6.172
Performance
Engineering
of Software
Systems

LECTURE 13

The Cilk Runtime System

Tao B. Schardl

Recall: Cilk Programming

Cilk allows programmers to make software run faster using parallel processors.

Serial matrix multiply

```
for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Running time T_S.

Cilk matrix multiply

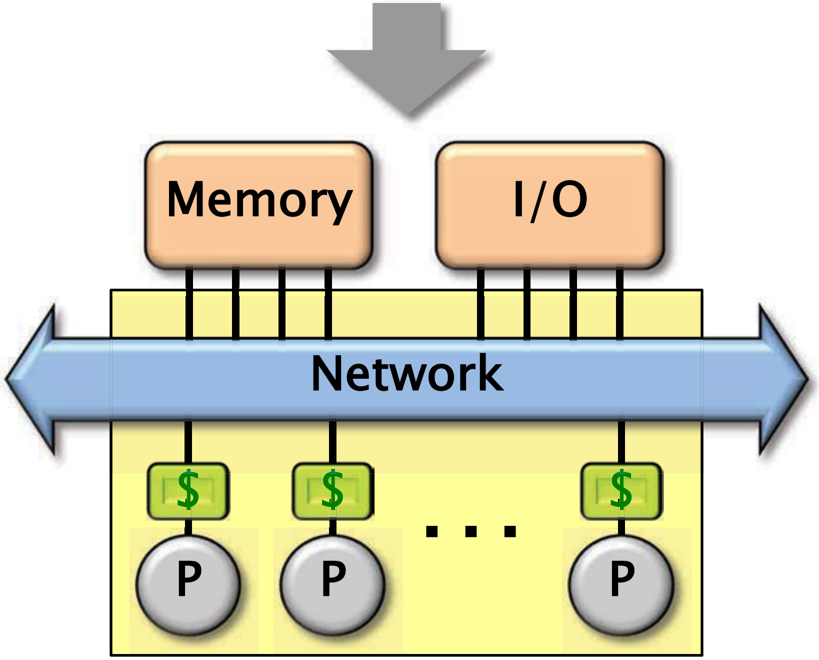
```
cilk_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Running time T_P on P processors.

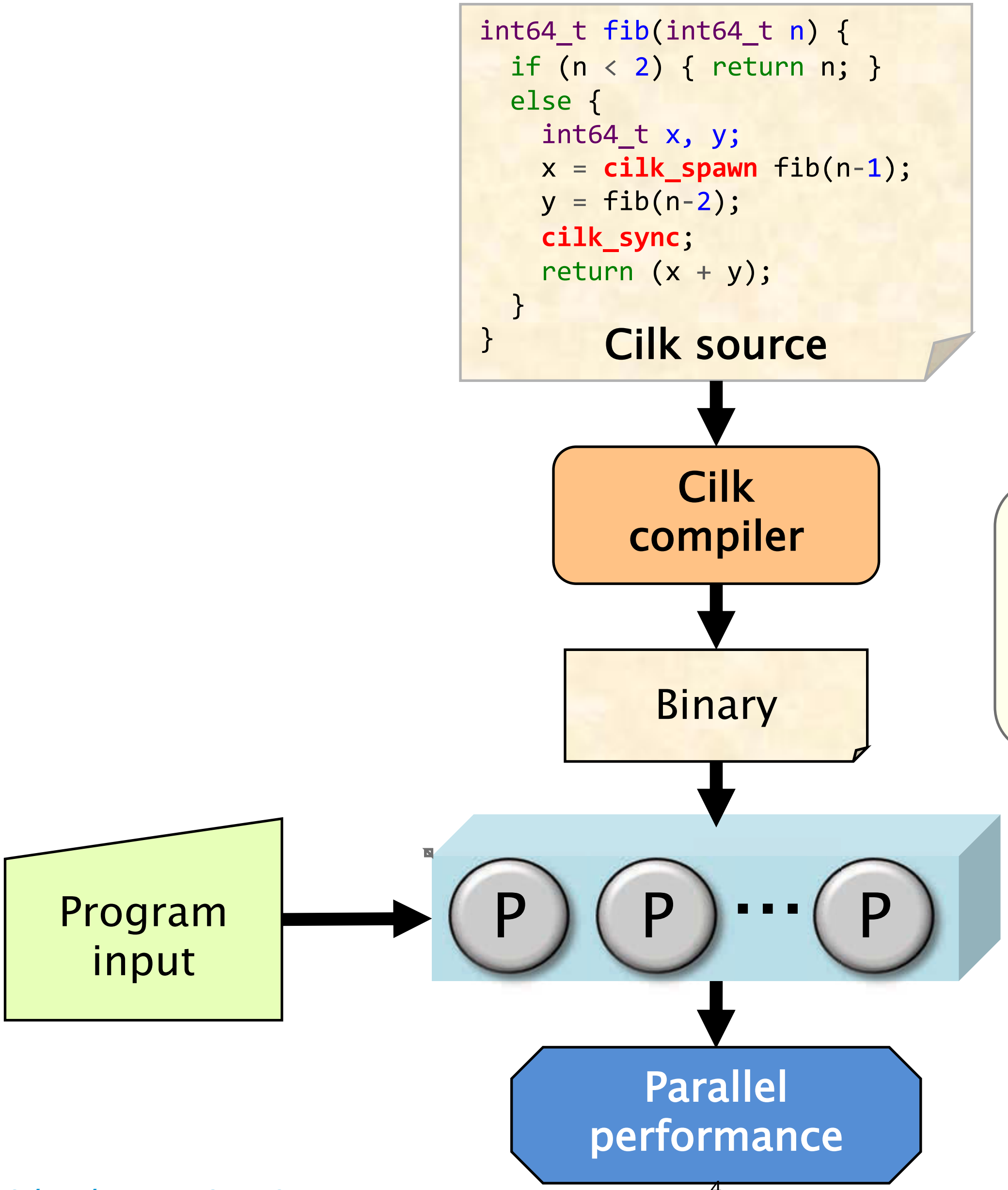
Recall: Cilk Scheduling

- The Cilk concurrency platform allows the programmer to express logical parallelism in an application.
- The Cilk scheduler maps the executing program onto the processor cores dynamically at runtime.
- Cilk's work-stealing scheduling algorithm is provably efficient.

```
int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```



Recall: Cilk Platform



```
int64_t fib(int64_t n) {
  if (n < 2) { return n; }
  else {
    int64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```

Cilk source

Cilk compiler

Binary

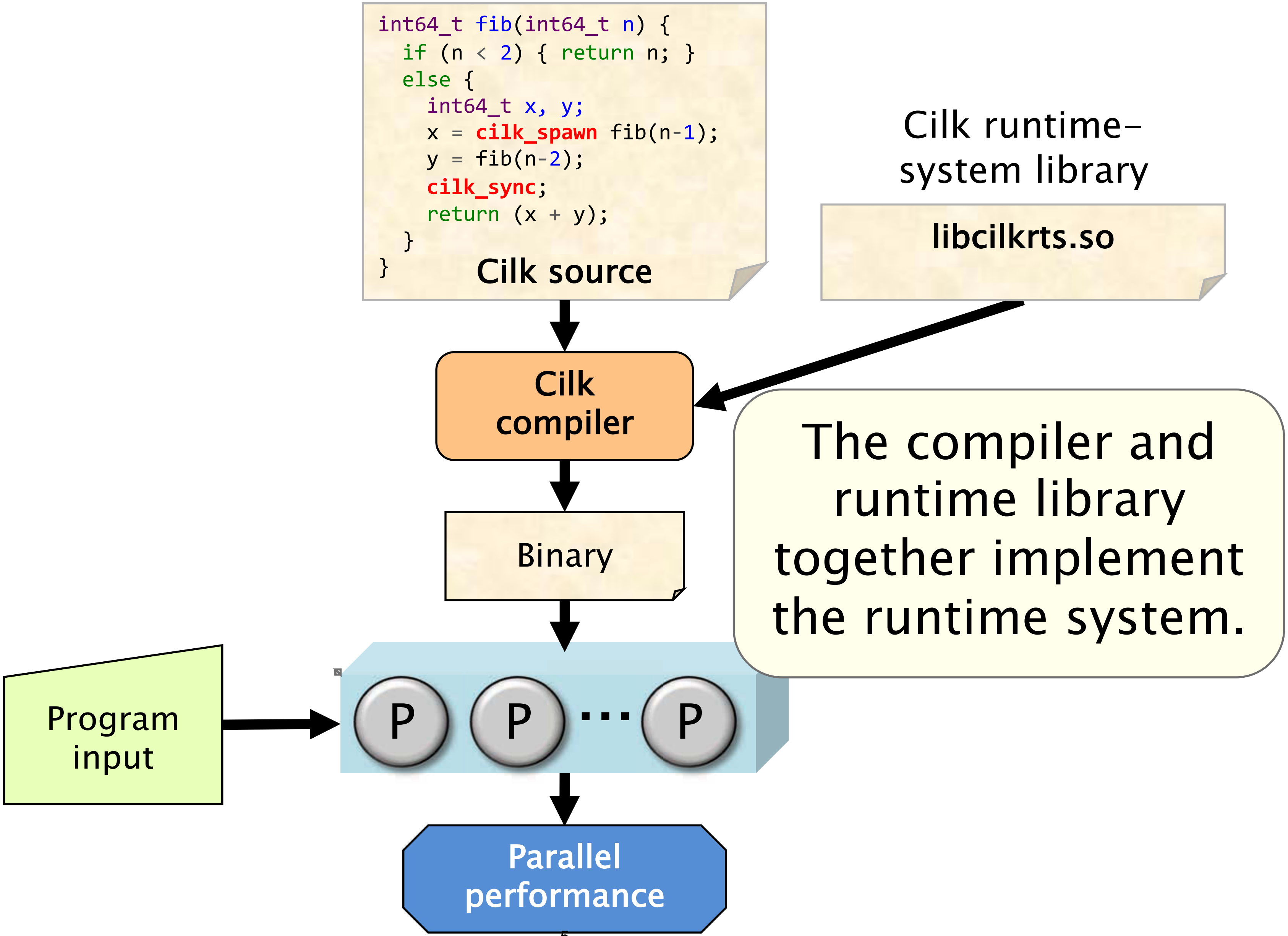
Program input

P P ⋯ P

Parallel performance

This lecture:
How does Cilk work?

A More Accurate Picture



```
int64_t fib(int64_t n) {
  if (n < 2) { return n; }
  else {
    int64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```

Cilk source

Cilk runtime-system library

libcilkrts.so

Cilk compiler

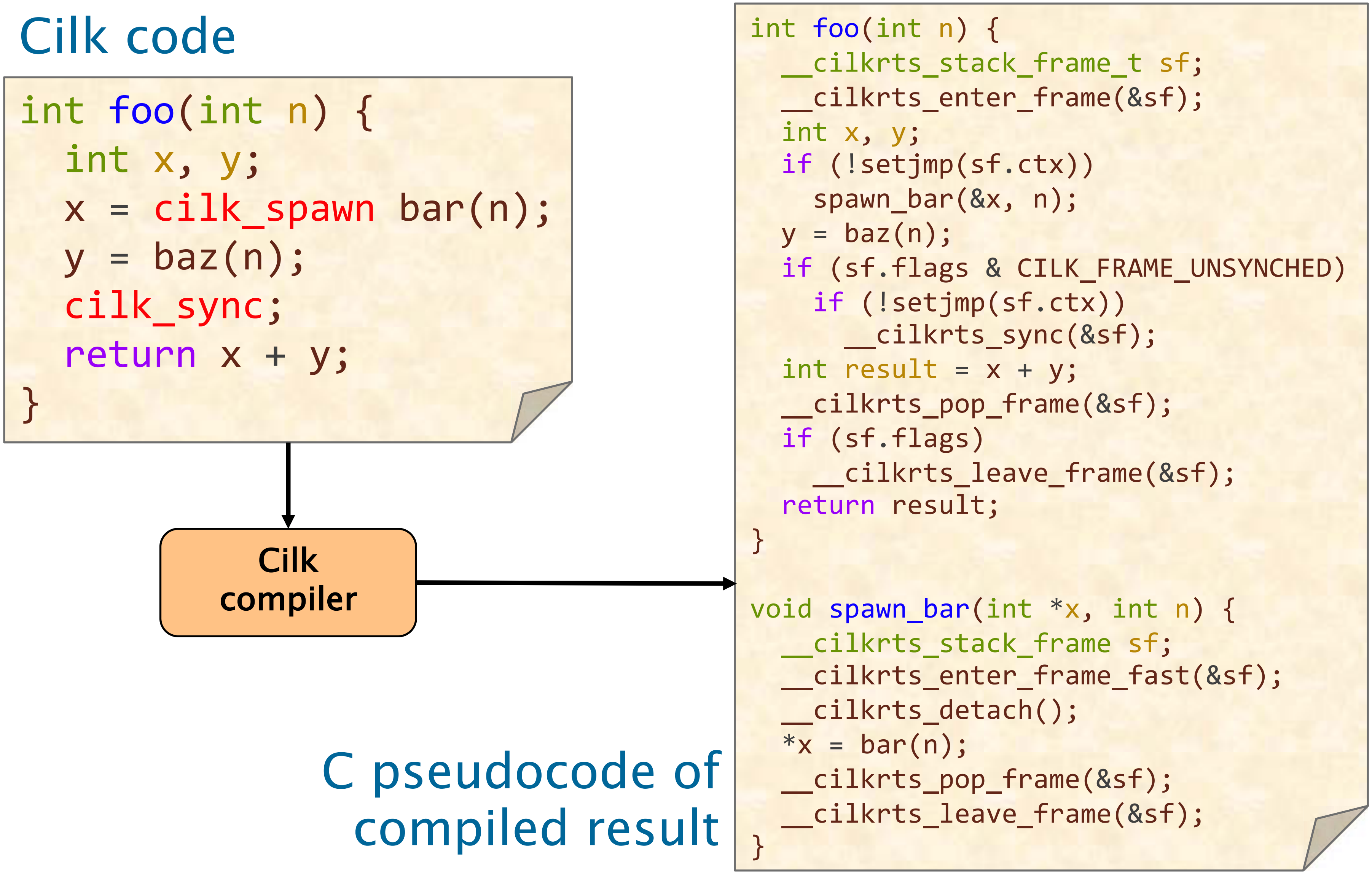
Binary

Program input

Parallel performance

The compiler and runtime library together implement the runtime system.

What the Compiler Generates



Cilk code

```
int foo(int n) {
  int x, y;
  x = cilk_spawn bar(n);
  y = baz(n);
  cilk_sync;
  return x + y;
}
```

Cilk compiler

C pseudocode of compiled result

```
int foo(int n) {
  __cilkrts_stack_frame_t sf;
  __cilkrts_enter_frame(&sf);
  int x, y;
  if (!setjmp(sf.ctx))
    spawn_bar(&x, n);
  y = baz(n);
  if (sf.flags & CILK_FRAME_UNSYNCHED)
    if (!setjmp(sf.ctx))
      __cilkrts_sync(&sf);
  int result = x + y;
  __cilkrts_pop_frame(&sf);
  if (sf.flags)
    __cilkrts_leave_frame(&sf);
  return result;
}

void spawn_bar(int *x, int n) {
  __cilkrts_stack_frame sf;
  __cilkrts_enter_frame_fast(&sf);
  __cilkrts_detach();
  *x = bar(n);
  __cilkrts_pop_frame(&sf);
  __cilkrts_leave_frame(&sf);
}
```

Outline

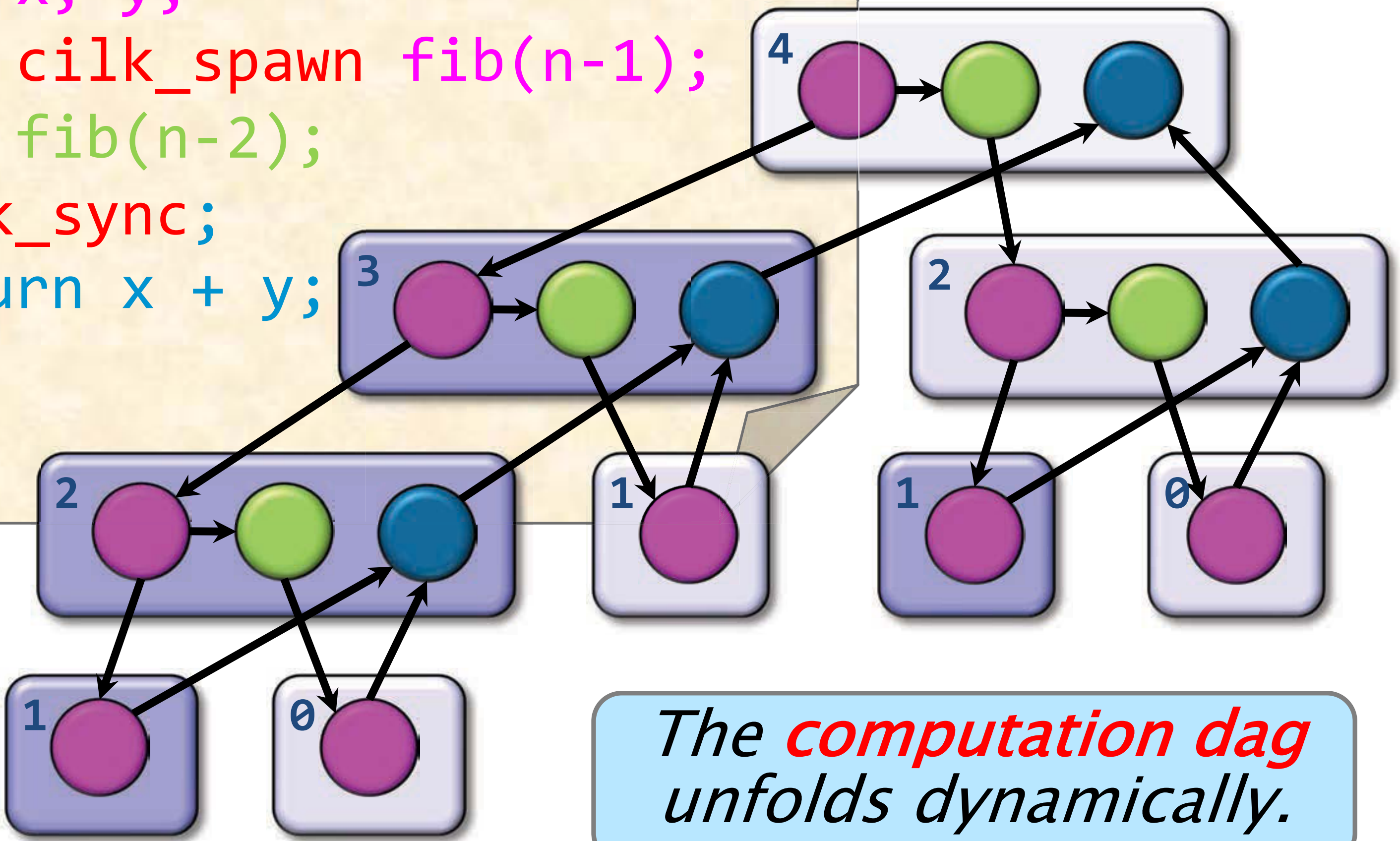
- REQUIRED FUNCTIONALITY
- PERFORMANCE CONSIDERATIONS
- IMPLEMENTING A WORKER DEQUE
- SPAWNING COMPUTATION
- STEALING COMPUTATION
- SYNCHRONIZING COMPUTATION

REQUIRED FUNCTIONALITY

Recall: Execution Model

```
int fib (int n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
  }
}
```

Example:
`fib(4)`



*The **computation dag** unfolds dynamically.*

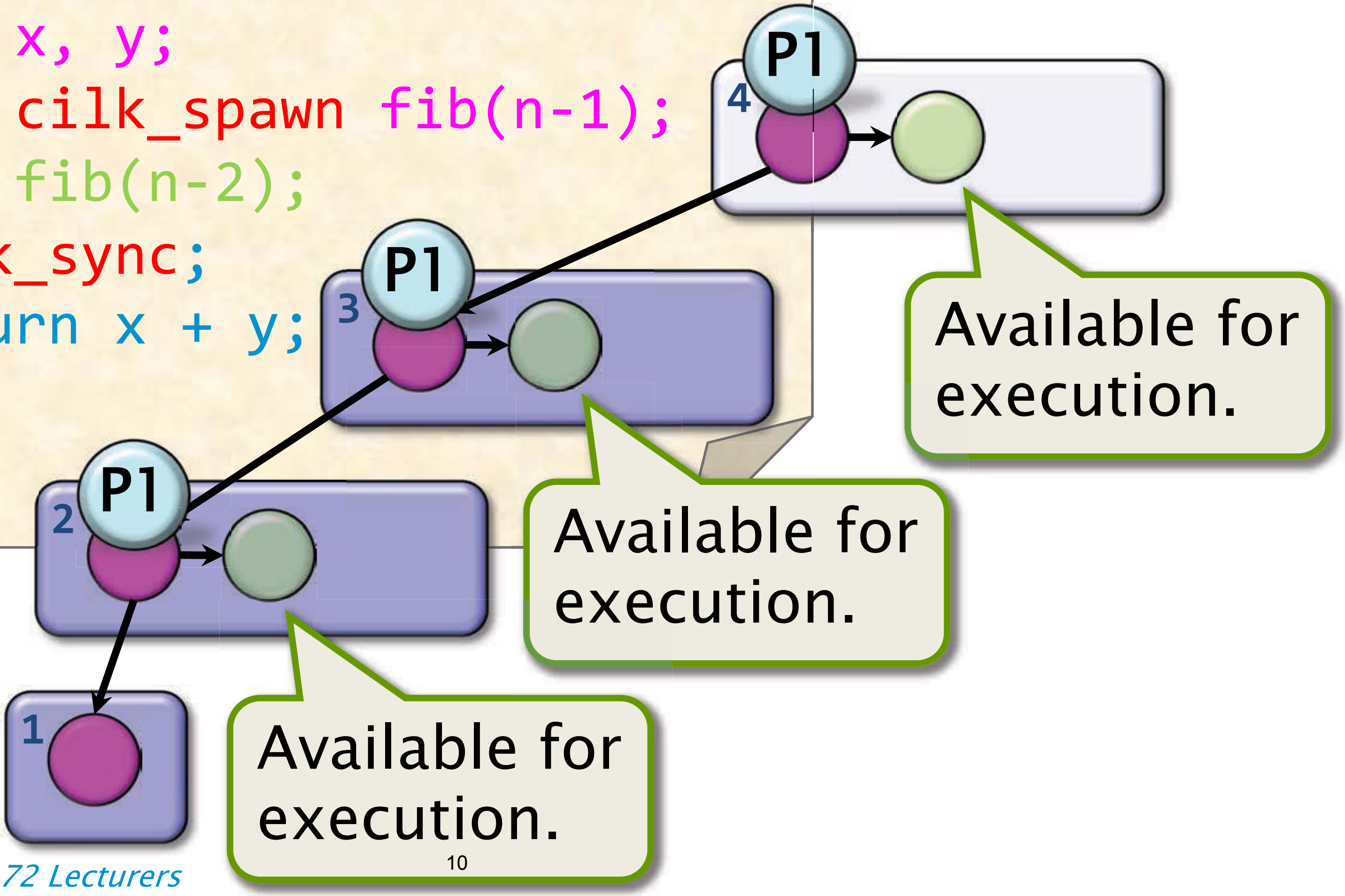
Serial Execution

P1

%rip →

```
int fib (int n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
  }
}
```

Example:
`fib(4)`



Available for execution.

Available for execution.

Available for execution.

Parallel Execution: Steals

P3

%rip P1

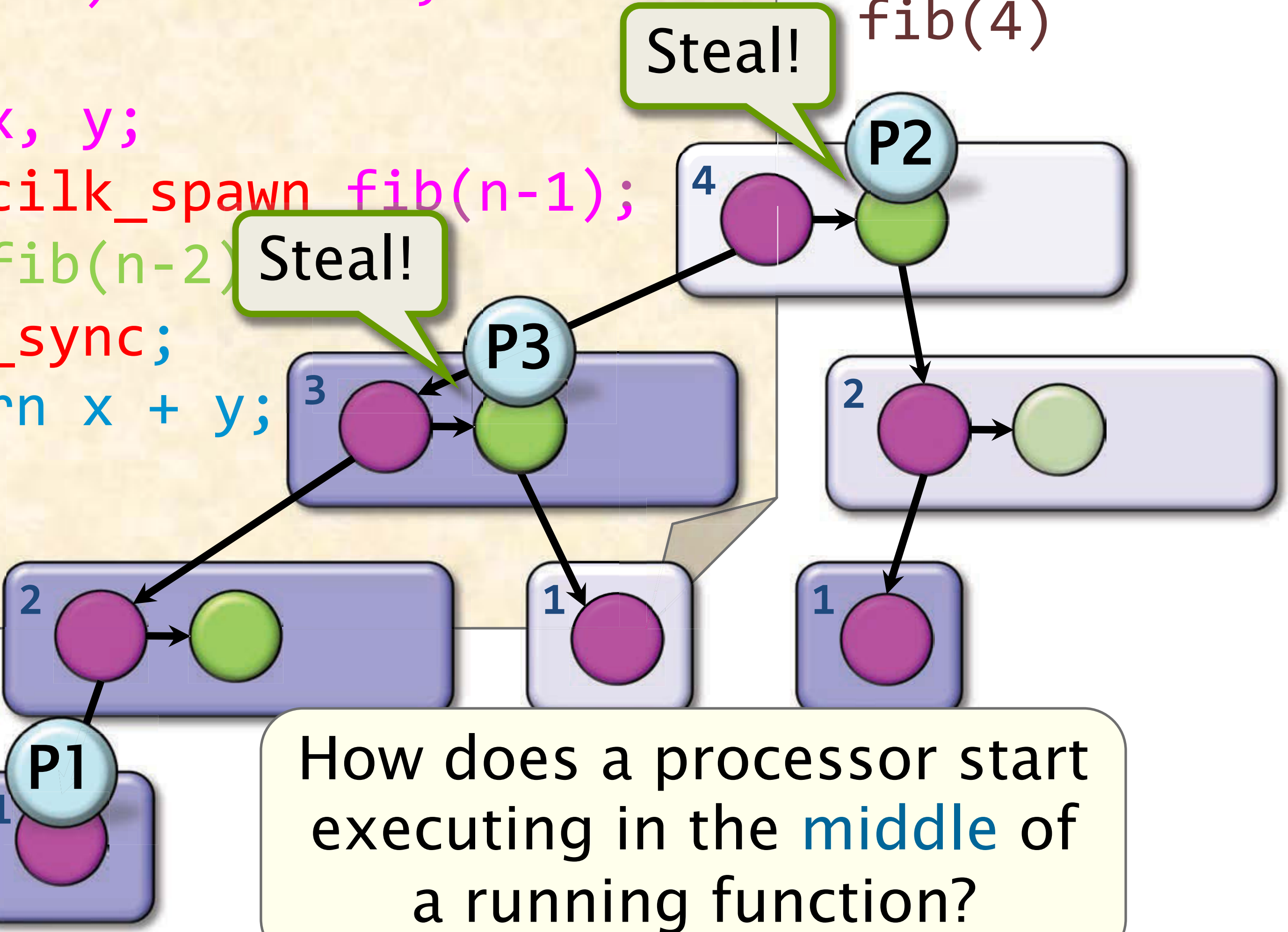
%rip

P3

%rip

```
int fib (int n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
  }
}
```

Example:
fib(4)



How does a processor start executing in the middle of a running function?

Parallel Execution: Syncs

P3 %rip → int fib (int n) {

P2 %rip → x = cilk_spawn fib(n-1);

P1 %rip → y = fib(n-2);

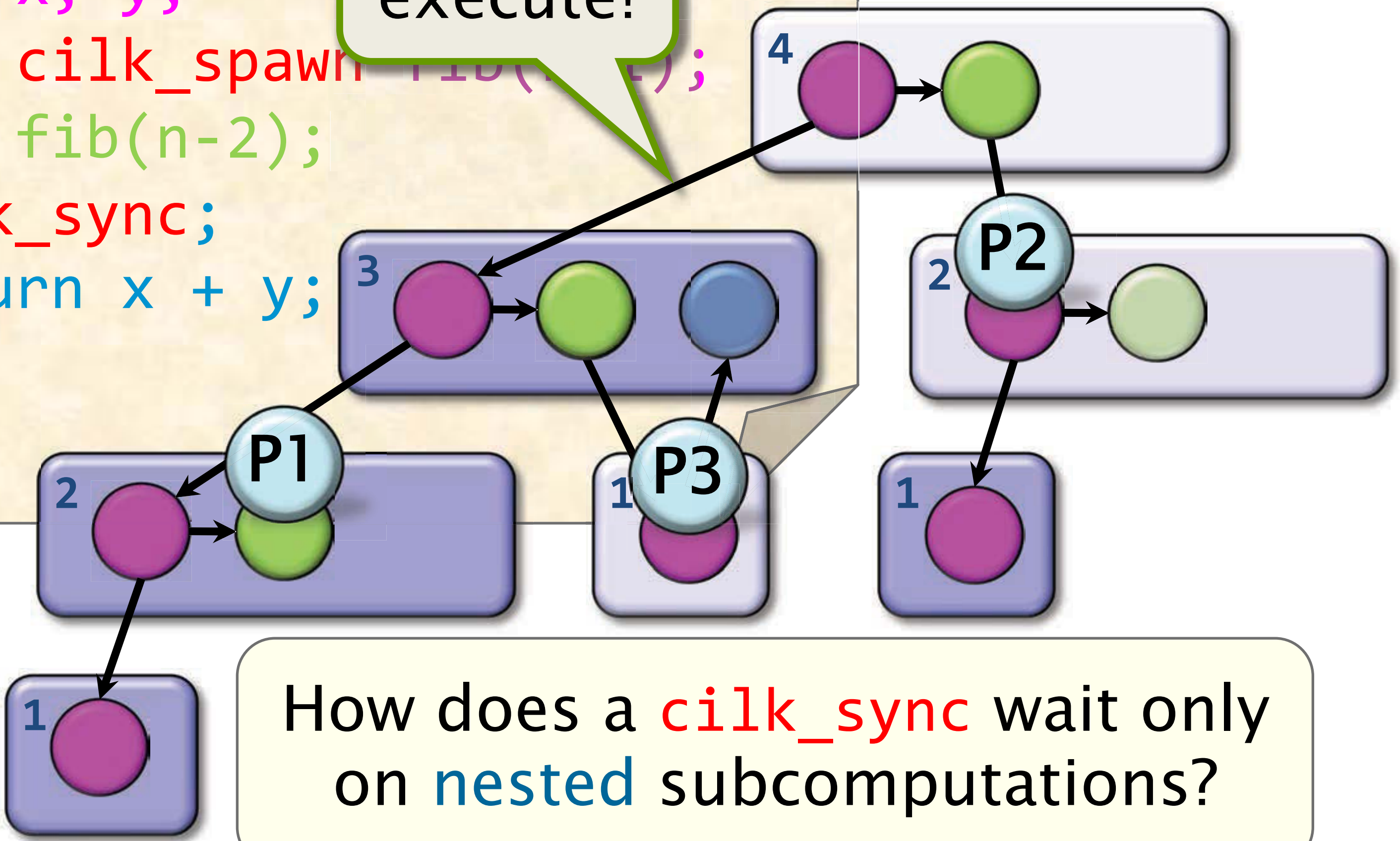
P3 %rip → cilk_sync;

```
int fib (int n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
  }
}
```

Example:

`fib(4)`

Can't execute!



How does a `cilk_sync` wait only on nested subcomputations?

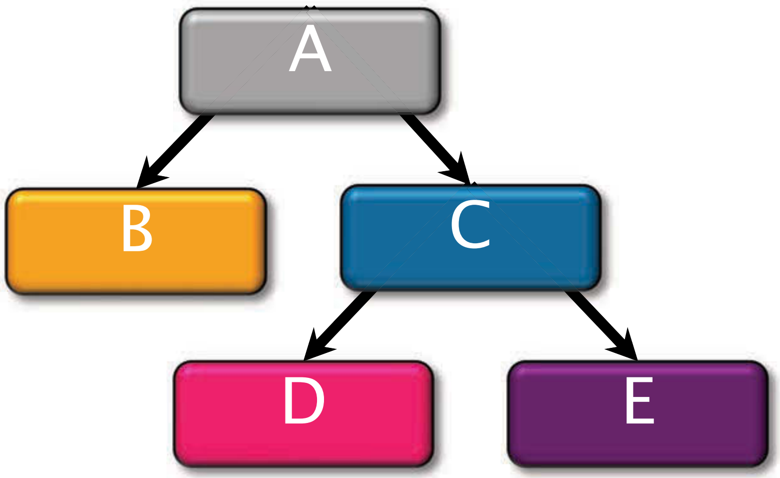
Required Functionality

- A single worker must be able to execute the computation on its own similarly to an ordinary serial computation.
- A thief must be able to jump into the middle of an executing function to steal a continuation.
- A sync must stall a function's execution until child subcomputations complete.

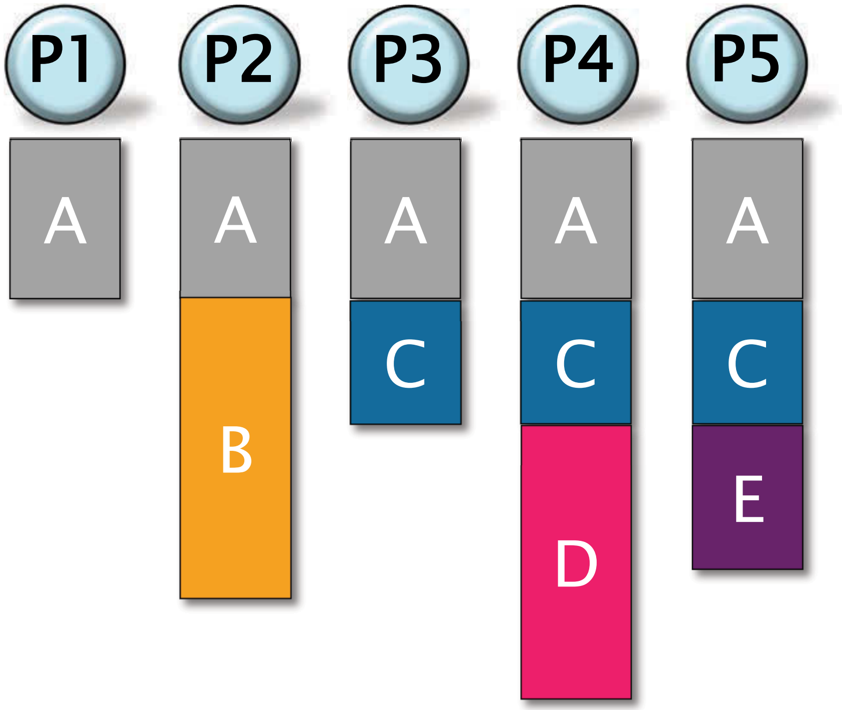
What other functionality is needed?

Recall: Cactus Stack

Cilk supports C’s rule for pointers: A pointer to stack space can be passed from parent to child, but not from child to parent.



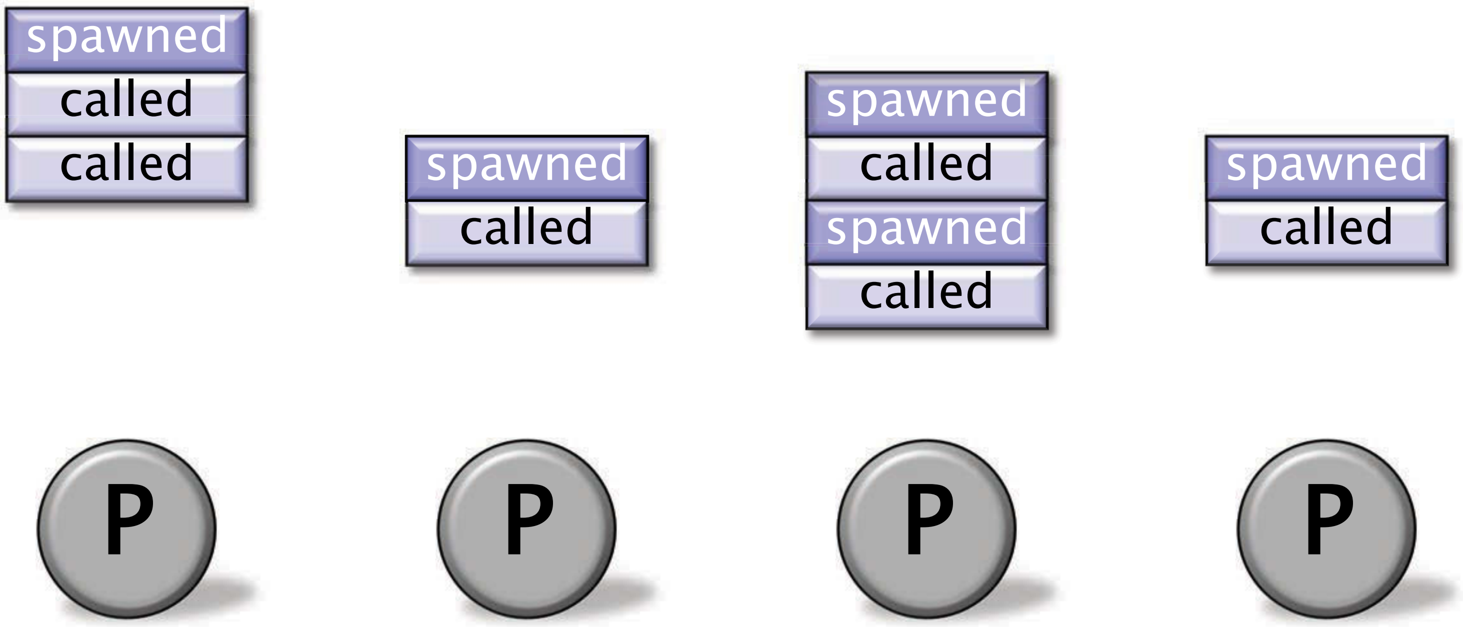
Views of stack



Cilk’s cactus stack supports multiple views in parallel.

Recall: Work Stealing

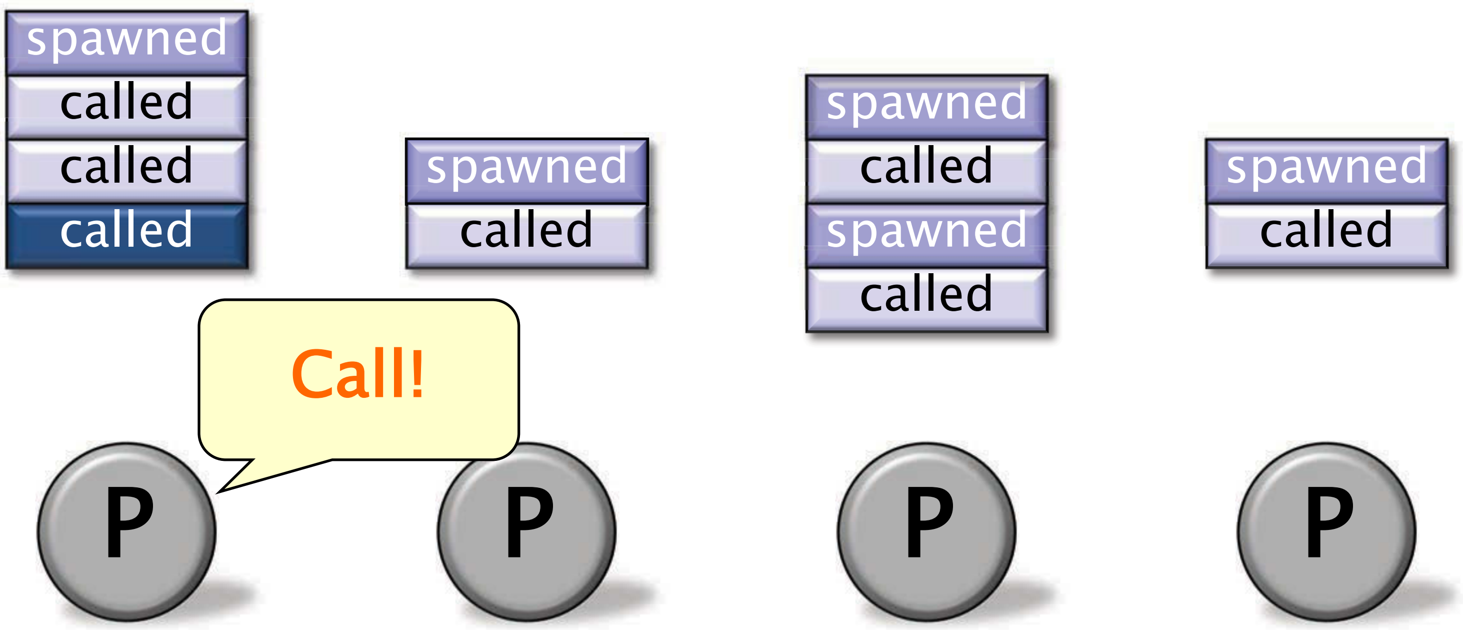
Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



Each deque contains a mixture of spawned frames and called frames.

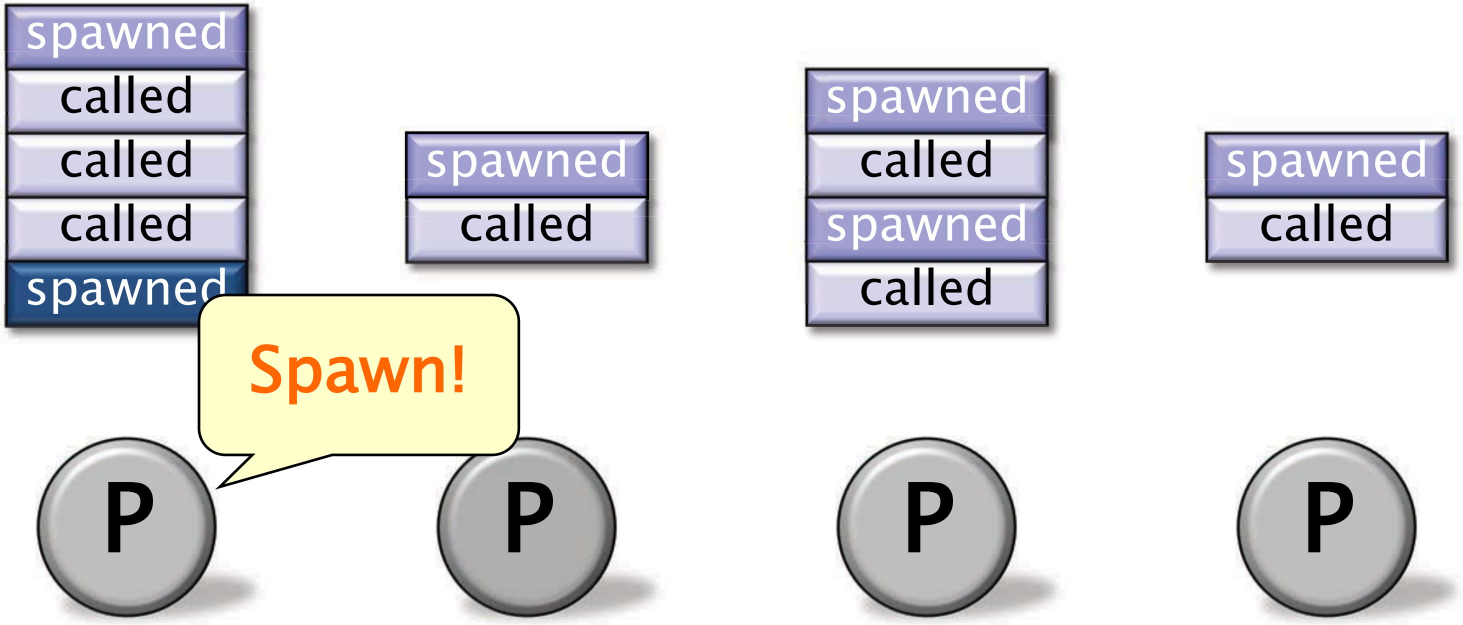
Recall: Work Stealing

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



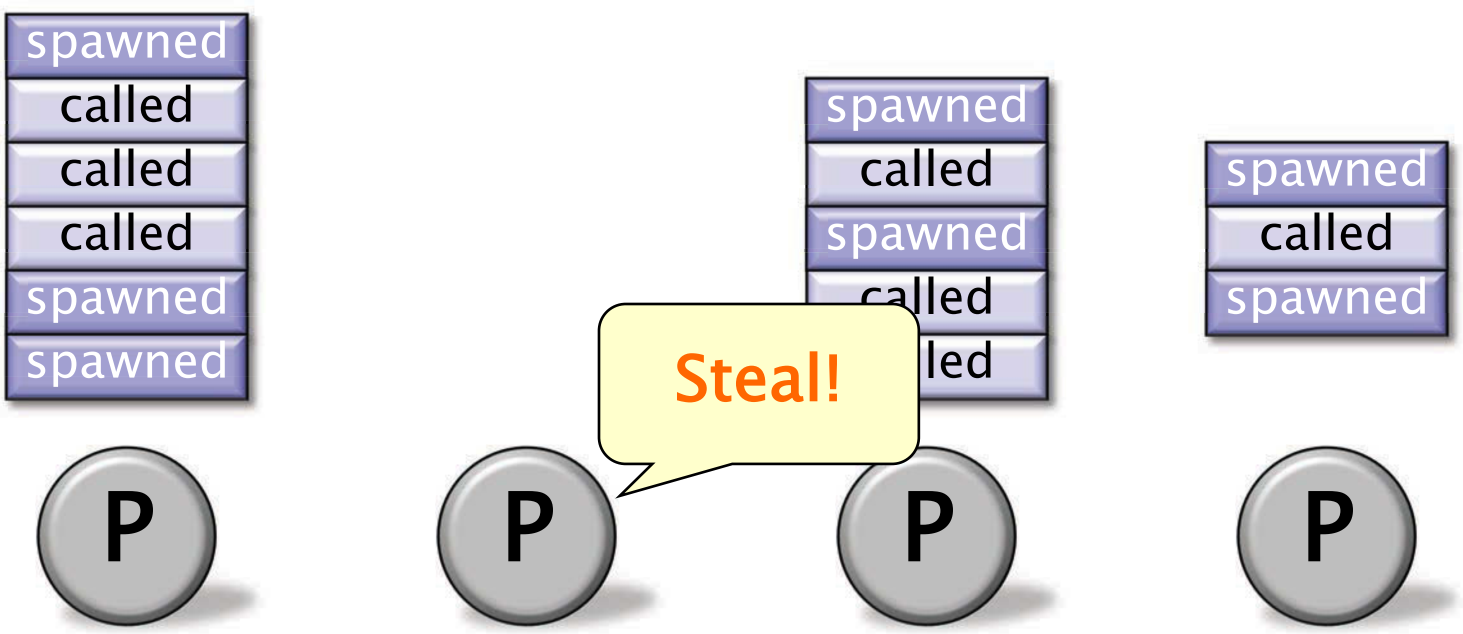
Recall: Work Stealing

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



Recall: Work Stealing

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].

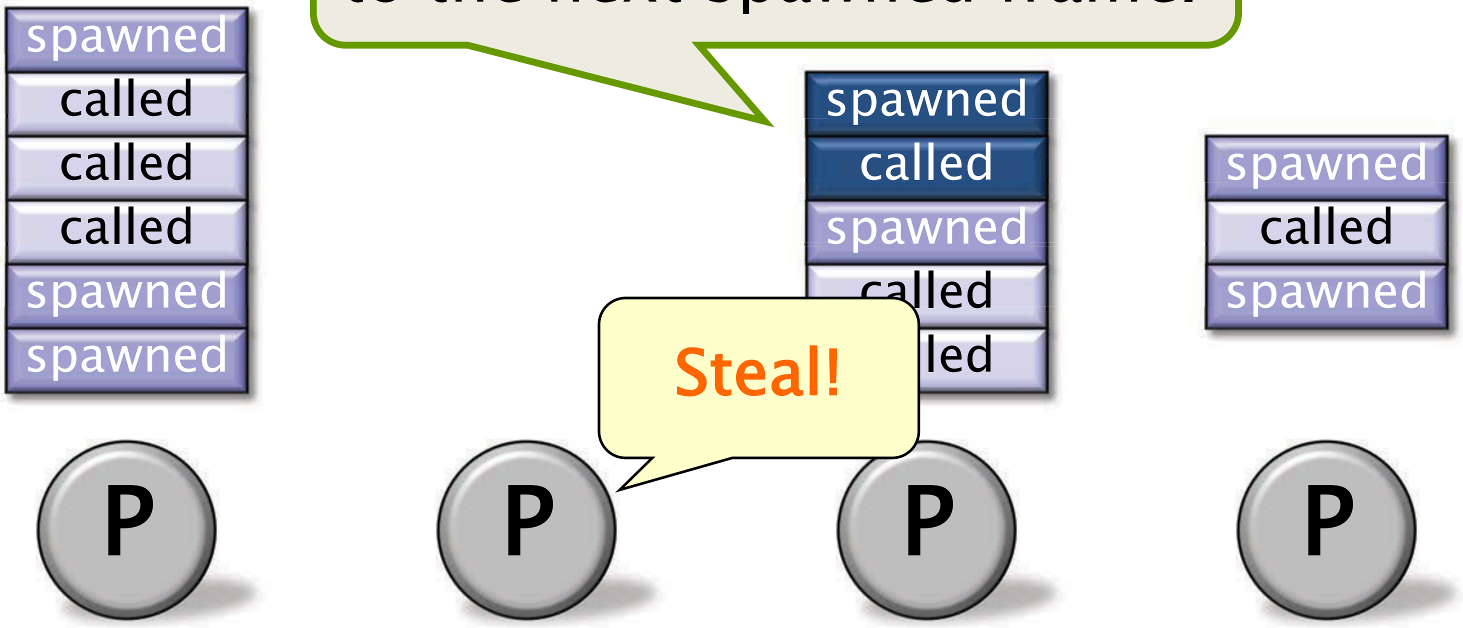


When a worker runs out of work, it steals from the top of a random victim's deque.

Recall: Work Stealing

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like .

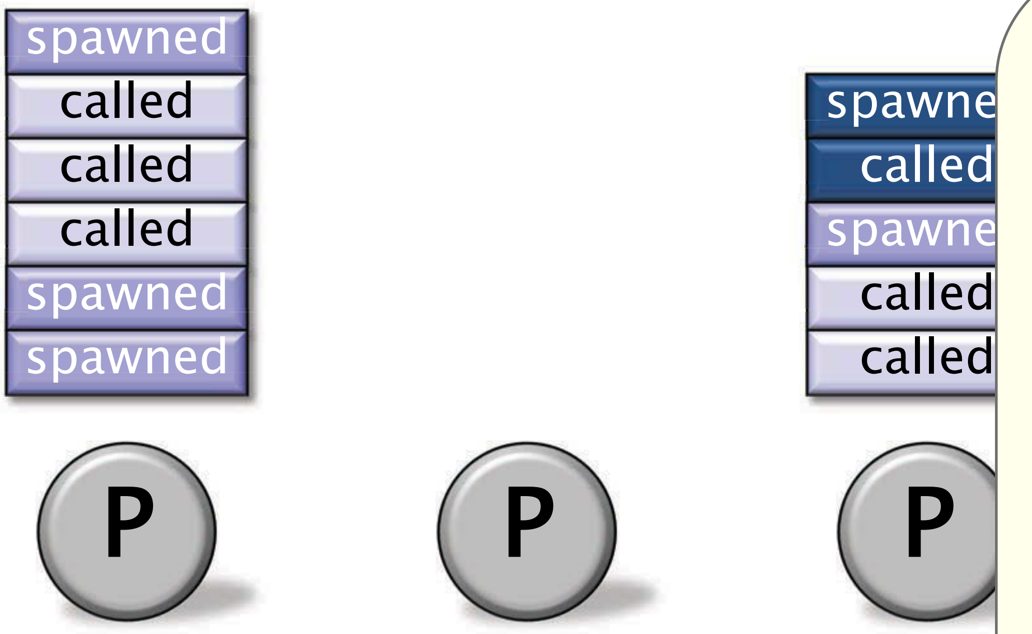
A steal takes all frames up to the next spawned frame.



When a worker runs out of work, it steals from the top of a random victim's deque.

Recall: Work Stealing

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



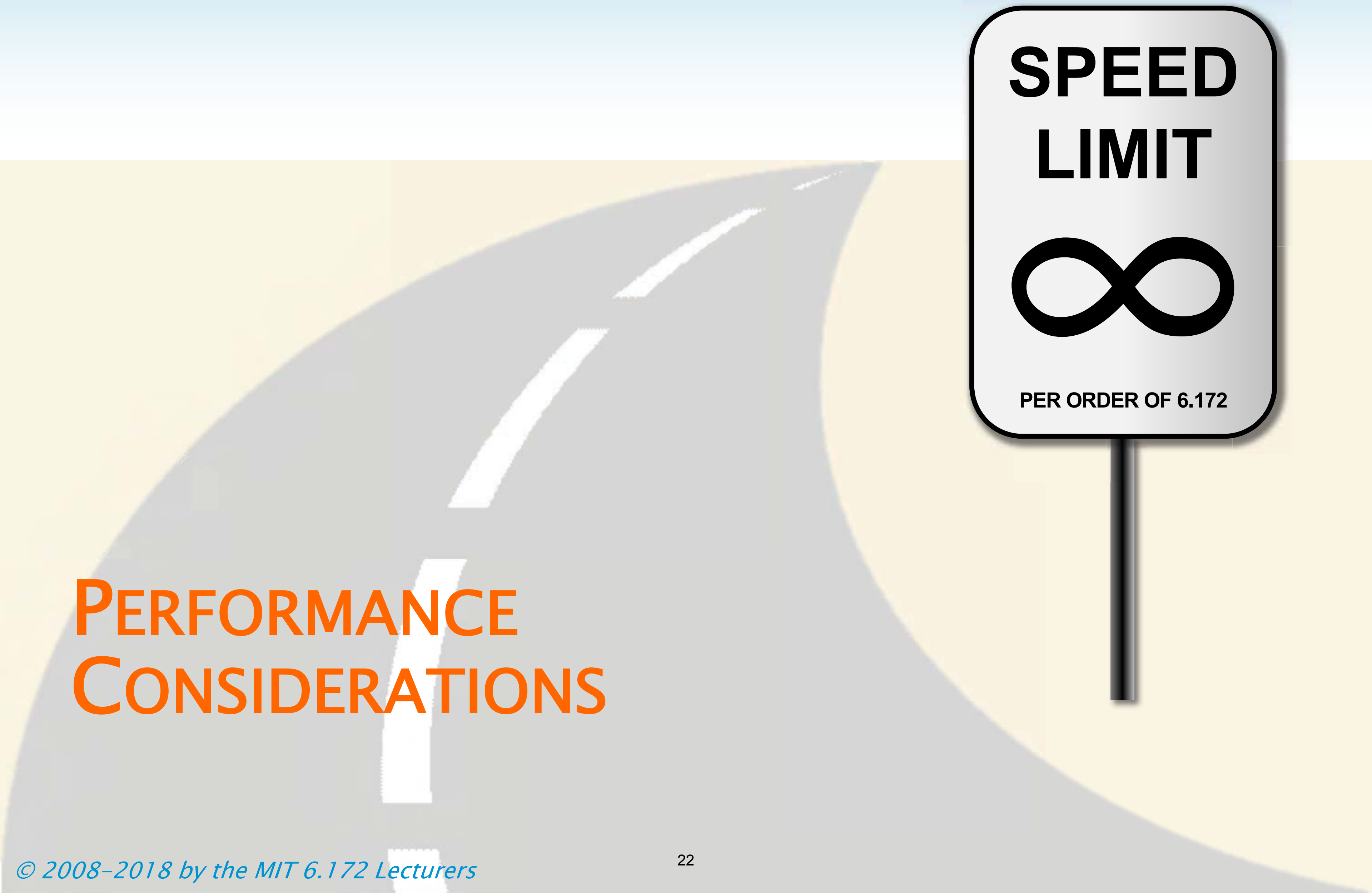
What is involved in stealing frames?

- What synchronization is needed?
- What happens to the stack?
- How efficient can this be?

When a worker runs out of work,
from the top of a random victim's deque.

Required Functionality

- A single worker must be able to execute the computation on its own similarly to an ordinary serial computation.
- A thief must be able to jump into the middle of an executing function to steal a continuation.
- A sync must stall a function's execution until child subcomputations complete.
- The runtime must implement a cactus stack for its parallel workers.
- Thieves must be able to handle mixtures of called spawned functions.



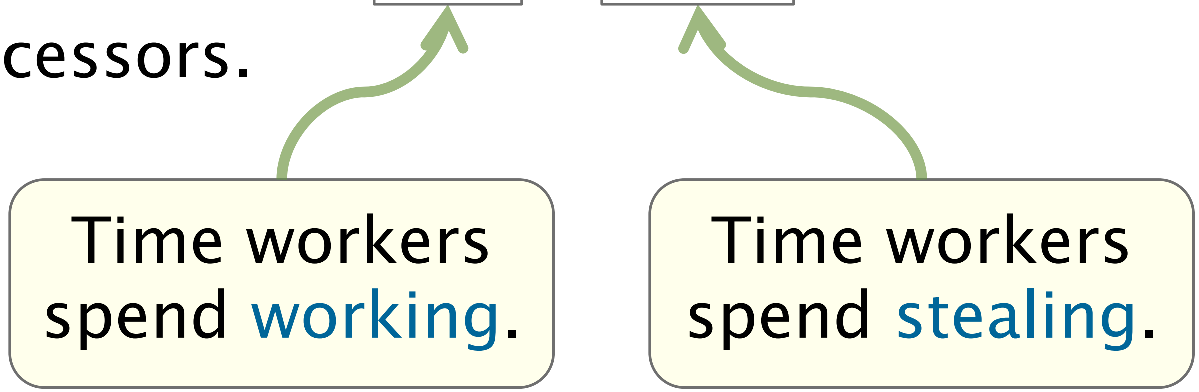
PERFORMANCE CONSIDERATIONS

Recall: Work-Stealing Bounds

Theorem [BL94]. The Cilk work-stealing scheduler achieves expected running time

$$T_P \approx \boxed{T_1/P} + \boxed{O(T_\infty)}$$

on P processors.



Time workers spend working.

Time workers spend stealing.

If the program achieves linear speedup, then workers spend most of their time working.

Parallel Speedup

Ideally, parallelizing a serial code makes it run P times faster on P processors.

Serial matrix multiply

```
for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Running time T_S.

Cilk matrix multiply

```
cilk_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

With sufficient parallelism, running time is $T_P \approx T_1/P$.

Goal: $T_P \approx T_S/P$, meaning that $T_S \approx T_1$.

Work Efficiency

Let T_S denote the work of a serial program. Suppose the serial program is parallelized. Let T_1 denote the work of the parallel program, and let T_∞ denote the span of the parallel program.

To achieve linear speedup on P processors over the serial program, i.e., $T_P \approx T_S/P$, the parallel program must exhibit:

- Ample parallelism: $T_1/T_\infty \gg P$.
- High work efficiency: $T_S/T_1 \approx 1$.

The Work-First Principle

To optimize the execution of programs with sufficient parallelism, the implementation of the Cilk runtime system works to maintain high work-efficiency by abiding by the work-first principle:

Optimize for the *ordinary serial execution*, at the expense of some additional computation in steals.

Division of Labor

The work-first principle guides the division of the Cilk runtime system between the compiler and the runtime library.

Compiler

- Uses a handful of small data structures, e.g., workers and stack frames.
- Implements optimized fast paths for execution of functions when no steals have occurred.

Runtime library

- Uses larger data structures.
- Handles slow paths of execution, e.g., when a steal occurs.

IMPLEMENTING A WORKER DEQUE

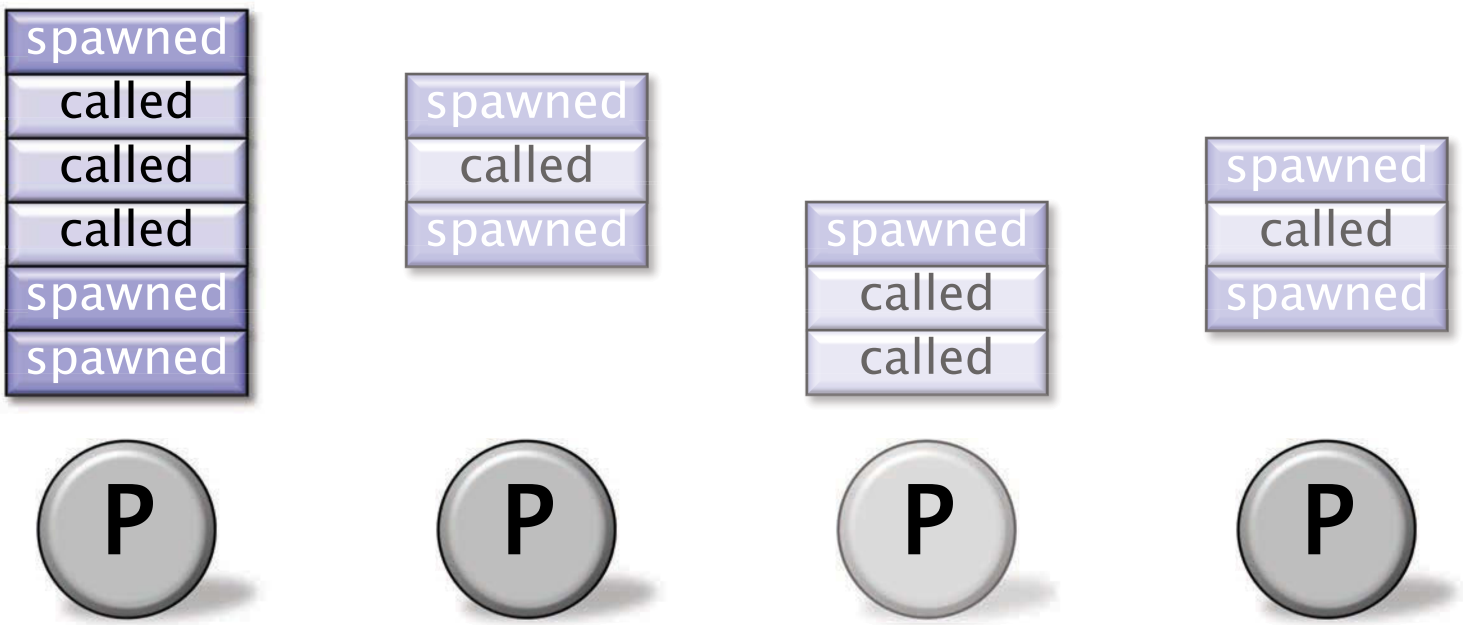
Running Example

```
int foo(int n) {
   int x, y;
   x = cilk_spawn bar(n);
   y = baz(n);
   cilk_sync;
   return x + y;
}
```

- Function foo is a spawning function, meaning that foo contains a `cilk_spawn`.
- Function bar is spawned by foo.
- The call to baz occurs in the continuation of the spawn.

Requirements of Worker Deques

PROBLEM: How do we implement a worker's deque?

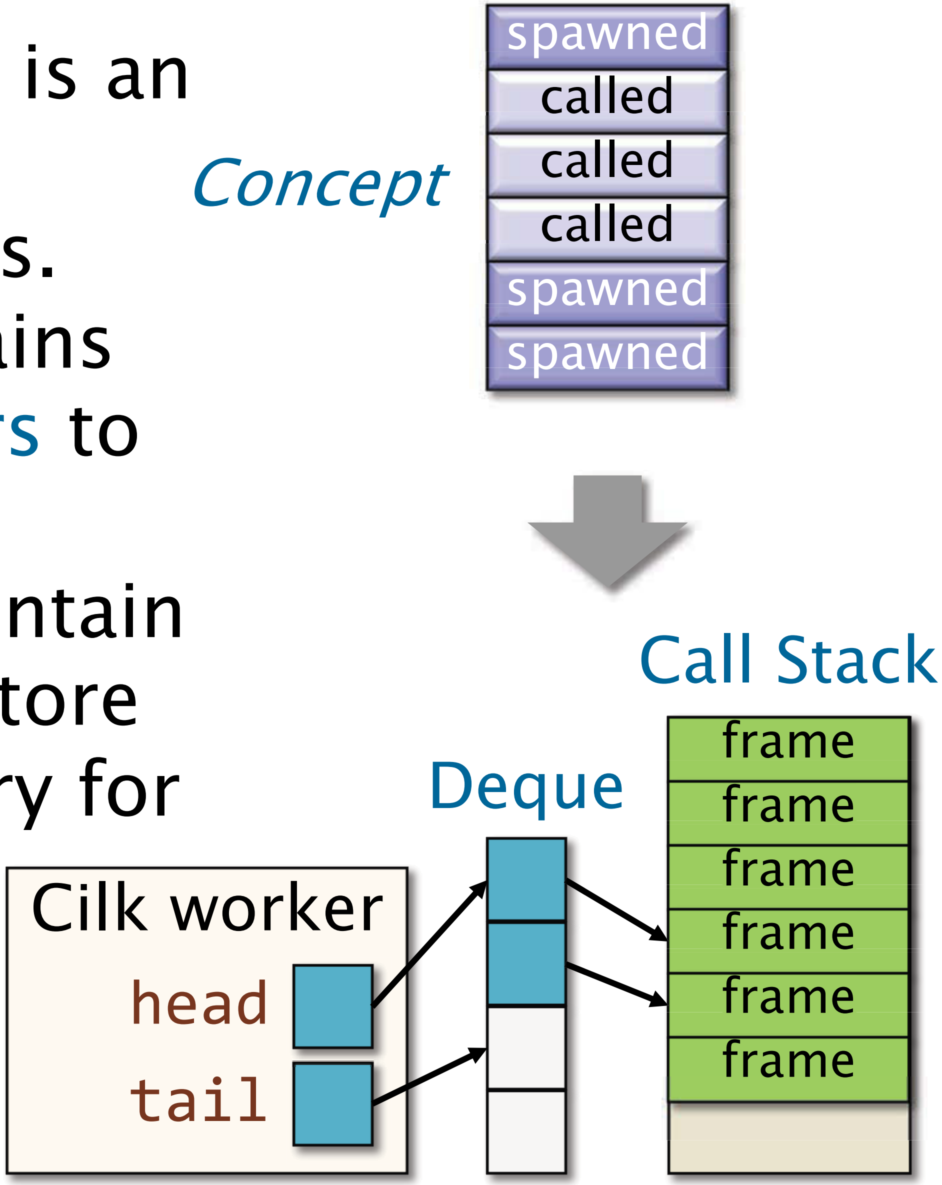


- The worker should operate its own deque like a stack.
- A steal needs to transfer ownership of several consecutive frames to a thief.
- A thief needs to be able to resume a continuation.

Basic Worker–Deque Design

IDEA: The worker deque is an external structure with pointers to stack frames.

- A Cilk worker maintains head and tail pointers to its deque.
- Stealable frames maintain a local structure to store information necessary for stealing the frame.

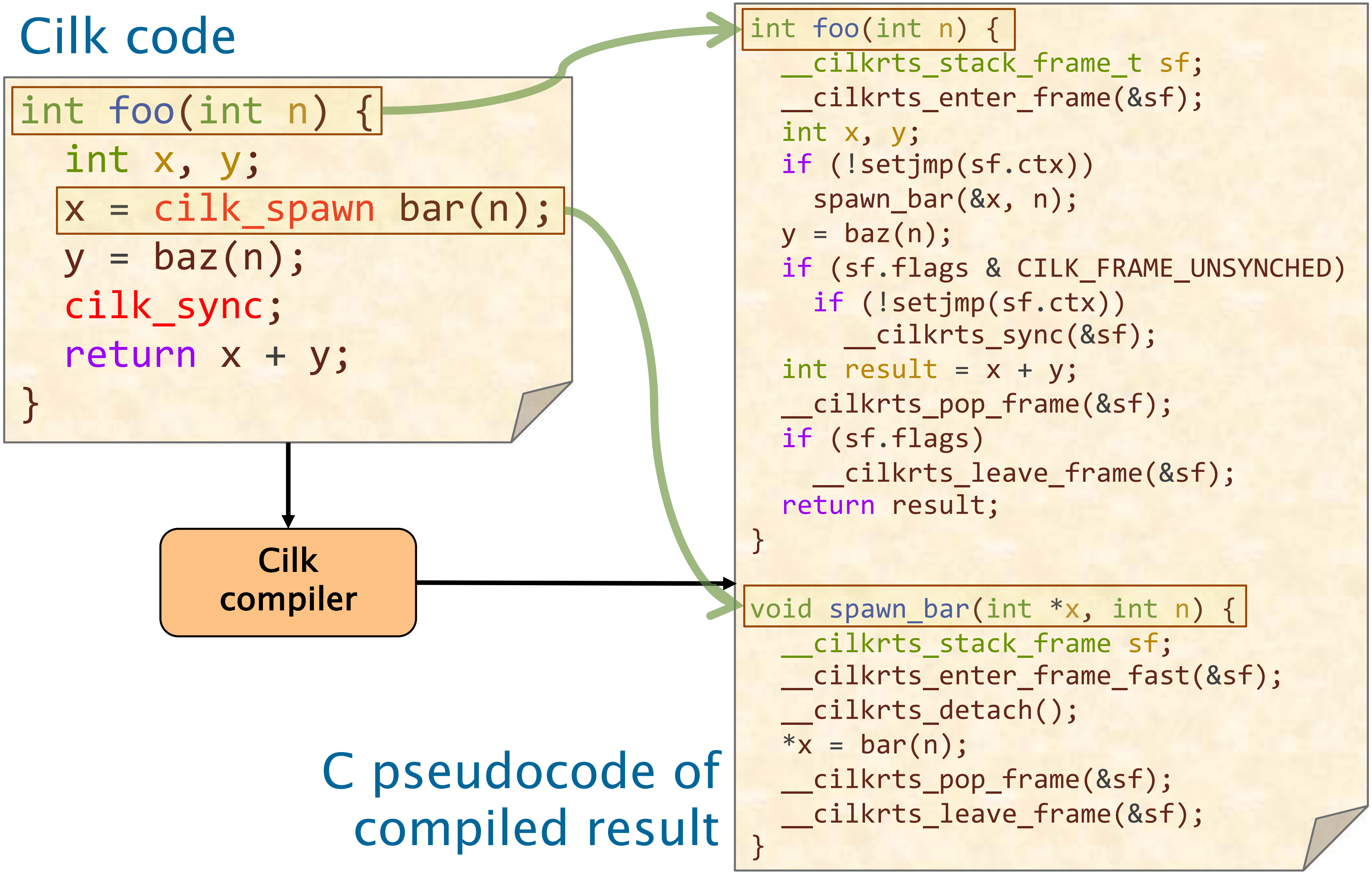


Implementation Details

The Intel Cilk Plus runtime elaborates on this idea as follows:

- Every spawned subcomputation runs in its own spawn-helper function.
- The runtime maintains three basic data structures as workers execute work:
  - A worker structure for every worker used to execute the program.
  - A Cilk stack-frame structure for each instantiation of a spawning function.
  - A spawn-helper stack frame for each instantiation of a `cilk_spawn`.

Spawn-Helper Functions



Cilk code

```
int foo(int n) {
  int x, y;
  x = cilk_spawn bar(n);
  y = baz(n);
  cilk_sync;
  return x + y;
}
```

Cilk compiler

C pseudocode of compiled result

```
int foo(int n) {
  __cilkrts_stack_frame_t sf;
  __cilkrts_enter_frame(&sf);
  int x, y;
  if (!setjmp(sf.ctx))
    spawn_bar(&x, n);
  y = baz(n);
  if (sf.flags & CILK_FRAME_UNSYNCHED)
    if (!setjmp(sf.ctx))
      __cilkrts_sync(&sf);
  int result = x + y;
  __cilkrts_pop_frame(&sf);
  if (sf.flags)
    __cilkrts_leave_frame(&sf);
  return result;
}

void spawn_bar(int *x, int n) {
  __cilkrts_stack_frame sf;
  __cilkrts_enter_frame_fast(&sf);
  __cilkrts_detach();
  *x = bar(n);
  __cilkrts_pop_frame(&sf);
  __cilkrts_leave_frame(&sf);
}
```

Cilk Stack-Frame Structures

```
int foo(int n) {
  __cilkrts_stack_frame_t sf;
  __cilkrts_enter_frame(&sf);
  int x, y;
  if (!setjmp(sf.ctx))
    spawn_bar(&x, n);
  y = baz(n);
  if (sf.flags & CILK_FRAME_UNSYNCHED)
    if (!setjmp(sf.ctx))
      __cilkrts_sync(&sf);
  int result = x + y;
  __cilkrts_pop_frame(&sf
  if (sf.flags)
    __cilkrts_leave_frame
  return result;
}
```

```
void spawn_bar(int *x, int n) {
  __cilkrts_stack_frame sf;
  __cilkrts_enter_frame_fast(&sf);
  __cilkrts_detach();
  *x = bar(n);
  __cilkrts_pop_frame(&sf);
  __cilkrts_leave_frame(&sf);
}
```

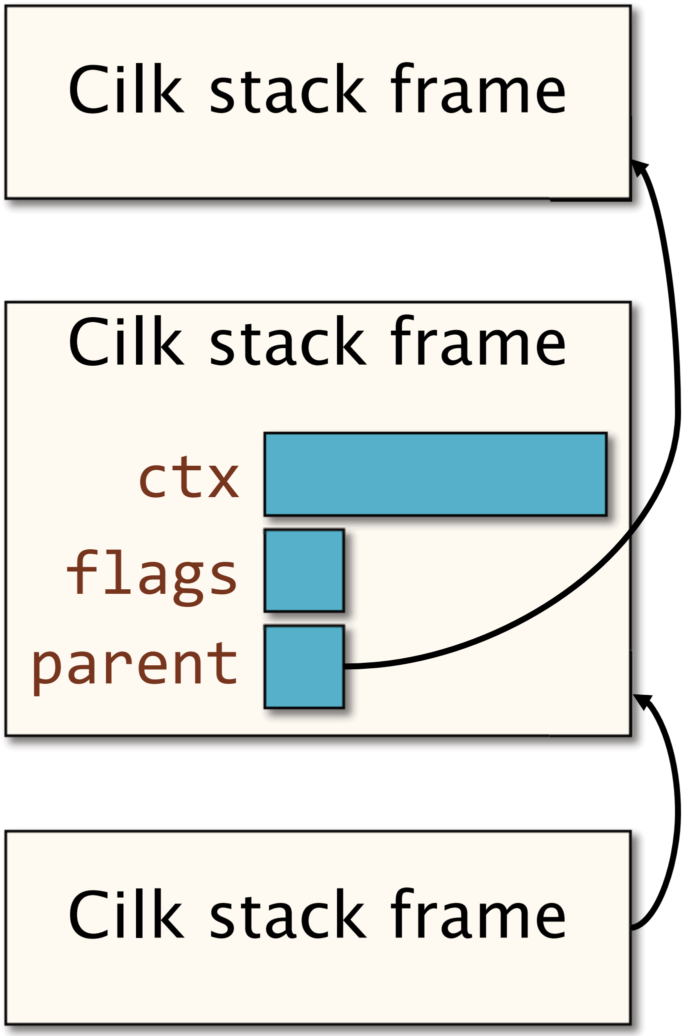
C pseudocode of compiled result

Cilk stack-frame structures

The Cilk Stack Frame (Simplified)

Each Cilk stack frame stores:

- A context buffer, `ctx`, which contains enough information to resume a function at a continuation, i.e., after a `cilk_spawn` or `cilk_sync`.
- An integer, `flags`, that summarizes the state of the Cilk stack frame.
- A pointer, `parent`, to its parent Cilk stack frame.

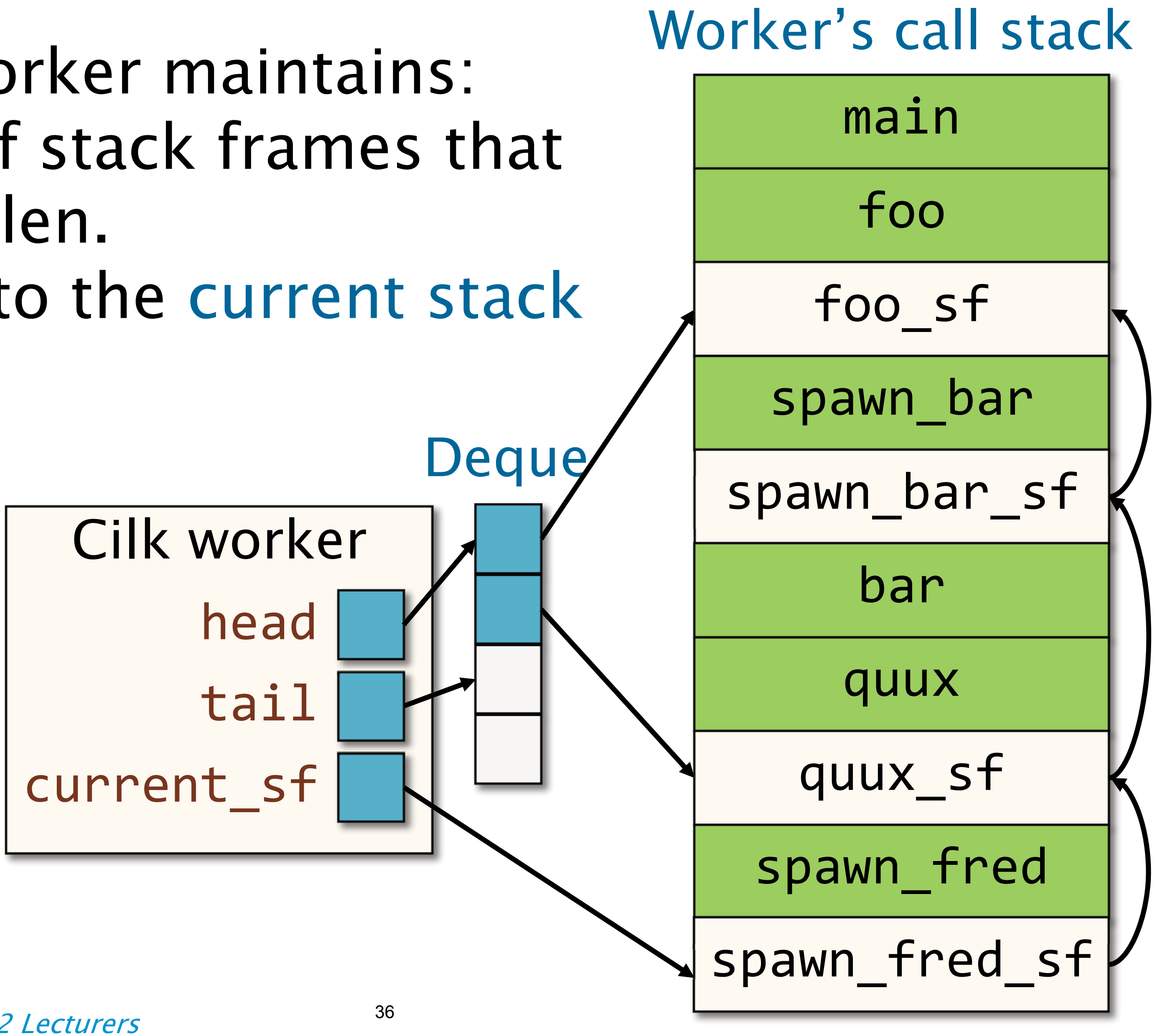


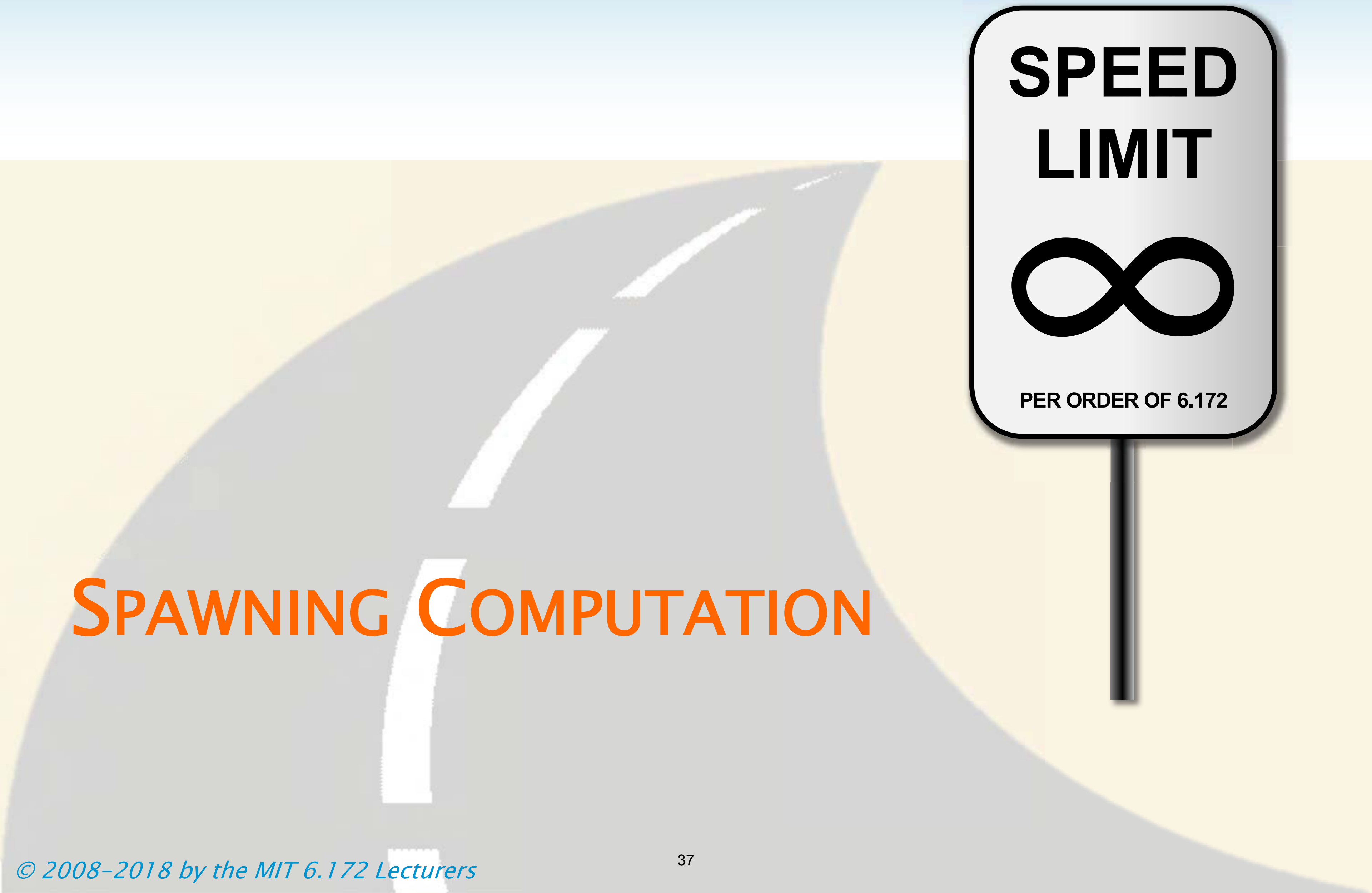
The Cilk Worker Structure (Simplified)

Each Cilk worker maintains:

- A deque of stack frames that can be stolen.
- A pointer to the current stack frame.

Example:
Function `foo` spawned `bar`, which called `quux`, which spawned `fred`.





SPAWNING COMPUTATION

Code for a Spawning Function

C pseudocode of a spawning function

```
int foo(int n) {
  __cilkrts_stack_frame_t sf;
  __cilkrts_enter_frame(&sf);
  int x, y;
  if (!setjmp(sf.ctx))
    spawn_bar(&x, n);
  y = baz(n);
  if (sf.flags & CILK_FRAME_UNSYNCHED)
    if (!setjmp(sf.ctx))
      __cilkrts_sync(&sf);
  int result = x + y;
  __cilkrts_pop_frame(&sf);
  if (sf.flags)
    __cilkrts_leave_frame(&sf);
  return result;
}
```

Create and initialize a Cilk stack-frame structure.

Prepare to spawn.

Invoke the spawn helper.

Perform a sync.

Clean up the Cilk stack-frame structure.

Clean up the deque.

Code for a Spawn Helper

C pseudocode of a spawn helper

```
void spawn_bar(int *x, int n) {
  __cilkrts_stack_frame sf;
  __cilkrts_enter_frame_fast(&sf);


  __cilkrts_detach();


  *x = bar(n);


  __cilkrts_pop_frame(&sf);
  __cilkrts_leave_frame(&sf);
}
```

Create and initialize a Cilk stack-frame structure.

Upate the deque to allow the parent to be stolen.

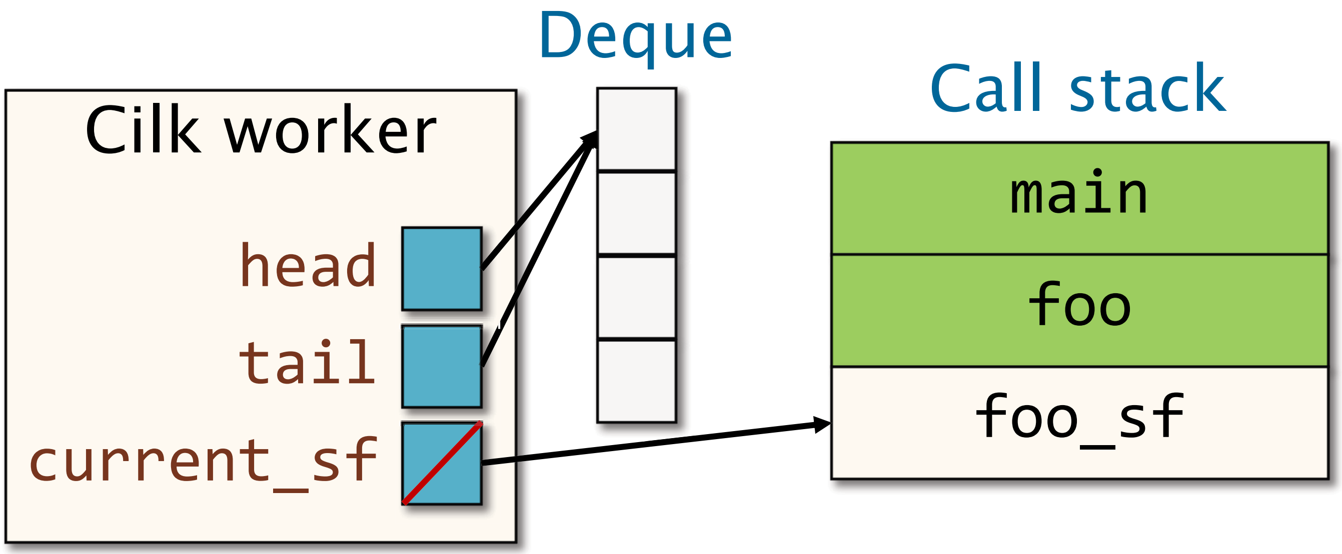
Invoke the spawned subroutine.

Clean up the Cilk stack-frame structure.

Clean up the deque and attempt to return.

Entering a Spawning Function

When execution enters a spawning function, the Cilk worker's current stack-frame structure is updated.



Preparing to Spawn

Cilk code

```
int foo(int n) {
  …
  x = cilk_spawn bar(n);
  …
}
```

C pseudocode

```
int foo(int n) {
  …
  if (!setjmp(sf.ctx))
    spawn_bar(&x, n);
  …
}
```

Cilk uses the `setjmp` function to allow thieves to steal the continuation.

The `setjmp` function stores information necessary for resuming the function at the `setjmp` into the given buffer.

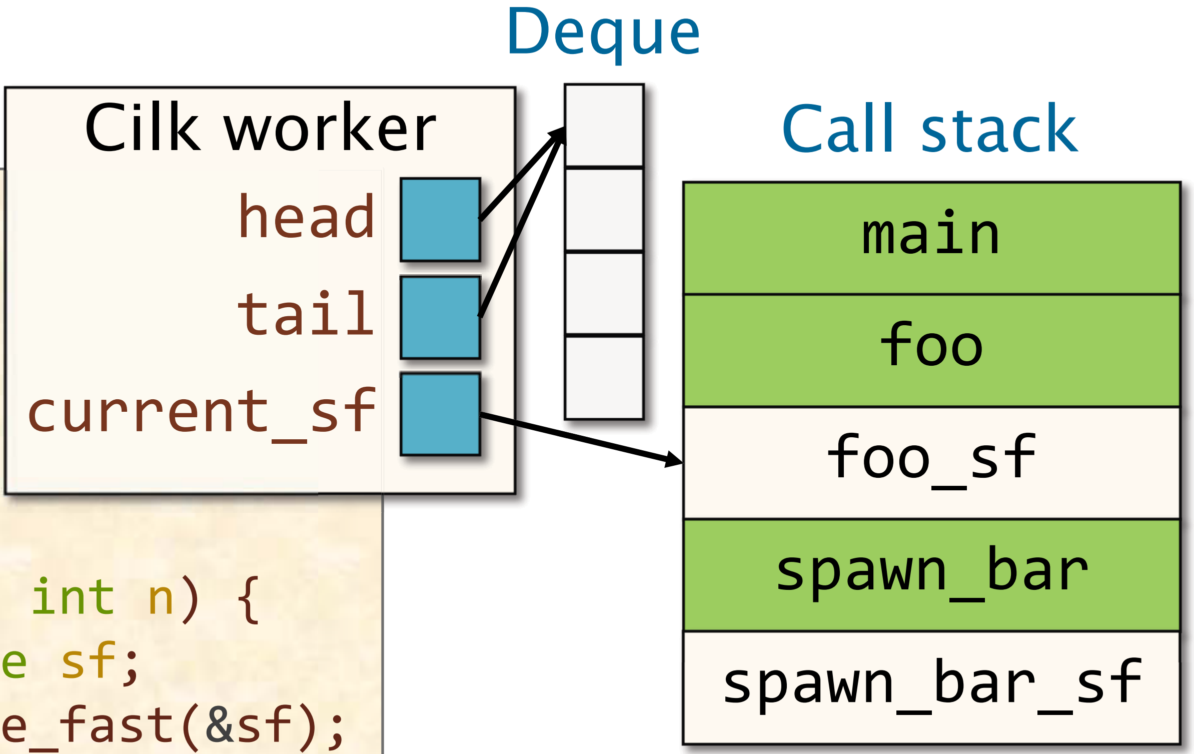
QUESTION: What information needs to be saved?

ANSWER: Registers `%rip`, `%rbp`, `%rsp`, and callee-saved registers.

Spawning a Function

C pseudocode

```
int foo(int n) {
  …
  if (!setjmp(sf.ctx))
    spawn_bar(&x, n);
  …
}
void spawn_bar(int *x, int n) {
  __cilkrts_stack_frame sf;
  __cilkrts_enter_frame_fast(&sf);
  __cilkrts_detach();
  *x = bar(n);
  …
}
```



Deque

Cilk worker

head

tail

current_sf

Call stack

main

foo

foo_sf

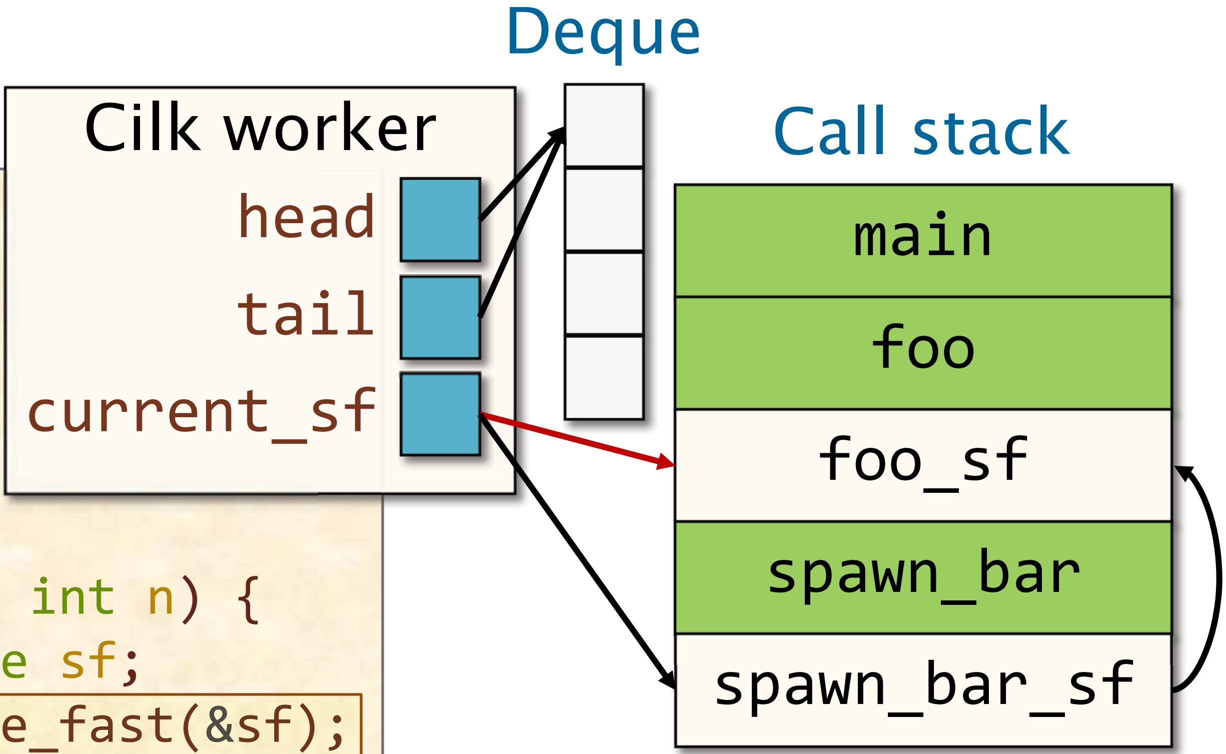
spawn_bar

spawn_bar_sf

Spawning a Function

C pseudocode

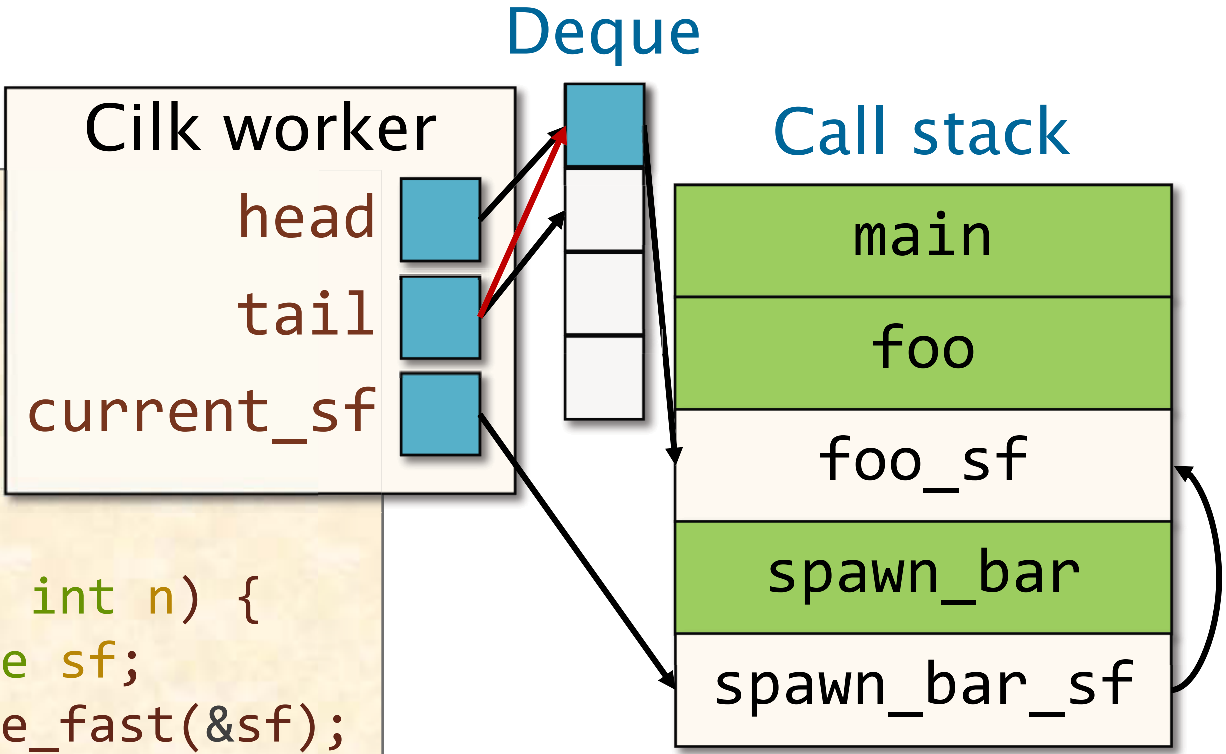
```
int foo(int n) {
  …
  if (!setjmp(sf.ctx))
    spawn_bar(&x, n);
  …
}
void spawn_bar(int *x, int n) {
  __cilkrts_stack_frame sf;
  __cilkrts_enter_frame_fast(&sf);
  __cilkrts_detach();
  *x = bar(n);
  …
}
```



Spawning a Function

C pseudocode

```
int foo(int n) {
   …
   if (!setjmp(sf.ctx))
     spawn_bar(&x, n);
   …
}
void spawn_bar(int *x, int n) {
   __cilkrts_stack_frame sf;
   __cilkrts_enter_frame_fast(&sf);
   __cilkrts_detach();
   *x = bar(n);
   …
}
```



Deque

Cilk worker

head

tail

current_sf

Call stack

main

foo

foo_sf

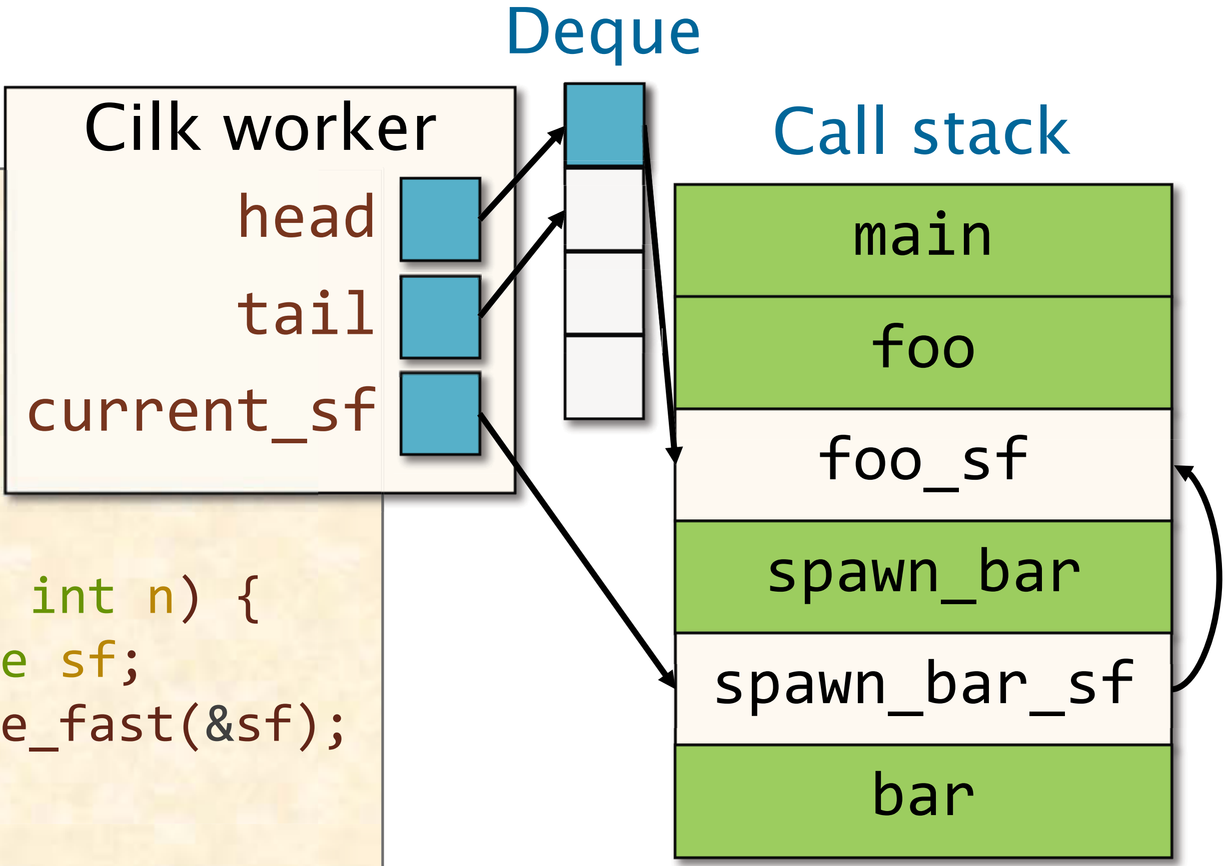
spawn_bar

spawn_bar_sf

Spawning a Function

C pseudocode

```
int foo(int n) {
  …
  if (!setjmp(sf.ctx))
    spawn_bar(&x, n);
  …
}
void spawn_bar(int *x, int n) {
  __cilkrts_stack_frame sf;
  __cilkrts_enter_frame_fast(&sf);
  __cilkrts_detach();
  *x = bar(n);
  …
}
```



Deque

Cilk worker

head

tail

current_sf

Call stack

main

foo

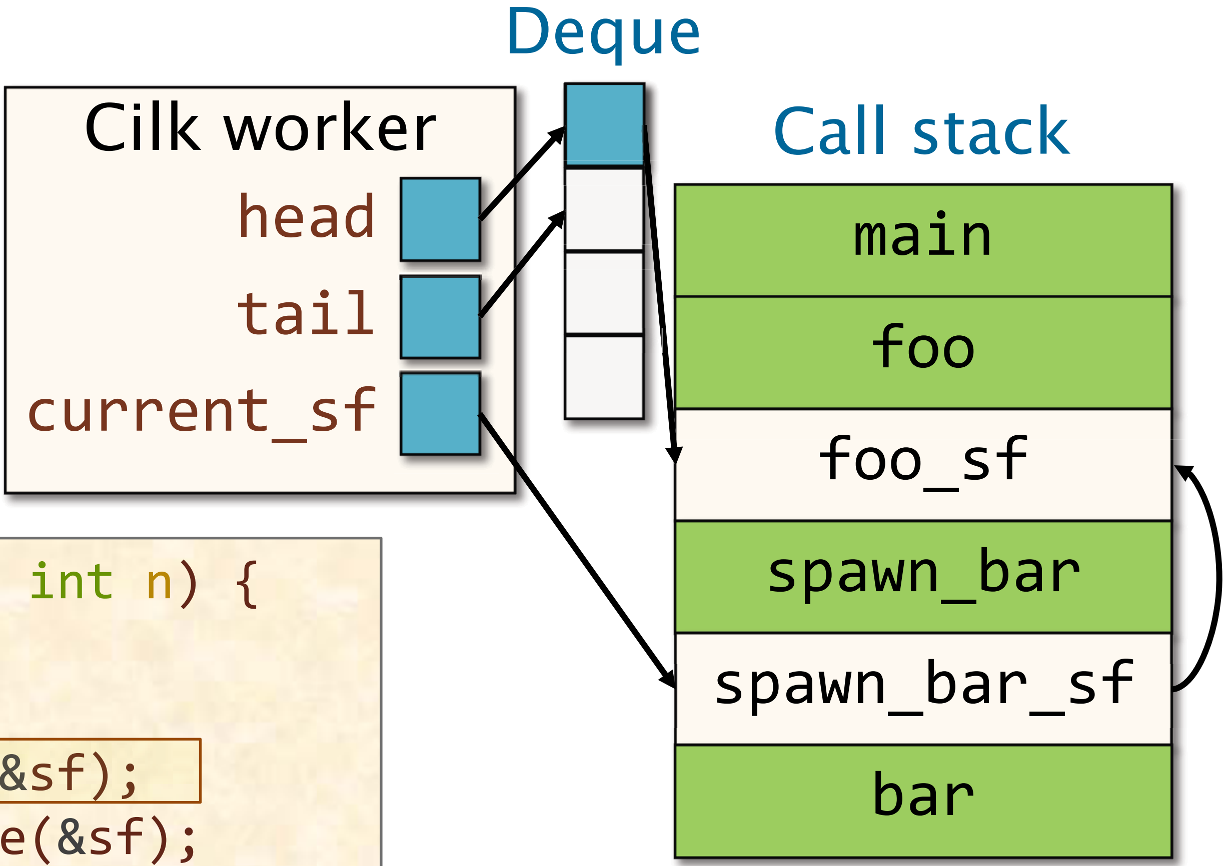
foo_sf

spawn_bar

spawn_bar_sf

bar

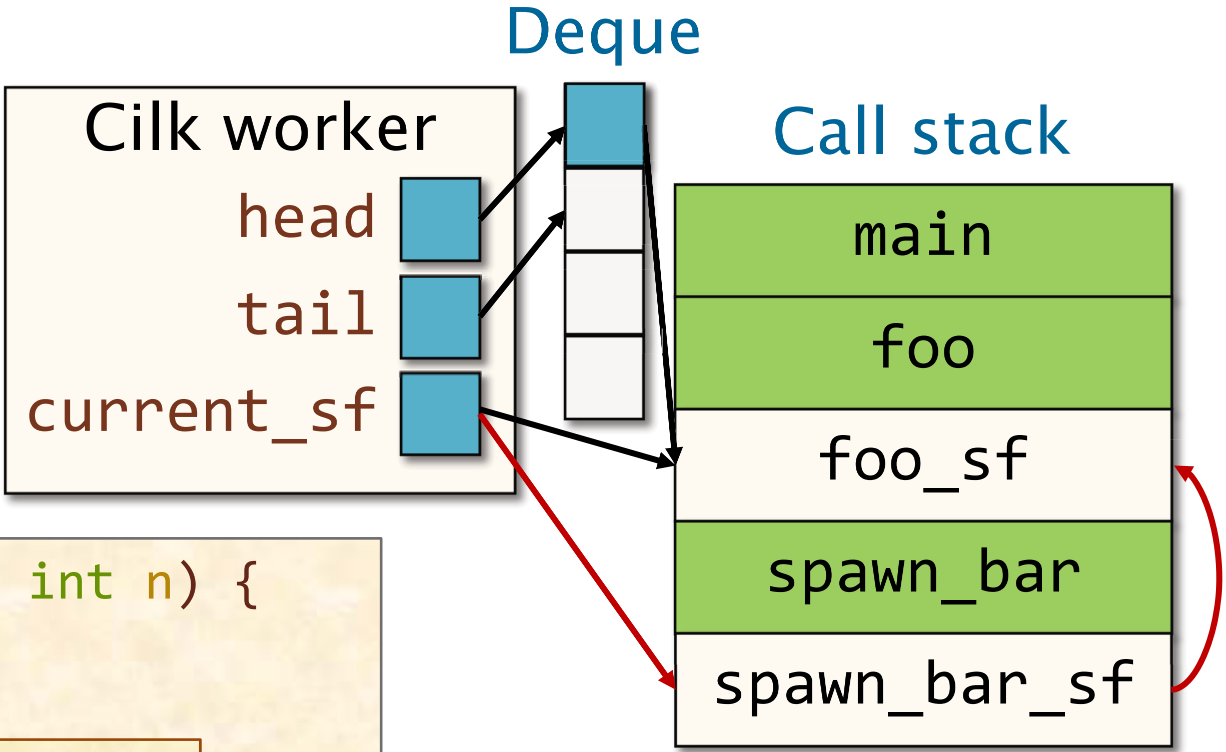
Returning from a Spawn



C pseudocode

```
void spawn_bar(int *x, int n) {
  …
  *x = bar(n);
  __cilkrts_pop_frame(&sf);
  __cilkrts_leave_frame(&sf);
}
```

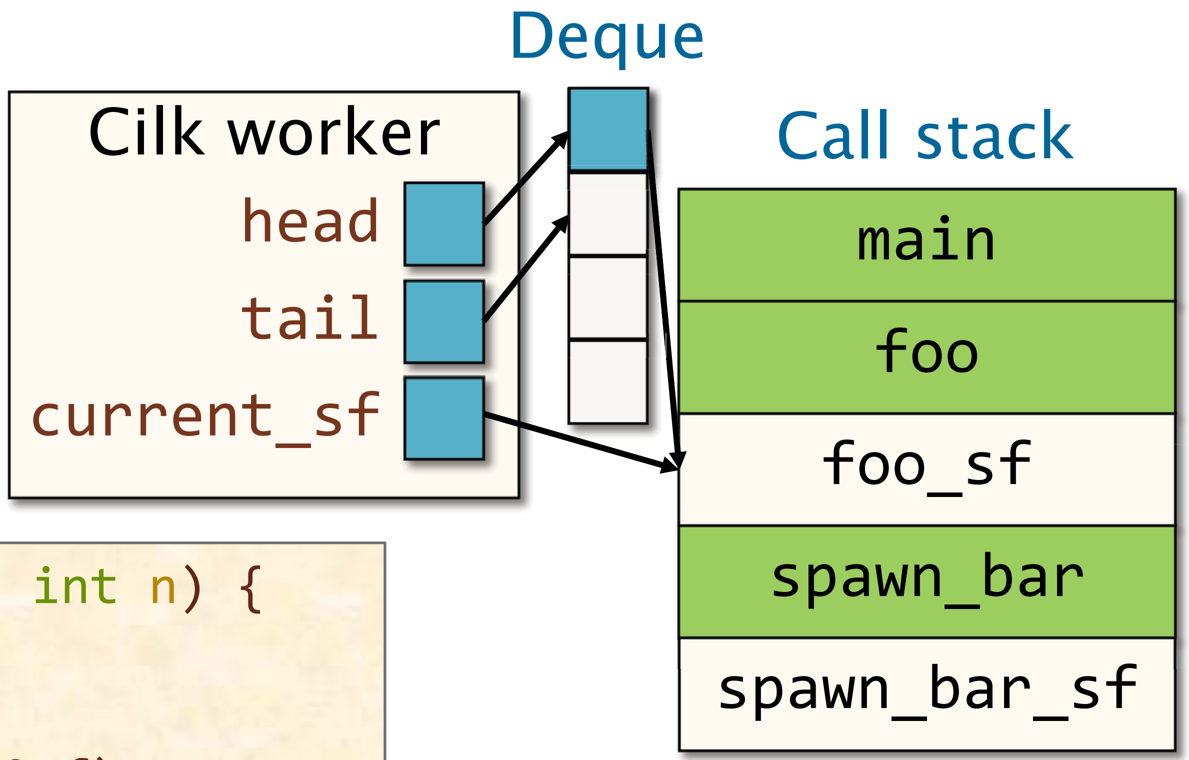
Returning from a Spawn



C pseudocode

```
void spawn_bar(int *x, int n) {
  …
  *x = bar(n);
  __cilkrts_pop_frame(&sf);
  __cilkrts_leave_frame(&sf);
}
```

Returning from a Spawn



Deque

Cilk worker

head

tail

current_sf

Call stack

main

foo

foo_sf

spawn_bar

spawn_bar_sf

C pseudocode

```
void spawn_bar(int *x, int n) {
  …
  *x = bar(n);
  __cilkrts_pop_frame(&sf);
  __cilkrts_leave_frame(&sf);
}
```

May or may not return, depending on what's in the worker's deque.

Popping the Deque

In `__cilkrts_leave_frame`, the worker tries to pop the stack frame from the tail of the deque. There are two possible outcomes:

1. If the pop succeeds, then the execution continues as normal.
2. If the pop fails, then the worker is out of work to do. It thus becomes a thief and tries to steal work from the top of a random victim's deque.

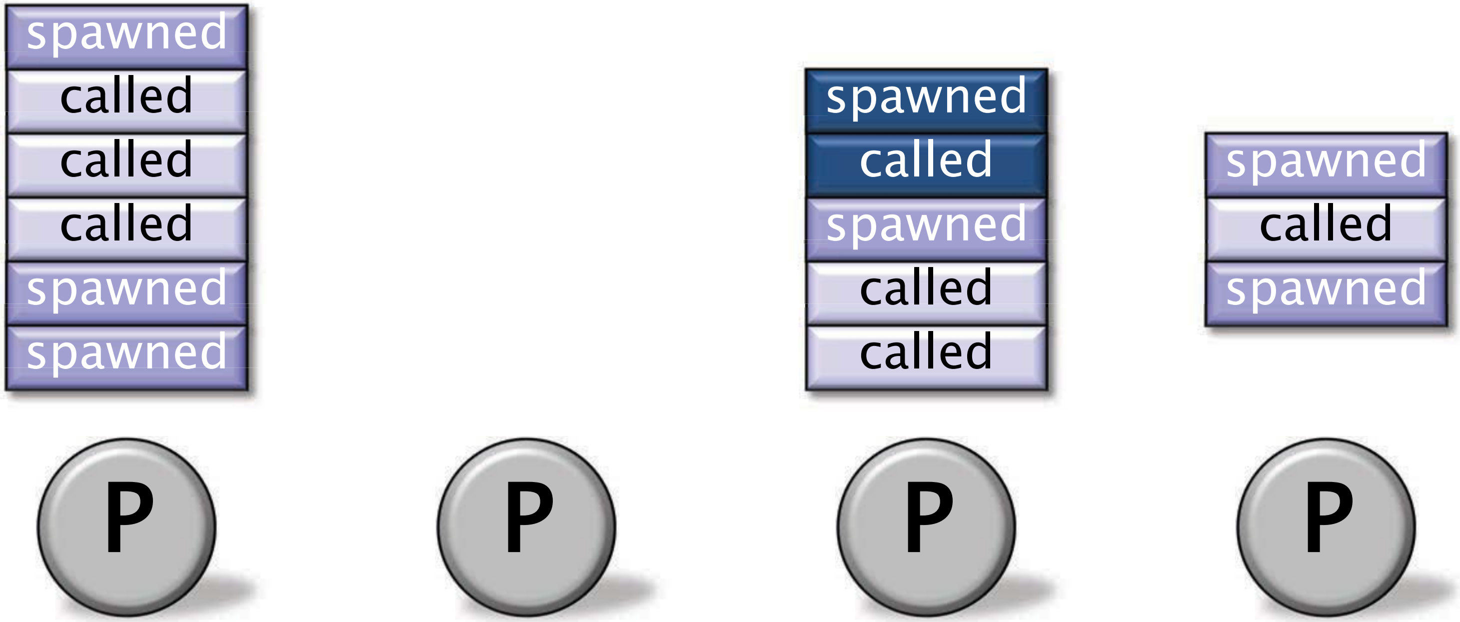
Question: Which case is more important to optimize?

Answer: Case 1.

STEALING COMPUTATION

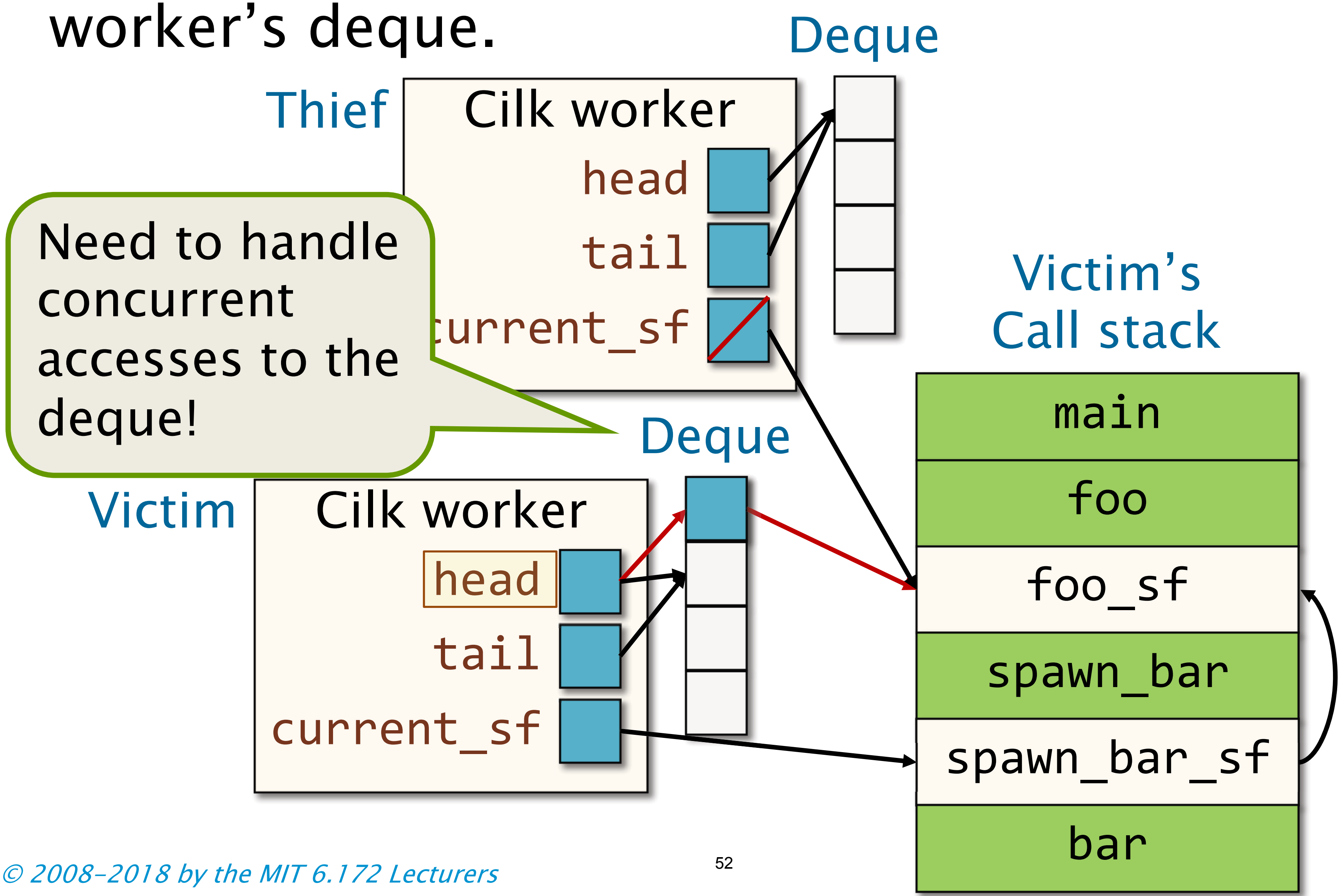
Recall: Stealing Work

Conceptually, a thief takes frames off of the top of a victim worker's deque.



Stealing a Frame

A thief steals from the head of the victim worker's deque.



Need to handle concurrent accesses to the deque!

Synchronizing Deque Accesses

Worker protocol

```
void push(
   tail++;
}
bool pop()
   tail--;
   if (head > tail) {
      tail++;
      lock(L);
      tail--;
      if (head > tail) {
         tail++;
         unlock(L);
         return FAILURE;
      }
      unlock(L);
```

The worker pops the

optimistically.

e worker and thief ordinate operations the deque using e THE protocol:

Thief proto

```
bool steal(
   lock(L);
   head++;
   if (head
      head--;
      unlock(L);
      return FAILURE;
   }
   unlock(L);
   return SUCCESS;
}
```

The thief always grabs a lock before operating on the deque.

The worker only grabs a lock if the deque appears to be empty.

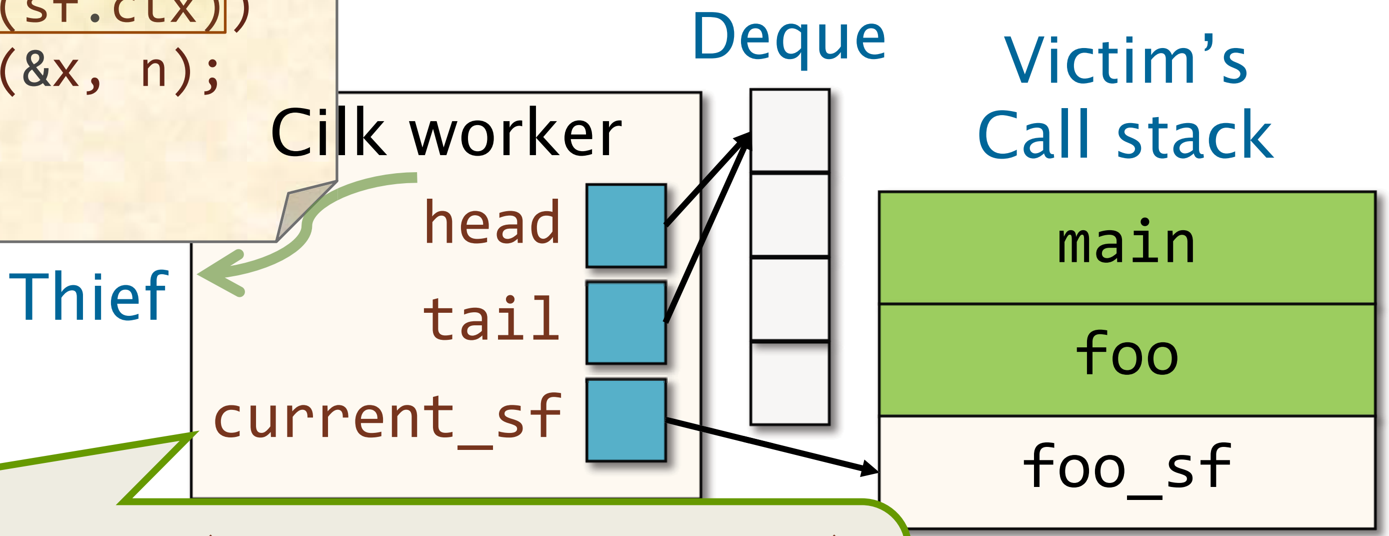
Resuming a Continuation

Cilk uses the `longjmp` function to resume a stolen continuation.

C pseudocode

```
int foo(int n) {
  …
  if (!setjmp(sf.ctx))
    spawn_bar(&x, n);
  …
}
```

Previously, the victim performed a `setjmp` to store register state in `foo_sf.ctx`.



Executing `longjmp(current_sf->ctx,1)` sets the thief's registers to start executing at the location of the `setjmp`.

Resuming a Continuation

The contract between `setjmp` and `longjmp` ensures the thief resumes the continuation.

- On its direct invocation, `setjmp(buffer)` returns `0`.
- Invoking `longjmp(buffer, x)` causes the `setjmp` to effectively return again with the integer value `x`.

C pseudocode

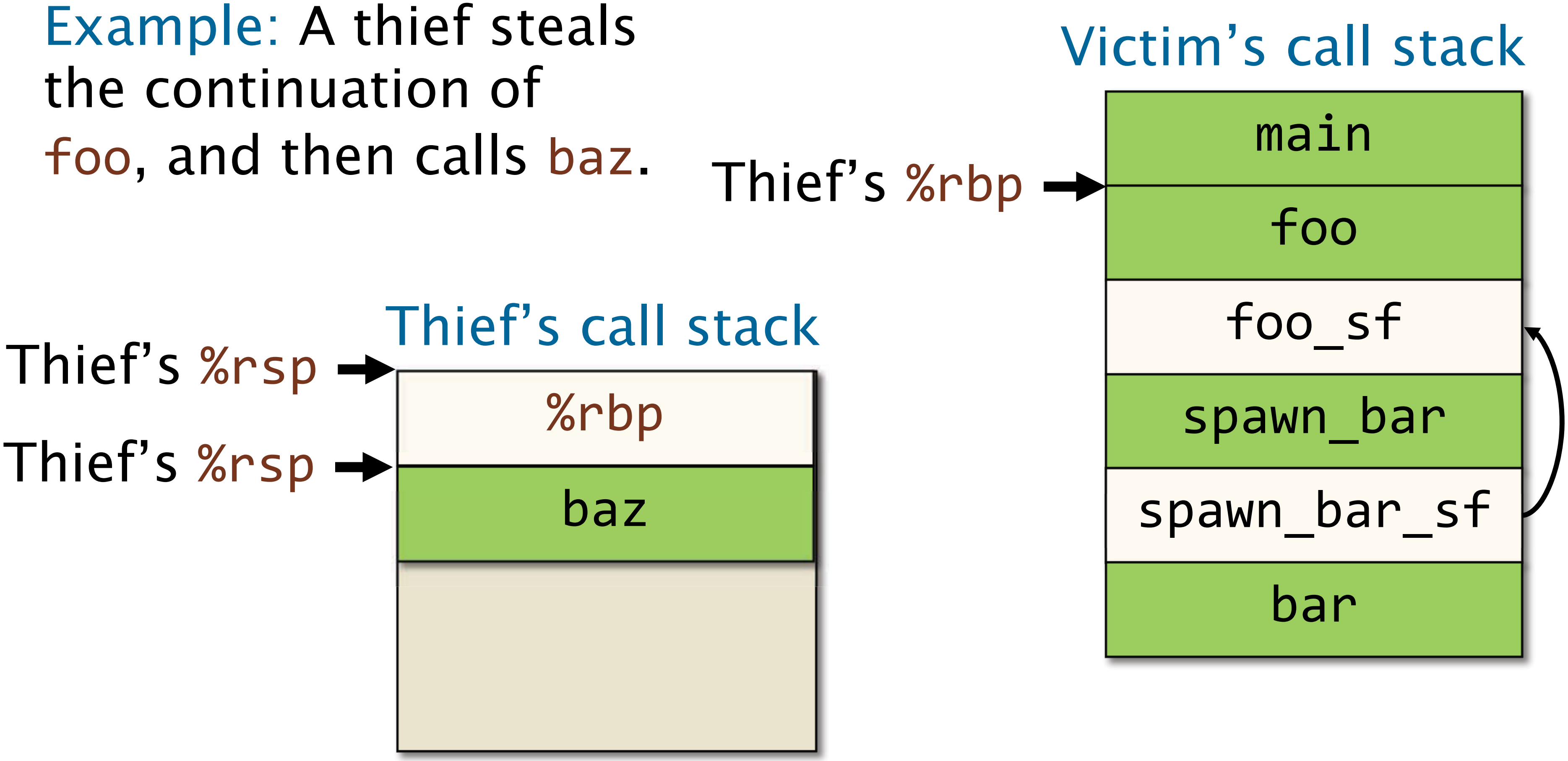
```
int foo(int n) {
  …
  if (!setjmp(sf.ctx))
    spawn_bar(&x, n);
  …
}
```

Because a thief reaches this point by calling `longjmp(current_sf->ctx,1)`, the condition fails, and the thief jumps to the continuation.

Implementing the Cactus Stack

Thieves maintain their own call stacks and use pointer tricks to implement the cactus stack.

Example: A thief steals the continuation of `foo`, and then calls `baz`.



SYNCHRONIZING COMPUTATION

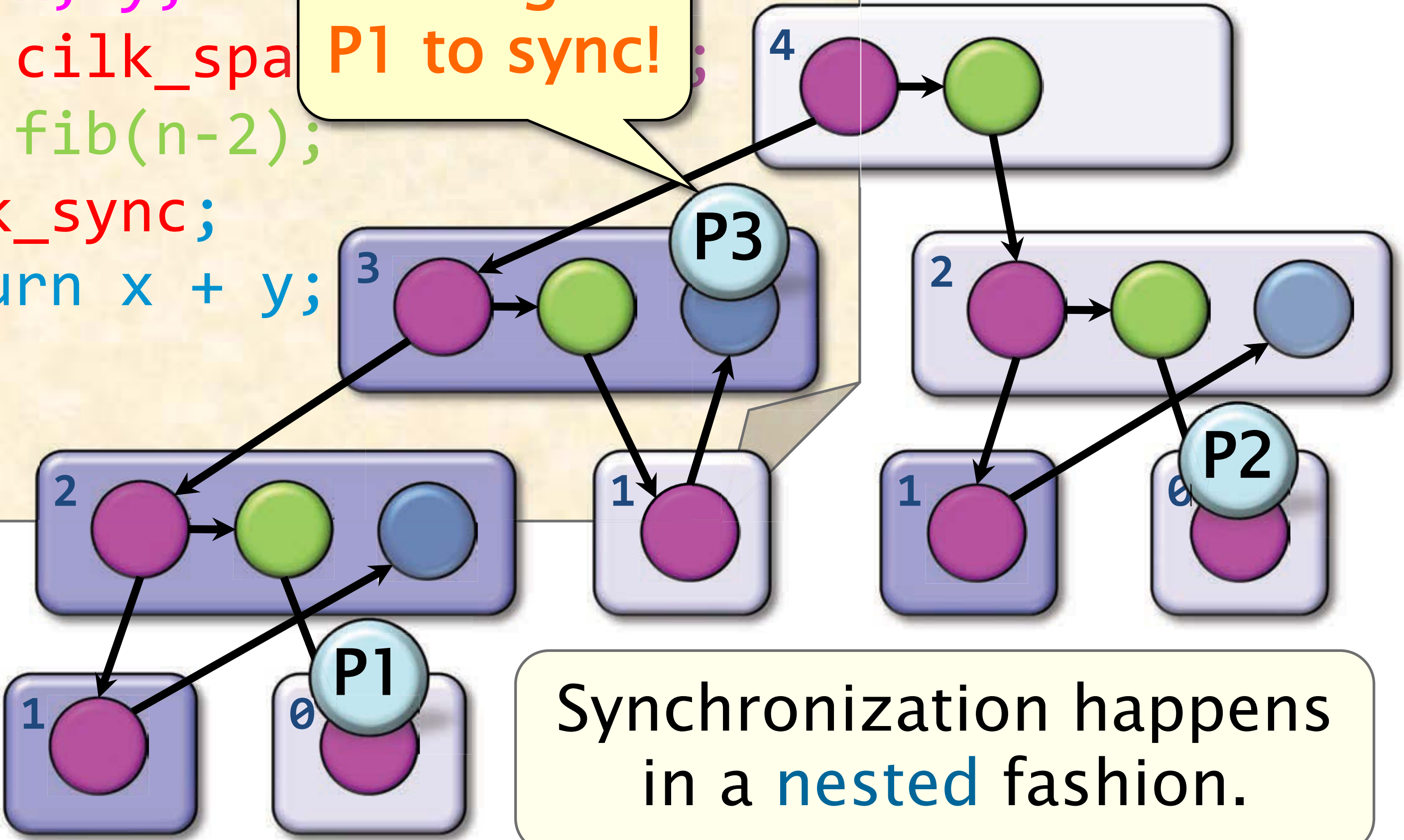
Recall: Nested Synchronization

```
int fib (int n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = cilk_spa                 ;
    y = fib(n-2);
    cilk_sync;
    return x + y;
  }
}
```

Example:

`fib(4)`

Waiting on P1 to sync!



Synchronization happens in a **nested** fashion.

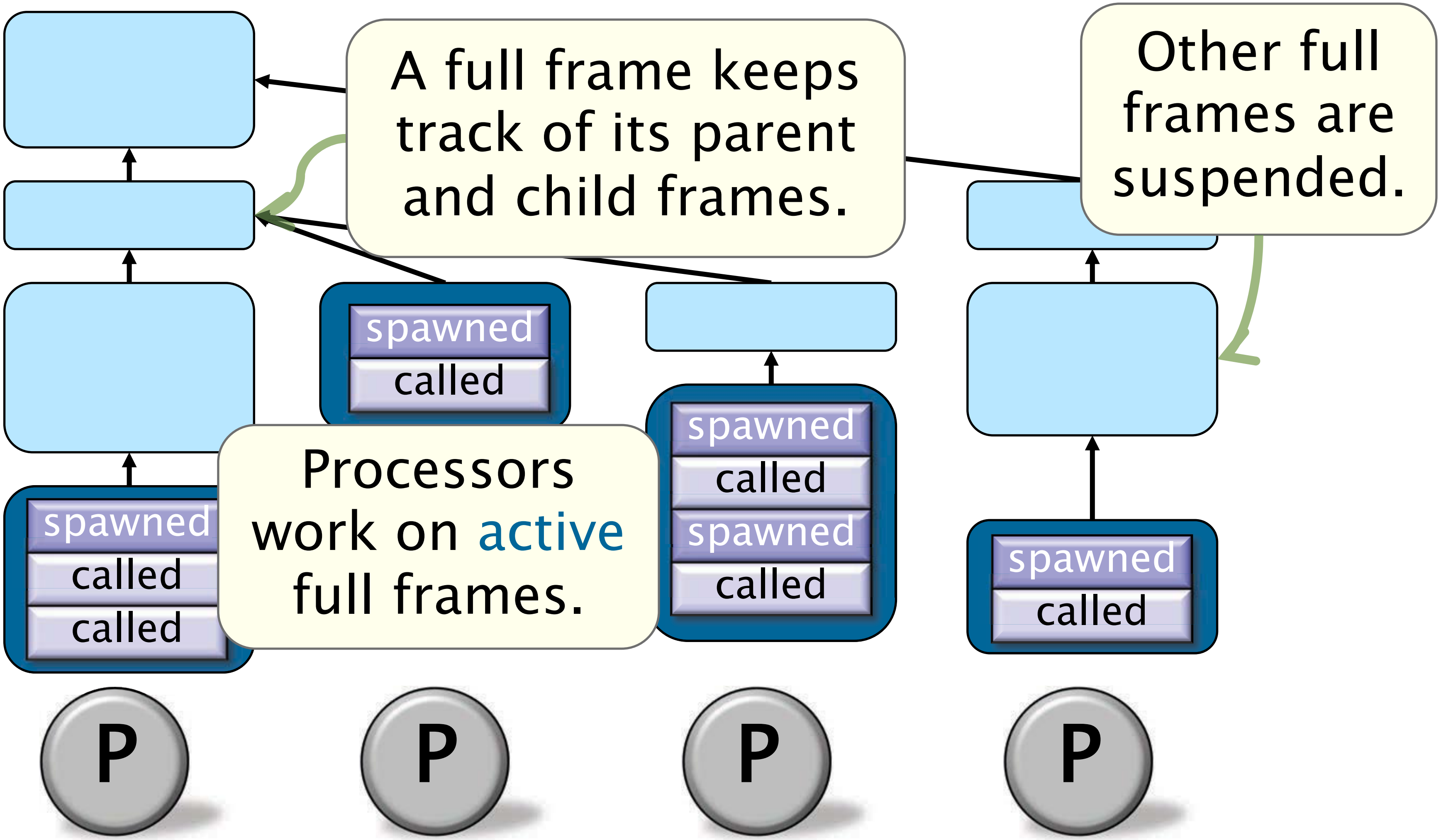
Synchronization Concerns

If a worker reaches a `cilk_sync` before all spawned subcomputations are complete, the worker should become a thief, but the worker's current function frame should not disappear!

- The existing subcomputations might access state in that frame, which is their parent frame.
- In the future, another worker must resume that frame and execute the `cilk_sync`.
- The `cilk_sync` only applies to nested subcomputations of the frame, not to all subcomputations or workers.

Full-Frame Tree

The Cilk runtime maintains a tree of full frames, which stores state for parallel subcomputations.



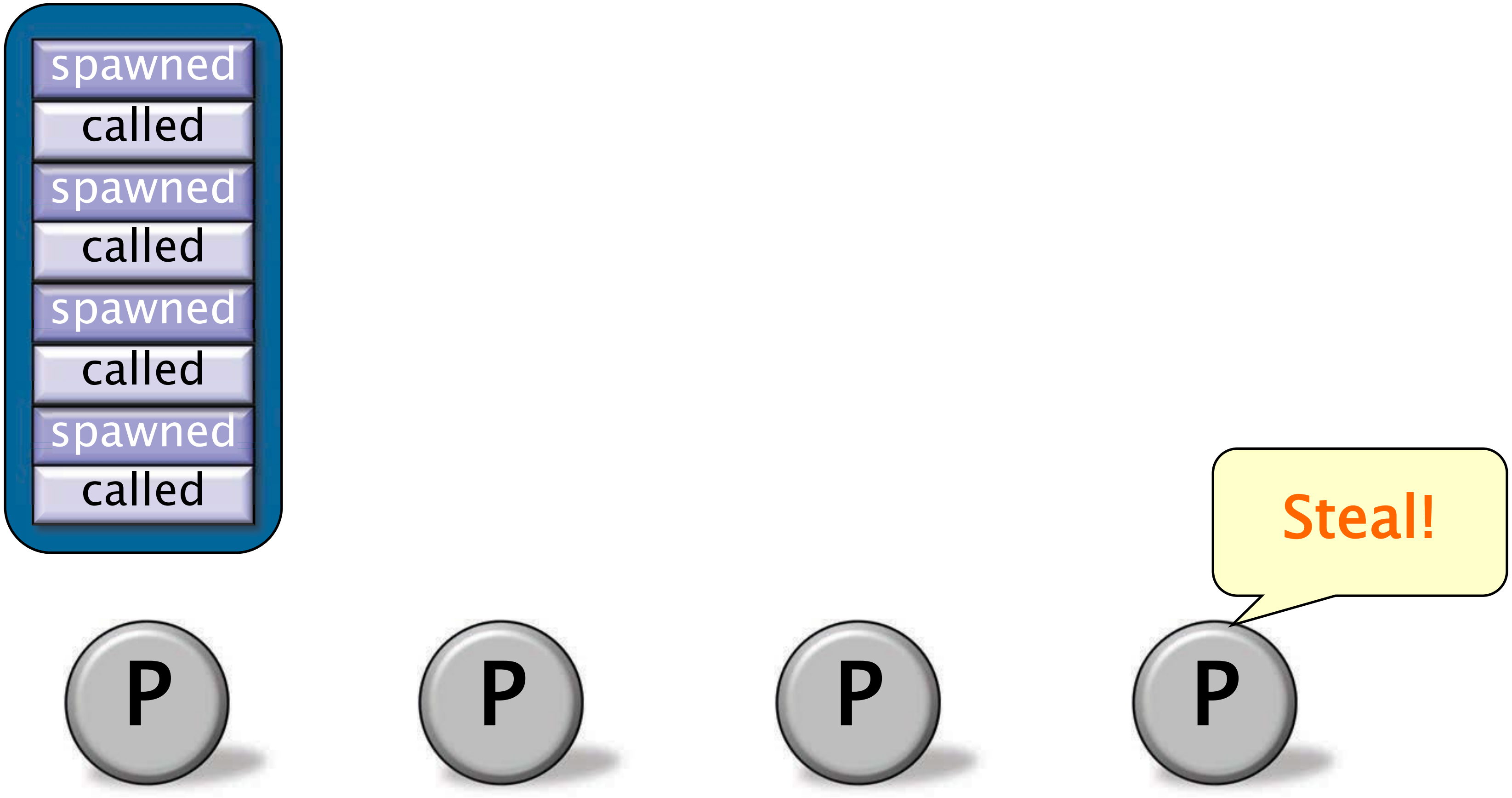
A full frame keeps track of its parent and child frames.

Other full frames are suspended.

Processors work on active full frames.

Maintaining Full Frames

Let’s see how steals can produce a tree of full frames.



spawned
called
spawned
called
spawned
called
spawned
called

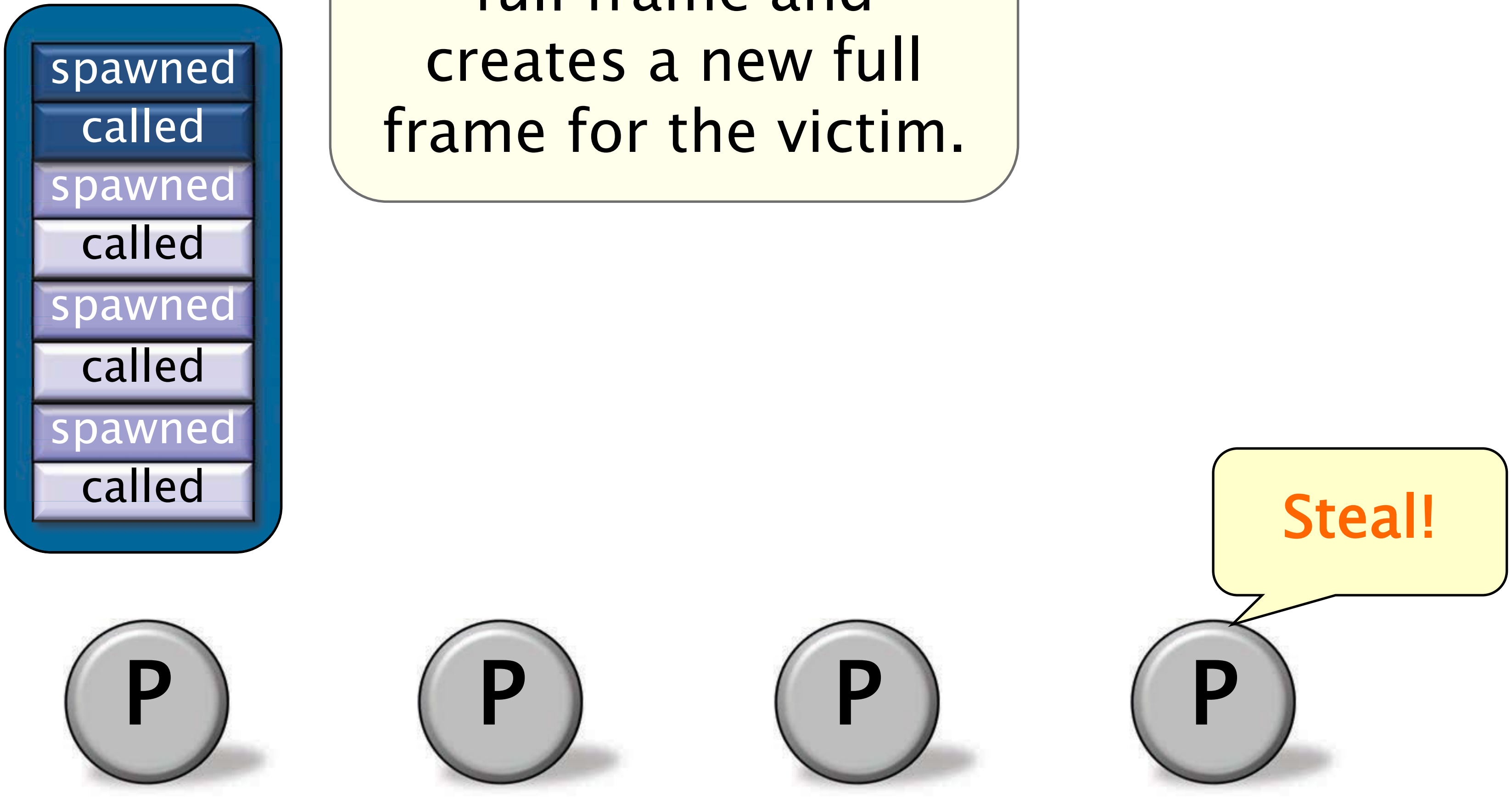
Steal!

P P P P

Maintaining Full Frames

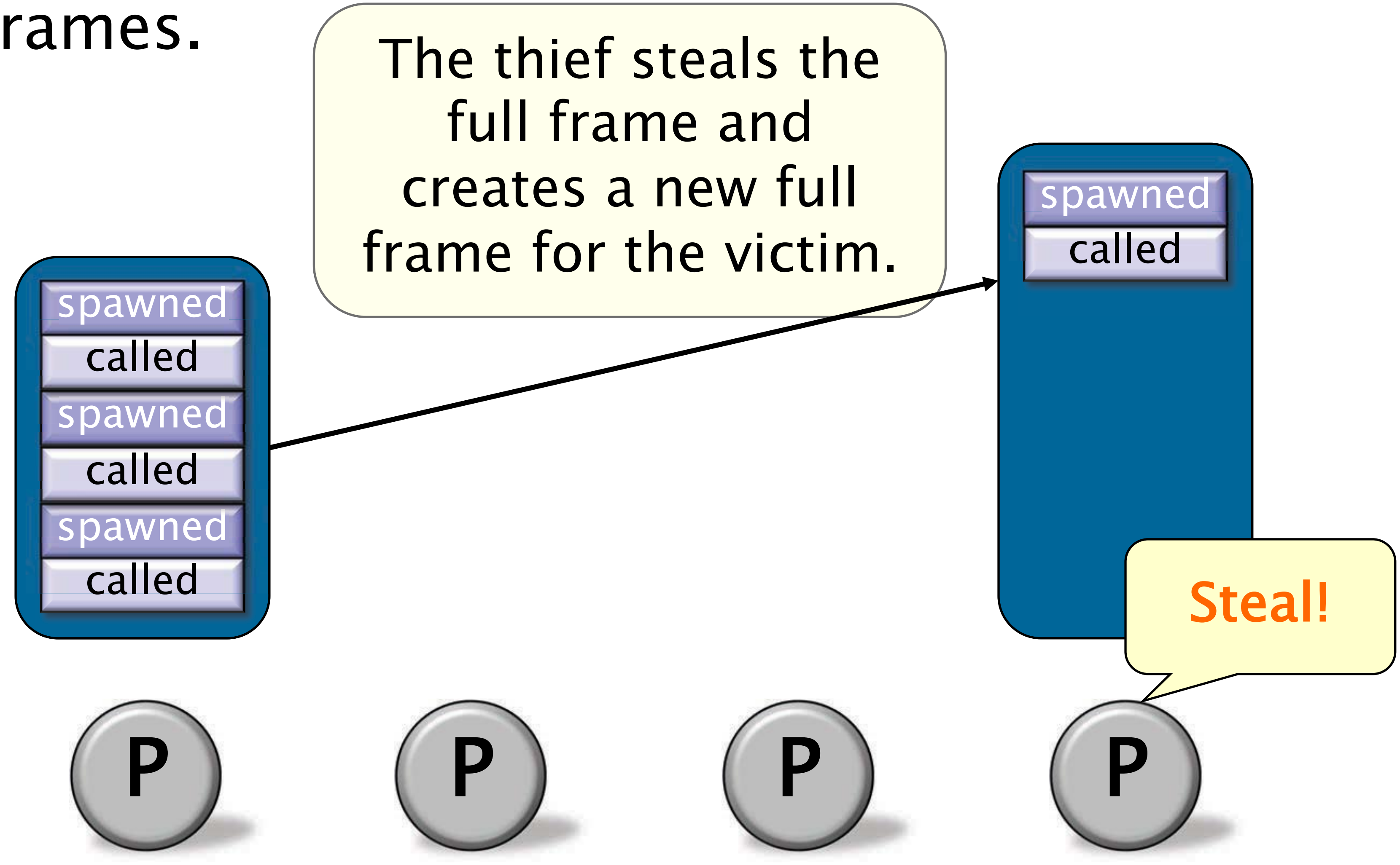
Let's see how steals can produce a tree of full frames.

The thief steals the full frame and creates a new full frame for the victim.



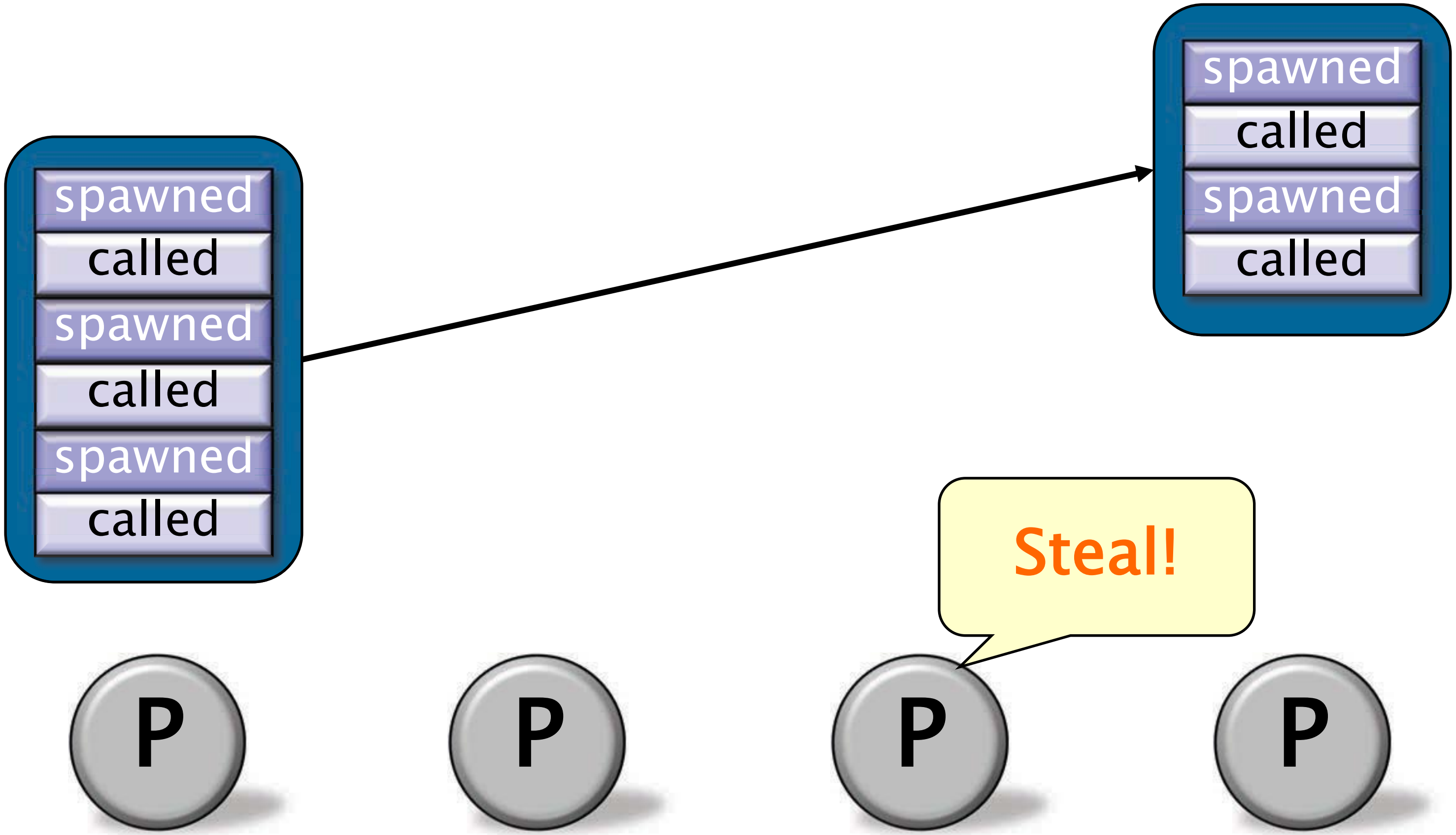
Maintaining Full Frames

Let’s see how steals can produce a tree of full frames.


The thief steals the full frame and creates a new full frame for the victim.

Maintaining Full Frames

Let's see how steals can produce a tree of full frames.

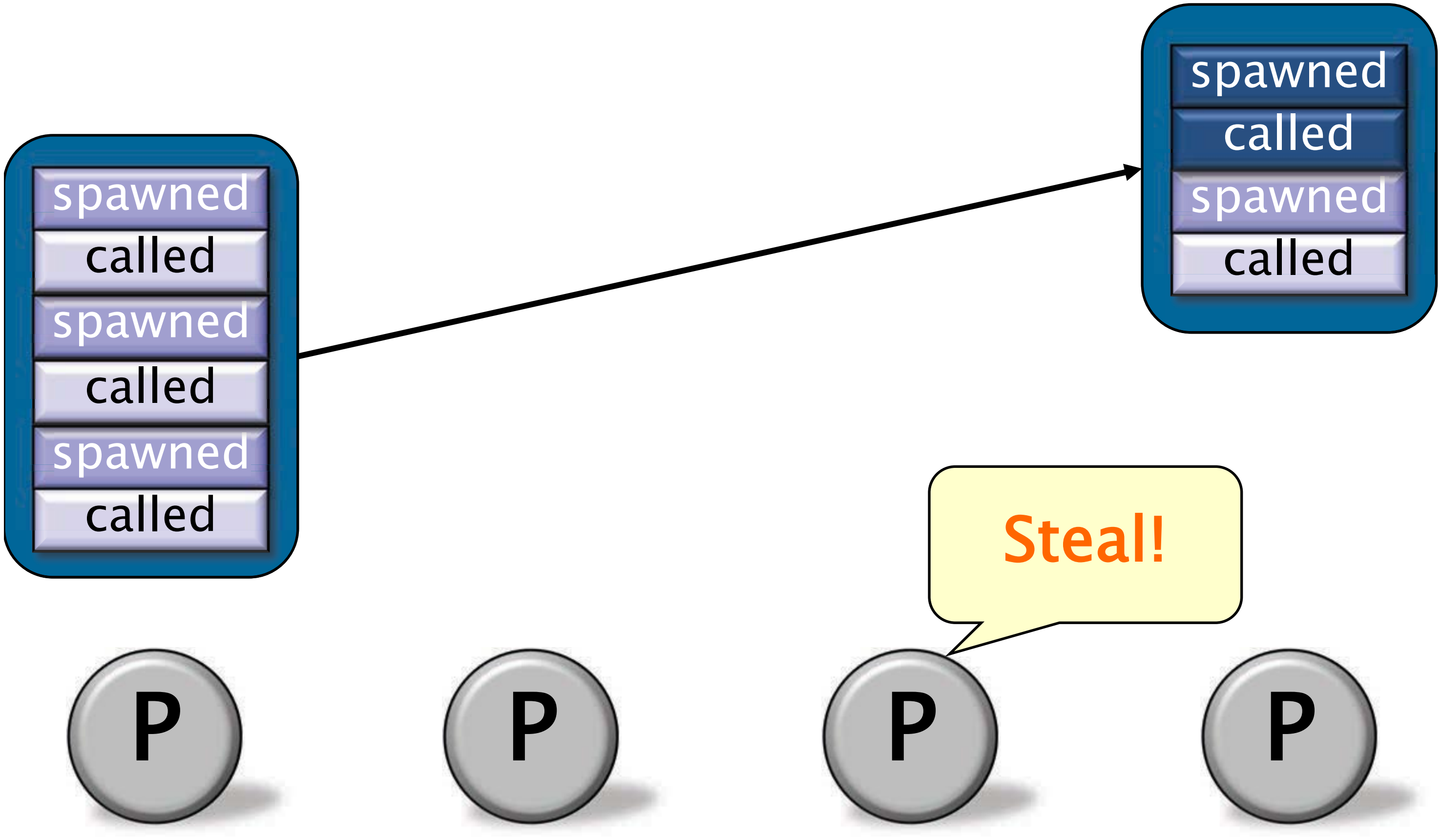


---

*Full-frame illustrations resized for cleanliness.

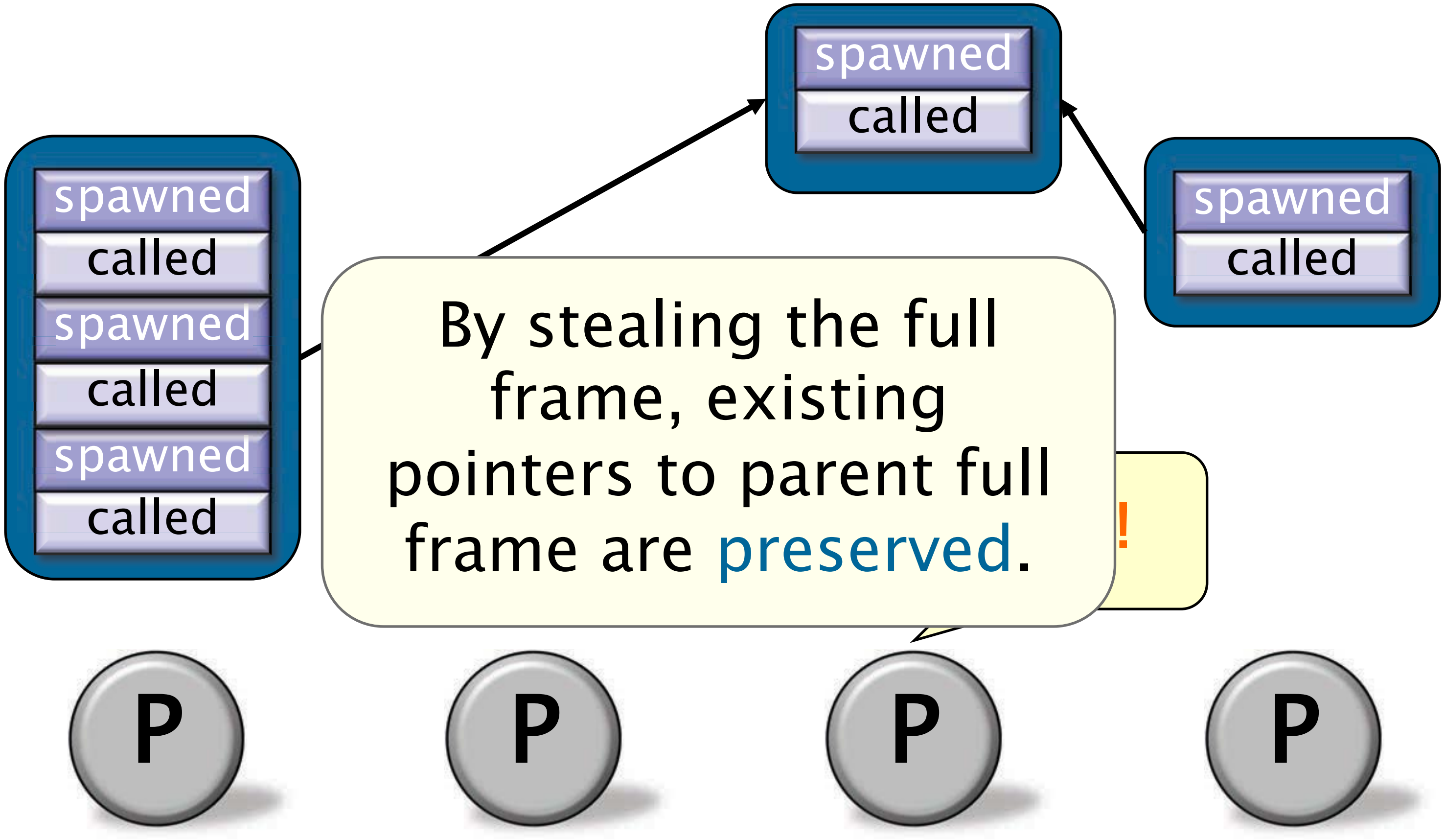
Maintaining Full Frames

Let’s see how steals can produce a tree of full frames.



Maintaining Full Frames

Let’s see how steals can produce a tree of full frames.



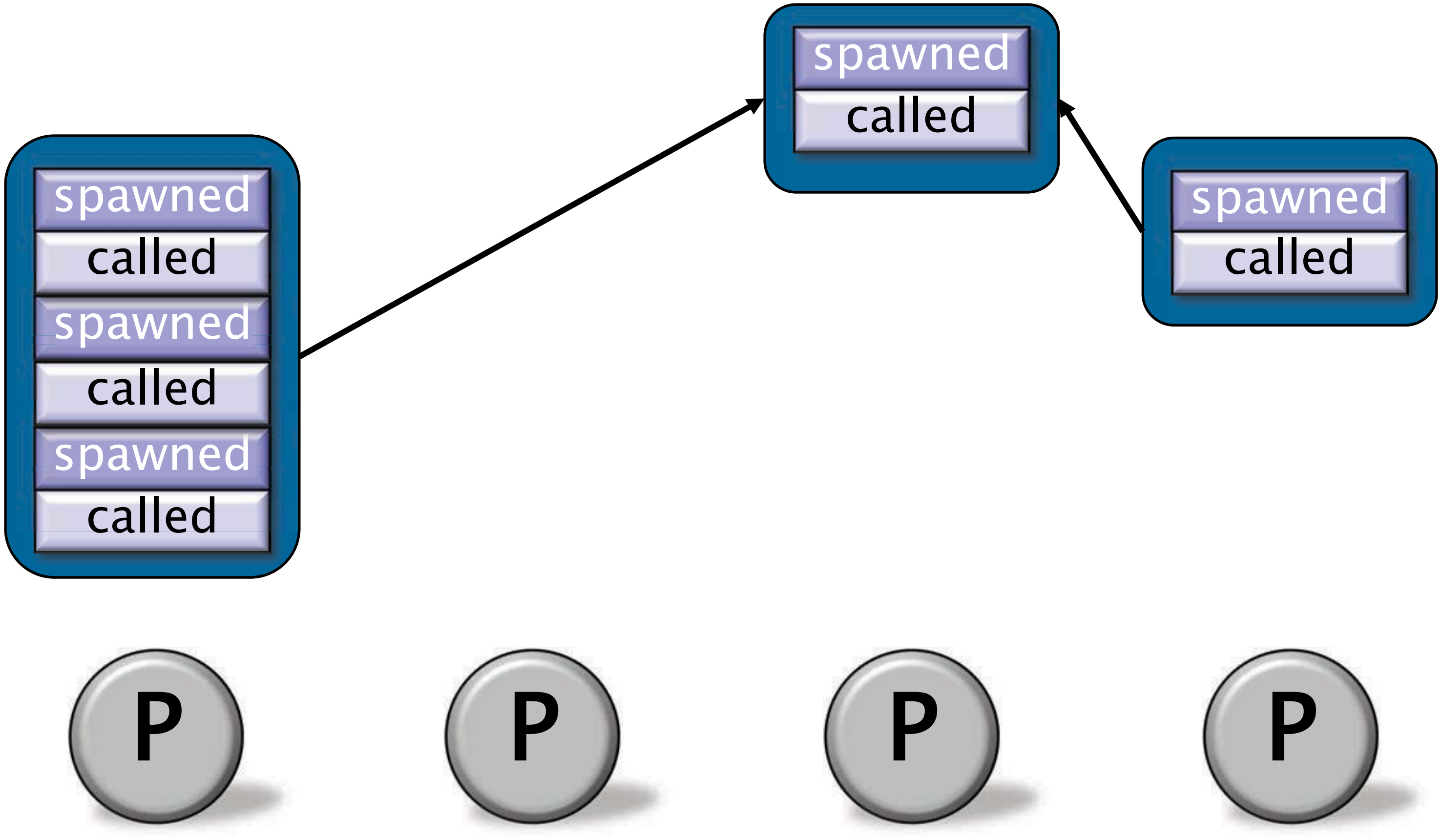
By stealing the full frame, existing pointers to parent full frame are preserved.

---

*Full-frame illustrations resized for cleanliness.

Maintaining Full Frames

Let's see how steals can produce a tree of full frames.

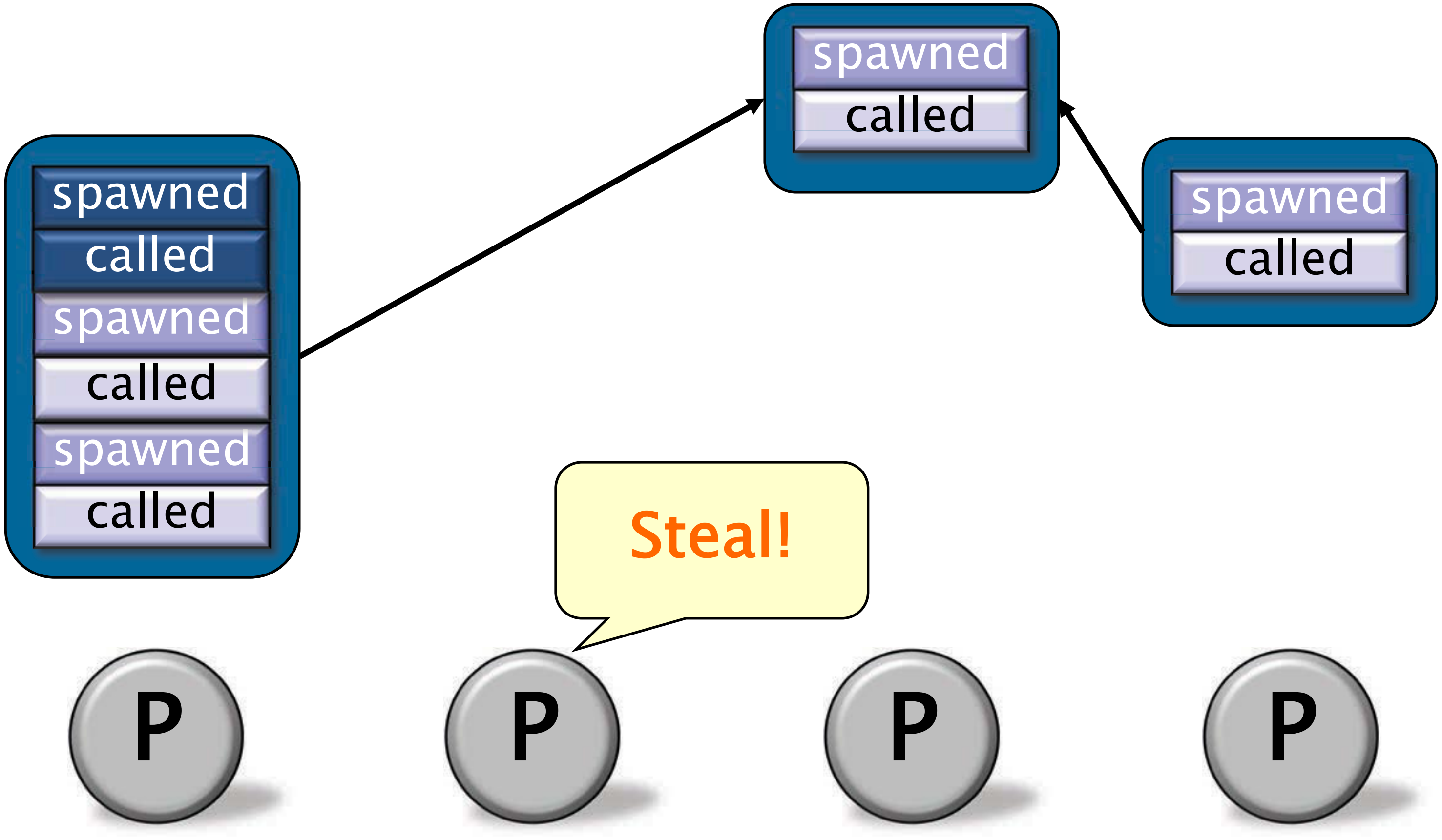


---

*Full-frame illustrations resized for cleanliness.

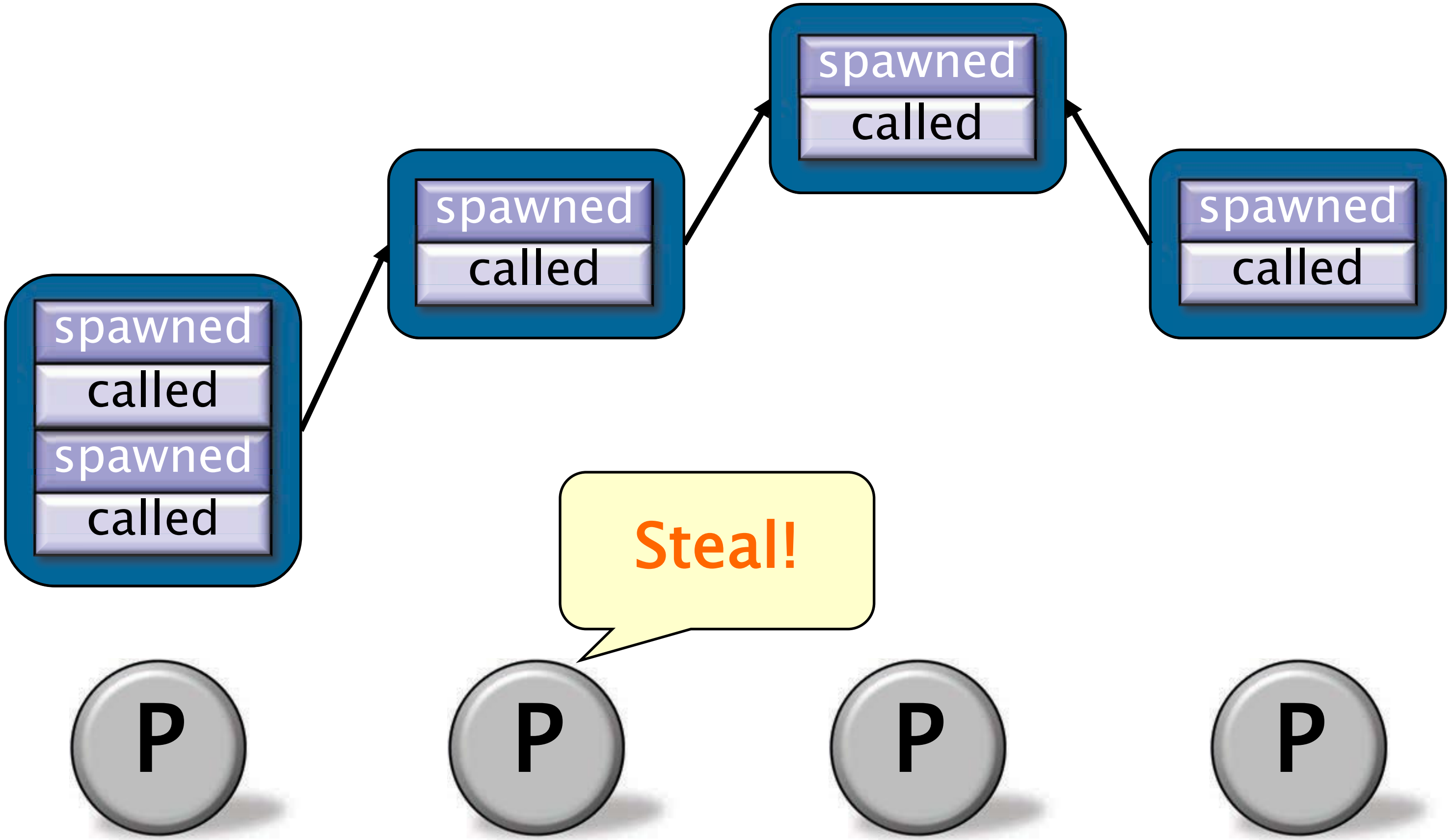
Maintaining Full Frames

Let's see how steals can produce a tree of full frames.



Maintaining Full Frames

Let's see how steals can produce a tree of full frames.

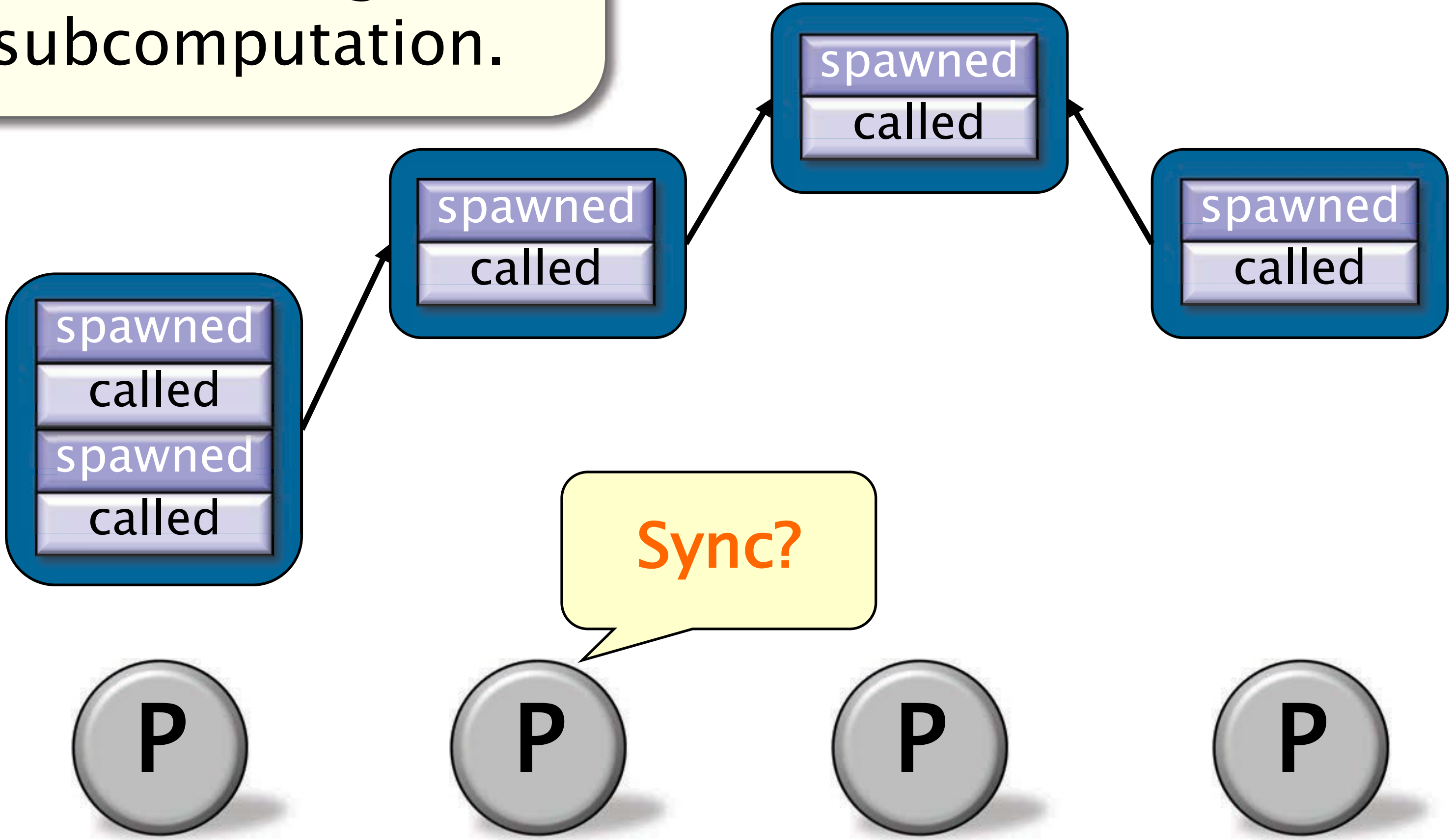


---

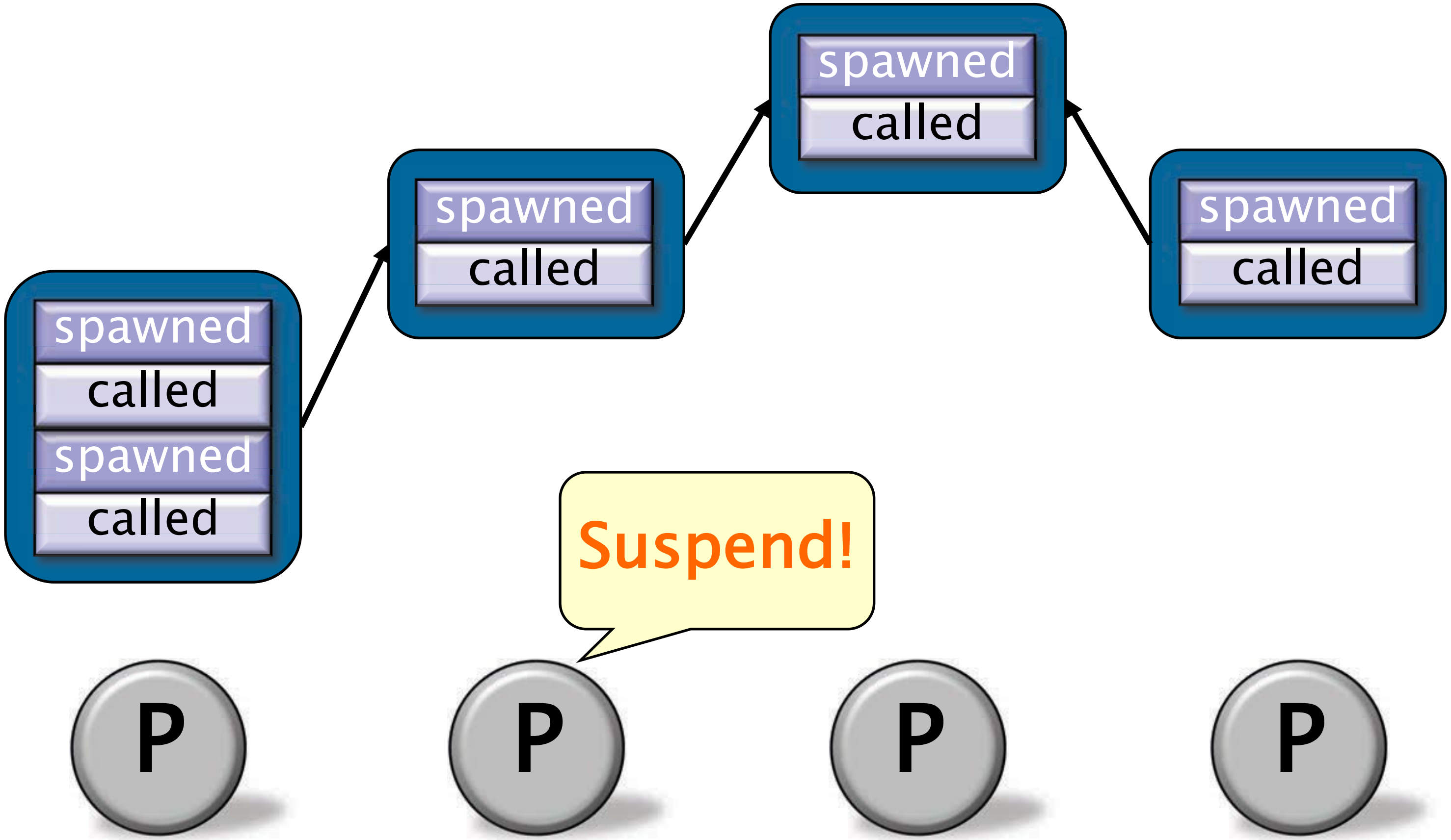
*Full-frame illustrations resized for cleanliness.

Suspending Full Frames

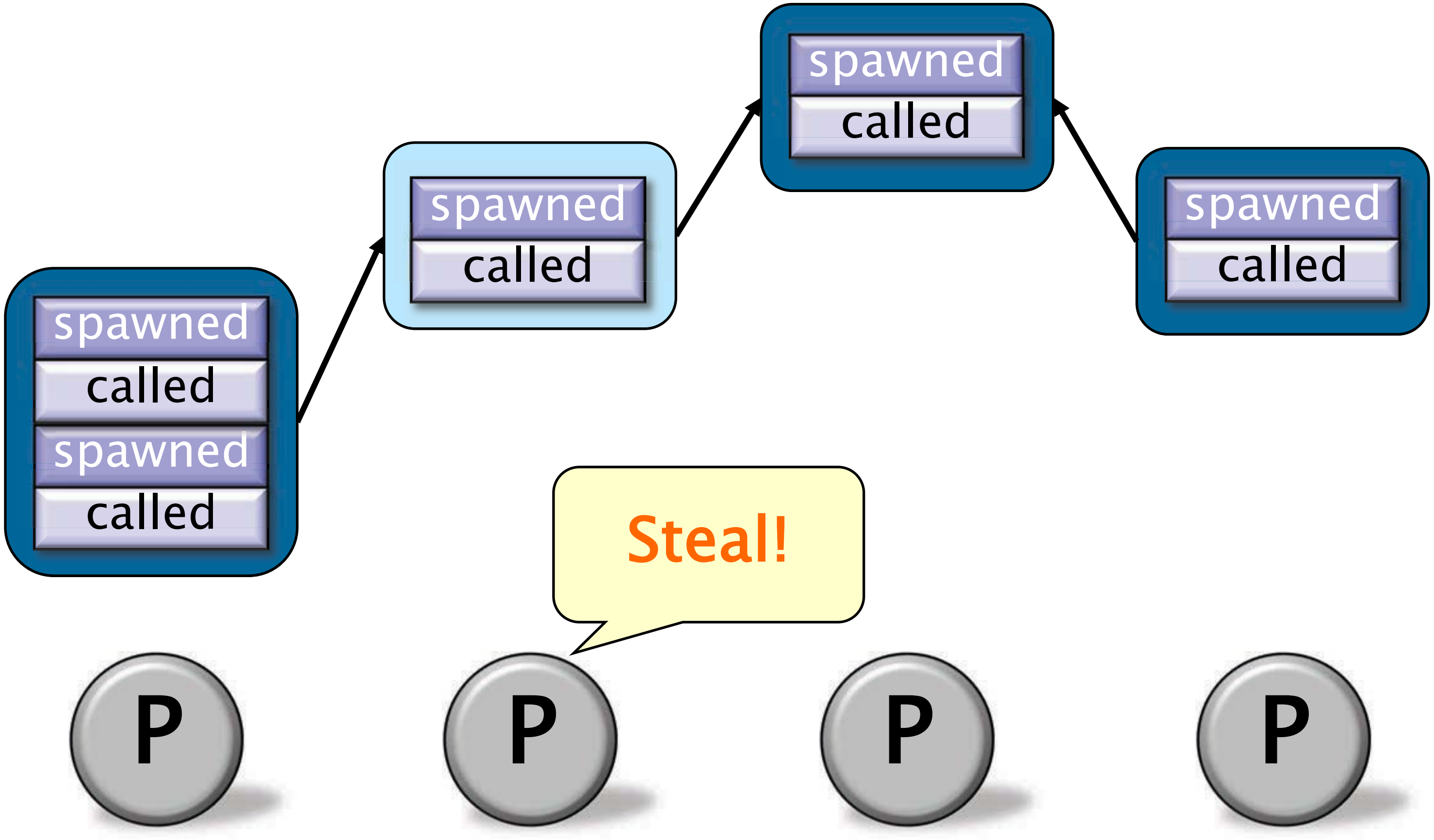
The full frame cannot sync because of the running child subcomputation.



Suspending Full Frames



Suspending Full Frames



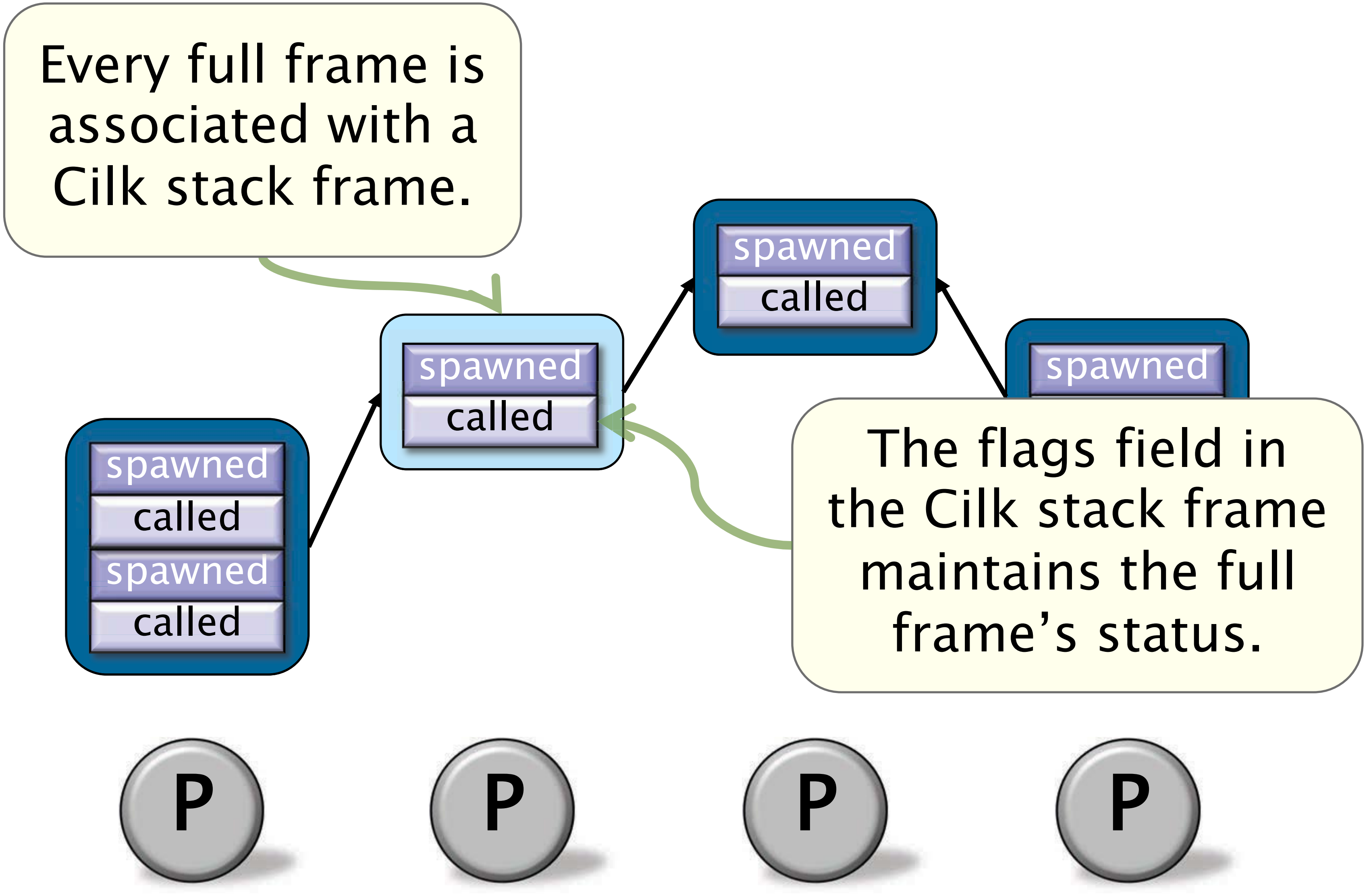
Common Case for Sync

QUESTION: If the program has ample parallelism, what do we expect will typically happen when the program execution reaches a `cilk_sync`?

ANSWER: The executing function contains no outstanding spawned children.

How does the runtime optimize for this case?

Stack Frames and Full Frames



Every full frame is associated with a Cilk stack frame.

The flags field in the Cilk stack frame maintains the full frame's status.

Compiled Code for Sync

Cilk code

```
int foo(int n) {
  int x, y;
  x = cilk_spawn bar(n);
  y = baz(n);
  cilk_sync;
  return x + y;
}
```

C pseudocode

```
int foo(int n) {
  …
  if (sf.flags &
        CILK_FRAME_UNSYNCHED)
    if (!setjmp(sf.ctx))
        __cilkrts_sync(&sf);
  …
}
```

The code compiled to implement a `cilk_sync` checks the flags field before performing an expensive call to `__cilkrts_sync` in the Cilk runtime library.

More Cilk Runtime Features

The Cilk runtime system implements many other features and optimizations:

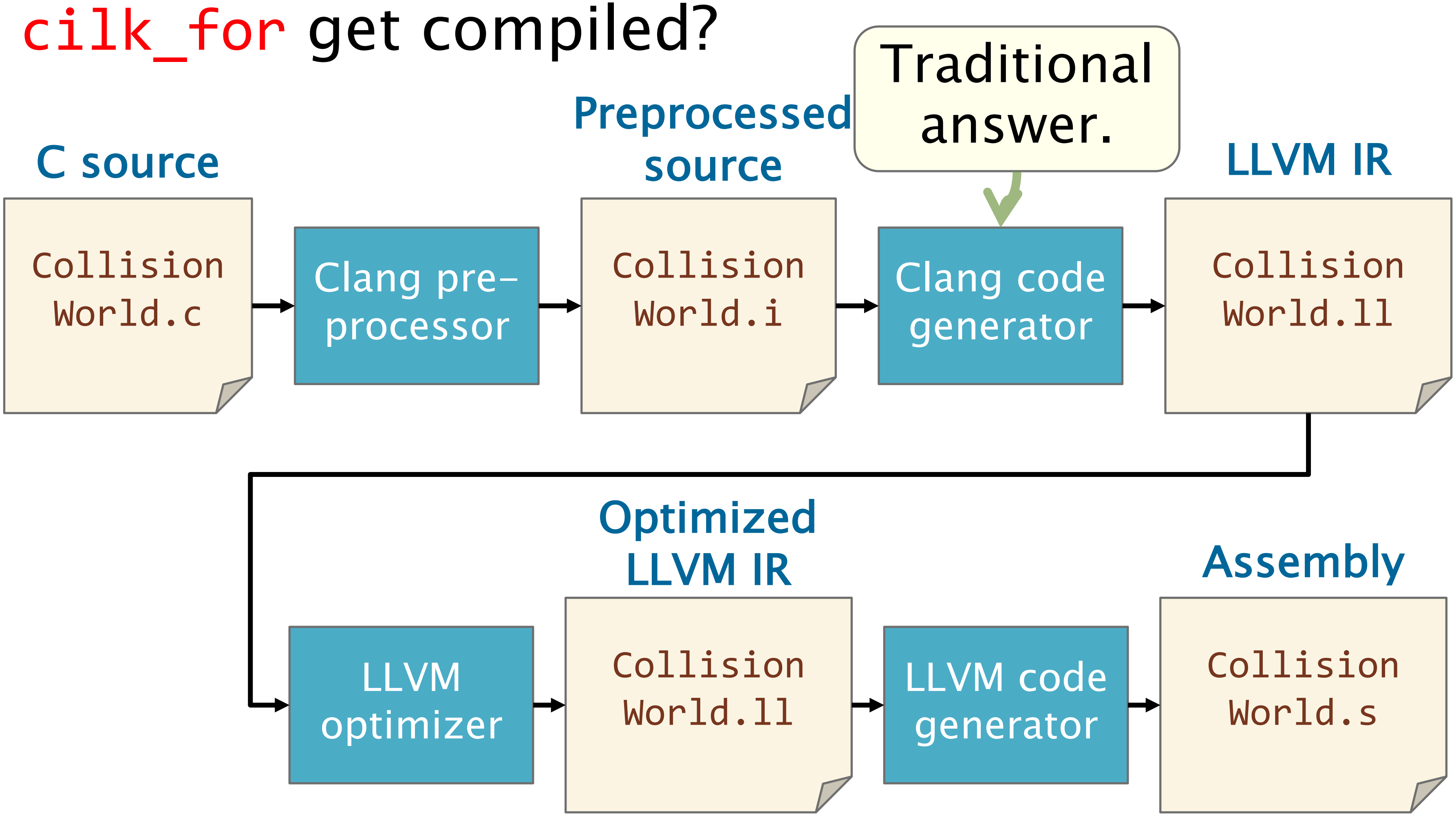
- Schemes for making the full-frame tree simpler and easier to maintain.
- Data structure and protocol enhancements to support C++ exceptions.
- Sibling pointers between full frames to support reducer hyperobjects.
- Pedigrees to assign a unique, deterministic ID to each strand efficiently in parallel.

THE TAPIR COMPILER

[SML17]

Compilation Pipeline

Question: When do `cilk_spawn`, `cilk_sync`, and `cilk_for` get compiled?



Traditional answer.

C source

Collision World.c

Clang pre-processor

Preprocessed source

Collision World.i

Clang code generator

LLVM IR

Collision World.ll

LLVM optimizer

Optimized LLVM IR

Collision World.ll

LLVM code generator

Assembly

Collision World.s

Example: Normalize

```
__attribute__((const))
double norm(const double *X, int n);

void normalize(double *restrict Y,
               const double *restrict X,
               int n) {
  for (int i = 0; i < n; ++i)
    Y[i] = X[i] / norm(X, n);
}
```

Test: Random vector of n=64M elements

Machine: AWS c4.8xlarge *Compiler:* GCC 6.2

Running time: $T_S = 0.312$ s

Optimizing the Serial Code

```
__attribute__((const))
double norm(const double *X, int n);

void normalize(double *restri
               const double *
               int n) {
  for (int i = 0; i < n; ++i)
    Y[i] = X[i] / norm(X, n);
}
```

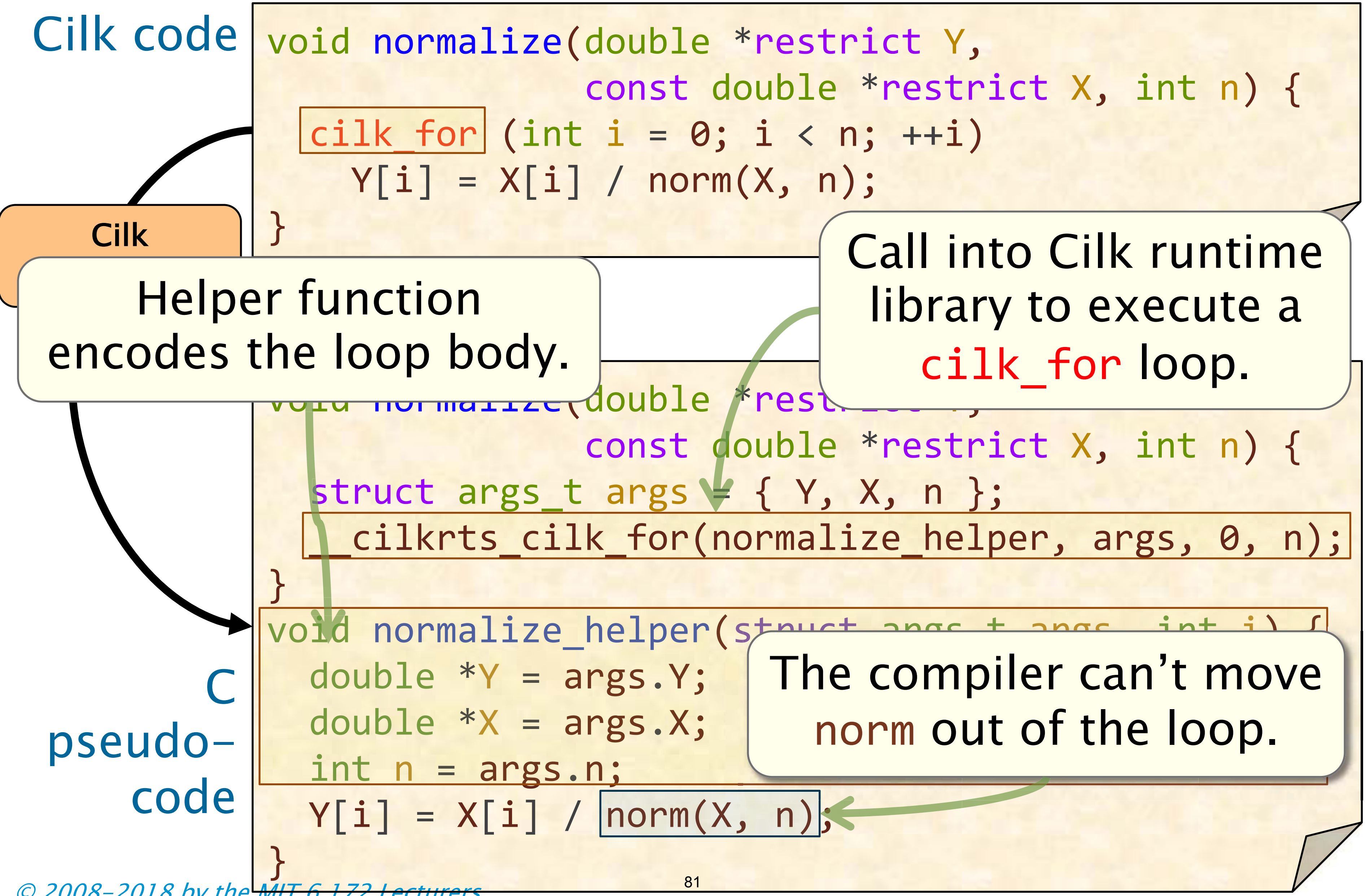
The `norm` function performs $\Theta(n)$ work.

GCC can move the call to `norm` out of the serial loop.

Work before hoisting: $T(n) = \Theta(n^2)$

Work after hoisting: $T(n) = \Theta(n)$

GCC Compiling Cilk Code



```
void normalize(double *restrict Y,
               const double *restrict X, int n) {
  cilk_for (int i = 0; i < n; ++i)
    Y[i] = X[i] / norm(X, n);
}
```

Helper function encodes the loop body.

Call into Cilk runtime library to execute a cilk_for loop.

```
void normalize(double *restrict Y,
               const double *restrict X, int n) {
  struct args_t args = { Y, X, n };
  __cilkrts_cilk_for(normalize_helper, args, 0, n);
}
void normalize_helper(struct args_t args, int i) {
  double *Y = args.Y;
  double *X = args.X;
  int n = args.n;
  Y[i] = X[i] / norm(X, n);
}
```

The compiler can't move norm out of the loop.

Performance of Parallel Normalize

```
__attribute__((const))
double norm(const double *X

void normalize(double *restrict Y,
               const double *restrict X,
               int n) {
  cilk_for (int i = 0;
    Y[i] = X[i] / norm
}
```

The `norm` function was also parallelized.

Terrible work efficiency!
$T_S/T_1 = 0.312/2600$
$\sim 1/8600$

Test: Random vector of $n=64\text{M}$ elements

Machine: AWS c4.8xlarge *Compiler:* GCC 6.2

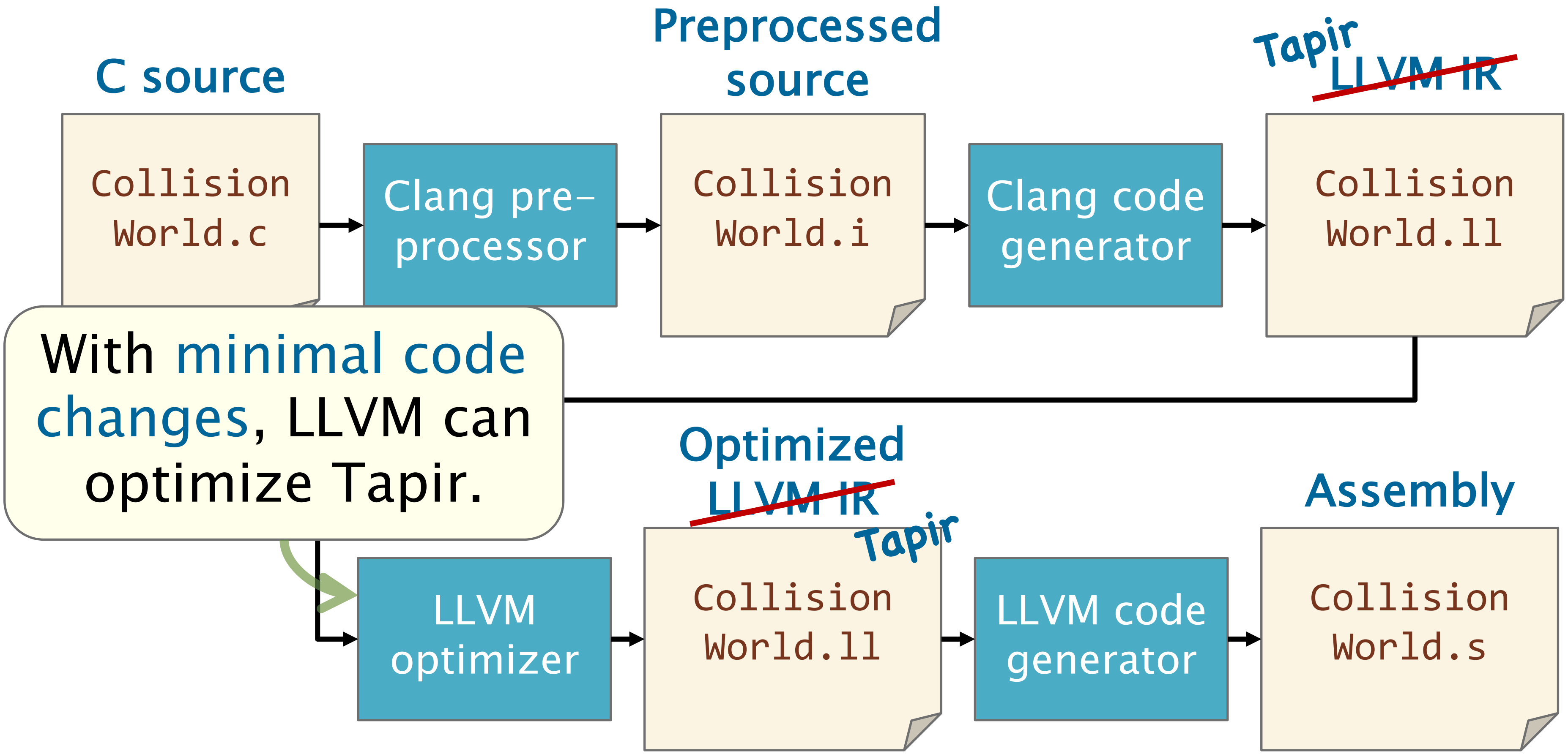
Running time of serial code: $T_S = 0.312$ s

18-core running time: $T_{18} = 180.657$ s

1-core running time: $T_1 = 2600.287$ s

Tapir’s Compilation Pipeline

Tapir embeds fork-join parallelism into LLVM’s IR.



C source

`Collision World.c`

Clang pre-processor

Preprocessed source

`Collision World.i`

Clang code generator

Tapir ~~LLVM IR~~

`Collision World.ll`

With minimal code changes, LLVM can optimize Tapir.

LLVM optimizer

Optimized ~~LLVM IR~~ Tapir

`Collision World.ll`

LLVM code generator

Assembly

`Collision World.s`

Unsafe Optimizations

Problem: There are many examples of optimizations on serial code that cannot be safely applied to parallel code [MP90].

Cilk code

```
void foo(int n) {
  cilk_for (int i = 0; i < n; ++i)
    bar(5*i);
}
```

Incorrectly optimized Cilk code

```
void foo(int n) {
  int tmp = 0;
  cilk_for (int i = 0; i < n; ++i) {
    bar(tmp);
    tmp += 5;
  }
}
```

Unleashing LLVM on parallel programs requires some care.

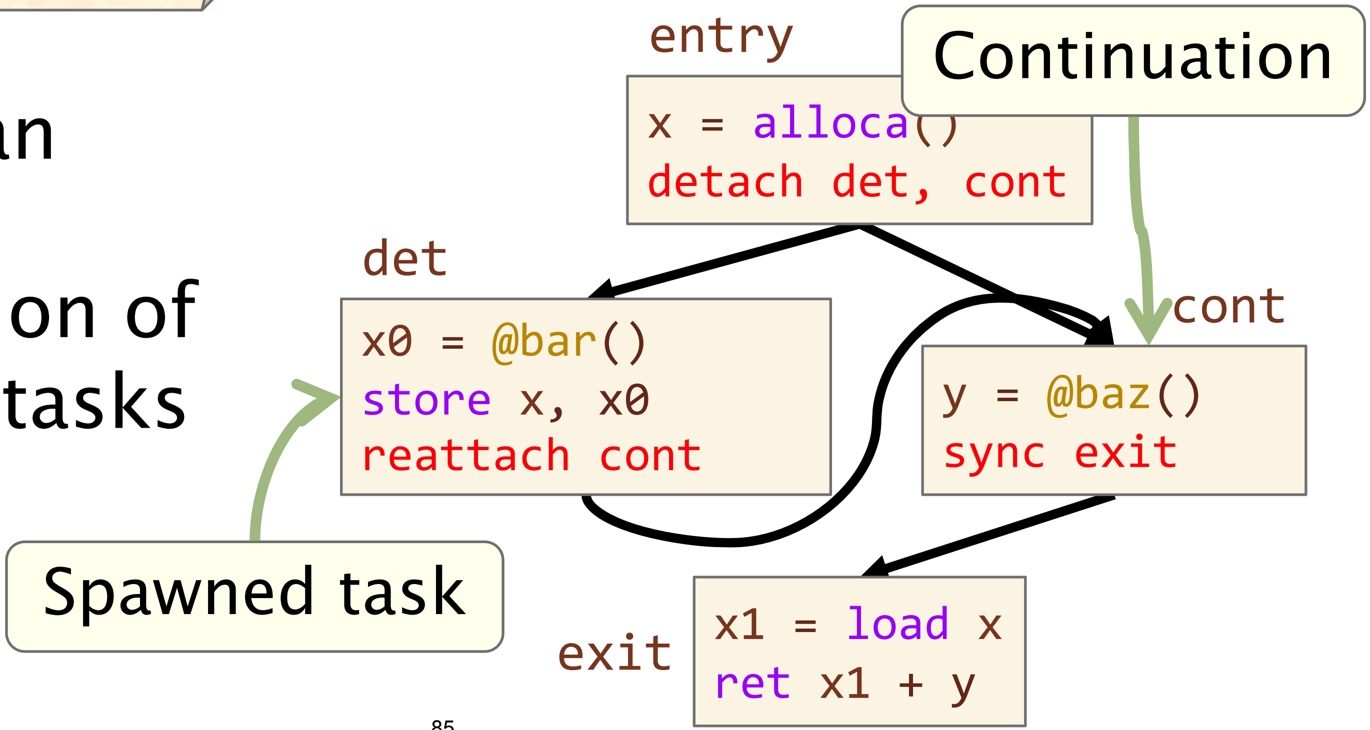
A Tapir CFG

```
int foo(int n) {
  int x, y;
  x = cilk_spawn bar(n);
  y = baz(n);
  cilk_sync;
  return x + y;
}
```

Tapir adds three constructs to LLVM’s IR: detach, reattach, and sync.

Tapir uses an asymmetric representation of the parallel tasks in the CFG.

Tapir control-flow graph



entry
```
x = alloca()
detach det, cont
```

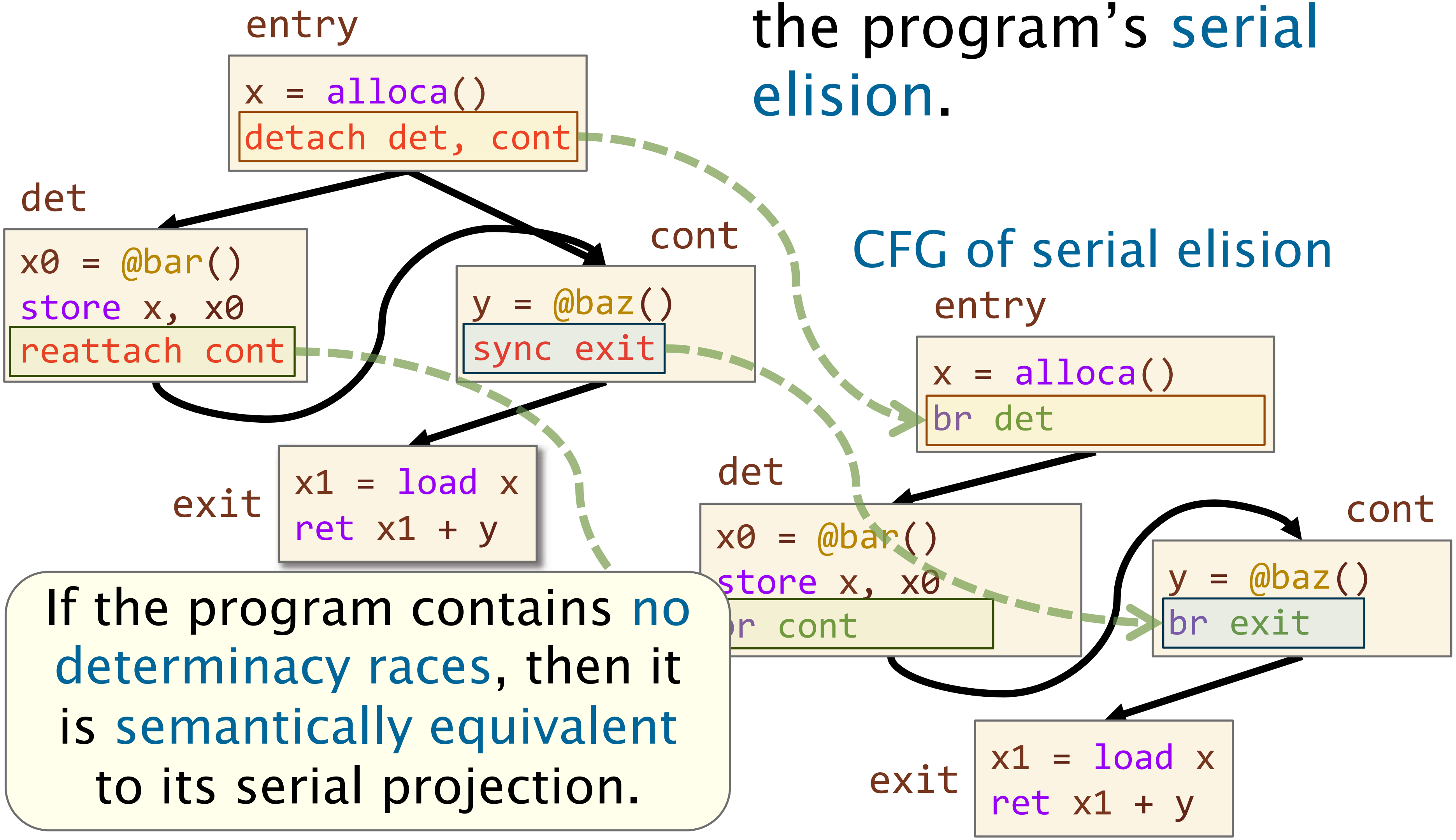
det
```
x0 = @bar()
store x, x0
reattach cont
```

cont
```
y = @baz()
sync exit
```

exit
```
x1 = load x
ret x1 + y
```

Serial Elision

This asymmetry models the program’s serial elision.



Tapir CFG

entry
```
x = alloca()
detach det, cont
```

det
```
x0 = @bar()
store x, x0
reattach cont
```

cont
```
y = @baz()
sync exit
```

exit
```
x1 = load x
ret x1 + y
```

CFG of serial elision

entry
```
x = alloca()
br det
```

det
```
x0 = @bar()
store x, x0
br cont
```

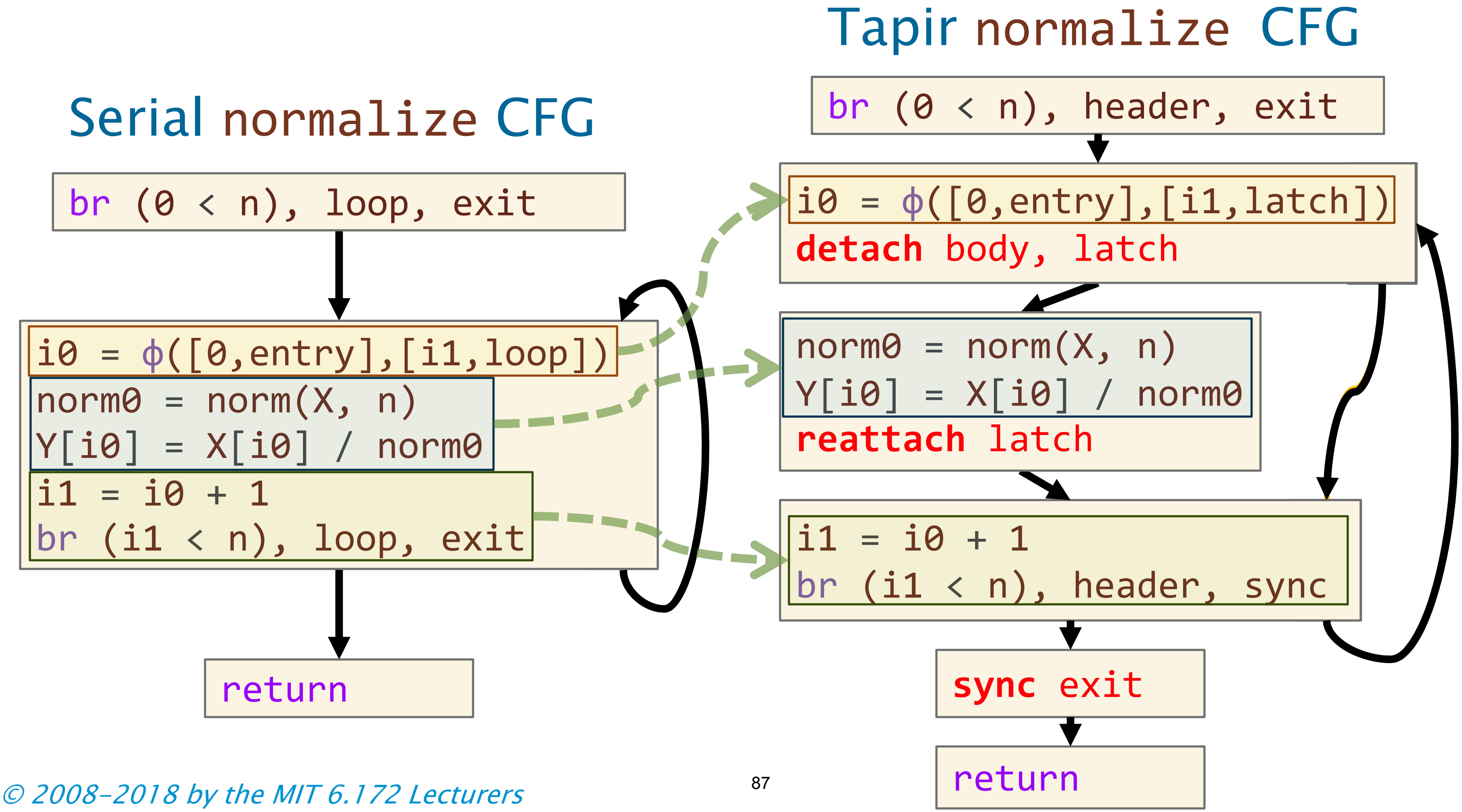
cont
```
y = @baz()
br exit
```

exit
```
x1 = load x
ret x1 + y
```

If the program contains no determinacy races, then it is semantically equivalent to its serial projection.

Parallel Loops in Tapir

In Tapir, parallel loops look similar to serial loops, with some differences due to parallelism.



Serial normalize CFG

```
br (0 < n), loop, exit
```

```
i0 = ϕ([0,entry],[i1,loop])
norm0 = norm(X, n)
Y[i0] = X[i0] / norm0
i1 = i0 + 1
br (i1 < n), loop, exit
```

```
return
```

Tapir normalize CFG

```
br (0 < n), header, exit
```

```
i0 = ϕ([0,entry],[i1,latch])
detach body, latch
```

```
norm0 = norm(X, n)
Y[i0] = X[i0] / norm0
reattach latch
```

```
i1 = i0 + 1
br (i1 < n), header, sync
```

```
sync exit
```

```
return
```

Impact on LLVM

LLVM can reason about a Tapir CFG as a relatively minor change to the CFG of the serial elision.

- Many standard compiler analyses required no changes.
- Memory analysis required a minor change to handle Tapir's constructs (~450 lines of code).
- Some optimizations, e.g., code hoisting and tail-recursion elimination, required some changes to work on Tapir CFG's.

In total, implementing Tapir involved adding or modifying ~6000 lines of LLVM's 4-million-line codebase.

Parallelize Normalize with Tapir

```
__attribute__((const))
double norm(const double *X, int n);

void normalize(double *restrict Y,
               const double *restrict X,
               int n) {
  cilk_for (int i = 0; i
    Y[i] = X[i] / norm(X
}
```

Good work efficiency:
$T_S/T_1 = 97\%$

Test: Random vector of $n=64M$ elements

Machine: AWS c4.8xlarge *Compiler:* Tapir/LLVM

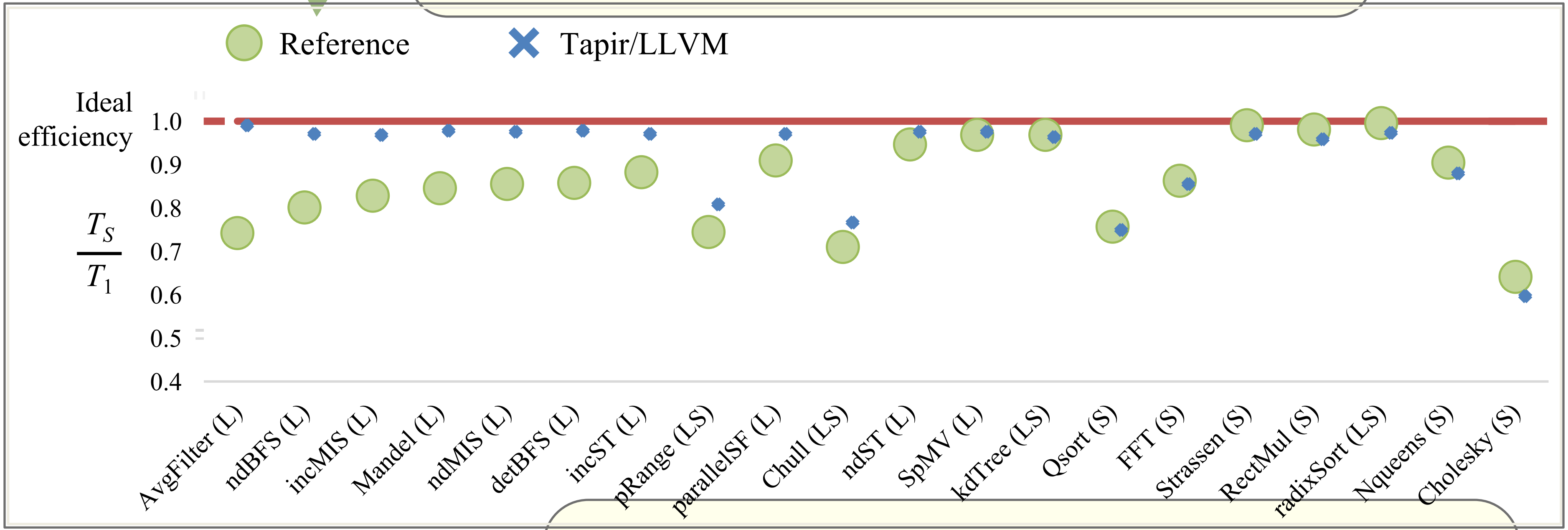
Running time of serial code: $T_S = 0.312$ s

1-core running time: $T_1 = 0.321$ s

18-core running time: $T_{18} = 0.081$ s

Work-Efficiency Improvement

Same as Tapir/LLVM except that Cilk constructs are compiled early.



Tapir/LLVM doesn't fix everything, but it helps parallel programs achieve good work efficiency.

CASE STUDY: OPENMP

Example: OpenMP Normalize

```
__attribute__((const))
double norm(const double *X

void normalize(double *restrict Y,
               const double *restrict X,
               int n) {
    agma omp parallel for
      (int i = 0; i < n; ++i)
     [i] = X[i] / norm(X, n);
}
```

The `norm` function was also parallelized.

Why do we get this performance?

Good work efficiency without Tapir?

Test: Random vector of n=64M elements

Machine: AWS c4.8xlarge *Compiler:* GCC 6.2

Running time of serial code: $T_S = 0.312$ s

1-core running time: $T_1 = 0.329$ s

18-core running time: $T_{18} = 0.205$ s

Parallel speedup is not great.

GCC Compiling OpenMP Code

OpenMP code

Op
cor

```
void normalize(double *restrict Y,
                         const double *restrict X, int n) {
  #pragma omp parallel for
  for (int i = 0; i < n; ++i)
                       orm(X,
```

Each processor invokes this helper method on n/P iterations.

The helper function’s loop on n/P iterations can be optimized.

```
                            estric
                            uble *restrict X, int n) {
  __kmpc_fork_call(omp_outlined, n, Y, X);
}
void omp_outlined(int n, double *restrict Y,
                         const double *restrict X) {
  int local_n = n; double *local_Y = Y, *local_X = X;
  __kmpc_for_static_init(&local_n, &local_Y, &local_X);
  double tmp = norm(X, n);
  for (int i = 0; i < local_n; ++i)
    local_Y[i] = local_X[i] / tmp;
  __kmpc_for_static_fini();
}
```

Analysis of OpenMP Normalize

Each processor invokes `omp_outlined` on n/P iterations.

The `norm` function performs $\Theta(n)$ work.

```
                         restrict Y,
                         ouble *restric
                         outlined, n, Y, X);
}
void omp_outlined(int n, double *restrict Y,
                  const double *restrict X) {
  int local_n = n; double *local
  __kmpc_for_static_init(&local_
  double tmp = norm(X, n);
  for (int i = 0; i < local_n; ++i)
    local_Y[i] = local_X[i] / tmp;
  __kmpc_for_static_fini();
}
```

The variable `local_n` is approximately n/P.

Work of omp_outlined: $T(n) = \Theta(n)$

Total work on P processors: $T(n) = \Theta(Pn)$

Summary of OpenMP Normalize

```
__attribute__((const))
double norm(const double *X, int n);

void normalize(double *restrict Y,
               const double *restrict X,
               int n) {
  #pragma omp parallel for
  for (int i = 0; i < n; ++i)
    Y[i] = X[i] / norm(X, n);
}
```

Work on P processors: $T(n) = \Theta(Pn)$

- This code is only work efficient on 1 processor.
- This code can never achieve more than minimal parallel speedup.

Takeaways

The work–first principle

Optimize for *ordinary serial execution*, at the expense of some additional computation in steals.

Two more takeaways:

- Think about the performance model for your program.
- Know what your parallel runtime system is doing.

6.172 Performance Engineering of Software Systems
Fall 2018