Disentangling Software
Thread-level Speculation

Clark Verbrugge
clump@cs.mcgill.ca

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 |

Real Time
Basic Idea

- E.g.

  - Fork
  - Join
  - Real Time

  - Non-speculative thread
  - Idle/Speculative thread
Basic Idea

- E.g.

  - Non-speculative thread
  - Idle/Speculative thread

  
  
  
  Real Time
  
  
  
  Fork
  
  
  
  Join
  
  
  Speculative
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

\[ x = \ldots \]

\[ \ldots = x \]

*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

```java
// smiley
foo(); // composer

// ... 

foo() {
  
  // ... 

  // ... 

  
  // ... 

  
}
```
Method-level Speculation

```java
foo();

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

Fork
Join
```
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

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

for (...) {
  
  
}

for (...) {
  
  
}
```
Loop-level Speculation

Fork

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

Join

```
}
```

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

```python
for (...) {
    ...
}
```
Loop-level Speculation

Fork

\[
\text{for (\ldots) } \{
\text{.}
\}
\]

V/C

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]

\[
\text{.}
\]
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

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

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

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

```c
foo() {

```
Thread Model: Forking

- Out-of-order

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

ping() {
  ...
}

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

```c
foo() {
```

Thread Model: Forking

- Out-of-order

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

s1
...
}

bar() {
    ...
    ping()
}

ping() {
    ...
}

foo() {
    ↓
    bar() {
```
Thread Model: Forking

• Out-of-order

```java
foo() {
    ...
    bar()
    s1 ☹ ...
}

bar() {
    ☹ ...
    ping()
    s2 ☻ ...
}

ping() {
    ...
}

foo() {
    ↓
    bar() {
```
Thread Model: Forking

- Out-of-order

```c
foo() {
    ...
    bar()
    s1 😃 ...
}
```
```
bar() {
    ...
    ping()
    s2 😃 ...
}
```
```
ping() {
    😾 ...
}
```

```plaintext
foo() {
  ↓
bar() {
  ↓
ping() {
```
Thread Model: Forking

- Out-of-order

```c
foo() {
    ...
    bar()
    s1  ...
}

bar() {
    ...
    ping()
    s2  ...
}

ping() {
    ...
}
```

```
foo() {
    bar() {
        ping()
        Join S2
    }
    s1  ...
}

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

```
foo() {
    bar() {
        ping()
        Join S2
    }
    s1  ...
}
```

```
foo() {
    bar() {
        ping()
        Join S2
    }
    s1  ...
}
```

```
foo() {
    bar() {
        ping()
        Join S2
    }
    s1  ...
}
```

```
foo() {
    bar() {
        ping()
        Join S2
    }
    s1  ...
}
```

```
foo() {
    bar() {
        ping()
        Join S2
    }
    s1  ...
}
```

```
foo() {
    bar() {
        ping()
        Join S2
    }
    s1  ...
}
```

```
foo() {
    bar() {
        ping()
        Join S2
    }
    s1  ...
}
```

```
foo() {
    bar() {
        ping()
        Join S2
    }
    s1  ...
}
```
Thread Model: Forking

• Out-of-order

```
foo() {
    ...
    bar()
    s1 ...
}

bar() {
    ...
    ping()
    s2 ...
}

ping() {
    ...
}

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

Thread Model: Forking

- In-order

```c
foo() {
    ...
    ☹ bar()
    ...
    ping()
    ...
    woot()
    ...
}
```
Thread Model: Forking

• **In-order**

```c
foo() {
    ...
    bar()  // Thread 1
    ...
    ping()  // Thread 2
    ...
    woot()  // Thread 3
    ...
}
```
Thread Model: Forking

• In-order

```c
foo() {
    ...
    bar() ...
    ping() ...
    woot() ...
}
```
Thread Model: Forking

- In-order

```c
foo() {
    ...
    bar()
    ...
    ping()
    ...
    woot()
    ...
}
```
Thread Model: Forking

- In-order

```c
foo() {
    ...
    bar()
    ...
    ping()
    ...
    woot()
    ...
}
```
Thread Model: Forking

- In-order

```cpp
foo() {
    ...
    bar() ...
    ...
    ping() ...
    ...
    woot() ...
    ...
}
```

S2 🙄

S1 😊
Thread Model: Forking

• In-order

```c
foo() {
    ...
    bar() → 😊
    ...
    ping() → S1 😊
    ...
    woot() → S2 😊
    ...
}
```
Thread Model: Forking

- In-order

```c
foo() {
    ...
    bar()
    ...
    ping()
    ...
    woot()
    ...
}
```

- S1
- S2
- S3
Thread Model: Forking

- In-order

```c
foo() {
    ...
    bar()
    ...
    ping()
    ...
    woot()
    ...
}
```

S3 $\rightarrow$ Join S1
S2 $\rightarrow$ S3

In-order sequence of tasks:
- S3 (foo)
- S2 (bar)
- S2 (ping)
- S2 (woot)
- S3 (foo)
Thread Model: Forking

• In-order

```c
foo() {
    ...
    bar()
    ...
    ping()
    ...
    woot()
    ...
}
```

Join S1

S2

S3
Thread Model: Forking

- In-order

```
foo() {
  ...
  bar()    ←
  ...
  ping()   ←
  ...
  woot()   ←
  ...
}
```

Join S1

Join S2
Thread Model: Forking

- Mixed
Thread Model: Forking

- Mixed

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

A() {
B();
W3
}

B() {
C();
W2
}

D() {
E();
W6
}

E() {
F();
W5
}

F() {
W4
}

C() {
W1
}
```
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

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

A() {
  B();
  W3
}

B() {
  C();
  W2
}

C() {
  W1
}

D() {
  E();
  W6
}

E() {
  F();
  W5
}

F() {
  W4
}
```
Thread Model: Forking

- Mixed

```plaintext
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
}
```

S1
S2
S3
S4
S5
NS
Thread Model: Forking

- Mixed

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

B();
W3

C();
W2

D();
E();
W6

E();
F();
W5

B();
C();
W2

C();
W1
```

Diagram:

```
NS
  └── S1
     └── S4
  └── S2
     └── S5
  └── S3
     └── S6
```
Thread Model: Forking

- Mixed
Thread Model: Forking

- **Out-of-order**
  - Non-spec allowed n children, spec not allowed
    - e.g., early SableSpMT

- **In-order**
  - Everyone allowed 1 child
    - e.g., SpLSC/SpLIP, BOP

- **Mixed**
  - Everyone allowed n children
    - Immediate children out-of-order
    - e.g., MUTLS
Thread Model: Forking

- (Anti-)Synergy with where to speculate

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

Out-of-order limits speedup to 2
Thread Model: Forking

• (Anti-)Synergy with where to speculate

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

Out-of-order limits speedup to 2
Thread Model: Forking

- (Anti-)Synergy with where to speculate

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

Head-recursion & in-order
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

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

Tail-recursion & out-of-order
Thread Model: Forking

- (Anti-)Synergy with where to speculate

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

Tail-recursion & out-of-order
Thread Model: Forking

Out-of-Order
Thread Model: Forking

In-Order
Thread Model: Forking
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

Join means signal to stop

V/C then spec thread (CPU, buffer) can be reused
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
Thread Model: Commit

- Sequential commit
Thread Model: Commit

• Sequential commit

```
A();
B();
C();
D();
E();
F();
```

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

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

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

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

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

```
F() {
    W4
}
```
Thread Model: Commit

- Sequential commit

```plaintext
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
  – **Validation** checks for RAW violations
    • Spec maintains a *read buffer*
      – Reading a stale value is detected
  – **Isolation** 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

Non-Speculative

Speculative

Speculative

Memory

Read Buffer

Write Buffer

Read Buffer

Write Buffer
Version Control: Lazy
Version Control: Lazy

Diagram:
- Non-Speculative
- Speculative
  - Read Buffer
  - Write Buffer
- Memory
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

Load/Store Vector

Speculative 1

Speculative 2

Speculative 3

Memory

Buffer

Read x

x: r*

x: r*
Version Control: Eager

Load/Store Vector

Speculative 1

Speculative 2

Speculative 3

Memory

x: w1

Write x

x
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

```plaintext
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

t_0

t_1

foo() {
  t_2
  bar() {
    t_3
  }
  t_4
}

t_5

ping() {
  t_6
}

t_7

t_8
Abstract Modelling: MLS, Lazy

Sequential code chunks.

```
t_0

 t_1

 foo() {
 t_2

  bar() {
 t_3

   }

  }

 t_4

 }

 t_5

 ping() {
 t_6

  }

 t_7

 t_8
```
Abstract Modelling: MLS, Lazy

```
t_0
t_1
foo() {
  t_2
  bar() {
    t_3
  }
  t_4
}
}
t_5
ping() {
  t_6
}
}
t_7
t_8
```
Abstract Modelling: MLS, Lazy

\[ t_0 \]
\[ t_1 \]
\[ \text{foo}() \{ \]
\[ t_2 \]
\[ \text{bar}() \{ \]
\[ t_3 \]
\[ \} \]
\[ t_4 \]
\[ \} \]
\[ t_5 \]
\[ \text{ping}() \{ \]
\[ t_6 \]
\[ \} \]
\[ t_7 \]
\[ t_8 \]

Call-sites

Matching continuations
Abstract Modelling: MLS, Lazy

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

\[ \begin{align*}
  t_0 \\
  t_1 \\
  \text{foo}() \{ \\
  \quad t_2 \\
  \quad \text{bar}() \{ \\
  \quad \quad t_3 \\
  \quad \} \\
  \} \\
  t_4 \\
\}
\]

\[ \begin{align*}
  t_5 \\
  \text{ping}() \{ \\
  \quad t_6 \\
  \} \\
  t_7 \\
  t_8
\]
Abstract Modelling: MLS, Lazy

t₀

t₁

foo() {
    t₂
    bar() {
        t₃
    }
    t₄
}

ping() {
    t₅
    t₆
}

}
Abstract Modelling: MLS, Lazy

Determine how long the call takes
Abstract Modelling: MLS, Lazy

Determine how long the call takes

Recursively process body

```plaintext
foo() {
    bar() {
        t3
    }
    t4
}
```
Abstract Modelling: MLS, Lazy

\[
t_0 \\
t_1 \\
\Rightarrow foo() \\
\quad t_2 \\
\quad \Rightarrow bar() \\
\quad \quad t_3 \\
\quad \} \\
\} \\
\Rightarrow t_4 \\
\} \\
\Rightarrow t_5 \\
ping() \\
\quad \} \\
\quad t_6 \\
\quad \} \\
\quad t_7 \\
\quad t_8
\]

- Determine how long the call takes
- Recursively process body
- Recursively process body
Abstract Modelling: MLS, Lazy

\[ t_0 \]
\[ t_1 \]
\[ \rightarrow \text{foo()} \{ \]
\[ t_2 \]
\[ \rightarrow \text{bar()} \{ \]
\[ t_3 \]
\[ } \]
\[ t_4 \]
\[ } \]
\[ t_5 \]

\[ \rightarrow \text{ping()} \{ \]
\[ t_6 \]
\[ } \]
\[ t_7 \]
\[ t_8 \]

Determine how long the call takes

Recursively process body

Recursively process body
Abstract Modelling: MLS, Lazy

Determine how long the call takes

Recursively process body

Recursively process body
Abstract Modelling: MLS, Lazy

t_0
t_1
foo() {
  t_2
  bar() {
    t_3
  }
  t_4
}
}
t_5
ping() {
  t_6
}
}
t_7
t_8

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
```

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
  - $T = t_i...t_j$
- 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!