A Quick Introduction to the Intel® Cilk™ Plus Runtime

Jim Sukha

Version 1.0

March 25, 2015

What is Intel® Cilk™ Plus?

Cilk Plus is an extension to C and C++ for task and vector multicore parallelism.

- Keywords for task parallelism
  - **`cilk_spawn`**, **`cilk_sync`**, and **`cilk_for`**
- Language extensions for vector parallelism
- Support in major compilers
  - Intel Cilk Plus: part of Intel compiler
  - GCC mainline, 4.9.2 and later
  - LLVM/Clang branch
- Open-source runtime

```
git clone
https://bitbucket.org/intelcilkruntime/intel-cilk-runtime.git
```

A Brief History of Cilk by Papers

- **MIT Cilk**: extension of C, originated out of MIT Laboratory for Computer Science in mid 1990's. Foundational research papers include:
  - **Scheduling Multithreaded Computations by Work Stealing**
    by Robert D. Blumofe and Charles E. Leiserson. *Journal of the ACM*, 720–748, September, 1999
  - **The Implementation of the Cilk-5 Multithreaded Language**
    by Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. *PLDI* 1998.
  - **Efficient Detection of Determinacy Races in Cilk Programs**
    by Mingdong Feng and Charles E. Leiserson. *SPAA* 1997.
  - **Detecting Data Races in Cilk Programs That Use Locks**
    by Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. *SPAA* 1998.
- **Cilk++**: commercial C++ implementation by Cilk Arts, a startup founded in 2006. Added support for reducers.
  - **Reducers and Other Cilk++ Hyperobjects**
    by Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. *SPAA* 2009.
- **Intel® Cilk™ Plus**: Intel acquired Cilk Arts in 2009.
  - Changed to use legacy-compatible calling conventions.
  - Added "Plus" extensions for vector parallelism.
  - Added support for pedigrees.

    Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms by Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. *PPoPP* 2012.

A Canonical Cilk Plus Program

A recursive Cilk Plus function:

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

The function **`fib`** is a spawning function --- it contains **`cilk_spawn`** and **`cilk_sync`** keywords.

The **`cilk_spawn`** indicates that `fib(n-1)` is allowed to execute in parallel with the continuation, `fib(n-2)` in this case.

Execution does not continue after **`cilk_sync`** in a parent function until all child functions spawned from the parent have returned.

How are **`cilk_spawn`** and **`cilk_sync`** implemented?

Cilk Plus: Compiler vs. Runtime

The implementation of Cilk Plus is split between a compiler and runtime library:

Compiler:

- Generates assembly code for a spawning function, which contains calls to the runtime.
- Fast paths optimized for execution with no steals.
- Represents a spawning function via a (Cilk runtime) **stack frame**.
- Difficult to modify because changes may affect ABI.

Runtime:

- Dynamically loaded library handles scheduling on multiple **worker threads** via work-stealing.
- Slower paths to handle steals.
- Represents a spawning function via a **full frame**, which is larger.
- Relatively easy to create new compatible versions.

Cilk Plus Runtime Data Structures

Cilk Plus has three primary data structures for tracking frames as a program executes:

1. Every worker maintains a **deque** to store stack frames that can be stolen.
2. The compiler maintains a **call chain** of stack frames, to track a worker's currently executing stack frame.
3. The runtime maintains a **full frame tree**, to track suspended frames and store other runtime state.

Caveat: This presentation focuses on explaining these three data structures and their associated runtime invariants. Some other important aspects of the runtime are not covered here.

Outline

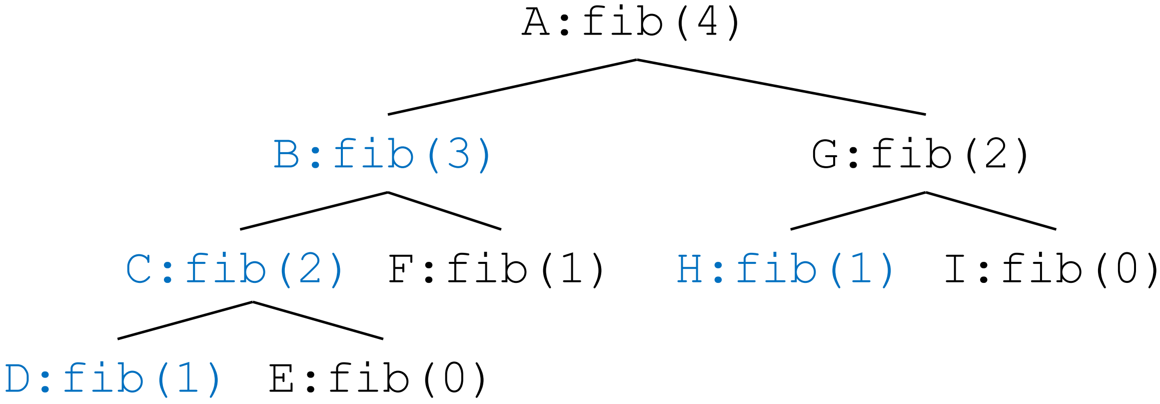
- Deques and work-stealing
- Full frames in the Cilk Plus runtime
- Compiling a spawning function
- Stealing work
- Reducers
- Other runtime features

Example: Stack Frames for `fib(4)`

To understand the data structures for Cilk Plus, consider the execution of `fib(4)`.

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

Each instance of fib creates an associated **stack frame**.



Each stack frame conceptually represents the state of a function, including local variables (e.g., `x`, and `y` for `fib`).

Deques for Work Stealing

A Cilk Plus program executes on worker threads.

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

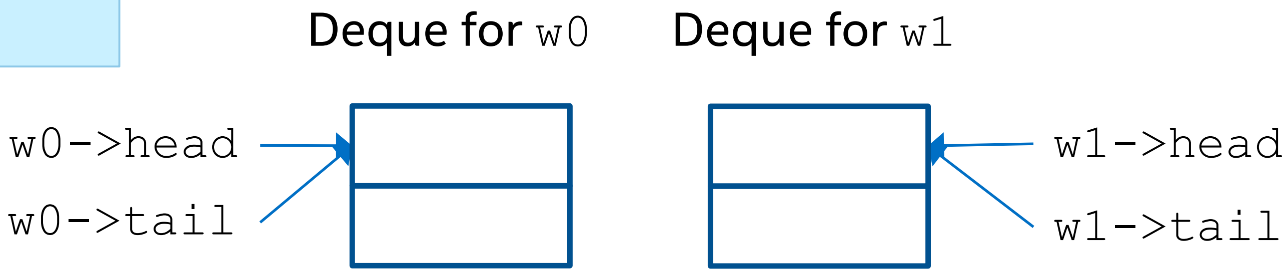
Every worker stores work to steal on a **deque**.

When a program begins, all deques are empty.

Current frame:

w0: A:fib4

w1: ---



Deque for `w0`

Deque for `w1`

Deques: `cilk_spawn`

Consider an execution of `fib(4)` that begins on worker `w0`.

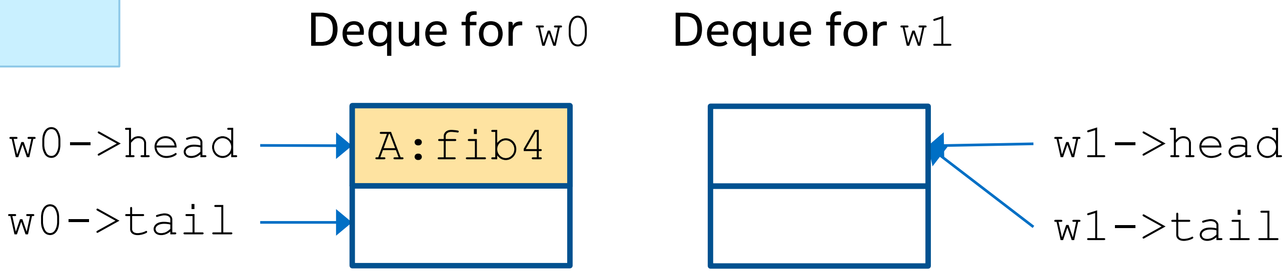
```
int fib(int n)
{
    if (n < 2)
        return n;
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

- When `w0` executes `cilk_spawn fib(3)`, it pushes the stack frame for the continuation of `A:fib4` onto its deque.

Current frame:

`w0:   B:fib3`

`w1:   ---`



Deques: `cilk_spawn`

Consider an execution of `fib(4)` that begins on worker `w0`.

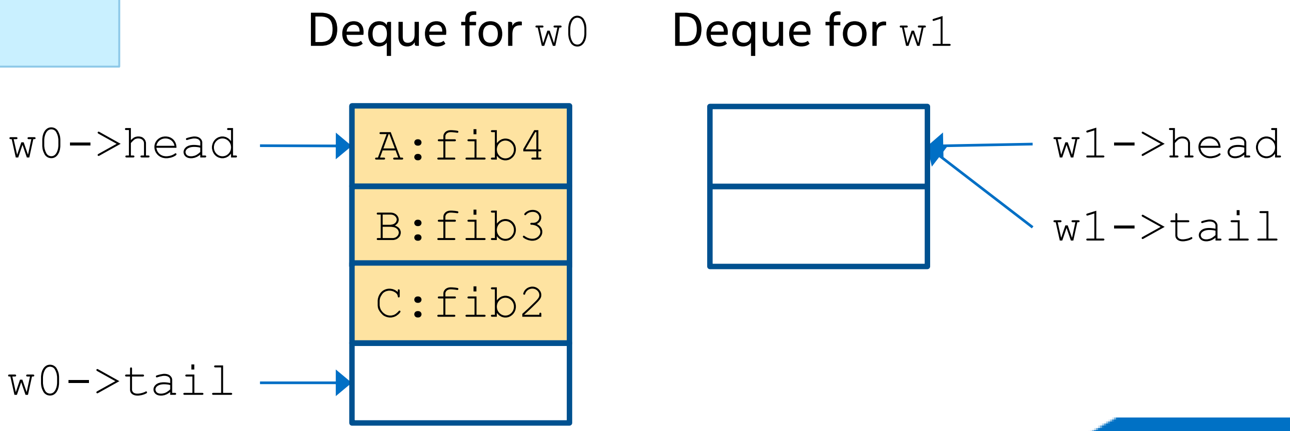
```
int fib(int n)
{
    if (n < 2)
        return n;
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

- When `w0` executes `cilk_spawn fib(3)`, it pushes the stack frame for the continuation of `A:fib4` onto its deque.
- As `w0` executes `B:fib3`, it will push additional frames onto the deque.

Current frame:

`w0:   D:fib1`

`w1:   ---`



Deques: Steal

Another worker `w1` can steal the work from the top of `w0`'s deque.

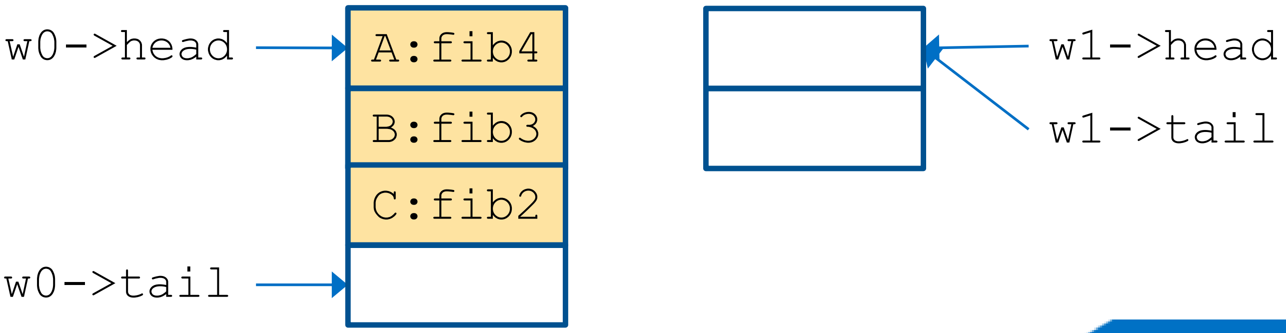
```
int fib(int n)
{
    if (n < 2)
        return n;
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

Current frame:

`w0:   D:fib1`

`w1:   ---`

Deque for `w0`   Deque for `w1`



Deques: Steal

Another worker `w1` can steal the work from the top of `w0`’s deque.

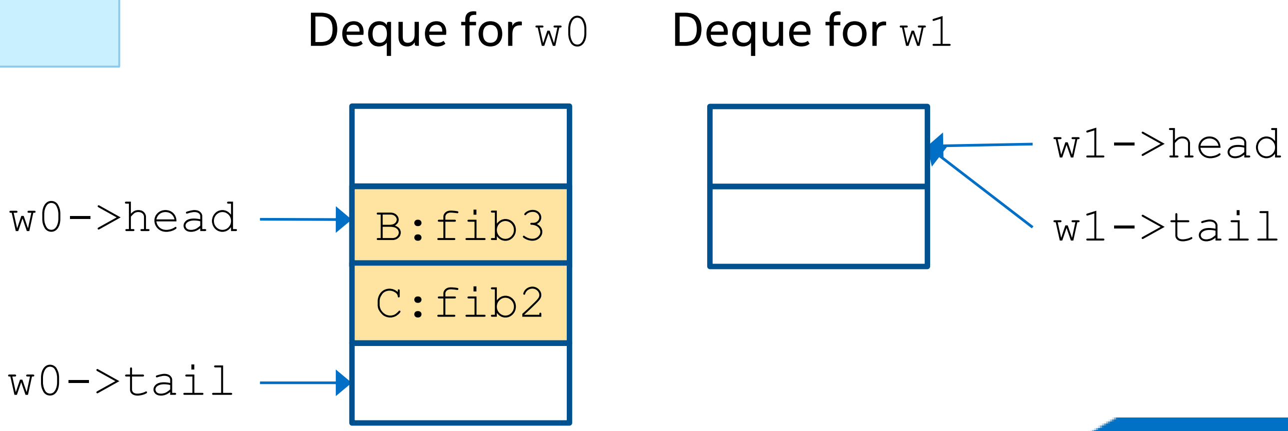
```
int fib(int n)
{
    if (n < 2)
        return n;
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

- Worker `w1` steals the continuation `A:fib4` by removing it from `w0`’s deque.

Current frame:

`w0:   D:fib1`

`w1:   A:fib4`



Deque for `w0`

Deque for `w1`

`w0->head`

`w0->tail`

`B:fib3`

`C:fib2`

`w1->head`

`w1->tail`

Deques: Steal

Another worker `w1` can steal the work from the top of `w0`’s deque.

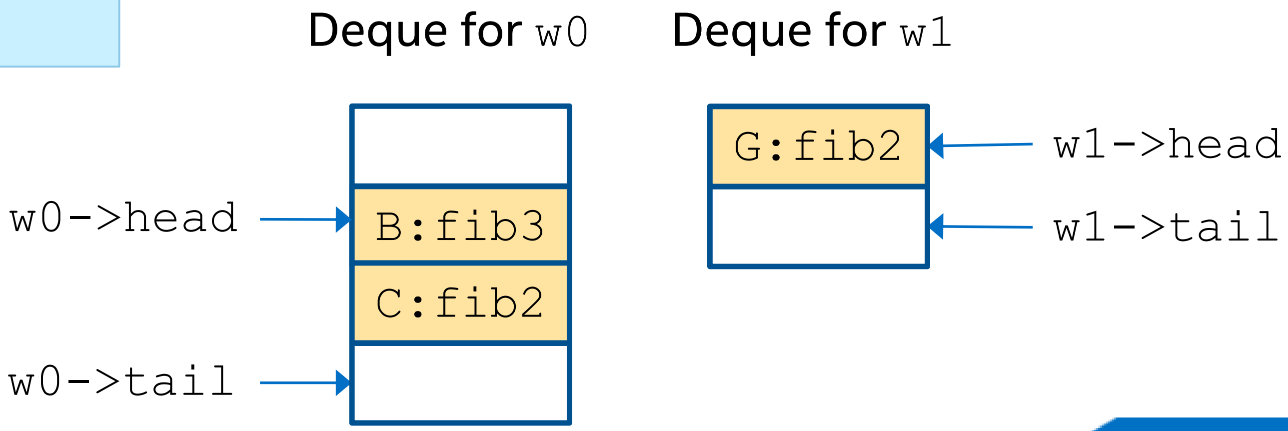
```
int fib(int n)
{
    if (n < 2)
        return n;
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

- Worker `w1` steals the continuation in `A:fib4` by removing it from `w0`’s deque.
- It then executes `G:fib2`, pushing additional frames onto its deque on subsequent `cilk_spawn` statements.

Current frame:

`w0:   D:fib1`

`w1:   H:fib1`



Deque for `w0`

`w0->head`
`B:fib3`
`C:fib2`
`w0->tail`

Deque for `w1`

`G:fib2`
`w1->head`
`w1->tail`

Deques: Return from Spawn [Fast]

When worker `w0` returns from a spawn, it checks to see if the continuation of that spawn has been stolen.

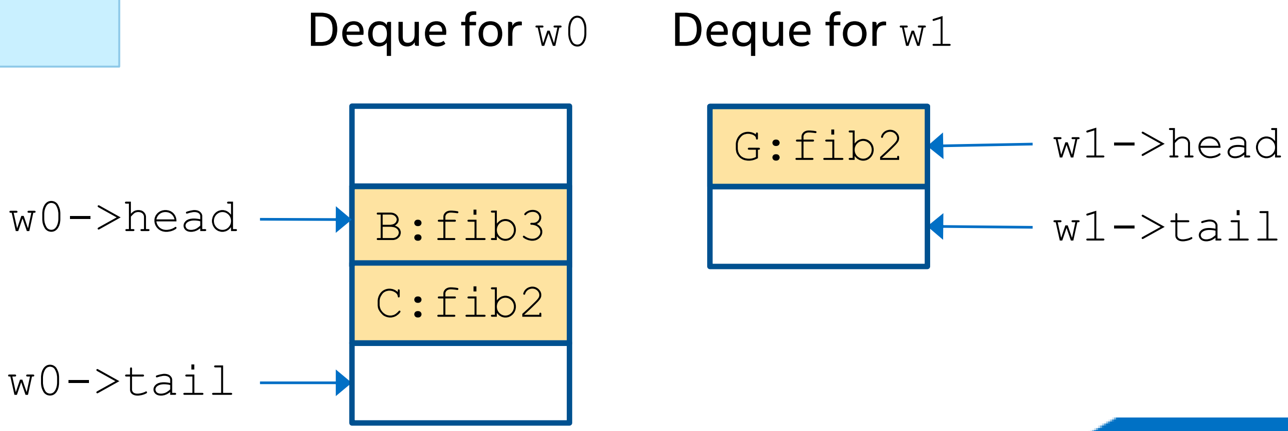
```
int fib(int n)
{
    if (n < 2)
        return n;
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

- If the continuation has not been stolen, `w0` pops the tail frame off its deque and keeps executing.

Current frame:

`w0:   D:fib1`

`w1:   H:fib1`



Deque for `w0`

`w0->head` → `B:fib3`

`C:fib2`

`w0->tail` →

Deque for `w1`

`G:fib2` ← `w1->head`

← `w1->tail`

Deques: Return from Spawn [Fast]

When worker `w0` returns from a spawn, it checks to see if the continuation of that spawn has been stolen.

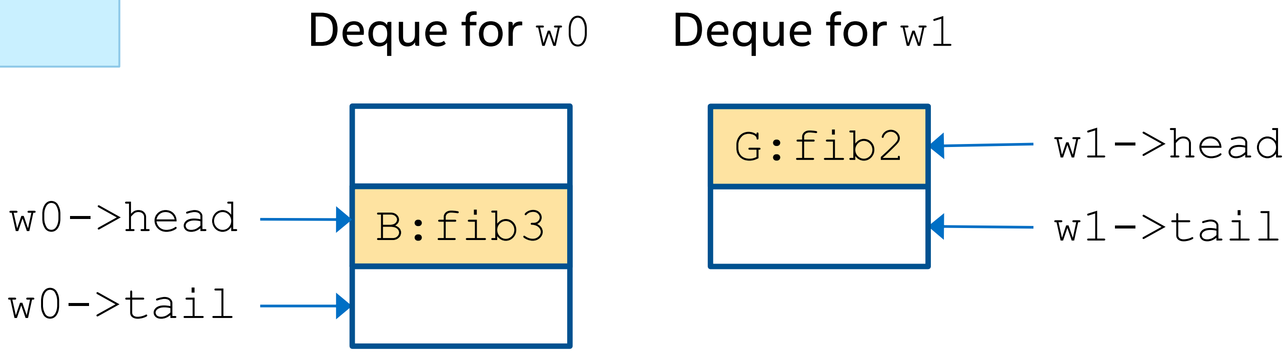
```
int fib(int n)
{
    if (n < 2)
        return n;
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

- If the continuation has not been stolen, `w0` pops the tail frame off its deque and keeps executing.

Current frame:

`w0:   C:fib2`

`w1:   H:fib1`



Deque for `w0`

Deque for `w1`

`w0->head` → `B:fib3`

`w0->tail`

`G:fib2` ← `w1->head`

`w1->tail`

Deques: `cilk_sync` [Fast]

Similarly, there is a fast path when worker `w1` reaches a **`cilk_sync`**.

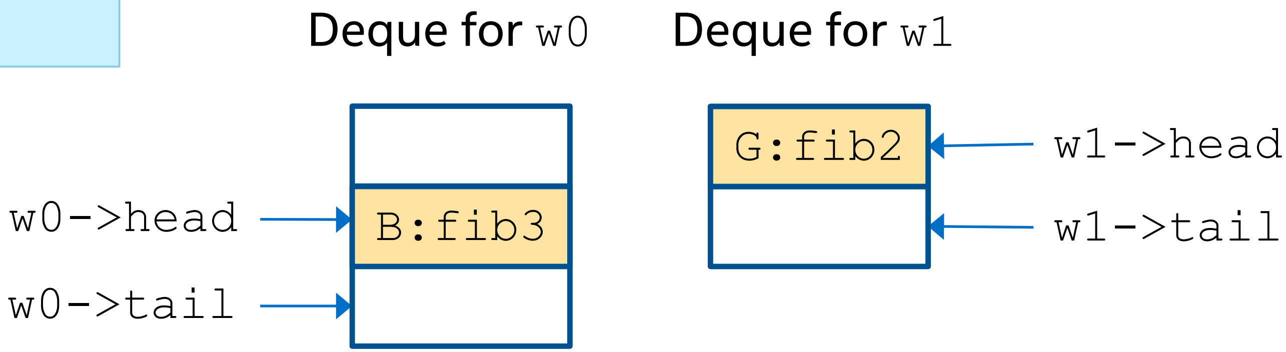
```
int fib(int n)
{
    if (n < 2)
        return n;
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

- For a `cilk_sync` inside a function that has never been stolen from, (e.g., `C:fib2` on `w0`), execution continues normally.

Current frame:

`w0:` **`C:fib2`**

`w1: H:fib1`



Deque for `w0`

`w0->head`

`B:fib3`

`w0->tail`

Deque for `w1`

`G:fib2`

`w1->head`

`w1->tail`

Deques: `cilk_sync` [Fast]

Similarly, there is a fast path when worker `w1` reaches a **`cilk_sync`**.

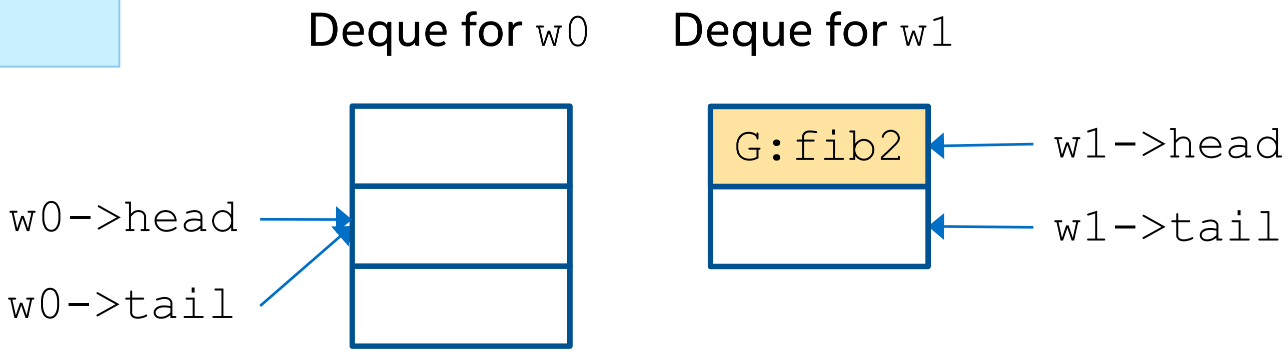
```
int fib(int n)
{
    if (n < 2)
        return n;
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

- For a `cilk_sync` inside a function that has never been stolen from, (e.g., `C:fib2` on `w0`), execution continues normally.

Current frame:

`w0:` **`B:fib3`**

`w1: H:fib1`



Deques: `cilk_sync` [Fast]

Similarly, there is a fast path when worker `w1` reaches a **`cilk_sync`**.

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

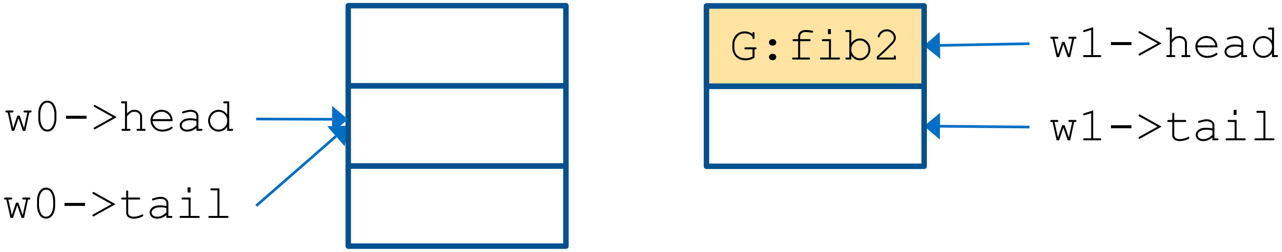
- For a `cilk_sync` inside a function that has never been stolen from, (e.g., `C:fib2` on `w0`), execution continues normally.

Current frame:

`w0:` **`F:fib1`**

`w1: H:fib1`

Deque for `w0` Deque for `w1`



Deques: Return from Spawn [Slow]

When worker `w0` returns from a spawn, it checks to see if the continuation of that spawn has been stolen.

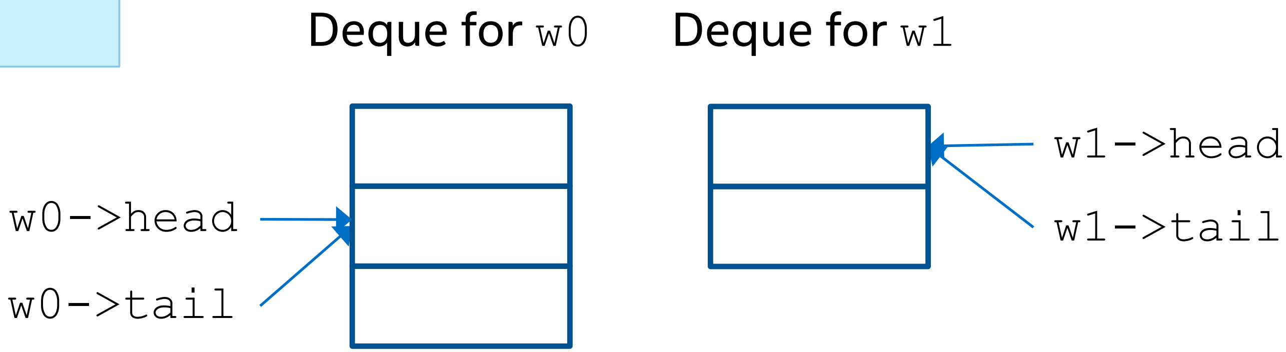
```
int fib(int n)
{
    if (n < 2)
        return n;
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

- If the continuation has not been stolen, `w0` pops the tail frame off its deque and keeps executing.
- If the continuation (e.g., of `A:fib4`) has been stolen, then `w0` has an empty deque. Then `w0` transfers control into the runtime.

Current frame:

`w0:   B:fib3`

`w1:   G:fib2`



Deques: `cilk_sync` [Slow]

Similarly, a worker can stall a **`cilk_sync`** only if its deque is empty.

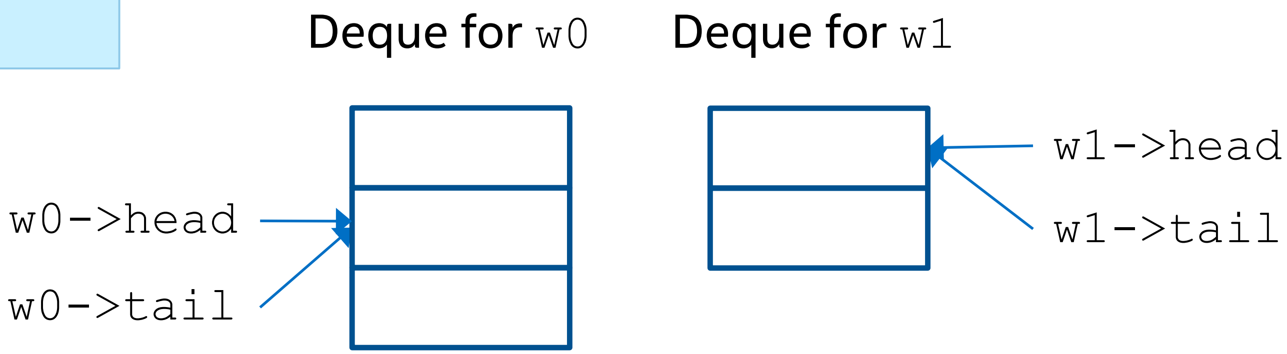
```
int fib(int n)
{
    if (n < 2)
        return n;
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

- If a steal has occurred in a function (e.g., `A:fib4`), then, execution may stall at the `cilk_sync`. In this case, control transfers to the runtime.

Current frame:

`w0: B:fib3`

`w1: A:fib4`

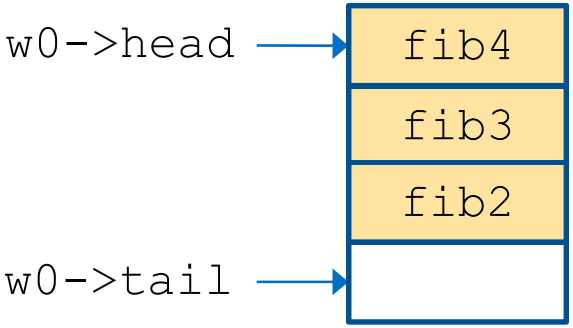


Deques: Invariants

Invariant 1: The order of stack frames (from head to tail) on any deque matches the nesting of functions on the worker's call stack (from shallowest to deepest nesting).

Invariant 2: Whenever a worker executing a function `f` must stall (at a return from `cilk_spawn` or a `cilk_sync`)

1. A steal of a continuation in `f` must have occurred, and
2. The worker's deque must be empty.



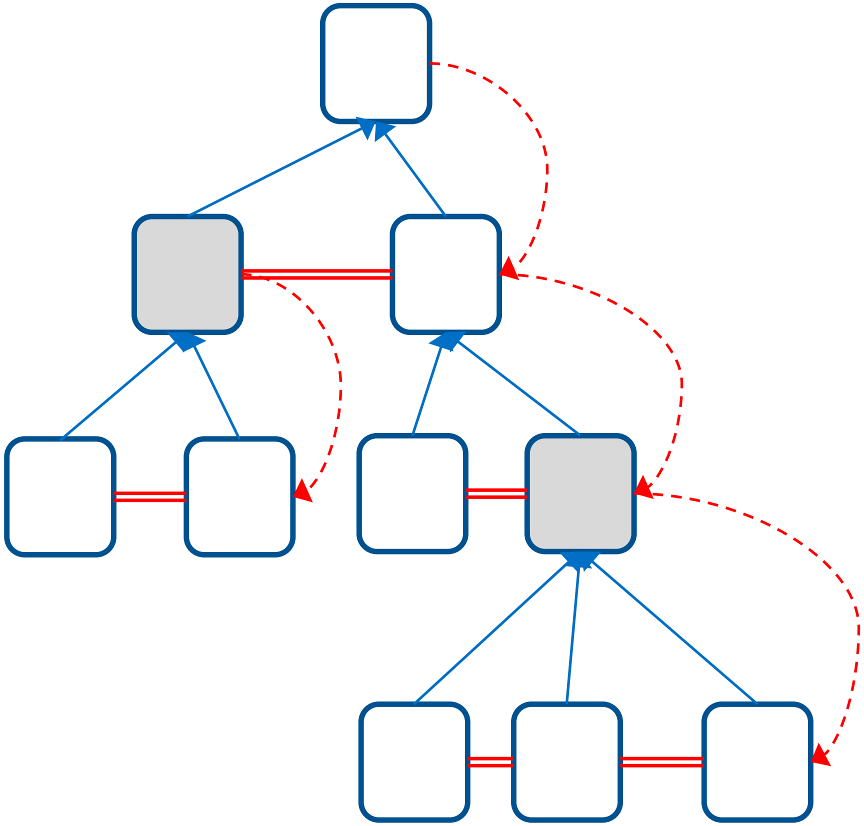
From invariant 2, (and the previous example), we see that deques can not be the only data structures keeping track of which frames to execute. Suspended frames are not on any deque!

Outline

- Deques and work-stealing
- Full frames in the Cilk Plus runtime
- Compiling a spawning function
- Stealing work
- Reducers
- Other runtime features

Runtime Data Structures: Full Frames

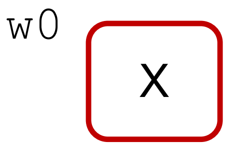
The Cilk Plus runtime mainly works with **full frames**, instead of stack frames.



- Full frames are connected in a **full frame tree**, with the parent-child relationship in the tree roughly corresponding to the parent-child relationship of stack frames.
- Each full frame is connected to its parent, rightmost child, left sibling, and right sibling.
- A frame can either be:
  - **Active**: corresponding to a worker that is executing, or it
  - **Suspended**: corresponding to a suspended stack frame.

Full Frames: Program Start

When a program starts executing on worker `w0`, the full frame tree has a single root frame.

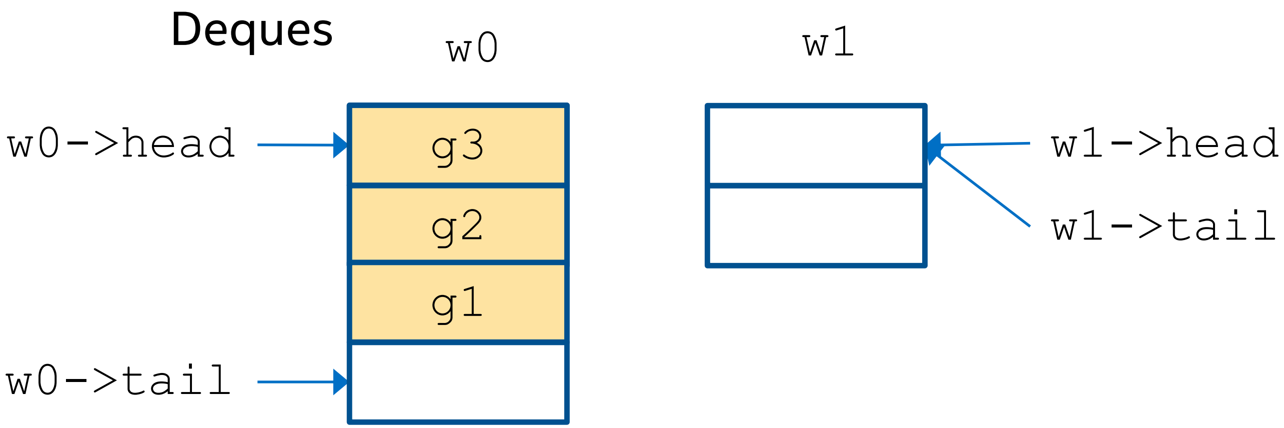


As it executes, `w0` may push new stack frames onto its deque, but it does not create any new full frames for itself.

Current frame:

`w0:  g0`

`w1:  ---`

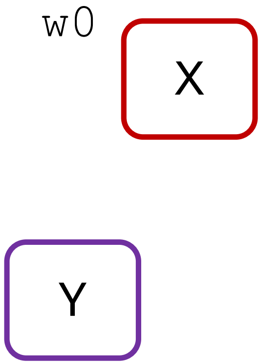


Full Frames: Successful Steal

A successful steal from by a thief worker `w1` on a victim `w0` will add one or more new full frames to the tree.

Example: `w1` steals g3 from `w0`

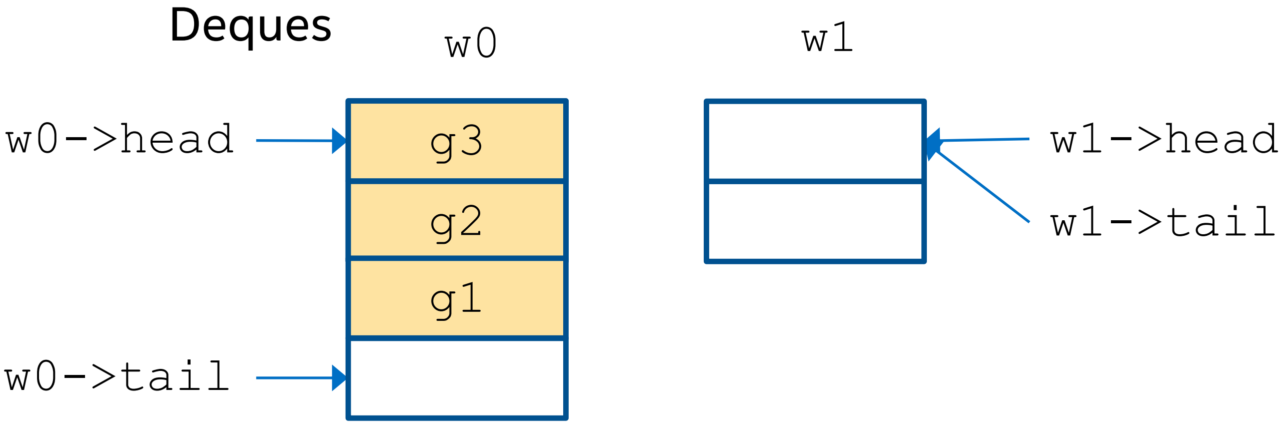
1. First, `w1` creates a new full frame Y for `w0`.



Current frame:

`w0:   g0`

`w1:   ---`

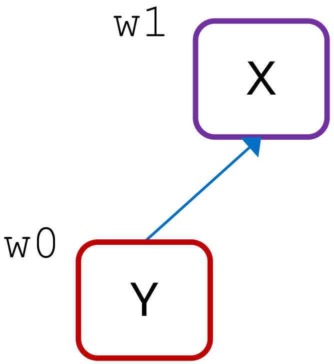


Full Frames: Successful Steal

A successful steal from by a thief worker `w1` on a victim `w0` will add one or more new full frames to the tree.

Example: `w1` steals g3 from `w0`

1. First, `w1` creates a new full frame Y for `w0`.
2. Then, `w1` steals X from `w0`.



`w1`

X

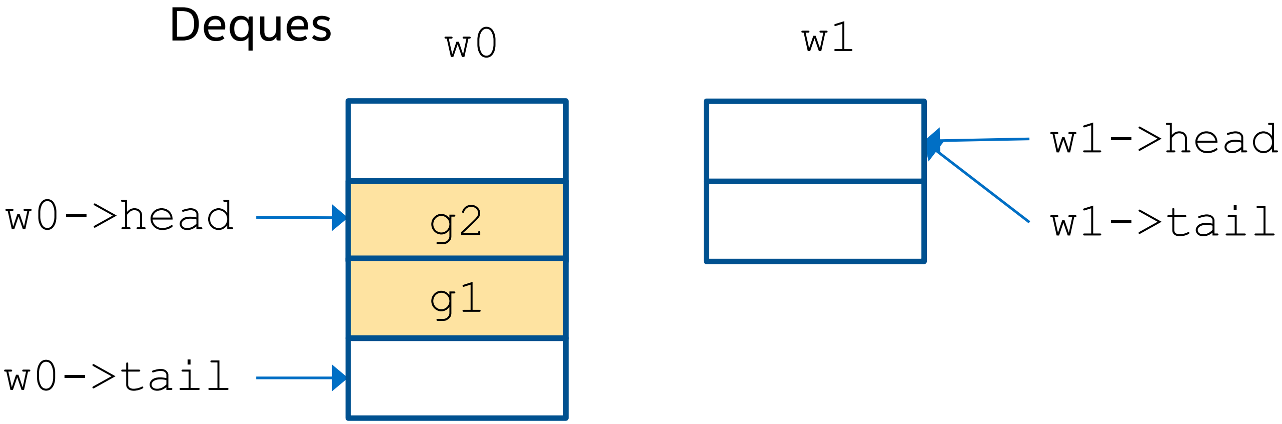
`w0`

Y

Current frame:

`w0:` `g0`

`w1:` **`g3`**



Deques

`w0`

`w1`

`w0->head`

`g2`

`g1`

`w0->tail`

`w1->head`

`w1->tail`

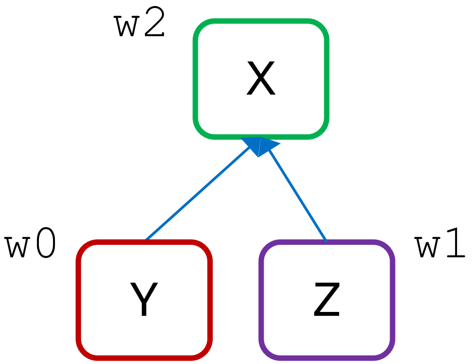
Full Frames: Successful Steal

More steals will add more full frames.

Example: `w2` steals g3 from `w1`

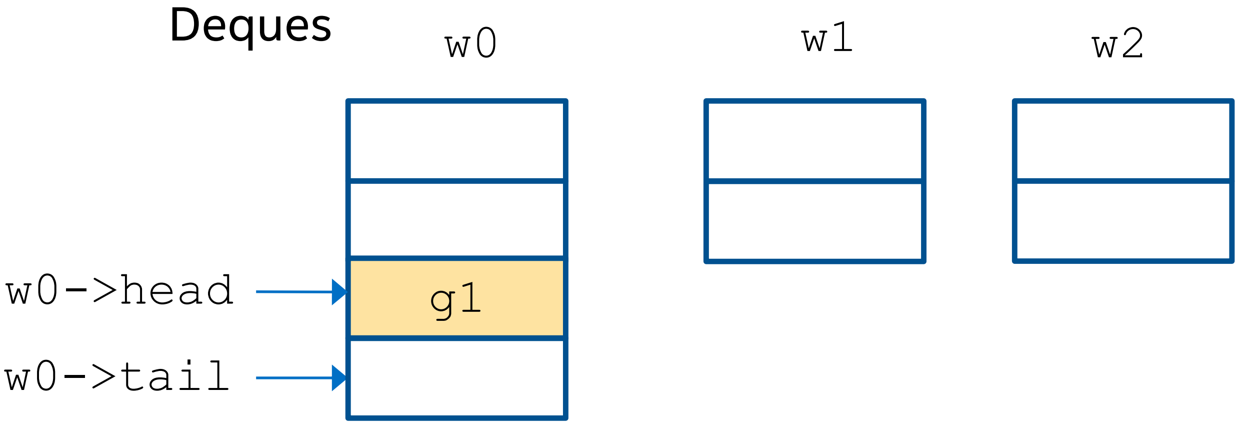
1. First, `w2` creates a new full frame Z for `w1`.
2. Then, `w2` steals X from `w1`.

Note: in the full frame tree on the right, links between siblings (e.g. Y and Z) or from parent to rightmost child (X to Z) are not shown.



Current frame:

`w0:   g0`
`w1:   g2`
`w2:   g3`

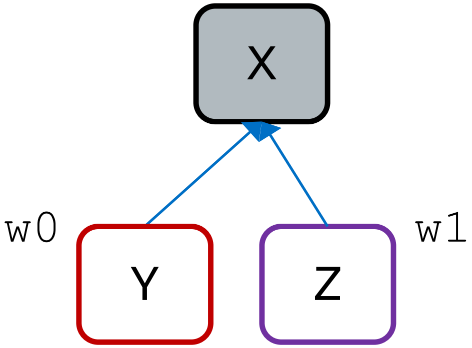


Full Frames: Suspend at `cilk_sync`

At a **`cilk_sync`**, a full frame will become suspended if execution stalls at the sync.

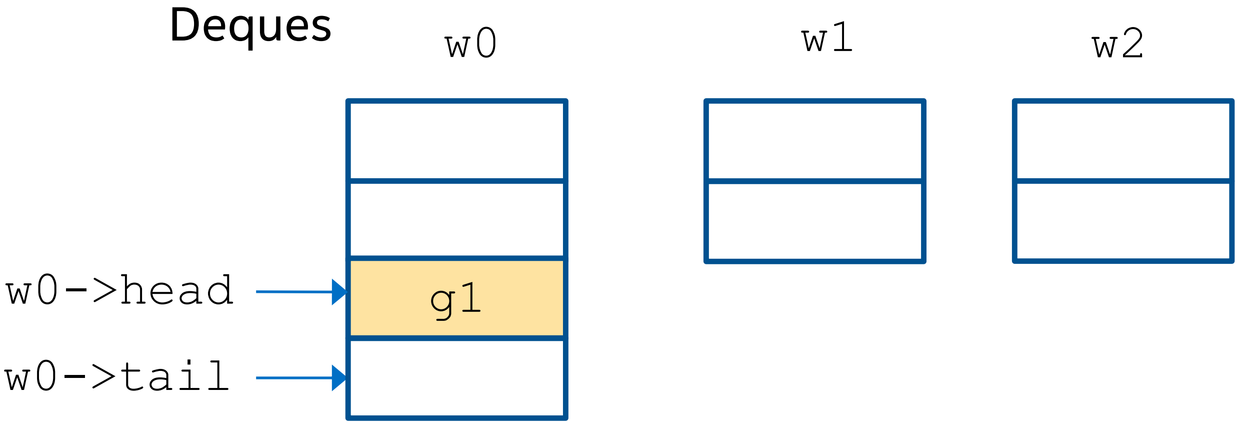
Example: `w2` stalls a `cilk_sync` in `g3`.

1. `w2` suspends frame X, and
2. `w2` jumps into the runtime to begin trying to steal work.



Current frame:

`w0:` `g0`
`w1:` **`g2`**
`w2:` **`g3`**

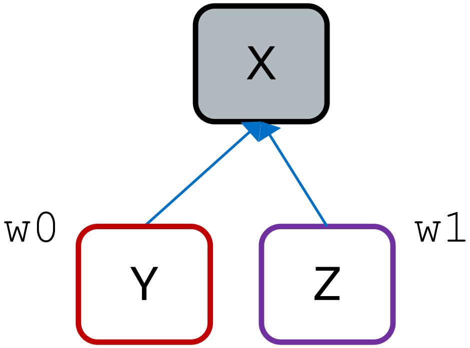


Full Frames: Return from a Spawn

A return from a spawn will end up splicing a full frame out of the tree.

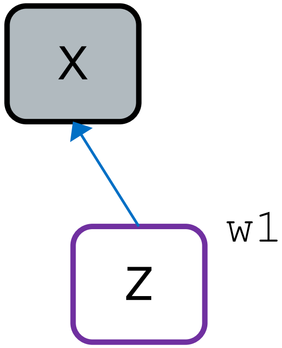
Example: `w0` returns from the spawn of `g1`.

`w0` finishes Y, and removes it from the tree.



Full Frames: Provably Good Steal

When a worker completes the last child of a frame, it will do a provably good "steal" to resume its parent if it is suspended.

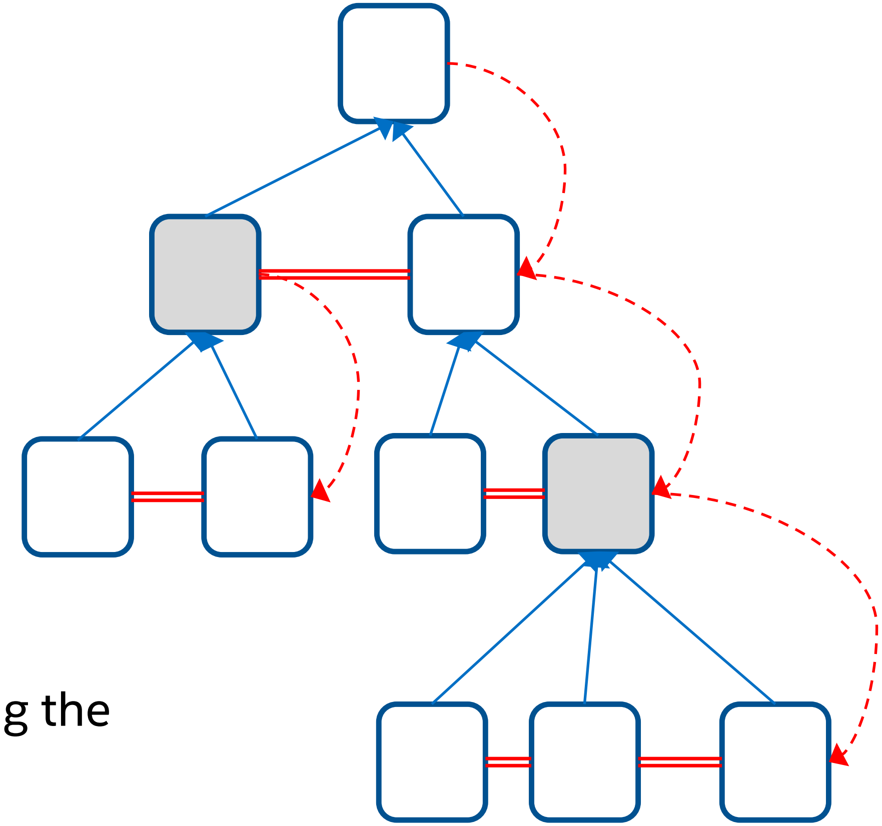


Example: when `w1` finishes Z, it will resume X.

This operation is called a "provably good steal" in part because it ensures that the theorem bounding the completion time of Cilk programs [BL99] holds. If a worker `w1` chooses to execute other work instead of X even though X is ready to resume, then the theorem no longer holds.

Invariants of the Full Frame Tree

The full frame tree maintains several invariants:



- A frame can either be:
  - **Active**: corresponding to a worker that is executing, or it
  - **Suspended**: corresponding to a suspended stack frame.
- The tree can have at most P active frames (one for each worker).
- All leaves of the tree must be active.

These invariants are important in bounding the space needed to store full frames.

Summary: Work-First Principle

Like the original MIT Cilk [FLR98], the design of Cilk Plus adopts the following principle:

> **Work-first principle:** Minimize the scheduling overhead borne by the work of a computation. Specifically, move overheads out of the work and onto the critical path.

- The runtime promotes stack frames into larger **full frames** lazily, only when a successful steal of a continuation in a function `f` occurs. This laziness moves overhead from every `cilk_spawn`, and pushes it onto each steal.
- Most of the code in the runtime library exists only to handle the case when a steal has occurred.

Outline

- Deques and work-stealing
- Full frames in the Cilk Plus runtime
- Compiling a spawning function
- Stealing work
- Reducers
- Other runtime features

Compiling a Simple Cilk Plus Function

Runtime data structures are great. But what code is the compiler really generating?

```
int f(int n)  {
    int x, y;
    x = cilk_spawn g(n);
    y = h(n);
    cilk_sync;
    return x + y;
}
```

```
int f(int n)   {
  __cilkrts_stack_frame f_sf;
  __cilkrts_worker* w = __cilkrts_get_tls_worker();
  f_sf.call_parent = w->current_stack_frame;
  f_sf.worker = w;
  w->current_stack_frame = &f_sf;

  int x, y;
  if (!CILK_SETJMP(f_sf.ctx))
    _cilk_spawn_helper_g(&x, n);

  y = h(n);

  if (f_sf.flags & CILK_FRAME_UNSYNCHED)
    if (!CILK_SETJMP(f_sf.ctx))
       __cilkrts_sync(f_sf);
  __cilkrts_pop_frame(&f_sf);
  if (f_sf.flags)
    __cilkrts_leave_frame(f_sf);
}
```

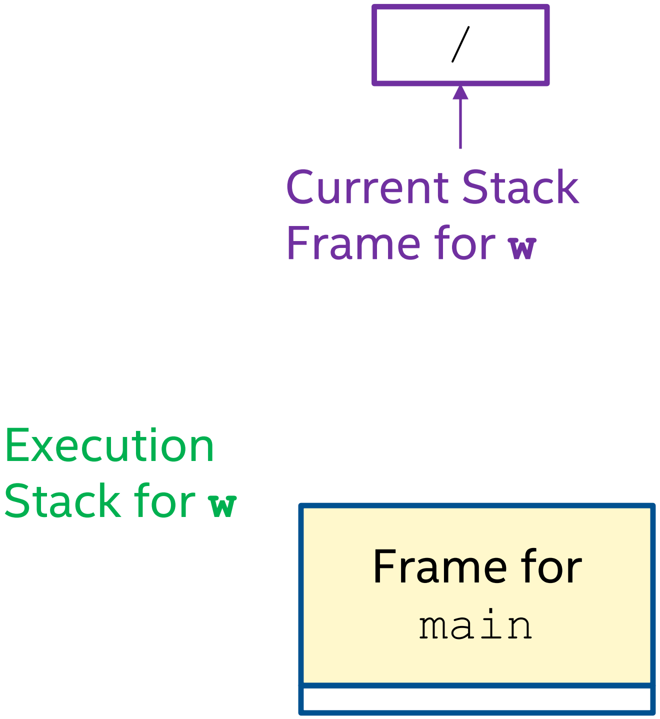
Compiler generates code for:

1. Enter/exit to a spawning function.
2. A `cilk_spawn`.
3. A `cilk_sync`.

(This example is simplified pseudocode.)

Compilation: Enter Frame

The compiler generates code to maintain a **call chain** of stack frames (of type `__cilkrts_stack_frame`):



/

Current Stack Frame for **w**

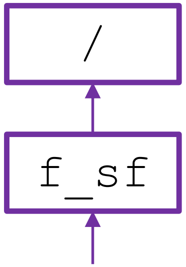
Execution Stack for **w**

Frame for `main`

```
int f(int n)   {
  __cilkrts_stack_frame f_sf;
  __cilkrts_worker* w = __cilkrts_get_tls_worker();
  f_sf.call_parent = w->current_stack_frame;
  f_sf.worker = w;
  w->current_stack_frame = &f_sf;

  int x, y;
  if (!CILK_SETJMP(f_sf.ctx))
    _cilk_spawn_helper_g(&x, n);

  y = h(n);

  if (f_sf.flags & CILK_FRAME_UNSYNCHED)
    if (!CILK_SETJMP(f_sf.ctx))
       __cilkrts_sync(f_sf);
  __cilkrts_pop_frame(&f_sf);
  if (f_sf.flags)
    __cilkrts_leave_frame(f_sf);
}
```

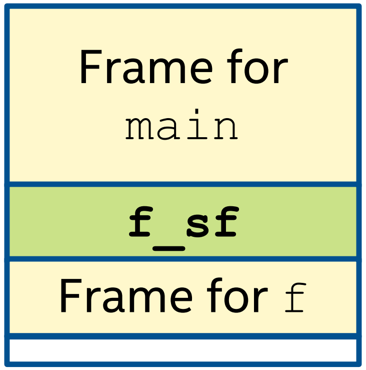
Compilation: Enter Frame

The compiler generates code to maintain a **call chain** of stack frames (of type `__cilkrts_stack_frame`):



Current Stack Frame for **`w`**

Execution Stack for **`w`**



```
int f(int n)   {
  __cilkrts_stack_frame f_sf;
  __cilkrts_worker* w = __cilkrts_get_tls_worker();
  f_sf.call_parent = w->current_stack_frame;
  f_sf.worker = w;
  w->current_stack_frame = &f_sf;

  int x, y;
  if (!CILK_SETJMP(f_sf.ctx))
    _cilk_spawn_helper_g(&x, n);

  y = h(n);

  if (f_sf.flags & CILK_FRAME_UNSYNCHED)
    if (!CILK_SETJMP(f_sf.ctx))
       __cilkrts_sync(f_sf);
  __cilkrts_pop_frame(&f_sf);
  if (f_sf.flags)
    __cilkrts_leave_frame(f_sf);
}
```

Compilation: Spawn

The compiler generates a `CILK_SETJMP` at the site of a spawn, saving state to allow another worker to steal and resume execution of the continuation.

The spawned function `g` is invoked by calling the spawn helper.

```
int f(int n)   {
  __cilkrts_stack_frame f_sf;
  __cilkrts_worker* w = __cilkrts_get_tls_worker();
  f_sf.call_parent = w->current_stack_frame;
  f_sf.worker = w;
  w->current_stack_frame = &f_sf;

  int x, y;
  if (!CILK_SETJMP(f_sf.ctx))
    _cilk_spawn_helper_g(&x, n);

  y = h(n);

  if (f_sf.flags & CILK_FRAME_UNSYNCHED)
    if (!CILK_SETJMP(f_sf.ctx))
      __cilkrts_sync(f_sf);
  __cilkrts_pop_frame(&f_sf);
  if (f_sf.flags)
    __cilkrts_leave_frame(f_sf);
}
```

Compilation: Spawn Helpers

The spawn helper for `cilk_spawn g(n)`:

1. Creates a spawn-helper frame stack frame `g_hf`, and pushes it onto the call chain.
2. Evaluates any arguments for `g`.
3. Marks `g_hf` as **detached**, and pushes its parent `f_sf` onto the deque.
4. Calls `g`.
5. Pops `g_hf` from call chain.
6. Calls `__cilkrts_leave_frame(g_hf)` to try to undo the detach and check for a stolen continuation on return from spawn.

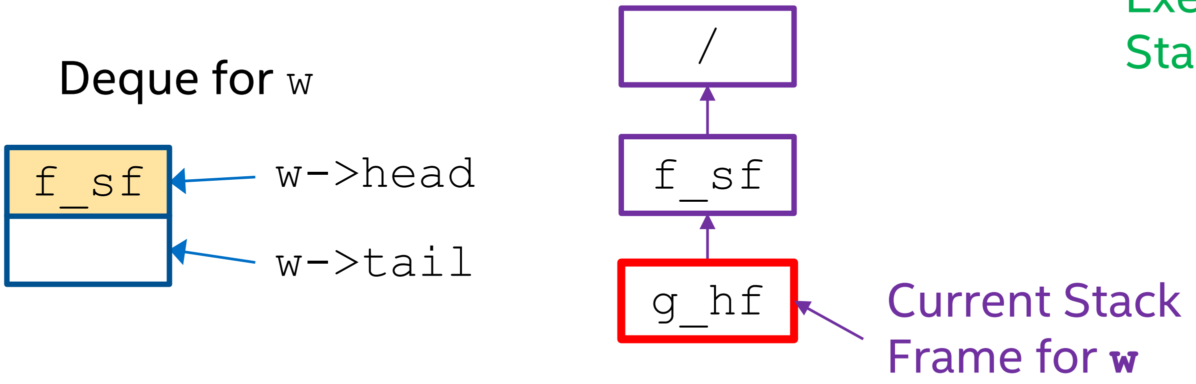
```
int f(int n)   {
  ...
  if (!CILK_SETJMP(f_sf.ctx))
    _cilk_spawn_helper_g(&x, n);
  ...
}

void cilk_spawn_helper_g(int* x, int n) {
  __cilkrts_stack_frame g_hf;
  __cilkrts_enter_frame_fast(&g_hf);

  // Evaluate arguments.
  // Nothing to do in this example.

  __cilkrts_detach();
  *x = g(n);

  __cilkrts_pop_frame(&g_hf);
  if (g_hf.flags)
    __cilkrts_leave_frame(g_hf);
}
```

More Details on `__cilkrts_leave_frame`

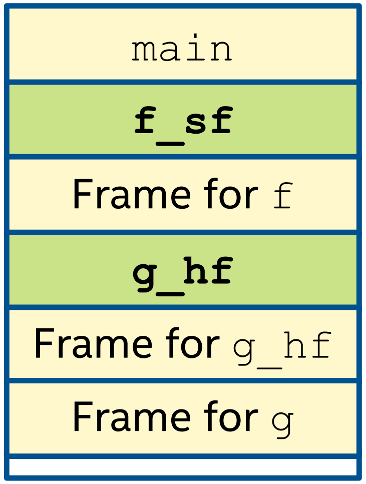
Before popping the spawn helper frame from the call chain and calling `__cilkrts_leave_frame`, the runtime state is as follows:

```
int f(int n)
{
    int x, y;
    x = cilk_spawn g(n);
    y = h(n);
    cilk_sync;
    return x + y;
}
```

- If continuation in `f` was not stolen, `__cilkrts_leave_frame` returns normally.
- **Otherwise, control jumps into the runtime library.**



Execution Stack for **w**



Calls, spawns, and spawning functions

Why on earth do we need all this complexity anyway?

- A function can invoked in two ways: called, or spawned.
- A function can be a spawning function, or a (normal) nonspawning C/C++ function.
- **These concepts are orthogonal!**

	Calls	**Spawns**
F is nonspawning	Nonspawning or spawning G	---
F is spawning	Nonspawning or spawning G	Nonspawning or spawning G

Spawn helpers:

- Enable spawning of a nonspawning function G, in a way that avoids recompilation of G.
- Manage the lifetimes of temporaries and return values.
- Deal with the case when the argument to a spawned function itself is a function that has nested parallelism, e.g., `cilk_spawn f(fib(20))`.

Calls, spawns, and spawning functions

Consider an example program:

	Calls	Spawns
F is nonspawning	Nonspawning or spawning G	---
F is spawning	Nonspawning or spawning G	Nonspawning or spawning G

1. A nonspawning function `main` calls a spawning function `a`.
2. A spawning function `f` calls a nonspawning function `g1`.
   A spawning function `a` calls a spawning function `a`.
3. A spawning function `f` spawns a nonspawning `g0`.
   A spawning function `a` spawns a spawning function `f`.

```
int main(void) {
  a(2);
  return 0;
}
void a(int d) {
  if (d > 0)
    a(d-1);
  else {
      cilk_spawn f();
      cilk_sync;
  }
}
void f() {
    cilk_spawn g0();
    g1();
    cilk_sync;
}
```

Outline

- Deques and work-stealing
- Full frames in the Cilk Plus runtime
- Compiling a spawning function
- Stealing work
- Reducers
- Other runtime features

Stealing work: the details

We put everything together, by walking through an example of a steal on the following program:

```
int main(void) {
  a(2);
  return 0;
}
void a(int d) {
  if (d > 0)
    a(d-1);
  else {
     cilk_spawn f();
     cilk_sync;
  }
}
void f() {
   cilk_spawn g0();
   g1();
   cilk_sync;
}
```

Recall, that Cilk Plus maintains the following data structures:

- Every worker has a deque to store stack frames that can be stolen.
- The compiler maintains a call chain of stack frames, to track the currently executing stack frame.
- The runtime maintains a tree with full frames, to track suspended frames and other runtime state.

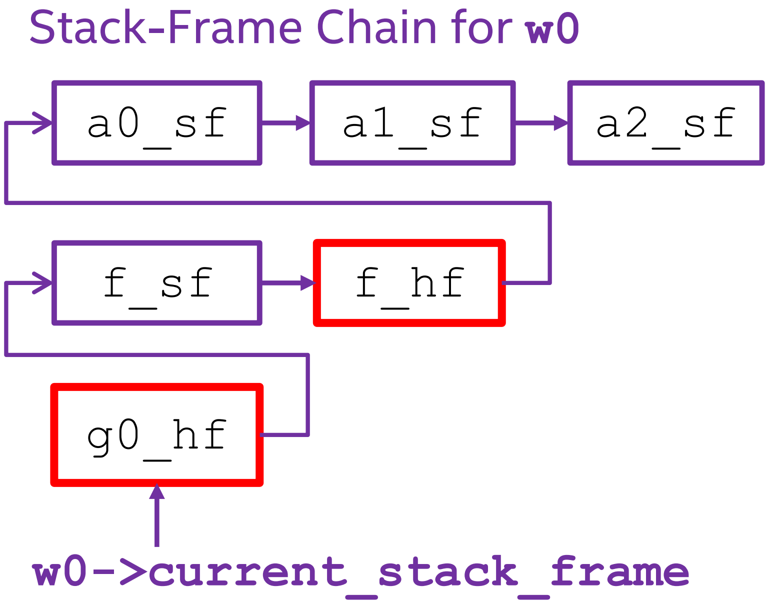
Steals and Call Chains

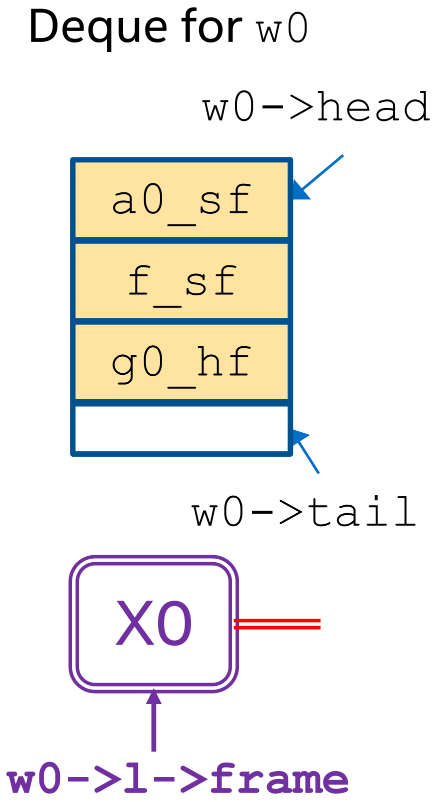
Calls and spawns affect the layout of deques and stack-frame chains.

Let `ai` denote the instance of `a(i)`

```
int main(void) {
  a(2);
  return 0;
}
void a(int d) {
  if (d > 0)
    a(d-1);
  else {
     cilk_spawn f();
     cilk_sync;
  }
}
void f() {
   cilk_spawn g0();
   g1();
   cilk_sync;
}
```

A steal of stack frame from a deque should steal a chain of stack frames, not just one stack frame!





Stealing Work, Step 1

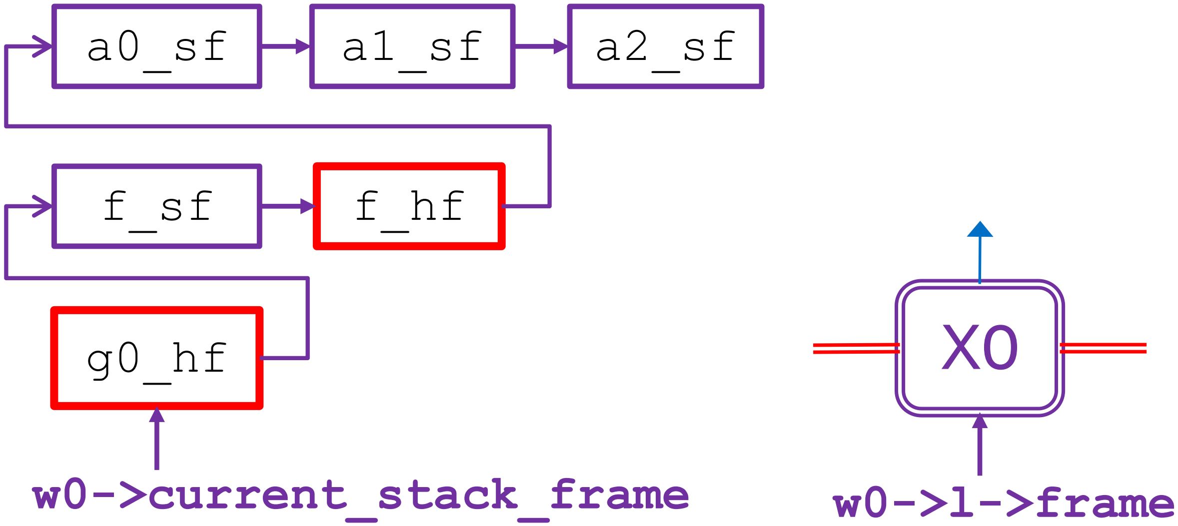
When a worker **`w1`** steals a call chain from **`w0`**, it promotes stack frames into full frames and links them in the full frame tree.

```
int main(void) {
  a(2);
  return 0;
}
void a(int d) {
  if (d > 0)
    a(d-1);
  else {
     cilk_spawn f();
     cilk_spawn f();
     cilk_sync;
  }
}
void f() {
   cilk_spawn g0();
   g1();
   cilk_sync;
}
```

To steal, `w1` tries to detach X0 from victim `w0`.

1. Pops `a0_sf` the from the top of `w0`’s deque.

Stack-Frame Chain for **`w0`**



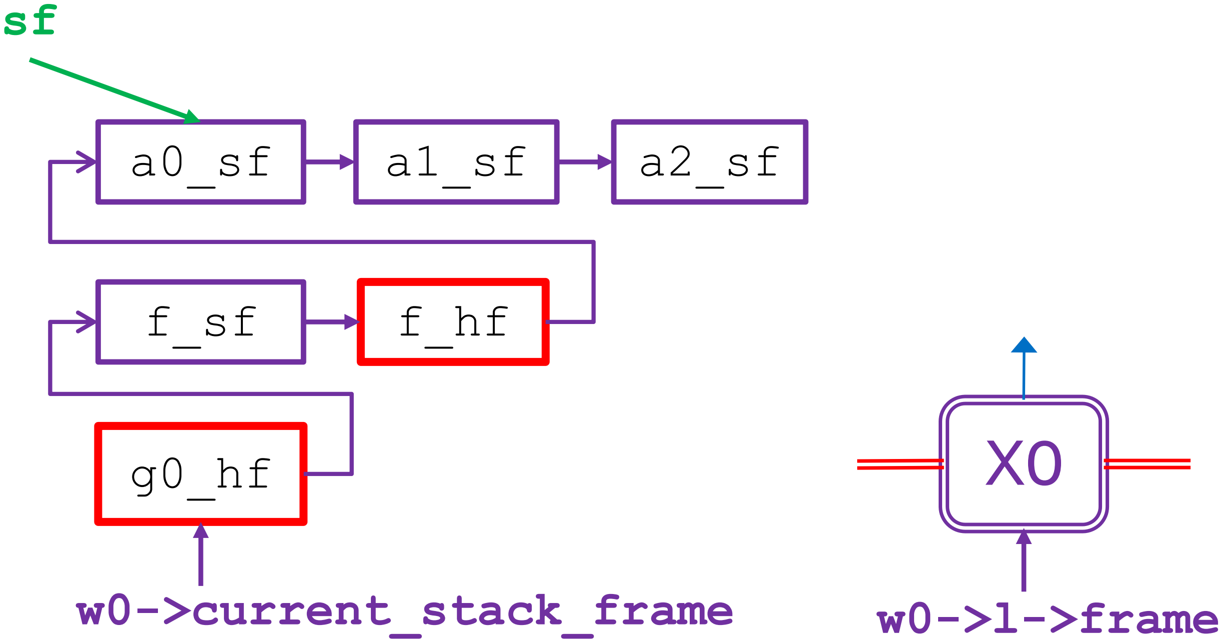
Stealing Work, Step 2

When a worker **`w1`** steals a call chain from **`w0`**, it promotes stack frames into full frames and links them in the full frame tree.

```
int main(void) {
  a(2);
  return 0;
}
void a(int d) {
  if (d > 0)
    a(d-1);
  else {
     cilk_spawn f();
     cilk_sync;
  }
}
void f() {
    cilk_spawn g0();
    g1();
    cilk_sync;
}
```

To steal, `w1` tries to detach X0 from victim `w0`.

1. Pops `a0_sf` the from the top of `w0`'s deque.
2. Calls `unroll_call_stack(w0, X0, a0_sf);`
   a) Reverse call chain



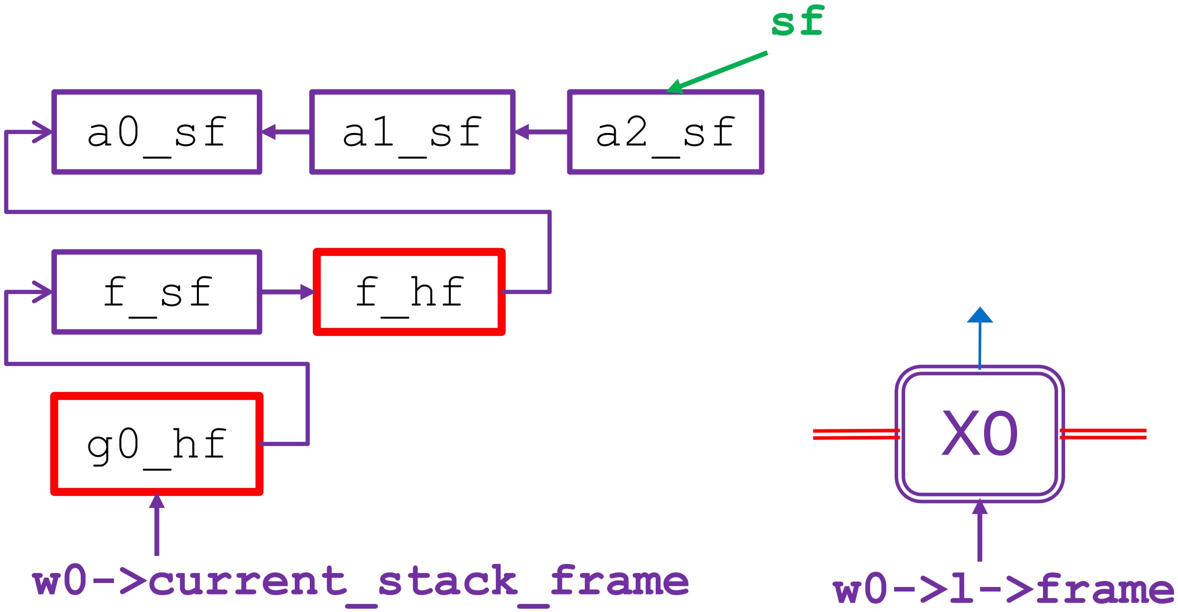
Stealing Work, Step 2(a)

When a worker **`w1`** steals a call chain from **`w0`**, it promotes stack frames into full frames and links them in the full frame tree.

```
int main(void) {
  a(2);
  return 0;
}
void a(int d) {
  if (d > 0)
    a(d-1);
  else {
     cilk_spawn f();
     cilk_sync;
  }
}
void f() {
    cilk_spawn g0();
    g1();
    cilk_sync;
}
```

To steal, `w1` tries to detach X0 from victim `w0`.

1. Pops `a0_sf` the from the top of `w0`'s deque.
2. Calls `unroll_call_stack(w0, X0, a0_sf);`
   a) Reverse call chain



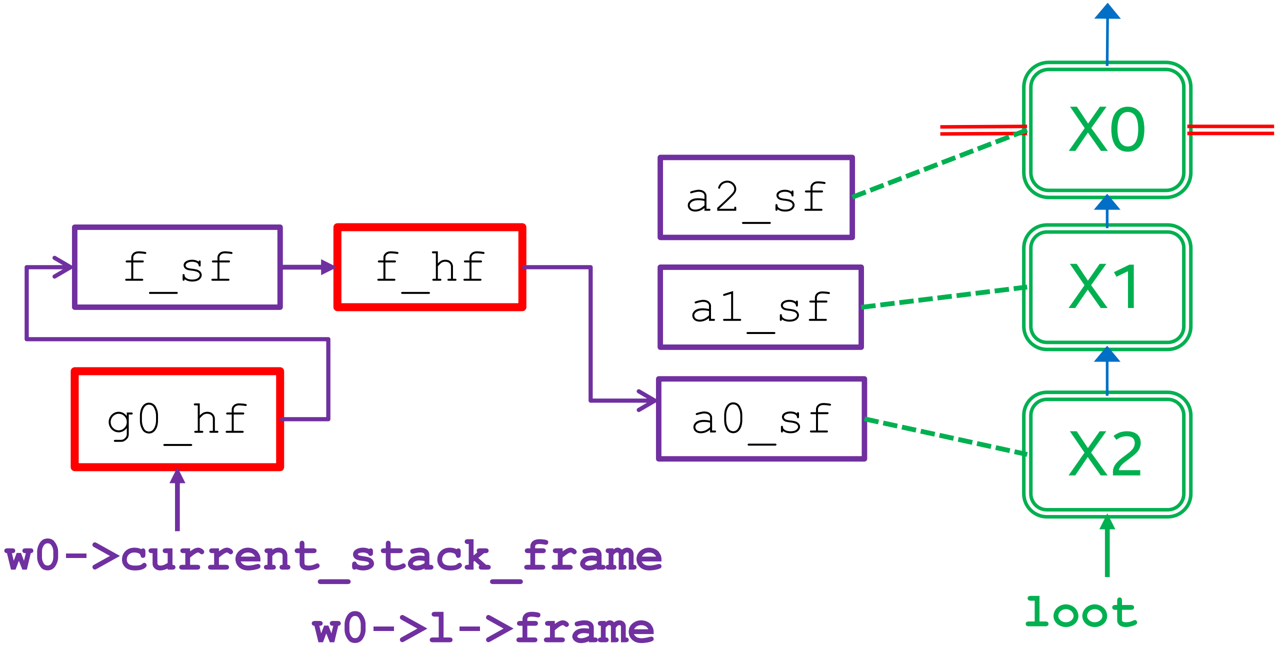
Stealing Work, Step 2(b)

When a worker **`w1`** steals a call chain from **`w0`**, it promotes stack frames into full frames and links them in the full frame tree.

```
int main(void) {
  a(2);
  return 0;
}
void a(int d) {
  if (d > 0)
    a(d-1);
  else {
    cilk_spawn f();
    cilk_sync;
  }
}
void f() {
   cilk_spawn g0();
   g1();
   cilk_sync;
}
```

To steal, `w1` tries to detach X0 from victim `w0`.

1. Pops `a0_sf` the from the top of `w0`'s deque.
2. Calls `unroll_call_stack(w0, X0, a0_sf);`
   - a) Reverse call chain
   - b) Promote stack frames to full frames



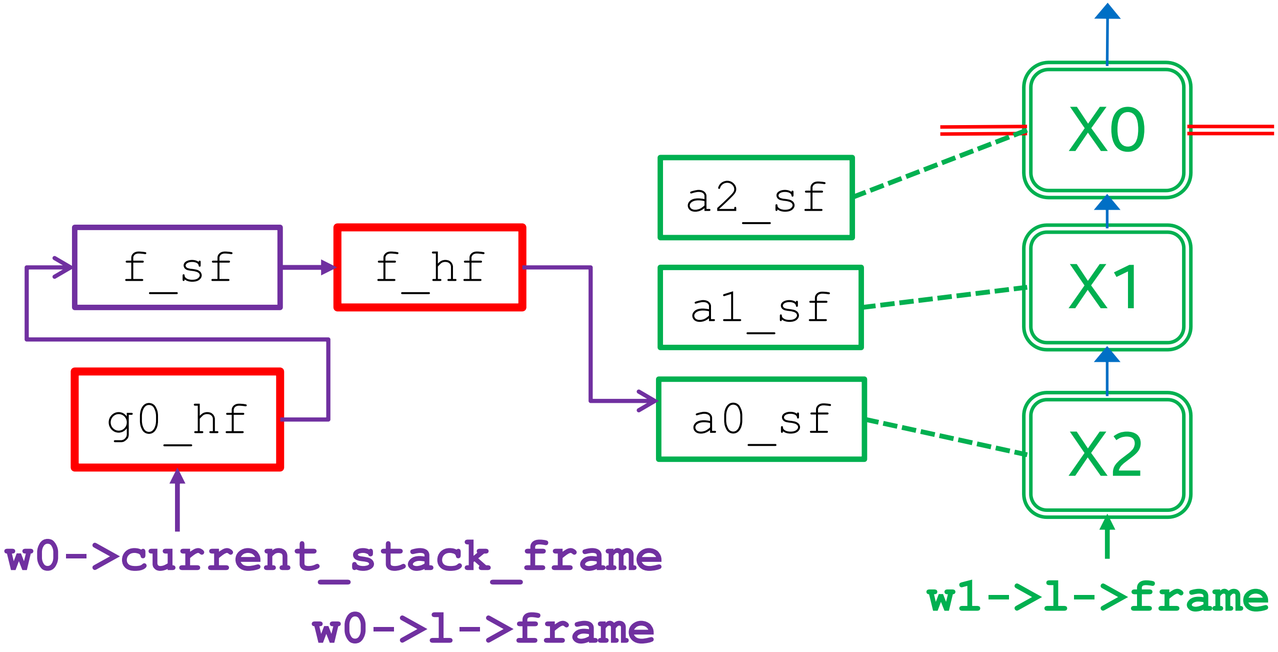
Stealing Work, Step 3

When a worker **`w1`** steals a call chain from **`w0`**, it promotes stack frames into full frames and links them in the full frame tree.

```
int main(void) {
  a(2);
  return 0;
}
void a(int d) {
  if (d > 0)
    a(d-1);
  else {
     cilk_spawn f();
     cilk_sync;
  }
}
void f() {
    cilk_spawn g0();
    g1();
    cilk_sync;
}
```

To steal, `w1` tries to detach X0 from victim `w0`.

1. Pops `a0_sf` the from the top of `w0`’s deque.
2. Calls `unroll_call_stack(w0, X0, a0_sf);`
3. Makes `loot` its active frame.



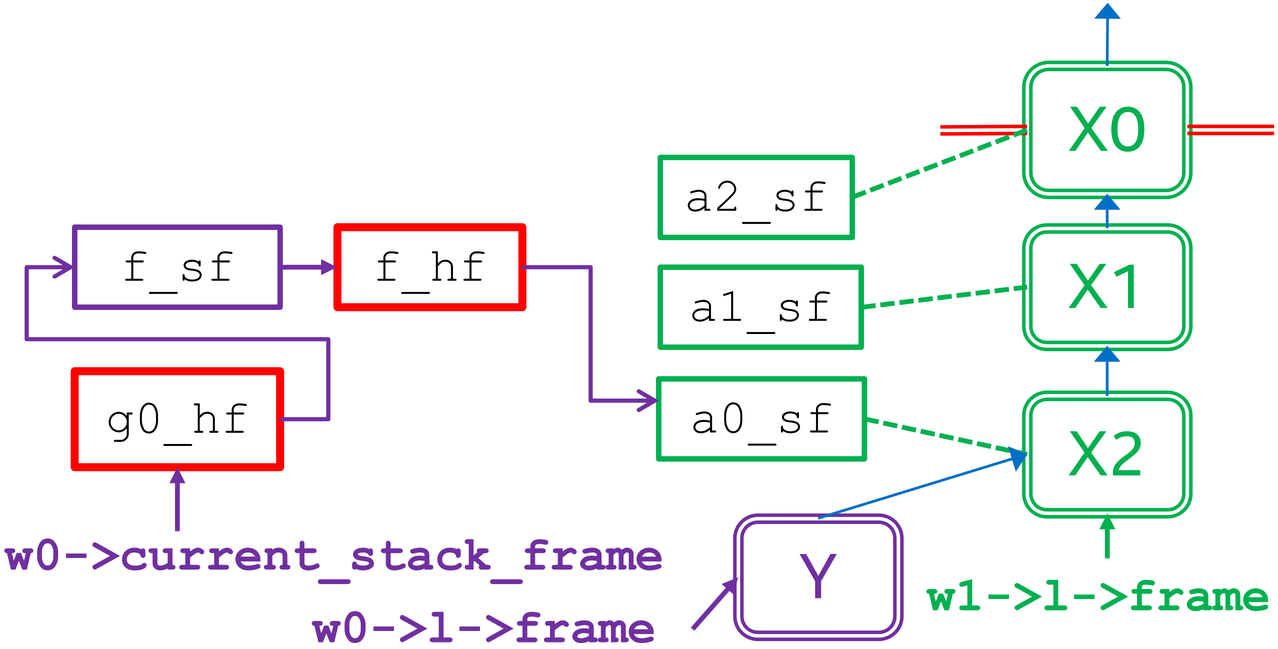
Stealing Work, Step 4

When a worker **`w1`** steals a call chain from **`w0`**, it promotes stack frames into full frames and links them in the full frame tree.

```
int main(void) {
  a(2);
  return 0;
}
void a(int d) {
  if (d > 0)
    a(d-1);
  else {
    cilk_spawn f();
    cilk_sync;
  }
}
void f() {
   cilk_spawn g0();
   g1();
   cilk_sync;
}
```

To steal, `w1` tries to detach X0 from victim `w0`.

1. Pops `a0_sf` the from the top of `w0`'s deque.
2. Calls `unroll_call_stack(w0, X0, a0_sf);`
3. Makes `loot` X2 its active frame.
4. Creates a child frame Y for victim `w0`.



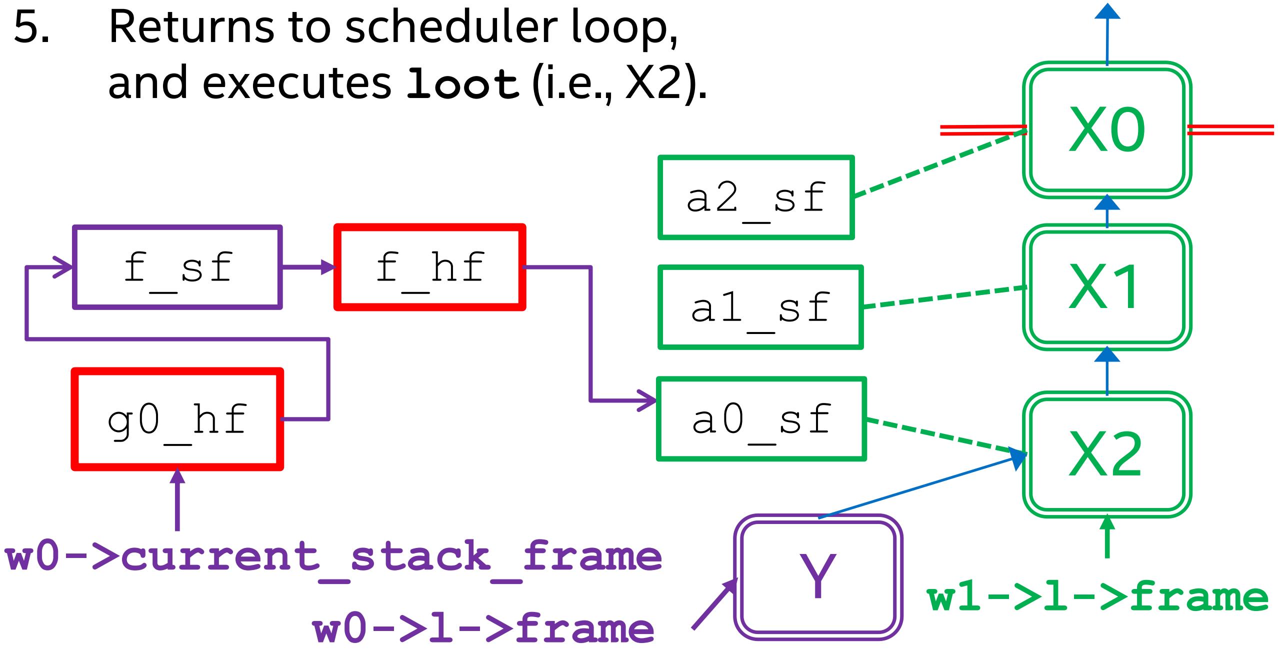
Stealing Work, Step 5

When a worker **`w1`** steals a call chain from **`w0`**, it promotes stack frames into full frames and links them in the full frame tree.

```
int main(void) {
  a(2);
  return 0;
}
void a(int d) {
  if (d > 0)
    a(d-1);
  else {
     cilk_spawn f();
     cilk_sync;
  }
}
void f() {
   cilk_spawn g0();
   g1();
   cilk_sync;
}
```

To steal, `w1` tries to detach X0 from victim `w0`.

1. Pops `a0_sf` the from the top of `w0`'s deque.
2. Calls `unroll_call_stack(w0, X0, a0_sf);`
3. Makes `loot` its active frame.
4. Creates a child frame Y for victim `w0`.
5. Returns to scheduler loop,
   and executes **`loot`** (i.e., X2).



Outline

- Deques and work-stealing
- Full frames in the Cilk Plus runtime
- Compiling a spawning function
- Stealing work
- Reducers
- Other runtime features

Reducers

Reducers can be used to eliminate races due to associative operations on shared variables.

```
int walk(node *n)
{
    int x=0, y=0;
    if (n->left)
      x = cilk_spawn walk(n->left);
    if (n->right)
      y = cilk_spawn walk(n->right);
    int z = g(n->value);
    cilk_sync;
    return x + y + z;
}
```

Runtime automatically creates “views” as needed and “reduces” them in a lock-free manner.

```
reducer_list_append<int> L;
int g(int n) {
     L.push_back(n);
     return n;
}
```

Serial Semantics for Reducers

Runtime “reduces” views deterministically for any associative reduction operation.

```
int walk(node *n)
{
    int x=0, y=0;
    if (n->left)
      x = cilk_spawn walk(n->left);
    if (n->right)
      y = cilk_spawn walk(n->right);
    int z = g(n->value);
    cilk_sync;
    return x + y + z;
}
```

For this example, we get the same (deterministic) list as in a serial execution!

```
reducer_list_append<int> L;
int g(int n) {
    L.push_back(n);
    return n;
}
```

Reducer Library

Cilk Plus provides a library of built-in reducers:

- `reducer_list_append` / `reducer_list_prepend`
- `reducer_max` / `reducer_max_index`
  `reducer_min` / `reducer_min_index`
- `reducer_opadd`
- `reducer_opand`, `reducer_opor`, `reducer_opxor`
- `reducer_string, reducer_wstring`
- `reducer_ostream`

Reducer Maps

The runtime implements reducers by maintaining reducer maps (objects of type `cilkred_map`).

- Each worker maintains its **active reducer map**, `w->reducer_map`, which is used while a worker is executing.
- Each access to a reducer triggers a lookup in the current worker's active reducer map to find the appropriate **view**. Some compilers may optimize and eliminate redundant reducer lookups.
- Each full frame `ff` stores two additional reducer maps, which may be accessed when neighboring full frames are completed and spliced out:
  - Child map (`ff->children_reducer_map`): may be filled by its leftmost child
  - Right map (`ff->right_reducer_map`): may be accessed by its right sibling or parent.

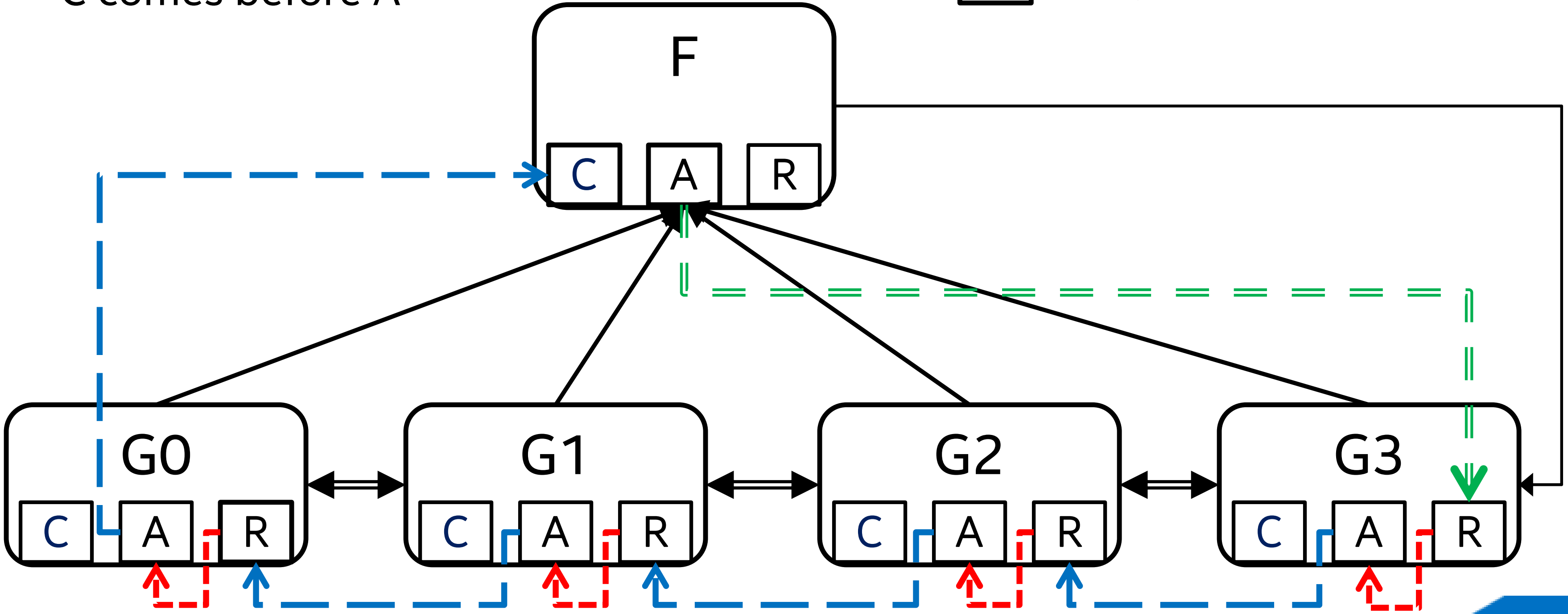
- Cilk Plus uses a simplified variant of the reducer protocol described in [FHLL09] to merge reducer maps together.

Reducer Maps

Example: Consider a full frame F with children G0, G1, G2, G3.

- Dashed arrows go from “right to left” (or later to earlier) in the serial execution order.
- For a frame which is synched, at most one of C and A is nonempty.
- For unsynched frames, (e.g., F), C comes before A

A	Active reducer map. Only used if frame is equal to `w->l->frame_ff` for some worker `w`.
C	Child reducer map.
R	Right reducer map.

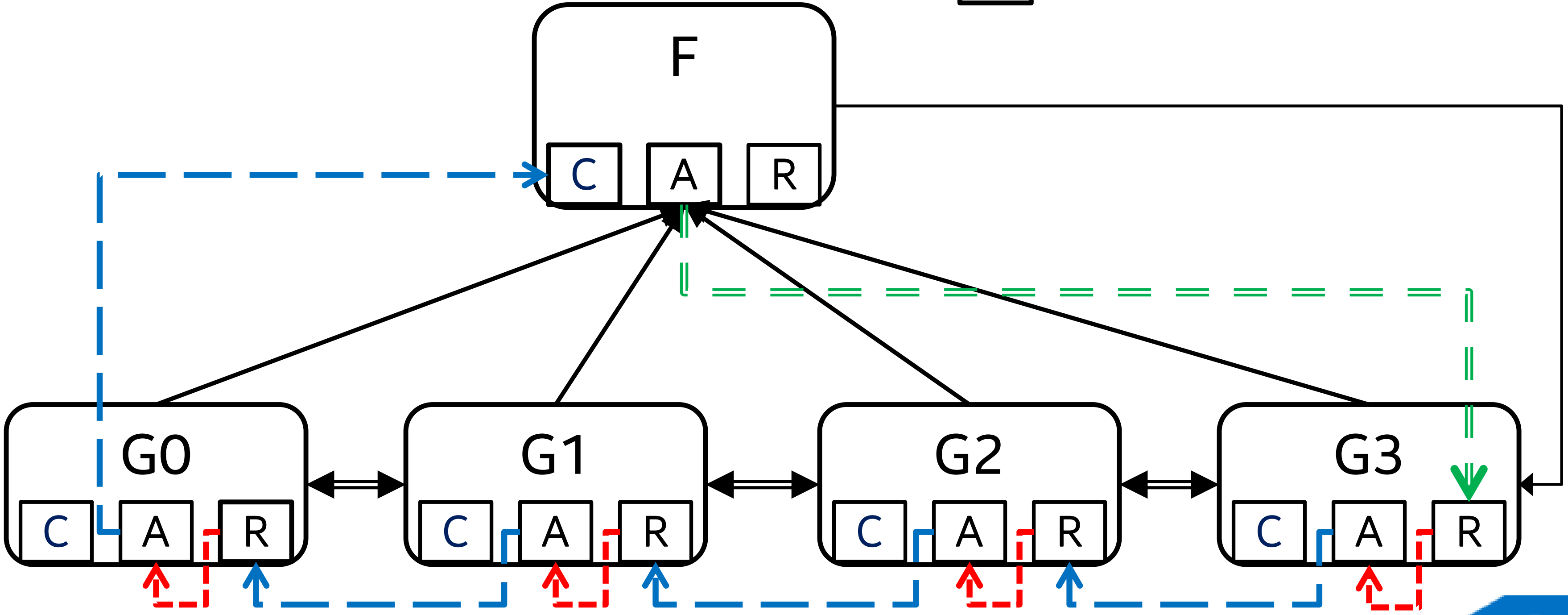


Reducer Protocol

Consider a full frame F with children G0, G1, G2, G3. Merging of reducers happens at two events:

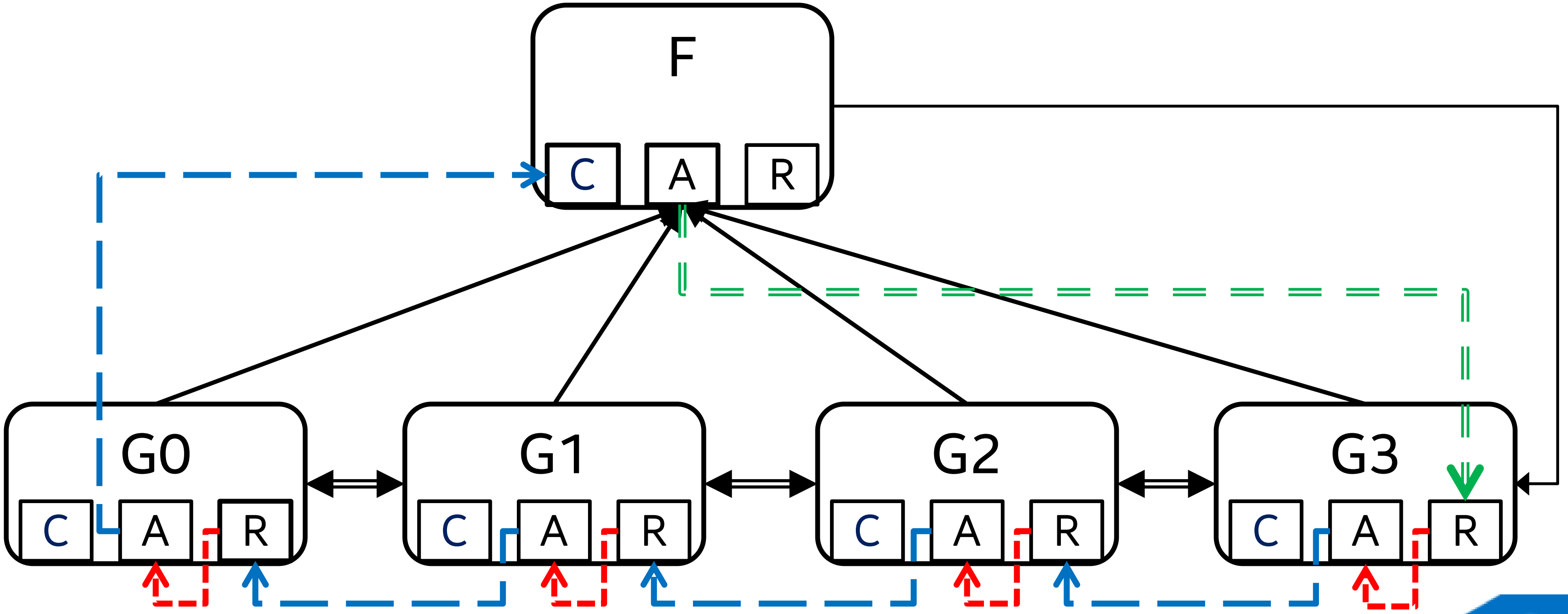
1. At a sync.
2. Return of a full frame.

A	Active reducer map. Only used if frame is equal to `w->l->frame_ff` for some worker `w`.
C	Child reducer map.
R	Right reducer map.



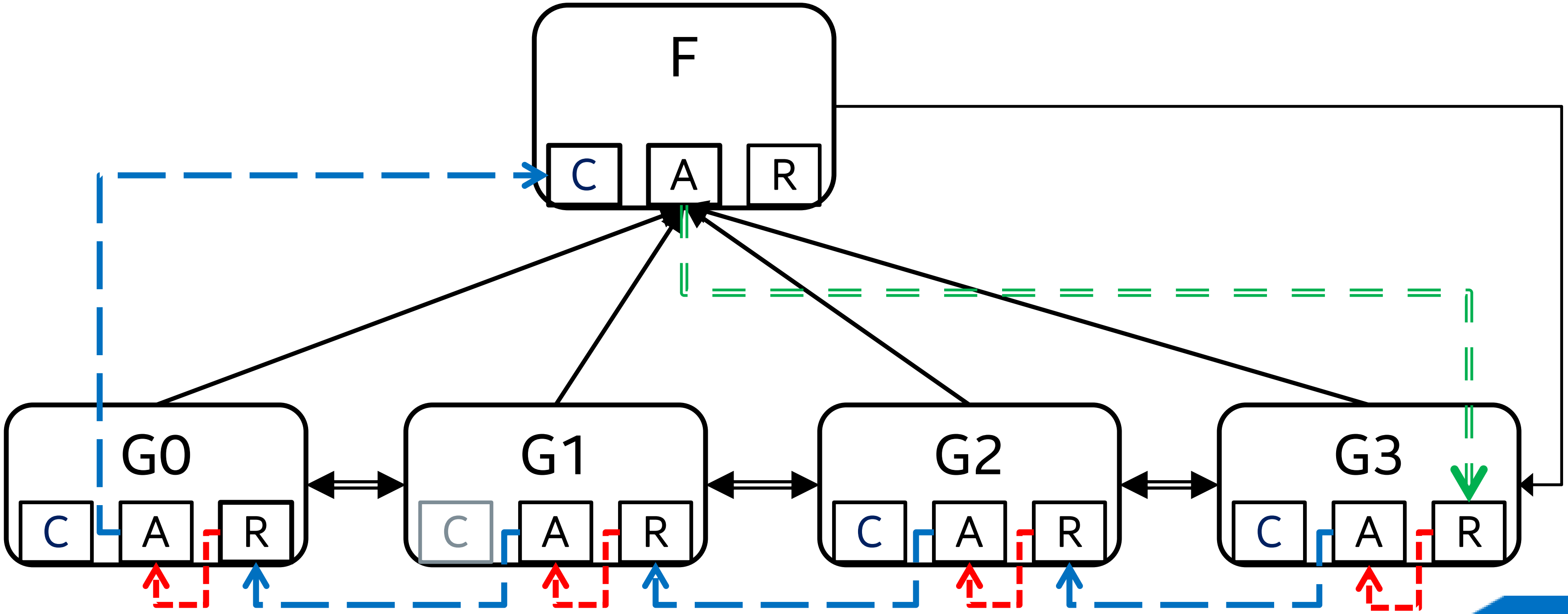
Reducer Protocol: Sync

1. At a sync (e.g., of F).
   - Remove child map C and active map A, and merge:
     X = F.C + F.A
   - Merge and deposit into rightmost child's right map:
     G3.R = G3.R + X



Reducer Protocol: Return

2. At a return of a frame (e.g., G1).
   - Child map C already merged at G1's implicit sync.
   - Remove active map A and right map R, then merge:
     X = G1.A + G1.R
   - Merge and deposit into left sibling's right map:
     G0.R = G0.R + X



Outline

- Deques and work-stealing
- Full frames in the Cilk Plus runtime
- Compiling a spawning function
- Stealing work
- Reducers
- Other runtime features

Other Runtime Features in Cilk Plus

There are several runtime features that we aren't covering here, which are part of the runtime:

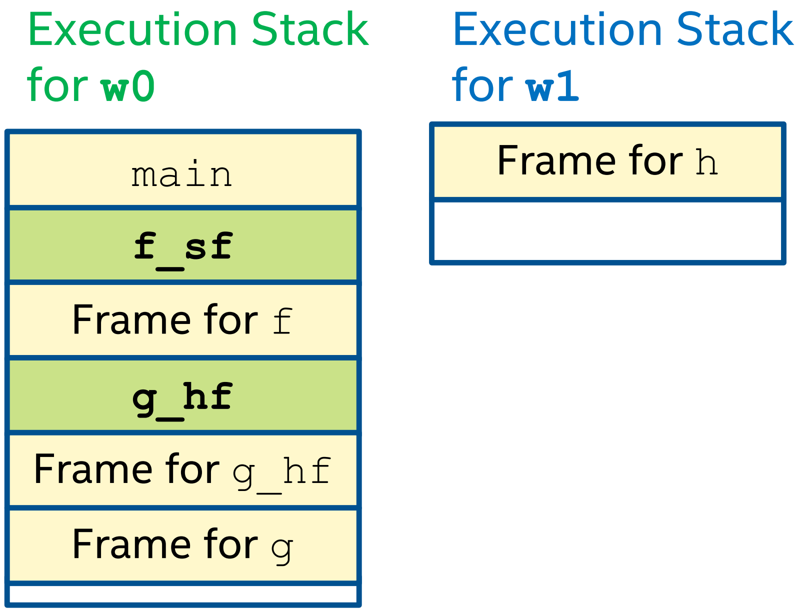
- **`cilk_for`** loops
  - Implemented in runtime code using recursive divide-and-conquer, with `cilk_spawn` and `cilk_sync`. May requires bootstrapping when building runtime; you need a Cilk Plus compiler to compile the Cilk Plus runtime.
- Pedigrees
  - Deterministic identifiers for strands in a Cilk Plus program. Compiler generates code to store pedigree nodes in stack frames and maintain them in a linked list. See [LSS12] for details.
- Programs with multiple user threads
  - Each user thread forms a separate team. In the work-stealing scheduler, a Cilk Plus system worker thread can steal from any team, but user threads can not steal from other teams.
- Exception handling
- Internal synchronization in the runtime
  - THE protocol on deques, locks on workers and full frames, etc.
    See comments in the source code for details…

Stack Switching in Cilk Plus

Suppose another worker `w1` steals the continuation of the `cilk_spawn` of `g` from worker `w0`.

- To resume the continuation in `f`, worker `w1` needs to use a different stack, since `h(n)` requires stack space for its frame.

```
int f(int n)
{
    int x, y;
    x = cilk_spawn g(n);
    y = h(n);
    cilk_sync;
    return x + y;
}
```


Execution Stack for **`w0`**

`main`

`f_sf`

Frame for `f`

`g_hf`

Frame for `g_hf`

Frame for `g`

Execution Stack for **`w1`**

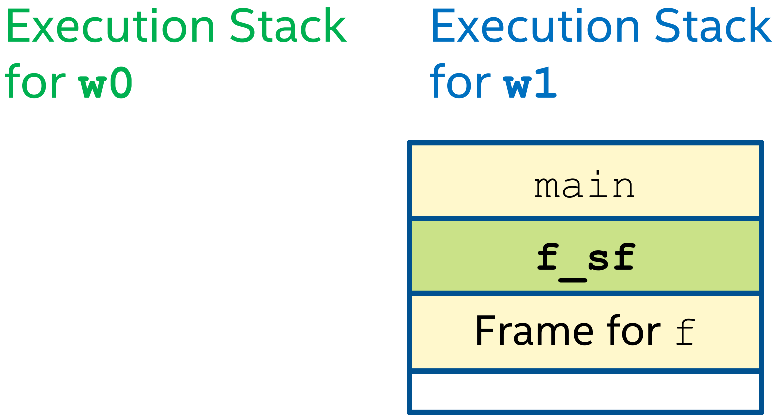
Frame for `h`

Stack Switching in Cilk Plus

Suppose another worker `w1` steals the continuation of the `cilk_spawn` of `g` from worker `w0`.

- To resume the continuation in `f`, worker `w1` needs to use a different stack, since `h(n)` requires stack space for its frame.
- After the `cilk_sync`, the runtime is guaranteed to resume execution of `f` back on the original stack, but possibly on a different worker!

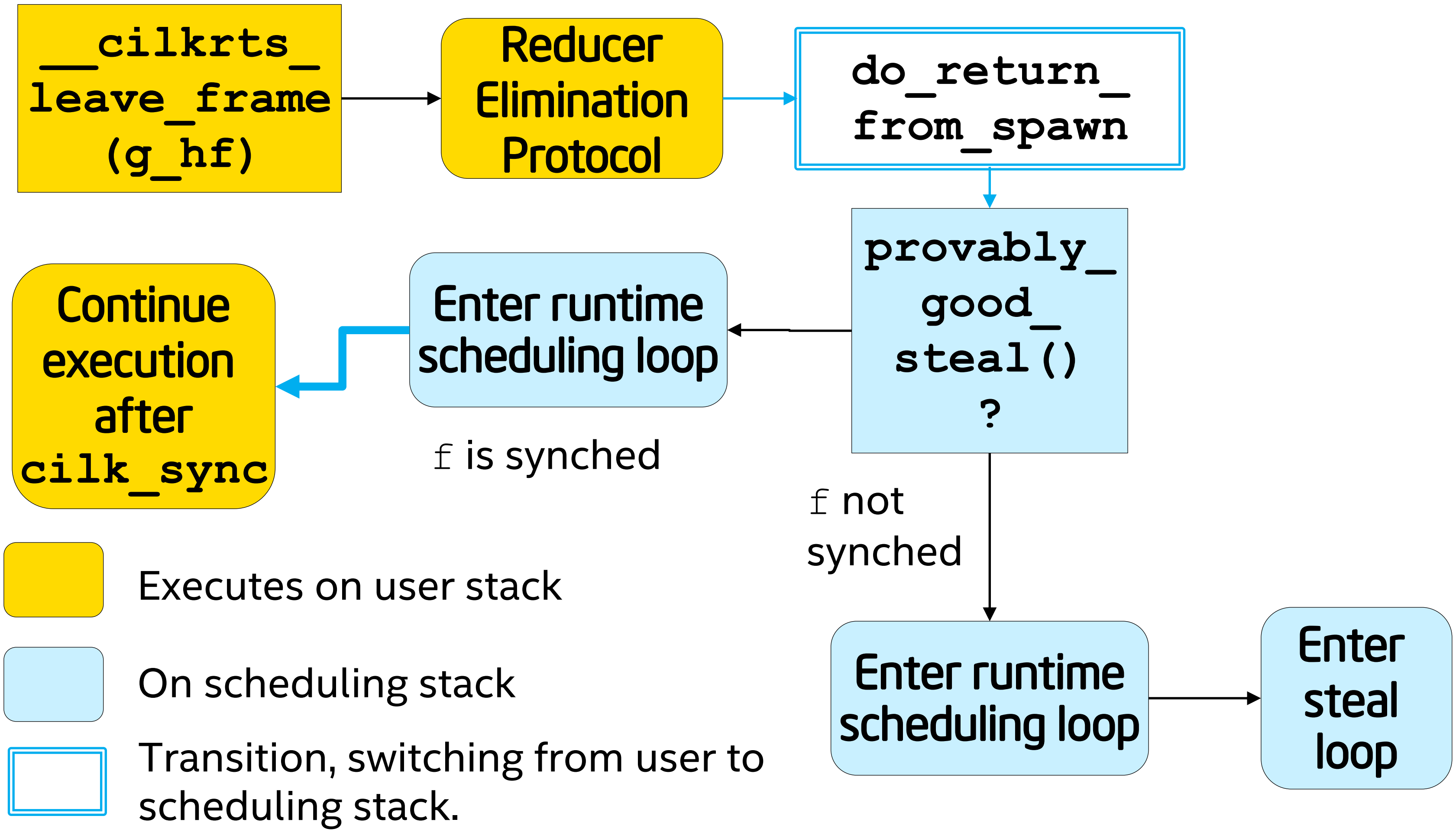
```
int f(int n)
{
    int x, y;
    x = cilk_spawn g(n);
    y = h(n);
    cilk_sync;
    return x + y;
}
```



Finally, when a worker thread jumps from executing user code into the runtime, it also switches to execute on a special **runtime scheduling stack**.

Return from `cilk_spawn`: Slow Path

The slow path for a return from a `cilk_spawn` changes stacks and enters the runtime:



`__cilkrts_leave_frame(g_hf)`

Reducer Elimination Protocol

`do_return_from_spawn`

`provably_good_steal()?`

`f` is synched

Enter runtime scheduling loop

Continue execution after `cilk_sync`

`f` not synched

Enter runtime scheduling loop

Enter steal loop

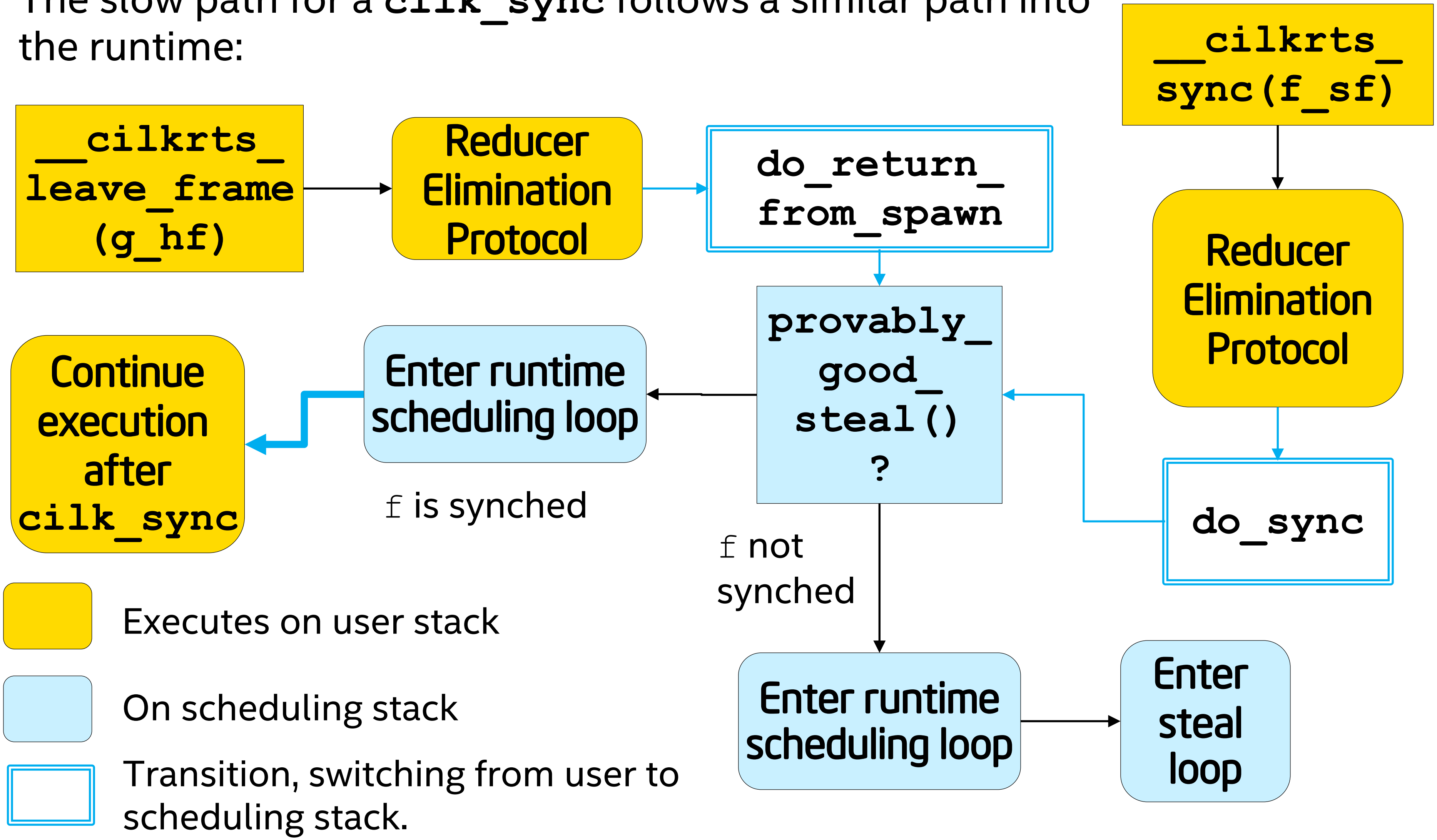
🟨 Executes on user stack

🟦 On scheduling stack

☐ Transition, switching from user to scheduling stack.

Stall at a `cilk_sync`: Slow Path

The slow path for a **`cilk_sync`** follows a similar path into the runtime:



`__cilkrts_leave_frame(g_hf)`

Reducer Elimination Protocol

`do_return_from_spawn`

`provably_good_steal()?`

Enter runtime scheduling loop

Continue execution after **`cilk_sync`**

`f` is synched

`f` not synched

Enter runtime scheduling loop

Enter steal loop

`__cilkrts_sync(f_sf)`

Reducer Elimination Protocol

`do_sync`

🟨 Executes on user stack

🟦 On scheduling stack

▭ Transition, switching from user to scheduling stack.

Cilk Plus Header Files

Some important header files in the `include` directory:

- All entry points and structures known by the Cilk Plus compiler.
  `internal/abi.h`
- Common macros and structures used by the runtime.
  `cilk/common.h`
- Prototype of Cilk Plus runtime API .
  `cilk/cilk_api.h`
- Example macros useful for hand-compilation of programs .
  `include/internal/cilk_fake.h`
- Reducer library .
  `include/cilk/reducer_*.h`

Cilk Plus Runtime Files

Some important runtime files:

- Definition of local state for each runtime worker (`w->l`).
  `local_state.h, local_state.c`
- Definition of global runtime state (`w->g`).
  `global_state.h, global_state.cpp`
- Definition of runtime ABI functions.
  `cilk-abi.c`
- Definition of `cilk_for` [Requires Cilk Plus compiler.]
  `cilk-abi-cilk_for.cpp`
- Heart of runtime scheduler.
  `scheduler.h, scheduler.c`
- Implementation of reducer maps.
  `reducer_impl.h, reducer_impl.cpp`
- Implementing stacks and stack switching.
  `cilk_fiber-*`
- Exceptions.
  `except*`

Worker Data Structure

The layout of the `__cilkrts_worker` structure is known to the Cilk Plus compiler.

- Deque pointers: `tail`, `head`, `exc`
- Worker id[1]: `self`
- Global runtime state: `g`
- Local state for a worker: `l`
- Current reducer map: `reducer_map`
- Current stack frame in call chain: `current_stack_frame`
- Reserved pointer: `reserved`
- System-dependent part of worker state: `sysdep`
- Current pedigree node. `__cilkrts_pedigree`

WARNING: Changing the layout of this structure also requires changing the compiler!

[1] For a worker, `w->self` is subtly different from the worker number returned by `__cilkrts_get_worker_number()`

Finding Out More About Cilk Plus

Community website at http://cilkplus.org

- Documentation
  - Language Specification
  - Cilk Plus ABI
- Cilk Plus Downloads:
  - Open source runtime
  - Binaries for Intel Cilk™ screen race detector and Intel Cilk™ view scalability analyzer.
  - Cilkpub library for user-contributed code. (Currently have deterministic parallel RNGs and parallel sort).
  - We welcome new contributions!
- Open Source Compilers
  - LLVM/Clang branch
  - GCC mainline 4.9.2
- Experimental Software
  - Branch of runtime supporting pipeline parallelism.
    https://www.cilkplus.org/piper-experimental-language-support-pipeline-parallelism-intel-cilk-plus

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2014, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

intel®