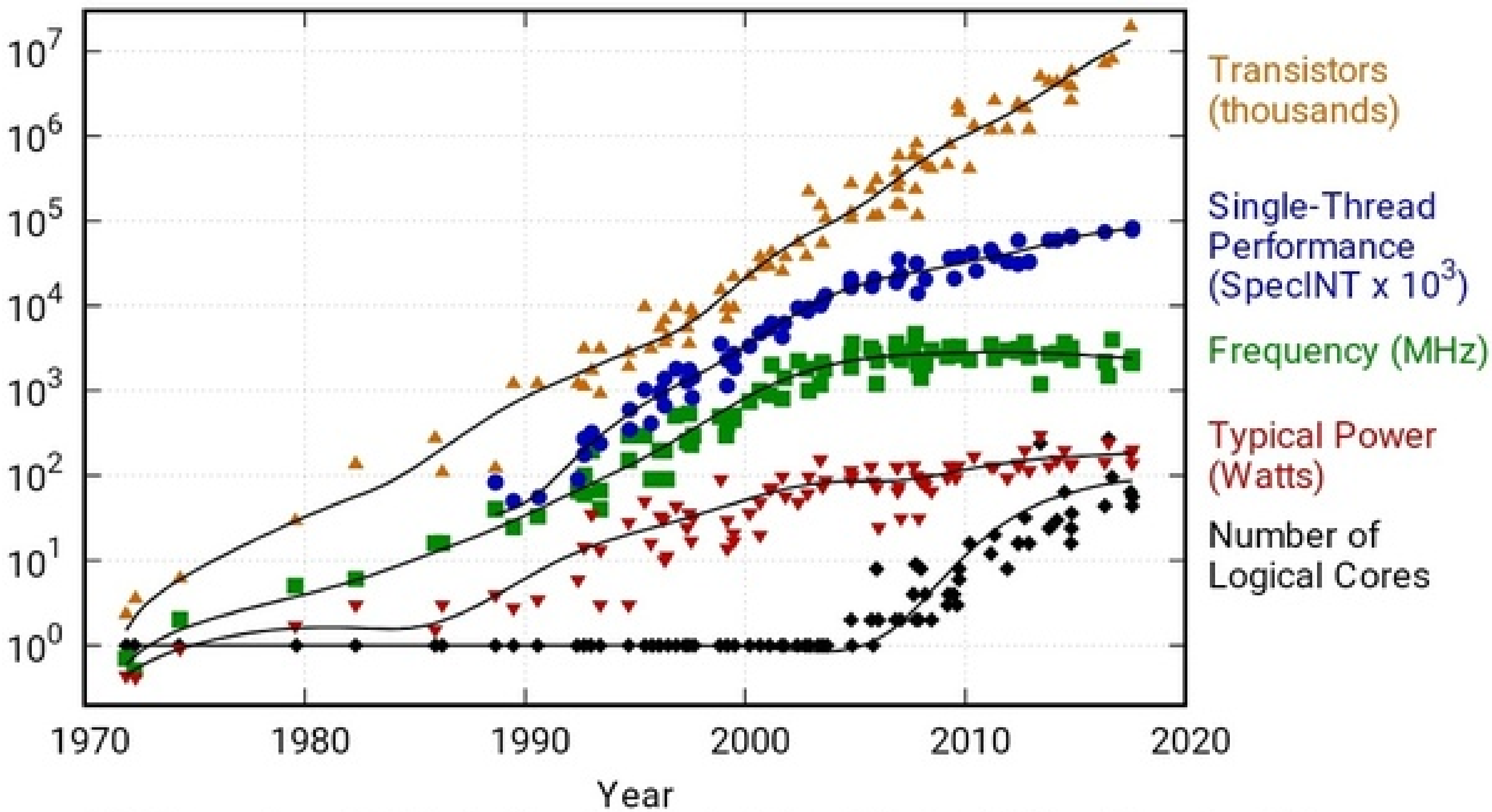
Threads by Compiler

15-411/15-611 Compiler Design

Seth Copen Goldstein

December 3, 2019

The Facts of Life in 2019

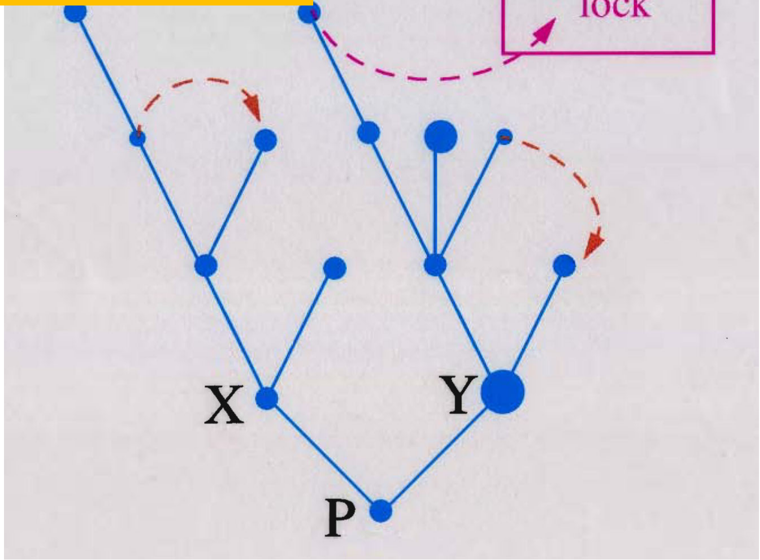


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

The Parallel Call

```
P() {
   fork X
   fork Y
   join
   ...
}
```

Too bad it’s expensive!



- Expressive
- Dynamic
- Synchronization dependencies
  ⇒ must support suspension
- Resource utilization
  ⇒ must support suspension

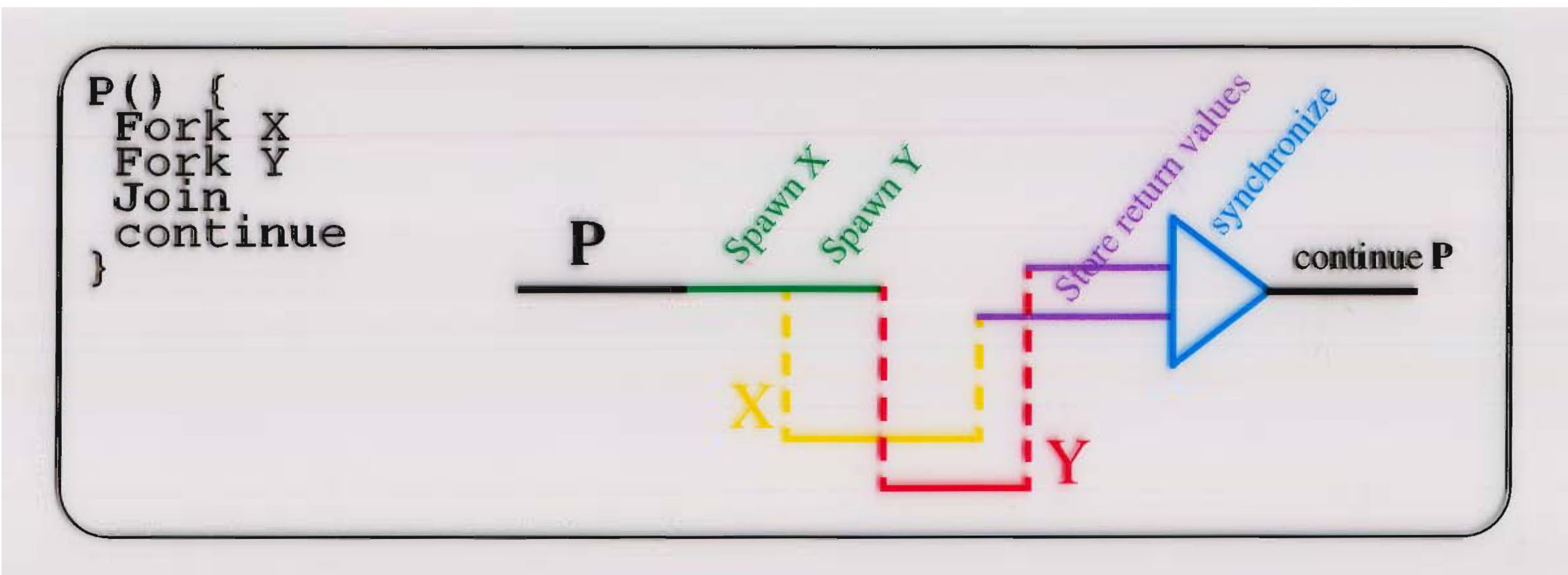
There is Hope

- Power of call is often not used
- Example:
  - All processors are b
  - Synchronization de
  - Lock is available
  - Data request is local
- Goal: zero overhead threading
- More realistically: Pay for what you need.

Why is zero-overhead Fork important?

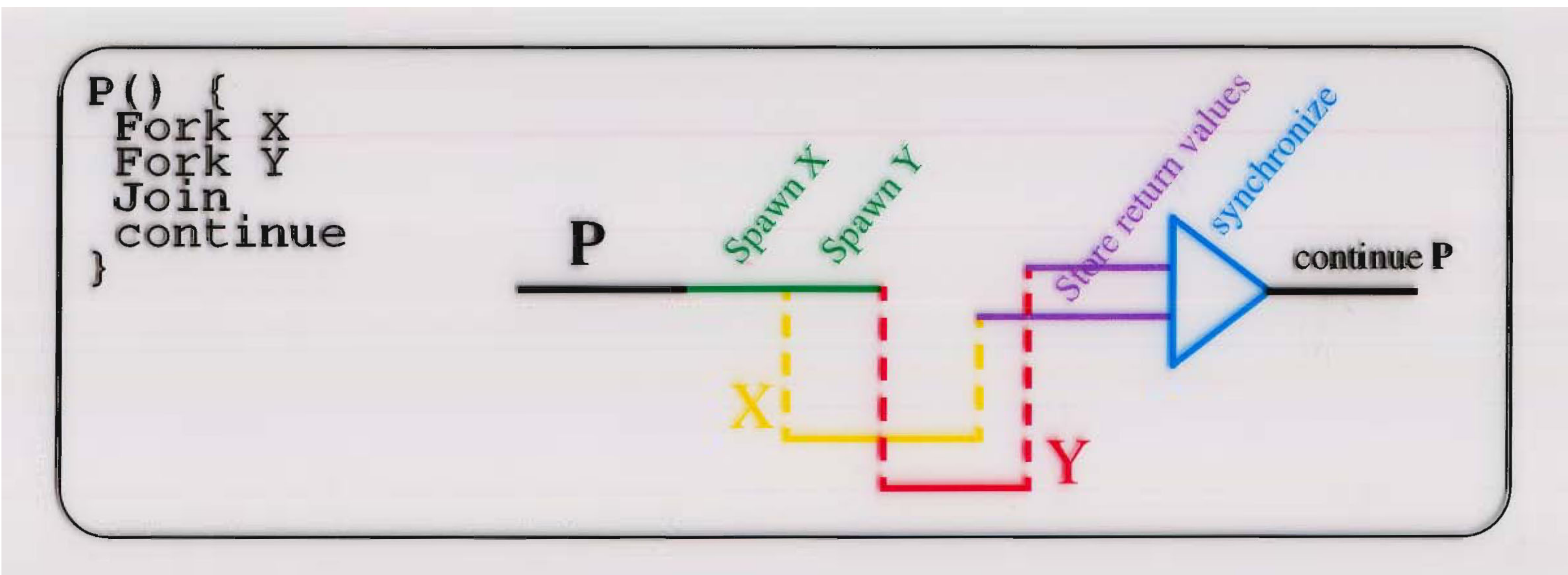
Parallel Call Inherently Expensive

Parallel	Sequential
• Parent and child run concurrently	



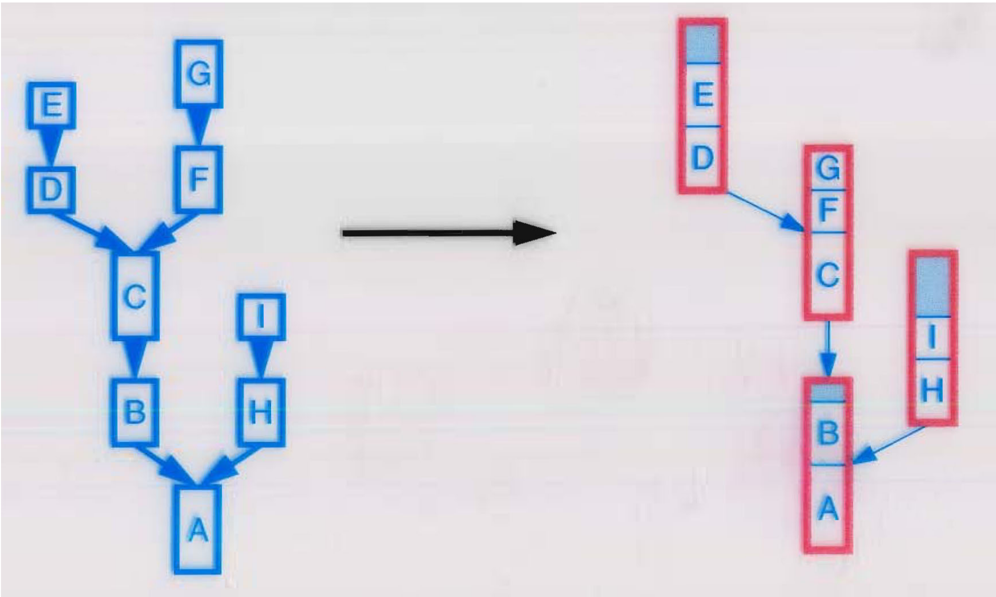
Parallel Call Inherently Expensive

Parallel	Sequential
• Parent and child run concurrently ⇒ Storage allocated on the heap • Data and control transferred separately • values passed through memory • explicit synchronization *always* needed.	• Parent suspends ⇒ use stack • Transferred together • Values in registers • No synchronization!

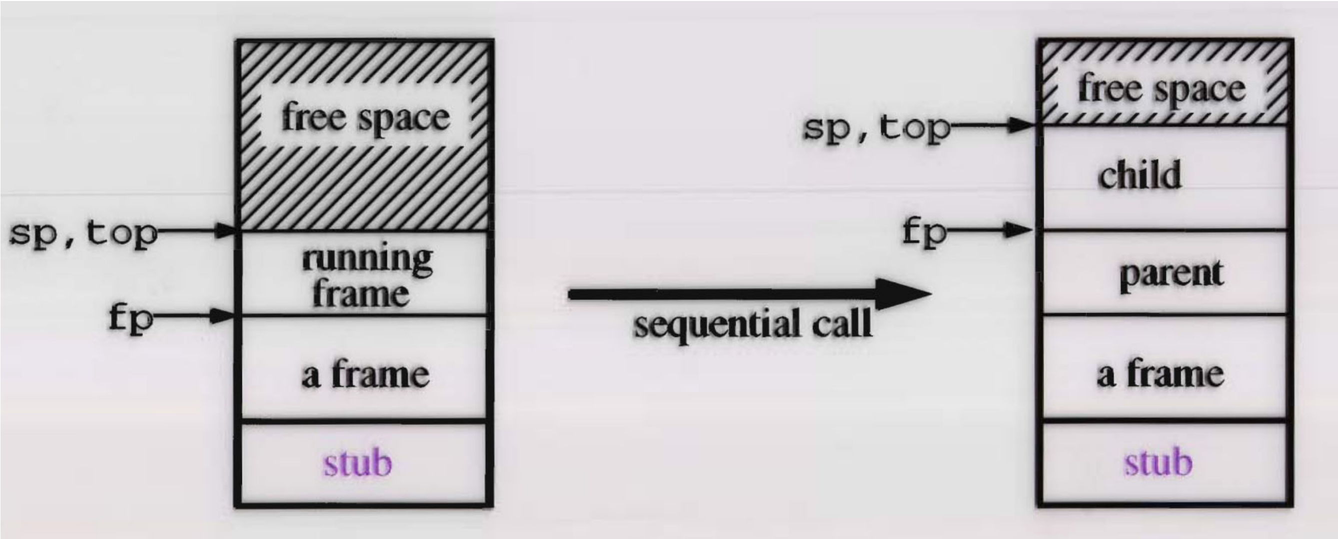


Storage Model

- Cactus Stack
- Invariants:
  - Child returns ⇒ All of its children are done
  - Child suspends ⇒ It currently has no work
  - Parent knows address of child on stack

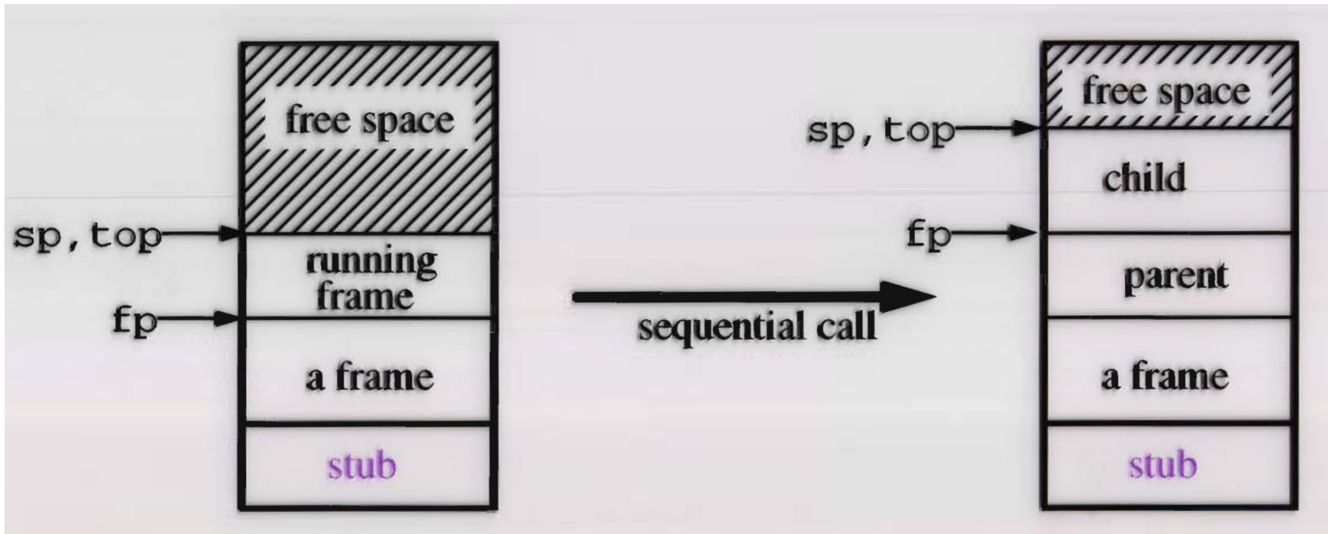


Sequential Call on Stacklet



Overhead?

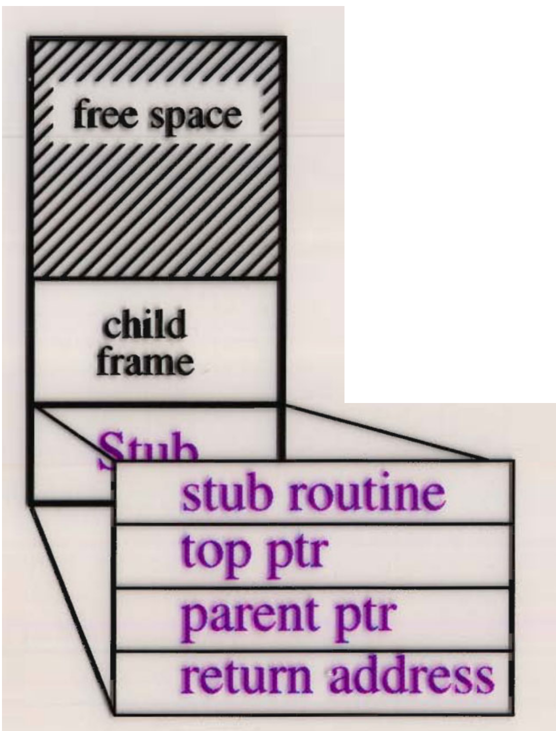
Sequential Call on Stacklet



- Only overhead due to checking for overflow
- Requires changes to the prolog
  (or, maybe not?)

Stubs

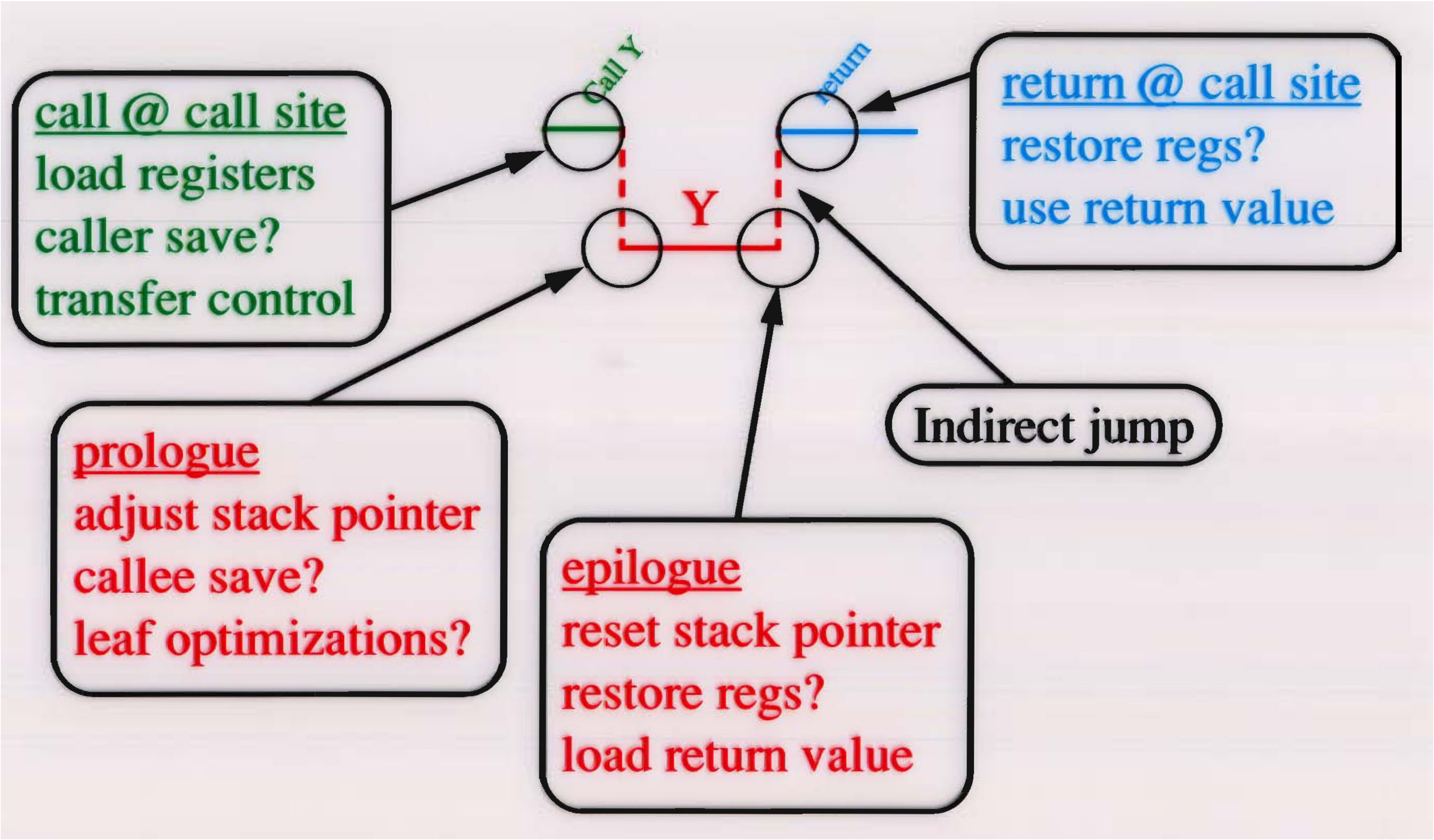
Stub routines handle the special cases



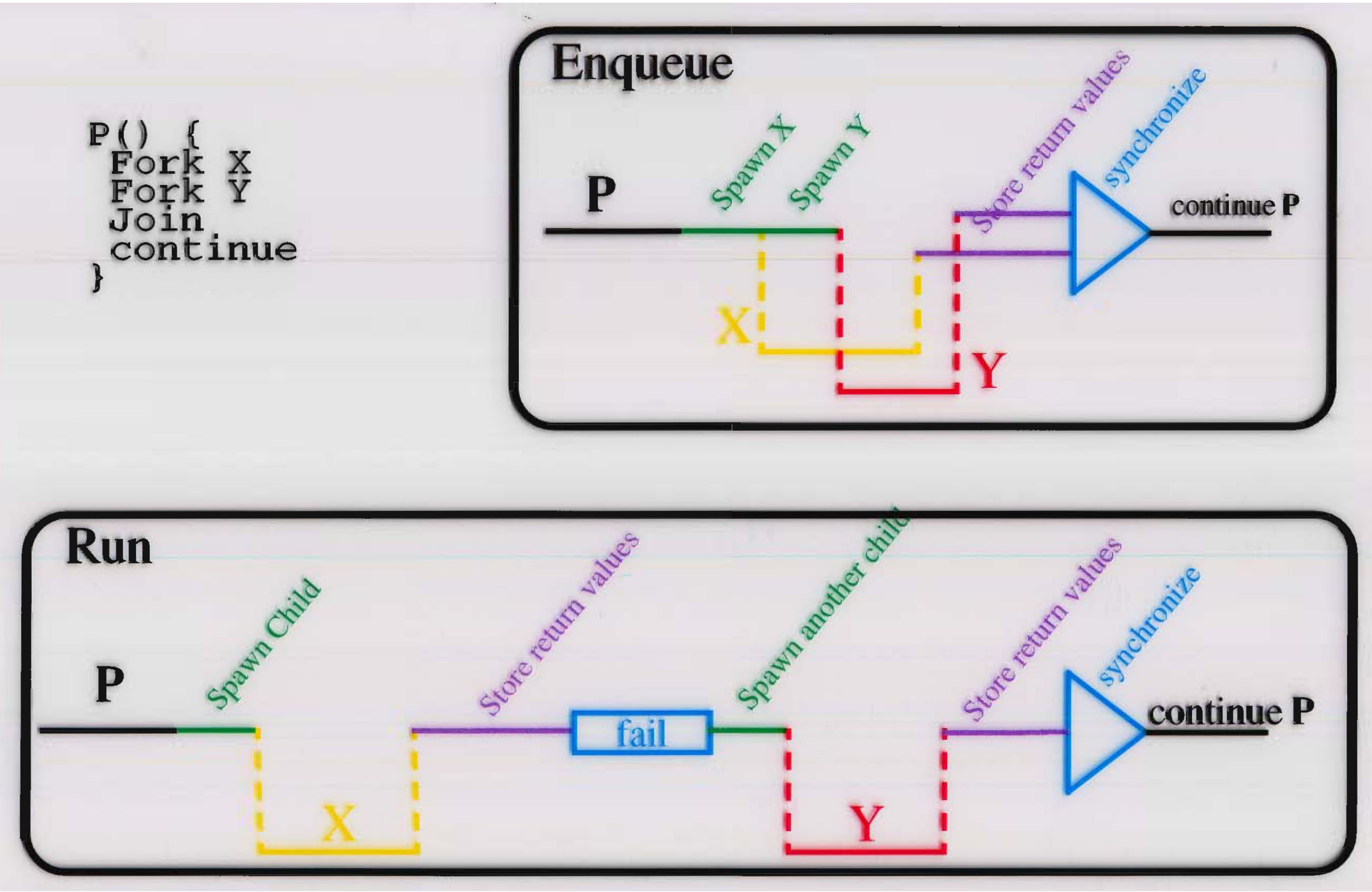
- Stacklet underflow
- Parallel return
- Remote return

Thus, no change in function epilog.

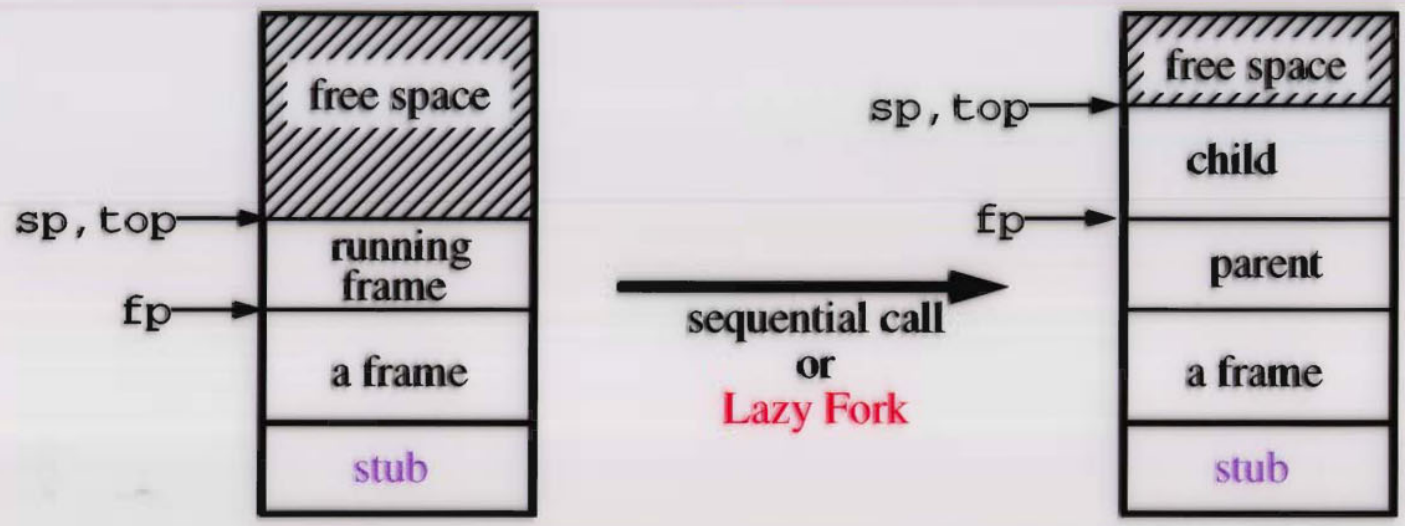
Compiler Focus



Behavior of parallel call on 1 proc

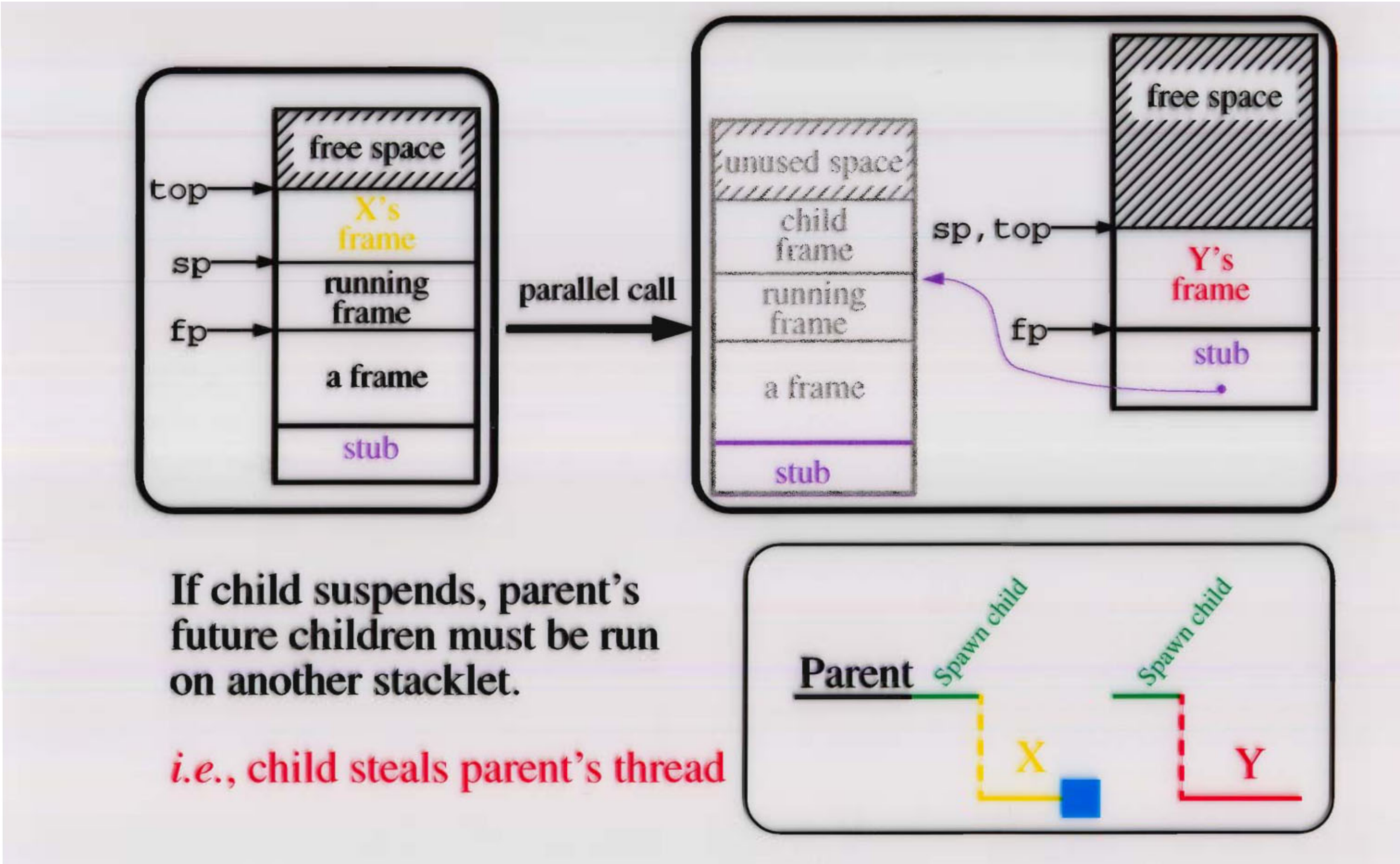


Lazy Fork



Lazy Fork starts off just like any sequential call.

Parallel Allocation



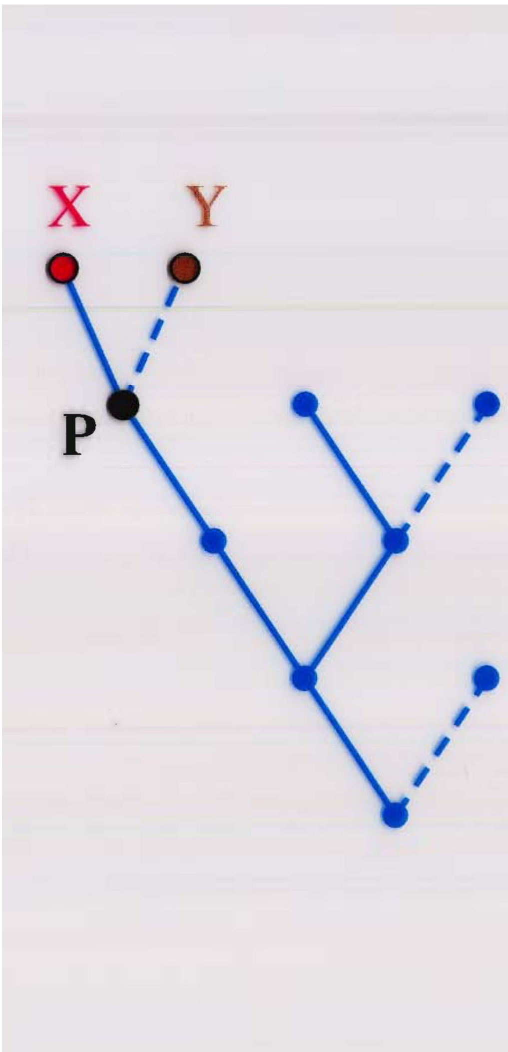
After the Lazy Fork

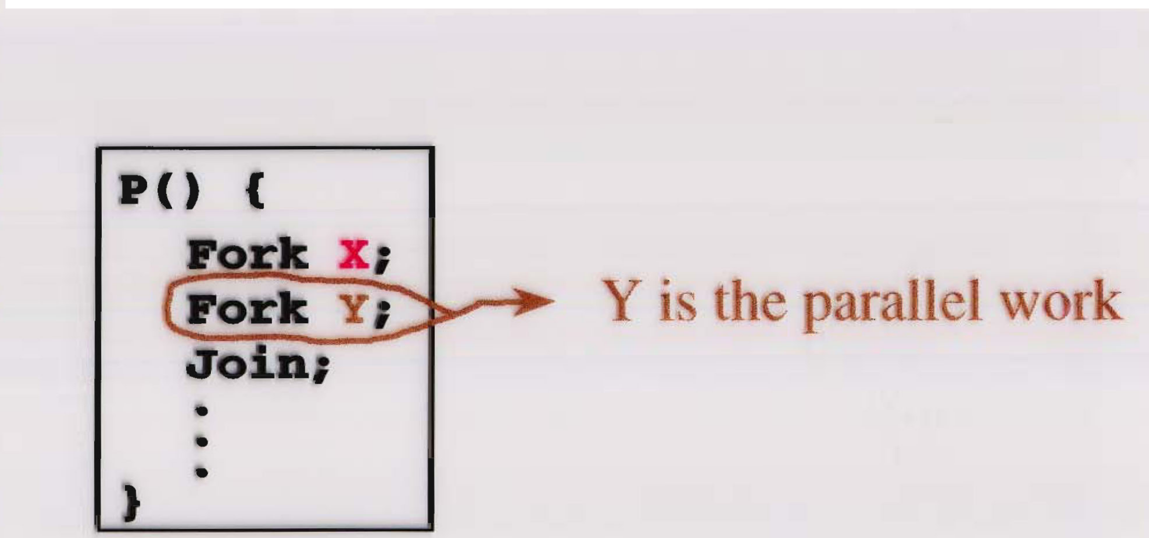
- What if parallelism is required?

- How is child converted into a thread?
- How is remaining work represented?
- How is it located?
- How is it distributed?

Representing Potential Parallel Work

- Parallel work can be encoded in return address
  $\Rightarrow$ Explicit Work Queue not needed
- However, Future Work $\neq$ Parallel Work

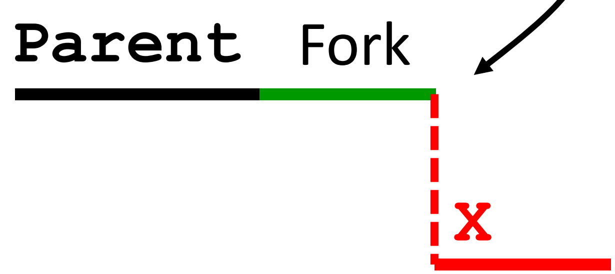




Using the Return Address

At this point in the execution, Y is ready to execute.

```
P() {
   Fork X(args);
   Fork Y(args);
   Join;
   :
}
```



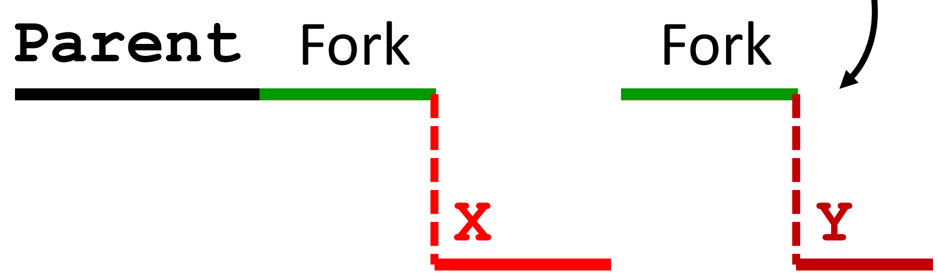
Future Work = Parallel Work

- return address serves 3 purposes:
  - saves results — sequential return
  - restores parent state — parallelism requires
  - continues execution at **Fork Y**

Using the Return Address

At this point in the execution, there is no parallel work.

```
P() {
   Fork X(args);
   Fork Y(args);
   Join;
   :
}
```



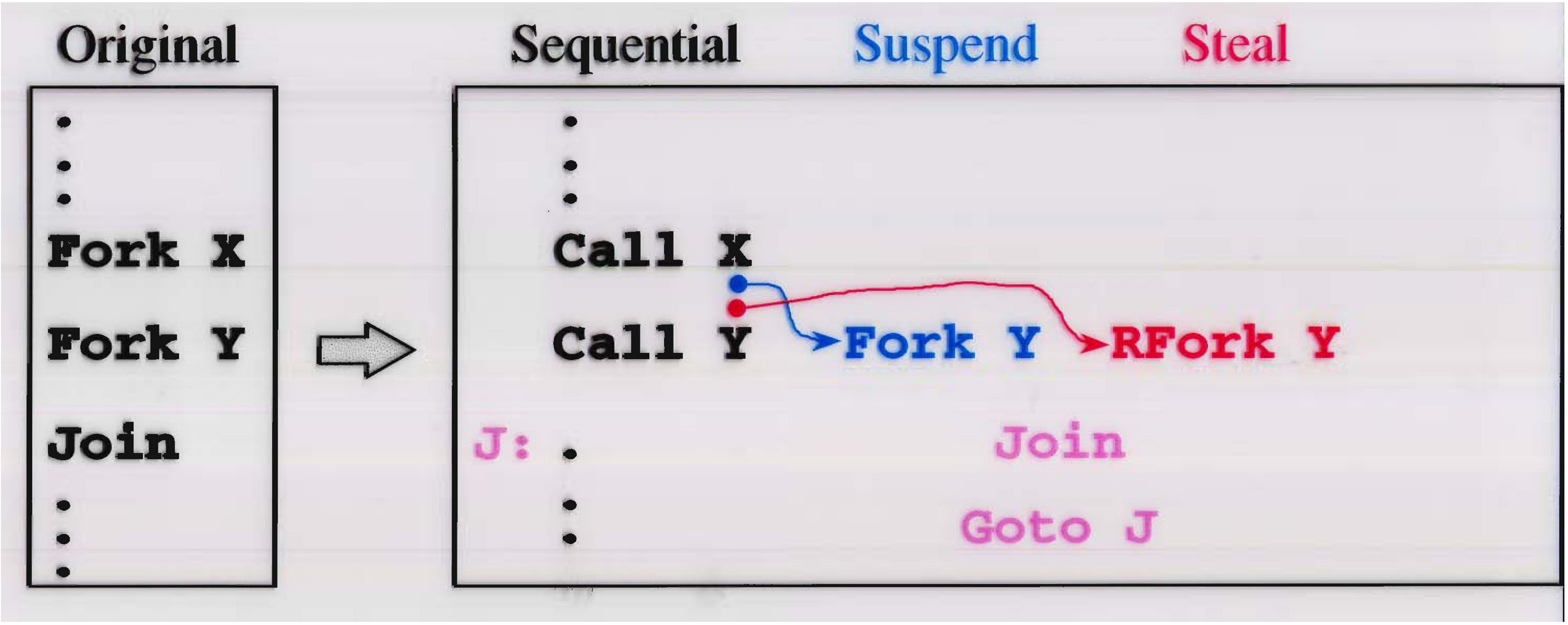
Future Work ≠ Parallel Work

- return address serves 3 purposes:
  - saves results
  - restores parent state
  - continues execution at `Join`

Thread Seeds

- Use return address to encode
  - Normal return
  - Suspension
  - Work stealing routine
- At time of call, parent plants a seed. Seed is dormant until:
  - Child returns $\Rightarrow$ seed inlined into parent
  - Child suspends $\Rightarrow$ seed is activated
  - Remote core grabs work
    $\Rightarrow$ seed is stolen

Code Gen Strategy

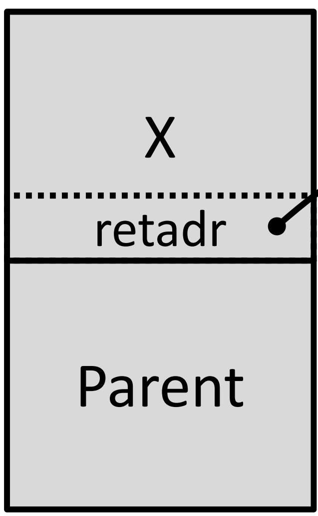


Create Three code streams:

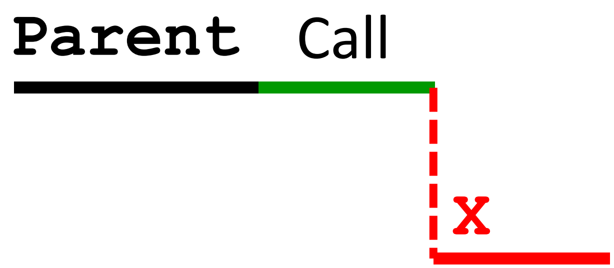
+ Eliminates unnecessary synchronization
+ Eliminates unnecessary bookkeeping
+ Optimizes for common case
- Increases code size

Using the Return Address

- Must encode state of parent:
  - Never had parallel work.

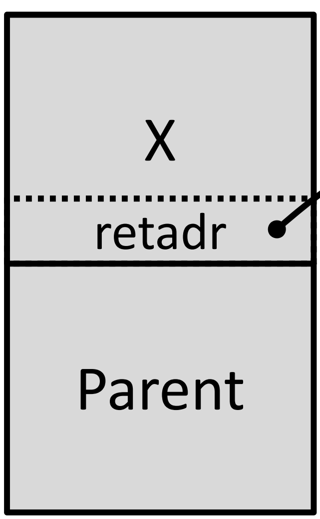



```
P() {
   X(args);
   Y(args);
   :
}
```

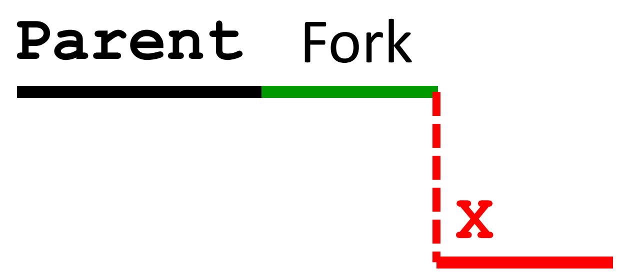


Using the Return Address

- Must encode state of parent:
  - Never had parallel work.
  - Has potentially parallel work.
    (currently has sequentially invoked child)

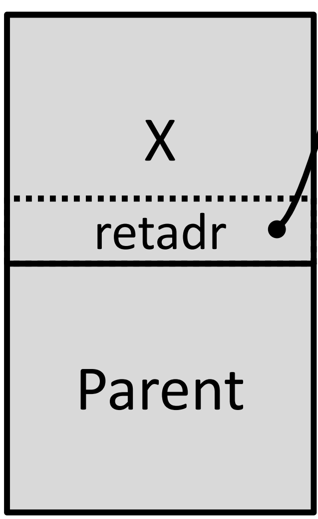



```
P() {
   Fork X(args);
   Fork Y(args);
   ⋮
}
```



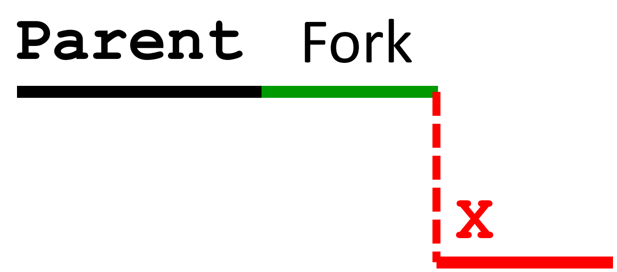
Using the Return Address

- Must encode state of parent:
  - Never had parallel work.
  - Has potentially parallel work.
    (currently has sequentially invoked child)
  - Has parallel work.
    (currently has converted sequentially invoked child)




```
//save result
if (--synch == 0) enq(P);
return;
```

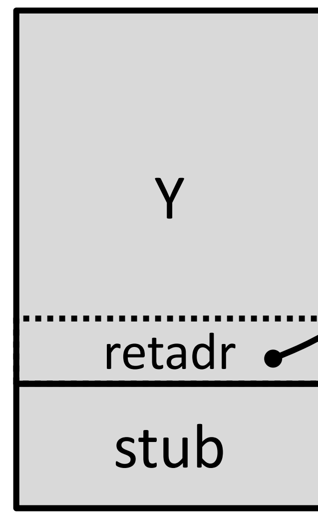
```
P() {
   Fork X(args);
   Fork Y(args);
   :
}
```



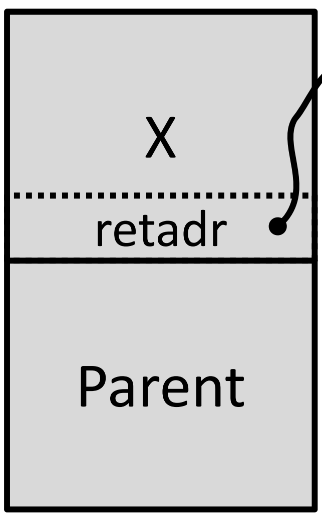
Using the Return Address

- Must encode state of parent:

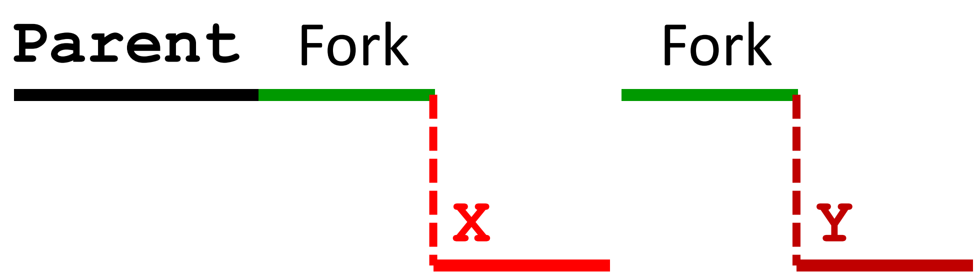
```
//save result
if (--synch == 0) enq(P);
return;
```



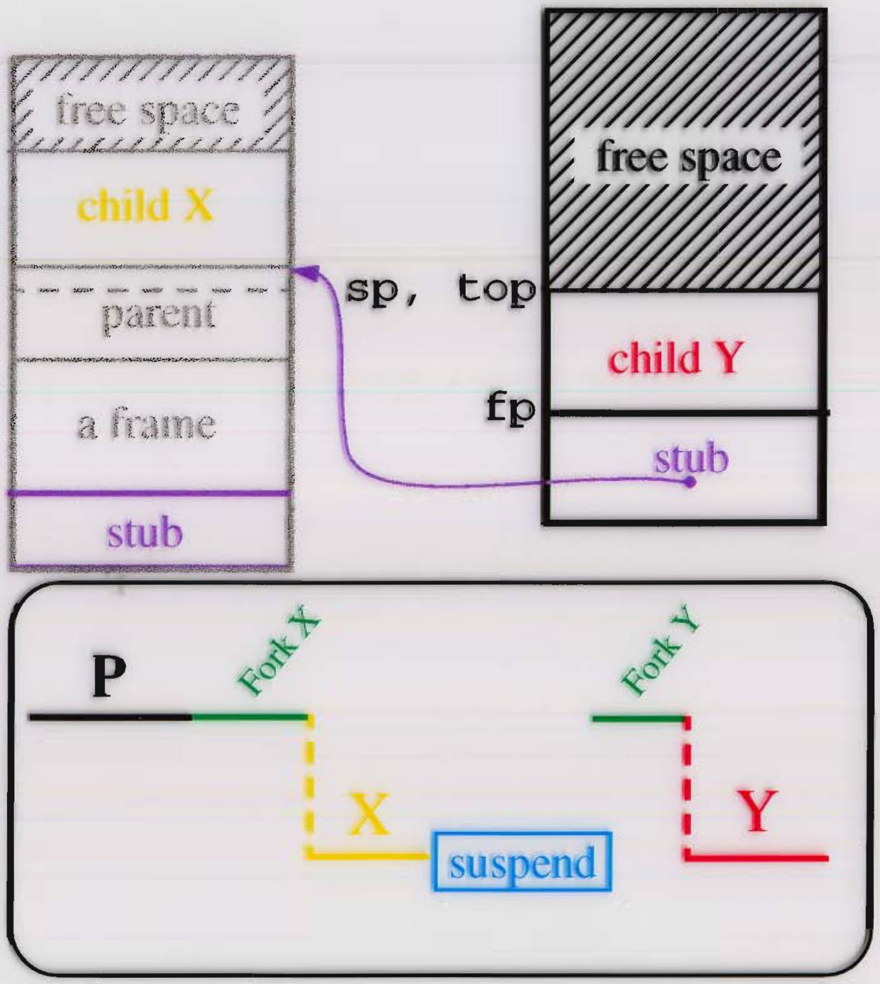
  - No more parallel work.
    (all children invoked, waiting at join point)




```
//save result
if (--synch == 0) enq(P);
return;
```



Parent Controlled Return Continuations



The suspension code spawns Y and then changes X's return address to reflect the new state of the parent.

Dealing with state

- In sequential context want **a**, **b**, **c**, **d**, … mapped to registers.

```
P() {
  c = Fork X(a);
  d = Fork Y(b);
  Join;
  ⋮
}
```

Dealing with state

- In sequential context want **a**, **b**, **c**, **d**, … mapped to registers.

```
P() {
  c = Fork X(a);
  d = Fork Y(b);
  Join;
  :
}
```

```
P() {
  c = call X(a);
  d = call Y(b);
  // Join;
  :
  // converted X ret
  c ← return value
  sync code
  // spawn Y
  change X's retadr
  Fork Y(b);
  d ← return value
  synch code
}
```

Dealing with state

- In sequential context want **a**, **b**, **c**, **d**, … mapped to registers.

```
P() {
  c = Fork X(a);
  d = Fork Y(b);
  Join;
  :
}
```

```
P() {
  c = call X(a);
  d = call Y(b);
  // Join;
  :
  // converted X ret
  c ← return value
  sync code
  // spawn Y
  change X's retadr
  Fork Y(b);
  d ← return value
  synch code
}
```

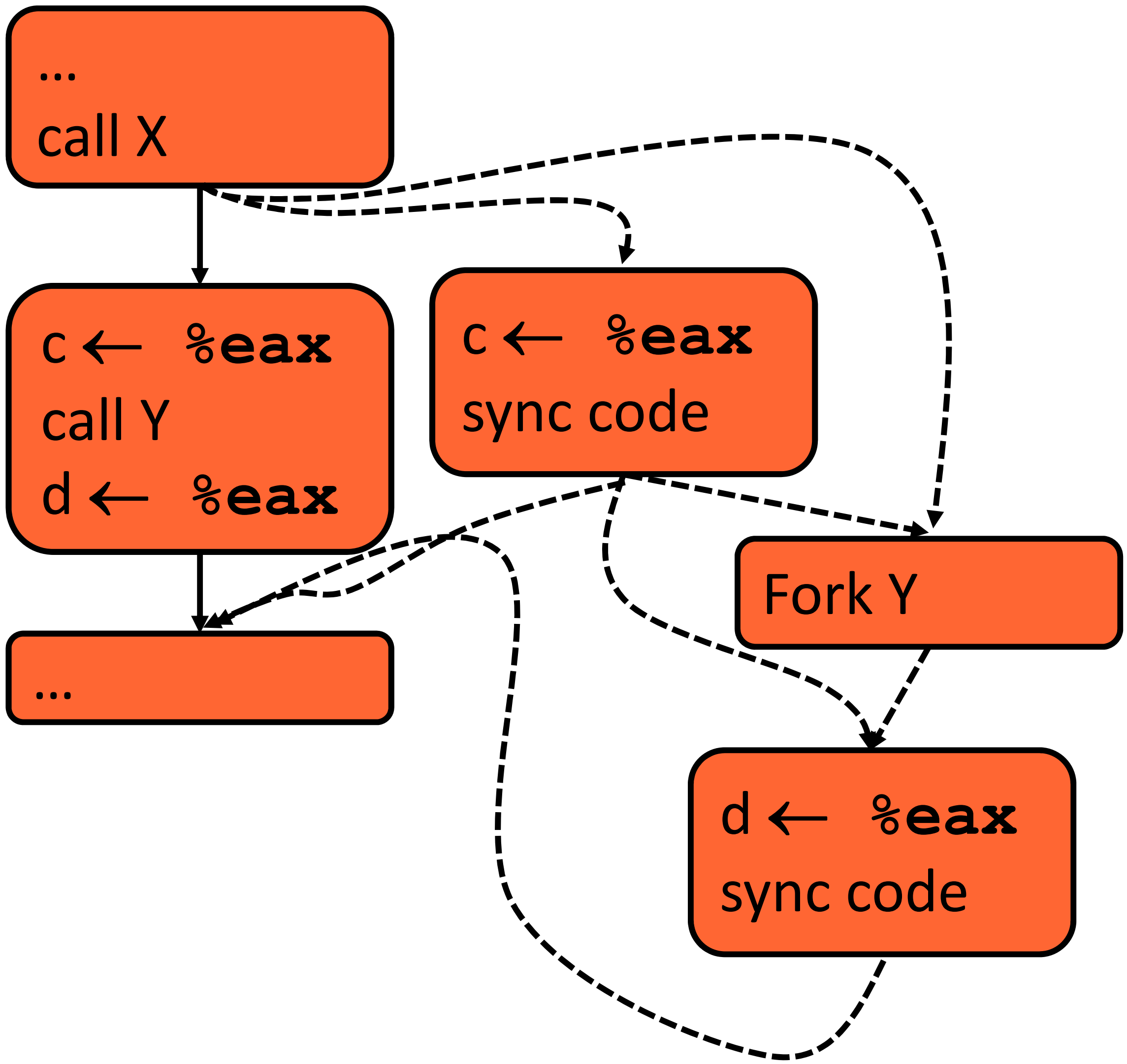
- However, if converted, **Fork Y** needs access to **b** and return from **X** needs to access **c**.

Original CFG

```
P() {
   c = call X(a);
   d = call Y(b);
   // Join;
   :
   :
   // converted X ret
   c ← return value
   sync code
   // spawn Y
   change X's retadr
   Fork Y(b);
   d ← return value
   synch code
}
```

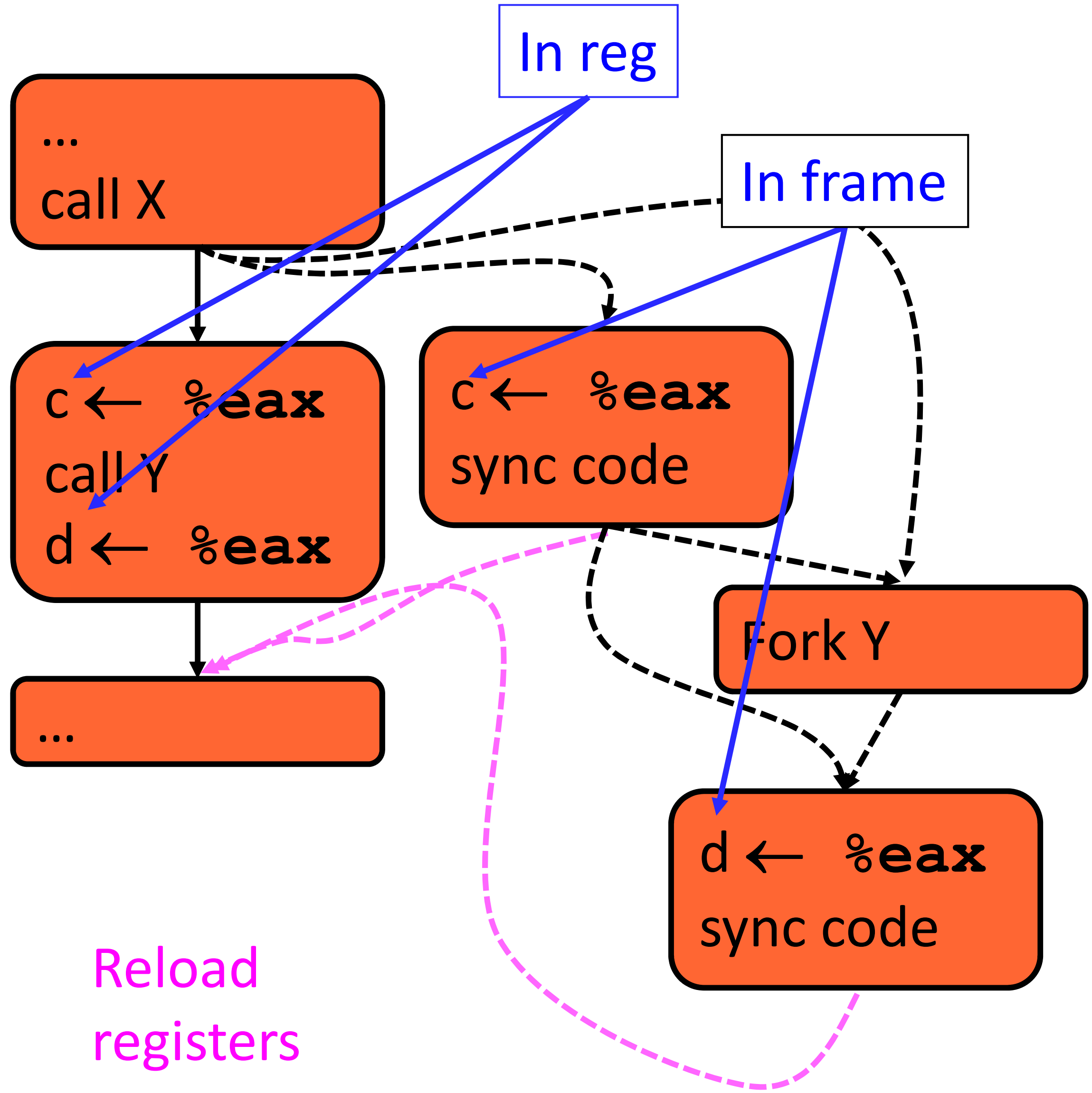
Modified CFG

```
P() {
   c = call X(a);
   d = call Y(b);
   // Join;
   :
   :
   // converted X ret
   c ← return value
   sync code
   // spawn Y
   change X's retadr
   Fork Y(b);
   d ← return value
   synch code
}
```



Modified CFG

```
P() {
   c = call X(a);
   d = call Y(b);
   // Join;
   :
   :
   // converted X ret
   c ← return value
   sync code
   // spawn Y
   change X's retadr
   Fork Y(b);
   d ← return value
   synch code
}
```

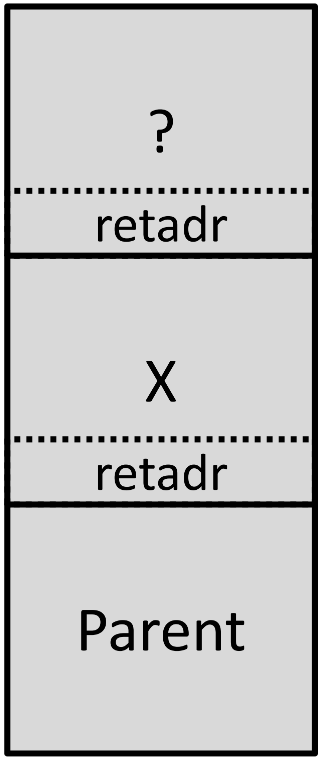


Implementing the Seed

- Every return address has an entry in a per procedure table mapping the retadr to:
  - conversion routine
  - local fork routine
  - remote fork routine
  - Information about stack size, registers, etc.
- Runtime looks at top of stack
  - locates most recent return address
  - finds entry in table
  - executes appropriate code using appropriate frame pointer

Compiled setjmp, longjmp

```
P() {
   c = call X(a);
   d = call Y(b);
   // Join;
   :
   // converted X ret
   c ← return value
   sync code
   // spawn Y
   change X's retadr
   Fork Y(b);
   d ← return value
   synch code
}
```



Synchronization

- There is a big data race: the return address!
- Locking on every call and return violates "pay for what you use"
- Possible Solutions:
  - user-level interrupts and message passing
  - fancy tricks with atomic xchg and do explicit indirect jmp for return

Summary

- Key: Only create parallel work when necessary.
- Exposing threads to the compiler is essential for high-performance fine-grained multi-threading.
- There are three axis to consider:
  - How to store thread state
  - Representation of (potentially) parallel work
  - How to convert potentially parallel work to actual parallel work
- Exploit compiler
- Exploit indirect jump

Logistics

- Lab6 due next week
- NO EXTENSIONS!
- FCEs
- More detailed feedback

- A very big Thank-you!