Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation

Tao B. Schardl

William S. Moses

Charles E. Leiserson

PPoPP 2017
February 7, 2017

CSAIL
MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

Example: Normalizing a Vector

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

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

Test: random vector, n = 64M. Machine: Amazon AWS c4.8xlarge.

Running time: 0.312 s

Example: Normalizing a Vector in Parallel

Cilk code for `normalize()`

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

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

A parallel loop replaces the original serial loop.

Test: random vector, n = 64M. Machine: Amazon AWS c4.8xlarge, 18 cores.

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

Running time on 18 cores: $T_{18} = 180.657$ s

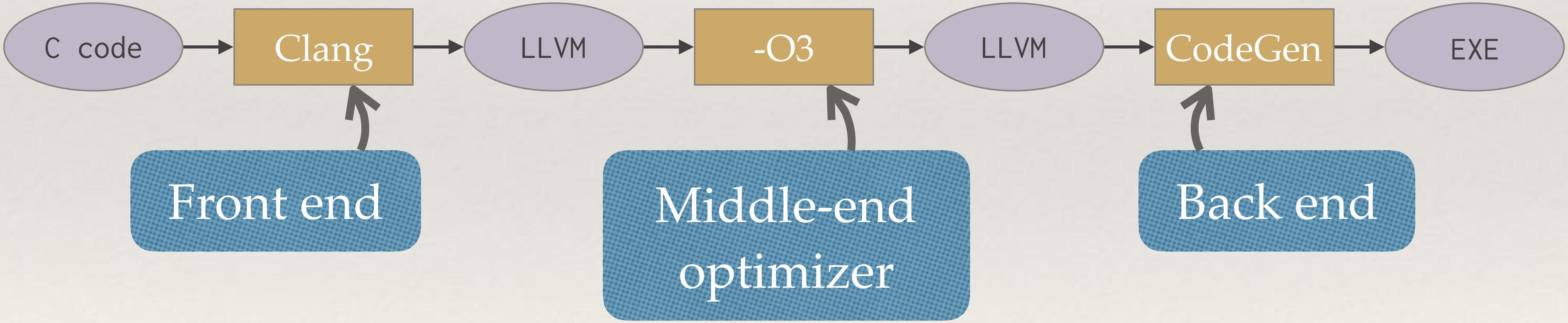
Running time on 1 core: $T_1 = 2600.287$ s

Terrible *work efficiency*:
$T_S / T_1 = 0.312/2600$
$\sim 1/8300$

The story for OpenMP is similar, but more complicated.

Code compiled using GCC 6.2. Cilk Plus/LLVM produces worse results.

The LLVM Compilation Pipeline



C code → Clang → LLVM → -O3 → LLVM → CodeGen → EXE

Front end

Middle-end optimizer

Back end

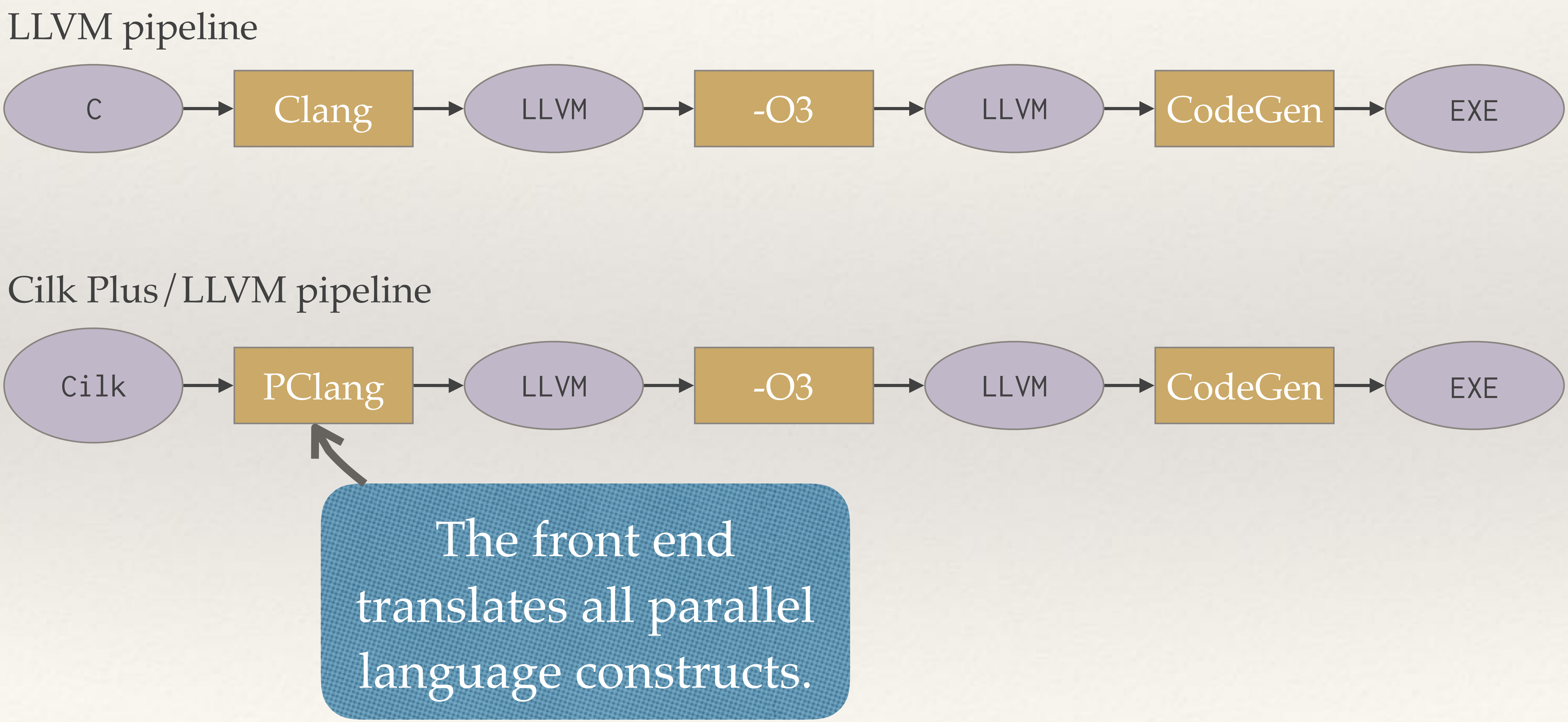
Effect of Compiling Serial Code

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

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

-O3

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

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

Compiling Parallel Code Today



LLVM pipeline

C → Clang → `LLVM` → -O3 → `LLVM` → CodeGen → `EXE`

Cilk Plus/LLVM pipeline

`Cilk` → PClang → `LLVM` → -O3 → `LLVM` → CodeGen → `EXE`

The front end translates all parallel language constructs.

Effect of Compiling Parallel Code

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

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

PClang

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

void normalize(double *restrict out, const double *restrict in, int n) {
  struct args_t args = { out, in, n };
  __cilkrts_cilk_for(normalize_helper, args, 0, n);
}

void normalize_helper(struct args_t args, int i) {
  double *out = args.out;
  double *in = args.in;
  int n = args.n;
  out[i] = in[i] / norm(in, n);
}
```

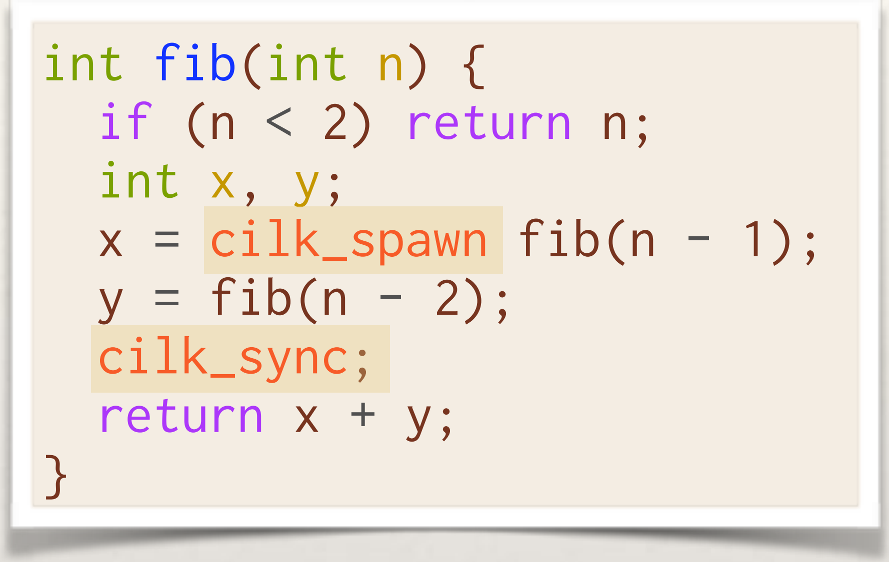
Call into runtime to execute parallel loop.

Helper function encodes the loop body.

Existing optimizations cannot move call to norm out of the loop.

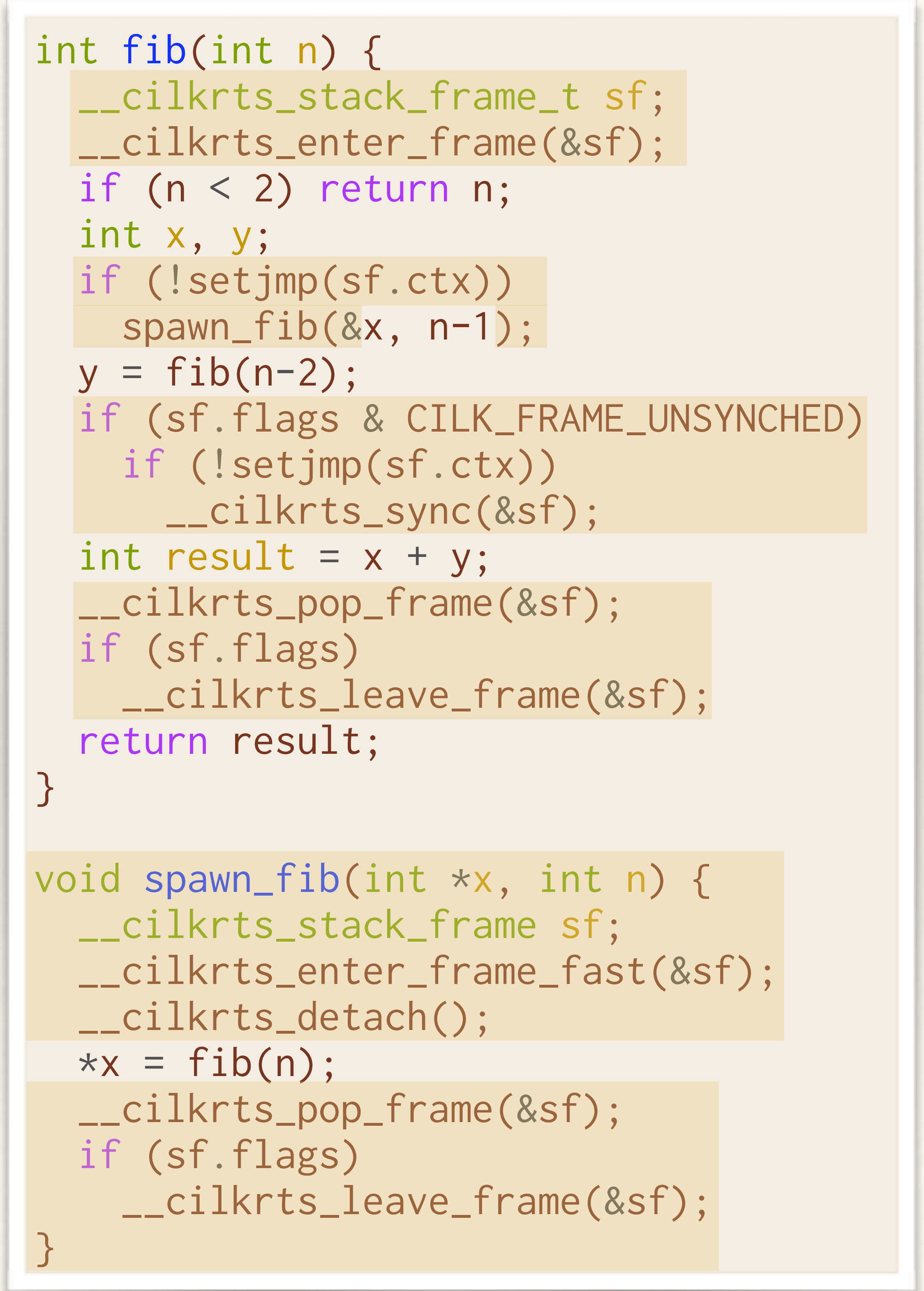
A More Complex Example

Cilk Fibonacci code



```
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;
}
```

PClang



```
int fib(int n) {
  __cilkrts_stack_frame_t sf;
  __cilkrts_enter_frame(&sf);
  if (n < 2) return n;
  int x, y;
  if (!setjmp(sf.ctx))
    spawn_fib(&x, n-1);
  y = fib(n-2);
  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_fib(int *x, int n) {
  __cilkrts_stack_frame sf;
  __cilkrts_enter_frame_fast(&sf);
  __cilkrts_detach();
  *x = fib(n);
  __cilkrts_pop_frame(&sf);
  if (sf.flags)
    __cilkrts_leave_frame(&sf);
}
```

Optimization passes struggle to optimize around these opaque runtime calls.

Old Idea: A Parallel IR

- Let's embed parallelism directly into the compiler's intermediate representation (IR)!

LLVM pipeline

C → Clang → LLVM → -O3 → LLVM → CodeGen → EXE

Cilk Plus/LLVM pipeline

Cilk → PClang → LLVM → -O3 → LLVM → CodeGen → EXE

A better Cilk compilation pipeline

Cilk → PClang → PLLVM → -O3 → PLLVM → CodeGen → EXE

New IR that encodes parallelism for optimization.

Previous Attempts at Parallel IR's

- Parallel precedence graphs [SW91, SHW93]
- Parallel flow graphs [SG91, GS93]
- Concurrent SSA [LMP97, NUS98]
- Parallel program graphs [SS94, S98]
- HPIR [ZS11, BZS13]
- SPIRE [KJAI12]
- INSPIRE [JPTKF13]
- LLVM's parallel loop metadata
- "[LLVMdev] [RFC] Parallelization metadata and intrinsics in LLVM (for OpenMP, etc.)" http://lists.llvm.org/pipermail/llvm-dev/2012-August/052477.html
- "[LLVMdev] [RFC] Progress towards OpenMP support" http://lists.llvm.org/pipermail/llvm-dev/2012-September/053326.html
- LLVM Parallel Intermediate Representation: Design and Evaluation Using OpenSHMEM Communications [KJIAC15]
- LLVM Framework and IR Extensions for Parallelization, SIMD Vectorization and Offloading [TSSGMGZ16]

Parallel IR: A Bad Idea?

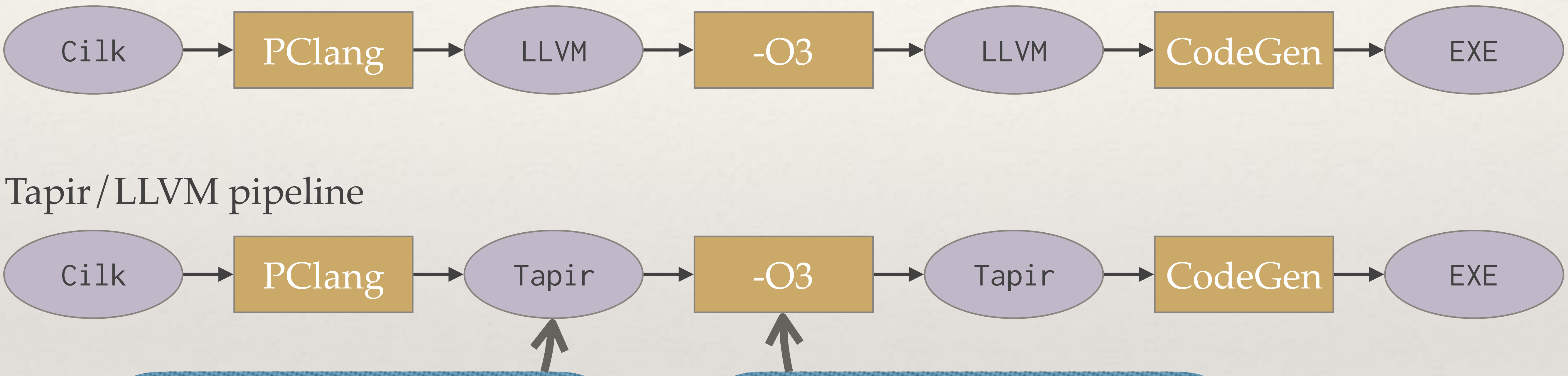
From “[LLVMdev] LLVM Parallel IR,” 2015:

- “[I]ntroducing [parallelism] into a so far ‘sequential’ IR will cause **severe breakage and headaches**.”
- “[P]arallelism is invasive by nature and would have to **influence most optimizations**.”
- “[It] is not an easy problem.”
- “[D]efining a parallel IR (with first class parallelism) is a research topic…”

Other communications, 2016–2017:

- “There are **a lot of information needs** to be represented in IR for [back end] transformations for OpenMP.” [Private communication]
- “If you support all [parallel programming features] in the IR, a *lot* [of LOC]… would probably have to be modified in LLVM.” [[RFC] IR-level Region Annotations]

Tapir: Task-based Asymmetric Parallel IR



Cilk Plus/LLVM pipeline

`Cilk` → PClang → `LLVM` → -O3 → `LLVM` → CodeGen → `EXE`

Tapir/LLVM pipeline

`Cilk` → PClang → `Tapir` → -O3 → `Tapir` → CodeGen → `EXE`

Tapir adds **three instructions** to LLVM IR that encode **fork-join parallelism**.

With **few changes**, LLVM's **existing optimizations and analyses** work on **parallel code**.

Normalizing a Vector in Parallel with Tapir

Cilk code for `normalize()`

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

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

Test: random vector, n = 64M. Machine: Amazon AWS c4.8xlarge, 18 cores.

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

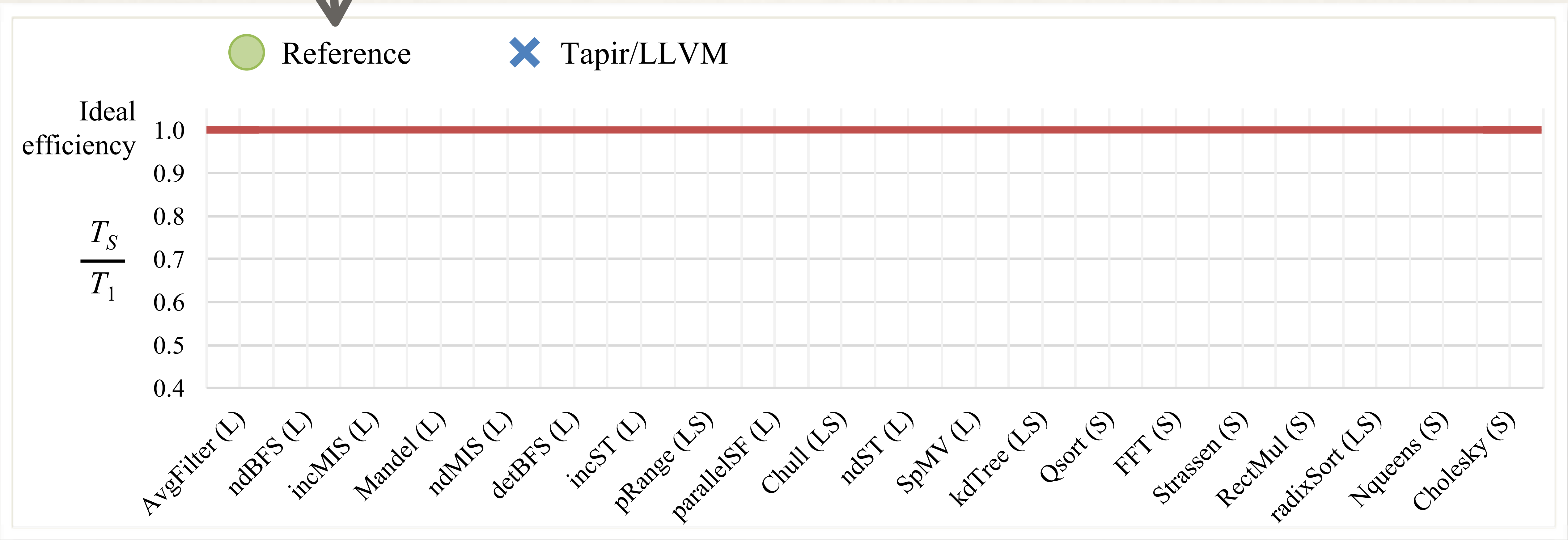
Compiled with Tapir/LLVM, running time on 1 core: $T_1 = 0.321$ s

Compiled with Tapir/LLVM, running time on 18 cores: $T_{18} = 0.081$ s

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

Work-Efficiency Improvement

Same as Tapir/LLVM, but the front end handles parallel language constructs the traditional way.

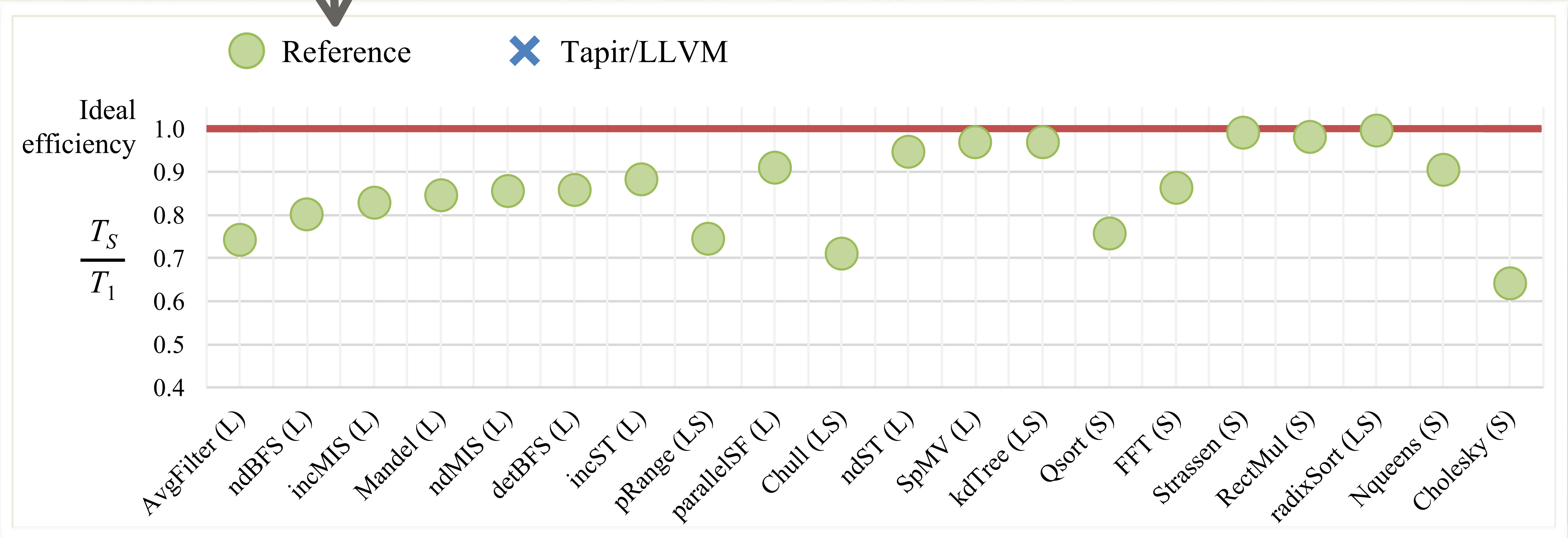


Decreasing difference between Tapir/LLVM and Reference

Test machine: Amazon AWS c4.8xlarge, with 18 cores clocked at 2.9 GHz, 60 GiB DRAM

Work-Efficiency Improvement

Same as Tapir/LLVM, but the front end handles parallel language constructs the traditional way.

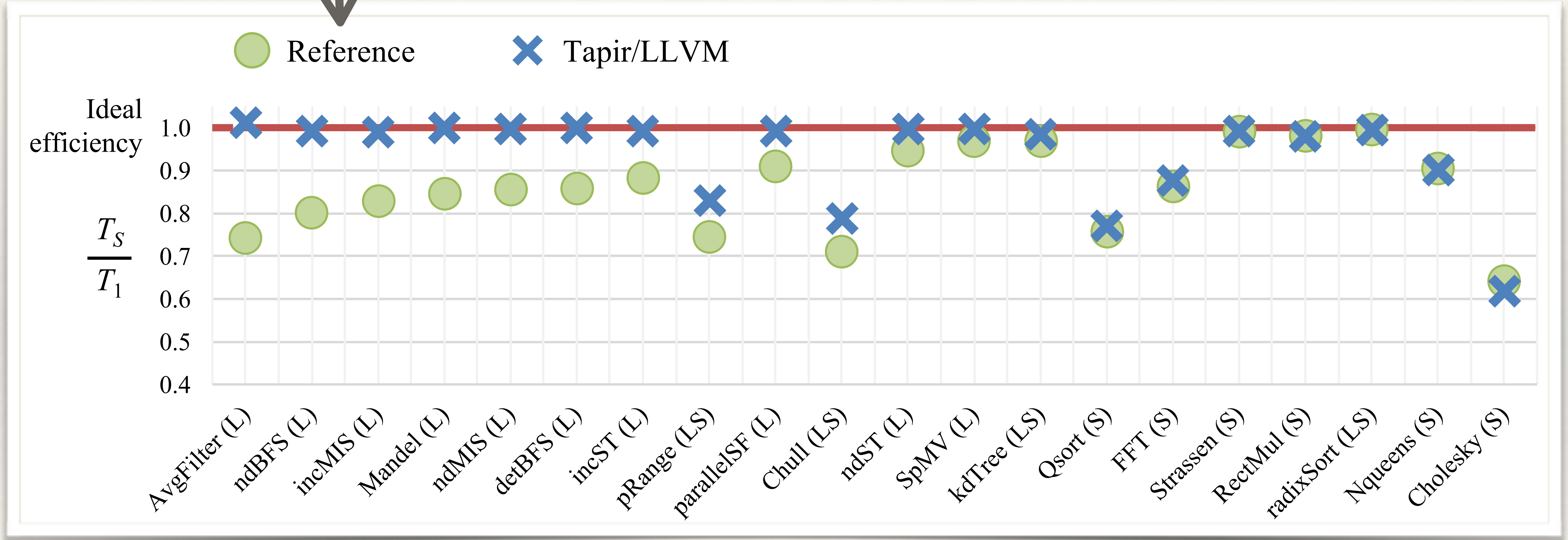


Decreasing difference between Tapir/LLVM and Reference

Test machine: Amazon AWS c4.8xlarge, with 18 cores clocked at 2.9 GHz, 60 GiB DRAM

Work-Efficiency Improvement

Same as Tapir/LLVM, but the front end handles parallel language constructs the traditional way.



Decreasing difference between Tapir/LLVM and Reference

Test machine: Amazon AWS c4.8xlarge, with 18 cores clocked at 2.9 GHz, 60 GiB DRAM

Implementing Tapir/LLVM

Compiler component	*LLVM 4.0svn (lines)*	*Tapir/LLVM (lines)*	
Instructions	105,995	943	1,768
Memory behavior	21,788	445	
Optimizations	152,229	380	
Parallelism lowering	0	3,782	
Other	3,803,831	460	
Total	4,083,843	6,010	

Compiler Analyses and Optimizations

What did we do to **adapt existing analyses and optimizations**?

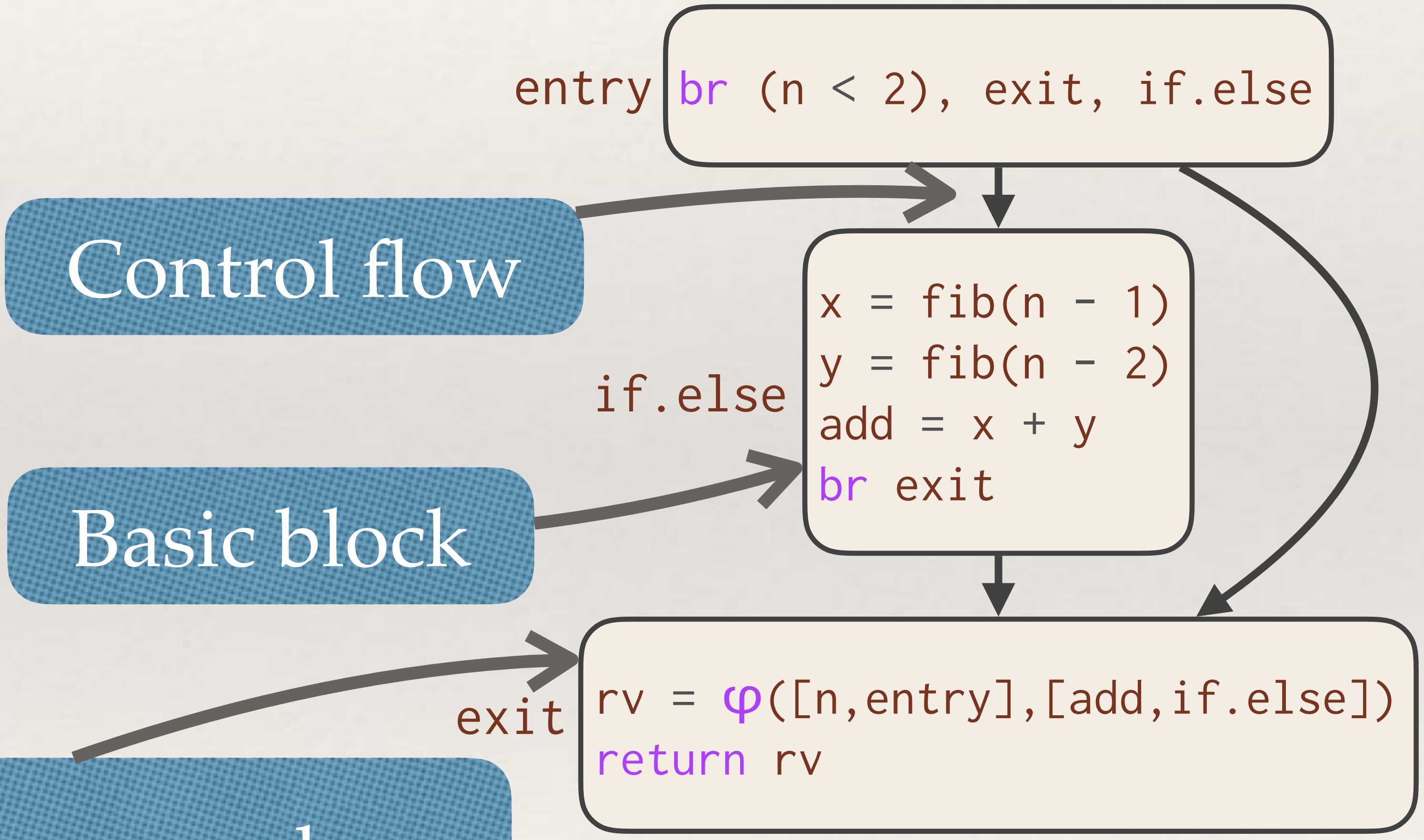
- Dominator analysis: no change
- Common-subexpression elimination: no change
- Loop-invariant-code motion: 25-line change
- Tail-recursion elimination: 68-line change

Tapir also enables **new parallel optimizations**, such as **unnecessary-synchronization elimination** and **puny-task elimination**, which were implemented in 52 lines total.

LLVM IR

LLVM represents each function as a **control-flow graph (CFG)**.

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



Control flow

Basic block

entry
```
br (n < 2), exit, if.else
```

if.else
```
x = fib(n - 1)
y = fib(n - 2)
add = x + y
br exit
```

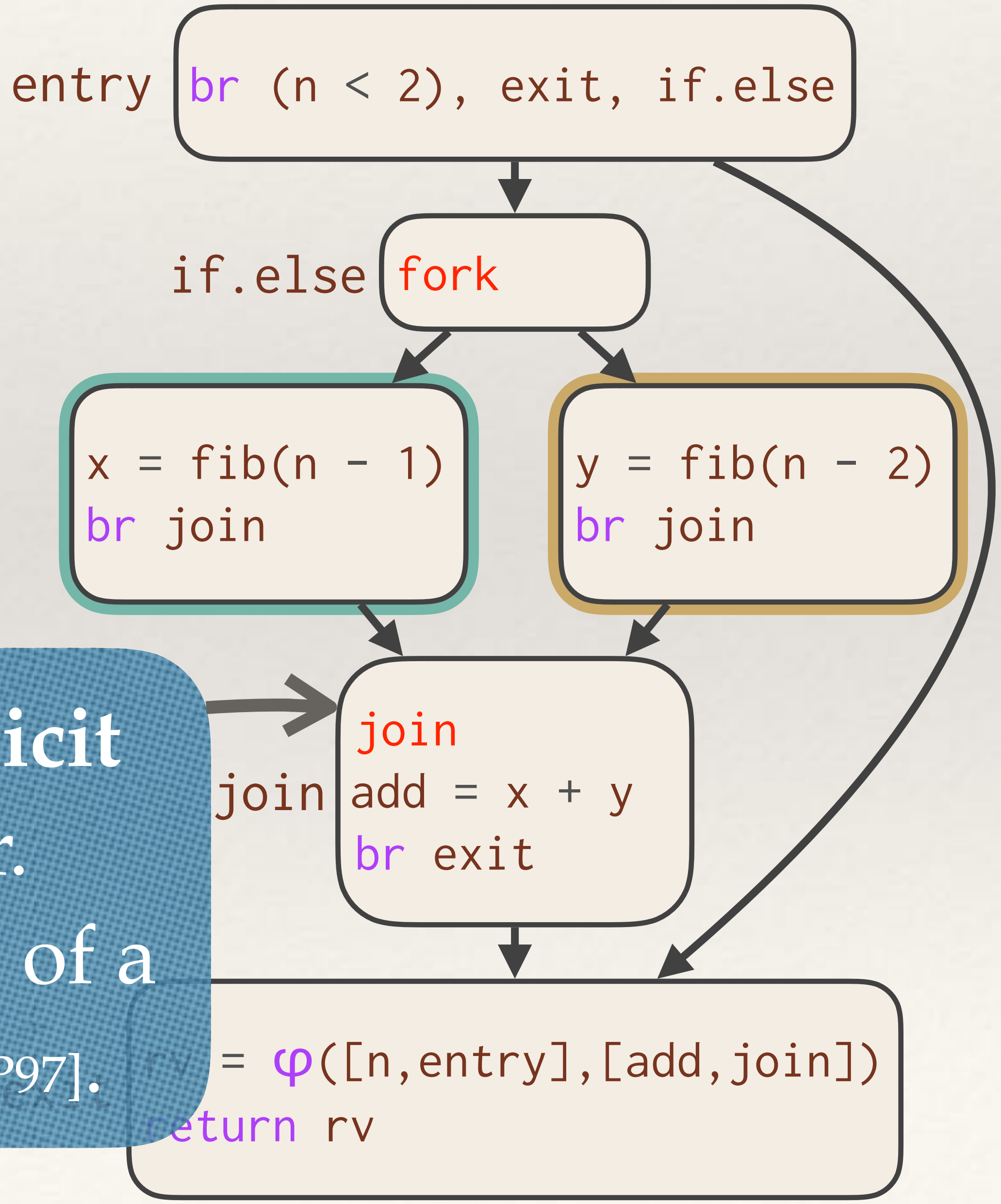
exit
```
rv = φ([n,entry],[add,if.else])
return rv
```

For serial code a basic block sees values from **just one predecessor** at runtime.

Example Previous Parallel IR

Previous parallel IR's based on CFG's model parallel tasks **symmetrically**.

```
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;
}
```



entry
```
br (n < 2), exit, if.else
```

if.else
```
fork
```

```
x = fib(n - 1)
br join
```

```
y = fib(n - 2)
br join
```

join
```
join
add = x + y
br exit
```

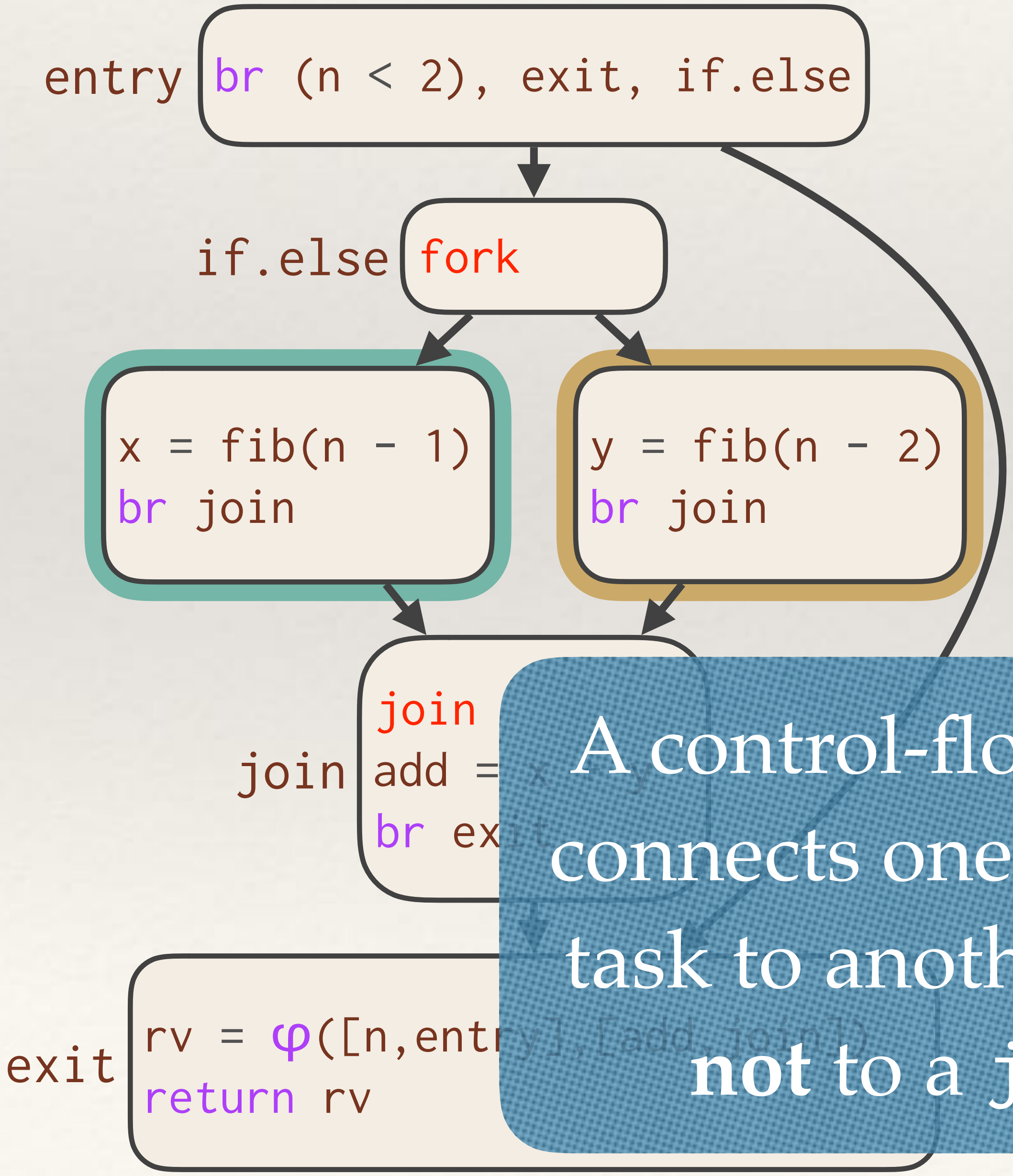
exit
```
rv = φ([n,entry],[add,join])
return rv
```

Problem: The `join` block **breaks implicit assumptions** made by the compiler.
Example: Values from **all** predecessors of a `join` must be available at runtime [LMP97].

Tapir vs. Previous Approaches

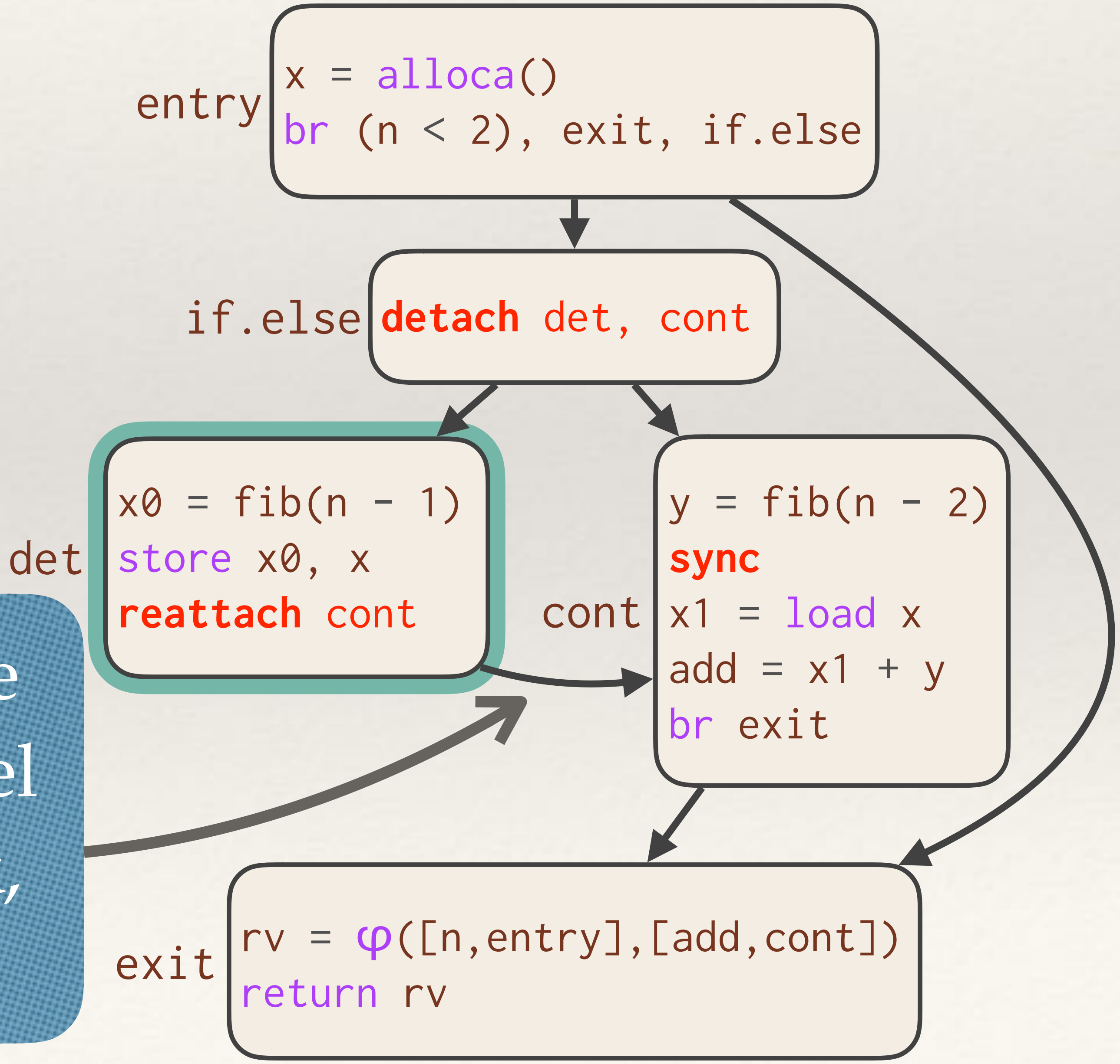
Tapir's instructions model parallel tasks **asymmetrically**.

Symmetric CFG



```
entry:    br (n < 2), exit, if.else
if.else:  fork
          x = fib(n - 1)
          br join
          y = fib(n - 2)
          br join
join:     join
          add = x
          br ex
exit:     rv = φ([n,entr
          return rv
```

Tapir CFG

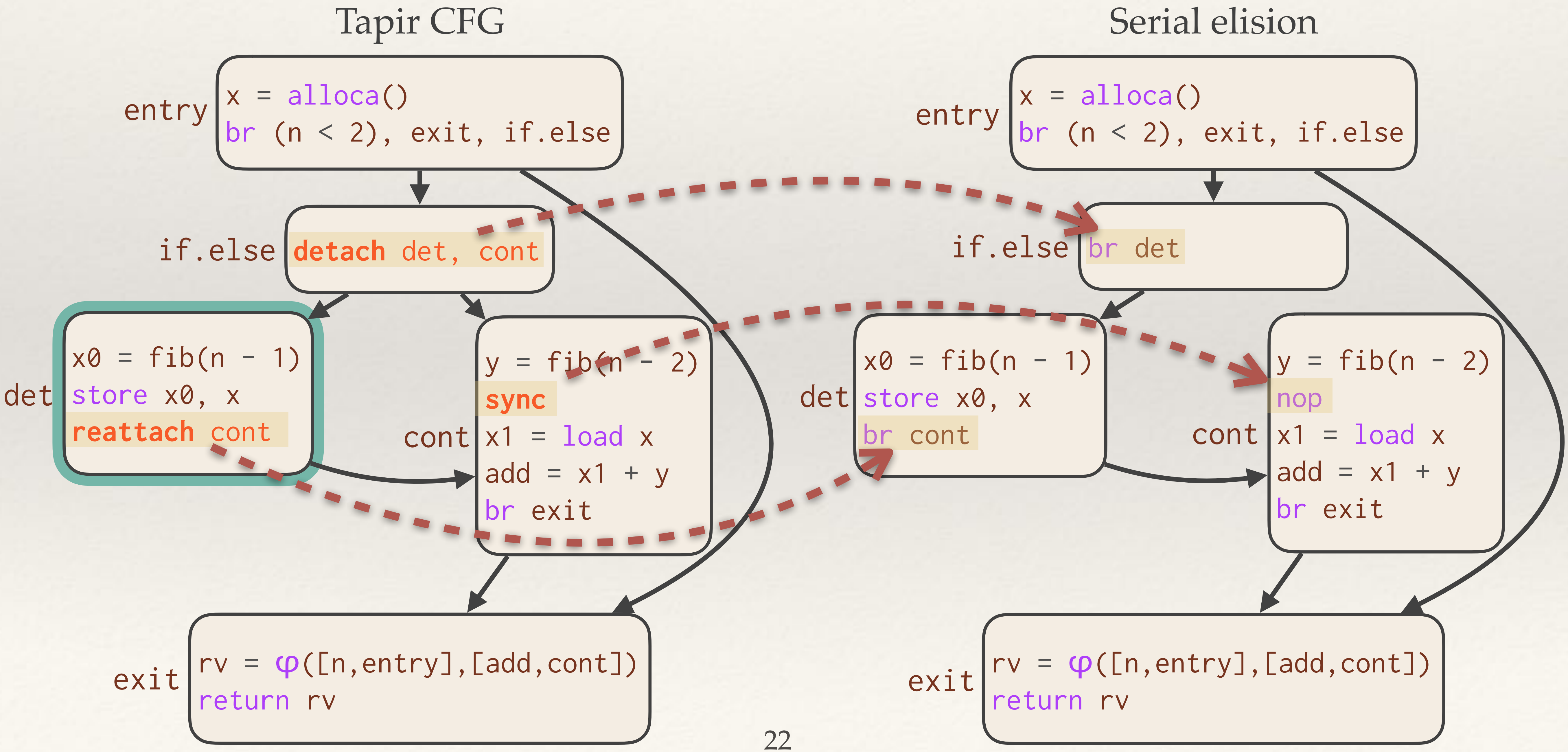


```
entry:    x = alloca()
          br (n < 2), exit, if.else
if.else:  detach det, cont
det:      x0 = fib(n - 1)
          store x0, x
          reattach cont
cont:     y = fib(n - 2)
          sync
          x1 = load x
          add = x1 + y
          br exit
exit:     rv = φ([n,entry],[add,cont])
          return rv
```

A control-flow edge connects one parallel task to another task, **not** to a `join`.

Serial Elision of a Tapir Program

Tapir models the **serial elision** of the parallel program.



Tapir CFG

```
entry:
x = alloca()
br (n < 2), exit, if.else

if.else:
detach det, cont

det:
x0 = fib(n - 1)
store x0, x
reattach cont

cont:
y = fib(n - 2)
sync
x1 = load x
add = x1 + y
br exit

exit:
rv = φ([n,entry],[add,cont])
return rv
```

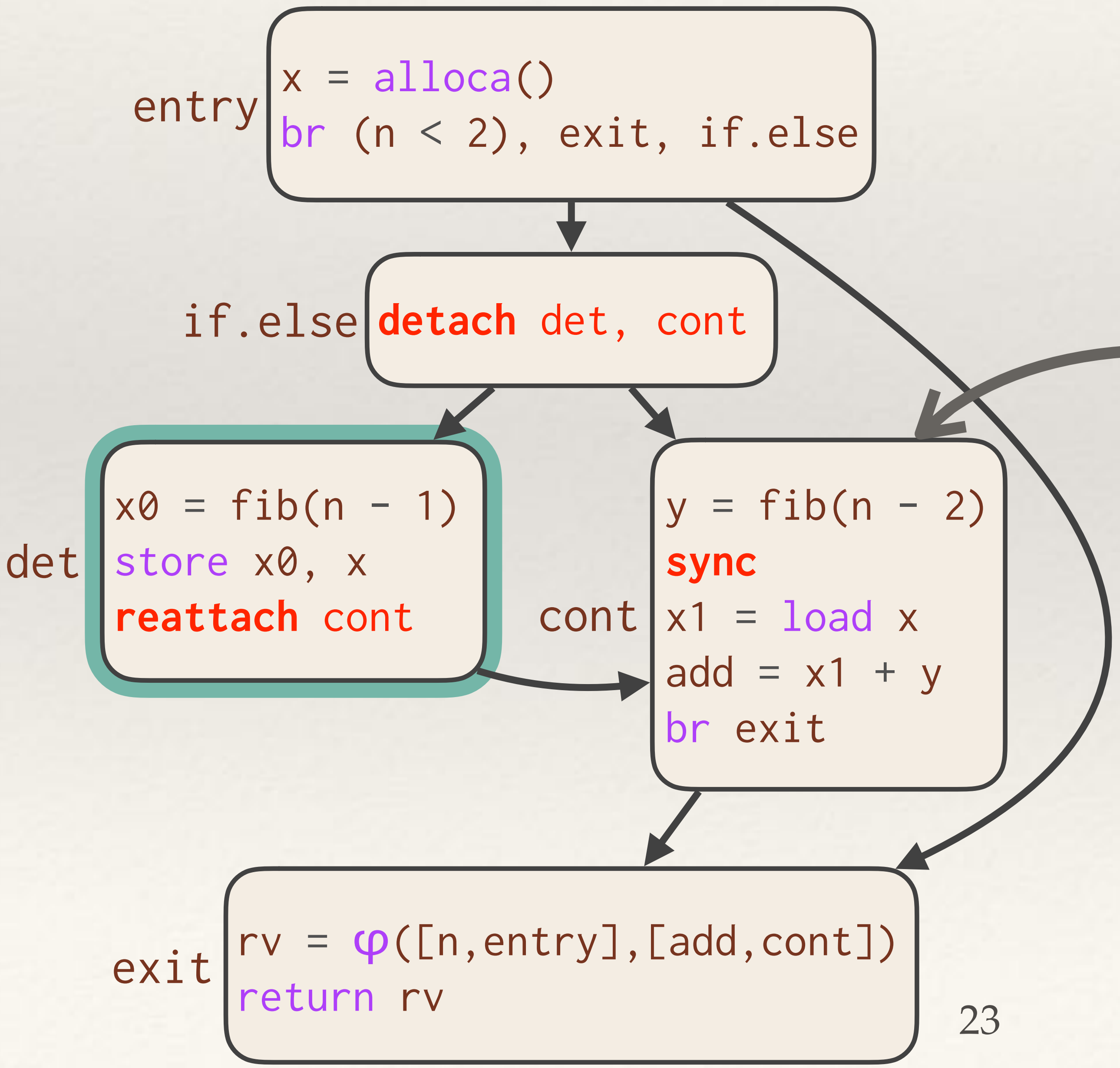
Serial elision

```
entry:
x = alloca()
br (n < 2), exit, if.else

if.else:
br det

det:
x0 = fib(n - 1)
store x0, x
br cont

cont:
y = fib(n - 2)
nop
x1 = load x
add = x1 + y
br exit

exit:
rv = φ([n,entry],[add,cont])
return rv
```

Reasoning About a Tapir CFG

Intuitively, much of the compiler can reason about a Tapir CFG as a **minor change** to that CFG's serial elision.

Tapir CFG



entry
```
x = alloca()
br (n < 2), exit, if.else
```

if.else
```
detach det, cont
```

det
```
x0 = fib(n - 1)
store x0, x
reattach cont
```

cont
```
y = fib(n - 2)
sync
x1 = load x
add = x1 + y
br exit
```

exit
```
rv = φ([n,entry],[add,cont])
return rv
```

Many parts of the compiler can apply **standard implicit assumptions** of the CFG to this block.

Status of Tapir

- Try Tapir/LLVM yourself!

```
git clone —recursive https://github.com/wsmoses/Tapir-Meta.git
```

- We have a **prototype front end** for Tapir/LLVM that is substantially compliant with the Intel Cilk Plus language specification.
- Tapir/LLVM achieves **comparable or better performance** versus GCC, ICC, and Cilk Plus/LLVM, and is becoming **comparably robust**.
- Last fall, a **software performance-engineering class** at MIT with ~100 undergrads used Tapir/LLVM as their compiler.
- Tapir/LLVM includes a **provably good** race detector for **verifying** the existence of race bugs **deterministically**.
- We're continuing to enhance Tapir/LLVM with bug fixes, new compiler optimizations, and other new features.

Backup

What Is Parallel Programming?

- Pthreads
- Message passing
- Vectorization
- Task parallelism
- Data parallelism
- Dataflow
- Multicore
- HPC
- GPU's
- Heterogeneous computing

- Shared memory
- Distributed memory
- Clients and servers
- Races and locks
- Concurrency
- Memory models
- Scheduling and load balancing
- Work efficiency
- Parallel speedup
- Etc.

Tapir does NOT directly address ALL of these.

Focus of Tapir

Tapir strives to make it easy for **average programmers** to write **efficient** programs that achieve **parallel speedup**.

- Multicores
- Task parallelism
- Simple and extensible
- Deterministic debugging
- Serial semantics

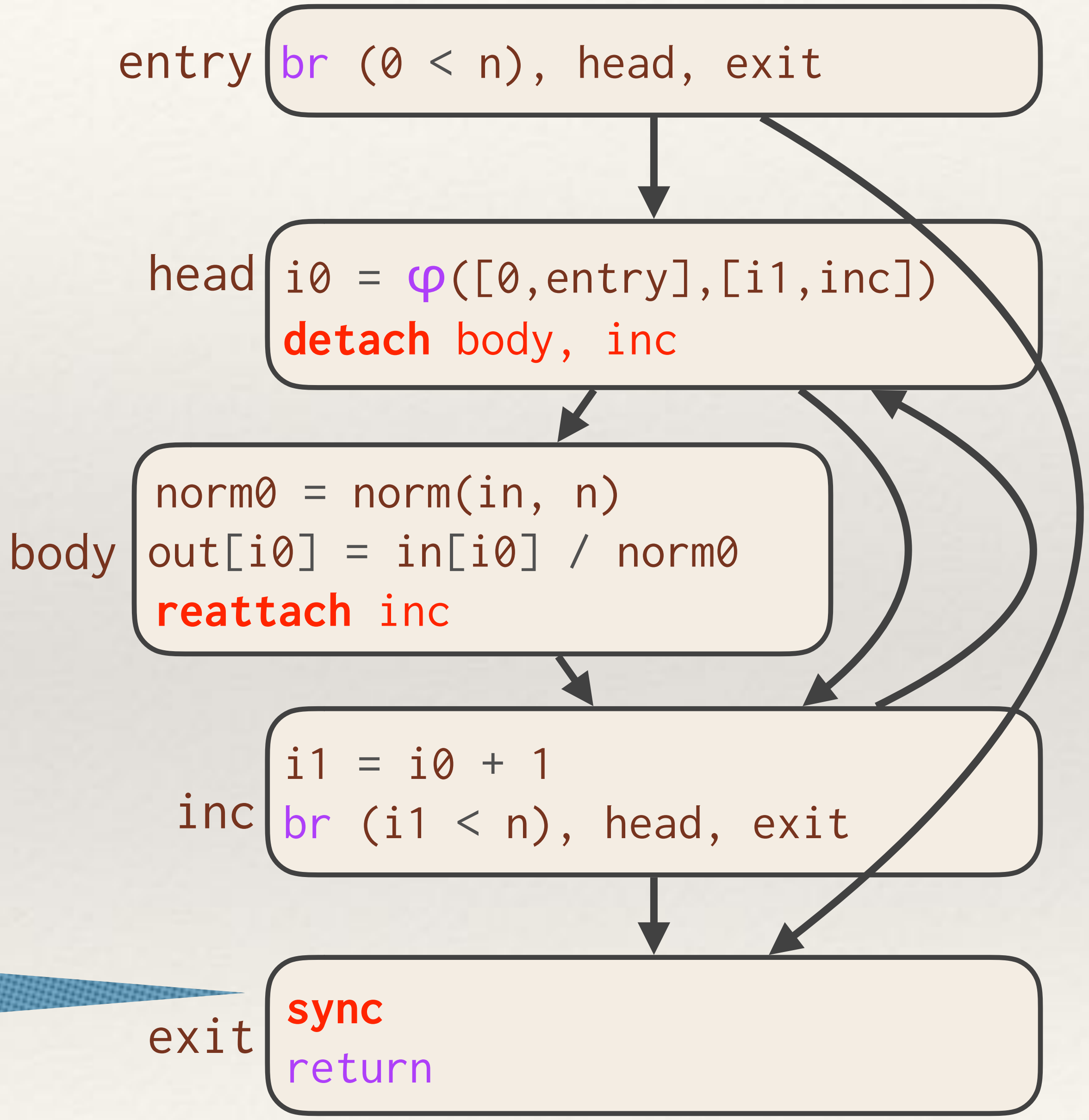
- Simple execution model
- Work efficiency
- Parallel speedup
- Composable performance
- Parallelism, not concurrency

Parallel Loops in Tapir

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

Parallel loop resembles a serial loop with a detached body.

The sync waits on a dynamic set of detached sub-CFG's.



```
entry  br (0 < n), head, exit

head   i0 = φ([0,entry],[i1,inc])
       detach body, inc

body   norm0 = norm(in, n)
       out[i0] = in[i0] / norm0
       reattach inc

inc    i1 = i0 + 1
       br (i1 < n), head, exit

exit   sync
       return
```

Race Bugs

Parallel programming is strictly harder than serial programming because of **race bugs**.

Example: A buggy `norm()` function

```
__attribute__((const))
double norm(const double *A, int n) {
  double sum = 0.0;
  #pragma omp parallel for
  for (int i = 0; i < n; ++i)
    sum += A[i] * A[i];
  return sqrt(sum);
}
```

Concurrent updates to `sum` can **nondeterministically** produce **different results.**

How do I spot these bugs in my million-line codebase?

How do I find a race if I'm "lucky" enough to never see different results?

What if the **compiler** creates the race?

A Compiler Writer's Nightmare

Bug 55555 - Transformation puts race into race-free code

Attachments

Parallel test case (text/plain)
2017-02-04, Angry Hacker

Add an attachment

Angry Hacker 2017-02-04

Created attachment 12345
Parallel test case

My parallel code is race free, but the compiler put a race in it!!
>:(

Compiled program

1 run ✔

10 runs ✔

1000 runs ✔

Despite the programmer's assertion, multiple runs indicate no problem.

- Is the compiler buggy?
- Is the programmer wrong?

Debugging Tapir/LLVM

Tapir/LLVM contains a **provably good** race detector for **verifying** the existence of race bugs **deterministically**.

- Given a program and an input — e.g., a regression test — the race-detection algorithm **guarantees** to find a race if one exists or **certify** that no races exist [FL99, UAFL16].
- The race-detection algorithm introduces **approximately constant overhead**.
- We used the race detector together with `opt` to **pinpoint optimization passes** that incorrectly introduce races.

What about Thread Sanitizer?

Efficient race detectors have been developed, including FastTrack [FF09] and Thread Sanitizer [KPIV11].

- These detectors are **best effort**: they are **not** guaranteed to find a race if one exists.
- These detectors are designed to handle **a few** parallel threads, comparable to the number of processors.
- Task-parallel languages are designed to get parallel speedup by exposing **orders of magnitude more** parallel tasks than processors.

Example: Normalizing a Vector with OpenMP

OpenMP code for `normalize()`

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

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

Test: random vector, n = 64M. Machine: Amazon AWS c4.8xlarge, 18 cores.

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

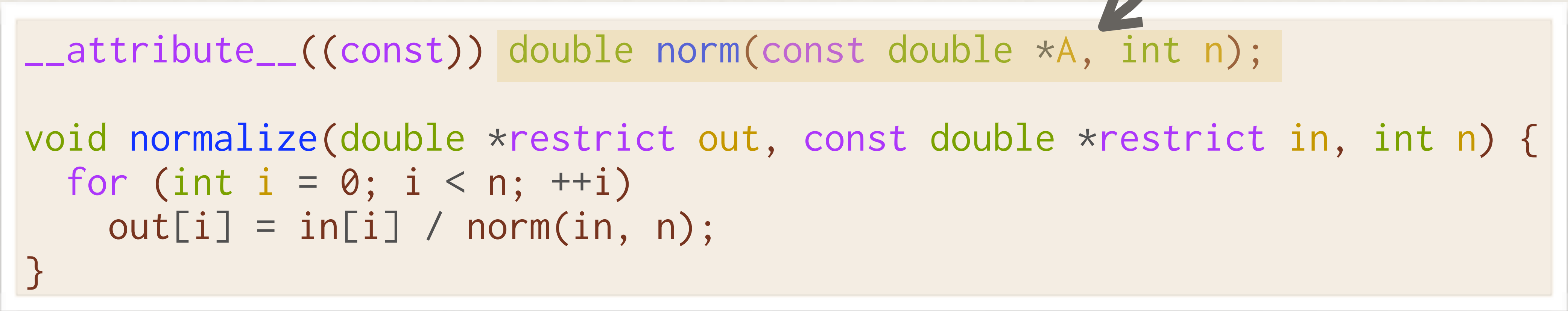
Compiled with LLVM 4.0, running time on 1 core: $T_1 = 0.329$ s

Compiled with LLVM 4.0, running time on 18 cores: $T_{18} = 0.205$ s

Great work efficiency without Tapir?

Work Analysis of Serial Normalize

$T(\text{norm}) = O(n)$



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

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

$T(\text{normalize}) =$
$n * T(\text{norm}) + O(n) =$
$O(n^2)$

Work Analysis After LICM

$T(\mathtt{norm}) = O(n)$

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

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

$$T(\mathtt{normalize}) = T(\mathtt{norm}) + O(n) = O(n)$$

Compiling OpenMP Normalize

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

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

PClang → -O3

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

void normalize(double *restrict out, const double *restrict in, int n) {
  __kmpc_fork_call(omp_outlined, n, out, in);
}

void omp_outlined(int n, double *restrict out,
                  const double *restrict in) {
  int local_n = n; double *local_out = out, *local_in = in;
  __kmpc_for_static_init(&local_n, &local_out, &local_in);
  double tmp = norm(in, n);
  for (int i = 0; i < local_n; ++i)
    local_out[i] = local_in[i] / tmp;
  __kmpc_for_static_fini();
}
```

Each processor runs the helper function once.

Helper function contains a serial copy of the original loop.

Work Analysis of OpenMP Normalize

How much **work** (total computation outside of scheduling) does this code do?

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

void normalize(double *restrict out, const double *restrict in, int n) {
  __kmpc_fork_call(omp_outlined, n, out, in);
}

void omp_outlined(int n, double *restrict out,
                  const double *restrict in) {
  int local_n = n; double *local_out = out, *local_in = in;
  __kmpc_for_static_init(&local_n, &local_out, &local_in);
  double tmp = norm(in, n);
  for (int i = 0; i < local_n; ++i)
    local_out[i] = local_in[i] / tmp;
  __kmpc_for_static_fini();
}
```

$T_1(\mathrm{norm}) = O(n)$

$T_1(\mathrm{omp\_outlined}) = T_1(\mathrm{norm}) + O(\mathrm{local\_n}) = O(n)$

$T(\mathrm{normalize}) = P * T_1(\mathrm{omp\_outlined}) = O(n * P)$

Let P be the number of processors.

What Does This Analysis Mean?

$T(\texttt{normalize}) = \mathrm{O}(\texttt{n} * P)$

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

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

Original serial running time: $T_S = 0.312$ s

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

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

- This code is only work-efficient on **one processor**.
- **Only minimal** parallel speedup is possible.
- The problem **persists** whether `norm` is serial or parallel.
- This code **slows down** when not all processors are available.

Tapir's Optimization Strategy

Tapir strives to optimize parallel code according the **work-first principle**:

- **First** optimize the **work**, not the parallel execution.
- Sacrifice **minimal** work to support parallel execution.

The work-first principle helps to ensure that parallel codes can achieve speedup in **all runtime environments**.

Task Parallelism

Task parallelism provides **simple linguistics** for **average programmers** to write parallel code.

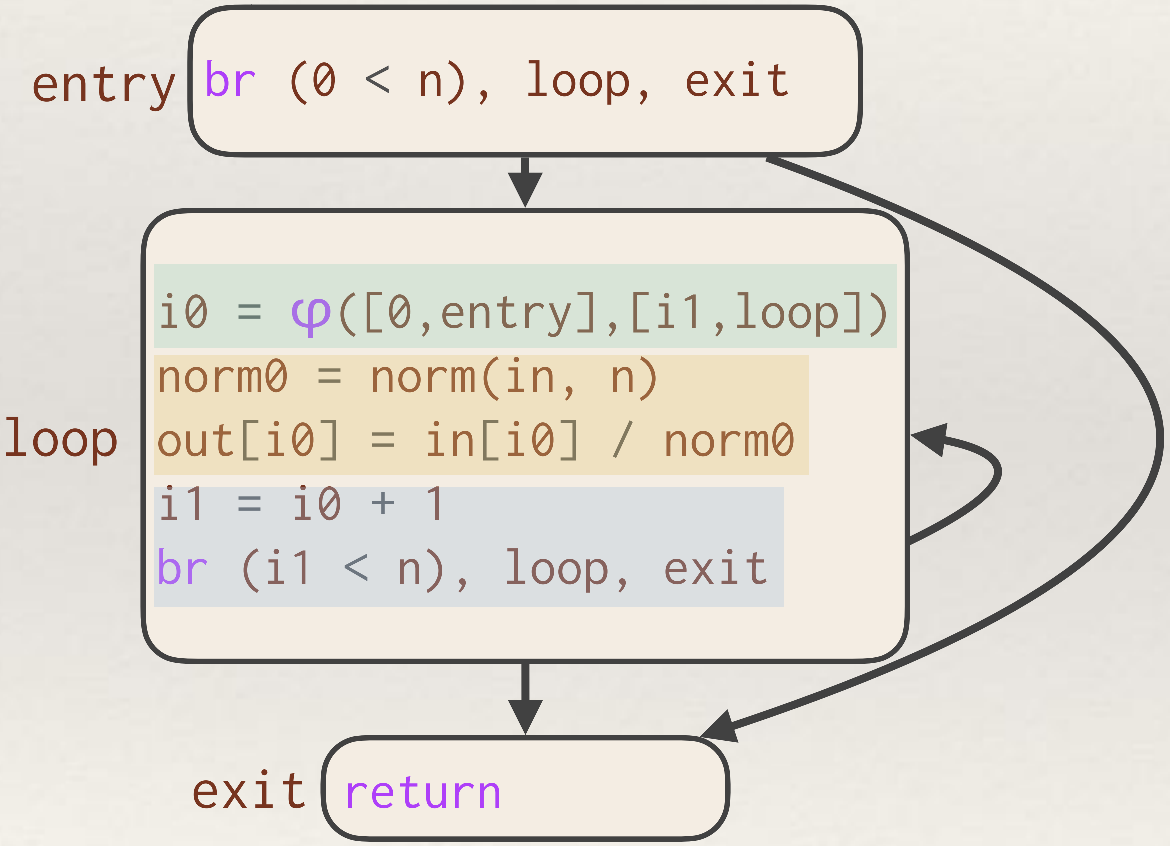
Example: parallel quicksort

```
void pqsort(int64_t array[], size_t l,
            size_t h) {
  if (h - l < COARSENING)
    return qsort_base(array, l, h);
  size_t part = partition(array, l, h);
  cilk_spawn pqsort(array, l, part);
  pqsort(array, part, h);
  cilk_sync;
}
```

The child function is *allowed* (but not required) to execute in parallel with the parent caller.

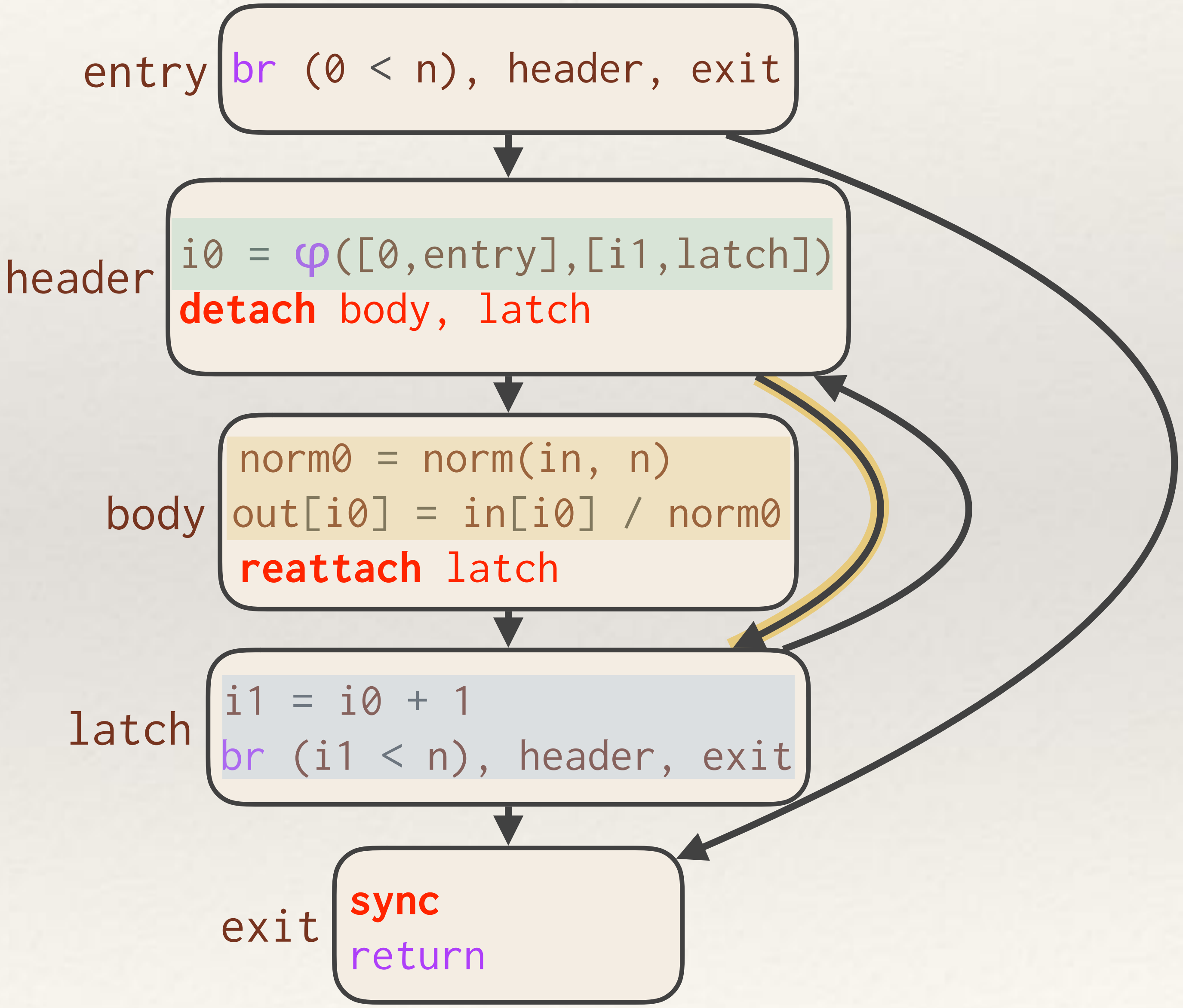
Control cannot pass this point until all spawned children have returned.

Example: Parallel Loops in Tapir



CFG for serial normalize

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

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

exit    return
```



Tapir CFG for parallel normalize

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

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

body    norm0 = norm(in, n)
        out[i0] = in[i0] / norm0
        reattach latch

latch   i1 = i0 + 1
        br (i1 < n), header, exit

exit    sync
        return
```

Concurrency Is Complicated

Interactions between threads can confound traditional compiler optimizations.

Thread 1

```
a = 1;
```

Thread 2

```
if (x.load(RLX))
  if (a)
    y.store(1, RLX);
```

Thread 3

```
if (y.load(RLX))
  x.store(1, RLX);
```

This program produces different results under the C11 memory model if Threads 1 and 2 are sequentialized [VBCMN15].

Parallelism Sans Concurrency

Task-parallel languages (e.g., Cilk, Habanero, OpenMP) let programmers write **parallel** code **without concurrency**.

C code for `normalize()`

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

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

Cilk code for `normalize()`

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

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

Same control, but `cilk_for` indicates an opportunity to speed up execution using parallel processors.

Weak Memory Models and Tapir

Tapir's task-parallel model, with serial semantics, helps ensure that **standard optimizations** are **legal**.

C11 optimization example,
written in Cilk pseudocode

```
cilk_spawn { a = 1; }

cilk_spawn {
  if (x.load(RLX))
    if (a)
      y.store(1, RLX);
}

cilk_spawn {
  if (y.load(RLX))
    x.store(1, RLX);
}
```

The serial semantics of `cilk_spawn` ensures that sequentialization is **always allowed.**

A Sweet Spot for Compiler Optimizations

- When optimizing **across threads**, standard compiler optimizations are **not always legal**.
- By enabling parallelism for a **single thread of control**, Tapir's model is **amenable** to standard compiler optimizations.
- **Vectorization** is another example of where compilers use parallelism to speed up a single thread of control.