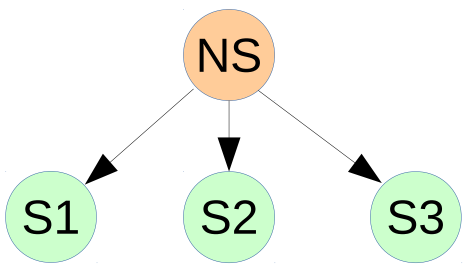
```
D() {
  E();
  W6
}
```

```
E() {
  F();
  W5
}
```

```
F() {
  W4
}
```

Thread Model: Forking

- Mixed




```
A();
D();
W7
```

```
A() {
  B();
  W3
}
```

```
B() {
  C();
  W2
}
```

```
C() {
  W1
}
```

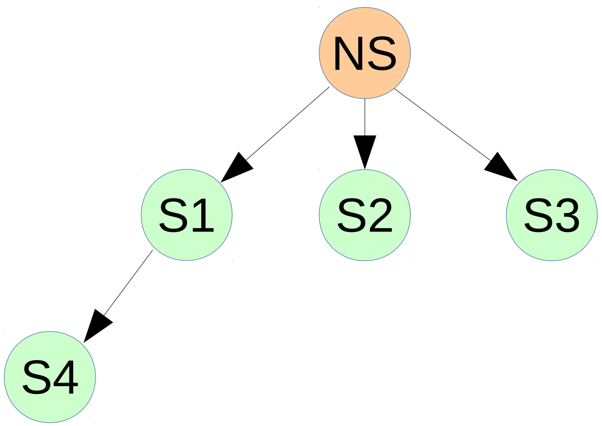
```
D() {
  E();
  W6
}
```

```
E() {
  F();
  W5
}
```

```
F() {
  W4
}
```

Thread Model: Forking

- Mixed




```
A();
D();
W7
```

```
A() {
  B();
  W3
}
```

```
B() {
  C();
  W2
}
```

```
C() {
  W1
}
```

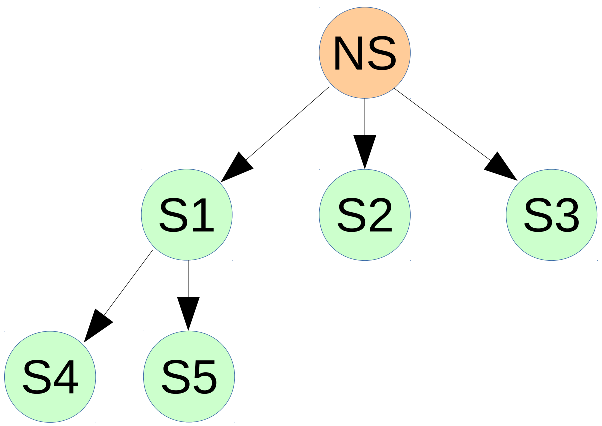
```
D() {
  E();
  W6
}
```

```
E() {
  F();
  W5
}
```

```
F() {
  W4
}
```

Thread Model: Forking

- Mixed




```
A();
D();
W7
```

```
A() {
  B();
  W3
}
```

```
B() {
  C();
  W2
}
```

```
C() {
  W1
}
```

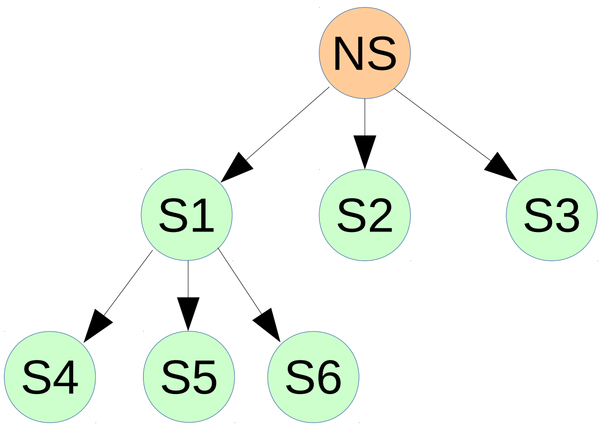
```
D() {
  E();
  W6
}
```

```
E() {
  F();
  W5
}
```

```
F() {
  W4
}
```

Thread Model: Forking

- Mixed




```
A();
D();
W7
```

```
A() {
  B();
  W3
}
```

```
B() {
  C();
  W2
}
```

```
C() {
  W1
}
```

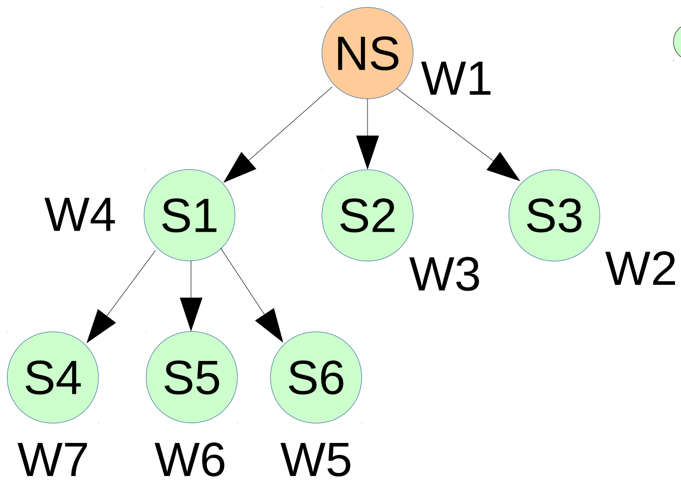
```
D() {
  E();
  W6
}
```

```
E() {
  F();
  W5
}
```

```
F() {
  W4
}
```

Thread Model: Forking

- Mixed




```
A();
D();
W7
```

```
A() {
  B();
  W3
}
```

```
B() {
  C();
  W2
}
```

```
C() {
  W1
}
```

```
D() {
  E();
  W6
}
```

```
E() {
  F();
  W5
}
```

```
F() {
  W4
}
```

Thread Model: Forking

- Out-of-order — e.g., early SableSpMT
  - Non-spec allowed n children, spec not allowed
- In-order — e.g., SpLSC/SpLIP, BOP
  - Everyone allowed 1 child
- Mixed — e.g., MUTLS
  - Everyone allowed n children
    - Immediate children out-of-order

Thread Model: Forking

- (Anti-)Synergy with where to speculate

```
for (...) {
    .
    .
}
```

```
for (...) {
    .
    .
}
```

```
for (...) {
    .
    .
}
```

Out-of-order limits speedup to 2

Thread Model: Forking

- (Anti-)Synergy with where to speculate

```
for (...) {
   .
   .
}
```

```
for (...) {
   .
   .
}
```

```
for (...) {
   .
   .
}
```

Out-of-order limits speedup to 2

Thread Model: Forking

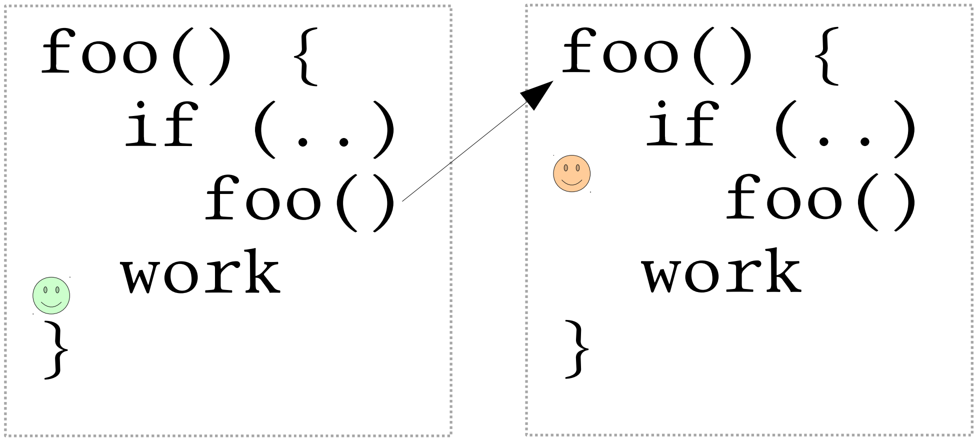
- (Anti-)Synergy with where to speculate

```
foo() {
  if (..)
    foo()
  work
}
```

Head-recursion & in-order

Thread Model: Forking

- (Anti-)Synergy with where to speculate

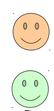


Head-recursion & in-order

Thread Model: Forking

- (Anti-)Synergy with where to speculate

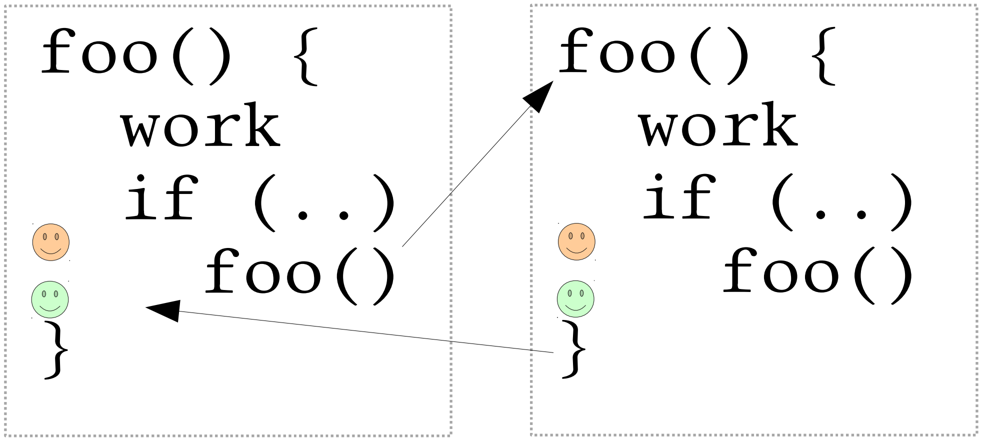
```
foo() {
  work
  if (..)
    foo()
}
```



Tail-recursion & out-of-order

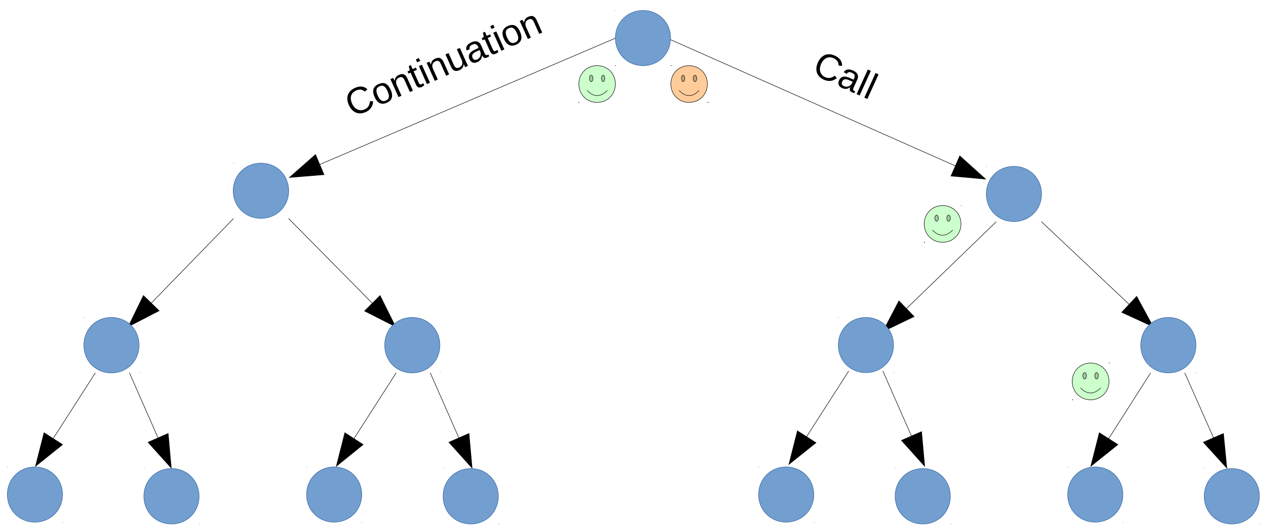
Thread Model: Forking

- (Anti-)Synergy with where to speculate



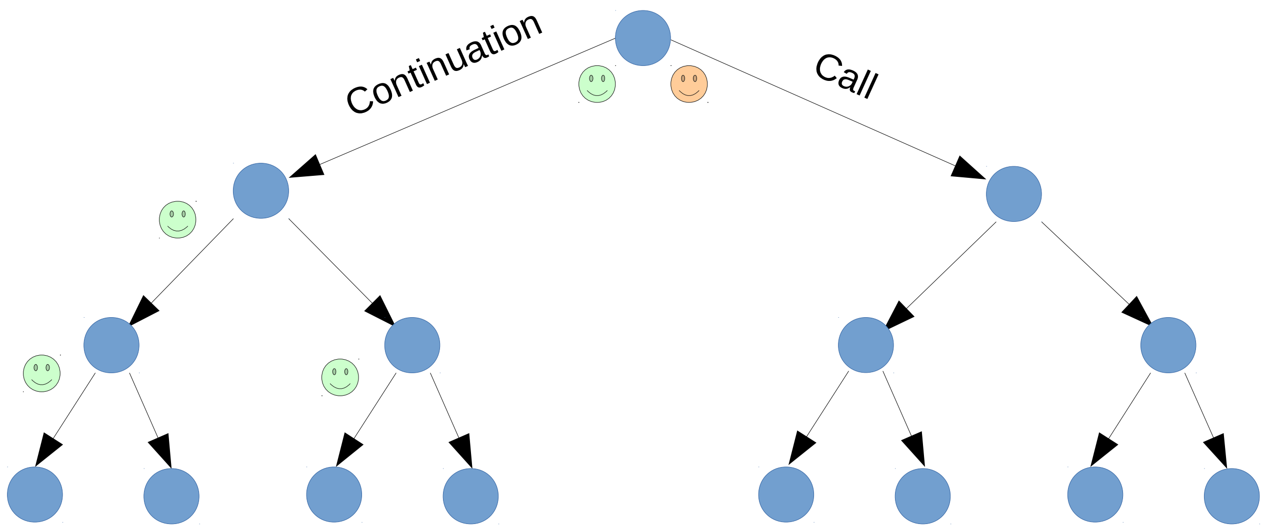
Tail-recursion & out-of-order

Thread Model: Forking



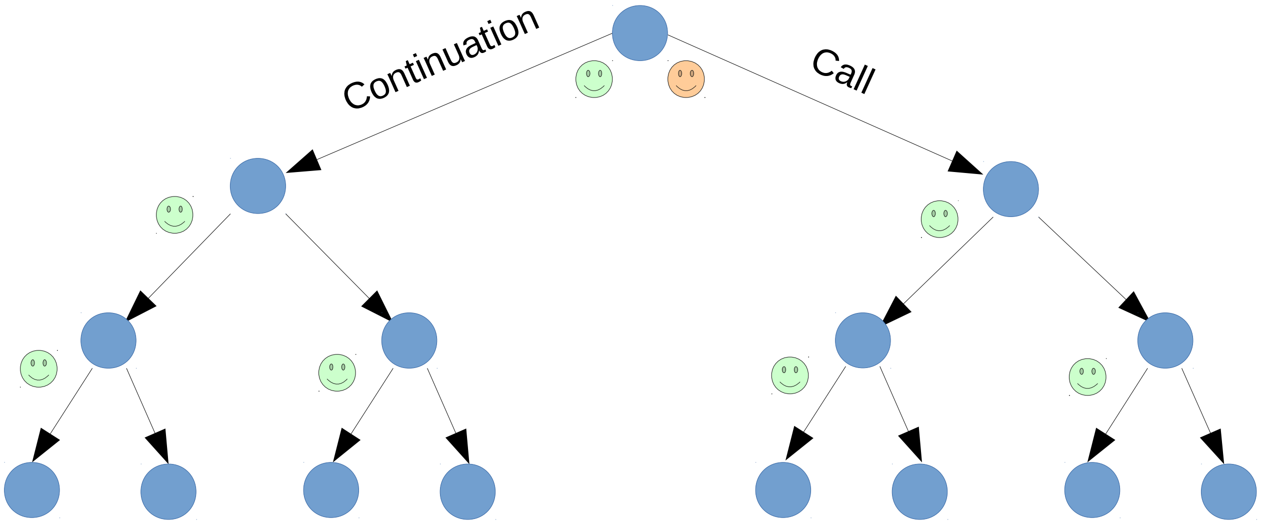
Out-of-Order

Thread Model: Forking



In-Order

Thread Model: Forking



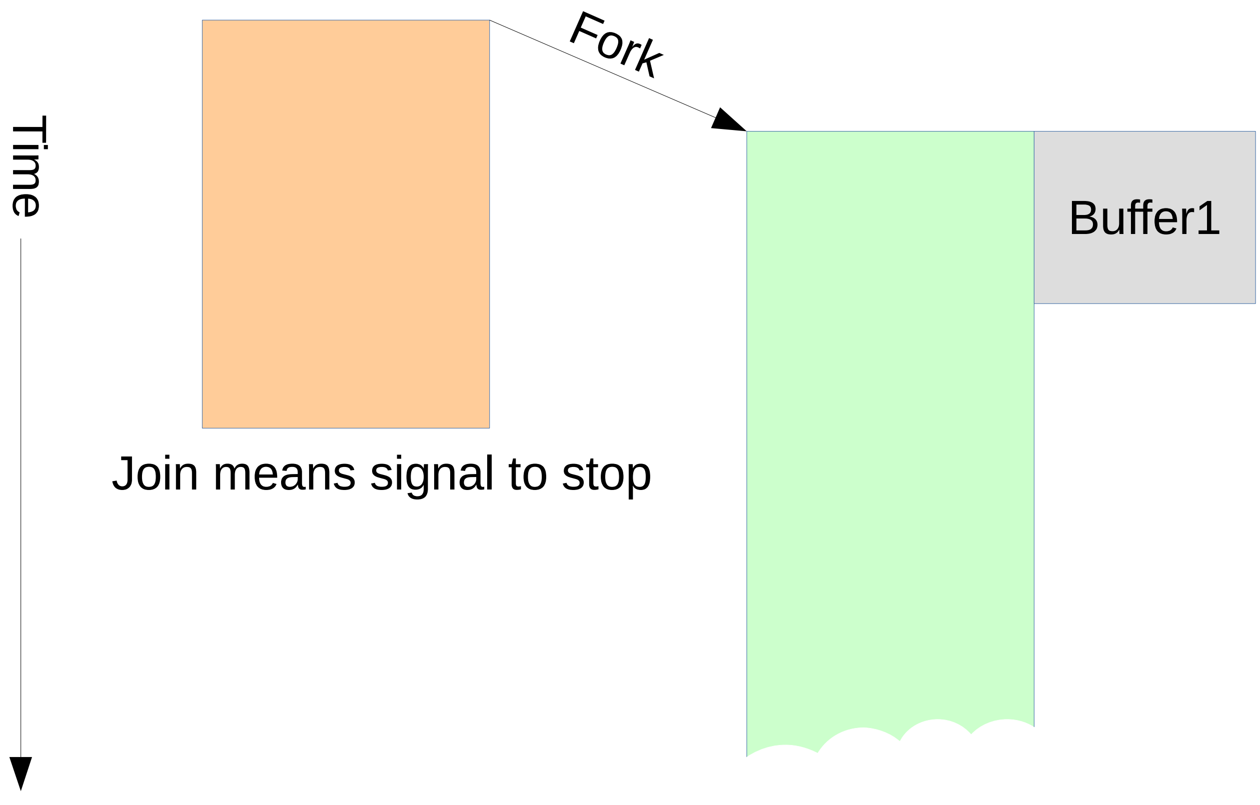
Mixed

Thread Model

- Spec threads execute until joined
- Spec threads managed by non-spec
  - Resources recycled after join

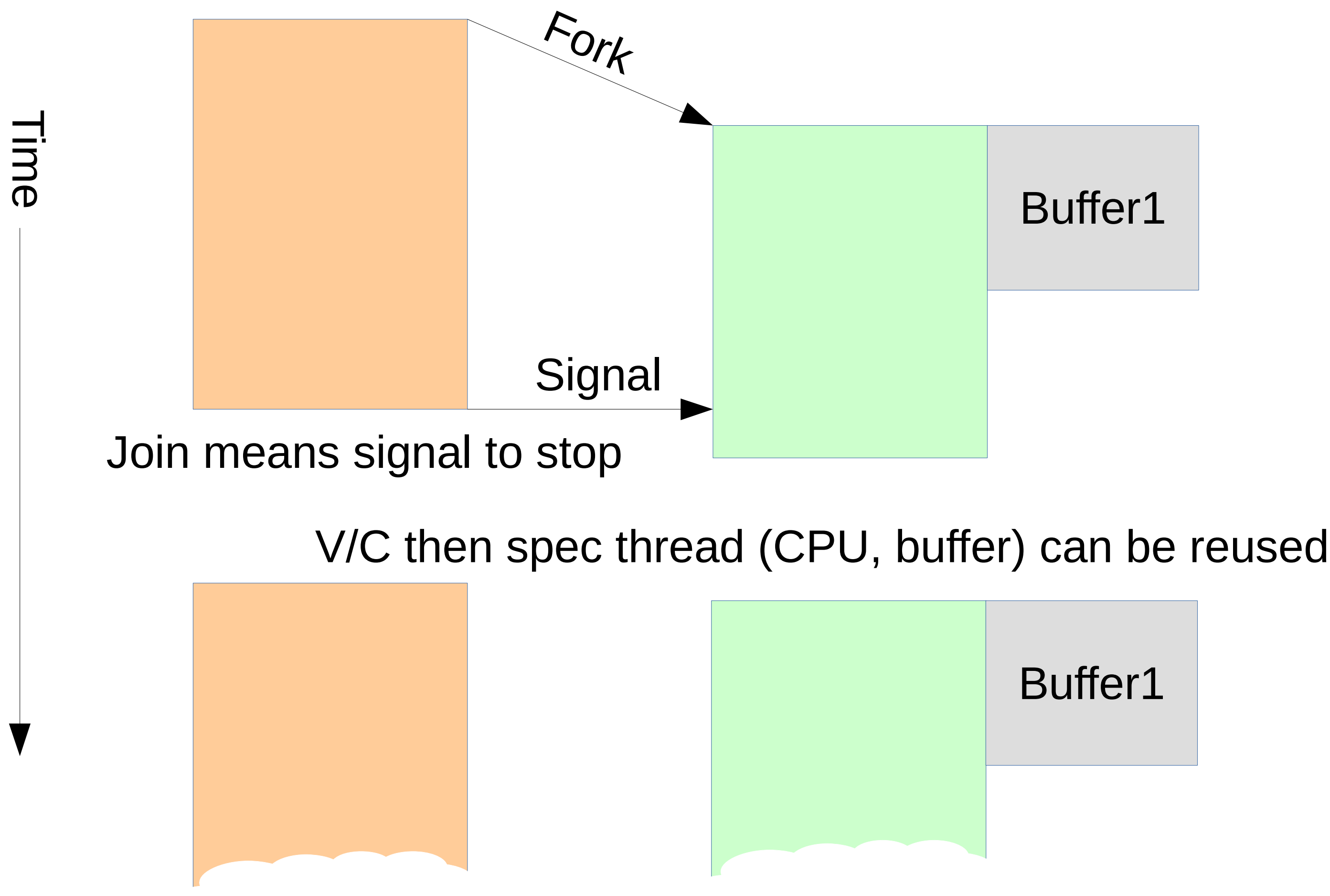
Thread Model: Recycling

- Joining implies reusability



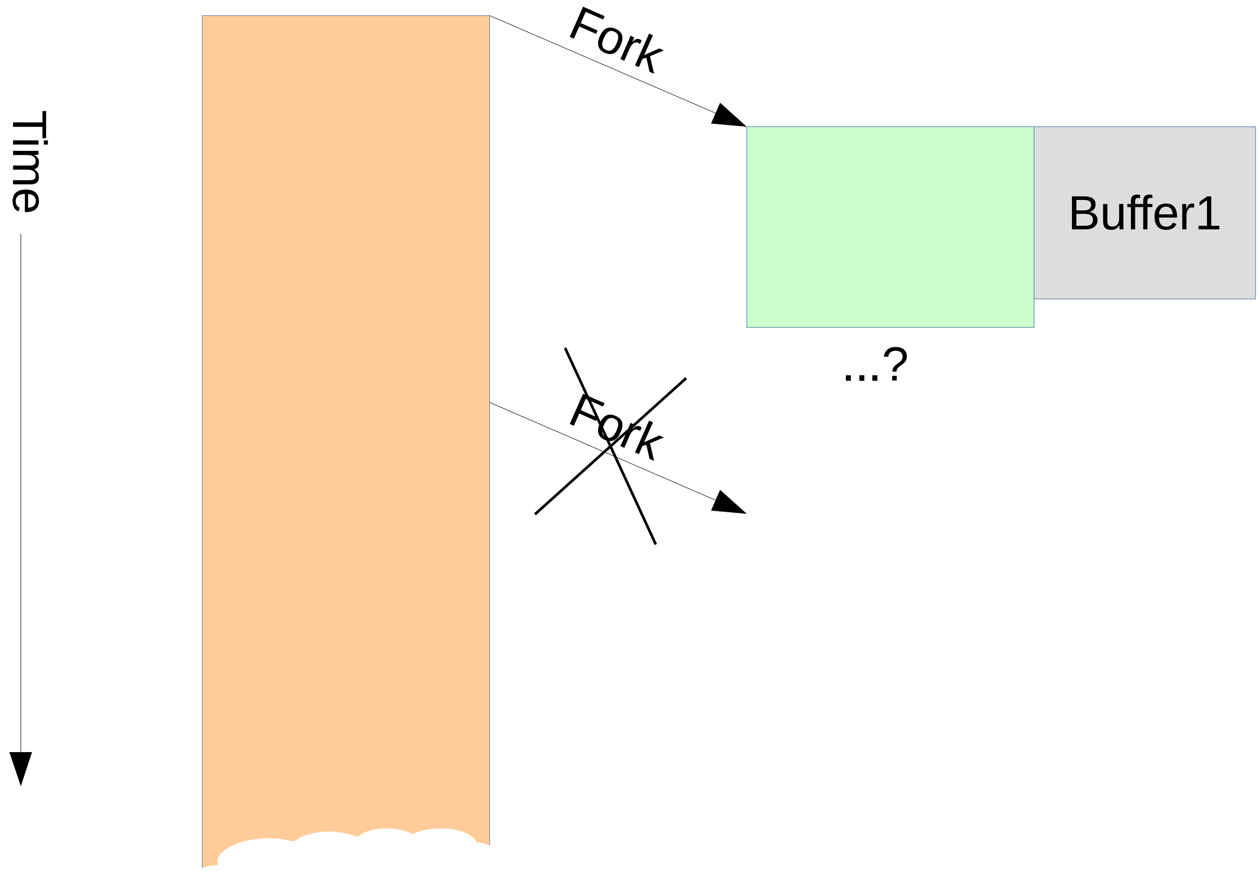
Thread Model: Recycling

- Joining implies reusability



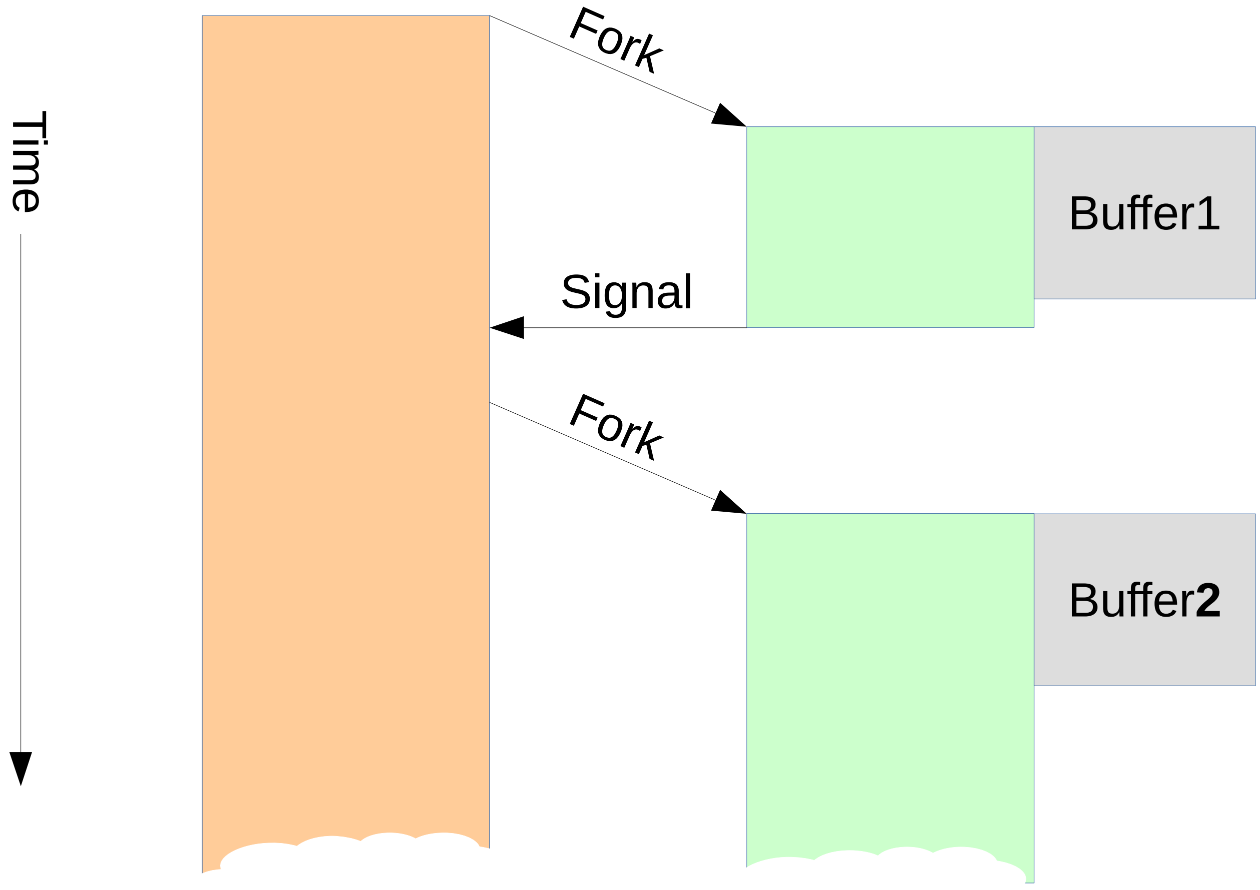
Thread Model: Recycling

- Spec thread is short - idle



Thread Model: Recycling

- Let spec thread declare itself reusable



Thread Model: Recycling

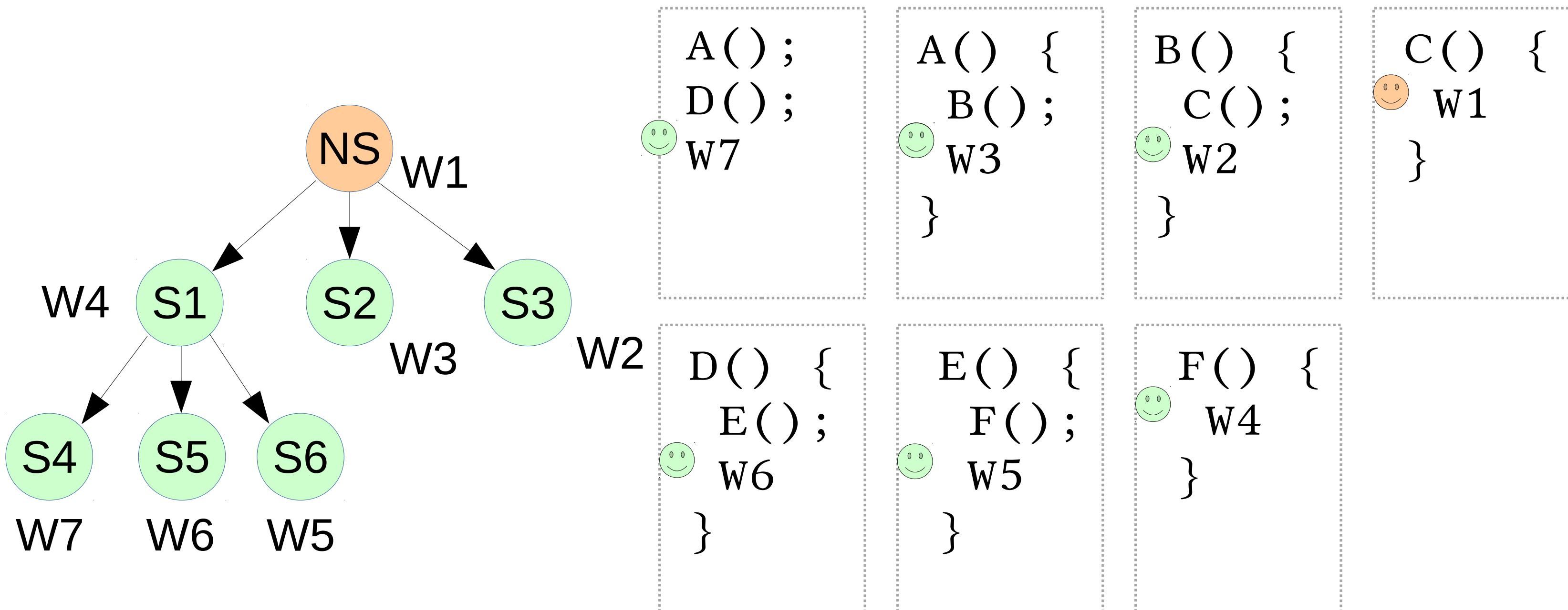
- Threads repurposed before join
  - More threads available for later spec
- More complicated buffer mgmt
  - More than 1 buffer per thread

Thread Model

- In-order, mixed => nested speculation
  - Degrees/levels of speculation
- Commit design choices
  - Who can join with whom

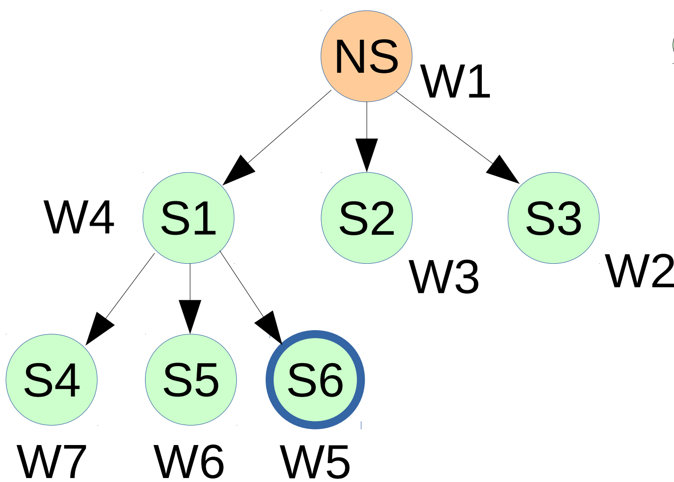
Thread Model: Commit

- Sequential commit



Thread Model: Commit

- Sequential commit




```
A();
D();
W7
```

```
A() {
  B();
  W3
}
```

```
B() {
  C();
  W2
}
```

```
C() {
  W1
}
```

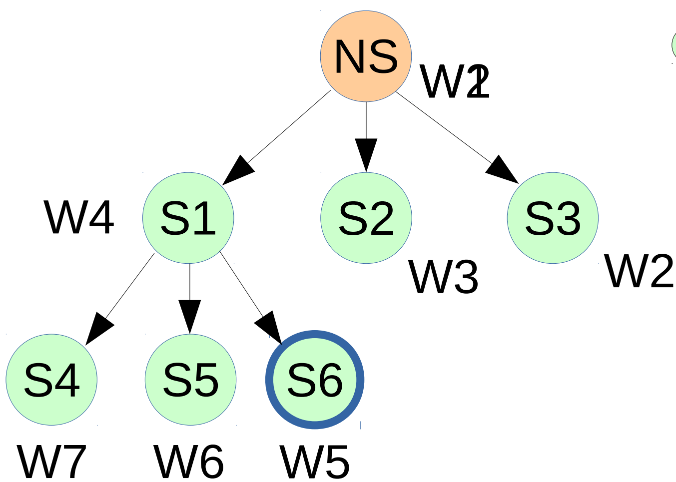
```
D() {
  E();
  W6
}
```

```
E() {
  F();
  W5
}
```

```
F() {
  W4
}
```

Thread Model: Commit

- Sequential commit




```
A();
D();
W7
```

```
A() {
  B();
  W3
}
```

```
B() {
  C();
  W2
}
```

```
C() {
  W1
}
```

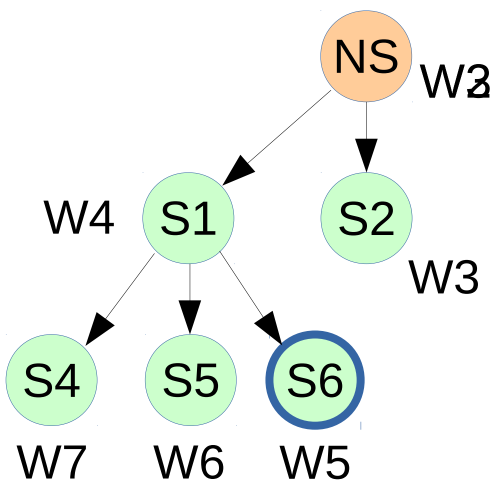
```
D() {
  E();
  W6
}
```

```
E() {
  F();
  W5
}
```

```
F() {
  W4
}
```

Thread Model: Commit

- Sequential commit




```
A();
D();
W7
```

```
A() {
  B();
  W3
}
```

```
B() {
  C();
  W2
}
```

```
C() {
  W1
}
```

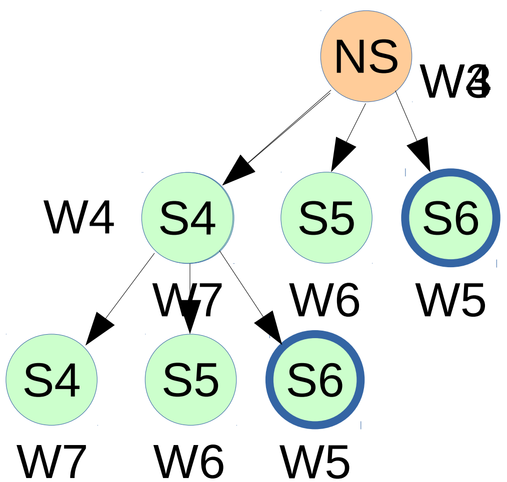
```
D() {
  E();
  W6
}
```

```
E() {
  F();
  W5
}
```

```
F() {
  W4
}
```

Thread Model: Commit

- Sequential commit




```
A();
D();
W7
```

```
A() {
  B();
  W3
}
```

```
B() {
  C();
  W2
}
```

```
C() {
  W1
}
```

```
D() {
  E();
  W6
}
```

```
E() {
  F();
  W5
}
```

```
F() {
  W4
}
```

Thread Model: Commit

- Sequential commit simplifies buffer mgmt
  - Always copy from spec buffer to main memory
  - Commit must precede buffer reuse

- Non-seq commit
  - More complex
    - merged validation reqs
    - buffer size may grow with each commit
    - extra sync (spec/spec as well as non-spec/spec)

Choices, Choices, Choices

- Speculative Targets
- Thread Model
- Versioning Model
- Performance Model

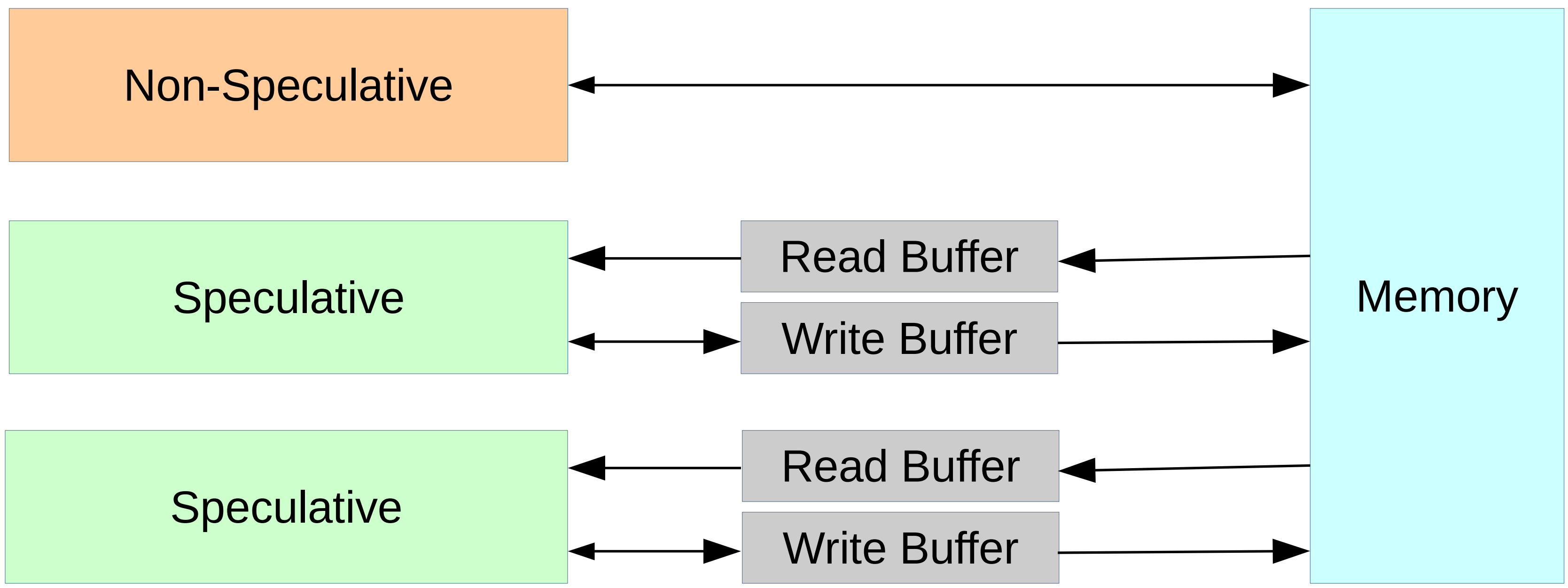
Version Control

- Multiple ways we can ensure state correctness

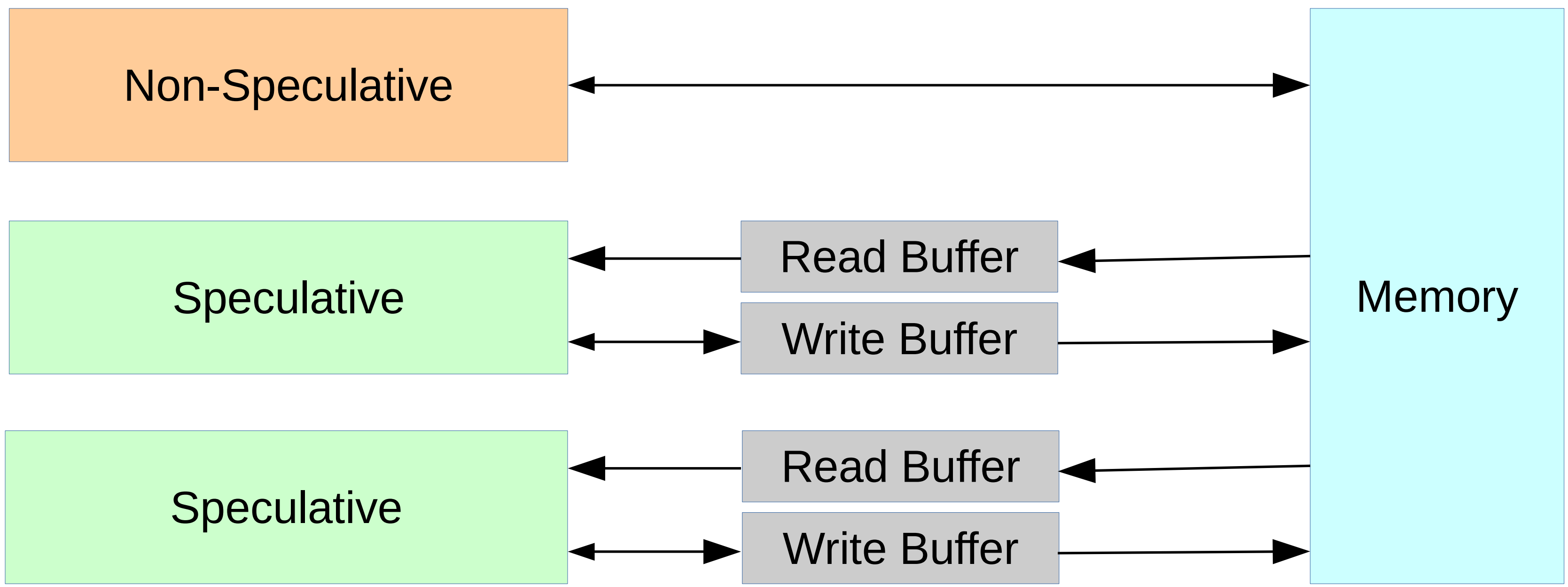
- ***Lazy*** scheme
  - <u>Validation</u> checks for RAW violations
    - Spec maintains a ***read buffer***
      - Reading a stale value is detected
  - <u>Isolation</u> obviates WAR and WAW worries
    - Spec maintains a ***write buffer***
      - Writes from the future postponed until ready

e.g., SableSpMT, SpLSC, Lector

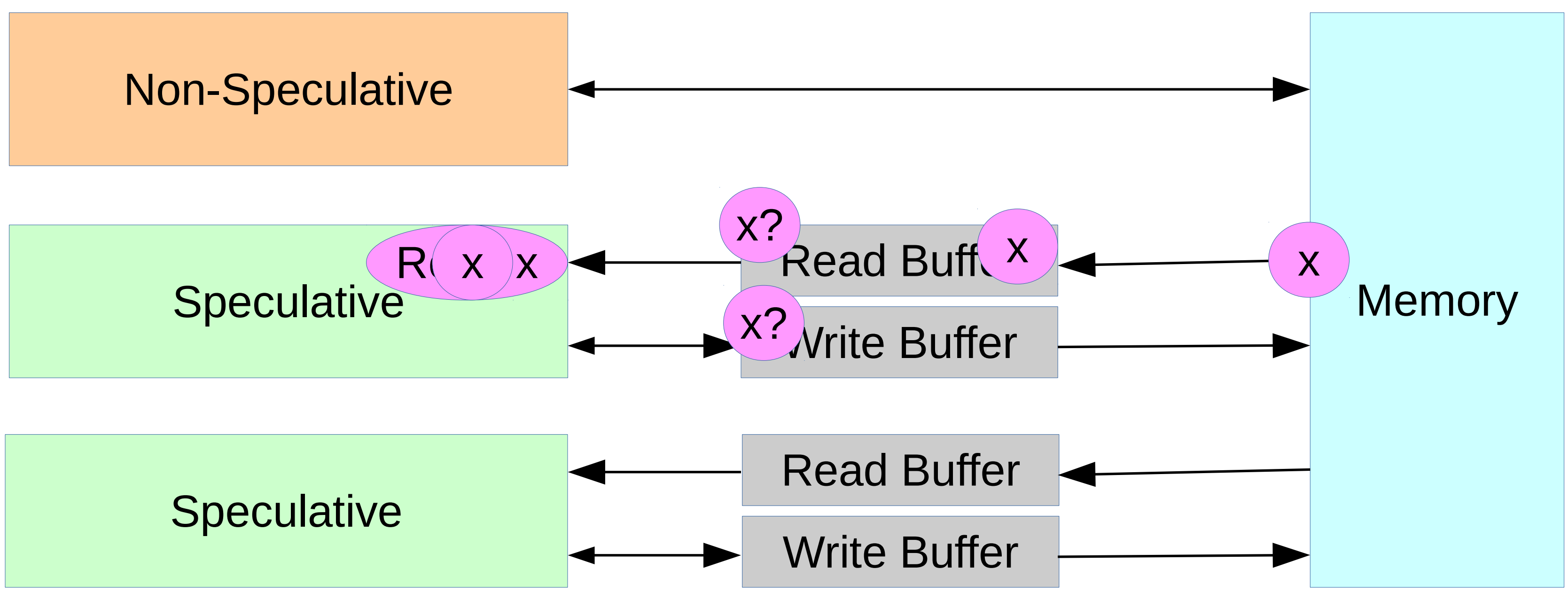
Version Control: Lazy



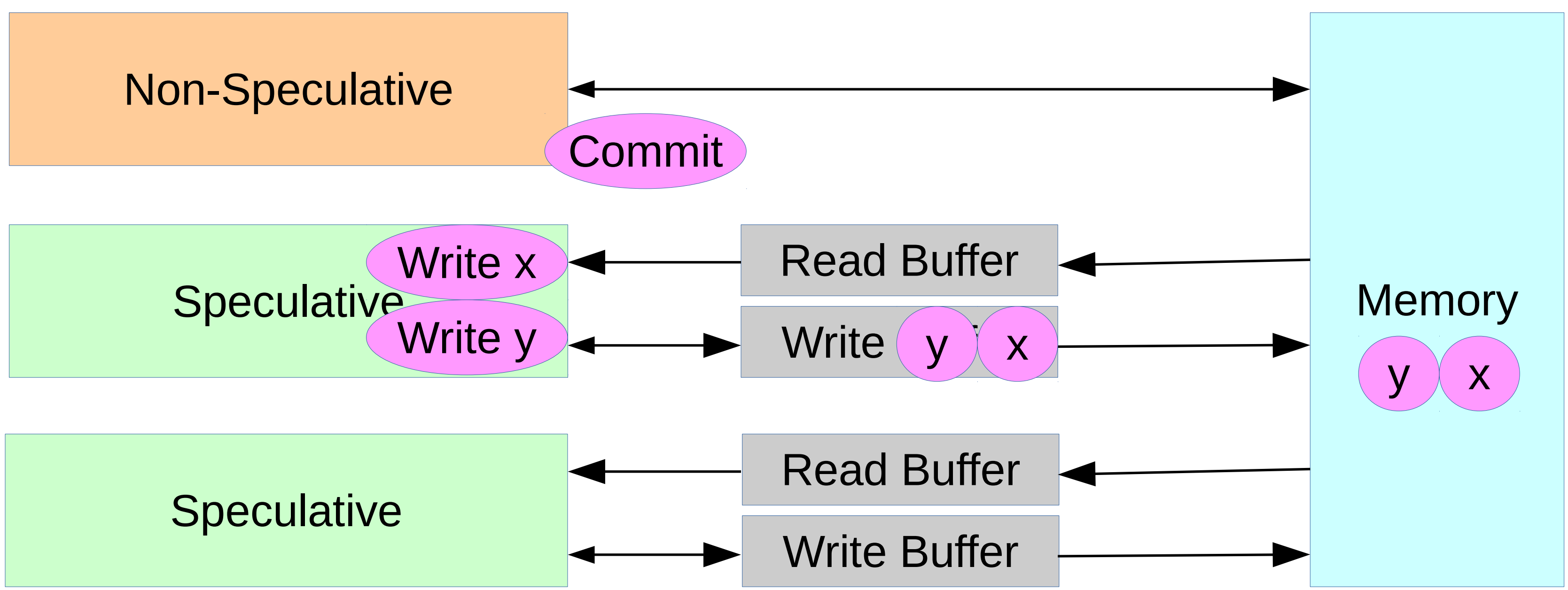
Version Control: Lazy



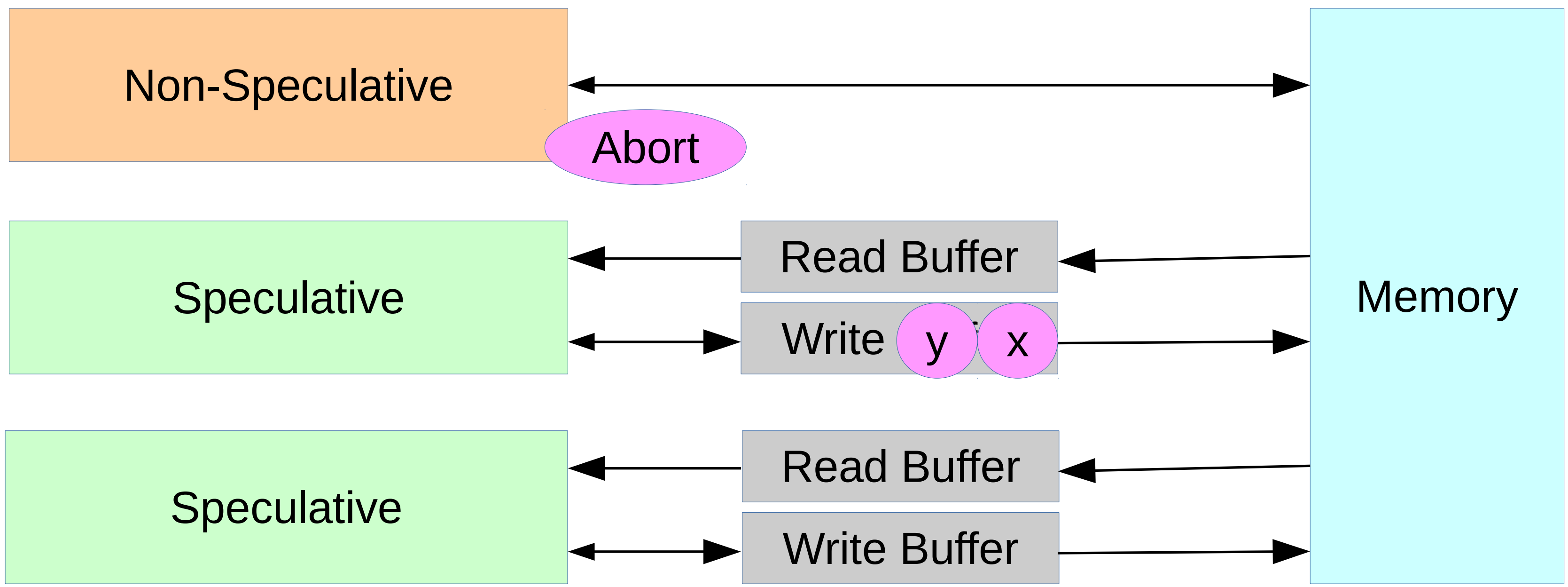
Version Control: Lazy



Version Control: Lazy



Version Control: Lazy



Version Control

- Lazy
  - Abort is cheap
  - Validation/Commit is expensive
    - Big buffers, lots of memory traffic
- Spec *should be* likely to succeed
  - We are avoiding misspeculation
  - Better would be
    - Abort is expensive
    - Validation/Commit is cheap

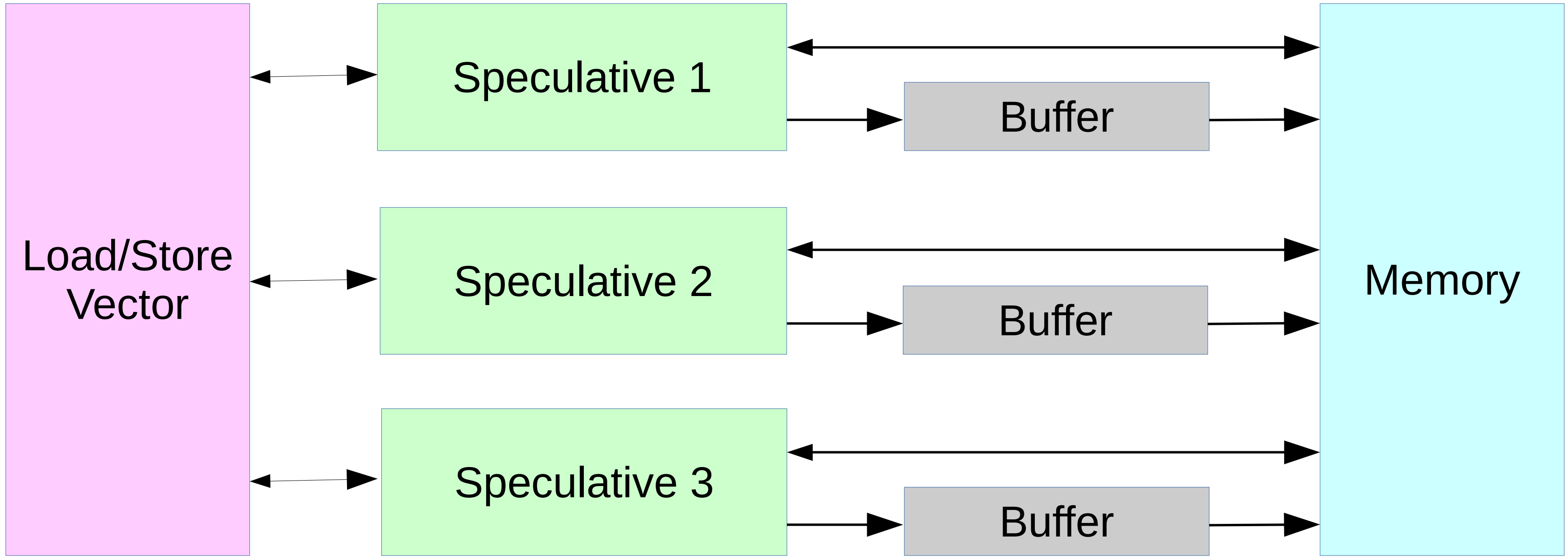
Version Control

- ***Eager*** scheme
  - Threads write to memory directly
    - Maintain a shadow/prior-version buffer for undo
  - No privileged non-spec

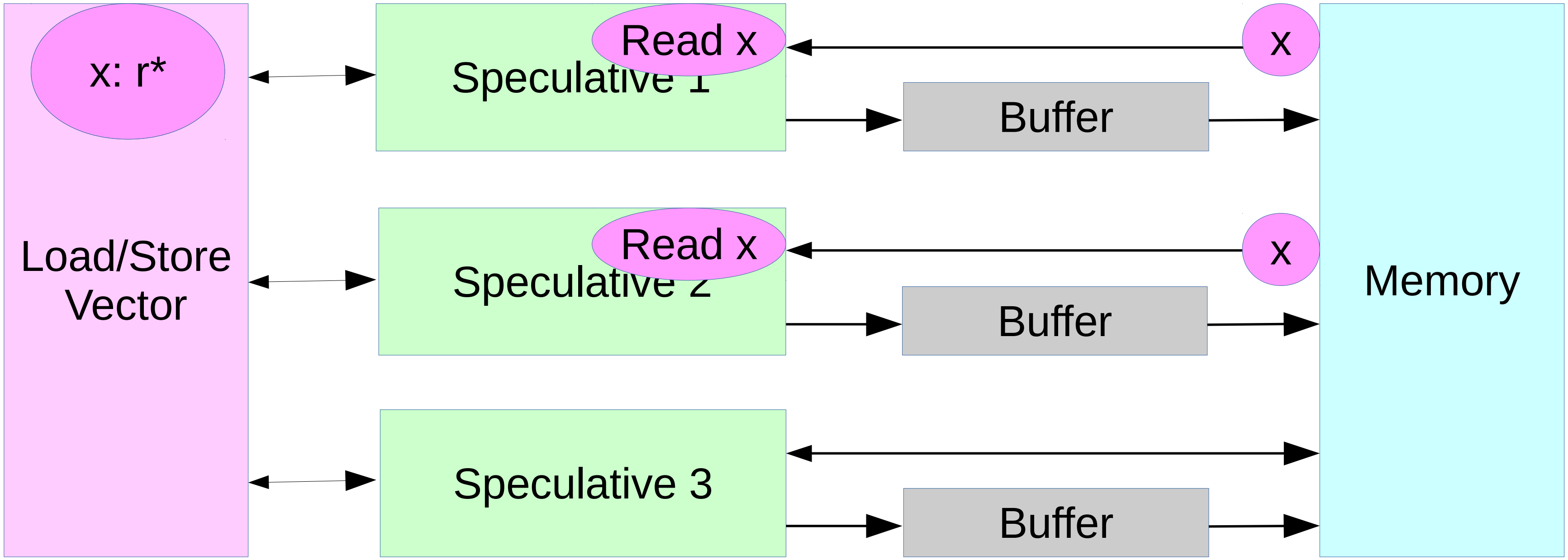
- Eager avoids commit overhead
  - Trades for more expensive undo
  - More complex version management

e.g., SpLIP, MiniTLS, MUTLS (hybrid)

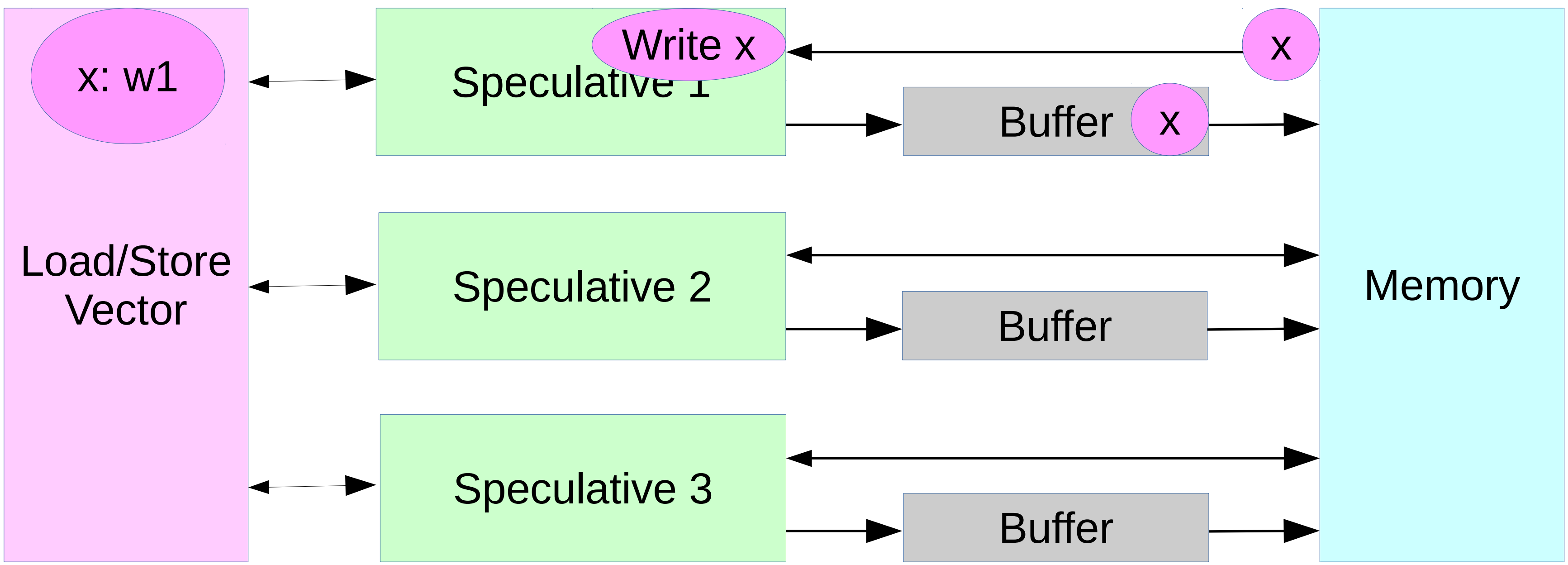
Version Control: Eager



Version Control: Eager



Version Control: Eager

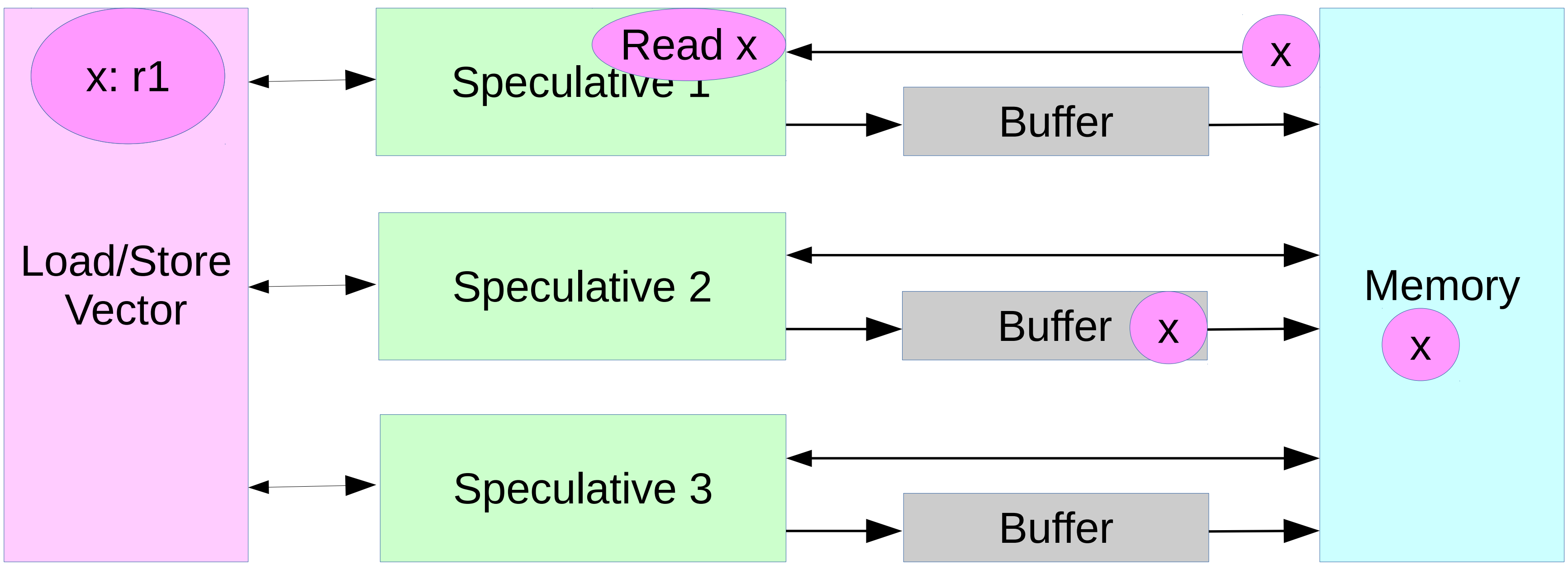


Version Control: Eager

- Load/Store vector tracks current version
  - Who wrote, who read
  - Thread id's for order decisions

- RAW error
  - Writer rolls back reader(s)
- WAR
  - Reader rolls back writer
- WAW
  - Writer rolls back previous writer

Version Control: Eager



Version Control: Eager

- V/C amortized, effectively parallelized
- Drawbacks
  - Version checks
  - Sensitive to WAR and WAW errors
  - Rollback much more complicated
- No progress guarantee

Choices, Choices, Choices

- Speculative Targets
- Thread Model
- Versioning Model
- Performance Model

Performance Model

- Resource limitations another concern
  - Forward-dependence in fork choices

```
foo() {
  ping();
  woot();
}
bar();
```

```
foo() {
  ping();
  woot();
}
bar();
```

Is (foo || bar) better than (ping || woot)?

Performance Model

- Can we determine potential benefit?
  - *Toward* an ahead-of-time model

- Data dependencies unknown
  - Assume best case
- Dynamic control flow unknown
  - Start with traces (post-facto)

Abstract Modelling: MLS, Lazy

```
t₀
t₁
foo() {
 t₂
 bar() {
  t₃
 }
 t₄
}
t₅
ping() {
 t₆
}
t₇
t₈
```

Abstract Modelling: MLS, Lazy

Sequential code chunks

```
t0
t1
foo() {
 t2
 bar() {    .
  t3        .
 }          .
 t4
}
t5
ping() {
 t6
}
t7
t8
```

Abstract Modelling: MLS, Lazy

Call-sites

```
t0
t1
foo() {
 t2
 bar() {
  t3
 }
 t4
}
t5
ping() {
 t6
}
t7
t8
```

Abstract Modelling: MLS, Lazy

Call-sites

Matching continuations

```
t0
t1
foo() {
 t2
 bar() {
  t3
 }
 t4
}
t5
ping() {
 t6
}
t7
t8
```

Abstract Modelling: MLS, Lazy

```
t0
t1
foo() {
  t2
  bar() {
    t3
  }
  t4
}
t5
ping() {
  t6
}
t7
t8
```

We have n>0 threads

Each call-site implies a fork choice

Split trace into pieces

Call, continuation

How many threads?

In-order: call (1), cont (n-1)
Out-of-order: call(n-1), cont(1)
Mixed: call(m), cont(n-m) over 0<m<n

Post-join: where do we pick up again?

How far did spec get before joining?
How far can we get before joining?

Abstract Modelling: MLS, Lazy

```
t0
t1
foo() {
 t2
 bar() {
  t3
 }
 t4
}
t5
ping() {
 t6
}
t7
t8
```

Abstract Modelling: MLS, Lazy

```
t0
t1
foo() {
 t2
 bar() {
  t3
 }
 t4
}
t5
ping() {
 t6
}
t7
t8
```

Abstract Modelling: MLS, Lazy

```
t₀
t₁
foo() {
 t₂
 bar() {
  t₃
 }
 t₄
}
t₅
ping() {
 t₆
}
t₇
t₈
```

Determine how long the call takes

Abstract Modelling: MLS, Lazy

```
t₀
t₁
foo() {
 t₂
 bar() {
  t₃
 }
 t₄
}
t₅
ping() {
 t₆
}
t₇
t₈
```

Determine how long the call takes

Recursively process body

Abstract Modelling: MLS, Lazy

```
t0
t1
foo() {
 t2
 bar() {
  t3
 }
 t4
}
t5
ping() {
 t6
}
t7
t8
```

Determine how long the call takes

Recursively process body

Recursively process body

Abstract Modelling: MLS, Lazy

```
t0
t1
foo() {
 t2
 bar() {
  t3
 }
 t4
}
t5
ping() {
 t6
}
t7
t8
```

Determine how long the call takes

Recursively process body

Recursively process body

Abstract Modelling: MLS, Lazy

```
t0
t1
foo() {
 t2
 bar() {
  t3
 }
 t4
}
t5
ping() {
 t6
}
t7
t8
```

Determine how long the call takes

Recursively process body

Recursively process body

Abstract Modelling: MLS, Lazy

```
t0
t1
foo() {
 t2
 bar() {
  t3
 }
 t4
}
t5
ping() {
 t6
}
t7
t8
```

Determine how long the call takes

Recursively process body

Recursively process body

Abstract Modelling: MLS, Lazy

```
t₀
t₁
foo() {
 t₂
 bar() {
  t₃
 }
 t₄
}
t₅
ping() {
 t₆
}
t₇
t₈
```

Total time: 6

Abstract Modelling: MLS, Lazy

- Full model considers all partitionings of
  - T = S;(A|B);C
  - A,B,C recursively parallelized
- Misspeculation, ...

- Expensive!
  - Based on finding all possible traces (timings)

Abstract Modelling: MLS, Lazy

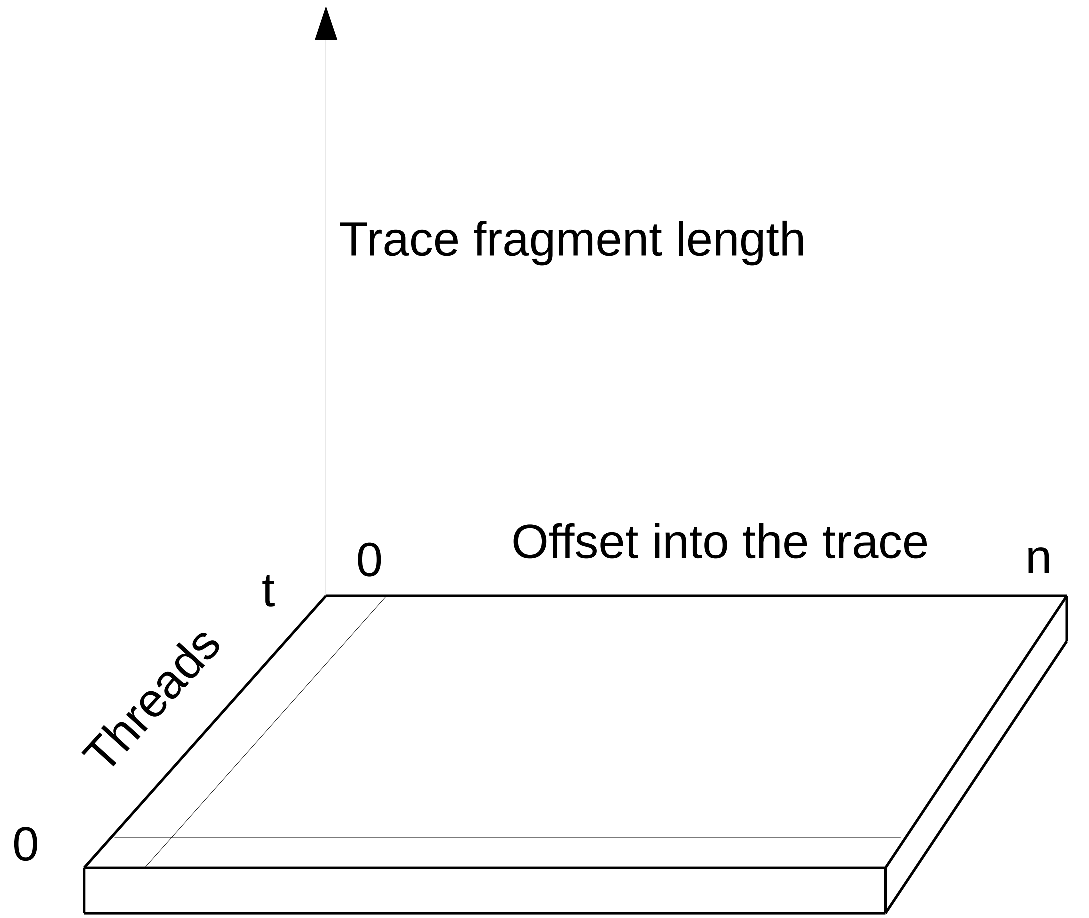
- Only interested in best-possible perf
- Lots of recursive calculations

- Dynamic programming model helps
  - Break down into a merge of smaller problems

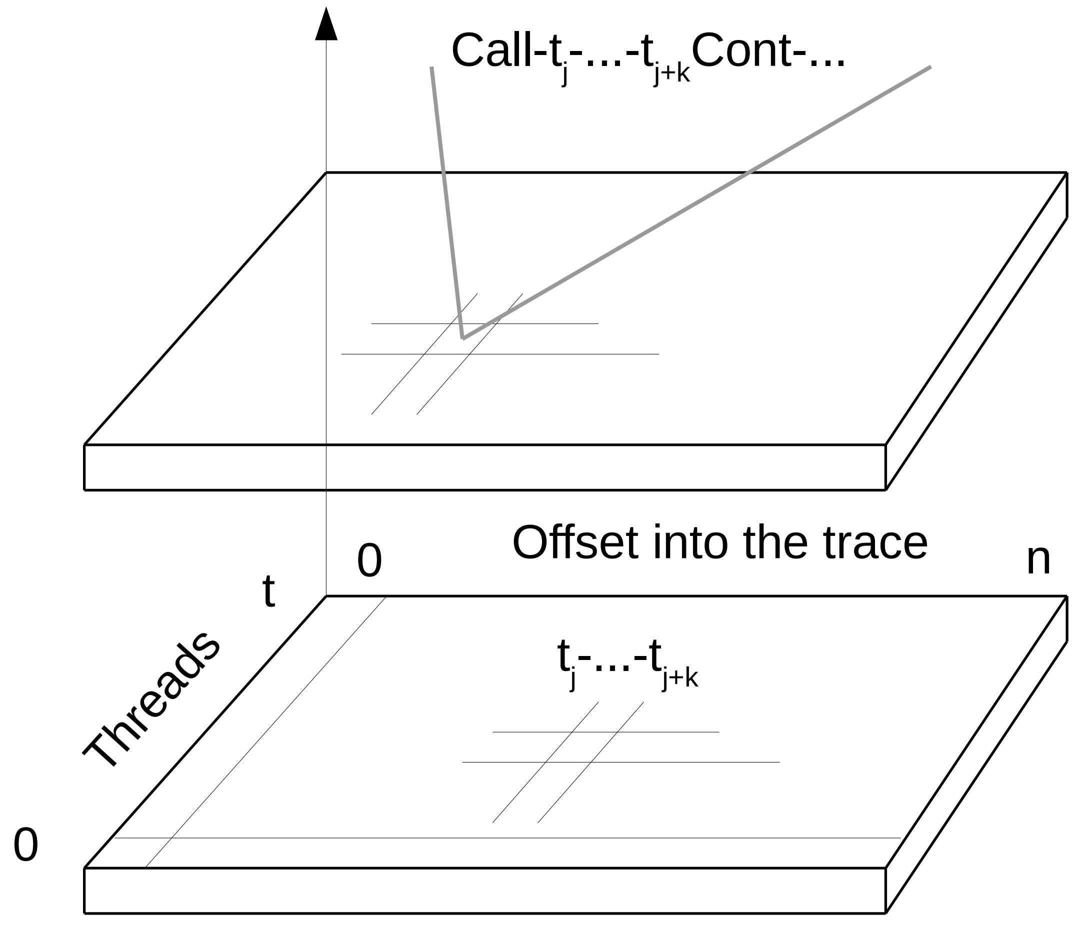
Abstract Modelling: MLS, Lazy

- Processing a partial trace from an offset
    - $T = t_i \ldots t_j$
- Find best performance given a thread budget
- Recursive:
    - Make a step, reduce to a small trace
    - Look at memoized perf of smaller traces
- Base case: each trace unit does 1 work

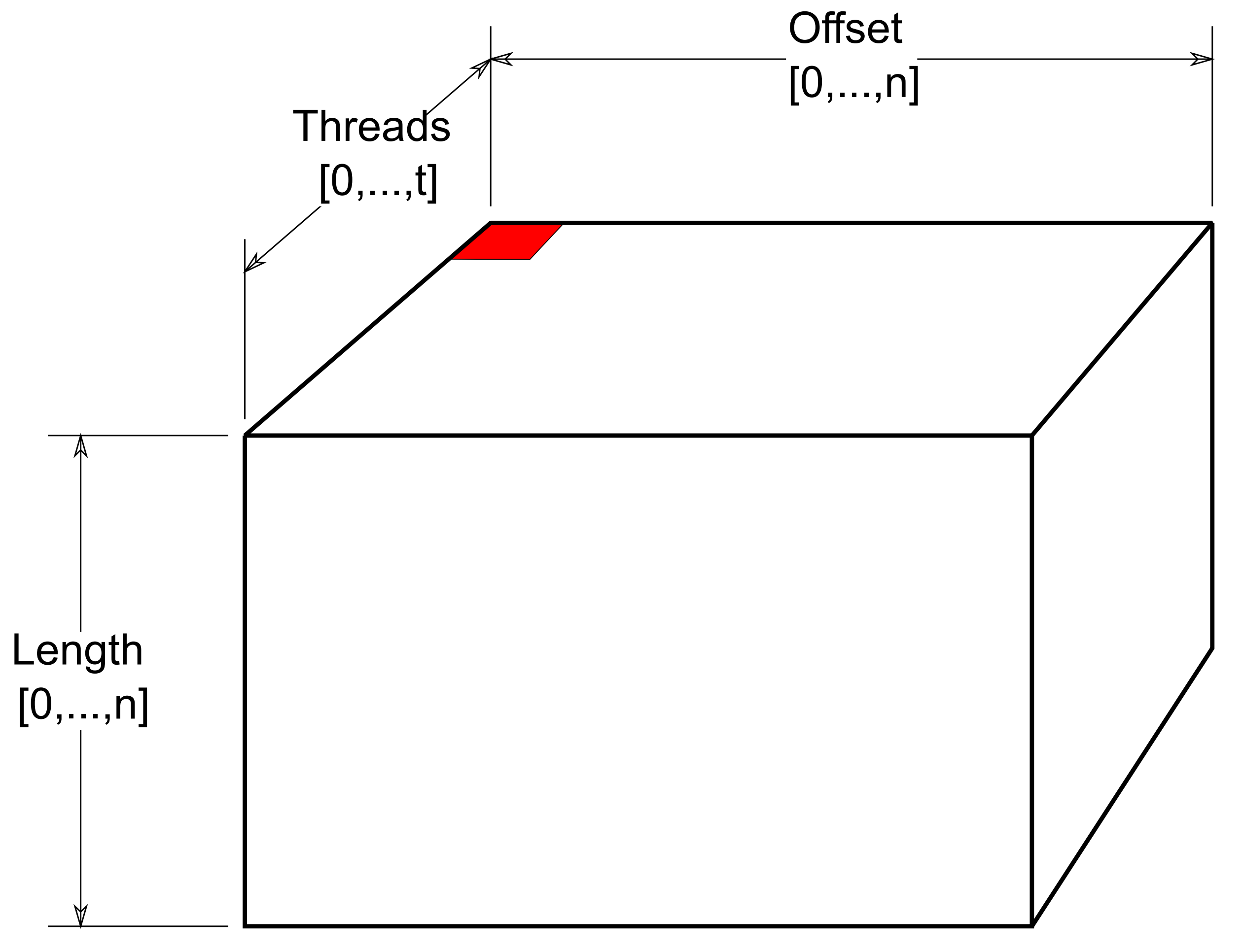
Abstract Modelling: MLS, Lazy



Abstract Modelling: MLS, Lazy



Abstract Modelling: MLS, Lazy



Abstract Modelling: MLS, Lazy

- Limitations
  - Trace-based
  - Unit work size
  - Loop-speculation?
    - Represent cyclic properties?
- Most interesting for showing it is possible

Conclusion

- Plenty of room for component optimizations
- But also major design choices

- Different progs respond differently
  - Adaptive, hybrid forms

- Modelling an interesting direction

Thank You for Listening

Questions!