Tapir

Embedding Fork-Join Parallelism into LLVM's Intermediate Representation

Aaron Li

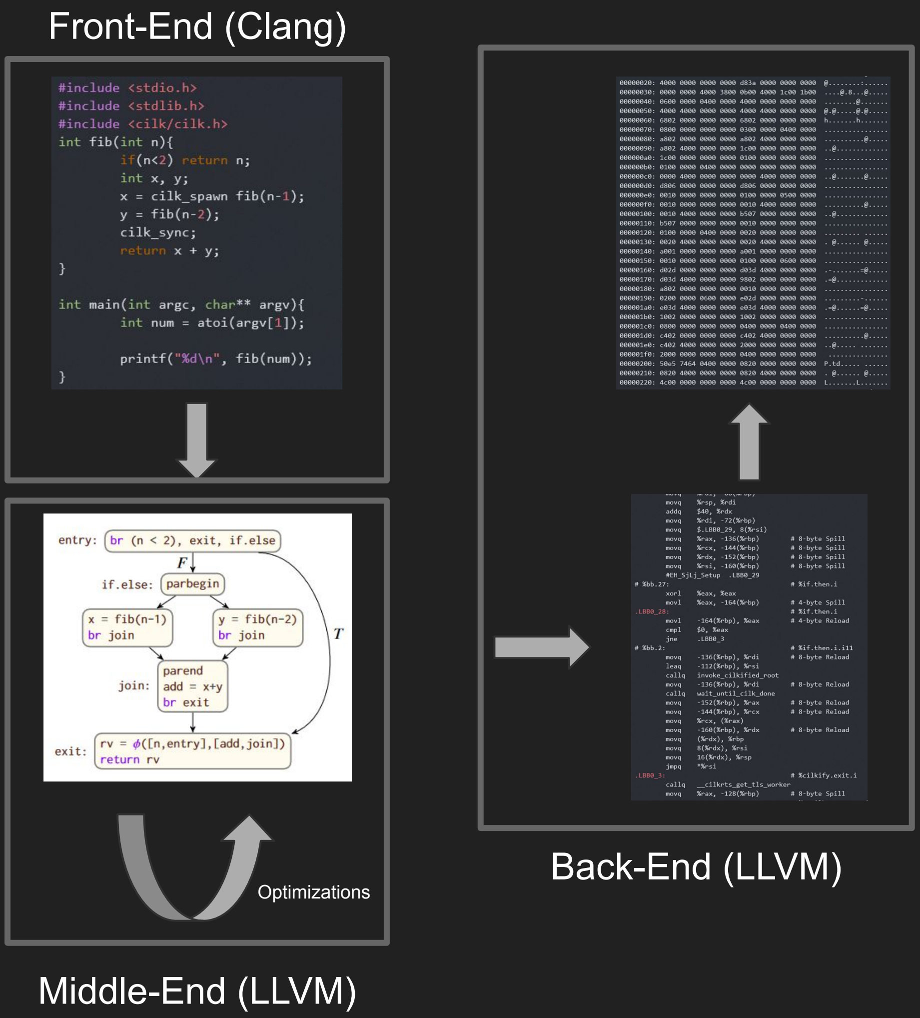
What is Tapir?

- A set of modifications to the Clang/LLVM compiler to better support parallel fork-join C code
  - Modifications to compiler front-end
  - Additions and modifications to LLVM intermediate representation (IR)
  - Additional LLVM optimizations specialized for parallel code
  - 6010 additional/modified lines of code to the ~4 million+ LOC LLVM codebase

More on Clang/LLVM

Clang/LLVM

- Open Source
- Clang front-end with LLVM middle-end and architecture specific back-end
- Clang converts C/C++ code into an LLVM Intermediate Representation and LLVM deals with optimizing the LLVM IR before finally converting to machine code



LLVM Intermediate Representation (IR)

- Unnamed Register: %<number>
- Named Register: %<name>
- Types
- Functions
- Labels
- Converting from higher level languages to LLVM IR is simplified by the IR’s ability to represent high level concepts

```
; ModuleID = 'add.c'
source_filename = "add.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: norecurse nounwind readnone uwtable
define dso_local i32 @add(i32 %x, i32 %y) local_unnamed_addr #0 {
entry:
  %add = add nsw i32 %y, %x
  ret i32 %add
}
```

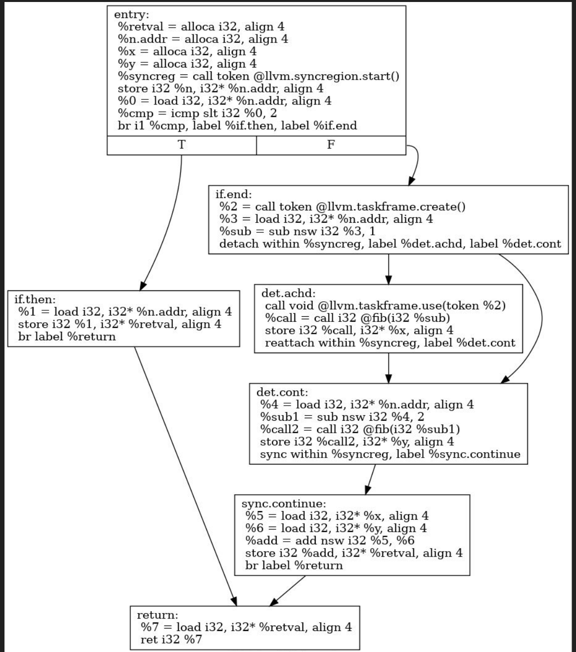
Tapir LLVM IR additions

```c
#include <stdio.h>
#include <stdlib.h>
#include <cilk/cilk.h>
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;
}

int main(int argc, char** argv){
        int num = atoi(argv[1]);

        printf("%d\n", fib(num));
}
```

- detach label b, label c
  - Terminates a block
  - Detaches b and allows it to run in parallel
  - Continues execution on current processor at label c
  - Every detach has a corresponding reattach
- reattach label c
  - Terminates a spawned block
  - Identifies the code under label c as being capable of being executed in parallel with label b
  - Destroys the spawned context
- sync
  - Blocks execution until all parallel tasks in the same context as this task reattaches

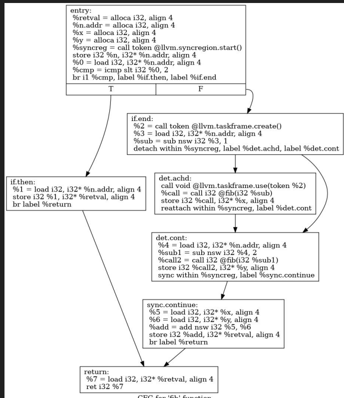


CFG for 'fib' function

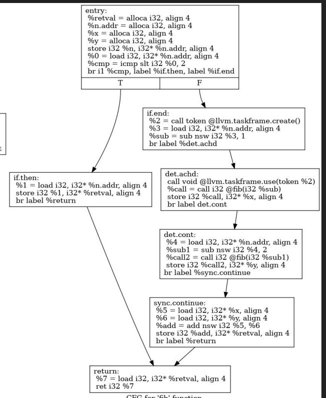
Detach: “Fork”
Reattach: “Join”

How these additions are employed

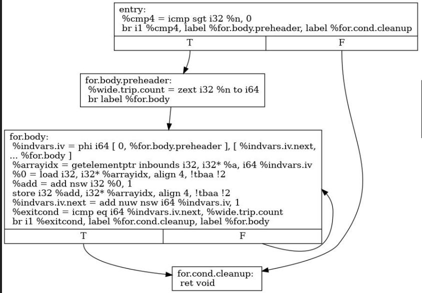
- Asymmetry (cilk_spawn)
  - Tapir IR can be converted back into serial code by replacing detach with a branch to the child function and replacing the reattach with a branch to the continuation function
- Parallel loops (cilk_for)
  - Tapir turns cilk_for loops into parallel loops that still resemble serial for loops
- Tapir IR can be interpreted as serial code



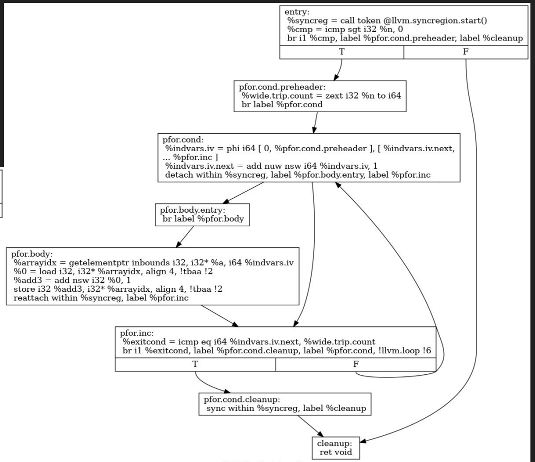
CFG for 'fib' function



CFG for 'fib' function



CFG for 'inc' function



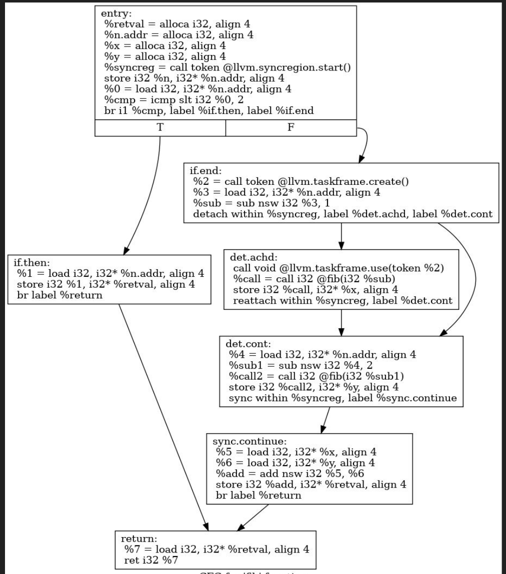
CFG for 'inc' function

How Tapir works with existing LLVM code

- Alias analysis
  - Prevents optimizations that would cause reordering of instructions that access the same memory
  - Tapir extends alias analysis to include detach and sync instructions
  - Tapir won't allow instruction reordering if a load or store instruction is being moved into a region that can be executed in parallel and the other parallel segment contains a load or store instruction that accesses the same memory location.
    - Tapir checks the latter by serializing the fork into two pseudo function calls which can then be analyzed by LLVM's alias analysis

How Tapir works with existing LLVM code

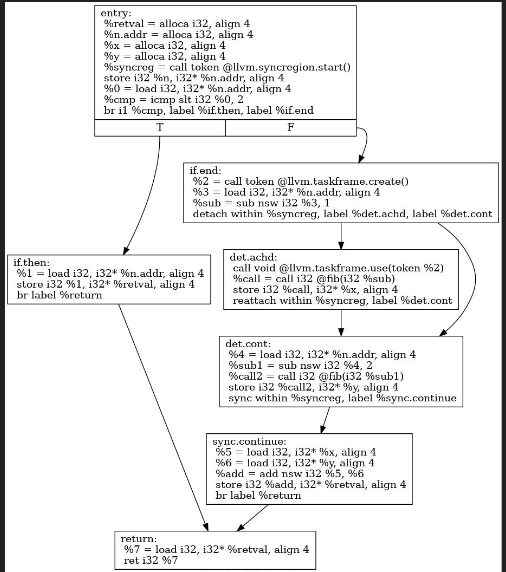
- Dominator analysis
  - Used to understand if a register value is available at a certain point in the control flow graph
  - This can be a problem for parallel code if the compiler assumes one segment will always execute before another
  - The detach/reattach nature of Tapir means a fork in Tapir code resembles a traditional if construct
  - LLVM’s Dominator analysis correctly determines behaviour with no modification!



CFG for 'fib' function

How Tapir works with existing LLVM code

- Data-Flow analysis
  - Knowing what values are present at any given point in a program
    - In a serial program it is the union of all predecessor states
  - In a parallel program, the continuation block doesn’t have access to the spawned child block’s variables
  - Tapir solves this by simply excluding the spawned child’s states from the union



CFG for 'fib' function

How Tapir works with existing LLVM code

- Common-subexpression elimination
  - LLVM built-in optimization
  - Redundant calculations are moved removed and replaced with the originally calculated value
  - Just works with Tapir code

```
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;
} }
```

```
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;
} }
```

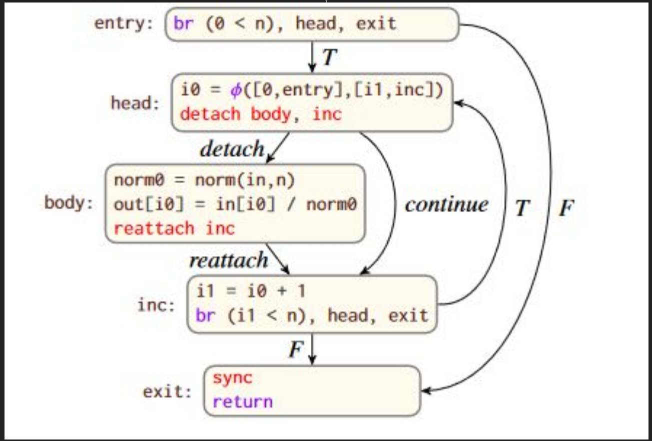
How Tapir works with existing LLVM code

- Loop-invariant code motion
  - Tapir simply analyzes the serial elision
    - Remove the continue path
    - Then, look for blocks in the loop body that dominate the exit block of the loop
  - 25 LOC change to LLVM

```
__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);
}
```

Corresponding CFG



How Tapir works with existing LLVM code

- Tail-recursion elimination
  - Replace recursive calls at the end of a function with a branch to the start of the function
  - Works like normal but remove all original sync's and place a new sync before each return of the resulting code
    - sync is only important in ensuring all spawned children are finished
  - Only 68 lines

*All the begin's are supposed to be start's

```
void pqsort(int* start, int* end) {
  if (begin == end) return;
  int* mid = partition(start, end);
  swap(end, mid);
  cilk_spawn pqsort(begin, mid);
  pqsort(mid+1, end);
  cilk_sync;
  return;
}
```

```
void pqsort(int* start, int* end) {
pqsort_start:
  if (begin == end) {
    cilk_sync;
    return;
  }
  int* mid = partition(start, end);
  swap(end, mid);
  cilk_spawn pqsort(begin, mid);
  start = mid+1;
  goto pqsort_start;
}
```

How Tapir works with existing LLVM code

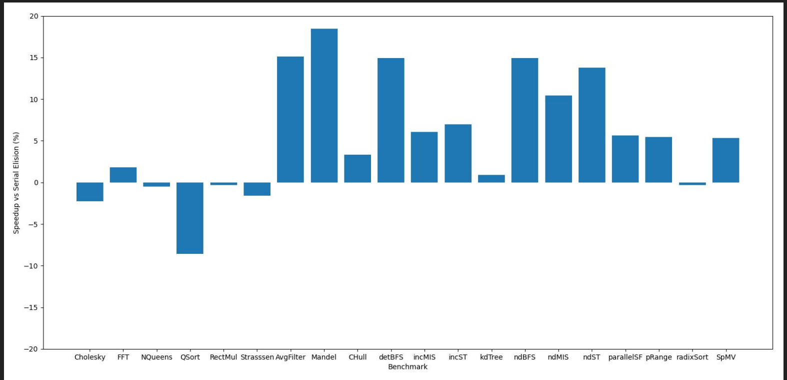
- Parallel-loop scheduling
  - For parallel loops with enough iterations, a divide and conquer strategy of spawning tasks is more efficient than the previously shown methods
- Unnecessary-synchronization elimination
  - Removes sync instructions that have nothing to wait on
- Puny-task elimination
  - Serializes child tasks if they do not contain enough work
    - Task spawn overhead likely more expensive than operation

Results

- Performance hit at worst was 8-9% in 1 of the 6 underperforming benchmarks and <=0.5% in 3 of the 6 underperforming benchmarks
- Performance uplift was at best 18-19% in 1 of the 14 improved benchmarks and >=10% in 6 of the 14 improved benchmarks
- Tapir/LLVM is the default compiler for Cilk programs today

		Cholesky	FFT	NQueens	QSort	RectMul	Strasssen	AvgFilter	Mandel	CHull	detBFS
T_S	Ref.	2.935	10.304	3.084	4.983	10.207	10.105	1.751	25.779	0.938	5.670
	Tapir	2.933	10.271	3.083	4.984	10.207	10.119	1.750	25.780	0.935	5.666
T_1	Ref.	4.572	11.919	3.409	6.581	10.413	10.196	2.355	30.520	1.316	6.596
	Tapir	4.739	11.733	3.419	6.461	10.415	10.196	1.730	25.774	1.187	5.673
T_{18}	Ref.	0.387	0.788	0.196	0.648	0.609	1.106	0.708	1.847	0.124	0.517
	Tapir	0.396	0.774	0.197	0.709	0.611	1.124	0.615	1.559	0.120	0.467
$\frac{T_S}{T_1}$	Ref.	0.642	0.862	0.904	0.757	0.980	0.991	0.743	0.845	0.710	0.801
	Tapir	0.619	0.875	0.902	0.771	0.980	0.991	1.012	1.000	0.788	0.992
$\frac{T_S}{T_{18}}$	Ref.	7.579	13.034	15.730	7.690	16.760	9.137	2.472	13.957	7.540	9.518
	Tapir	7.407	13.270	15.650	7.028	16.705	8.990	2.846	16.536	7.792	10.942
		incMIS	incST	kdTree	ndBFS	ndMIS	ndST	parallelSF	pRange	radixSort	SpMV
T_S	Ref.	4.993	4.190	5.473	3.950	9.210	4.069	5.136	2.564	3.775	1.780
	Tapir	5.006	4.173	5.466	3.956	9.253	4.053	5.136	2.559	3.775	1.783
T_1	Ref.	6.030	4.733	5.640	4.930	10.760	4.286	5.646	3.438	3.795	1.836
	Tapir	5.043	4.203	5.546	3.980	9.246	4.063	5.183	3.083	3.800	1.786
T_{18}	Ref.	0.559	0.352	0.342	0.415	0.774	1.925	0.414	0.348	0.284	0.118
	Tapir	0.527	0.329	0.339	0.361	0.701	1.692	0.392	0.330	0.285	0.112
$\frac{T_S}{T_1}$	Ref.	0.828	0.882	0.969	0.801	0.856	0.946	0.910	0.744	0.995	0.969
	Tapir	0.990	0.993	0.986	0.992	0.996	0.998	0.991	0.830	0.993	0.997
$\frac{T_S}{T_{18}}$	Ref.	8.932	11.855	15.982	9.518	11.899	2.105	12.406	7.353	13.292	15.085
	Tapir	9.474	12.684	16.124	10.942	13.138	2.395	13.102	7.755	13.246	15.893

- T□: Running time of serial elision (with 1 worker)
- T_1: Running time of parallel code with 1 worker
- T_{18}: Running time of parallel code with 18 workers (Running on an AWS EC2 c4.8xlarge instance)
  - Paper: The c4.8xlarge is a dual CPU 36 core instance
    - AWS: the c4.8xlarge is a 36 vCPU instance
  - Paper: The Intel Xeon E5-2666 v3 is an 18 core CPU
    - Various sources say the E5-2666 v3 is a 10 core CPU



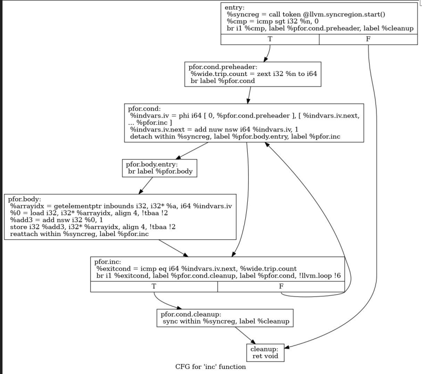
Useful Commands

- clang -S --emit-llvm <file.c>
  - Produces .ll LLVM IR from Cilk code
  - Playing around with -O0, -O1, … ,-O3 can give various levels of readability to resulting LLVM IR
- opt --dot-cfg <ir.ll>
  - Produces a .dot file for visualizing the CFG with graphviz
- dot -Tpng <dot.dot>
  - Produces a png of the given .dot file

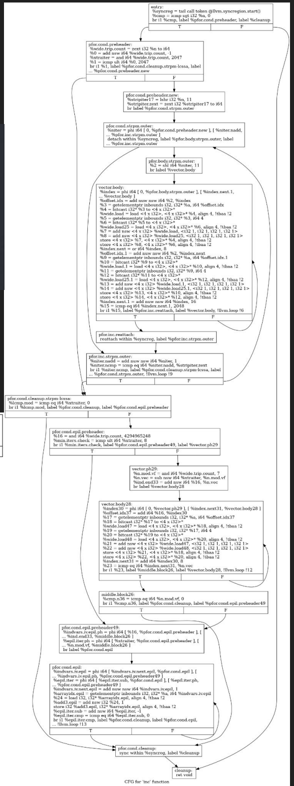
inc.c

```c
#include<cilk/cilk.h>
void inc(int *a, int n){
        cilk_for(int i = 0; i<n; i++){
                a[i] += 1;
        }
}
```

-O1



-O3



Using -O3 on inc.c resulted in LLVM producing SIMD SSE2/MMX/AVX2 code