How To Use LLVM To Optimize Your Parallel Programs

William S. Moses

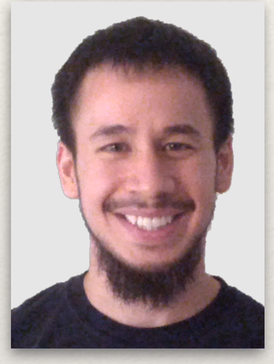
2018 US LLVM Developers Meeting
October 18, 2018

CSAIL
MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

Tutorial "TA":

George Stelle

Tapir Authors:



Tao B. Schardl

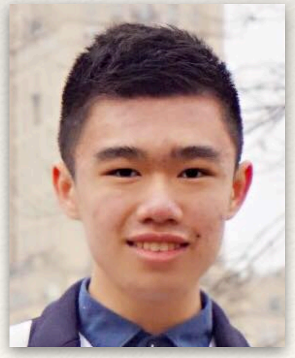
William S. Moses

Charles E. Leiserson

Parallel Opt Authors:

William S. Moses

George Stelle



Jiahao Li

Install Pre-Requisites

- This is going to be an interactive tutorial!
- In the background, make sure you have docker installed (https://docs.docker.com/install/)
- Pull the pre-prepared docker instance
  - docker pull wsmoses/tapir-built
- Download the git repo for the tutorial
  - git clone https://github.com/wsmoses/tapir-tutorial
- Test installation (good idea run in separate terminal tab/tmux)
  - cd tapir-tutorial/fib && make run

Introduction (as everyone gets set up)

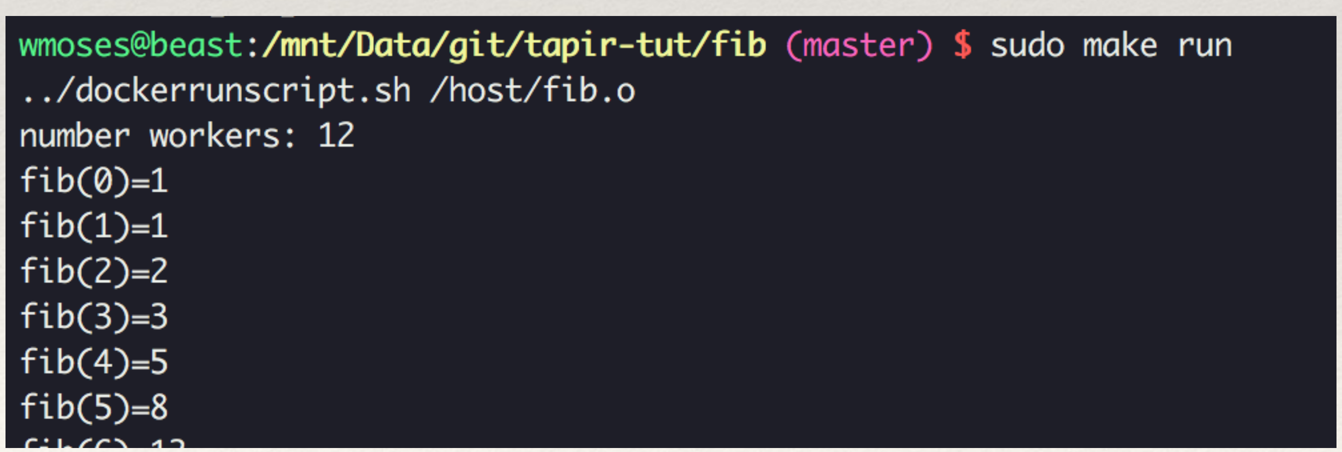
- Building a parallel language / framework can often be a difficult, laborious task
- Once built, compilers and tools for such frameworks often create code that is far from optimal (we'll see this shortly)
- This means users have to spend more time writing code that doesn't run as fast
- This talk will illustrate how support for parallelism in LLVM will both make parallel programs run faster and also make it easier for languages to incorporate parallelism

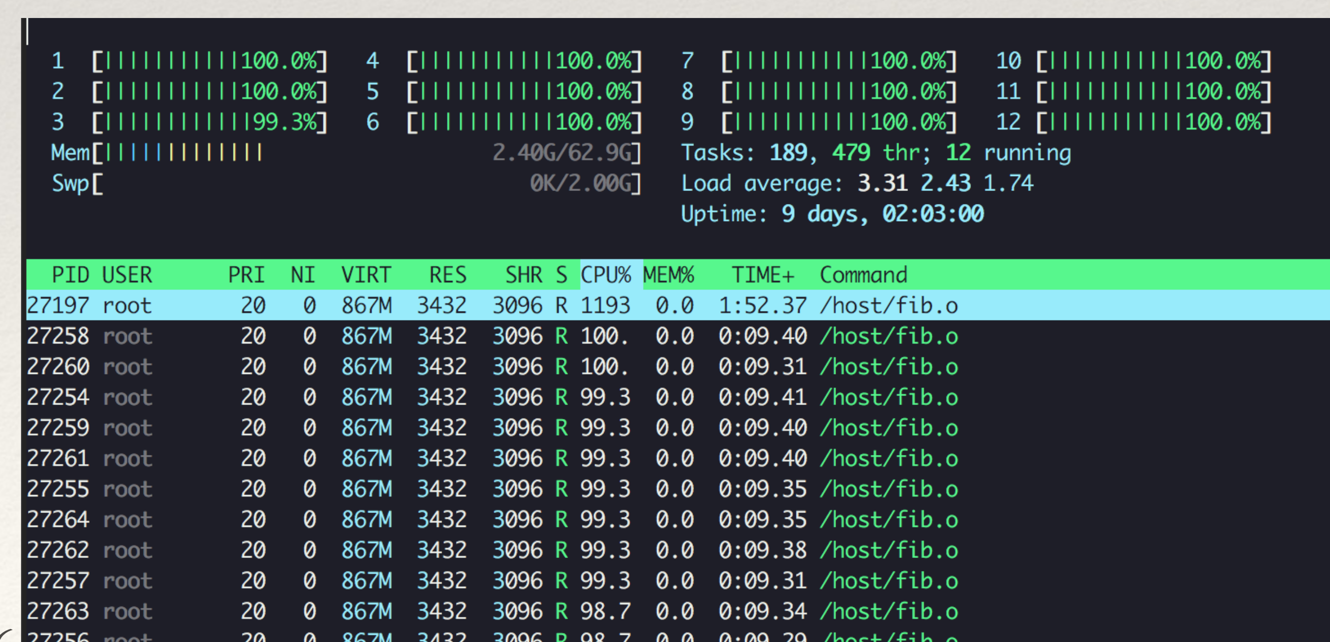
Introduction (as everyone gets set up)

- In this tutorial, we'll be using Tapir — an extension to LLVM developed by Moses (that's me), Schardl, and Leiserson at MIT that allows it to reason about parallel programs
- For those who wish to try it out themselves it's available on Github: https://github.com/wsmoses/Tapir-LLVM
- For those who want to see parallelism introduced into mainline LLVM, please come to the BOF later today!

Tutorial 0: Verify Installation

- Go into tapir-tutorial/fib and “make run“
- You should see fibonacci numbers slowly printing out
- If you want to kill it run “docker kill tapirdocker”
- You should see the program running in parallel





Tutorial 0: Verify Installation

- Go into tapir-tutorial/fib and “make run“
- You should see fibonacci numbers slowly printing out
- If you want to kill it run “docker kill tapirdocker”
- You should see the program running in parallel

```
wmoses@beast:/mnt/Data/git/tapir-tut/fib (master) $ sudo make run
../dockerrunscript.sh /host/fib.o
number workers: 12
fib(0)=1
fib(1)=1
fib(2)=2
fib(3)=3
fib(4)=5
fib(5)=8
```

```
  1  [|||||||||||100.0%]   4  [|||||||||||100.0%]   7  [|||||||||||100.0%]   10 [|||||||||||100.0%]
  2  [|||||||||||100.0%]   5  [|||||||||||100.0%]   8  [|||||||||||100.0%]   11 [|||||||||||100.0%]
  3  [||||||||||||99.3%]   6  [|||||||||||100.0%]   9  [|||||||||||100.0%]   12 [|||||||||||100.0%]
  Mem[|||||||||||||                           2.40G/62.9G]   Tasks: 189, 479 thr; 12 running
  Swp[                                          0K/2.00G]   Load average: 3.31 2.43 1.74
                                                             Uptime: 9 days, 02:03:00

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
27197 root       20   0  867M  3432  3096 R 1193  0.0  1:52.37 /host/fib.o
27258 root       20   0  867M  3432  3096 R 100.  0.0  0:09.40 /host/fib.o
27260 root       20   0  867M  3432  3096 R 100.  0.0  0:09.31 /host/fib.o
27254 root       20   0  867M  3432  3096 R 99.3  0.0  0:09.41 /host/fib.o
27259 root       20   0  867M  3432  3096 R 99.3  0.0  0:09.40 /host/fib.o
27261 root       20   0  867M  3432  3096 R 99.3  0.0  0:09.40 /host/fib.o
27255 root       20   0  867M  3432  3096 R 99.3  0.0  0:09.35 /host/fib.o
27264 root       20   0  867M  3432  3096 R 99.3  0.0  0:09.35 /host/fib.o
27262 root       20   0  867M  3432  3096 R 99.3  0.0  0:09.38 /host/fib.o
27257 root       20   0  867M  3432  3096 R 99.3  0.0  0:09.31 /host/fib.o
27263 root       20   0  867M  3432  3096 R 98.7  0.0  0:09.34 /host/fib.o
27256 root       20   0  867M  3432  3096 R 98.7  0.0  0:09.29 /host/fib.o
```

Tutorial 0: Verify Installation

fib.c

```c
#include <stdio.h>
#include <cilk/cilk.h>
#include <cilk/cilk_api.h>

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

int main() {
    printf("number workers: %d\n", __cilkrts_get_nworkers());
    fflush(0);
    for(int i=0; i<230; i++) {
        printf("fib(%d)=%d\n", i, fib(i));
        fflush(0);
    }
}
```

- Cilk code to compute a large number of fibonacci numbers in parallel
- Not fastest algorithm, but let's us check everything is working

Tutorial 0: Verify Installation

Makefile

```
all: fib.ll fib.o

clean:
	rm -f fib.ll fib.o

fib.ll: fib.c
	../dockerscript.sh clang -O3 -fcilkplus /host/fib.c -fdetach -o /host/fib.ll -S -emit-llvm

fib.o: fib.c
	../dockerscript.sh clang -O3 -fcilkplus /host/fib.c -fdetach -o /host/fib.o

run: fib.o
	../dockerrunscript.sh /host/fib.o

kill:
	docker kill tapirdocker

```

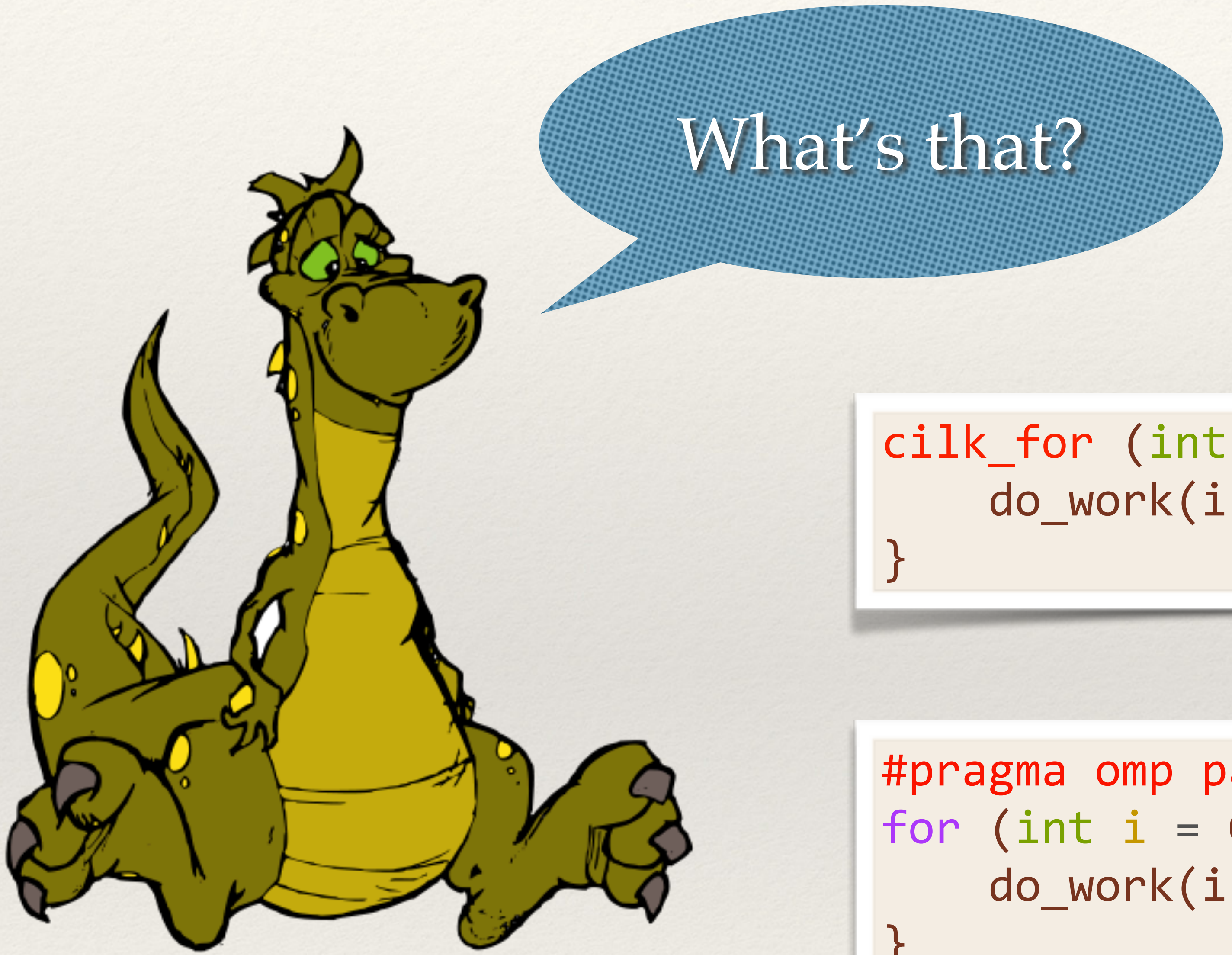
- We can open fib.ll to see what the program looks like in LLVM
- Special scripts to compile/run using docker container (can use your own machine if things are set up happily)

Tutorial 0: Verify Installation

```
Makefile

all: fib.ll fib.o

clean:
	rm -f fib.ll fib.o

fib.ll: fib.c
	../dockerscript.sh clang -O3 -fcilkplus /host/fib.c -fdetach -o /host/fib.ll -S -emit-llvm

fib.o: fib.c
	../dockerscript.sh clang -O3 -fcilkplus /host/fib.c -fdetach -o /host/fib.o

run: fib.o
	../dockerrunscript.sh /host/fib.o

kill:
	docker kill tapirdocker
```

- We can open fib.ll to see what the program looks like in LLVM
- Special scripts to compile/run using docker container (can use your own machine if things are set up happily)

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);
}
```

Tutorial 1: Normalizing a Vector

```c
__attribute__((const)) double mag(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] / mag(in, n);
}
```

- Goal: make the fastest (parallel) normalize code we can!
- To start, let's see how the serial code does
- Go into tapir-tutorial/norm-mp and run make run

```
wmoses@beast:/mnt/Data/git/tapir-tut/norm-mp (master) $ sudo make run
../dockerscript.sh clang -O3 -fopenmp /host/norm.c -o /host/norm.o
../dockerrunscript.sh /host/norm.o 2000
SER Normalize Runtime 0.000016 0.000500
```

Tutorial 1: Normalizing a Vector

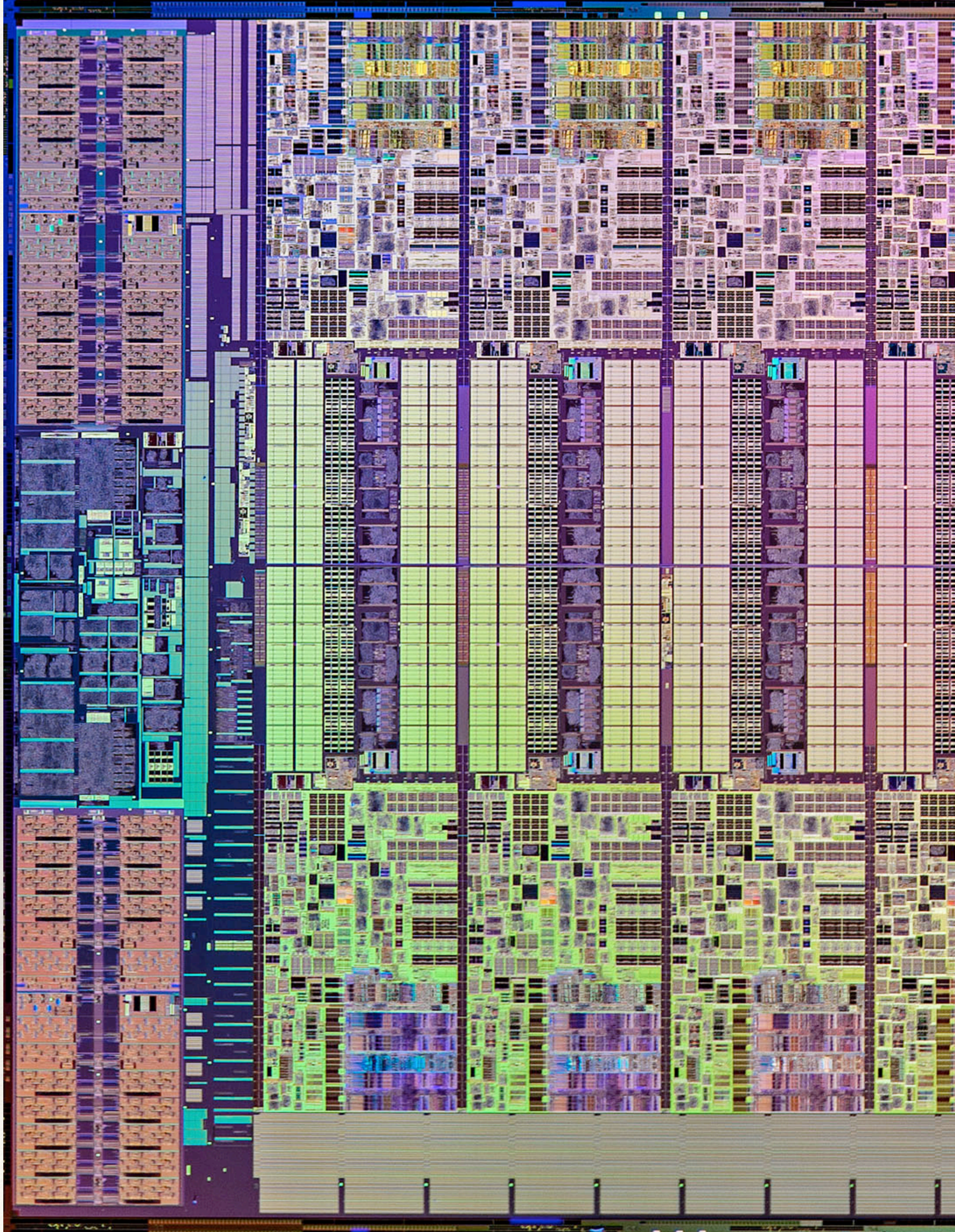
```
wmoses@beast:/mnt/Data/git/tapir-tut/norm-mp (master) $ sudo make run
../dockerscript.sh clang -O3 -fopenmp /host/norm.c -o /host/norm.o
../dockerrunscript.sh /host/norm.o 2000
SER Normalize Runtime 0.000016 0.000500
```

Runtime (seconds)

Size of vector

```
run: norm.o
	../dockerrunscript.sh /host/norm.o 2000

run-mp: normmp.o
	../dockerrunscript.sh /host/normmp.o 2000

run-all: run run-mp

```

Idea: Let’s Run in Parallel!



Tutorial 1: Normalizing a Vector

```
wmoses@beast:/mnt/Data/git/tapir-tut/norm-mp (master) $ sudo make run-all
../dockerrunscript.sh /host/norm.o 2000
SER Normalize Runtime 0.000020 0.000500
../dockerrunscript.sh /host/normmp.o 2000
OMP Normalize Runtime 0.015653 0.000500
```

Parallel is slower :(

Tutorial 1: Normalizing a Vector

Maybe we need bigger vector?

```
wmoses@beast:/mnt/Data/git/tapir-tut/norm-mp (master) $ sudo make run-all
../dockerrunscript.sh /host/norm.o 2000
SER Normalize Runtime 0.000020 0.000500
../dockerrunscript.sh /host/normmp.o 2000
OMP Normalize Runtime 0.015653 0.000500
```

Tutorial 1: Normalizing a Vector

Maybe we need bigger vector?

```
wmoses@beast:/mnt/Data/git/tapir-tut/norm-mp (master) $ sudo make run-all
../dockerrunscript.sh /host/norm.o 2000
SER Normalize Runtime 0.000020 0.000500
../dockerrunscript.sh /host/normmp.o 2000
OMP Normalize Runtime 0.015653 0.000500
```

```
wmoses@beast:/mnt/Data/git/tapir-tut/norm-mp (master) $ sudo make run-all
../dockerscript.sh /host/norm.o 200000
SER Normalize Runtime 0.002014 0.000005
../dockerscript.sh /host/normmp.o 200000
OMP Normalize Runtime 2.871470 0.000005
```

Nope

What happened?

- Try to figure out why it's running slower
- The LLVM files are helpful

What happened?

```
__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] / mag(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 = mag(in, n);
  for (int i = 0; i < n; ++i)
    out[i] = in[i] / tmp;
}
```

What happened?

```
__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] / mag(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 = mag(in, n);
  for (int i = 0; i < n; ++i)
    out[i] = in[i] / tmp;
}
```

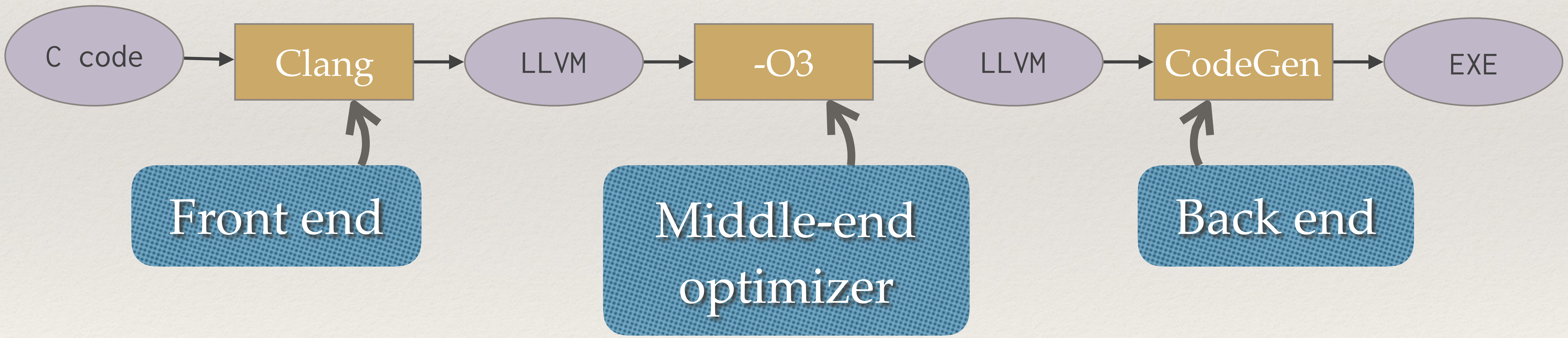
This did NOT happen for the parallel code!

What happened?

```
; Function Attrs: noinline nounwind uwtable
define void @normalize(double* noalias, double* noalias, i32) local_unnamed_addr #3 {
  %4 = alloca double*, align 8
  %5 = alloca double*, align 8
  %6 = alloca i32, align 4
  store double* %0, double** %4, align 8, !tbaa !12
  store double* %1, double** %5, align 8, !tbaa !12
  store i32 %2, i32* %6, align 4, !tbaa !14
  call void (%ident_t*, i32, void (i32*, i32*, ...)*, ...) @__kmpc_fork_call(%ident_t* nonnull @0, i32 3, void (i32*, i32*, ...)* bitcast (void (i32*, i32*, i32*, double**, double**)* @.omp_outlined. to void (i32*, i32*, ...)*), i32* nonnull %6, double** nonnull %4, double** nonnull %5) #7
  ret void
}
```

The body of the loop got outlined

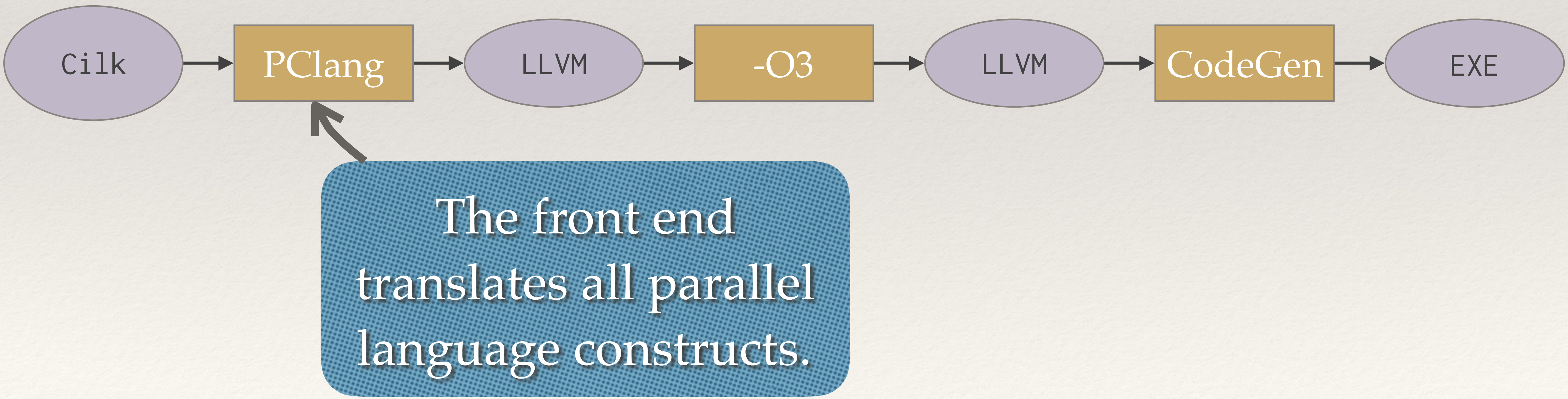
The LLVM Compilation Pipeline



Compiling Parallel Code

LLVM pipeline

Cilk Plus/LLVM pipeline



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

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.

Remember fib?

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**.

Tutorial 2: Tapir Instructions

- Go into tapir-tutorial/norm and run make tapir
- There are two files tapirpre.ll and tapirpost.ll
- Let's take a look at tapirpre.ll and the source code (norm.c)

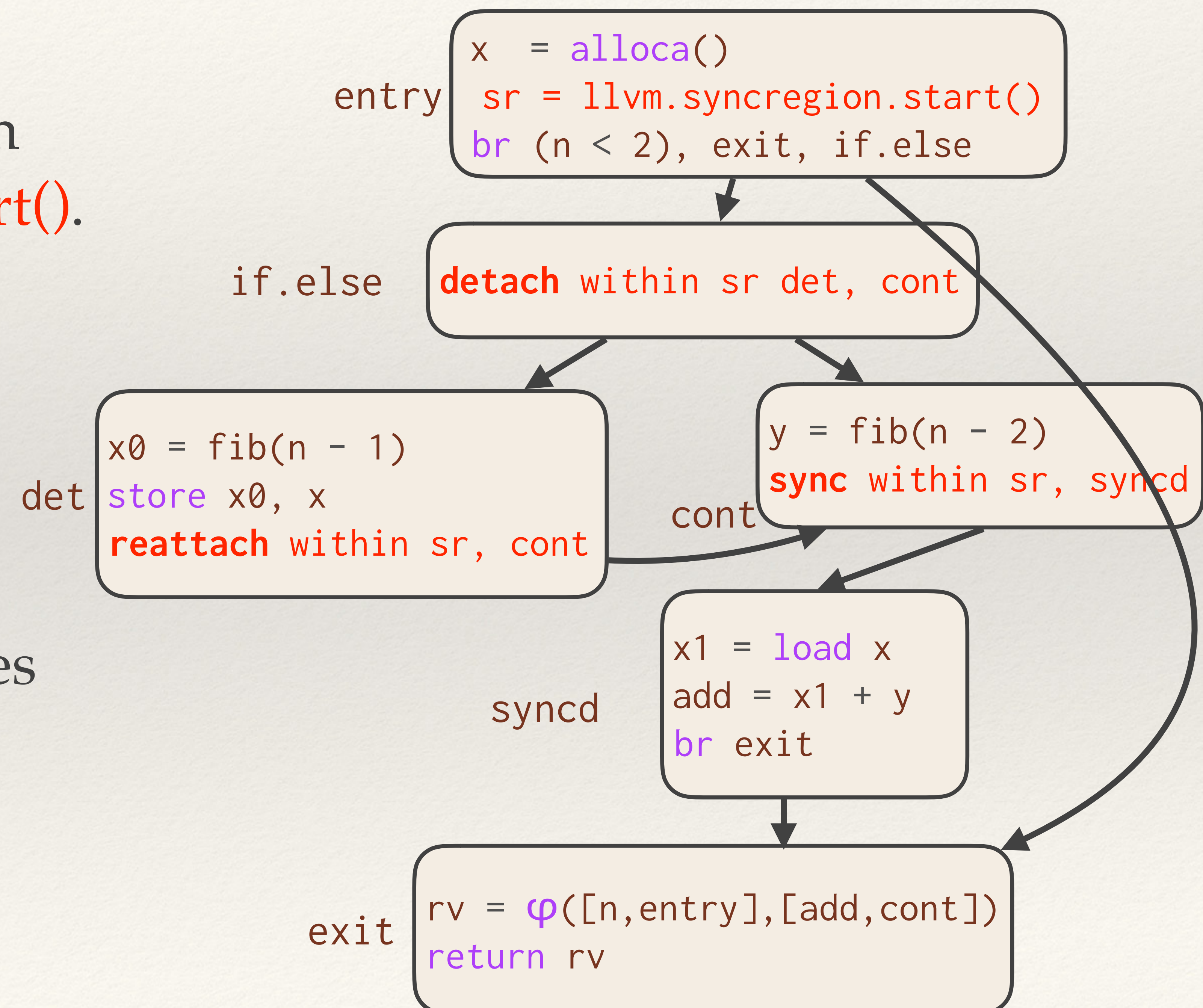
Tutorial 2: Tapir Instructions

- Go into tapir-tutorial/norm and run make tapir
- There are two files tapirpre.ll and tapirpost.ll
- Let's take a look at tapirpre.ll and the source code (norm.c)
- New instructions: detach, reattach, and sync

Tapir Semantics

- Tapir introduces three new terminators into LLVM's IR: detach, reattach, sync, and an intrinsic llvm.syncregion.start().
- 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.

Tapir CFG

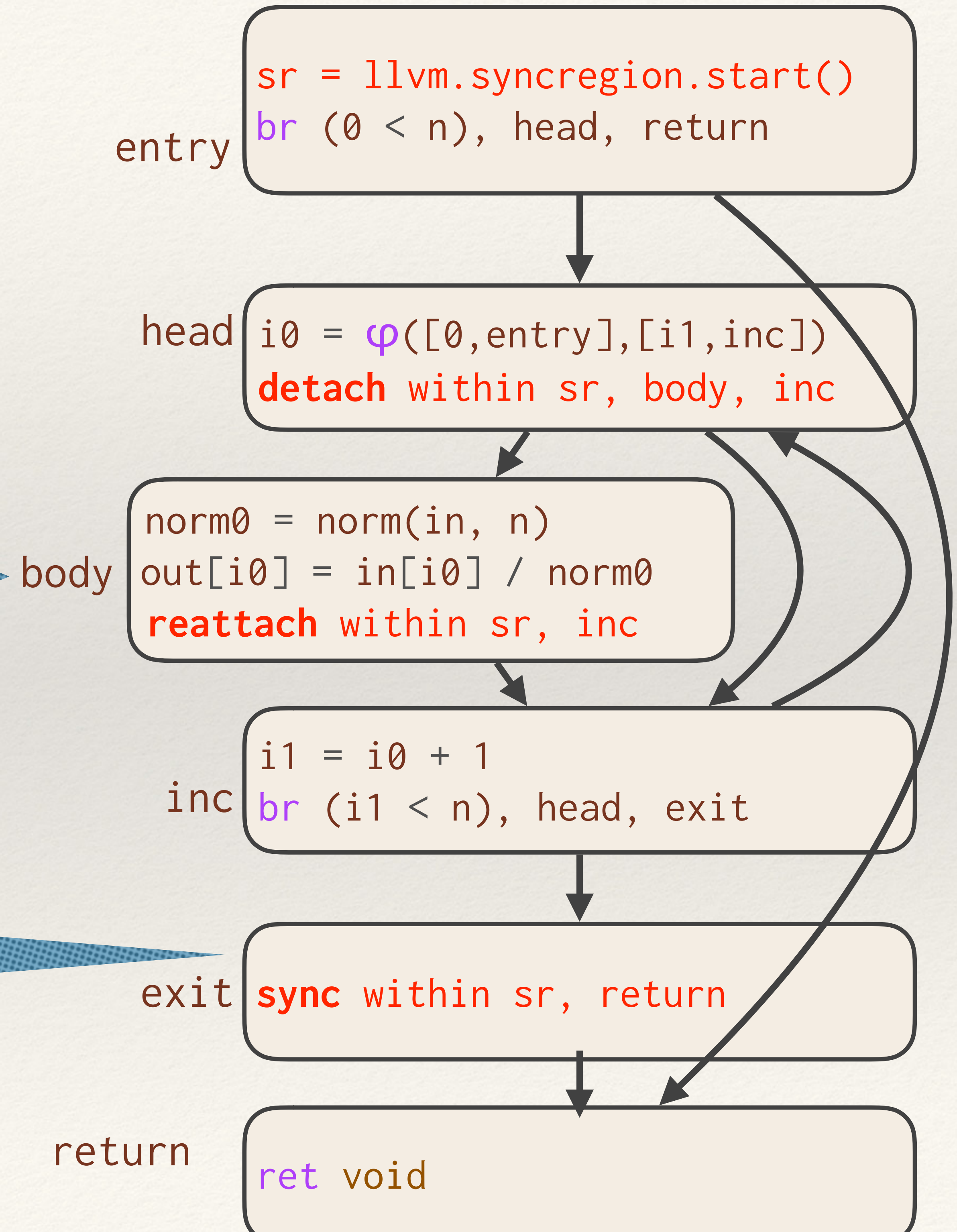


Parallel Loops in Tapir

```
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.

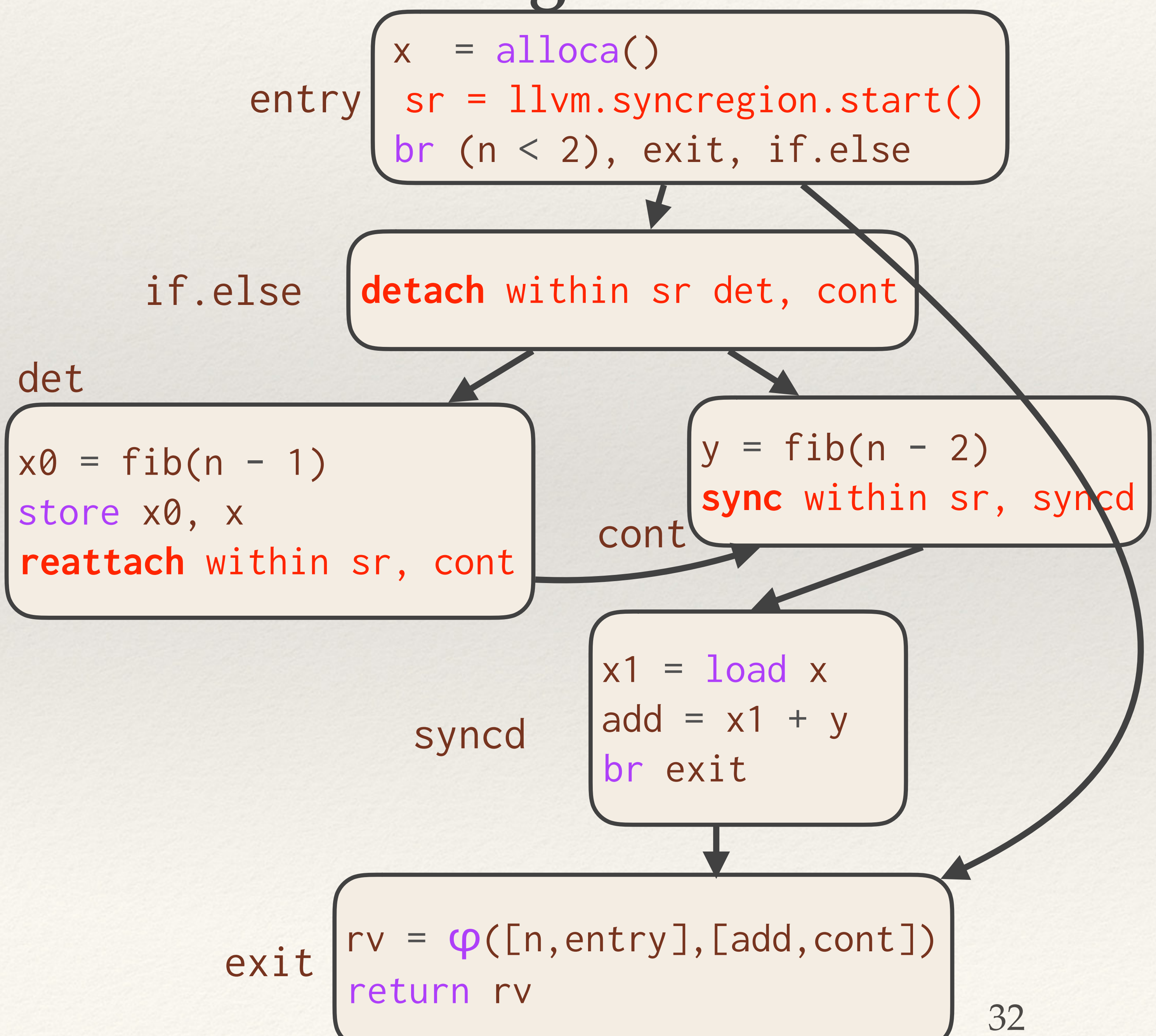


Tutorial 2: Tapir Instructions

- As expected, in Tapir post, the call to magnitude is moved outside of the loop.
- Let's get a closer look: cd tapir-tutorial/licm
- Run make
- What is happening?
- We can also look at tapir-tutorial/norm at the fast and slow versions (going through tapir, but electing to not run optimizations until after lowered to runtime calls)

How does this work?

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

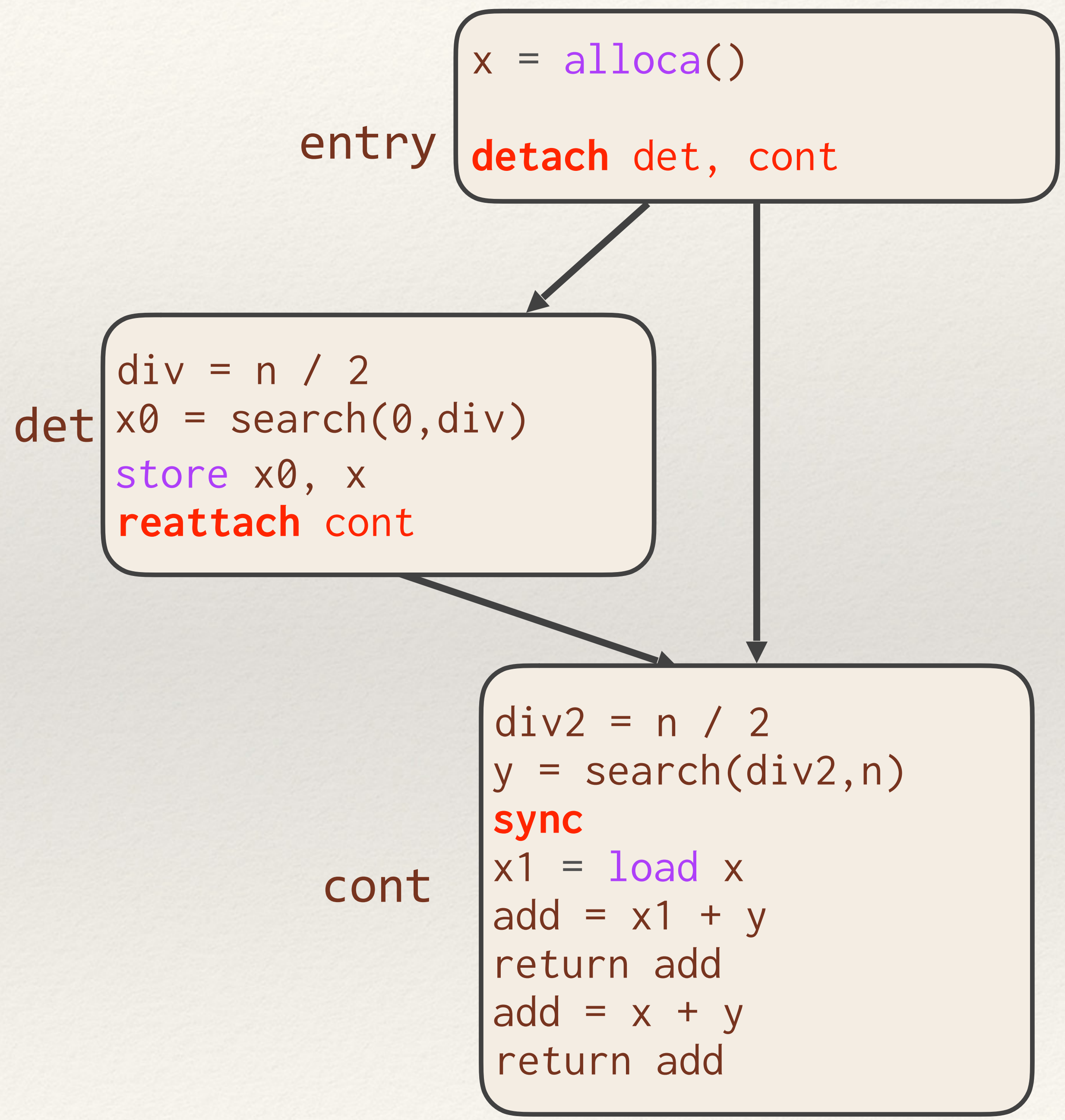


Many parts of the compiler can apply **standard implicit assumptions** of the CFG to this block.

Case Study: Common Subexpression Elimination

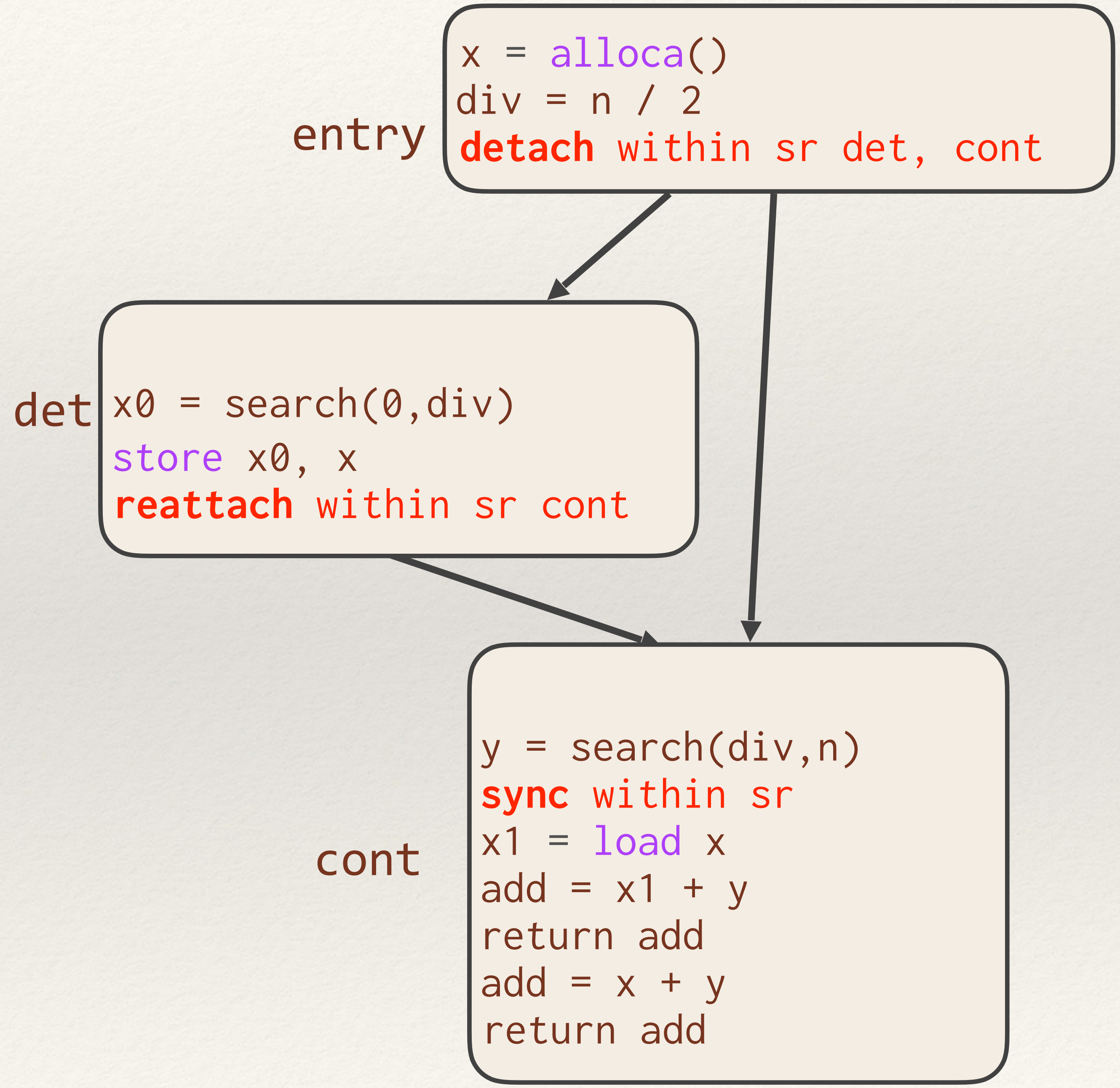
- CSE “just works.”
- Finding duplicate expressions and condensing them at their lowest common ancestor works fine for **detach**/ **reattach**.

```
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

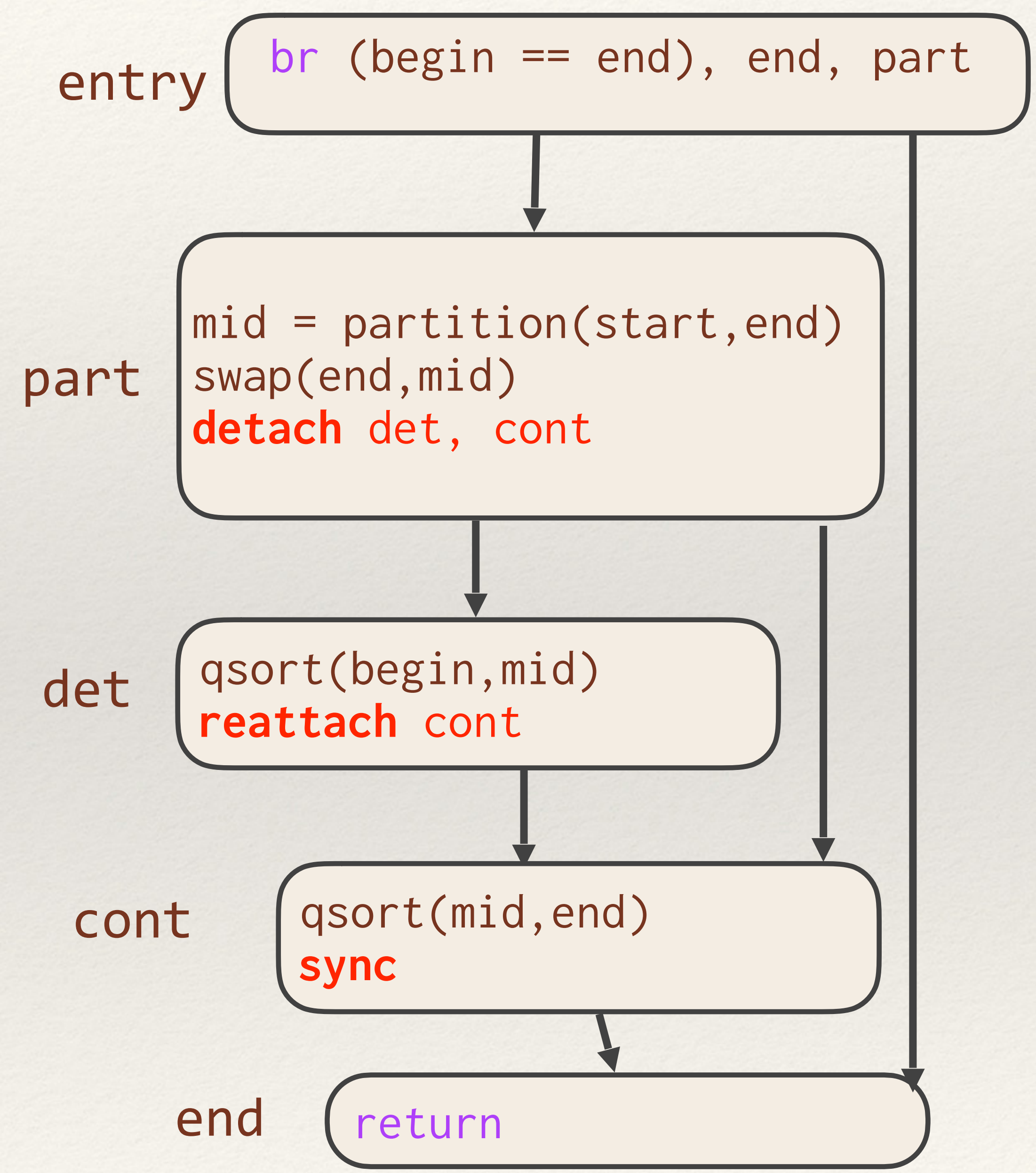
- CSE "just works."
- Finding duplicate expressions and condensing them at their lowest common ancestor works fine for `detach`/ `reattach`.




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

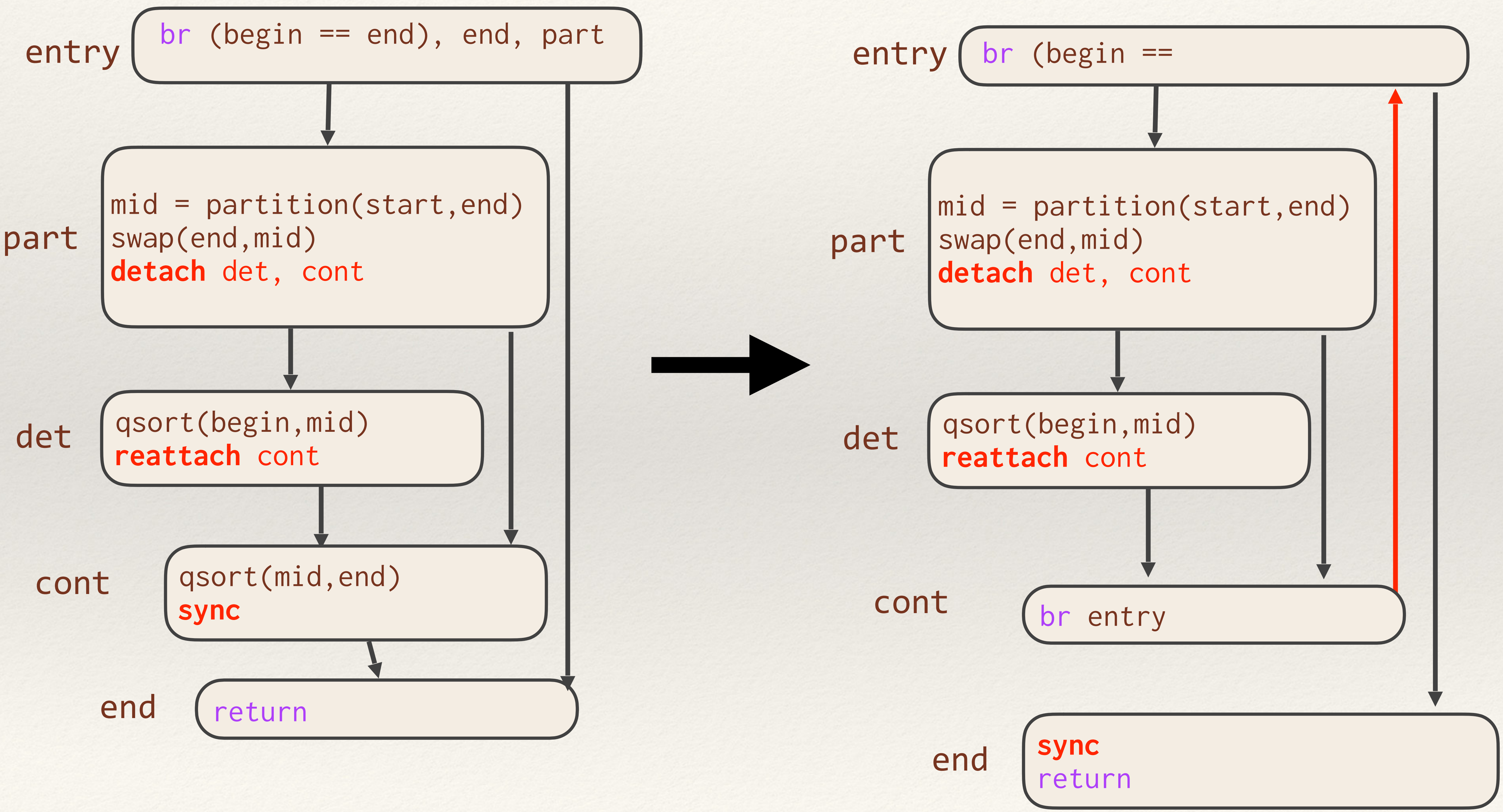
Case Study: Parallel Tail-Recursion Elimination

- A minor modification allows TRE to run on parallel code.
- Ignore `sync`'s before a recursive call and add `sync`'s before intermediate returns.




```
void qsort(int* begin, int* end) {
  if (begin == end) return;
  int* mid = partition(start, end);
  swap(end, mid);
  cilk_spawn qsort(begin, mid);
  qsort(mid, end);
  cilk_sync;
}
```

Case Study: Parallel Tail-Recursion Elimination



Compiler Analyses and Optimizations

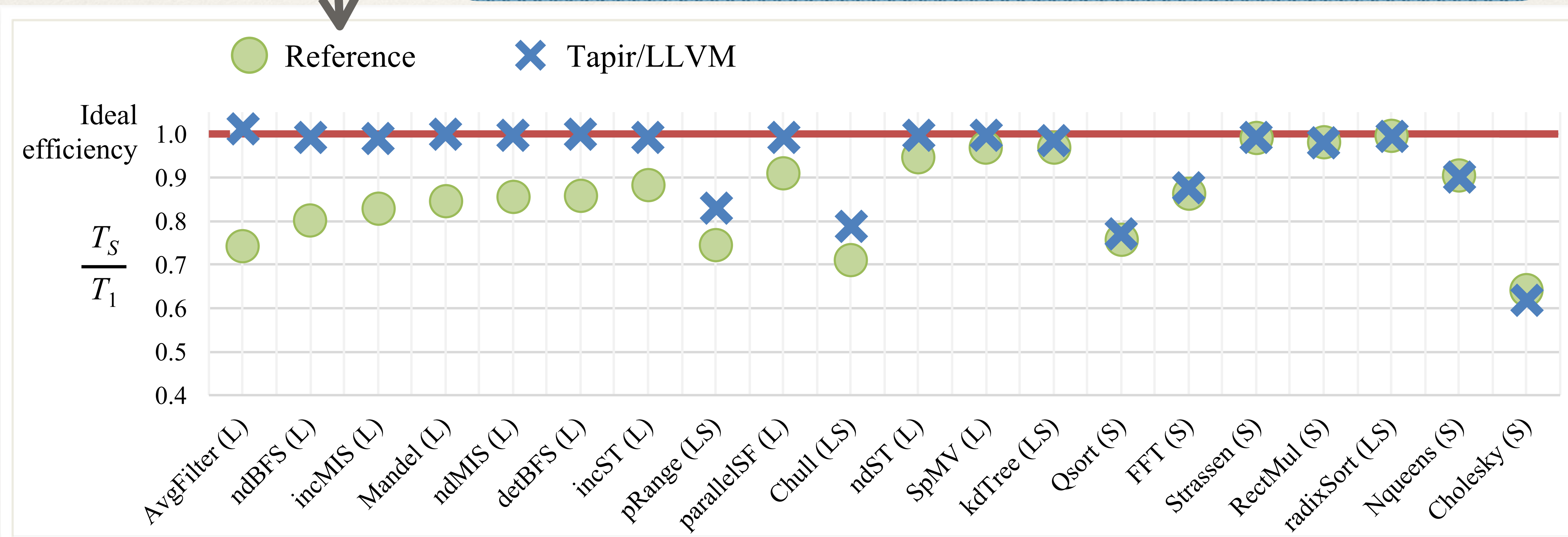
What did we do to **adapt existing analyses and optimizations**?

- 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

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 Optimization: Scheduling

- 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: Scheduling

- 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

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;
}
```

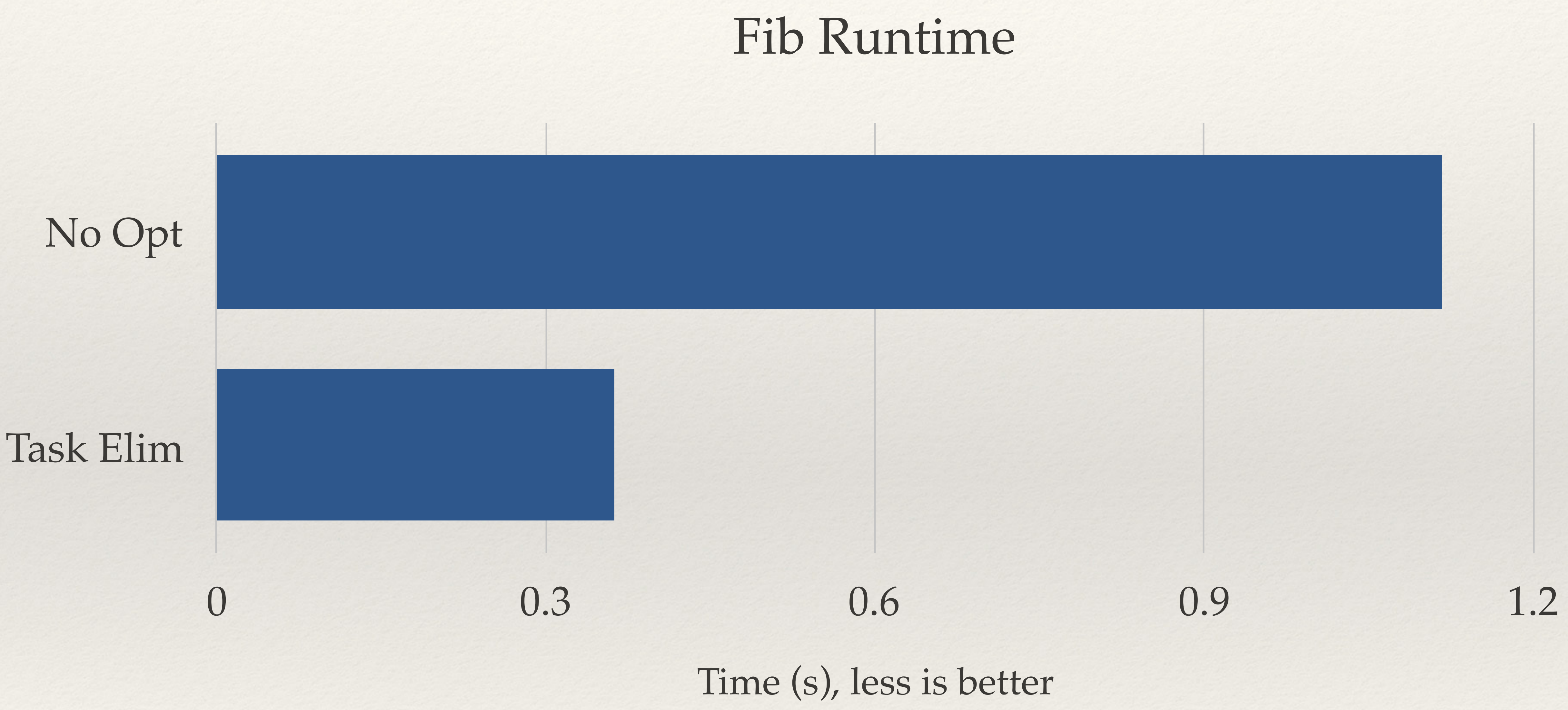
Sounds trivial, but especially useful for OpenMP!

Example Opt: Task Elimination

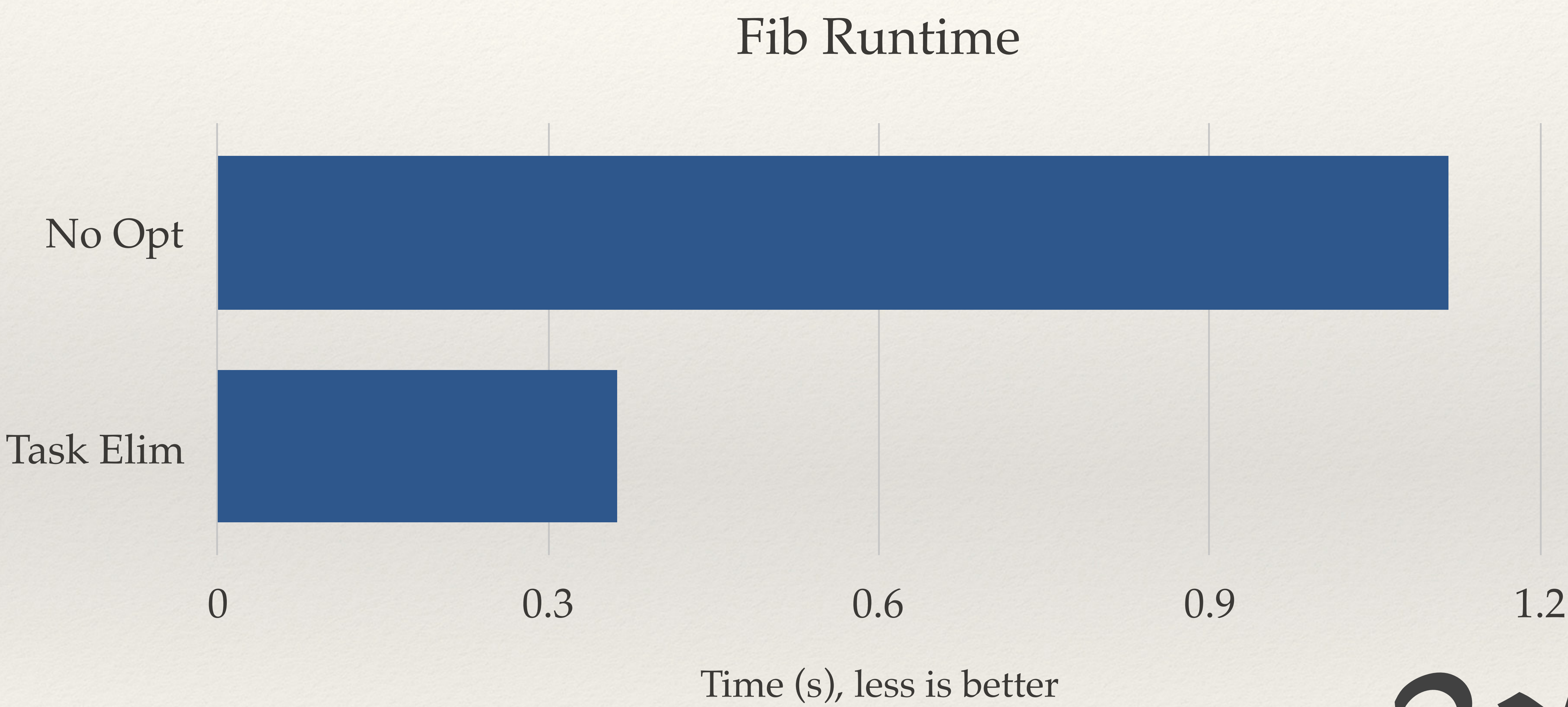
```
void fib(int n) {
  if (n < 2) return n;
  int x, y;
  #pragma omp task shared(x)
  x = fib(n-1);
  #pragma omp task shared(y)
  y = fib(n-2);
  #pragma omp taskwait
  return x+y;
}
```

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

Case Study: Task Elimination



Case Study: Task Elimination



~3x

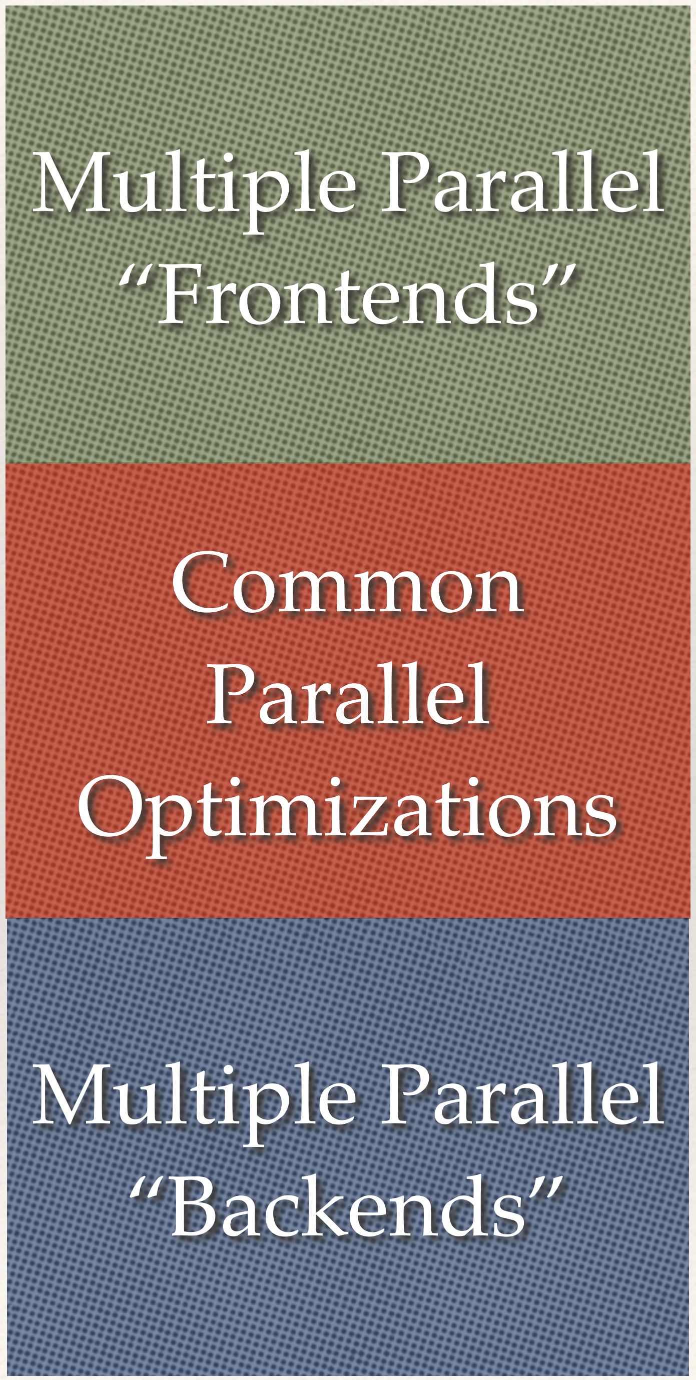
Parallel Optimizations Today

- 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

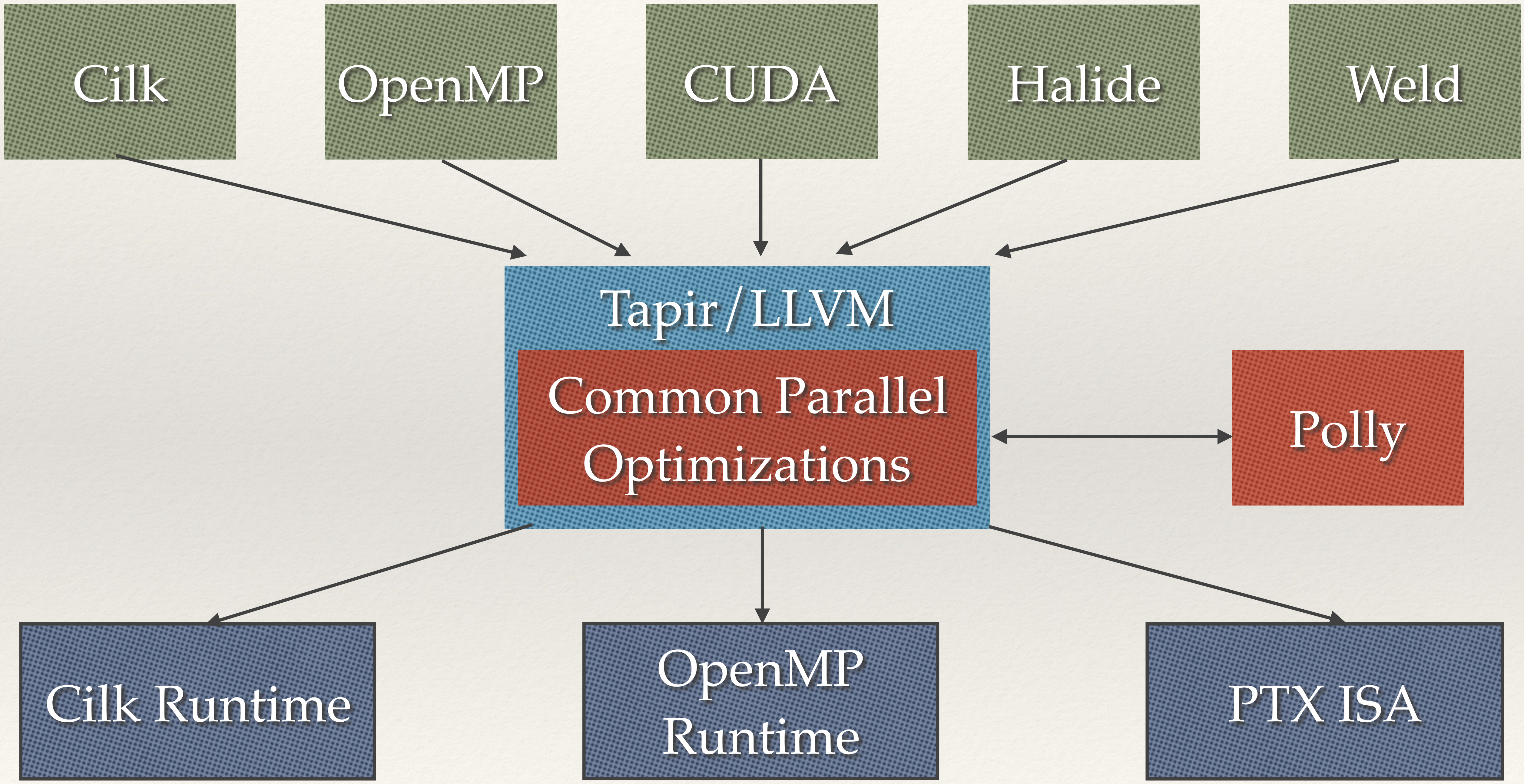
Cilk Frontend	OpenMP Frontend	Halide Frontend	Weld Frontend
Cilk Parallel Optimizations (shrink wrap)	OMP Parallel Optimizations (strip mine)	Halide Parallel Optimizations (scheduling)	Weld Parallel Optimizations, LICM
LLVM w/ Cilk Runtime Calls	LLVM w/ OpenMP Runtime Calls	LLVM w/ Halide Runtime Calls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

Rhino: The Parallel Compiler Dream

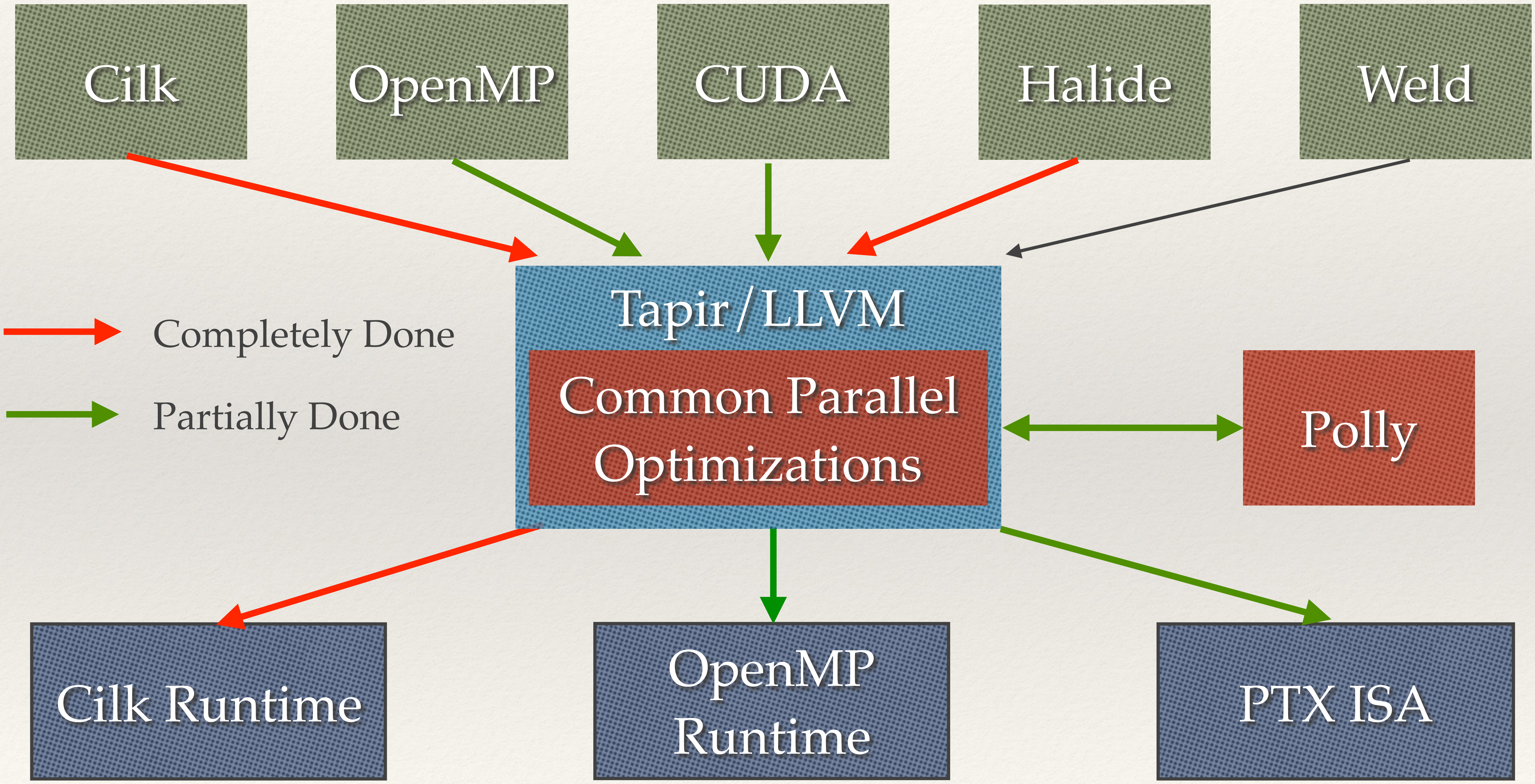


- Tapir is a nice way of representing and working with parallel programs
- Use Tapir as a common parallel intermediate representation for various parallel frontends and backends
- Benefits
  - Enable cross-framework compilation
  - Have one set of common parallel optimizations that can be shared by all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

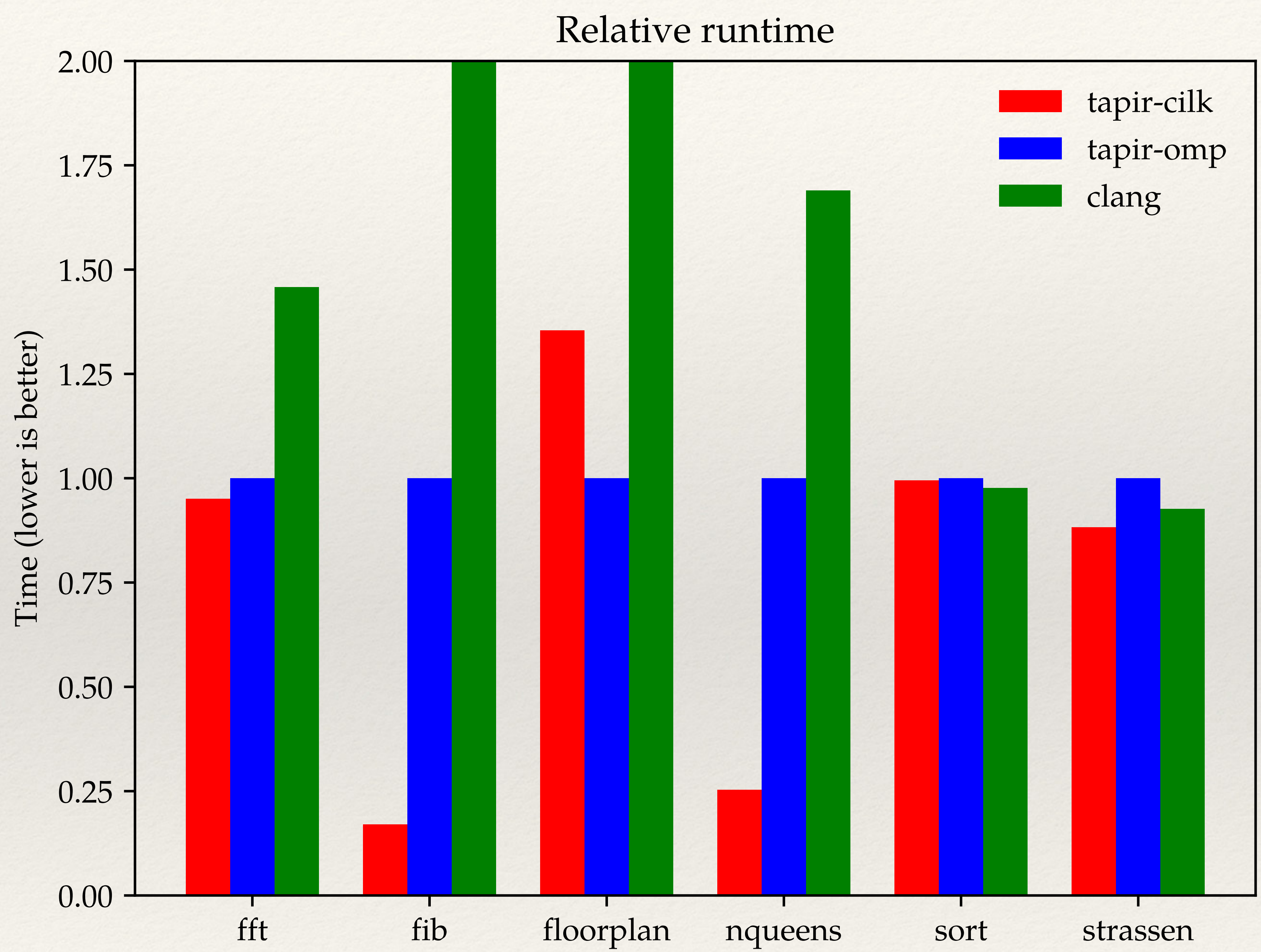
Rhino: The Parallel Compiler Dream



Rhino: The Parallel Compiler Dream

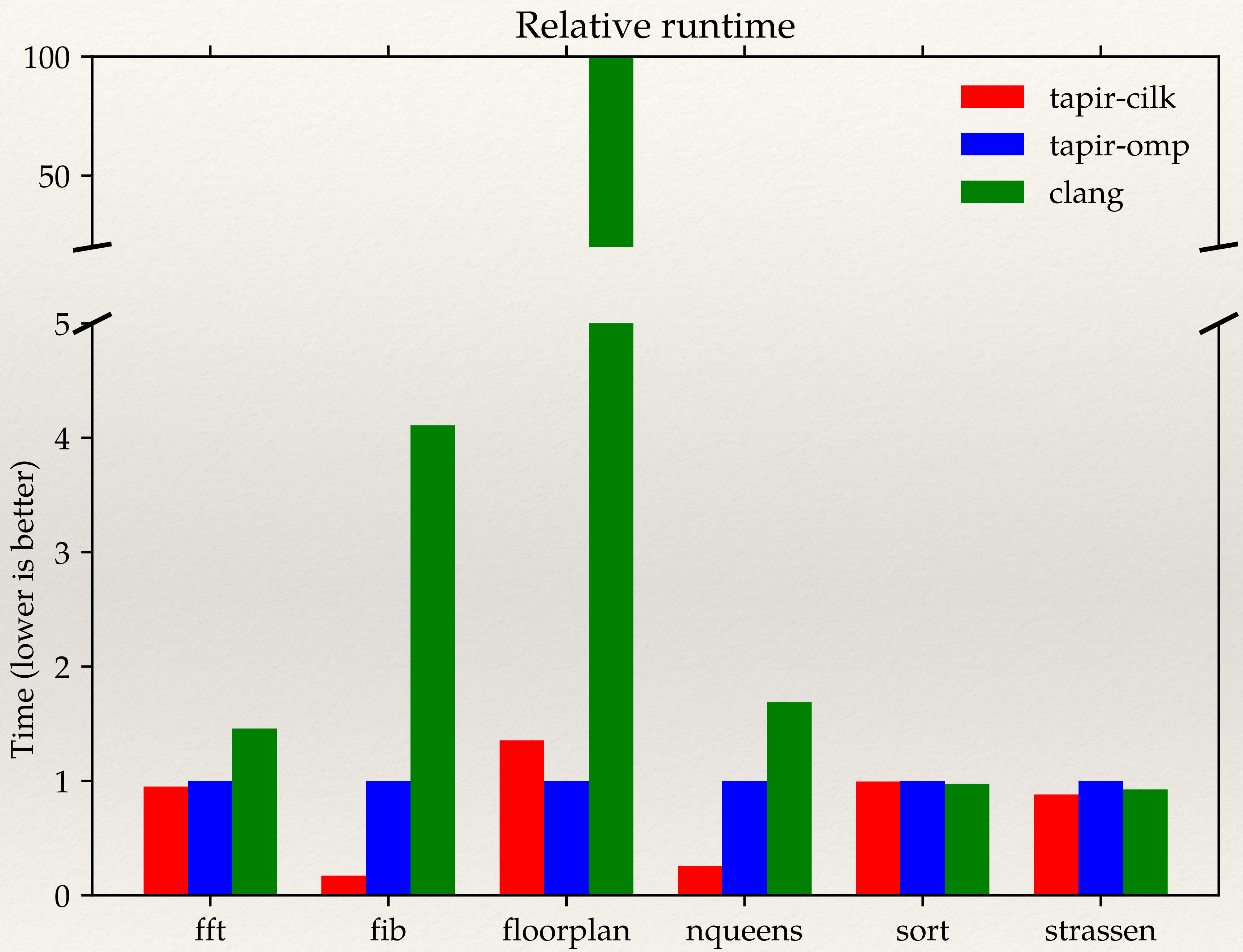


Parallel Runtime Choice



Examples from Barcelona OpenMP benchmark suite

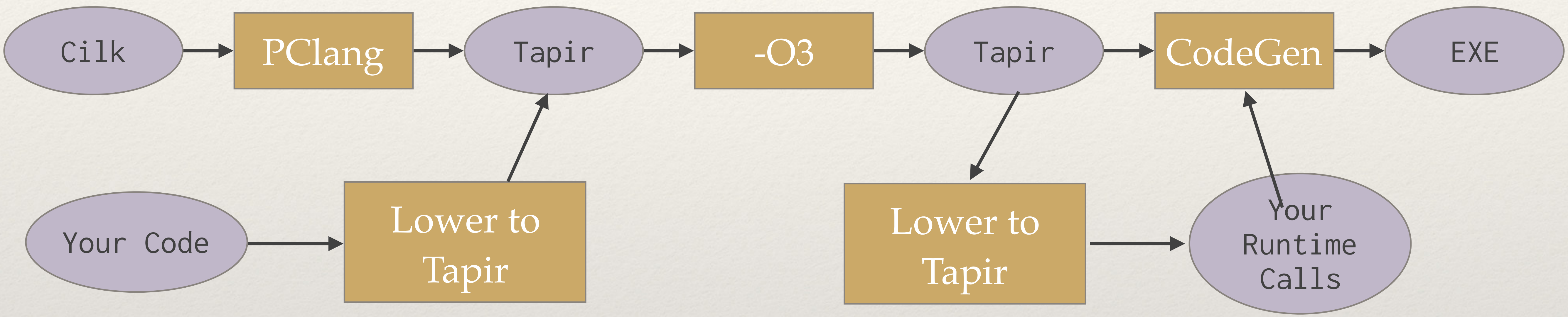
Parallel Runtime Choice



Examples from Barcelona OpenMP benchmark suite

How to Optimize YOUR Parallel Code

Tapir/LLVM pipeline



- To connect to Tapir, you need to do one or two things:
  - Modify your frontend to emit Tapir instructions when emitting LLVM
  - Add a tapirTarget that will lower tapir instructions to your runtime calls

Adding a Tapir Frontend

- Represent the parallelism in your program using detach'ed CFG's and dependencies using sync instructions / regions (a sync instruction synchronizes all the tasks in the region)

- Let's look at how to lower a parallel for loop from Halide

```
// Make our phi node.
PHINode *phi = builder->CreatePHI(i32_t, 2);
phi->addIncoming(min, preheader_bb);

builder->CreateDetach(body_bb, latch_bb, SyncRegionStart);
builder->SetInsertPoint(body_bb);

BasicBlock *parent_continue_block = continue_block;
continue_block = latch_bb;
Value *parent_sync_region = sync_region;
sync_region = SyncRegionStart;

// Within the loop, the variable is equal to the phi value
sym_push(op->name, phi);

// Emit the loop body
codegen(op->body);

return_with_error_code(ConstantInt::get(i32_t, 0));

builder->SetInsertPoint(latch_bb);

// Update the counter
Value *next_var = builder->CreateNSWAdd(phi, ConstantInt::get(i32_t, 1));

// Add the back-edge to the phi node
phi->addIncoming(next_var, builder->GetInsertBlock());

// Maybe exit the loop
Value *end_condition = builder->CreateICmpNE(next_var, max);
builder->CreateCondBr(end_condition, loop_bb, after_bb);

builder->SetInsertPoint(after_bb);

// Pop the loop variable from the scope
sym_pop(op->name);

builder->CreateSync(sync_bb, SyncRegionStart);
```

Put the body of the loop in the detach

Not shown here, but reattach at end of task in codegen

Join all the tasks together at the end

Adding a Tapir Backend

- Three stages / options for lowering: Polly SCoP-based, loop-based, task based
- Higher level stages will run before lower level (i.e. you can create tasks during loop-based lowering, which will be lowered later)

Building A Backend

```
class TapirTarget {
public:
  virtual ~TapirTarget() {};
  //! For use in loopspawning grainsize calculation
  virtual Value *GetOrCreateWorker8(Function &F) = 0;
  virtual void createSync(SyncInst &inst,
                          ValueToValueMapTy &DetachCtxToStackFrame) = 0;
  virtual Function *createDetach(DetachInst &Detach,
                                 ValueToValueMapTy &DetachCtxToStackFrame,
                                 DominatorTree &DT, AssumptionCache &AC) = 0;
  virtual bool shouldProcessFunction(const Function &F);
  virtual void preProcessFunction(Function &F) = 0;
  virtual void postProcessFunction(Function &F) = 0;
  virtual void postProcessHelper(Function &F) = 0;
  virtual bool processMain(Function &F) = 0;
  virtual bool processLoop(LoopSpawningHints LSH, LoopInfo &LI, ScalarEvolution &SE, DominatorTree &DT,
                           AssumptionCache &AC, OptimizationRemarkEmitter &ORE) = 0;
  //! Helper to perform DAC
  bool processDACLoop(LoopSpawningHints LSH, LoopInfo &LI, ScalarEvolution &SE, DominatorTree &DT,
                           AssumptionCache &AC, OptimizationRemarkEmitter &ORE);
};
```

Building A Backend

```
llvm::MyBackend::MyBackend(){}

void llvm::MyBackend::preProcessFunction(Function &F) {}
void llvm::MyBackend::postProcessFunction(Function &F) {}
void llvm::MyBackend::postProcessHelper(Function &F) {}
bool llvm::MyBackend::processMain(Function &F) {
  return false;
}
bool llvm::MyBackend::processLoop(LoopSpawningHints LSH, LoopInfo &LI, ScalarEvolution &SE, DominatorTree &DT,
                    AssumptionCache &AC, OptimizationRemarkEmitter &ORE) {
  return false;
}
```

- Don’t need to implement pieces we don’t need
- Our “backend” doesn’t require special modification of functions, main, or handles loop differently (though it could if we desired)

Building A Backend

```
llvm::MyBackend::MyBackend(){}

void llvm::MyBackend::preProcessFunction(Function &F) {}
void llvm::MyBackend::postProcessFunction(Function &F) {}
void llvm::MyBackend::postProcessHelper(Function &F) {}
bool llvm::MyBackend::processMain(Function &F) {
  return false;
}
bool llvm::MyBackend::processLoop(LoopSpawningHints LSH, LoopInfo &LI, ScalarEvolution &SE, DominatorTree &DT,
                    AssumptionCache &AC, OptimizationRemarkEmitter &ORE) {
  return false;
}
```

- Don't need to implement pieces we don't need
- Our "backend" doesn't require special modification of functions, main, or handles loop differently (though it could if we desired)

```
//Process instruction statement into runtime calls
Function *llvm::MyBackend::createDetach(DetachInst &detach,
                                         ValueToValueMapTy &DetachCtxToStackFrame,
                                         DominatorTree &DT, AssumptionCache &AC) {
  auto VoidTy = Type::getVoidTy(detach.getContext());
  auto Int8Ty = Type::getInt8Ty(detach.getContext());
  auto Int8PtrTy = PointerType::getUnqual(Int8Ty);
  auto M = detach.getParent()->getParent()->getParent();
  BasicBlock *detB = detach.getParent();
  BasicBlock *Spawned  = detach.getDetached();
  BasicBlock *Continue = detach.getContinue();


  CallInst *cal = nullptr;
  Function *extracted = extractDetachBodyToFunction(detach, DT, AC, &cal, ".bnd");
  assert(extracted && "could not extract detach body to function");

  // Unlink the detached CFG in the original function.  The heavy lifting of
  // removing the outlined detached-CFG is left to subsequent DCE.

  // Replace the detach with a branch to the continuation.
  BranchInst *ContinueBr = BranchInst::Create(Continue);
  ReplaceInstWithInst(&detach, ContinueBr);

  // Rewrite phis in the detached block.
  {
    BasicBlock::iterator BI = Spawned->begin();
    while (PHINode *P = dyn_cast<PHINode>(BI)) {
      P->removeIncomingValue(detB);
      ++BI;
    }
  }

  IRBuilder<> builder(cal);
  std::vector<Value *> Args = {builder.CreatePointerCast(extracted, Int8PtrTy)};
  for(unsigned i=0; i<cal->getNumArgOperands(); i++) {
    Args.push_back(cal->getArgOperand(i));
  }
  Type *TypeParams[] = {Int8PtrTy};
  FunctionType *FnTy = FunctionType::get(VoidTy, TypeParams, /*isVarArg*/true);
  CallInst *runtimecall = CallInst::Create(M->getOrInsertFunction("mybackend_detach", FnTy), Args);

  ReplaceInstWithInst(cal, runtimecall);

  return extracted;
}
```

Outline task

Ignore previous task

Call task with runtime rather than direct call

Building A Backend

```
//Process sync instruction into runtime calls
void llvm::MyBackend::createSync(SyncInst &SI, ValueToValueMapTy &DetachCtxToStackFrame) {
  auto M = SI.getParent()->getParent()->getParent();
  auto VoidTy = Type::getVoidTy(SI.getContext());
  auto Int8Ty = Type::getInt8Ty(SI.getContext());
  auto Int8PtrTy = PointerType::getUnqual(Int8Ty);
  IRBuilder<> builder(&SI);
  std::vector<Value *> Args = {builder.CreatePointerCast(SI.getParent()->getParent(), Int8PtrTy)};
  Type *TypeParams[] = {Int8PtrTy};
  FunctionType *FnTy = FunctionType::get(VoidTy, TypeParams, /*isVarArg*/false);
  CallInst *call = builder.CreateCall(M->getOrInsertFunction("mybackend_sync", FnTy), Args);

  // Replace the detach with a branch to the continuation.
  BranchInst *PostSync = BranchInst::Create(SI.getSuccessor(0));
  ReplaceInstWithInst(&SI, PostSync);
}
```

- Our sample backend simply calls a syncronize instruction, with the local function pointer (which is perhaps used to modify a structure of tasks in the function)

Building A Backend

```
//Get number of workers * 8
Value* llvm::MyBackend::GetOrCreateWorker8(Function &F) {
  auto M = F.getParent();
  auto Int32Ty = Type::getInt32Ty(F.getContext());
  IRBuilder<> builder(F.getEntryBlock().getFirstNonPHIOrDbgOrLifetime());
  std::vector<Value *> Args = {};
  FunctionType *FnTy = FunctionType::get(Int32Ty, /*isVarArg*/false);
  CallInst *call = builder.CreateCall(M->getOrInsertFunction("get_num_workers", FnTy), Args);
  Value *P8 = builder.CreateMul(call, ConstantInt::get(Int32Ty, 8));
  return P8;
}
```

- The number of workers is used for the default loop processing to coarsen base cases
- In our sample backend this is a simple runtime call

Building A Backend

- That's it!
- All together (including the header) ~150 LOC to implement a backend
- We can take advantage of all the Tapir optimizations and we automatically have frontend language (Cilk, OpenMP, etc) that compiles to Tapir as valid programs / benchmarks!

```
wmoses@beast:~/git/Tapir/build/bin (ptx) $ ./opt -S oldfib.ll -tapir2target -tapir-target=mybackend
; ModuleID = 'oldfib.ll'
source_filename = "oldfib.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @fib(i32 %n) local_unnamed_addr #0 {
entry:
  %x = alloca i32, align 4
  %syncreg = tail call token @llvm.syncregion.start()
  %cmp = icmp slt i32 %n, 2
  br i1 %cmp, label %return, label %if.end

if.end:                                           ; preds = %entry
  %x.0.x.0..sroa_cast = bitcast i32* %x to i8*
  call void @llvm.lifetime.start.p0i8(i64 4, i8* nonnull %x.0.x.0..sroa_cast)
  call void (i8*, ...) @mybackend_detach(i8* bitcast (void (i32, i32*)* @fib_det.achd.bnd to i8*), i32 %n, i32* %x)
  br label %det.cont

det.achd:                                         ; No predecessors!
  unreachable

det.cont:                                         ; preds = %if.end
  %sub1 = add nsw i32 %n, -2
  %call2 = tail call i32 @fib(i32 %sub1)
  call void @mybackend_sync(i8* bitcast (i32 (i32)* @fib to i8*))
  br label %sync.continue

sync.continue:                                    ; preds = %det.cont
  %x.0.load8 = load i32, i32* %x, align 4
  %add = add nsw i32 %x.0.load8, %call2
  call void @llvm.lifetime.end.p0i8(i64 4, i8* nonnull %x.0.x.0..sroa_cast)
  br label %return

return:                                           ; preds = %sync.continue, %entry
  %retval.0 = phi i32 [ %add, %sync.continue ], [ 1, %entry ]
  ret i32 %retval.0
}
```

Tutorial 3: Shared Tools

- Tools built for one framework can be used by any framework that uses Tapir
- Let's get a look at one tool, a race detector: cd tapir-tutorial/san
- Useful for detecting bugs in code, but ALSO for bugs in your frontend/backend (say accidentally making a private variable public)

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 and tools
  - Mix-and-match parallel runtimes
- Ongoing development (bug fixes, new optimizations, etc).
- Available on GitHub!
  https://github.com/wsmoses/Tapir-LLVM.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)

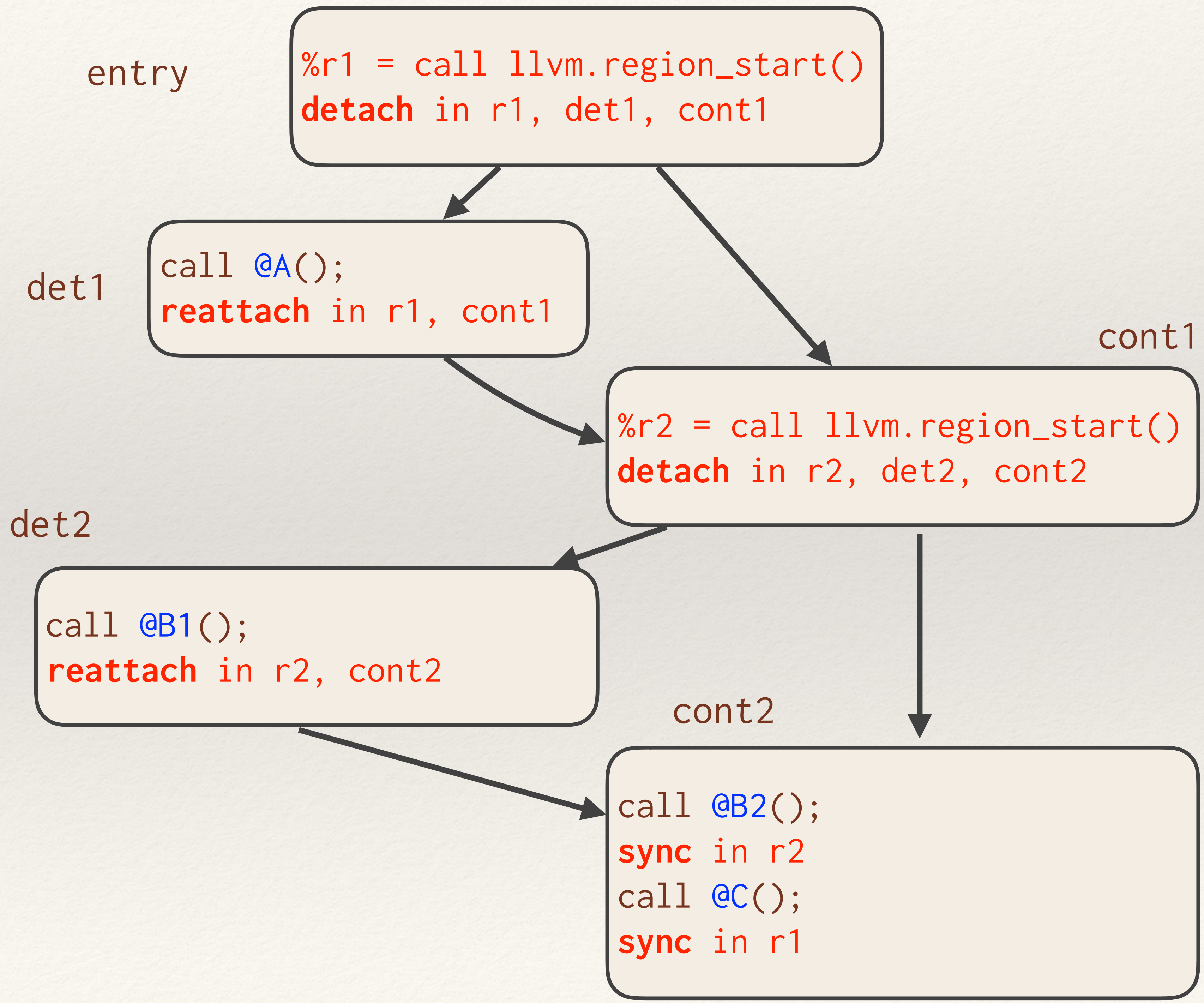
```
f = detach foo();
ϕ = {}

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

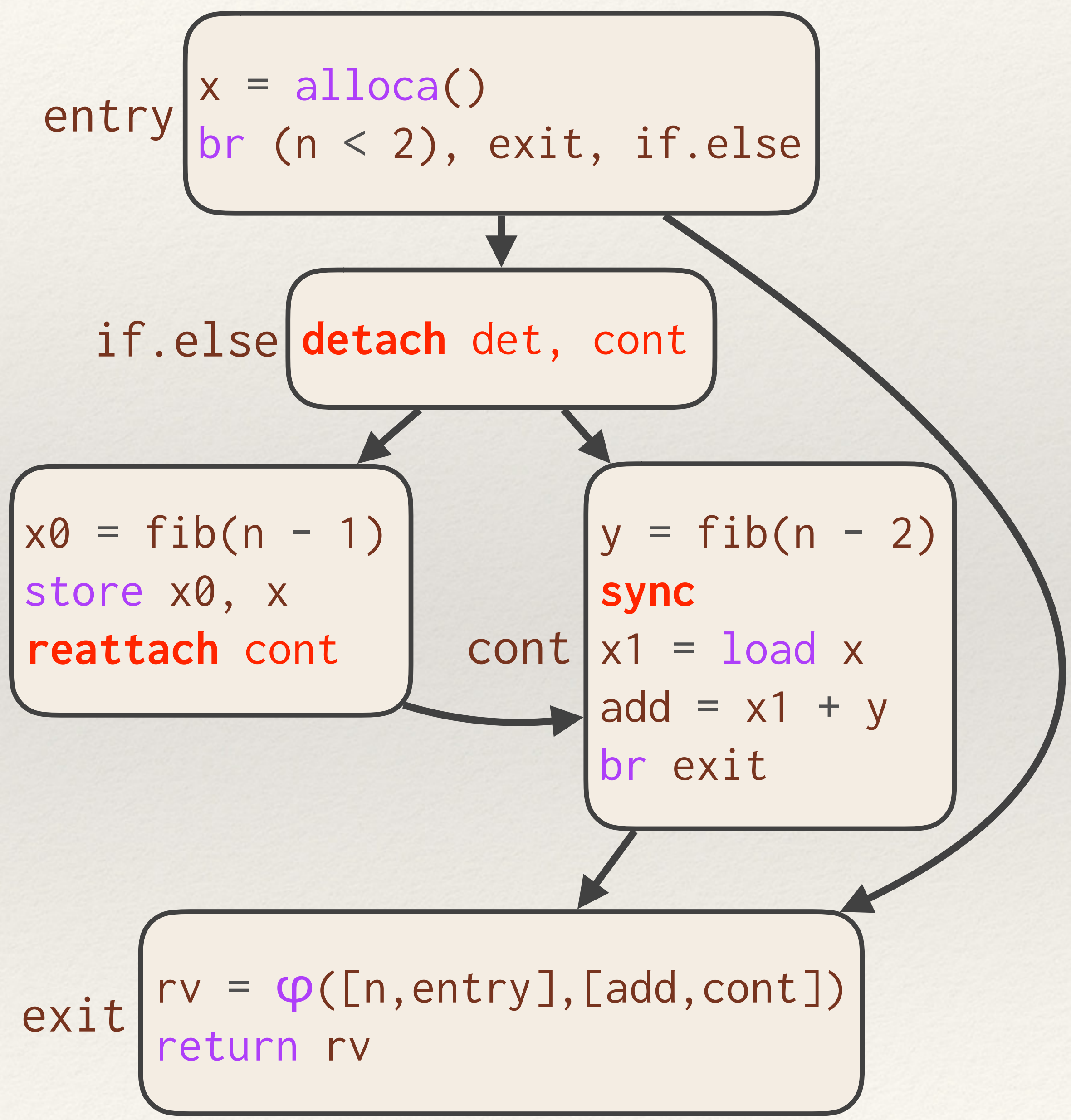
Idea 2: Individualized Sync

```
void main() {

  detach A();

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

  C();
  sync;
}
```



Maintaining Correctness

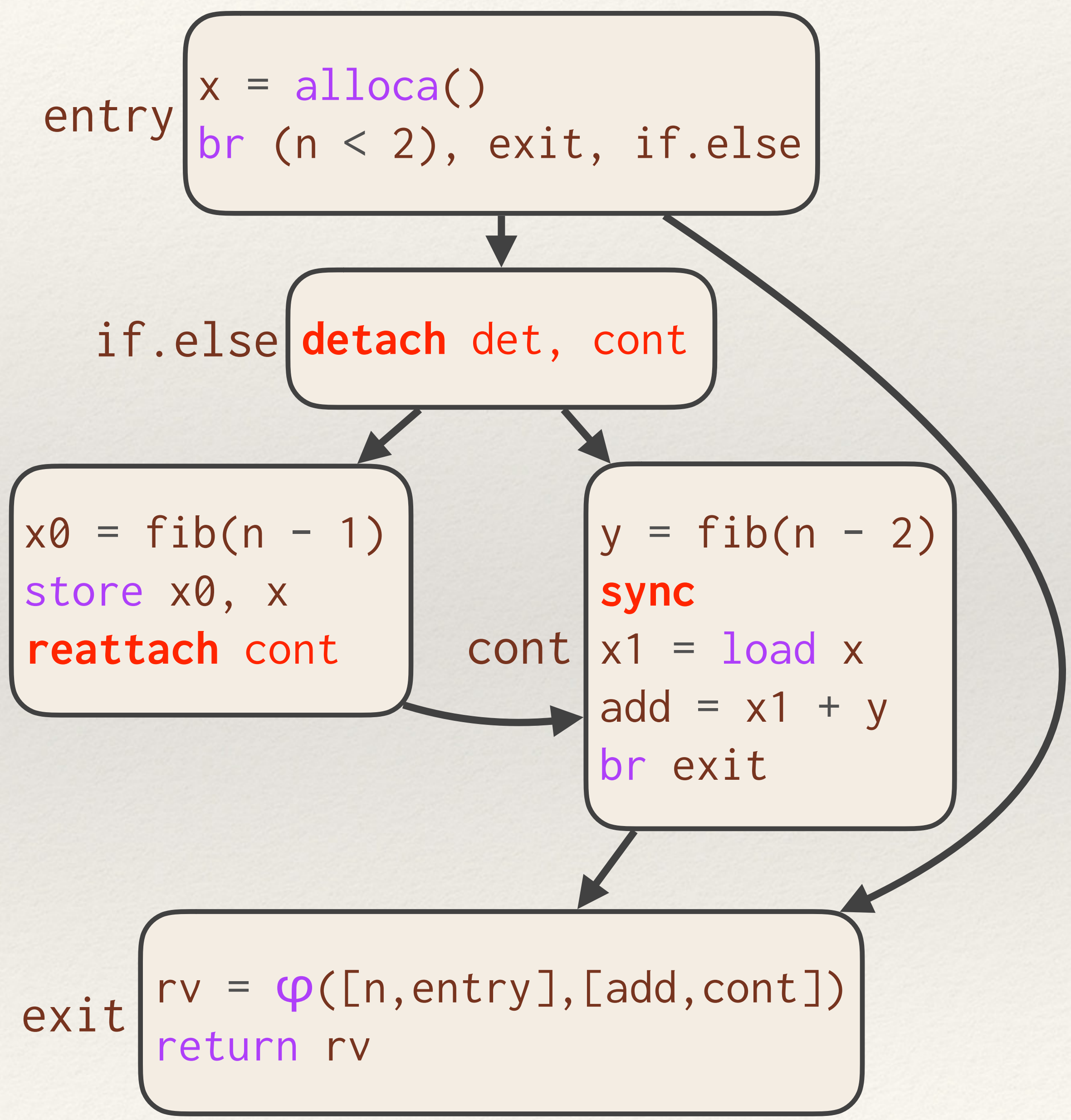
Problem: How does the compiler ensure that code motion does not introduce a determinacy race into otherwise race-free code?



Maintaining Correctness

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

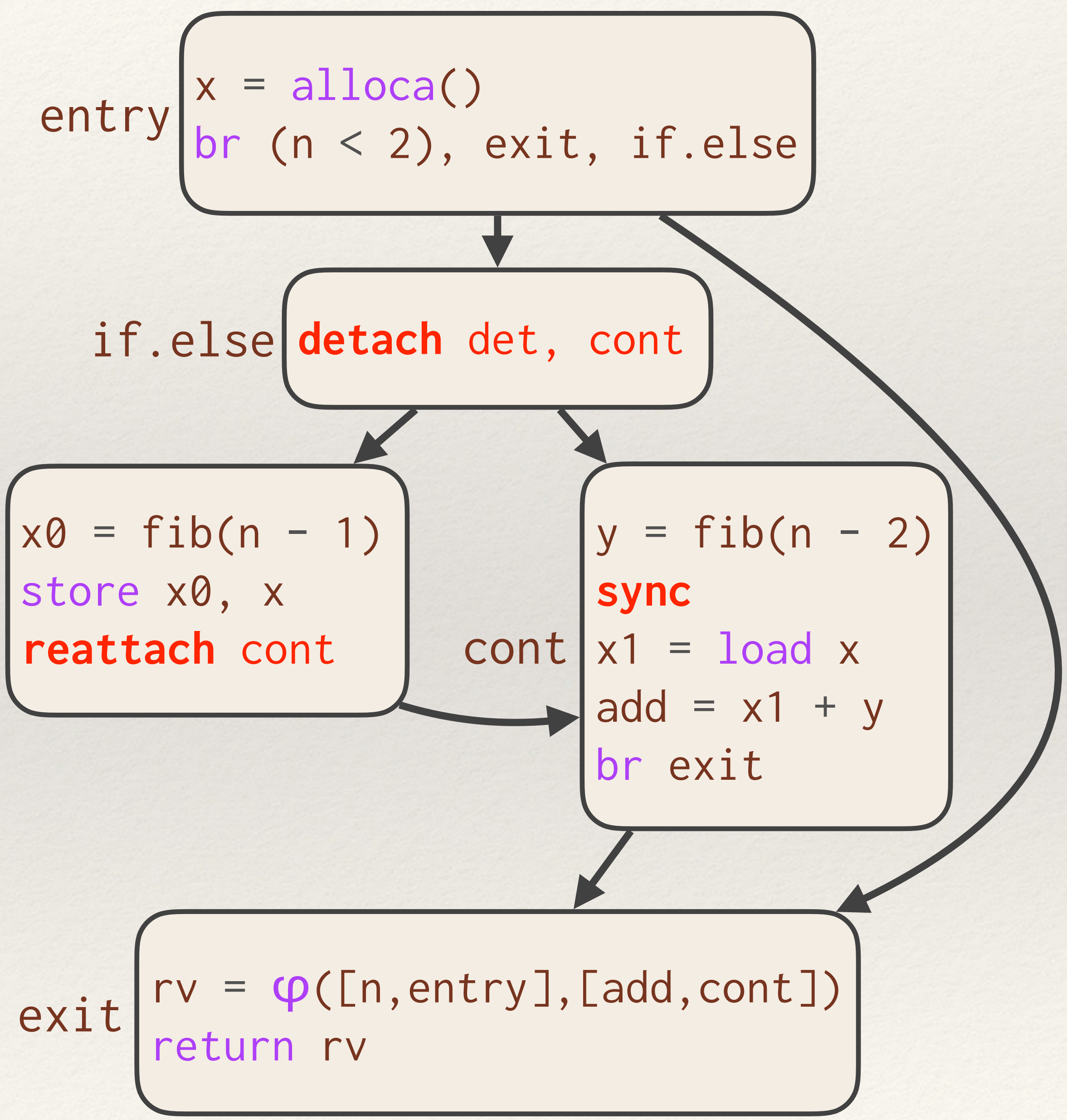
- It suffices to consider moving memory operations around each new instruction.



Maintaining Correctness

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

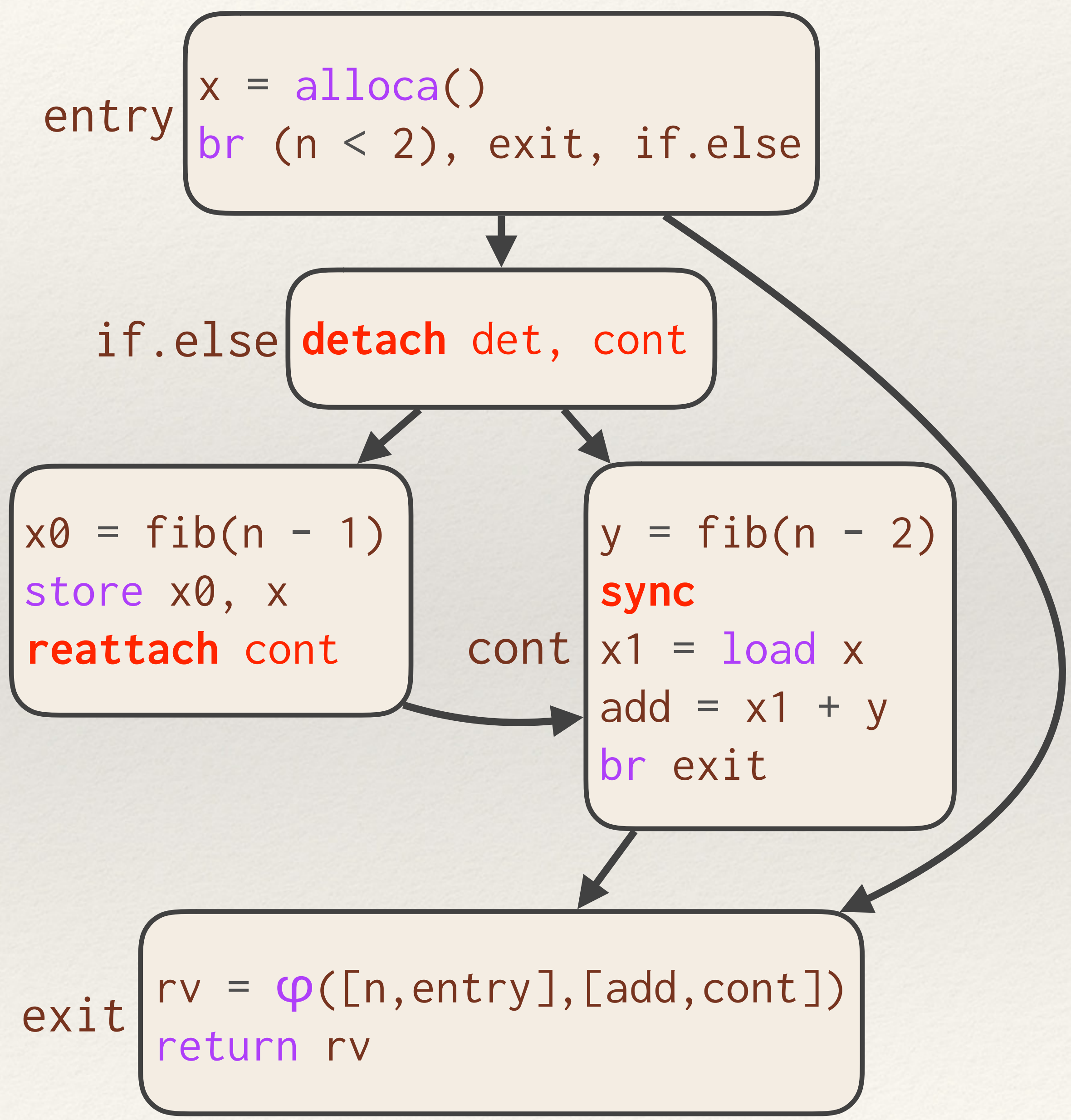
- It suffices to consider moving memory operations around each new instruction.
- Moving code above a `detach` or below a `sync` serializes it and is always valid.



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.