Leveraging LLVM to Optimize Parallel Programs

William S. Moses

2017 US LLVM Developers Meeting
October 18, 2017

CSAIL
MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

Dougie Kogut

Charles E. Leiserson

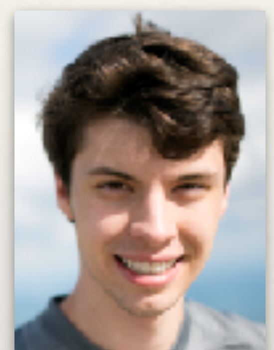
Jiahao Li

Tao B. Schardl

Bojan Serafimov

George Stelle

Running LLVM Optimizations on Parallel Code

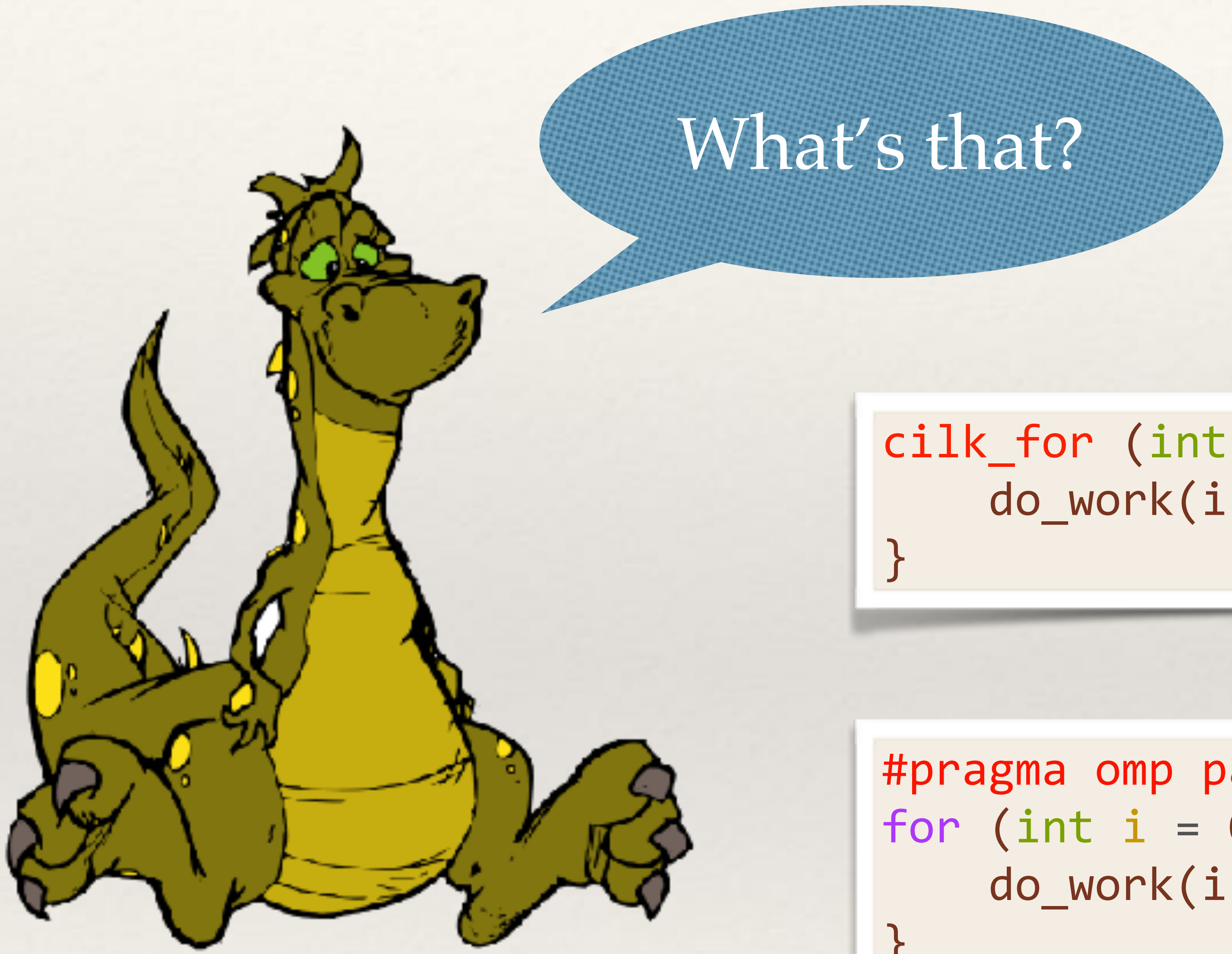


Tao B. Schardl

William S. Moses

Charles E. Leiserson

Compilers Don't Understand Parallel Code




```
cilk_for (int i = 0; i < n; ++i) {
    do_work(i);
}
```

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    do_work(i);
}
```

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.

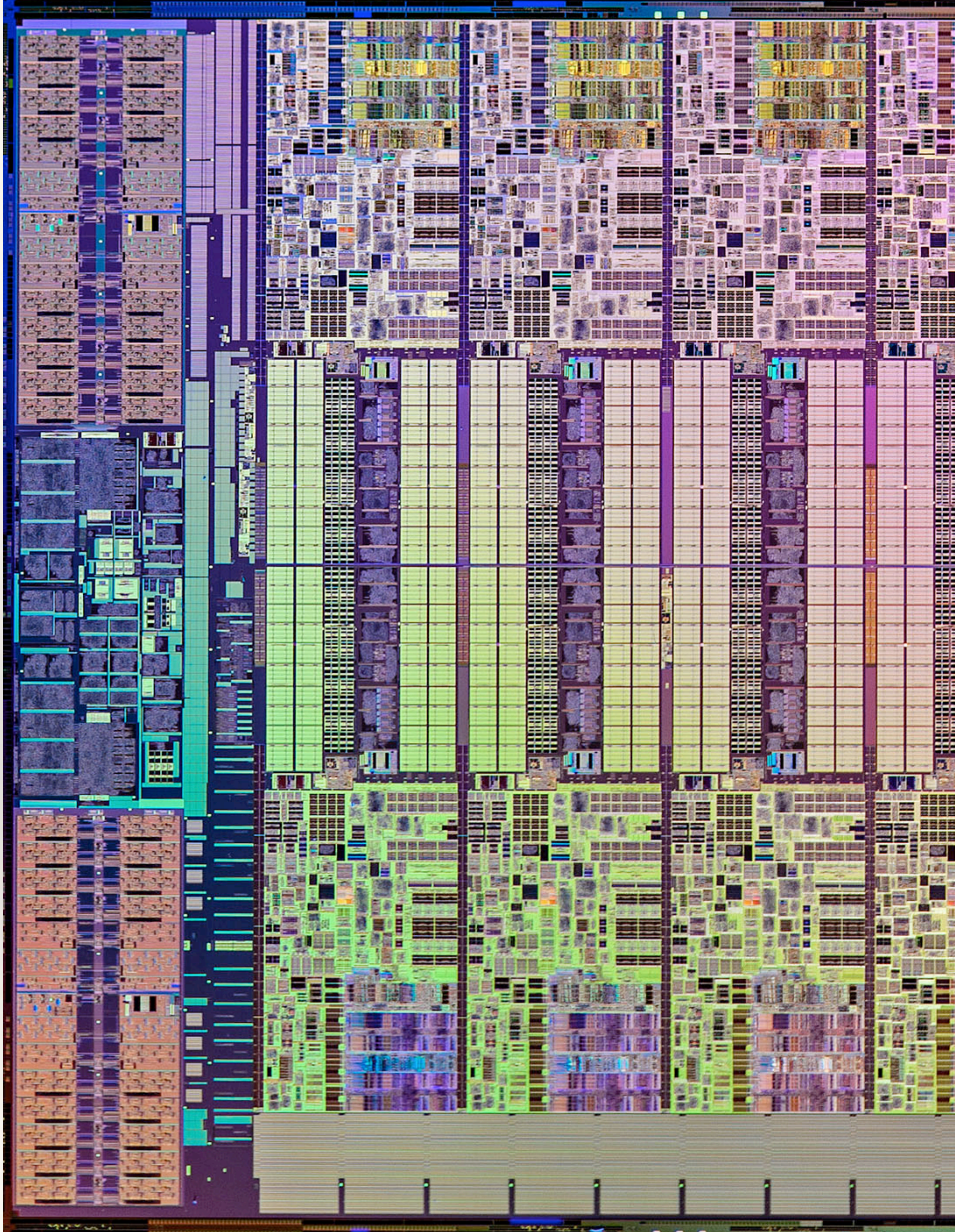
Example: Normalizing a Vector

```
__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.

Running time: 0.312 s

Idea: Run in Parallel!



Example: Normalizing a Vector in Parallel

```
__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.

Original serial running time: $T_S = 0.312$ s

Example: Normalizing a Vector in Parallel

```
__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.

Original serial running time: $T_S = 0.312$ s

18-core running time: 180.657s

Example: Normalizing a Vector in Parallel

```
__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.

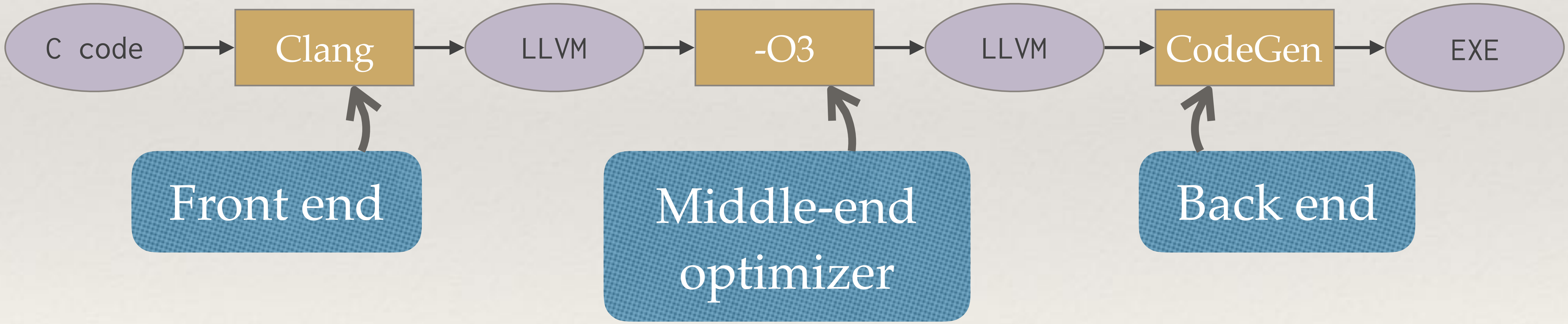
Original serial running time: $T_S = 0.312$ s

18-core running time: 180.657s

1-core running time: 2600.287s

What happened?

The LLVM Compilation Pipeline



Effect of Compiling Serial Code

```c
__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);
}
```

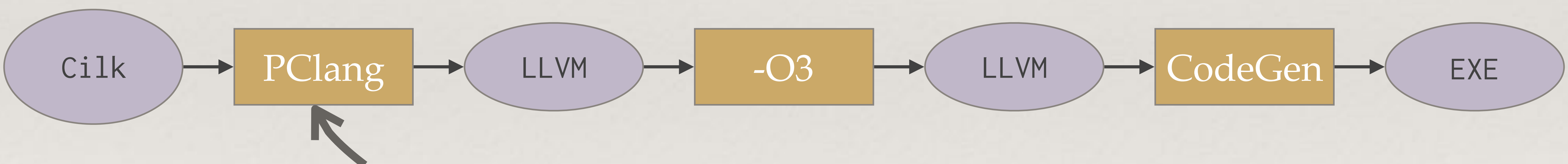
-O3

```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;
}
```

Compiling Parallel Code

LLVM pipeline

Cilk Plus/LLVM pipeline



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.

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);
}
```

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

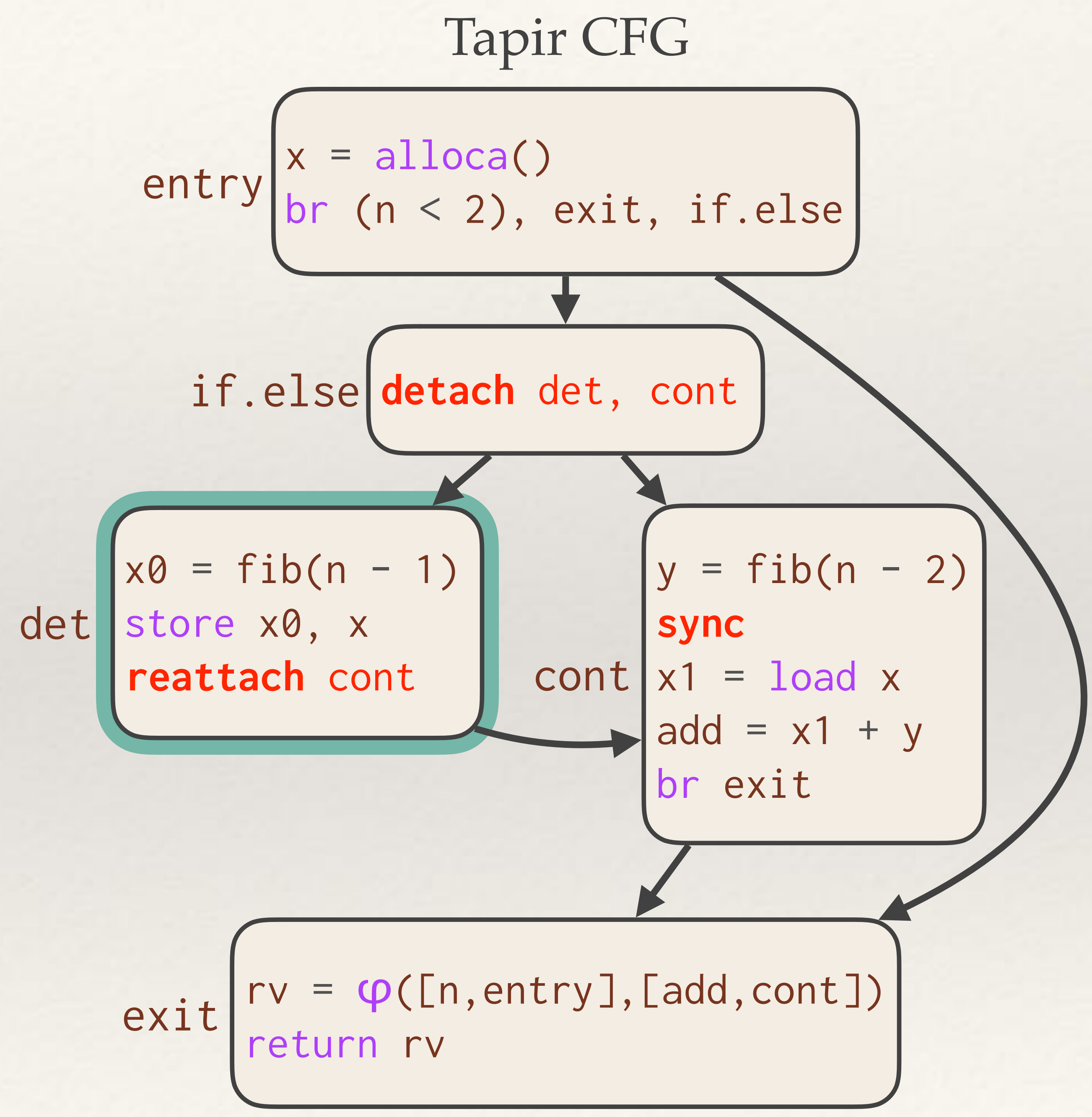
Compiled with Tapir/LLVM, running time on 1 core: $T_1 = 0.321$ s

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

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

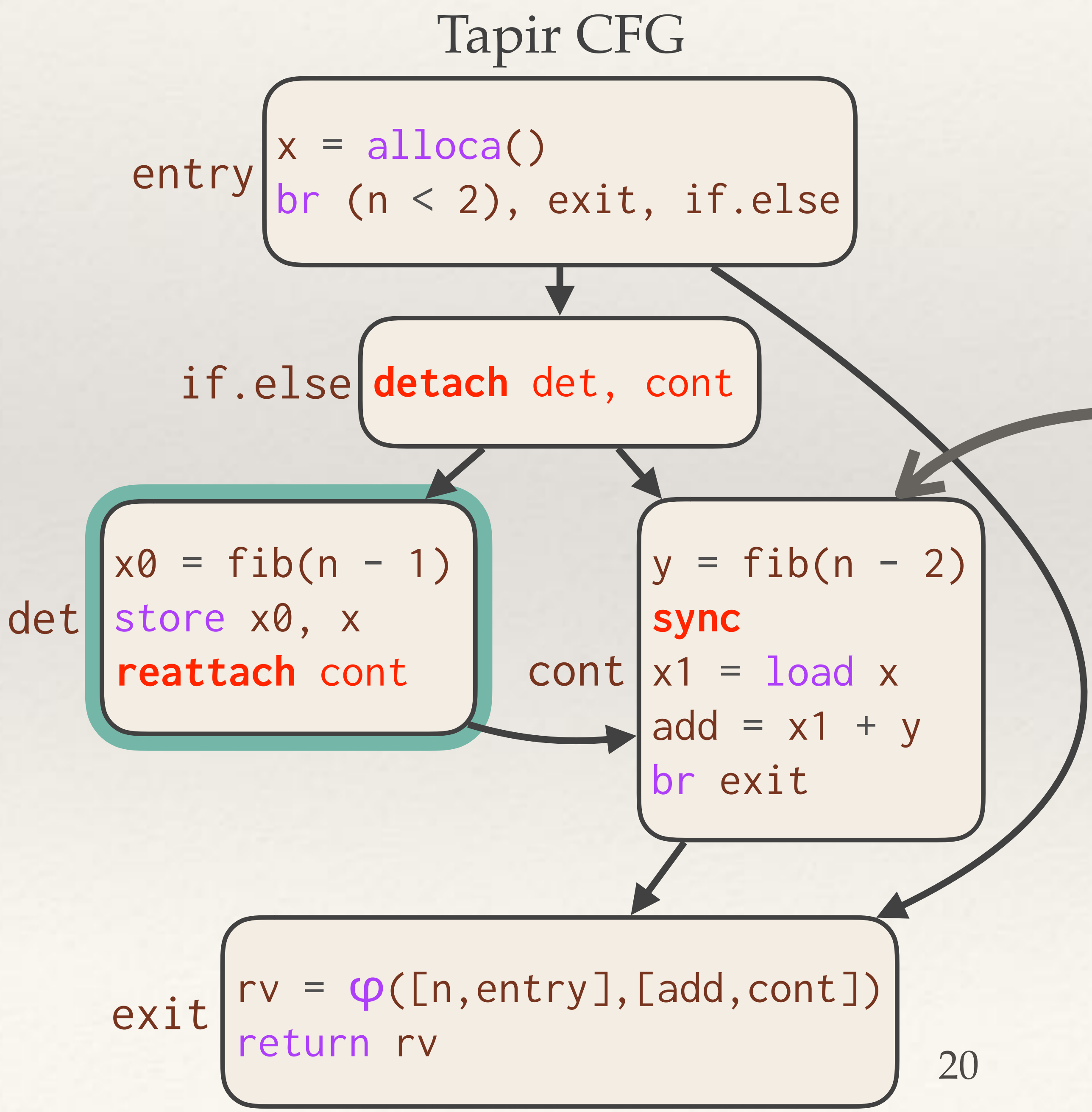
Tapir Semantics

- Tapir introduces three new opcodes into LLVM's IR: detach, reattach, and sync.
- The successors of a detach terminator are the detached block and continuation and may run in parallel.
- Execution after a sync ensures that all detached CFG's in scope have completed execution.



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.

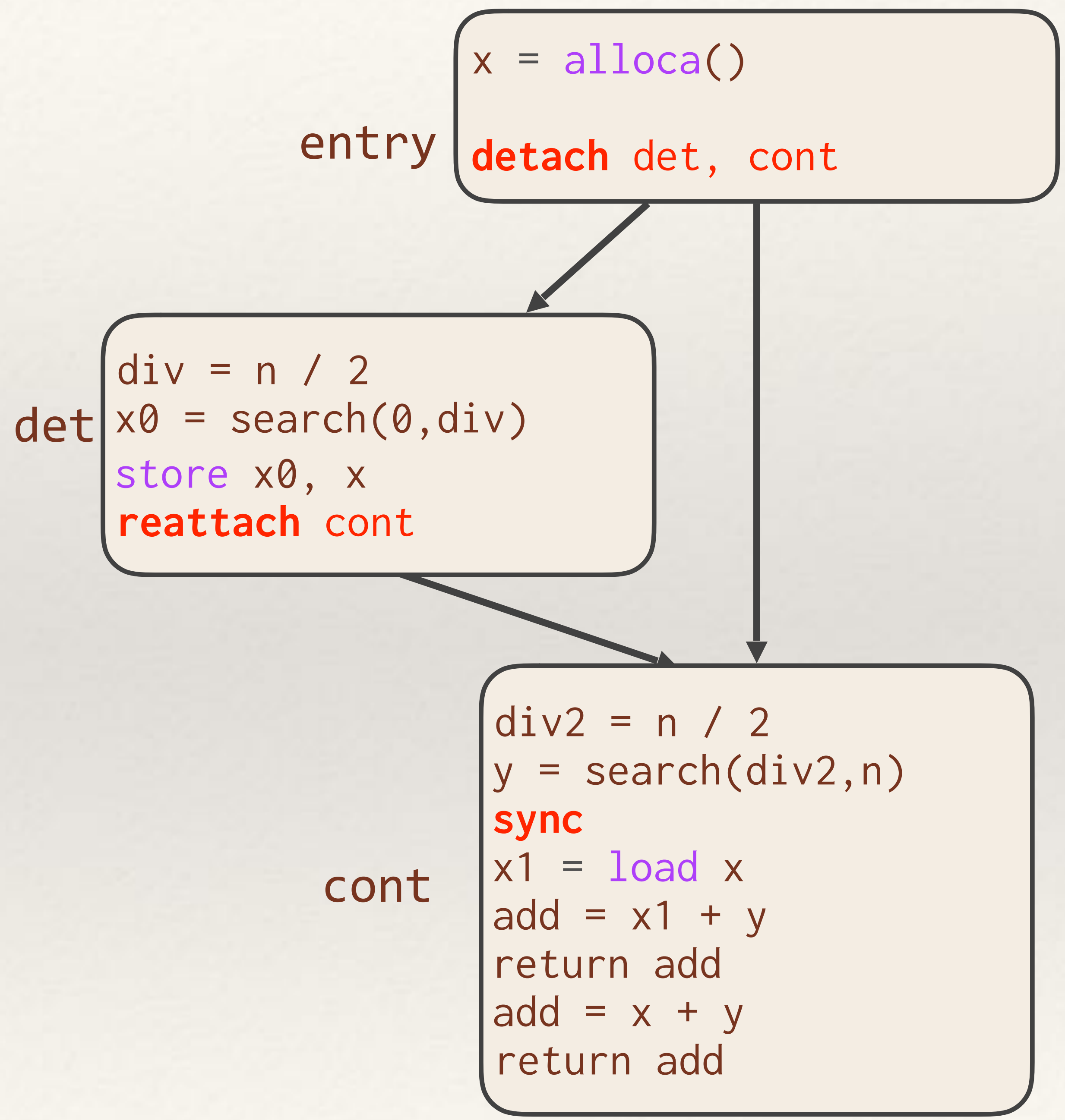


Many parts of the compiler can apply **standard implicit assumptions** of the CFG to this block.

Case Study: Common Subexpression Elimination

- CSE "just works."
- Finding duplicate expressions and condensing them at their lowest common ancestor works fine for **detach** / **reattach**.

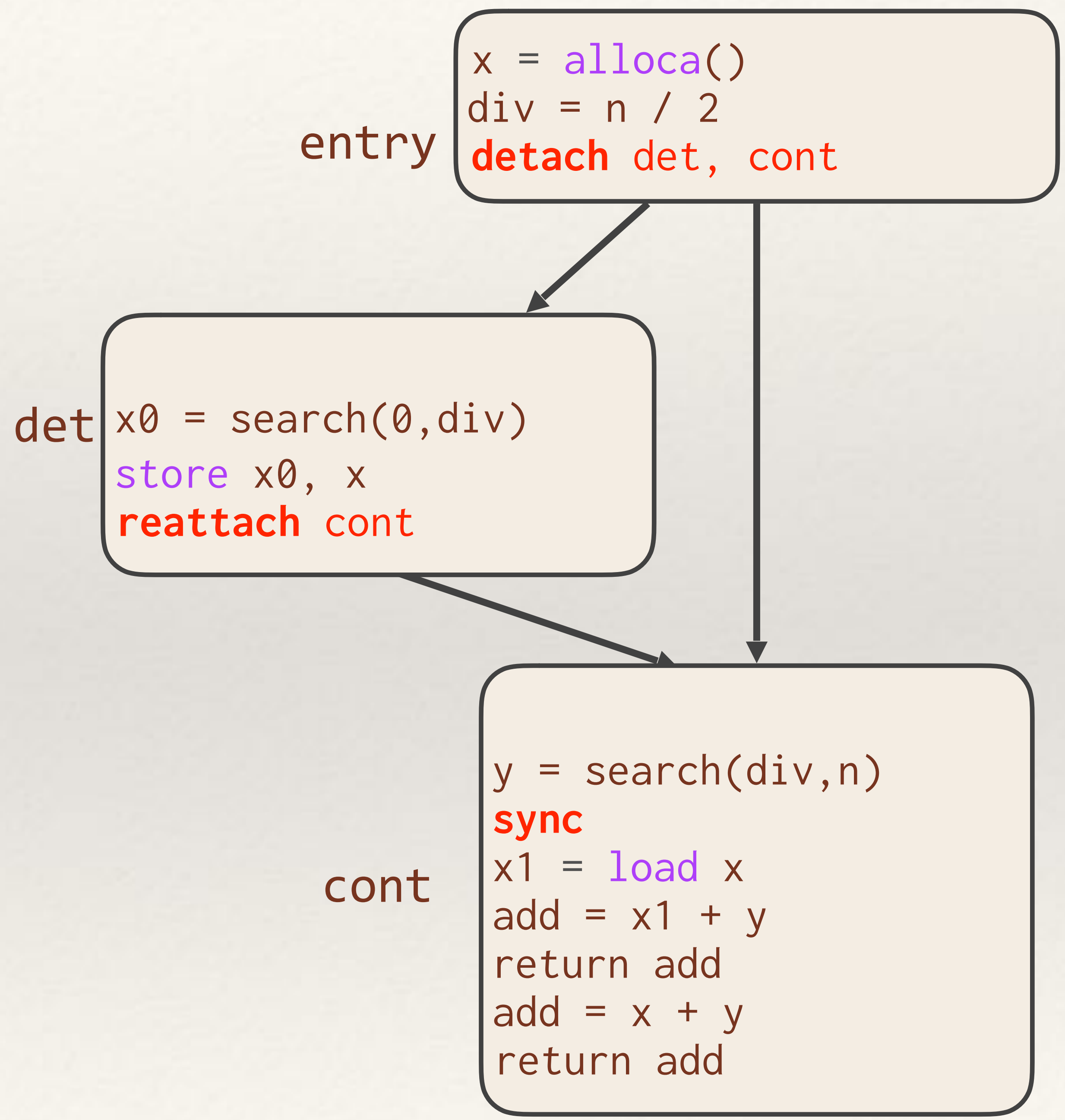
```
void query(int n) {
  int x = detach
          { search(0,n/2); }
  int y = search(n/2,n);
  sync;
 return x + y;
}
```



Case Study: Common Subexpression Elimination

- CSE “just works.”
- Finding duplicate expressions and condensing them at their lowest common ancestor works fine for `detach`/ `reattach`.

```
void query(int n) {
  int x = detach
          { search(0,n/2); }
  int y = search(n/2,n);
  sync;
 return x + y;
}
```



Compiler Analyses and Optimizations

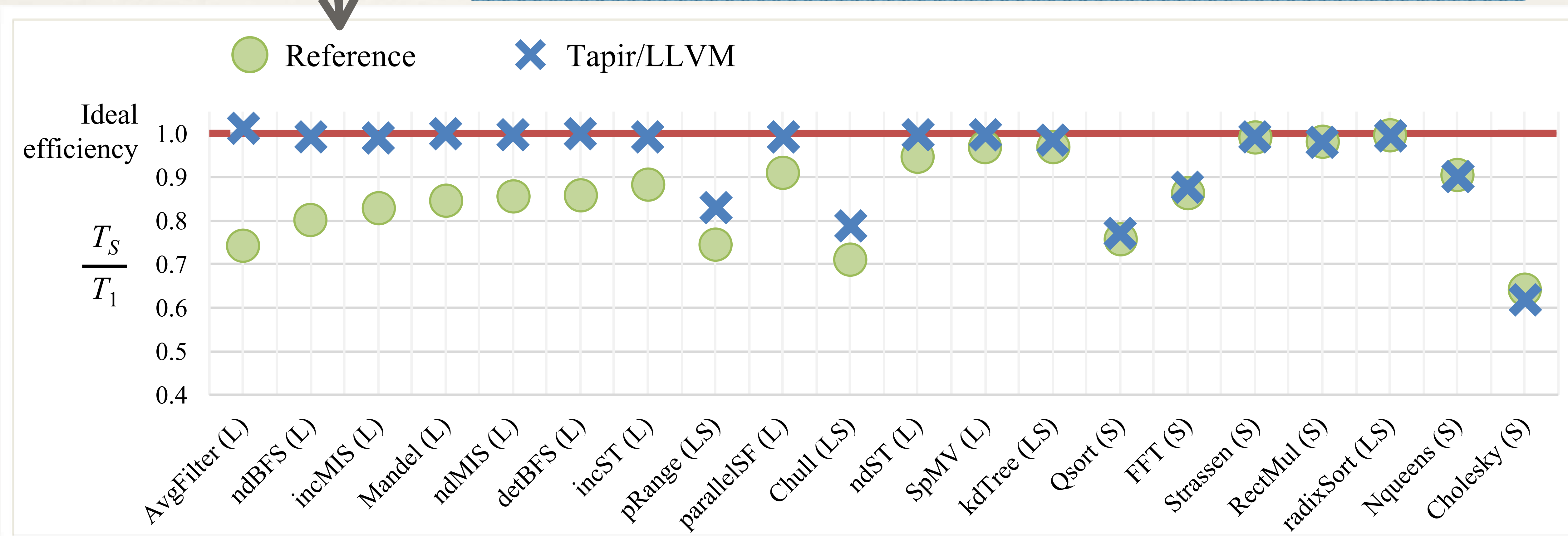
What did we do to **adapt existing analyses and optimizations**?

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

Suite	Benchmark	Description
Cilk	Cholesky	Cholesky decomposition
	FFT	Fast Fourier transform
	NQueens	n-Queens solver
	QSort	Hoare quicksort
	RectMul	Rectangular matrix multiplication
	Strassen	Strassen matrix multiplication
Intel	AvgFilter	Averaging filter on an image
	Mandel	Mandelbrot set computation
PBBS	CHull	Convex hull
	detBFS	BFS, deterministic algorithm
	incMIS	MIS, incremental algorithm
	incST	Spanning tree, incremental algorithm
	kdTree	Performance test of a parallel k-d tree
	ndBFS	BFS, nondeterministic algorithm
	ndMIS	MIS, nondeterministic algorithm
	ndST	Spanning tree, nondeterministic algorithm
	parallelSF	Spanning-forest computation
	pRange	Compute ranges on a parallel suffix array
	radixSort	Radix sort
	SpMV	Sparse matrix-vector multiplication

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, 2.9 GHz, 60 GiB DRAM

What Else Does A Parallel IR Buy Us?

William S. Moses

George Stelle

Jiahao Li

Dougie Kogut

Bojan Serafimov

Parallel-Specific Optimizations

To ensure reasonable performance, parallel frameworks implement parallel-specific optimizations

Example Opt: Coarsening

- Combine detached statements to overcome the overhead of running in parallel

```
void scale(double *restrict A, double s, int n) {
  parallel_for (int i = 0; i < n; i++) {
    A[i] *= s;
} }
```

↓

```
void scale(double *restrict A, double s, int n) {
  parallel_for (int i = 0; i < n; i+=4) {
    for (int i2 = 0; i2 < 4; i2++) {
      A[i+i2] *= s;
} } }
```

Example Opt: Coarsening

- Combine detached statements to overcome the overhead of running in parallel

```
void scale(double *restrict A, double s, int n) {
  parallel_for (int i = 0; i < n; i++) {
    A[i] *= s;
} }
```

```
void scale(double *restrict A, double s, int n) {
  parallel_for (int i = 0; i < n; i+=4) {
    for (int i2 = 0; i2 < 4; i2++) {
      A[i+i2] *= s;
} } }
```

~4x

Example Opt: Task Elimination

- If you have a detached task immediately followed by a sync, remove the detach.

```
void foo() {
  detach bar();
  detach baz();
  sync;
}
```

→

```
void foo() {
  detach bar();
         baz();
  sync;
}
```

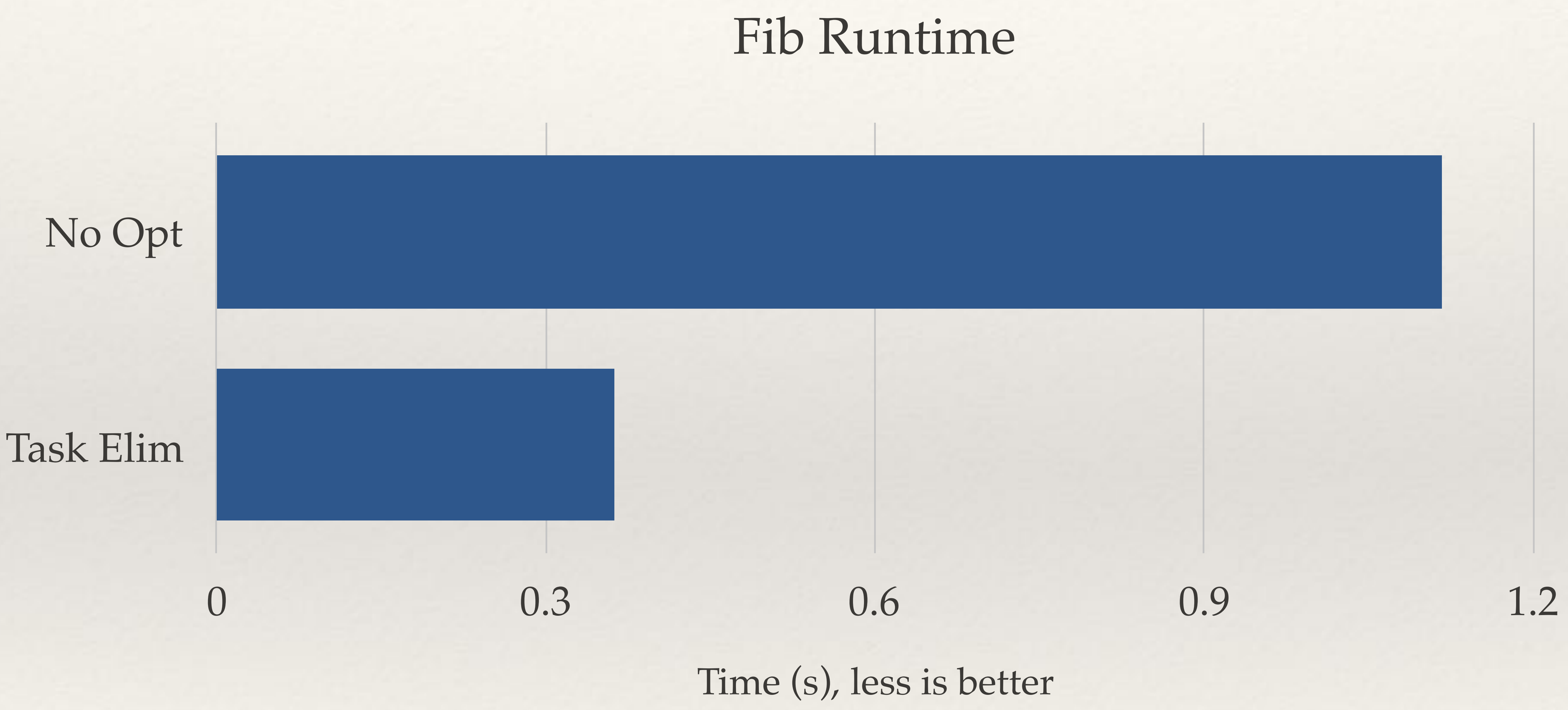
Sounds trivial, but especially useful for OpenMP!

Example Opt: Task Elimination

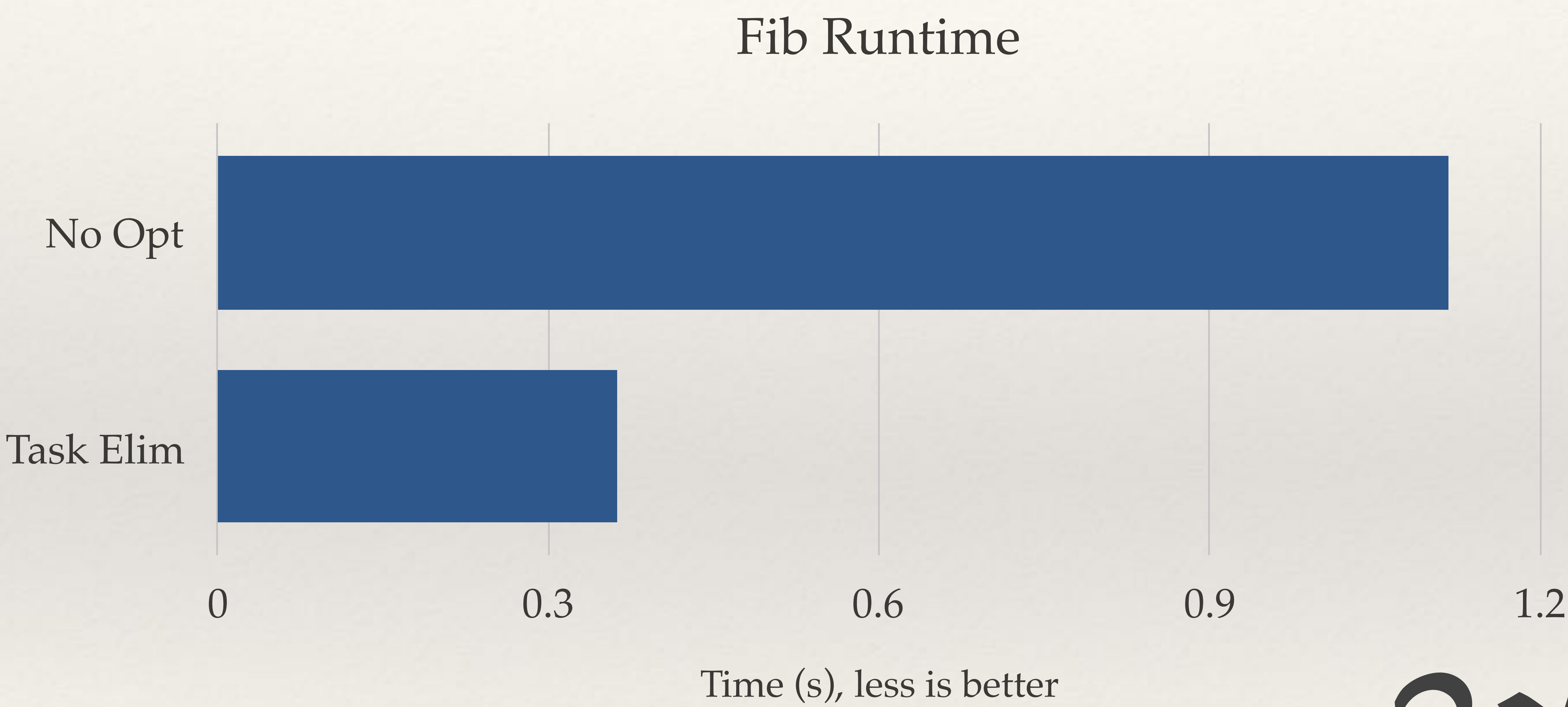
```
void fib(int n) {
  if (n < 2) return n;
  int x, y;
  #pragma omp task shared(x)
  x = fib(n-1);
  #pragma omp task shared(y)
  y = fib(n-2);
  #pragma omp taskwait
  return x+y;
}
```

Linguistically OpenMP tasks encourages users to write code that needs this optimization!

Case Study: Task Elimination



Case Study: Task Elimination



~3x

Parallel Optimizations Today

- Every parallel framework today is independent, requiring large amounts of code duplication.
- Duplication from framework to framework
- Duplication from low level (i.e. LICM in LLVM) to high level

Parallel Pipeline Today

Cilk Frontend	OpenMP Frontend	Halide Frontend	Weld Frontend
Cilk Parallel Optimizations (shrink wrap)	OMP Parallel Optimizations (strip mine)	Halide Parallel Optimizations (scheduling)	Weld Parallel Optimizations, LICM
LLVM w/ Cilk Runtime Calls	LLVM w/ OpenMP Runtime Calls	LLVM w/ Halide Runtime Calls	LLVM w/ Weld Runtime Calls
Cilk Runtime	OpenMP Runtime	Halide Runtime	Weld Runtime

Rhino: The Parallel Compiler Dream

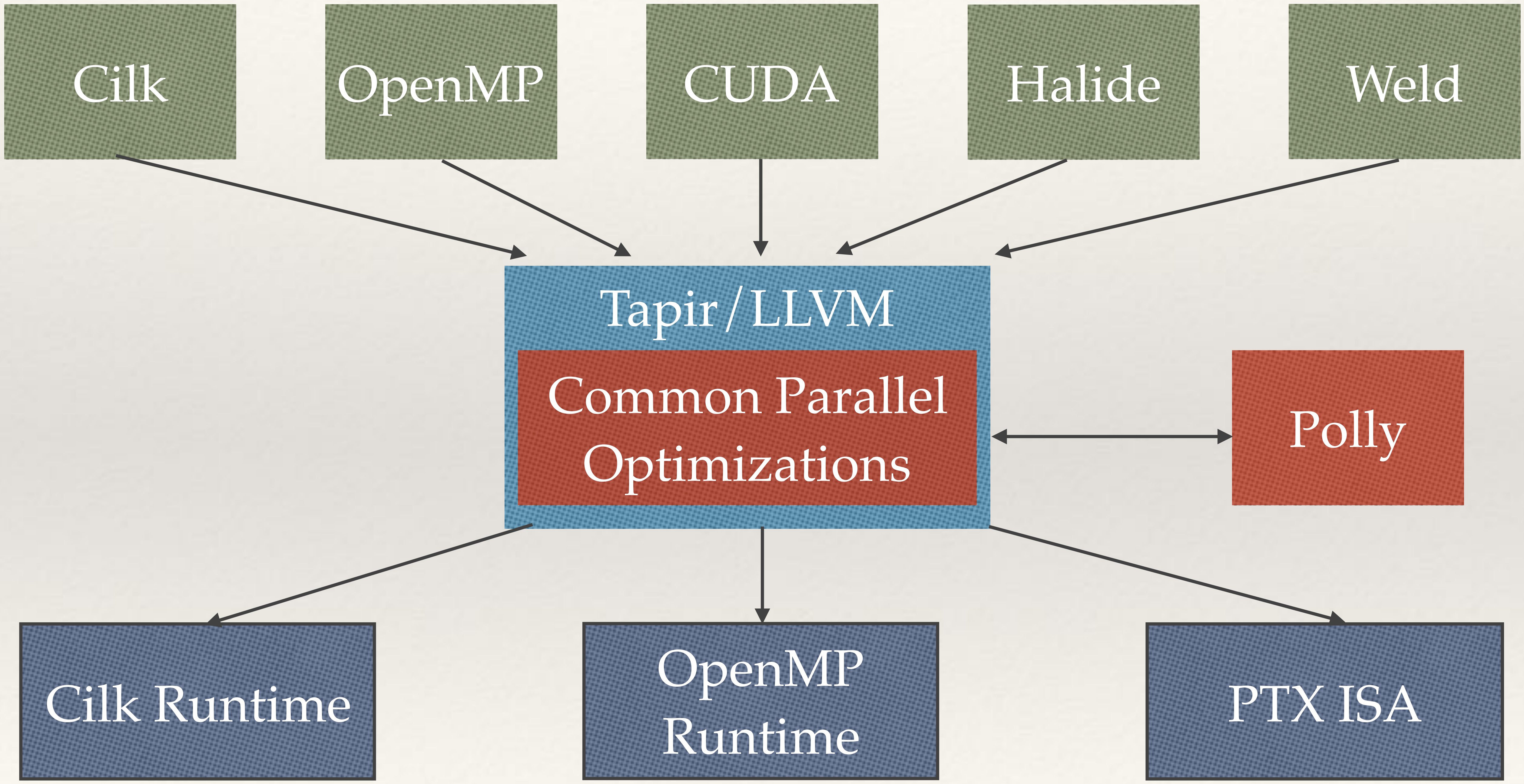
Multiple Parallel "Frontends"

Common Parallel Optimizations

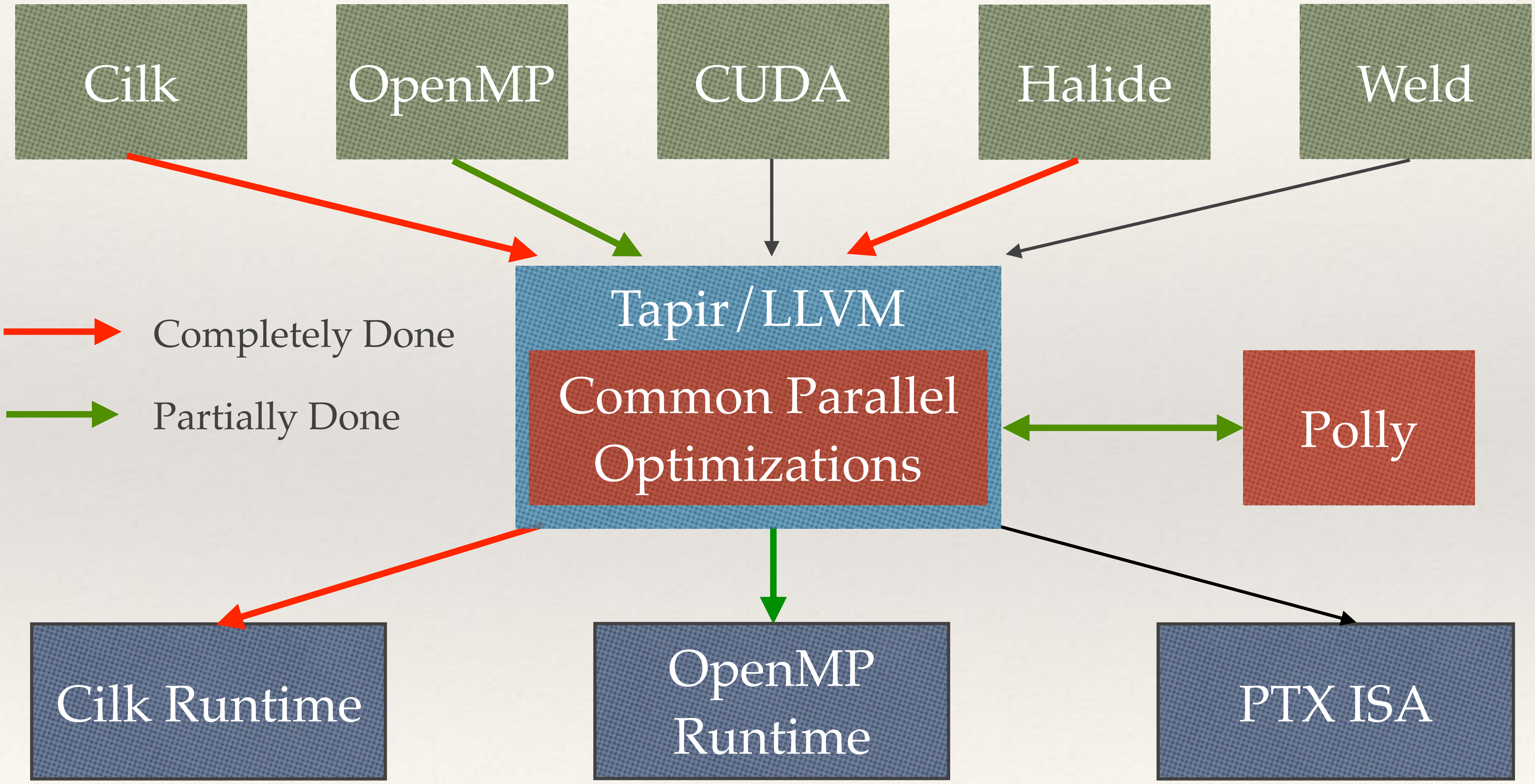
Multiple Parallel "Backends"

- Tapir is a nice way of representing and working with parallel programs
- Use Tapir as a common parallel intermediate representation for various parallel frontends and backends
- Benefits
  - Enable cross-framework compilation
  - Have one set of common parallel optimizations that can be shared by all

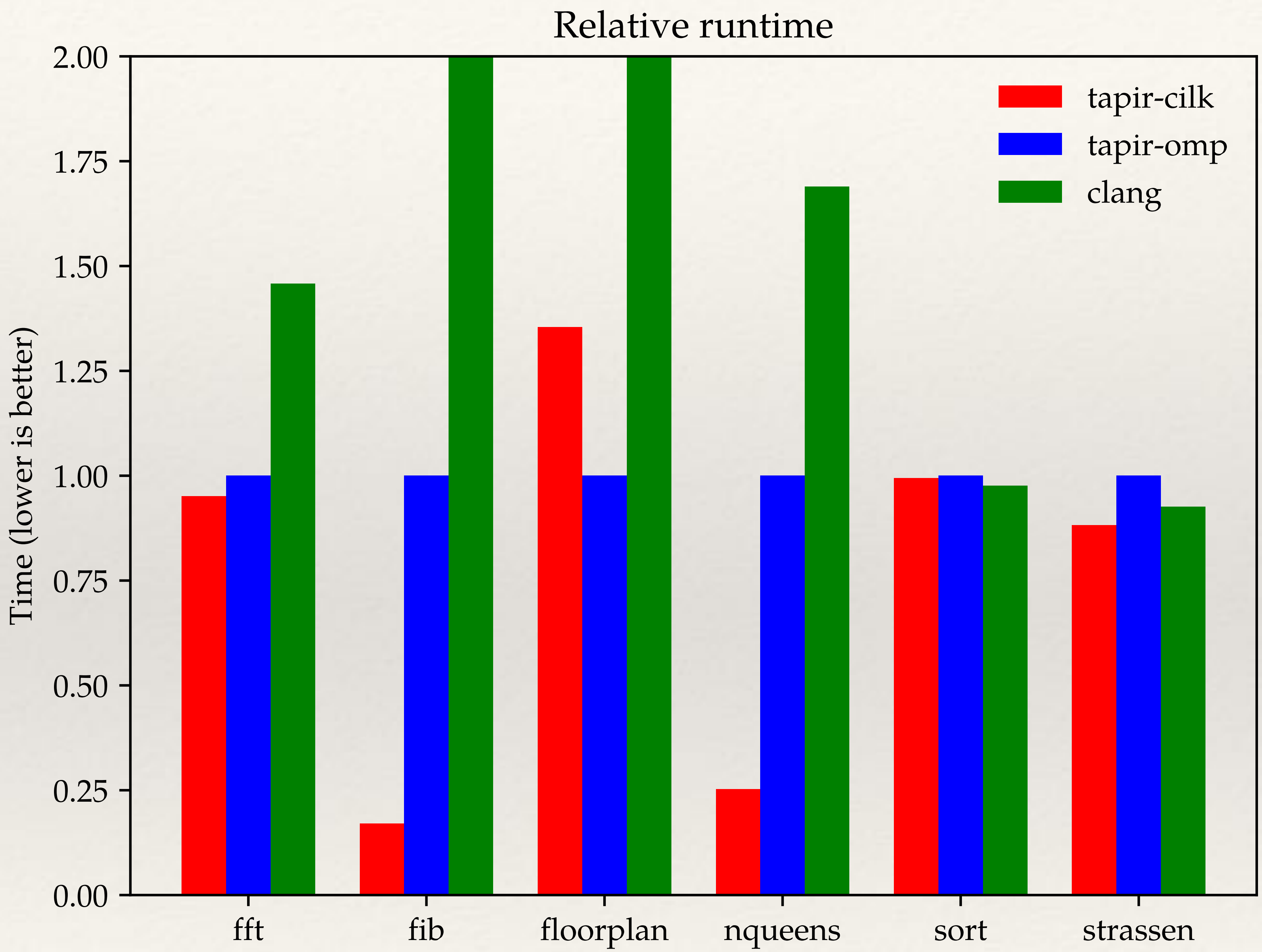
Rhino: The Parallel Compiler Dream



Rhino: The Parallel Compiler Dream

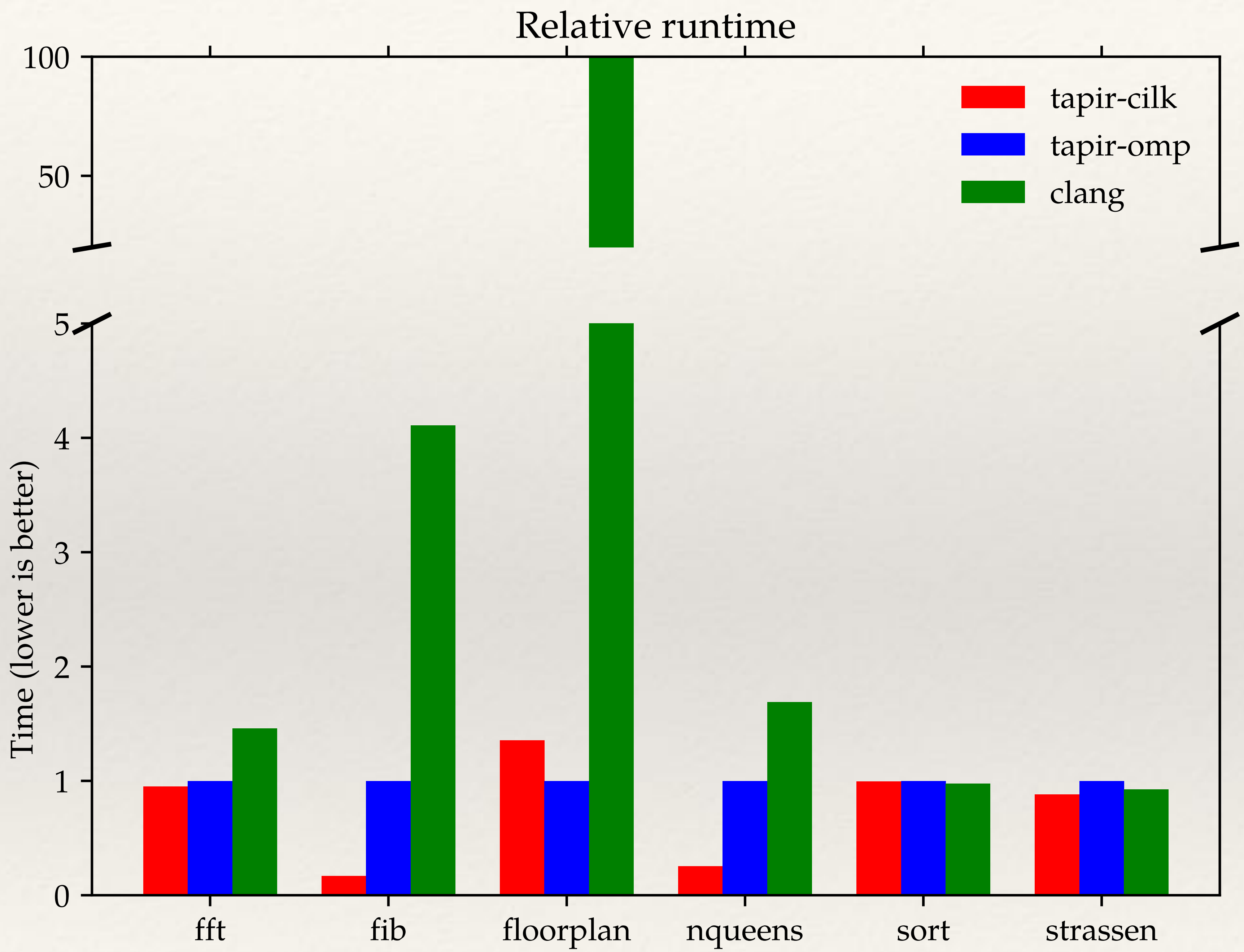


Parallel Runtime Choice



Examples from Barcelona OpenMP benchmark suite

Parallel Runtime Choice



Examples from Barcelona OpenMP benchmark suite

Takeaways

- With little modification, the compiler can do a lot of things to make your parallel programs faster
  - Run (serial) optimizations on parallel code
  - Build and share parallel optimizations
  - Mix-and-match parallel runtimes
- Ongoing development (bug fixes, new optimizations, etc).
- Available on GitHub!
  https://github.com/wsmoses/Parallel-IR.git

Backup Slides!

Obstacle

- When designing parallel optimization passes, we ran into the issue where we couldn't represent the optimized code inside of Tapir!

```
void B() {
  detach B1();
         B2();
  sync;
}

void main() {
  detach A();
         B();
         C();
  sync;
}
```

A is parallel to C

Inlining →

```
void main() {
  detach A();
  detach B1();
         B2();
  sync;
        C();
  sync;
}
```

A must execute before C

Obstacle

- Tapir assumes detaches/syncs (or specifically detaches/syncs) are scoped to a function, whereas we need something more precise.
- How much more precise?
  - Provide a sync to individual detaches?
  - Provide a sync to groups of detaches?

Idea 1: Individualized Sync

- Permit synchronization of specific parallel statements
- Most general model

```
void main() {
  a = detach A();
  b = detach B1();
             B2();
  sync a;
             C();
  sync b;
}
```

Idea 1: Individualized Sync

- Representing arbitrary sets to sync dramatically increases complexity
- Generality of model restricts possible runtimes
- Harder to optimize! (Previously could assume that a detached statement no longer can alias after a sync)

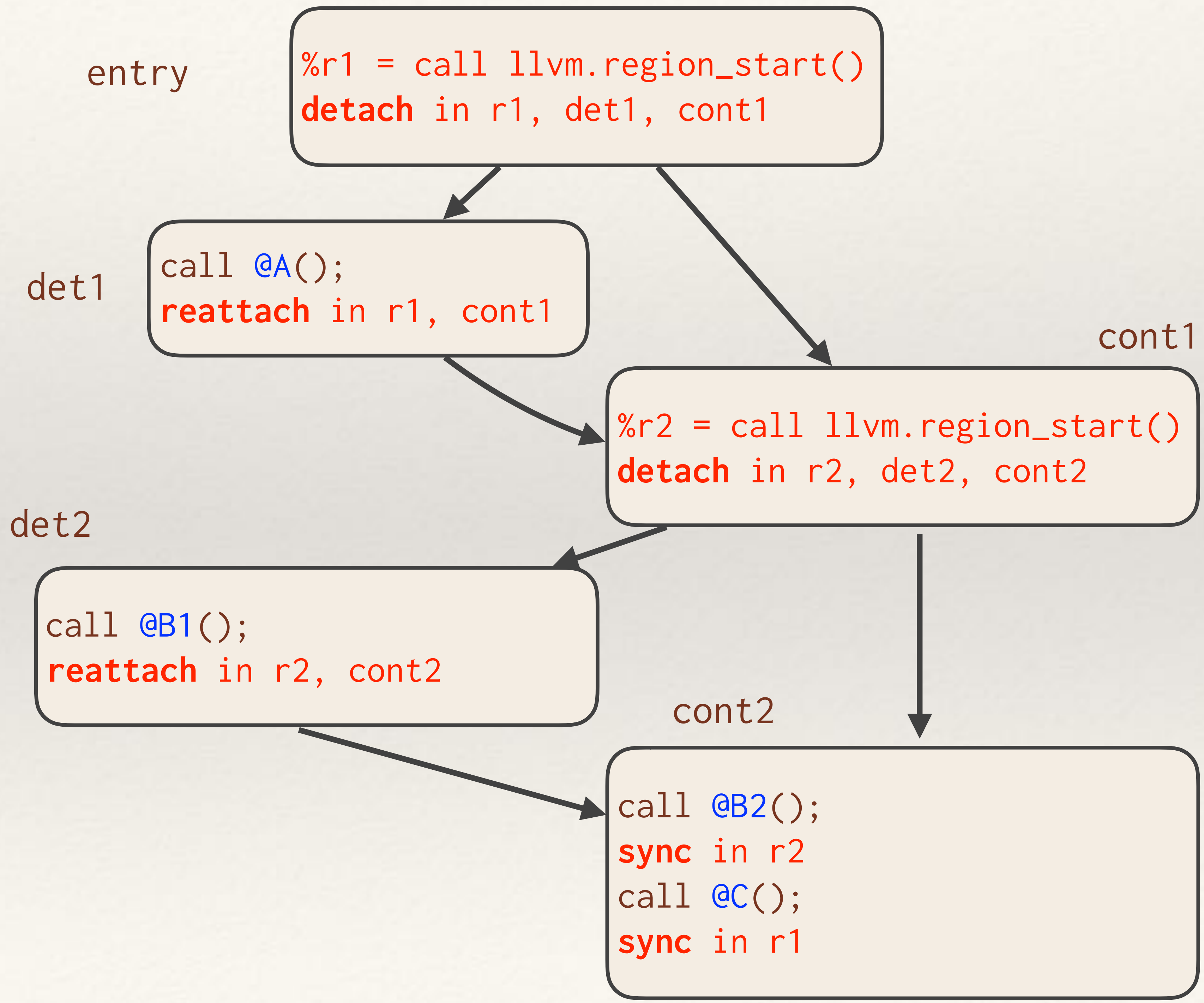
```
f = detach foo();
ϕ = {}

for (int i = 0; i < n; ++i) {
  γ0 = phi [(ϕ, entry), (γ1, loop)]
  a  = detach A(i);
  γ1 = union [ γ0, a ]
}
γ2 = phi [(ϕ, entry), (γ1, loop)]

sync γ2;

bar();
```

Idea 2: Scoped Sync

- Represent parallelism in nested parallel regions
- A sync now acts on all detaches in that region
- Doesn't change runtime compatibility
- Maintain guarantee that no detaches (now in the region) continue after a sync
  - This implies that all parallel optimizations developed for vanilla Tapir work, except using a parallel region scope instead of function scope

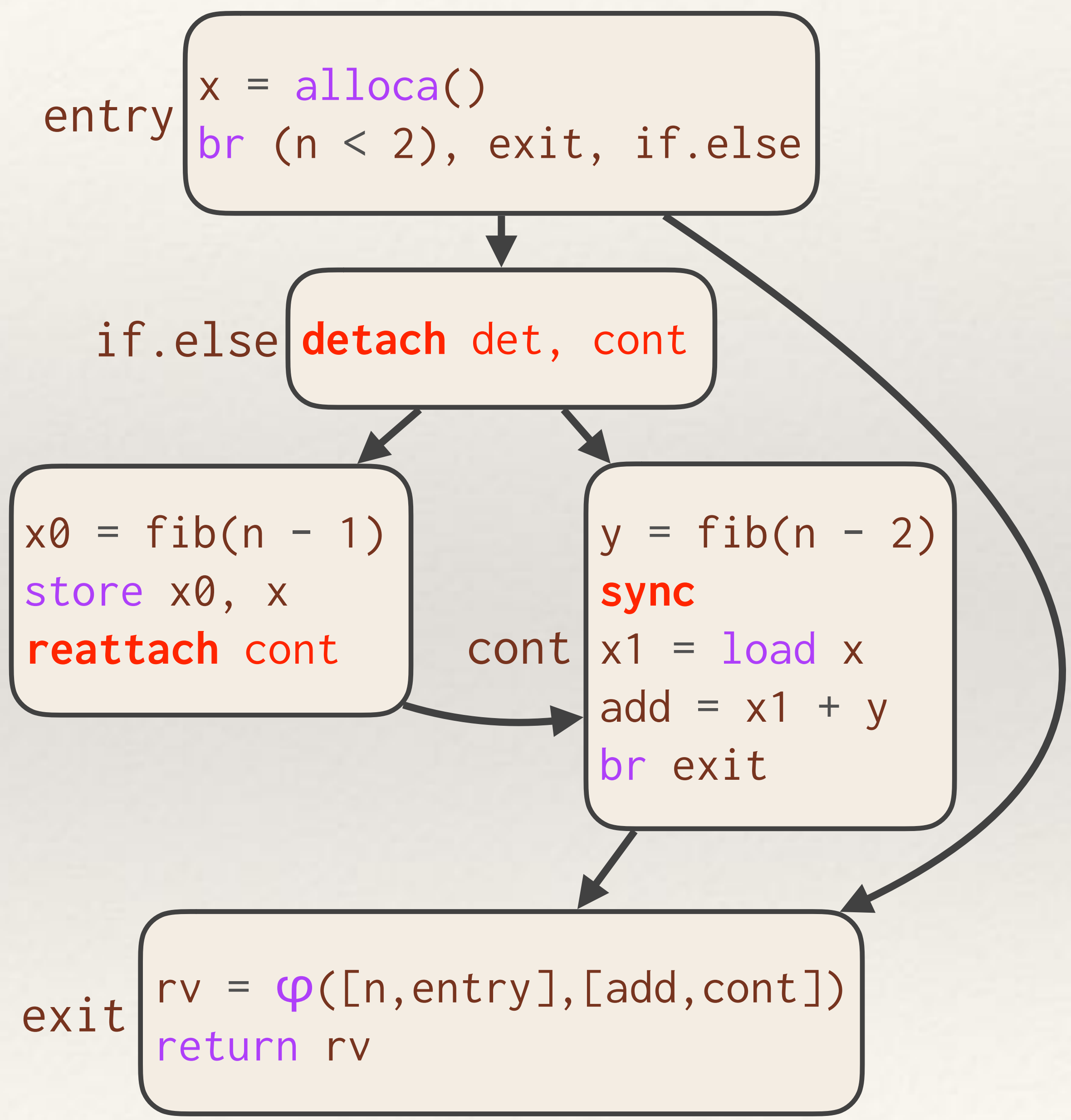
Idea 2: Individualized Sync

```
void main() {

  detach A();

  parallel_region {
    detach B1();
           B2();
    sync;
  }

  C();
  sync;
}
```



Maintaining Correctness

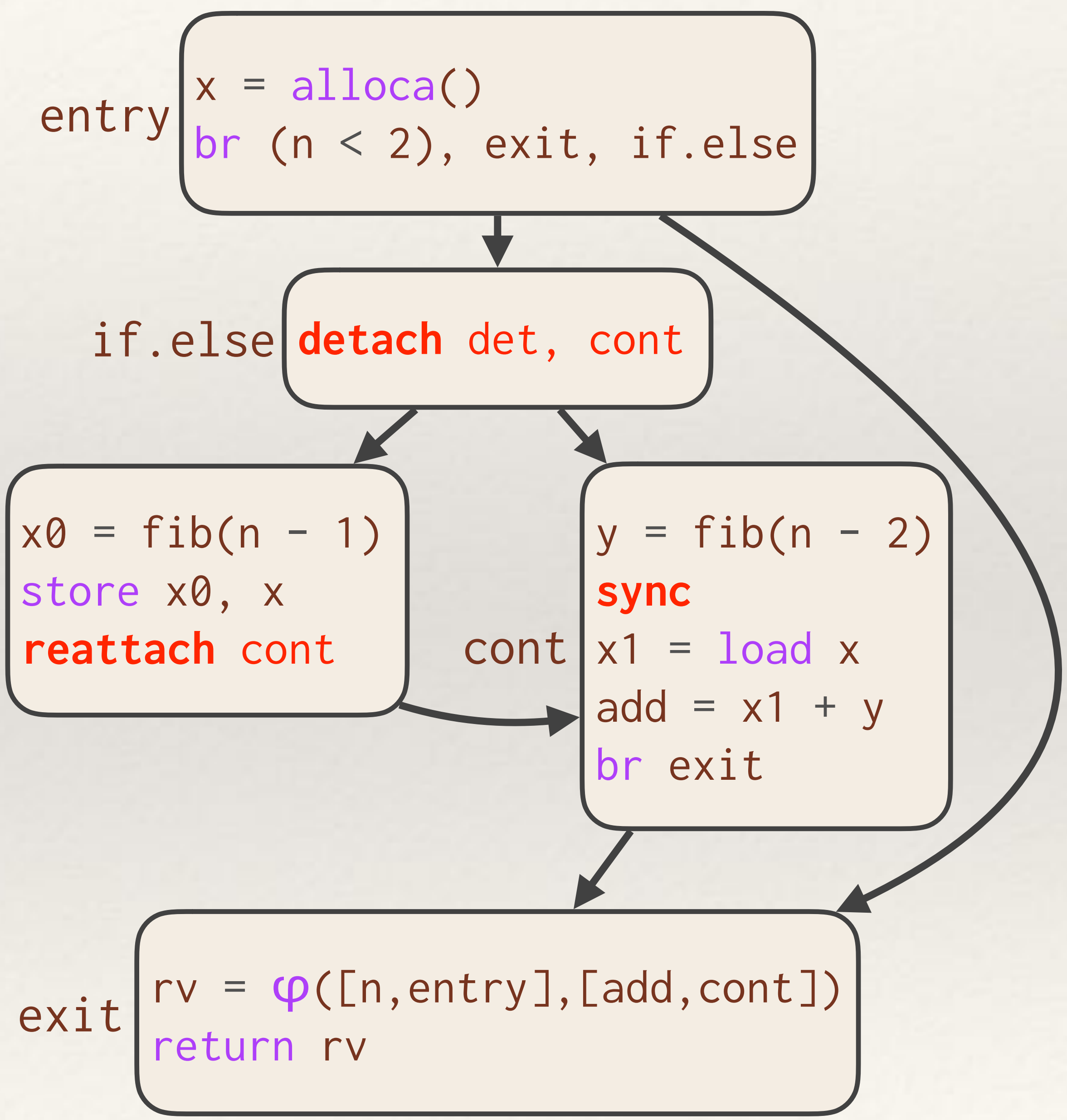
Problem: How does the compiler ensure that code motion does not introduce a determinacy race into otherwise race-free code?



Maintaining Correctness

Problem: How does the compiler ensure that code motion does not introduce a determinacy race into otherwise race-free code?

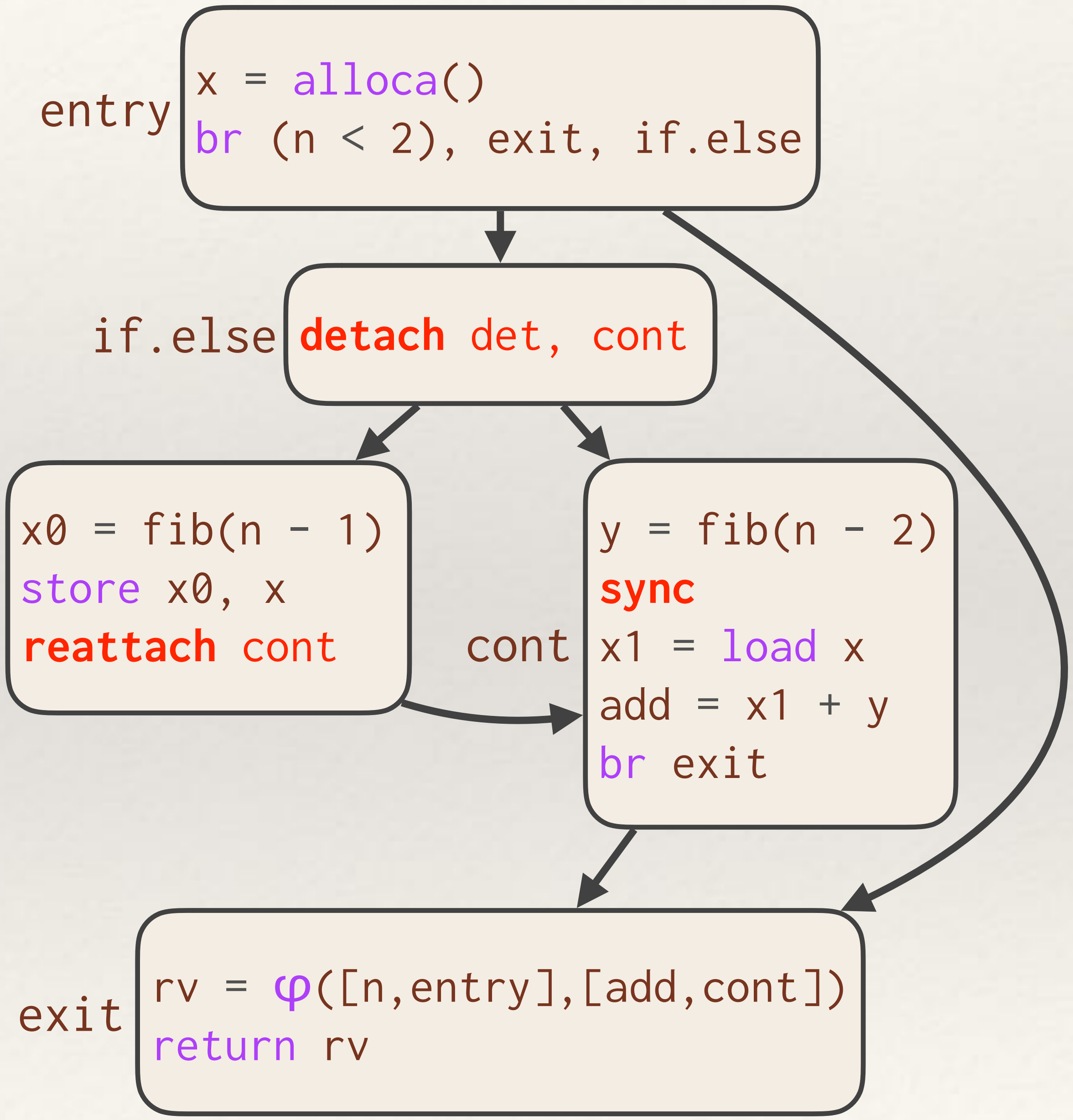
- It suffices to consider moving memory operations around each new instruction.



Maintaining Correctness

Problem: How does the compiler ensure that code motion does not introduce a determinacy race into otherwise race-free code?

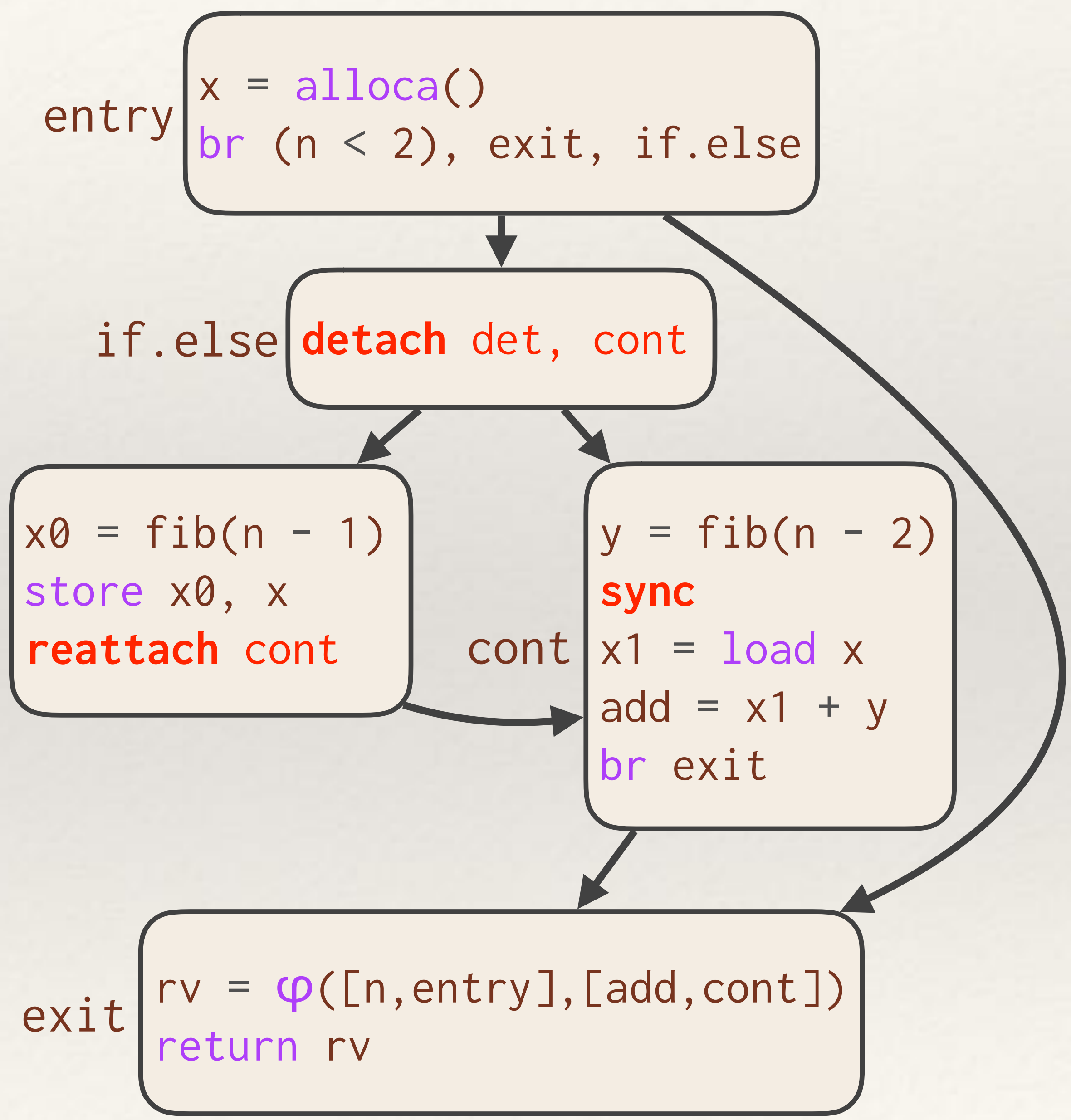
- It suffices to consider moving memory operations around each new instruction.
- Moving code above a `detach` or below a `sync` serializes it and is always valid.



Maintaining Correctness

Problem: How does the compiler ensure that code motion does not introduce a determinacy race into otherwise race-free code?

- It suffices to consider moving memory operations around each new instruction.
- Moving code above a `detach` or below a `sync` serializes it and is always valid.
- Other potential races are handled by giving `detach`, `reattach`, and `sync` appropriate attributes and by slight modifications to `mem2reg`.



Valid serial passes cannot create race bugs.

Most of LLVM's existing serial passes "just work" on parallel code.

Example Optimization: Fusion

- Existing code written in parallel frameworks can leverage polyhedral optimizations such as loop fusion or tiling with no extra effort

```
void add(double * A, double * B, double * C, int n) {
  parallel_for (int i = 0; i < n; i++) {
    A[i] += B[i];
  }
  parallel_for (int i = 0; i < n; i++) {
    A[i] += C[i];
  }
}
```

```
void add(double * A, double * B, double * C, int n) {
  parallel_for (int i = 0; i < n; i++) {
    A[i] += B[i];
    A[i] += C[i];
  }
}
```

Example Optimization: Fusion

- Existing code written in parallel frameworks can leverage polyhedral optimizations such as loop fusion or tiling with no extra effort

```
void add(double * A, double * B, double * C, int n) {
  parallel_for (int i = 0; i < n; i++) {
    A[i] += B[i];
  }
  parallel_for (int i = 0; i < n; i++) {
    A[i] += C[i];
  }
}
```

```
void add(double * A, double * B, double * C, int n) {
  parallel_for (int i = 0; i < n; i++) {
    A[i] += B[i];
    A[i] += C[i];
  }
}
```

~2x