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

William S. Moses

IBM PL Day 2016

December 5, 2016

Joint work with
Tao B. Schardl and Charles E. Leiserson

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: 0.396 s

Example: Normalizing a Vector in Parallel

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: $T_S = 0.396$ s

Running time on 18 cores: $T_{18} = 167.731$ s

Running time on 1 core: $T_1 = 2316.063$ s

Terrible ***work efficiency***:
$T_S/T_1 = 0.396/2316$
$\sim 1/5800$

Example: Normalizing a Vector in Parallel

Affects Cilk and other frameworks too!

```
__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);
}
```

Tapir: Task-Based Parallel IR

- Tapir is an extension to LLVM that embeds fork-join parallelism in the Intermediate Representation (IR).
- Tapir allows standard compiler optimizations to operate across parallel control constructs.
- Tapir/LLVM required only about 5000 lines of code compared with the 3-million lines in the LLVM codebase.

Outline

- Why Compilers Optimize Parallel Constructs Poorly
- Old Idea: Parallel IR
- Tapir: A New Twist on an Old Idea
- Evaluation of Tapir
- Conclusion

Outline

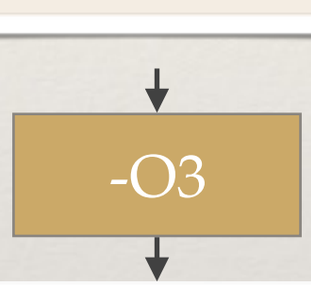
- Why Compilers Optimize Parallel Constructs Poorly
- Old Idea: Parallel IR
- Tapir: A New Twist on an Old Idea
- Evaluation of Tapir
- Conclusion

The Compilation Pipeline

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);
}
```



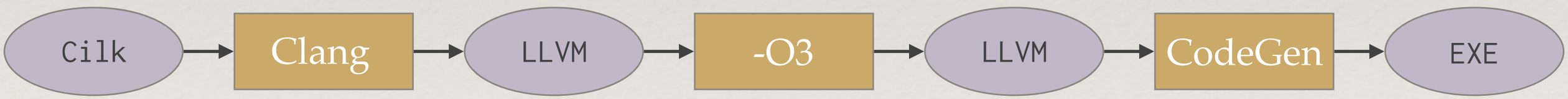

```
__attribute__((const)) double norm(const double *A, int n);

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

LLVM pipeline

CilkPlus/LLVM pipeline



Front-end translates 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);
}
```


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

void normalize(double *restrict out, const double *rest
  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;
}
```

Clang

```
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.

Outline

- Why Compilers Optimize Parallel Constructs Poorly
- Old Idea: Parallel IR
- Tapir: A New Twist on an Old Idea
- Evaluation of Tapir
- Conclusion

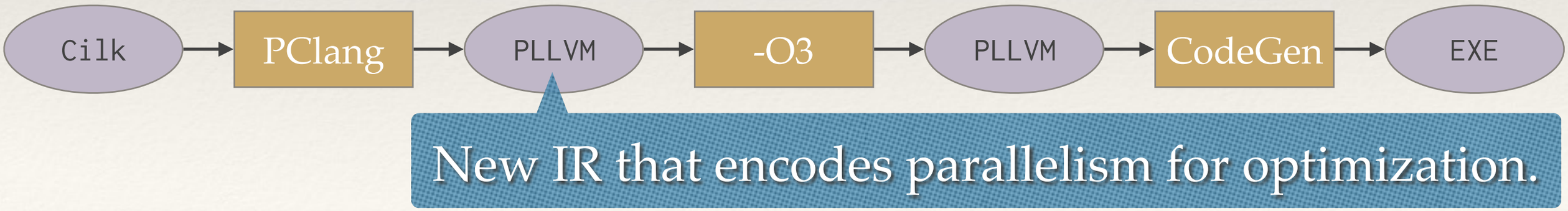
A Parallel IR

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

LLVM pipeline

CilkPlus/LLVM pipeline

A better compilation pipeline



Previous Attempts at Parallel IR's

- Parallel precedence graphs [SW91, SHW93]
- Parallel flow graphs [SG91, GS93]
- Concurrent SSA [LMP97, NUS98]
- HPIR [ZS11, BZS13]
- SPIRE [KJAI12]
- INSPIRE [JPTKF13]
- Parallel program graphs [SS94, S98]
- "[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

A Hard Problem

From the llvm-dev mailing list:

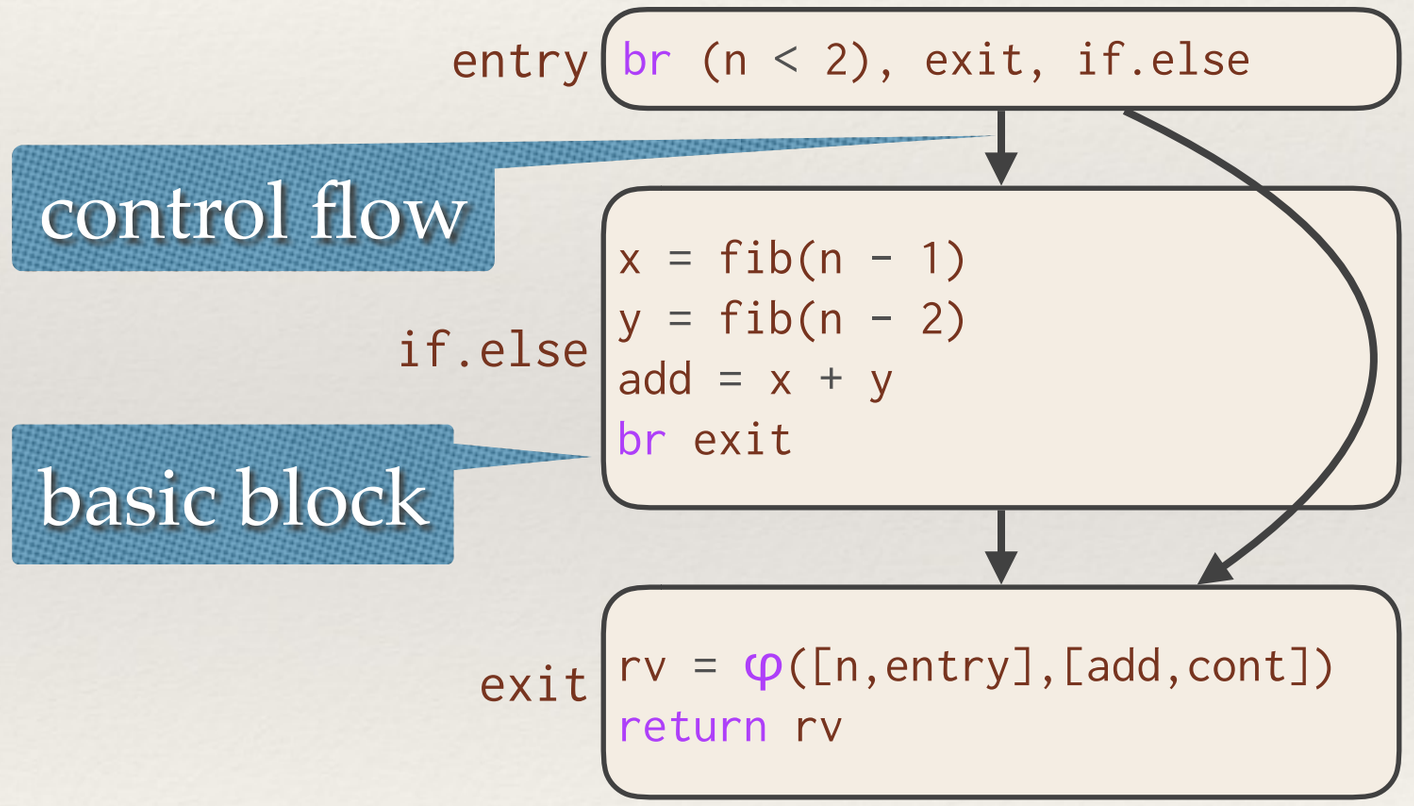
- "[I]ntroducing [parallelism] into a so far 'sequential' IR will cause severe breakage and headaches."
- "[P]arallelism is invasive by nature and would have to influence most optimizations."
- "[It] is not an easy problem."
- "[D]efining a parallel IR (with first class parallelism) is a research topic…"

Source: http://lists.llvm.org/pipermail/llvm-dev/2015-March/083134.html

Background: LLVM IR

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

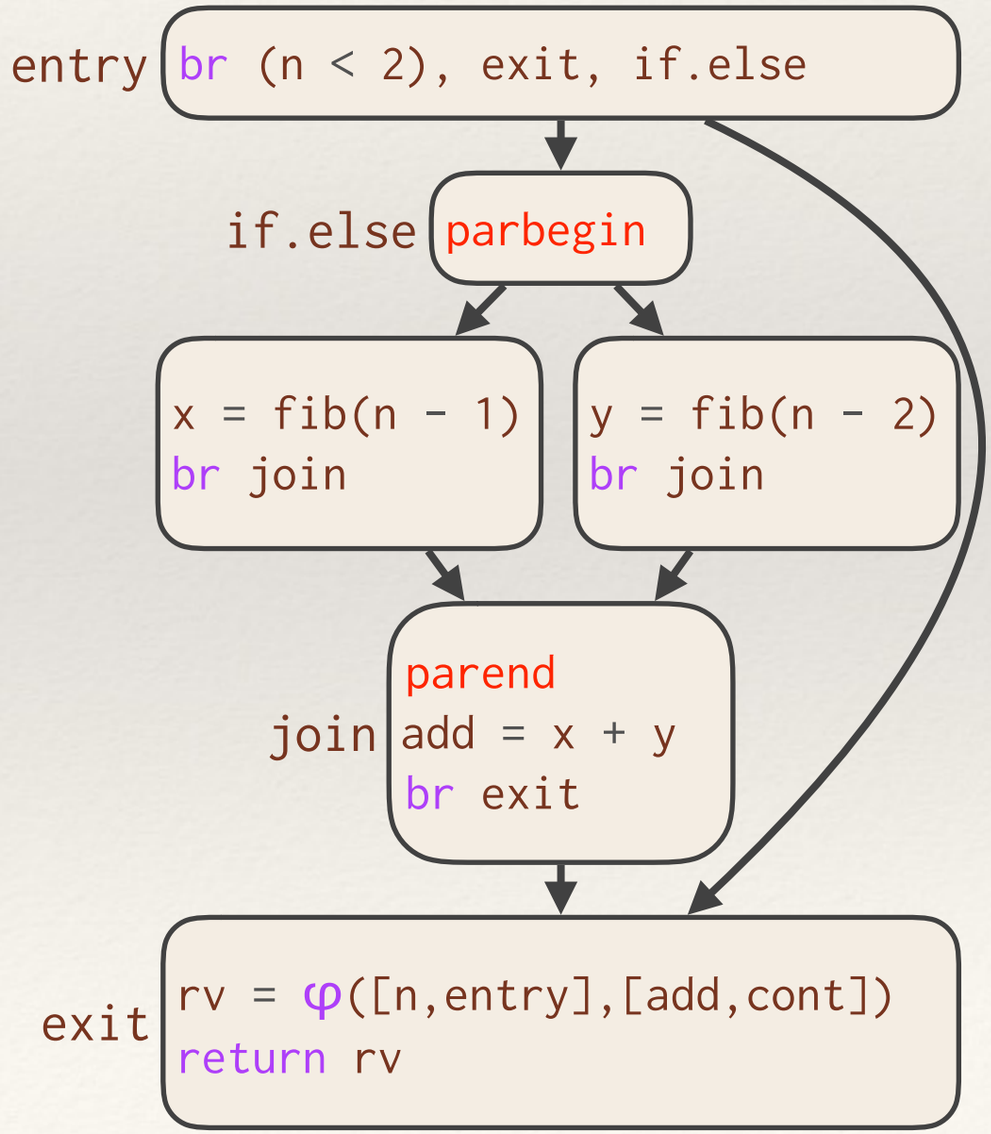
```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  x = fib(n - 1);
  y = fib(n - 2);
  return x + y;
}
```



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;
}
```



Typical Issues with Parallel IR's

- Parallel IR is incompatible with existing optimizations or analyses for serial code.
- Parallel IR requires many changes to the compiler.
- Parallel IR offers minimal benefits to optimization.
- Parallel IR is language specific.
- For LLVM, symmetric modeling violates the Linear Assumption: that each block is entered by one predecessor.

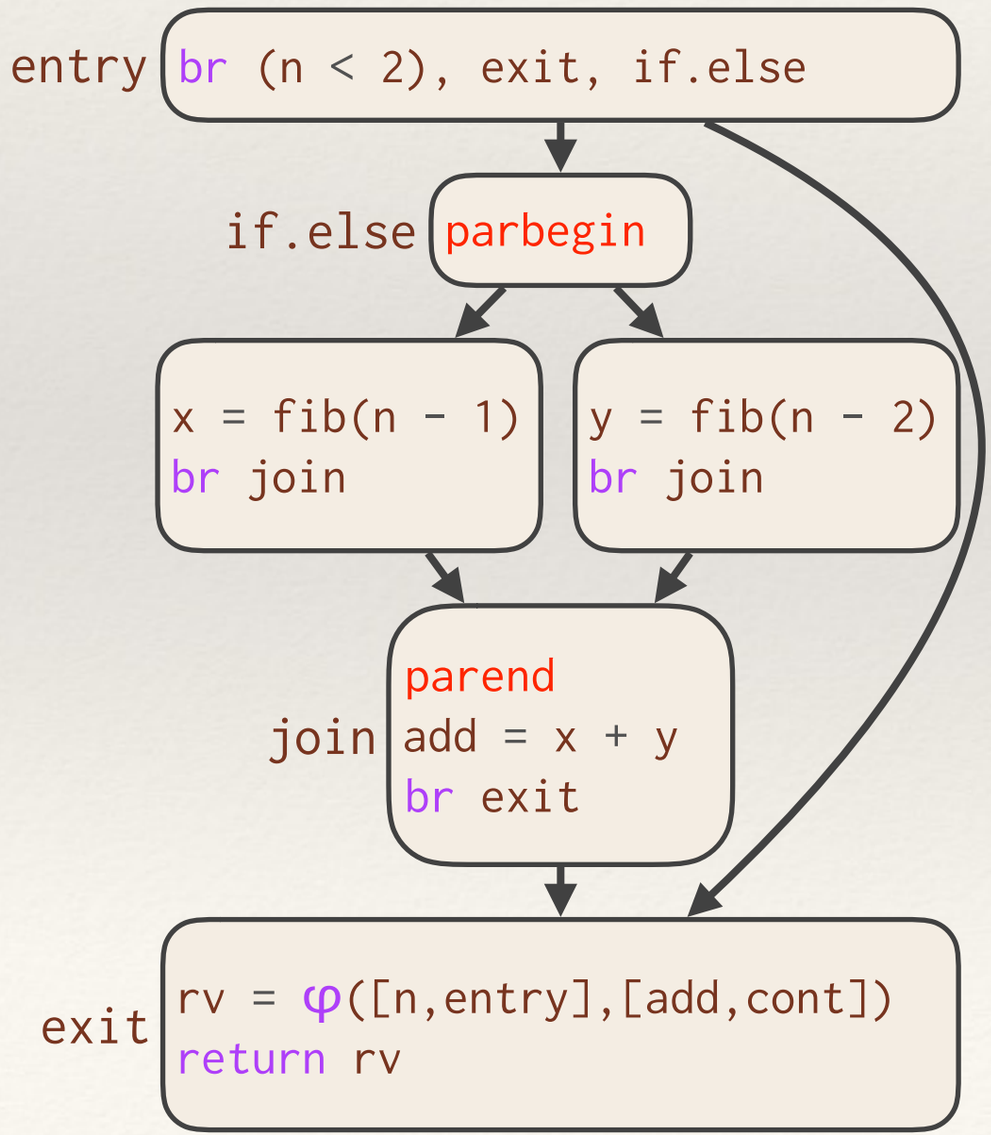
Outline

- Why Compilers Optimize Parallel Constructs Poorly
- Old Idea: Parallel IR
- Tapir: A New Twist On an Old Idea
- Evaluation of Tapir
- Conclusion

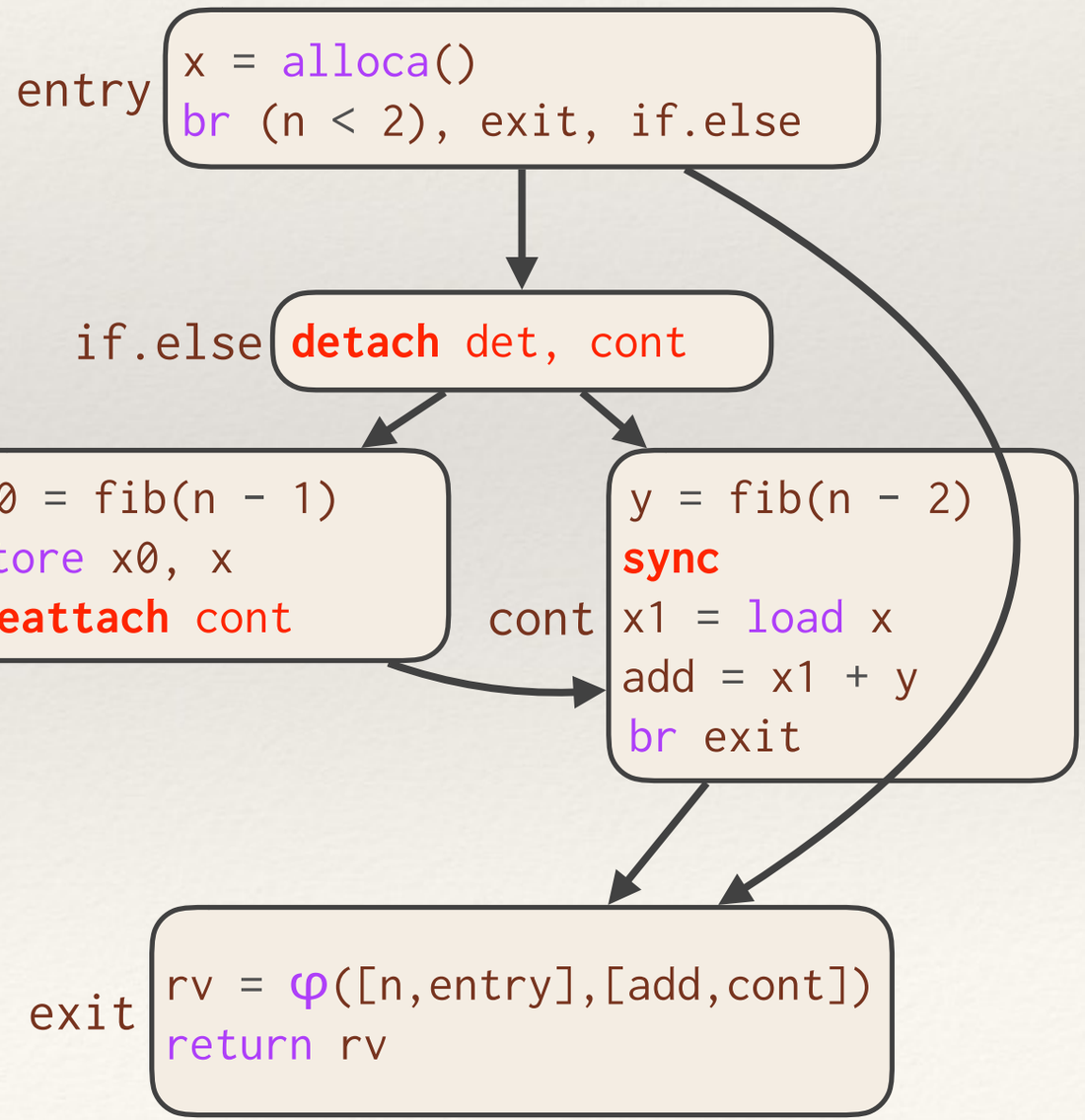
Tapir's CFG

Tapir extends LLVM IR with **three instructions** that model parallel tasks **asymmetrically**.

Symmetric CFG



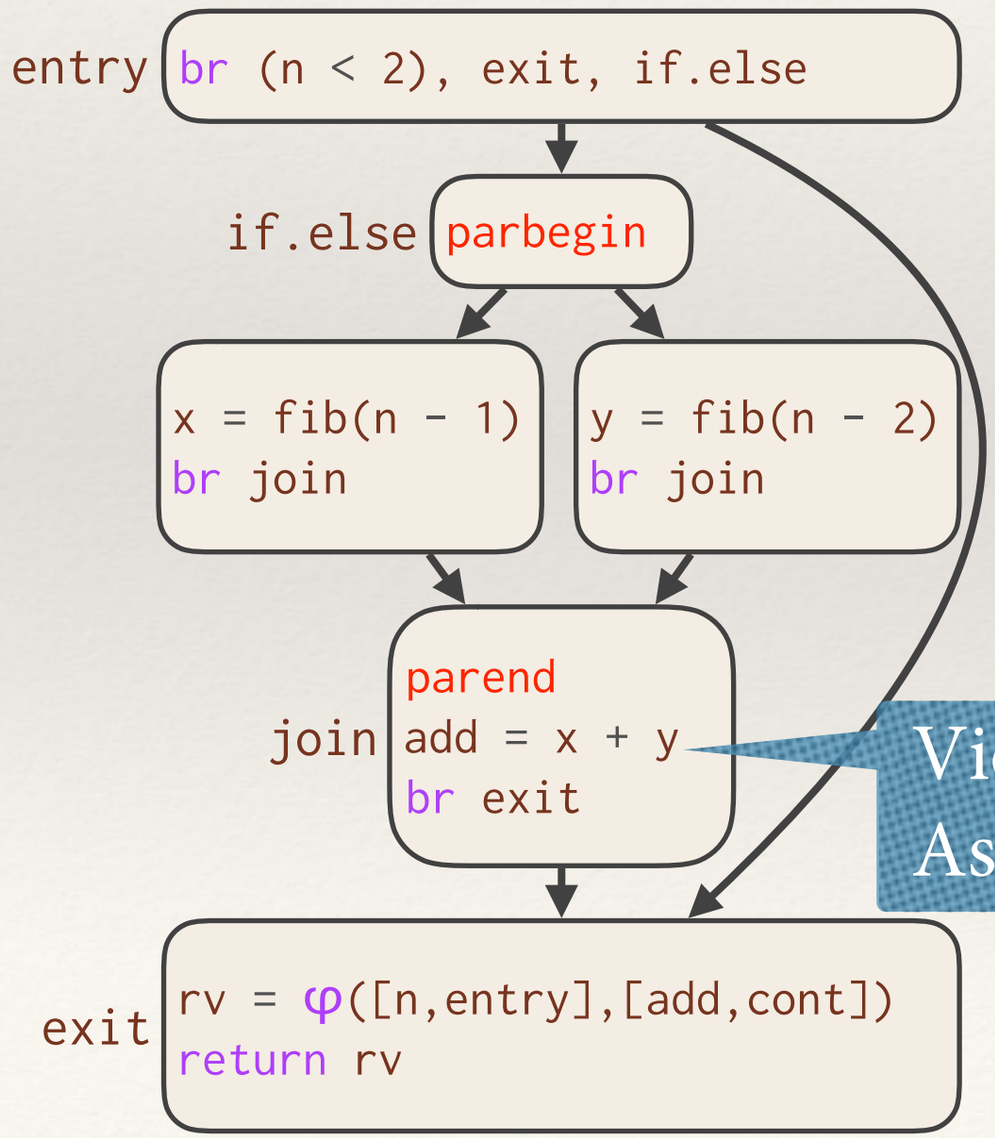
Tapir CFG



Tapir's CFG

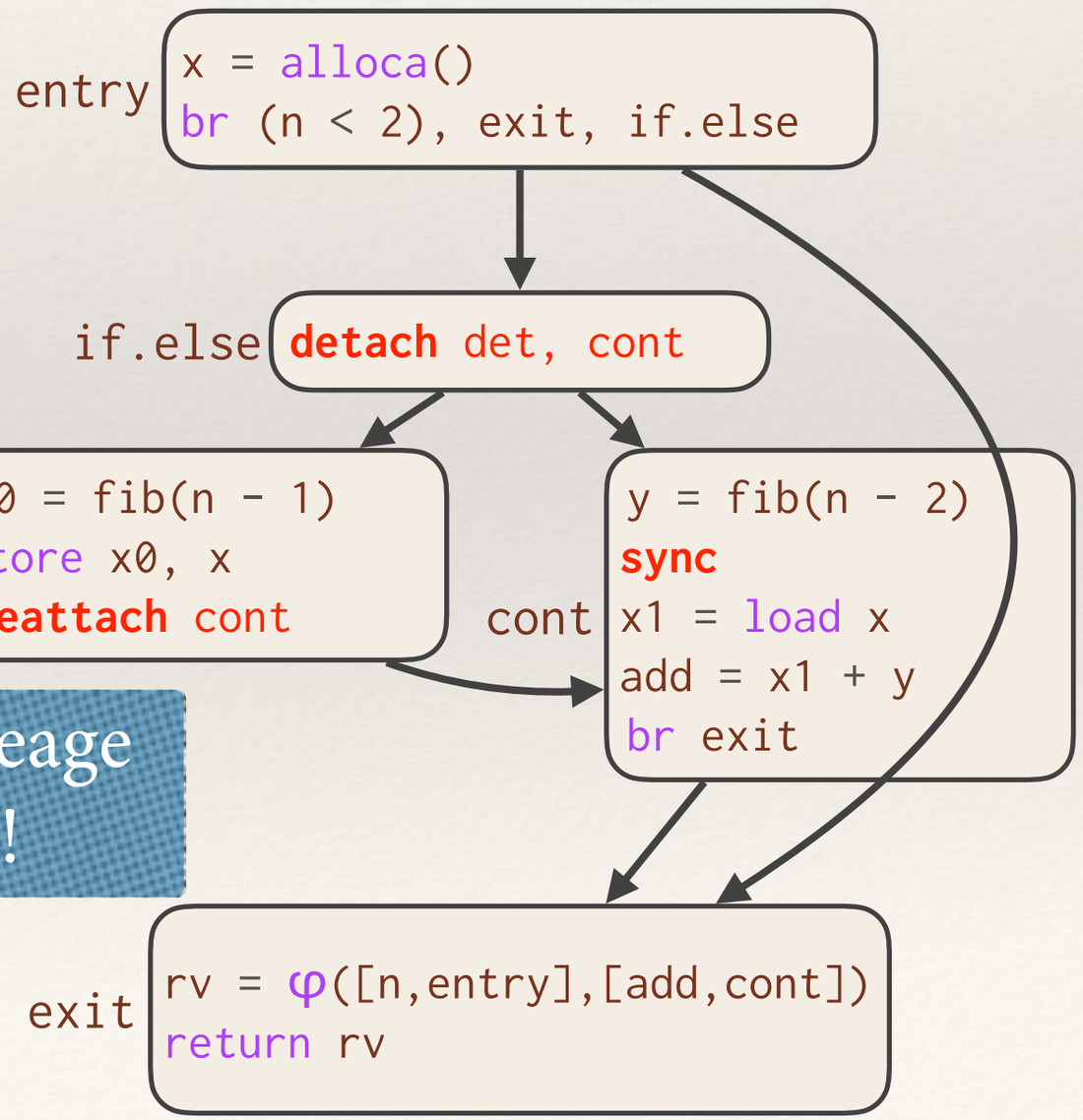
Tapir extends LLVM IR with **three instructions** that model parallel tasks **asymmetrically**.

Symmetric CFG



Violates Lineage Assumption!

Tapir CFG



Tapir's Pipeline

CilkPlus/LLVM pipeline

Tapir/LLVM pipeline (simplified)

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

With few changes, LLVM's existing optimization passes can optimize across parallel control flow.

Compiler Optimizations

What does Tapir do to **adapt existing optimizations**?

- Common-subexpression elimination: no change
- Loop-invariant-code motion: minor change
- Tail-recursion elimination: minor change

Tapir also enables **new parallel optimizations**, such as:

- Unnecessary-synchronization elimination
- Puny-task elimination
- Parallel-loop scheduling (new pass combined with existing unrolling and vectorization passes)

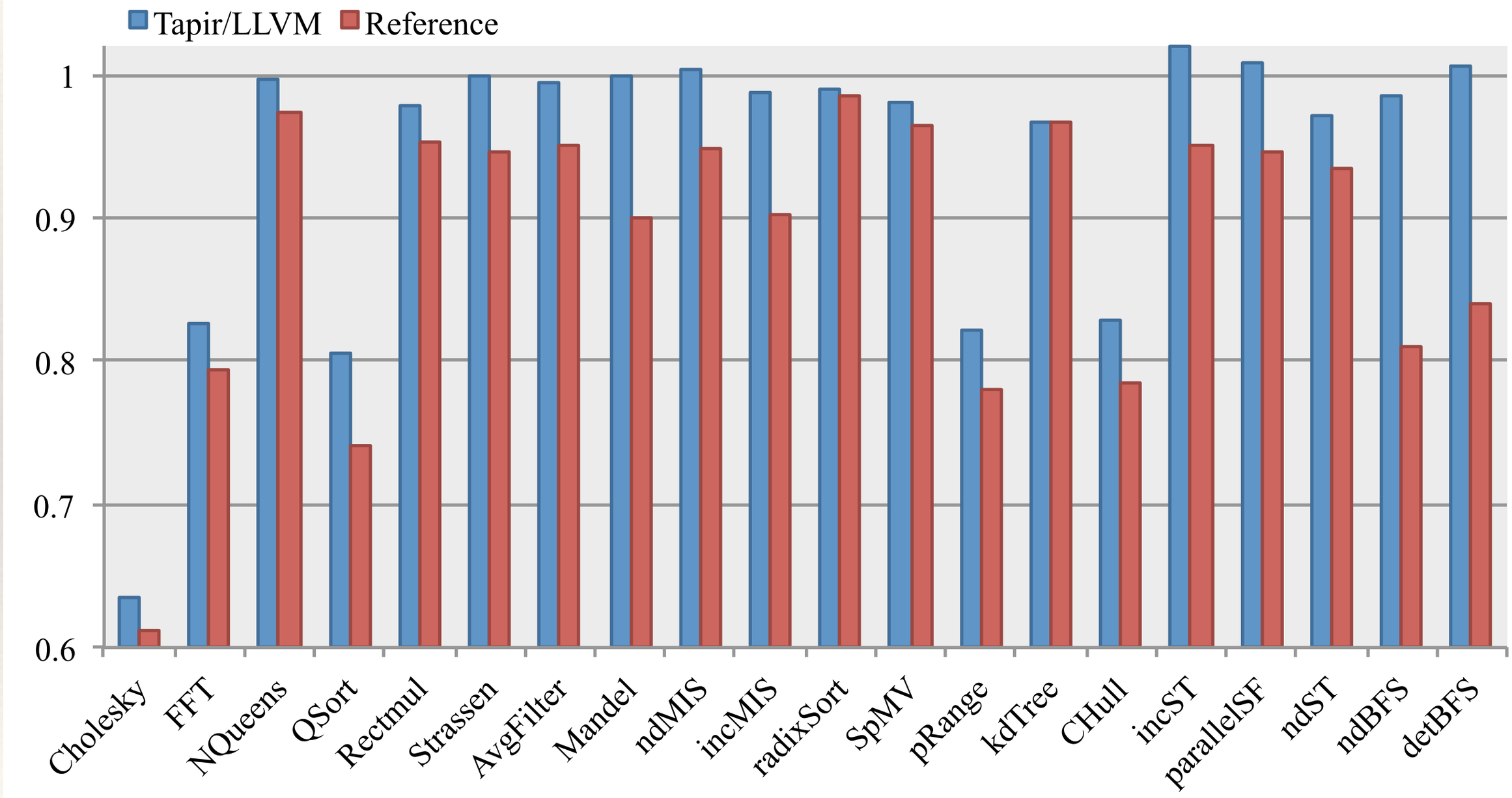
Outline

- Why Compilers Optimize Parallel Constructs Poorly
- Old Idea: Parallel IR
- Tapir and Why It Works
- Evaluation of Tapir
- Conclusion

Code Complexity of Tapir/LLVM

Compiler component	*LLVM 3.8 (lines)*	*Tapir/LLVM (lines)*	
Instructions	148,588	900	1,888
Memory behavior	10,549	588	
Optimizations	140,842	255	
Code generation	205,378	145	
Parallelism lowering	0	1,903	
New parallel optimizations	0	1,332	
Other	2,854,566	0	
Total	3,359,893	5,123	

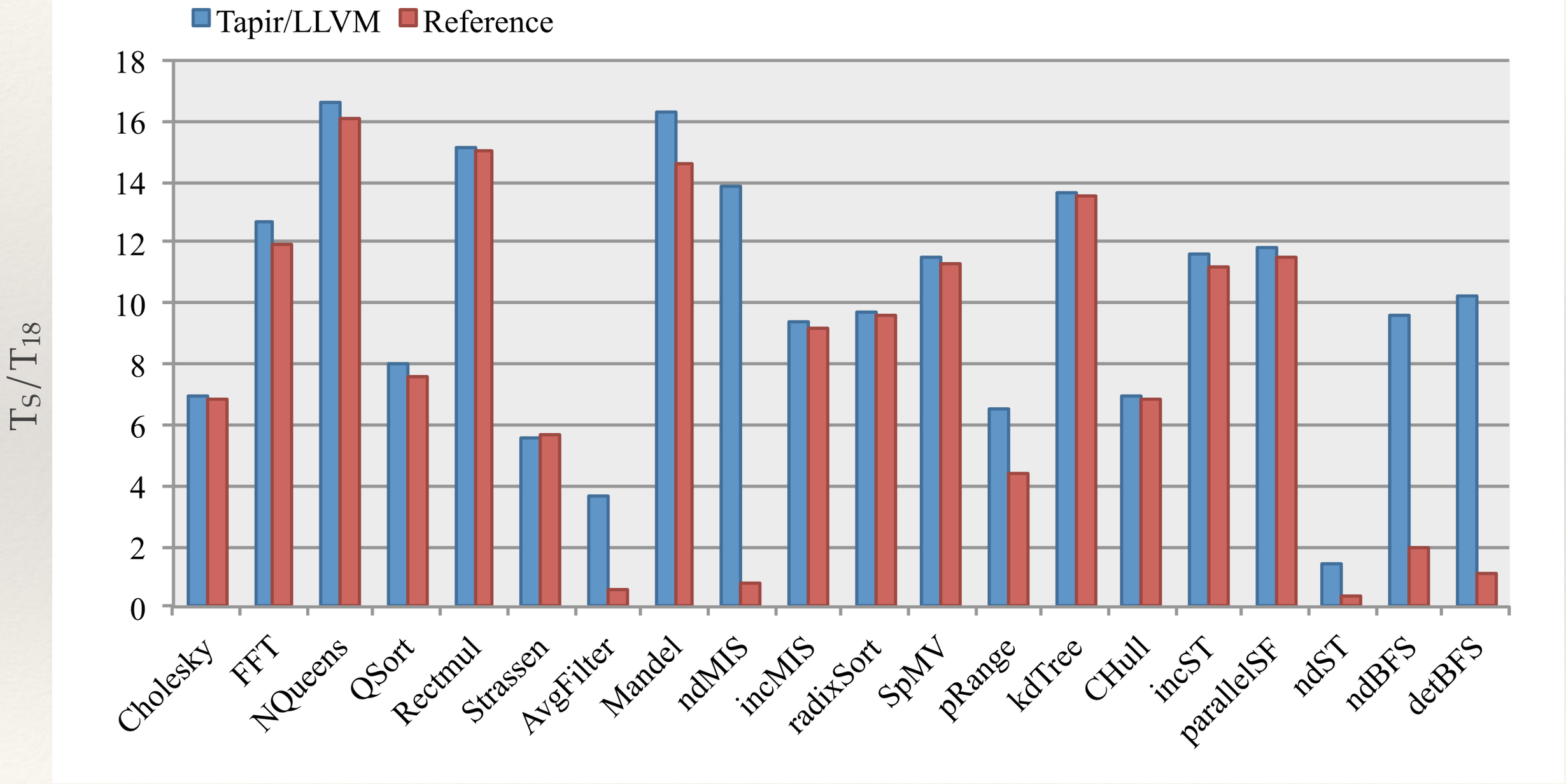
Work-Efficiency Improvement



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

Preliminary results

Speedup Improvement



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

Preliminary results

Outline

- Why Compilers Optimize Parallel Constructs Poorly
- Old Idea: Parallel IR
- Tapir and Why It Works
- Evaluation of Tapir
- Conclusion

Normalizing 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 compiled with LLVM: $T_S = 0.397$ s

Compiled with Tapir, running time on 1 core: $T_1 = 0.400$ s

Compiled with Tapir, running time on 18 cores: $T_{18} = 0.157$ s

Great work efficiency:
$T_S/T_1 \sim 0.990$

Status of Tapir

- We implemented Tapir in LLVM, along with a prototype Cilk front-end and a pass for lowering Tapir to Cilk runtime calls.
- Tapir is currently in use by over 120 MIT students.
- Out Tapir implementation appears to exhibit **fewer bugs** than GCC, ICC, or CilkPlus/LLVM when compiling Cilk codes.
- We have a provably good **determinacy-race detector** for Tapir programs, which we used to debug code transformations.
- We're continuing to explore new optimizations.
- Try Tapir yourself!
  Email me at wmoses@mit.edu