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

Best paper at PPoPP 2017

Authors: Schardl, T. B., Moses, W. S., & Leiserson, C. E.

Presenters: Tiancheng Ge, Fanzhong Kong, Zihan Li

Motivation

- Optimize parallel programs.

```c
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);
}
```

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

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

- If the code is serial, this optimization is very easily using LLVM.
- But the program is parallel...

Motivation

Mainstream compilers

- At *front end*, parallel code → other representations.
- cannot recognize the new representation at *middle end* → hard to do optimization.

Previous approaches do to optimization:

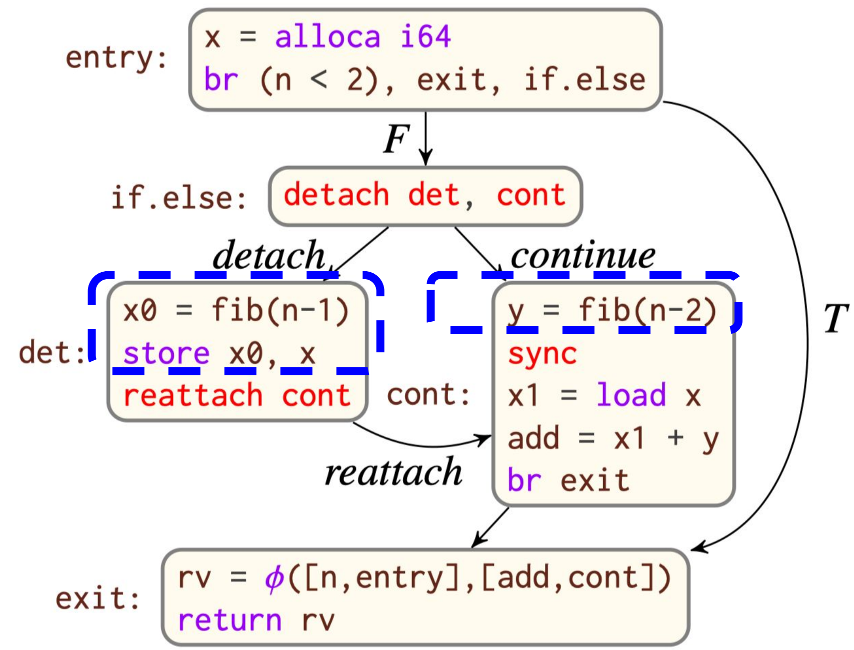
- Intrinsic functions to mark parallelism. LLVM support ✘
- Use separate IR for parallel code. Too much work ✘
- *Augment* existing IR for parallel code, and reuse existing optimizations in LLVM.

Overview: Tapir (Task-based Asymmetric Parallel IR)

- Tapir uses three additional LLVM IR instructions:
- detach *detached block, continuation block*
  - Create a detached thread
- reattach, *continuation block*
  - Terminate the thread
- sync (thread.join())
  - Wait other threads to finish
- Example:

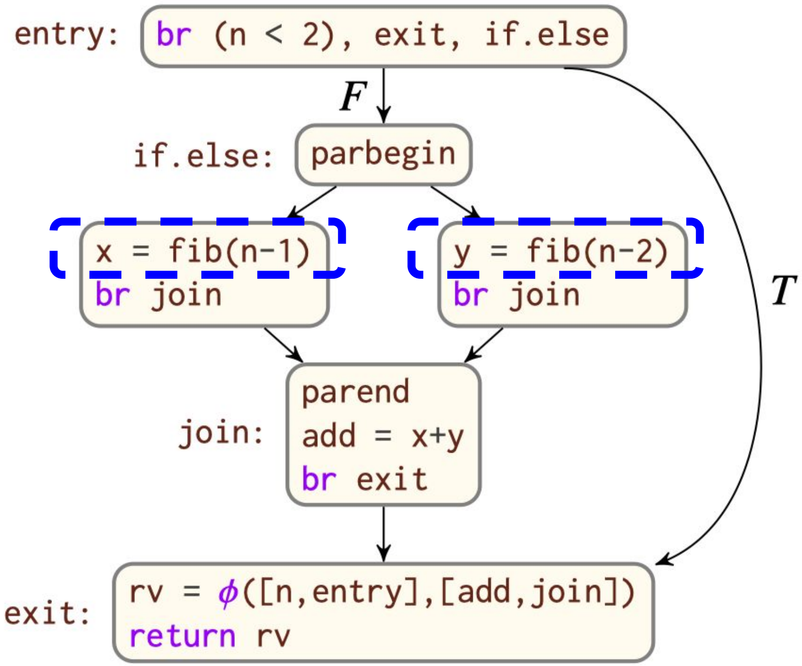
  Execute two pieces of code

  in parallel.

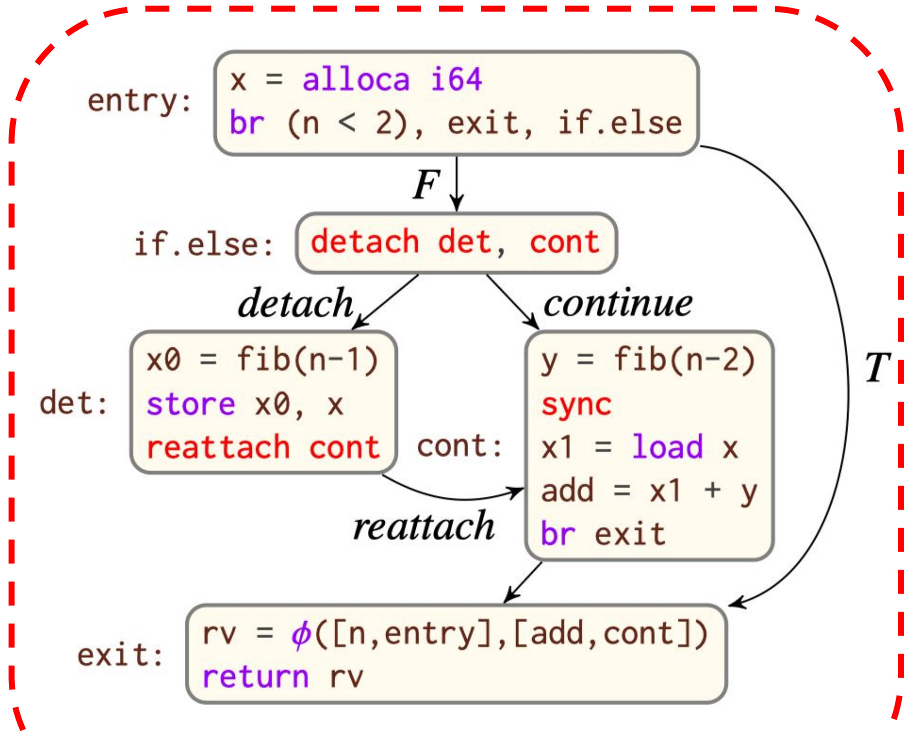


Symmetry vs. Asymmetry

- Tapir uses *asymmetric* parallel tasks.



Symmetric

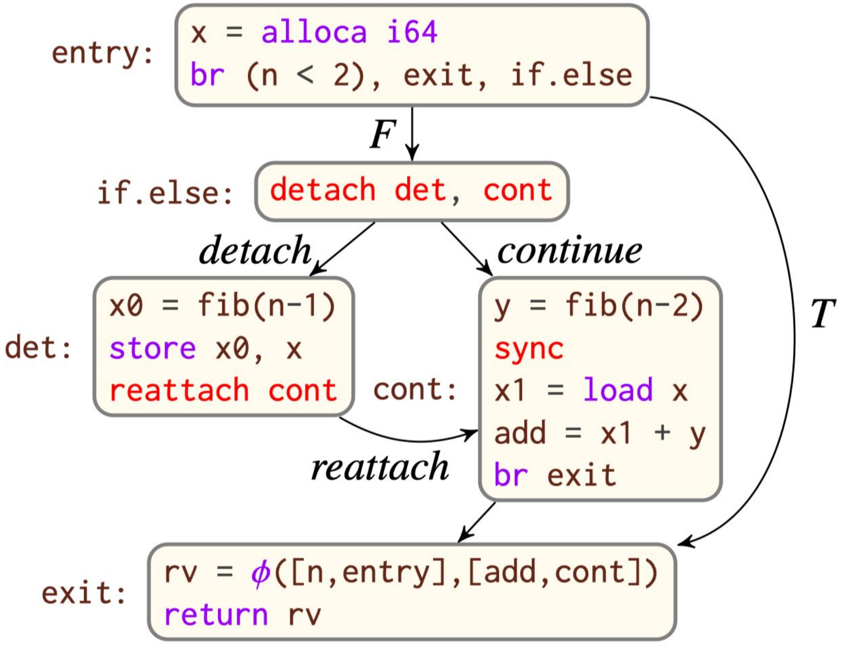


Asymmetric

- Symmetrical parallel assume all the control flow edges must be taken before the join block, which is bad...

Asymmetry

- In asymmetrical parallel setting, we don't need to have a joining point.
- Allows LLVM’s dominator analysis to analyze Tapir programs correctly without any changes.
- Reuse most optimizations in LLVM.



Analysis Pass

Constraints on transformations

- must preserve the program's serial semantics
- must not introduce any new behaviors

Alias analysis

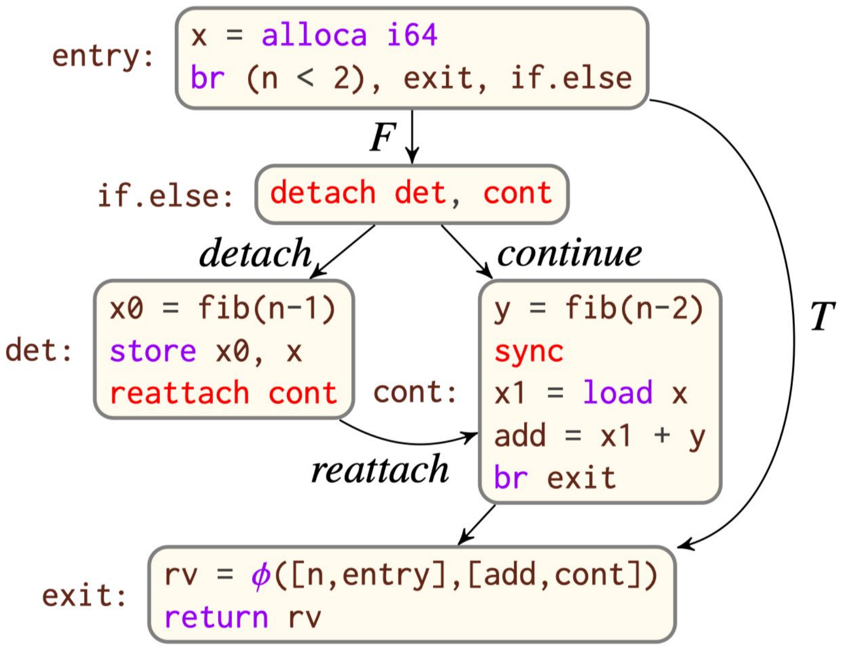
k - load or store

i - detach

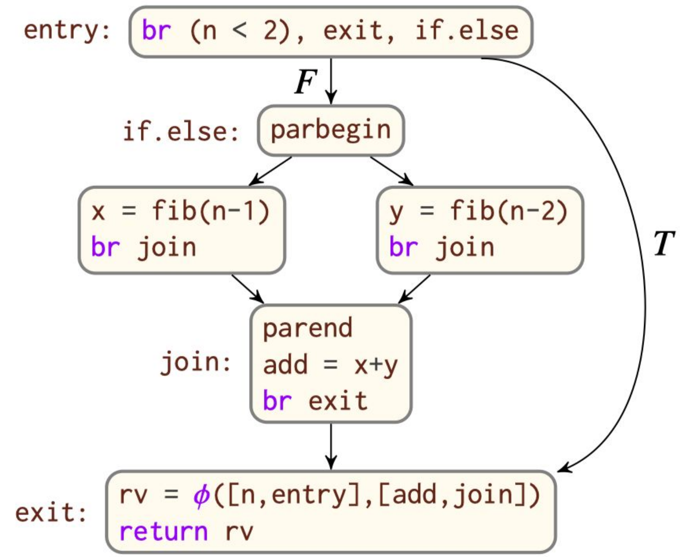
j - sync

1. k moves from before i to after i
2. k moves from after i to before i
3. k moves from before j to after j
4. k moves from after j to before j

Reattach is treated as a compiler fence



Analysis Pass



Dominator analysis

No modification required

Data-flow analysis

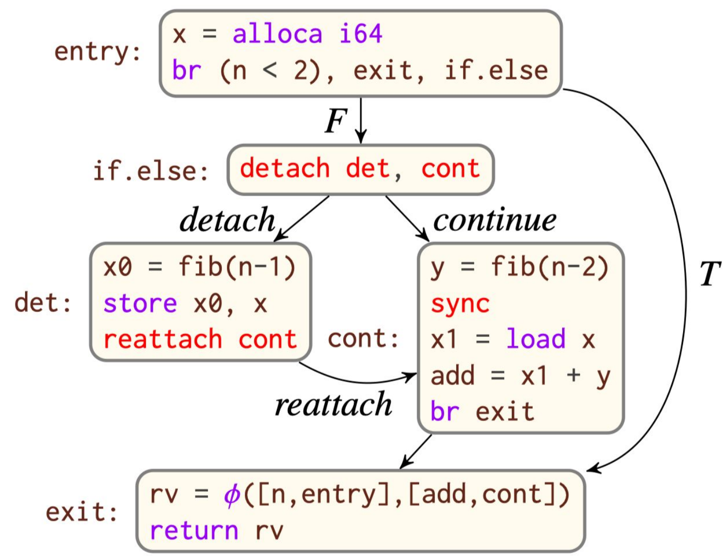
For variables stored in shared memory

$$\text{IN}(b) = \bigcup_{(a,b)\in E} \text{OUT}(a) .$$

For register variables

$$\text{IN}(b) = \bigcup_{(a,b)\in E-E_R} \text{OUT}(a) ,$$

E_R is the set of reattach edges in E



Optimization Pass

Common-subexpression elimination

Loop-invariant code motion

Tail-recursion elimination

Parallel-loop scheduling and lowering

- No modification required
- Modification required
- New optimizations

a

```
void search(int low, int high) {
  if (low == high) search_base(low);
  else {
    cilk_spawn search(low, (low+high)/2);
    search((low+high)/2 + 1, high);
    cilk_sync;
} }
```

b

```
void search(int low, int high) {
  if (low == high) search_base(low);
  else {
    int mid = (low+high)/2;
    cilk_spawn search(low, mid);
    search(mid + 1, high);
    cilk_sync;
} }
```

Common-subexpression elimination

Optimization Pass

Loop-invariant code motion

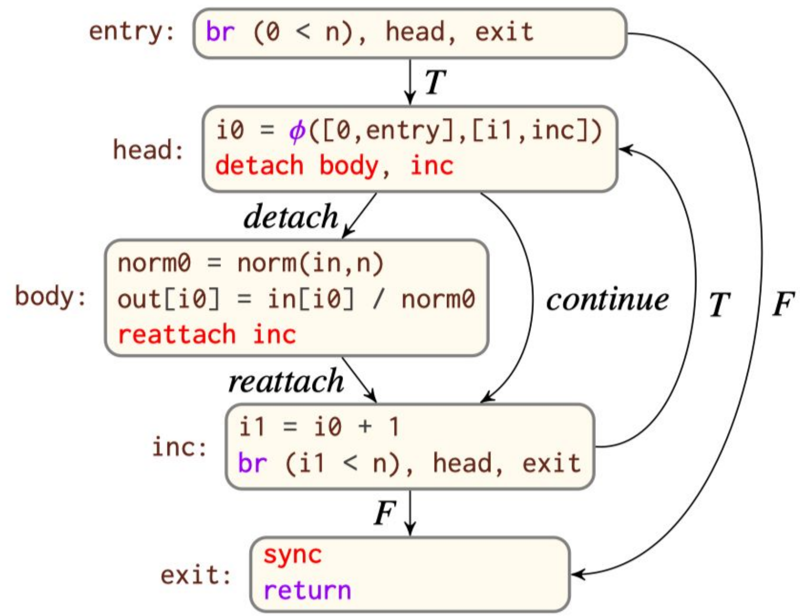
- Src(X) not modified in loop body
- X is the only op to modify dest(X)
- If X is a load or store, then there are no writes to address(X) in loop
- ...
- If X not executed on every iteration, then X must provably not cause exceptions

Problem:

Continue edge shortcuts all body instructions

Solution:

Analyzing the serial version of the loop



Optimization Pass

Tail-recursion elimination

Replace a recursive call at the end of a function with a branch to the start of the function.

In the example: replace function call at line 54 with goto at line 88

Move sync to just before function return

a

```
49 void pqsort(int* start, int* end) {
50   if (start == end) return;
51   int* mid = partition(start, end);
52   swap(end, mid);
53   cilk_spawn pqsort(start, mid);
54   pqsort(mid+1, end);
55   cilk_sync;
56   return;
57 }
```

c

```
78 void pqsort(int* start, int* end) {
79 pqsort_start:
80   if (start == end) {
81     cilk_sync;
82     return;
83   }
84   int* mid = partition(start, end);
85   swap(end, mid);
86   cilk_spawn pqsort(start, mid);
87   start = mid+1;
88   goto pqsort_start;
89 }
```

Optimization Pass

Tail-recursion elimination

Why is it safe to move sync?

Figure b is one level inlining of Figure a

Redundant call to sync at line 71 and line 75

a

```
49 void pqsort(int* start, int* end) {
50   if (start == end) return;
51   int* mid = partition(start, end);
52   swap(end, mid);
53   cilk_spawn pqsort(start, mid);
54   pqsort(mid+1, end);
55   cilk_sync;
56   return;
57 }
```

b

```
58 void pqsort(int* start, int* end) {
59   if (start == end) return;
60   int* mid = partition(start, end);
61   swap(end, mid);
62   cilk_spawn pqsort(start, mid);
63
64   start = mid+1;
65   // Begin inlined code
66   if (start == end) goto join;
67   mid = partition(start, end);
68   swap(end, mid);
69   cilk_spawn pqsort(start, mid);
70   pqsort(mid+1, end);
71   cilk_sync;
72   // End inlined code
73
74 join:
75   cilk_sync;
76   return;
77 }
```

Optimization Pass

Parallel-loop scheduling and lowering

For a parallel loop with a large number of iterations

- schedule the iterations in a recursive divide-and-conquer fashion

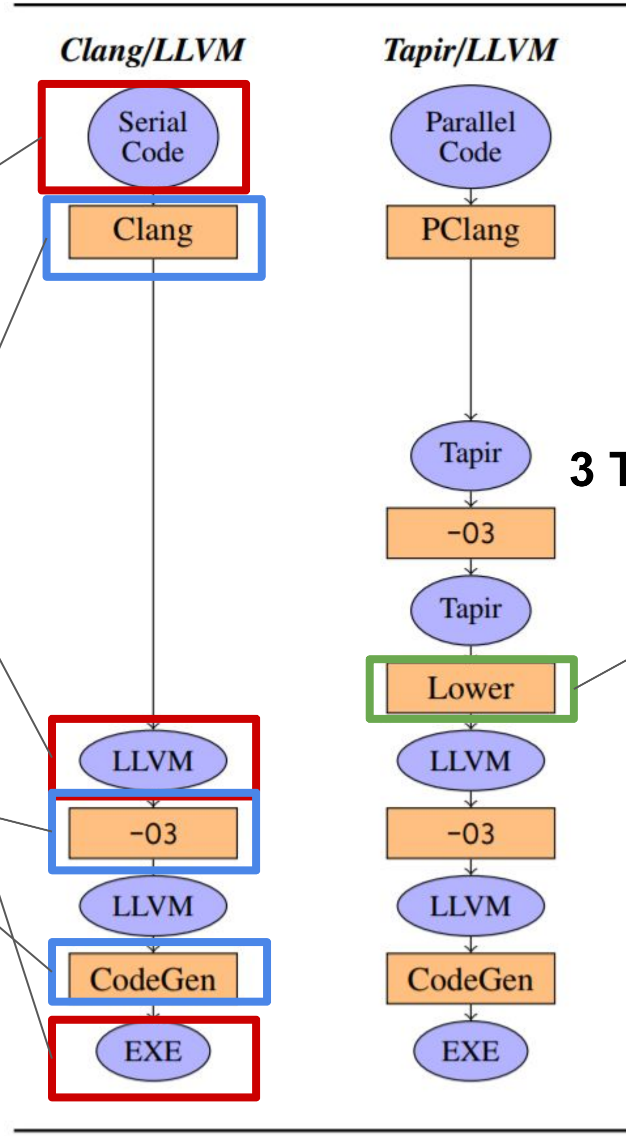
For parallel loops with few iterations

- simply spawning off the iterations

Model Pipeline - Legend

Entity: Code status

Action: Code Trans.



3 Tapir Instr. →LLVM IR

Benchmarking

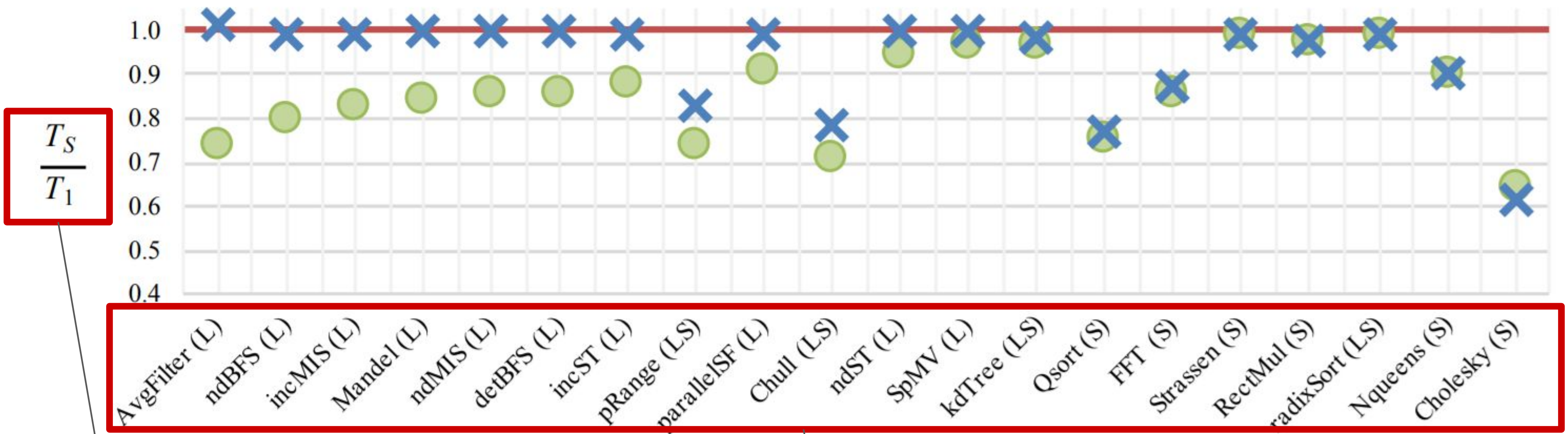
20 Benchmark programs in total

- **From Intel/MIT/CMU Cilk code Sample**

Run over AWS c4.8xlarge spot instance

Run 10 times and take the minimum (not average)

Data Evaluation

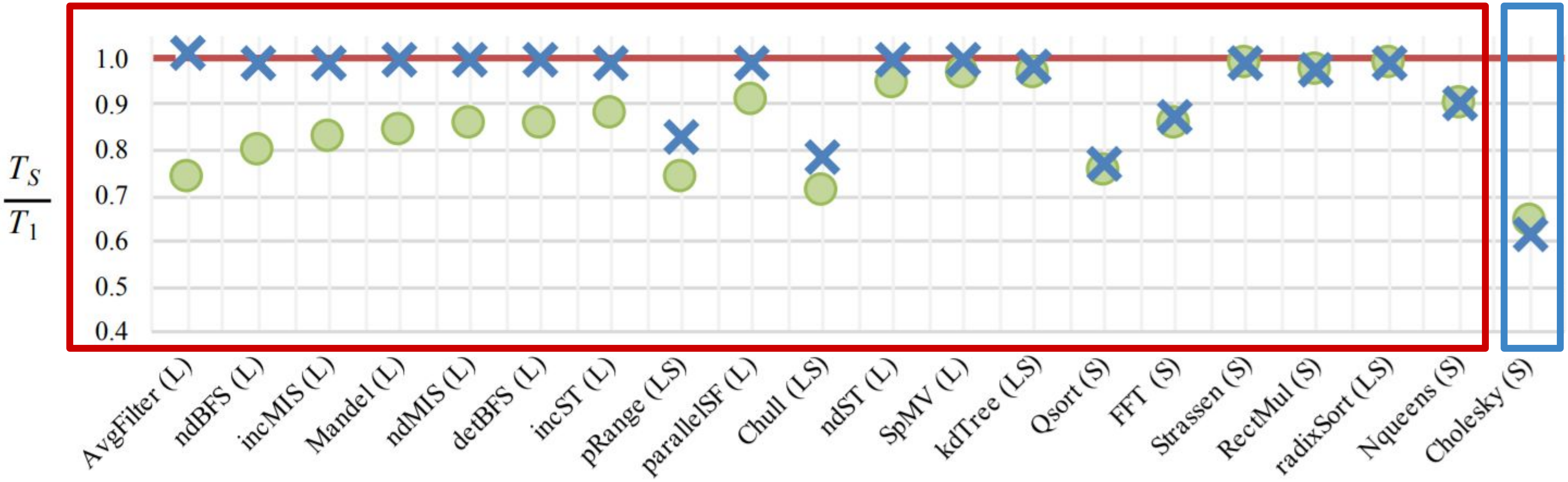


Speed up

Want this **larger**

Different benchmarks

Data Evaluation



- **Red part:** Tapir behaves much better (normal case)
- **Blue part**: Tapir behaves even worse
  - Some **additional** llvm optimizations before
  - **Fixed** this issue in 2019
- The same pattern holds for 18 cores experiment

Discussion & Conclusion

Pros

- **Ease** of implementation (**0.15 %** code to modify)
- **No extra efforts** for developer
- **Extensible** (easy to add pass)

Cons

- **No** comparison with other implementations
- Can only be applied to **static** compilation (No JIT, dynamic)

Q & A

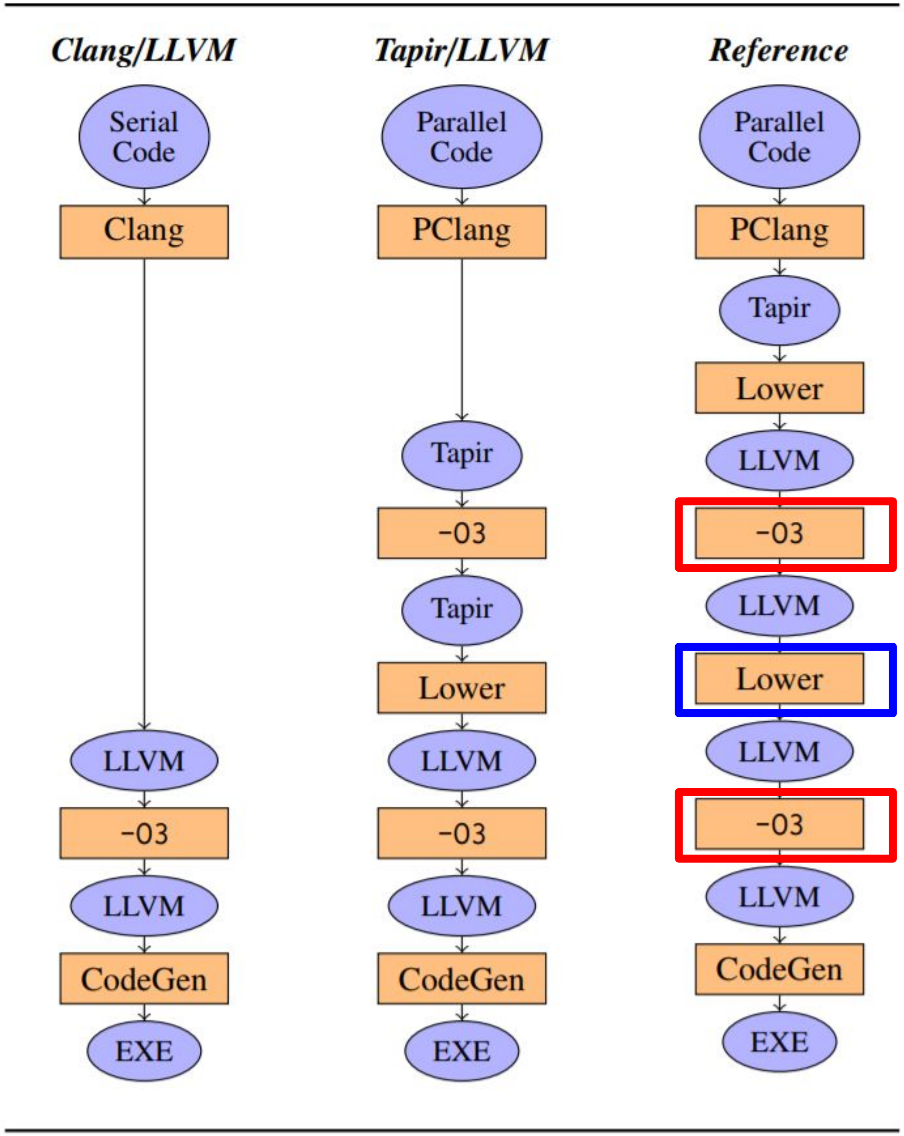
Model Pipeline

2 x O3 > 1 x O3

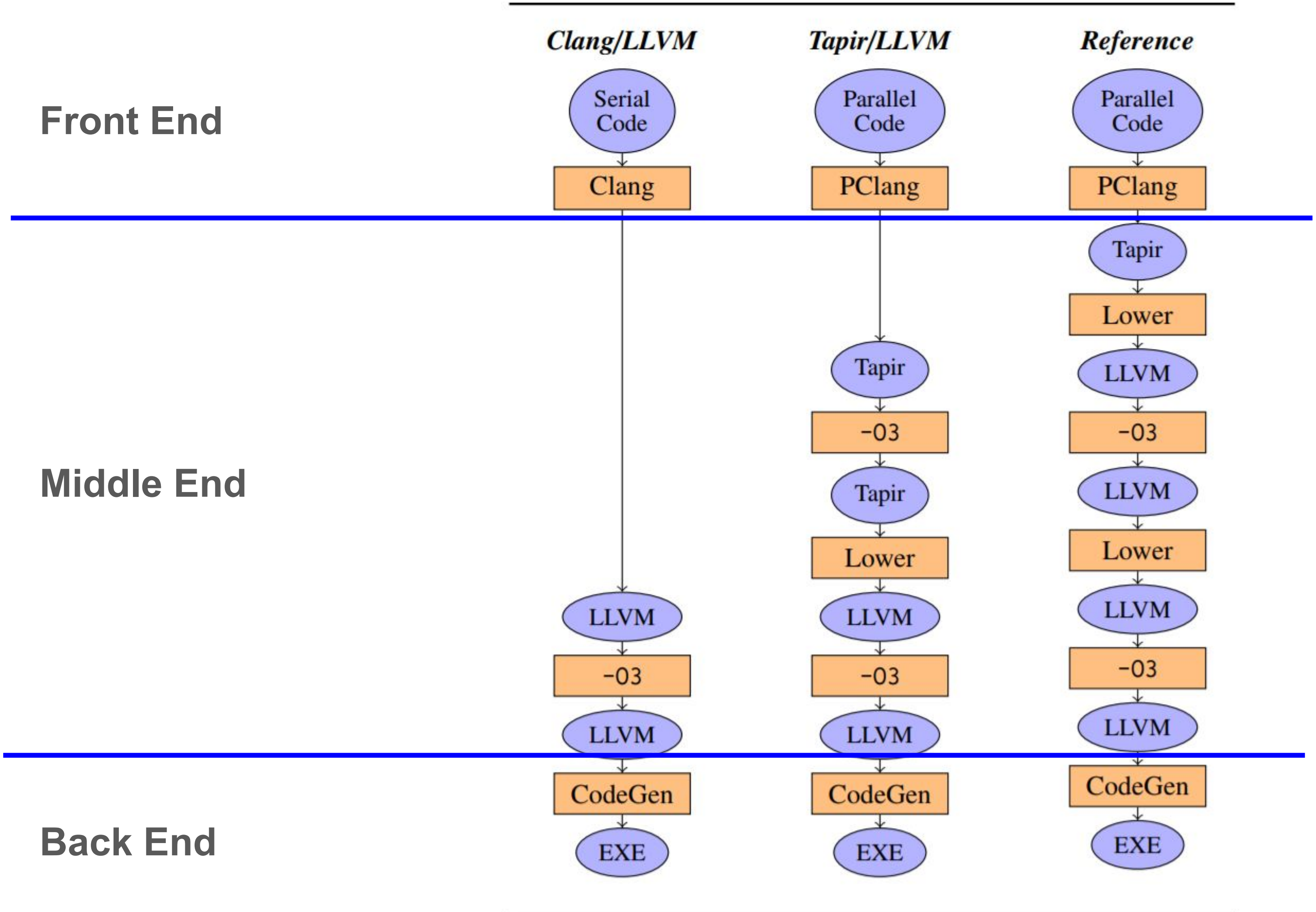
- **Counter-intuitive, but true**
- **13% faster for mtx. mult.**
- **Keep consistent**

The second Lower Trans.

- **Useless**
- **Keep consistent**



Model Pipeline



Review Pipeline

Additional optimization

- **Happened before Lower**
- **Not useful for most cases**