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

Tao B. Schardl William S. Moses Charles E. Leiserson

MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street
Cambridge, MA 02139
neboat@mit.edu wmoses@mit.edu cel@mit.edu

Abstract

This paper explores how fork-join parallelism, as supported by concurrency platforms such as Cilk and OpenMP, can be embedded into a compiler's intermediate representation (IR). Mainstream compilers typically treat parallel linguistic constructs as syntactic sugar for function calls into a parallel runtime. These calls prevent the compiler from performing optimizations across parallel control constructs. Remedying this situation is generally thought to require an extensive reworking of compiler analyses and code transformations to handle parallel semantics.

Tapir is a compiler IR that represents logically parallel tasks asymmetrically in the program's control flow graph. Tapir allows the compiler to optimize across parallel control constructs with only minor changes to its existing analyses and code transformations. To prototype Tapir in the LLVM compiler, for example, we added or modified about 6000 lines of LLVM's 4-million-line codebase. Tapir enables LLVM's existing compiler optimizations for serial code — including loop-invariant-code motion, common-subexpression elimination, and tail-recursion elimination — to work with parallel control constructs such as spawning and parallel loops. Tapir also supports parallel optimizations such as loop scheduling.

Keywords Cilk; compiling; control-flow graph; fork-join parallelism; LLVM; multicore; OpenMP; optimization; parallel computing; serial semantics; Tapir.

1. Introduction

Mainstream compilers, such as GCC [68], ICC [24], and LLVM [32], now support ***fork-join parallelism*** [17, 18, 23, 40], where subroutines can be spawned in parallel and iterations of a parallel loop can execute concurrently on modern multicore machines. In particular, these compilers provide support for the Cilk Plus [21] and OpenMP [5, 54] linguistic extensions for fork-join parallelism.[1] The execution of a fork-join program generates a series-parallel dag [14] of logically parallel tasks. The execution and synchronization of parallel tasks is managed "under the covers" by a runtime system, which typically implements a randomized work-stealing scheduler [4, 8, 9, 16] to schedule and load-balance the computation among parallel worker threads. Fork-join programs provide serial semantics [16], and they admit efficient tools to detect "determinacy races" or validate their absence [14, 15, 70].

But although these mainstream compilers support fork-join parallelism, they fail to optimize parallel code as well as they optimize serial code. Consider, for example, the parallel `cilk_for` loop on lines 5–6 in Figure 1a, which indicates that iterations of the loop are free to execute in parallel. In a serial version of this loop, where the `cilk_for` keyword is replaced by an ordinary `for` keyword, each of the compilers GCC 5.3.0, ICC 16.0.3, and Cilk Plus/LLVM 3.9.0 observes that the call to `norm` on line 6 produces the same value in every iteration of the loop, and they optimize the loop by computing this value only once before the loop executes. This optimization dramatically reduces the total time to execute `normalize` from $\Theta(n^2)$ to $\Theta(n)$. Although this same optimization can, in principle, be performed on the actual parallel loop in the figure, no mainstream compiler performs this code-motion optimization. The same is true when the parallel loop is written using OpenMP, as shown in Figure 1b.

This failure to optimize stems from how these compilers for serial languages implement parallel linguistic constructs. The compiler for a serial language, such as C [28] or C++ [69], can be viewed as consisting of three phases: a front end, a middle end, and a back end. The front end parses and type-checks the input program and translates it to an ***intermediate***

PPoPP '17, February 04-08, 2017, Austin, TX, USA
 ISBN 978-1-4503-4493-7/17/02…$15.00
DOI: http://dx.doi.org/10.1145/3018743.3018758

[1] Unlike the other compilers, LLVM support for Cilk Plus is not in the main branch, however, but in a separate Cilk Plus/LLVM branch [23].

a

```
01 __attribute__((const)) double norm(const double *A, int n);
02
03 void normalize(double *restrict out,
04                const double *restrict in, int n) {
05   cilk_for (int i = 0; i < n; ++i)
06     out[i] = in[i] / norm(in, n);
07 }
```

b

```
08 __attribute__((const)) double norm(const double *A, int n);
09
10 void normalize(double *restrict out,
11                const double *restrict in, int n) {
12   #pragma omp parallel for
13   for (int i = 0; i < n; ++i)
14     out[i] = in[i] / norm(in, n);
15 }
```

Figure 1. A function that GCC, ICC, and Cilk Plus/LLVM all fail to optimize effectively. **a** A Cilk version of the code. The `cilk_for` loop on lines 5–6 allows each iteration of the loop to execute in parallel. The `norm` function computes the norm of a vector in $\Theta(n)$ time. The call to `norm` on line 6 can be safely moved outside of the loop, but none of these three mainstream compilers perform this code motion, even though they all do so when the `cilk_for` keyword is replaced with an ordinary `for` keyword. **b** The corresponding OpenMP code.

representation (IR), which represents the control flow of the program as a more-or-less language-independent ***control-flow graph (CFG)*** [2, Sec. 8.4.3]. The middle end consists of optimization passes that transform the IR into a more-efficient form. These optimizations tend to be independent of the instruction-set architecture of the target computer. The back end translates the optimized IR into machine code, performing low-level machine-dependent optimizations.

GCC, ICC, and Cilk Plus/LLVM all ***lower*** the parallel constructs — transform the parallel constructs to a more-primitive representation — in the front end. To compile the code in Figure 1a, for example, the front-end translates the parallel loop in lines 5–6 into IR in two steps. (The OpenMP code in Figure 1b is handled similarly.) First, the loop body (line 6) is lifted into a helper function. Next, the loop itself is replaced with a call to a library function implemented by the Cilk Plus runtime system, which takes as arguments the loop bounds and helper function, and handles the spawning of the loop iterations for parallel execution. Since this process occurs in the front end, it renders the parallel loop unrecognizable to middle-end loop-optimization passes, such as code motion. In short, these compilers treat parallel constructs as syntactic sugar for opaque runtime calls, which confounds the many middle-end analyses and optimizations.

Previous approaches

This paper aims to enable middle-end optimizations involving fork-join control flow by embedding parallelism directly into the compiler IR, an endeavor that has historically been challenging [36, 38]. For example, it is well documented [47] that traditional compiler transformations for serial programs can jeopardize the correctness of parallel programs. In general, four types of approaches have been proposed to embed parallelism in a mainstream compiler IR.

First, the compiler can use metadata to delineate logical parallelism. LLVM's parallel loop metadata [41], for example, is attached to memory accesses in a loop to indicate that they have no dependence on other iterations of the same loop. LLVM can only conclude that a loop is parallel if all its memory accesses are labeled with this metadata. Unfortunately, encoding parallel loops in this way is fragile, since a compiler transformation that moves code into a parallel loop risks serializing the loop from LLVM's perspective.

Second, the compiler can use intrinsic functions to demark parallel tasks. (For examples, see [37, 55, 72].) Often, either existing serial analyses and optimizations must be shut down when code contains these intrinsics, or the intrinsics offer minimal opportunities for compiler optimization.

Third, the compiler can use a separate IR to encode logical parallelism in the program. The HPIR [7, 72], SPIRE [29], and INSPIRE [27] representations, for instance, model parallel constructs using an alternative IR, such as one based on the program's abstract syntax tree [2, Sec. 2.5.1]. Such an IR can support optimizations involving parallel constructs without requiring changes to existing middle-end optimizations. But adopting a separate IR into a mainstream compiler has historically been criticized [39] as requiring considerable effort to engineer, develop, and maintain the additional IR to the same standards as the compiler's existing serial IR.

Fourth, the compiler can augment its existing IR to encode logical parallelism, which is the approach that Tapir follows. Unlike Tapir, all prior research on parallel precedence graphs [66, 67], parallel flow graphs [19, 65], concurrent control-flow graphs [34, 53], and parallel program graphs [59, 60] represent parallel tasks as symmetric entities in a CFG. For the parallel `fib` function in Figures 2a and 2b, for example, the parallel flow graph in Figure 2c illustrates how forked subcomputations might be represented symmetrically. Some of these approaches struggle to represent common parallel constructs, such as parallel loops [29, 34], while others exhibit problems when subjected to standard compiler analyses and transformations for serial programs [19, 31, 34, 58, 59, 66, 67]. Existing serial-program analyses in LLVM, for example, assume that a basic block with multiple predecessors can observe the variables of only one predecessor at runtime. For the parallel flow graph in Figure 2c, however, instructions in the `join` block must observe the values of `x` and `y` from both of its predecessors, as has been observed by [34]. Parallel loops exacerbate this problem by allowing a dynamic number of tasks to join at the same basic block. Previous research [1, 58] has proposed solutions to these problems, including additional representations of the program and augmented analyses that account for interleavings of parallel instructions, but adopt-

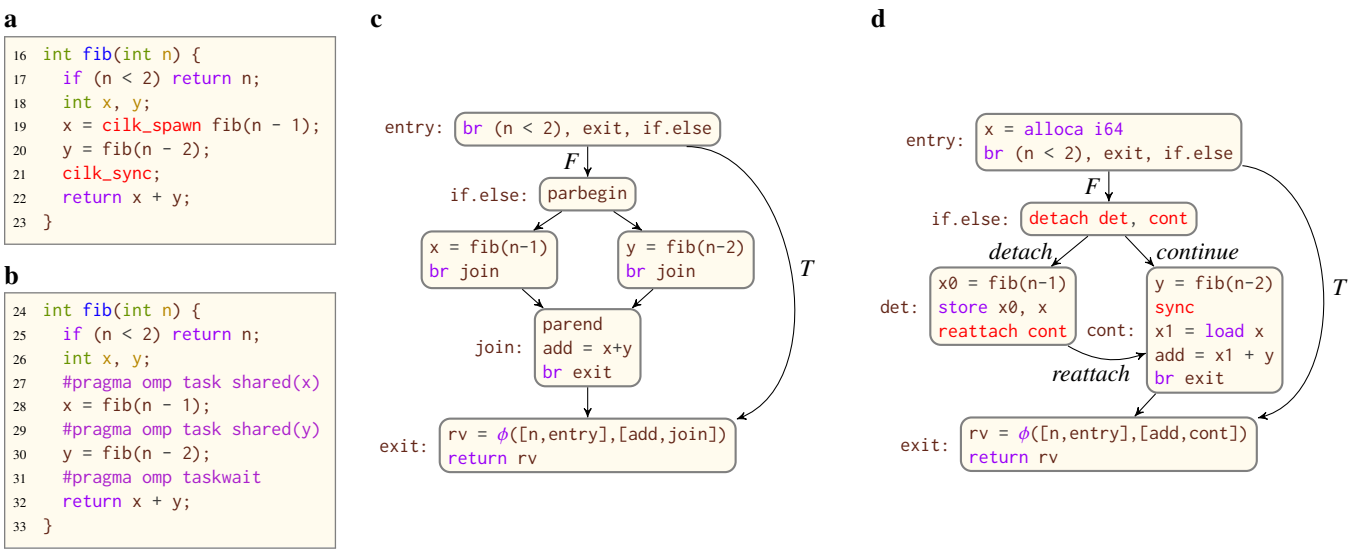

Figure 2. Comparison between a traditional CFG with symmetric parallelism and Tapir's CFG with asymmetric parallelism. **a** The Cilk function `fib` computes Fibonacci numbers. The `cilk_spawn` on line 19 allows the two recursive calls to `fib` to execute in parallel, and the `cilk_sync` on line 21 waits for the spawned call to return. A serial execution of `fib` executes `fib(n-1)` before `fib(n-2)`. **b** A comparable implementation of `fib` using OpenMP task parallelism. **c** A CFG for `fib` that encodes parallelism symmetrically. Rectangles denote basic blocks, which contain C-like pseudocode for `fib`. Edges denote control flow between basic blocks. The `parbegin` and `parend` statements create and synchronize the parallel calls to `fib`. The `br` instruction encodes either an unconditional or a conditional branch. True and false edges from a conditional branch are labeled *T* and *F*, respectively. The ϕ instruction, used to support a static-single-assignment (SSA) form of the program (see Section 2), takes as its arguments pairs that associate a value with each predecessor basic block of the current block. At runtime the ϕ instruction returns the value associated with the predecessor basic block that executed immediately before the current block. **d** The Tapir CFG for `fib`, which encodes parallelism asymmetrically. The `alloca` instruction allocates shared-memory storage on the call stack for a local variable. Section 2 defines the `detach`, `reattach`, and `sync` instructions and the *detach*, *reattach*, and *continue* edge types.

ing these techniques into a mainstream compiler seems to require extensive changes to the existing codebase.

The Tapir approach

This paper introduces Tapir, a compiler IR that represents logical fork-join parallelism asymmetrically in the program's CFG. The asymmetry corresponds to the assumption of ***serial semantics*** [16], which means it is always semantically correct to execute parallel tasks in the same order as an ordinary serial execution.

Tapir adds three instructions — `detach`, `reattach`, and `sync` — to the IR of an ordinary serial compiler to express fork-join parallel programs with serial semantics. Figure 2d illustrates the Tapir CFG for the `fib` function. As with the symmetric parallel flow graph in Figure 2c, Tapir places the logically parallel recursive calls to `fib` in separate basic blocks. But these blocks do not join at a synchronization point symmetrically. Instead, one block connects to the other, reflecting the serial execution order of the program.

The Tapir approach provides five advantages:

1. Introducing fork-join parallelism into the compiler is relatively easy.
2. The IR is expressive and can represent fork-join control constructs from different parallel-language extensions.
3. Tapir parallel constructs harmonize with the invariants associated with existing representations of serial code.
4. Standard serial optimizations work on parallel code with few modifications.
5. The optimizations enabled by Tapir's parallelism constructs are effective in practice.

We discuss each of these advantages in turn.

Ease of implementation

Tapir's asymmetric representation of logically parallel tasks makes it relatively simple to integrate Tapir into an existing compiler's intermediate representation such as LLVM IR [41]. Figure 3 documents the lines of code added, modified, or deleted to implement a prototype of Tapir in LLVM. As Figure 3 shows, Tapir/LLVM was implemented with about 6000 lines, compared to LLVM's roughly 4-million-line codebase. Moreover, fewer than 2000 lines of code were needed to adapt LLVM's existing compiler analyses and transformations to accommodate Tapir.

The breakdown of lines is as follows. The lines for "Instructions" add Tapir's instructions to LLVM IR and adapt LLVM's routines for reading and writing LLVM IR and bitcode files. Conceptually, these changes allow LLVM to correctly compile a Tapir program to a serial executable with no optimizations. The lines for "Memory Behavior" control how Tapir instructions may interact with memory operations, preventing the compiler from creating any races. The lines for "Optimizations" perform any adjustments required

Compiler Component	*LLVM 4.0svn*	*Tapir/LLVM*	
Instructions	105,995	943	1,768
Memory Behavior	21,788	445	
Optimizations	152,229	380	
Parallelism Lowering	0	3,782	
Other	3,803,831	460	
Total	4,083,843	6,010	

Figure 3. Breakdown of the lines of code added, modified, or deleted in LLVM to implement the Tapir/LLVM prototype.

for LLVM analyses and transformations to compile a Tapir program at optimization level `-O3`. Most of these modifications are not necessary for creating a correct executable but are added to allow the compiler to perform additional optimizations, such as parallel tail-recursion elimination (described in Section 4). The lines for "Parallelism Lowering" translate Tapir instructions into Cilk Plus runtime calls and allow the code to be race-detected with a provably good race detector [14]. The lines for "Other" address a bug in LLVM's implementation of `setjmp` and implement useful features for our development environment.

Expressiveness of Tapir

Tapir can express logical fork-join parallelism in parallel programs that have serial semantics. For example, Figure 2 illustrates how Tapir can express the parallelism encoded by the `cilk_spawn` and `cilk_sync` linguistics from Cilk++ [35] and Cilk Plus [21], as well as the parallelism encoded by OpenMP `task` and `taskwait` clauses [5]. Similarly, Tapir can express the parallelism encoded by OpenMP parallel sections [54] and Habanero's `async` and `finish` constructs [11]. Tapir can also express parallel loops, including `cilk_for` loops and OpenMP parallel loops that have serial semantics (described in Section 2). Other parallel constructs can be represented as well, although parallel operations that cannot be expressed in terms of fork-join parallelism, such as OpenMP's `ordered` clause, cannot be represented directly using Tapir's `detach`, `reattach`, and `sync` instructions.

Tapir makes minimal assumptions about the consistency [10, 56] of concurrent memory accesses. Tapir assumes that memory is shared among parallel tasks and that virtual-register state is local to each task. Parallel instructions in Tapir can exhibit a ***determinacy race***[2] [14] if they access the same memory location concurrently and at least one instruction writes to that location. Tapir itself does not fully define the possible outcomes of a determinacy race, and instead defers to existing compiler mechanisms, such as LLVM's atomic memory-ordering constraints [41], to define whichever memory model they choose. For any targeted runtime system, Tapir relies on a correct implementation of lowering in order to implement the necessary synchronization, but Tapir is oblivious to how that runtime system implements the synchronization.

[2] Determinacy races are also called general races [50] and are distinct from data races, which involve nonatomic accesses to critical regions.

Serial semantics

By grounding its model of parallelism in serial semantics, Tapir enables common compiler optimizations for serial code to work on parallel code. Intuitively, because Tapir always allows parallel tasks to execute in their ordinary serial execution order, the compiler can to optimize parallel code in any manner that preserves the serial semantics of the program and does not introduce new determinacy races. These mild constraints support common optimizations on parallel code, such as sequentialization, which can be invalid under models of parallelism without serial semantics [71].

Optimizations

In practice, we have found that Tapir enables a wide variety of standard compiler optimizations to work with parallel code. The prototype implementation of Tapir/LLVM, for example, successfully moves the call to `norm` in Figure 1 outside of the loop, just as it would for a serial `for` loop. As Section 4 discusses, Tapir enables other optimizations, including common-subexpression elimination [48, Sec. 12.2], loop-invariant-code motion [48, Sec. 13.2], and tail-recursion elimination [48, Sec. 15.1], to work on parallel code. Tapir also enables new optimizations on parallel control flow.

Evaluation of Tapir/LLVM

The compiler optimizations that Tapir enables are effective in practice. We evaluated the Tapir approach by measuring the performance of 20 Cilk application benchmarks compiled using Tapir/LLVM. We compared the performance of these executables to those produced by a comparable reference compiler, called Reference. Conceptually, Reference lowers parallel linguistic constructs directly into runtime calls, as mainstream compilers do today, but otherwise performs the same set of optimization passes as Tapir/LLVM. Section 6 describes our experimental setup in detail, including the design of Reference.

Figure 4 presents the results of comparing Tapir/LLVM and Reference in terms of the "work efficiency" of the compiled benchmarks. To perform this comparison, we compiled each benchmark using each compiler and then ran the executable on a single processing core of a multicore machine to measure its ***work***, the 1-core running time, denoted T_1. We also used each compiler to compile, run, and measure the 1-core running time of the ***serial elision*** [16] of each benchmark, denoted T_S, in which the benchmark is converted into a corresponding serial program by replacing all parallel linguistic constructs with their serial equivalents. We then computed the ***work efficiency*** of each compiled benchmark, which is the ratio T_S/T_1 of the running time T_S of the benchmark's serial elision divided by the work T_1 of the benchmark. In theory, the maximum possible work efficiency is $T_S/T_1 = 1$, but in practice, quirky behaviors of the compiler and multicore architecture can occasionally produce work efficiencies greater than 1. As Figure 4 shows, for most benchmarks, the executables compiled using Tapir/LLVM achieve equal or higher work efficiency than those compiled

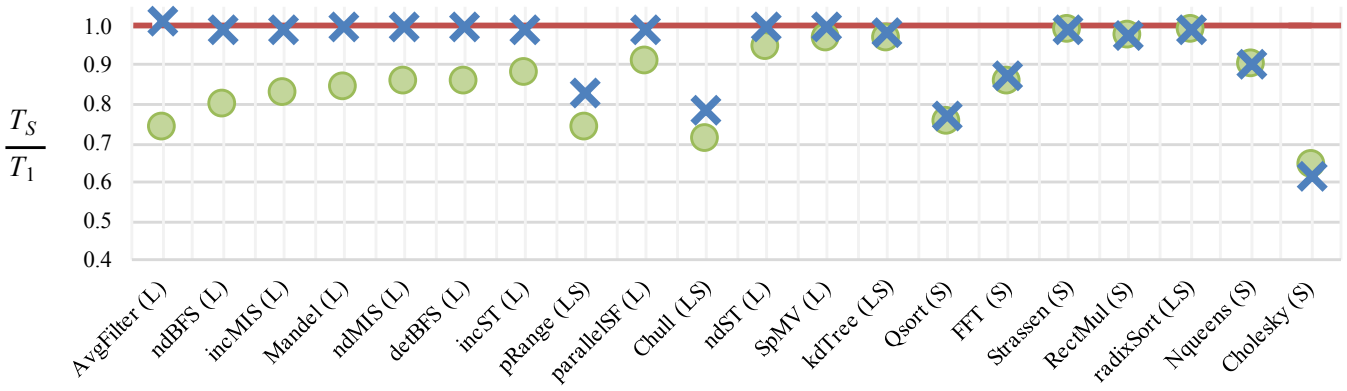

Figure 4. Comparison of the work efficiency of 20 parallel application benchmarks compiled using Tapir/LLVM (X's) and the comparable Reference compiler (O's), described in Section 6, which lowers parallelism in the compiler front end. Each point plots the work efficiency T_S/T_1 of a compiled benchmark, where T_1 is the work of the benchmark and T_s is the running time of the serial elision of the benchmark. Higher values indicate better work efficiency. The horizontal line at 1.0 plots the theoretically maximum work efficiency $T_S/T_1 = 1$. Benchmarks are sorted by decreasing difference in work efficiency between Tapir/LLVM- and Reference-compiled executables. Benchmarks marked with an "L" use parallel loops, and benchmarks marked with an "S" use `cilk_spawn`.

using Reference. Moreover, for many benchmarks, and particularly those implemented using parallel loops, Tapir/LLVM produces executables that achieve nearly optimal work efficiency. Section 6 elaborates on these experiments.

Contributions

This paper makes the following research contributions:

- The design of a compiler IR that represents fork-join parallelism asymmetrically, which enables existing serial optimizations to operate on parallel code and which also enables parallel optimizations.
- The implementation of Tapir/LLVM in the LLVM compiler by modifying about 6000 source lines of code (0.15% of the 4-million-line LLVM codebase).
- The implementation of parallel optimizations such as unnecessary synchronization elimination and parallel-loop scheduling.
- Experiments that demonstrate the advantage of embedding fork-join parallelism into a compiler's IR, as opposed to dealing with parallelism only in the compiler's front end.

Outline

The remainder of this paper is organized as follows. Section 2 describes Tapir's representation and properties. Section 3 discusses how analysis passes can be adapted to operate on Tapir programs. Section 4 describes various optimizations on parallel control flow that Tapir enables. Section 5 describes auxiliary software we developed to exercise and test Tapir/LLVM. Section 6 discusses our evaluation of the effectiveness of Tapir. Section 7 discusses related work. Section 8 provides some concluding remarks. An appendix describes how to set up Tapir/LLVM and how to download and run our suite of application benchmarks.

2. Tapir

This section describes how Tapir represents logically parallel tasks asymmetrically in the CFG of a program. We define Tapir's three new instructions and how they interact with LLVM's static single-assignment (SSA) form [2, Sec. 6.2.4]. Although we describe Tapir as an extension to LLVM IR [41], we see no reason why other compilers cannot gain similar advantages from Tapir-like instructions.

Like LLVM IR, Tapir treats a program function as a CFG $G = (V, E, v_0)$, where

- the set V of vertices represents the function's ***basic blocks***: sequences of LLVM instructions, where control flow can only enter through the first instruction and leave from the last instruction;
- the set E of edges denote control flow between (basic) blocks; and
- the designated vertex $v_0 \in V$ represents the ***entry point*** of the function.

Tapir instructions

Tapir extends LLVM IR with three instructions: `detach`, `reattach`, and `sync`. The `detach` and `reattach` instructions together delineate logically parallel tasks, and the `sync` instruction imposes synchronization on parallel tasks. The three instructions have the following syntax, where $b, c \in V$:

```
detach label b, label c
reattach label c
sync
```

The `label` keywords indicate that b and c are (labels of) basic blocks in V.

The `detach` and `reattach` instructions together delineate a parallel task as follows. A `detach` instruction terminates

the block a that contains it and takes a ***detached*** block b and a ***continuation*** block c as its arguments. The `detach` instruction ***spawns*** the task starting at block b, allowing that task to execute in parallel with block c. The control-flow edge $(a, b) \in E$ is a ***detach*** edge, and the edge $(a, c) \in E$ is a ***continue*** edge. A `reattach` instruction, meanwhile, terminates the block a' that contains it and takes a single ***continuation*** block c as its argument, inducing a ***reattach*** edge $(a', c) \in E$ in the CFG. The `reattach` terminates the task spawned by a preceding `detach` instruction with the same continuation block. Together, a `detach` instruction and associated `reattach` instructions demark the start and end of a parallel task and indicate that that task can execute in parallel with their common continuation block.

For the example in Figure 2d, the `detach` in the `if.else` block and the `reattach` in the `det` block share the same continuation block `cont`. Together, this `detach` and this `reattach` indicate that the `det` block is a parallel task which can execute in parallel with the `cont` block. In general, a parallel task delineated by `detach` and `reattach` can consist of many basic blocks in a single-entry subgraph.

The `detach` and `reattach` instructions in a CFG obey several structural properties. We say a `reattach` instruction j ***reattaches*** a `detach` instruction i if i and j share a common continuation block and there is a path from the detached block of i to j. Tapir assumes that every CFG $G = (V, E, v_0)$ obeys the following invariants on every `detach` instruction i and `reattach` instruction j in G:

1. A `reattach` instruction reattaches exactly one `detach` instruction.
2. If j reattaches i, then every path from v_0 to the block terminated by j passes through the detach edge of i, that is, the detach edge of i ***dominates*** j.
3. Every path starting from the detached block of i must reach a block terminated by a `reattach` instruction that reattaches i.
4. If j reattaches i and a path from i to j passes through the detach edge of another `detach` instruction i', then it must also pass through a `reattach` instruction j' that reattaches i'.
5. Every cycle containing a `detach` instruction i must pass through a `reattach` instruction that reattaches i.
6. The continuation block of j cannot contain any ϕ instructions [2, Sec 6.2.4].

These invariants imply that, at runtime, a `detach` instruction i with detached block b and continuation block c spawns the execution of a ***detached sub-CFG***, which is the single entry sub-CFG starting at b induced by all blocks on paths from b to a `reattach` instruction that reattaches i.

The dynamic execution of the program organizes memory as a tree of ***parallel contexts***. A new parallel context is created as a child of the current context when control enters a function or follows a detach edge. When control executes a `reattach` instruction or leaves a function, the context is destroyed and the parent's context becomes the current context. An `alloca` instruction allocates shared memory in the current context.

The `sync` instruction synchronizes tasks spawned within its parallel context. At runtime, a `sync` instruction dynamically waits for the set of sub-CFG's detached in the same parallel context or any of its descendant parallel contexts to reach a `reattach` instruction. In the Tapir CFG illustrated in Figure 2d, for example, the `sync` instruction in the `cont` block simply waits for the execution of the `det` block to complete. Unlike `reattach` instructions, `sync` instructions are not explicitly associated with `detach` instructions, and they, in fact, can be executed within conditionals. A `sync` instruction j ***syncs*** a `detach` instruction i if i and j belong to the same parallel context and the CFG detached by i cannot be guaranteed to have completed when j executes.

Static single-assignment form

LLVM's static single-assignment (SSA) form [2, Sec. 6.2.4] must be adapted for Tapir programs. SSA form ensures that each virtual register is set at most once in a function. LLVM IR employs the ϕ ***instruction*** [2, Sec 6.2.4] to combine definitions of a variable from different predecessors of a basic block. In adapting SSA to Tapir, one concern is that a ϕ instruction might allow registers defined in the detached sub-CFG to be used in the continuation. A basic block containing a ϕ instruction must avoid inheriting register definitions from predecessors that are connected by reattach edges. Otherwise, a register in the detached sub-CFG might not have been computed by the time the continuation executes.

We implemented this constraint by simply forbidding reattach edges from going into basic blocks with ϕ instructions. But what if the continuation c of a `detach` instruction begins with a ϕ instruction? In this case, we create a new basic block c' containing only a branch instruction to c. We reroute the reattach and continuation edges originally going to c so that they go instead to c'. All other edges going to c are left in place.

The reason this solution works is as follows. No reattach edges in the resulting CFG go to blocks containing ϕ instructions. Because a detached sub-CFG does not dominate any outside block, registers in the detached CFG can only be used in ϕ instructions of the immediate successors of the detached sub-CFG. Since the continuation is the only immediate successor of the detached sub-CFG and it contains no ϕ instructions, no registers from the detached sub-CFG may be accessed in the continuation.

Asymmetry in Tapir

The `detach` and `reattach` instructions express parallel tasks asymmetrically both syntactically in the structure of the CFG and semantically in the way memory state is managed. Both asymmetries are illustrated in Figure 2d.

First, the CFG detached by a `detach` instruction is connected by a reattach edge to the continuation block of that instruction, even though they can execute in parallel. For ex-

ample, the reattach edge between det and cont in Figure 2d breaks the symmetry between them. Reattach edges reflect the serial semantics of a Tapir program, which dictates that a serial execution of the program executes the detached CFG to completion before starting to execute the continuation block. In fact, the parallel task delineated by a detach and a reattach instruction can be ***serialized*** by replacing the detach instruction with an unconditional branch to its detached block and replacing the reattach with an unconditional branch to its continuation block. In contrast, parallel flow graphs and similar previously explored representations join logically parallel tasks in the CFG at a synchronization point. By supporting separate reattach and sync instructions, Tapir decouples the termination of a parallel task from its synchronization.

Second, although memory state is shared among all parallel tasks in Tapir, a virtual register defined in a detached sub-CFG is not accessible in its parent parallel context. For example, the continuation block cont in Figure 2d cannot assume that the register value x0 returned by fib(n-1) in block det is accessible, because the two basic blocks belong to different parallel contexts. Thus, cont must load it again after the sync instruction.

Parallel loops in Tapir

Figure 5 illustrates Tapir's default representation of the parallel loops from Figure 1. As Figure 5 shows, Tapir can represent a parallel loop in the CFG as an ordinary loop, where the head block repeatedly spawns the body block, and the exit block syncs the detached CFG's. Section 4 describes how this representation of parallel loops allows existing compiler loop optimizations to operate on Tapir parallel loops with only minor modifications. Although this loop structure can exhibit poor parallel performance when the loop body is small, separate optimization passes in Tapir/LLVM (see Section 4) transform this parallel-loop representation into a divide-and-conquer form that exhibits good performance.

3. Analysis passes

This section describes how LLVM's analysis passes can be adapted to operate on Tapir programs. We first discuss constraints on how Tapir programs can be safely transformed. Implementing these contraints on LLVM optimization passes primarily involves adapting standard compiler analyses — specifically alias analysis [2, Ch. 12], dominator analysis [2, Ch. 9], and data-flow analysis [2, Ch. 9] — to accomodate Tapir's instructions. We describe how each of these analyses was minimally modified to support Tapir.

Constraints on transformations

To be correct, a code transformation on a Tapir program must preserve the program's serial semantics, and it must not introduce any new behaviors into the program's set of behaviors. A program can exhibit more than one behavior if it contains a determinacy race. In general, the result of a determinacy race can vary nondeterministically from run to run depending on the order in which the participating instructions access the memory location. To avoid introducing new behaviors, code transformations must not create determinacy races, although they can eliminate determinacy races. Many existing serial optimizations can be adapted to respect these properties by adapting the standard compiler analyses they rely on. We now describe how LLVM's alias, dominator, and data-flow analyses were adapted for Tapir.

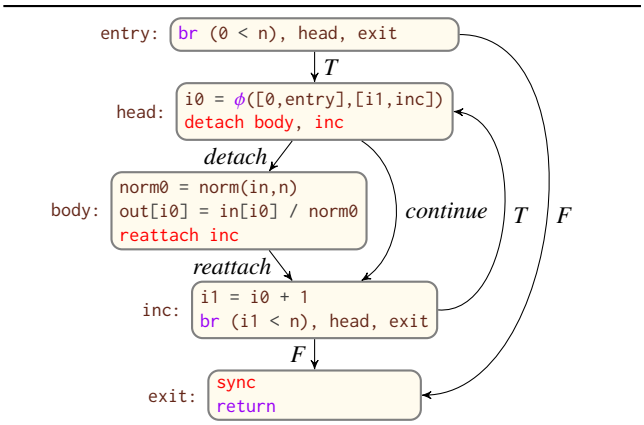

Figure 5. Tapir CFG for the parallel loops in Figure 1, using a format similar to the CFG's in Figure 2.

Alias analysis

LLVM uses alias analysis [2, Ch. 12] to determine whether different instructions might reference the same locations in memory, and in particular, to restrict the reordering of instructions that access the same memory. Tapir/LLVM modifies LLVM's alias analysis to prevent optimizations that move code around from introducing determinacy races. In particular, Tapir adapts LLVM's alias analysis to treat the instructions as if they access memory. For example, consider an instruction k that performs a load or a store. There are four cases to consider when moving k around either a detach instruction i or a sync instruction j:

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

Neither Case 2 nor Case 3 can introduce a determinacy race, because both motions serialize the execution of k with respect to the sub-CFG detached by i. Cases 1 and 4 might introduce a determinacy race, however, if k loads or stores a memory location that is also accessed by the CFG detached by i. To handle Case 1, i is treated as if it were a function call that accesses all memory locations accessed in the CFG detached by i. Similarly, for Case 4, j is treated as if it were a function call that accesses all memory locations accessed by all instructions that j might sync. A reattach instruction is treated as a compiler fence that prevents instructions from

moving across it. With these modifications, existing rules in LLVM that restrict reordering of loads and stores properly restrict memory reordering around Tapir's instructions.

Dominator analysis

Optimization passes determine what values are available to an instruction in part by using dominator analysis [2, Ch. 9], which deduces the dominance relation between all basic blocks and edges in a CFG. To handle Tapir programs correctly, optimization passes must not mistakenly cause instructions to use virtual registers that are defined in logically parallel tasks. If instruction i dominates instruction j, than an optimization pass might assume that the value produced by i is always available when j executes.

The asymmetry of Tapir's representation allows LLVM's dominator analysis to analyze Tapir programs correctly without any changes. Ignoring the names of edges, the difference between the CFG $G = (V, E, v_0)$ of a Tapir program and the CFG $G' = (V, E', v_0)$ of its serial elision is the set $E - E'$ of continue edges, each of which connects a `detach` instruction to its continuation. A continue edge short-cuts a detached sub-CFG, changing the continuation's immediate dominator from the detached sub-CFG to the block containing the `detach` instruction itself. This configuration of detach, reattach, and continue edges looks much like an ordinary `if` construct in which the detached sub-CFG is conditionally executed. As a result, dominator analysis never concludes that an instruction in a detached sub-CFG can execute before the corresponding continuation block.

Data-flow analysis

A wide class of code transformations, including those that might move instructions across a reattach edge, rely on data-flow analysis [2, Ch. 9] to examine the propagation of values along different paths through a CFG $G = (V, E, v_0)$. Fundamental to data-flow analysis is an understanding of the set of possible program states at the beginning and end of each basic block $b \in V$, denoted $\text{IN}(b)$ and $\text{OUT}(b)$, respectively.

To illustrate how LLVM's data-flow analyses were adapted to Tapir, let us examine the particular case of forward data-flow analysis. (Backward data-flow analysis is similar.) In an ordinary serial CFG, forward data-flow analysis evaluates $\text{IN}(b)$ as the union of $\text{OUT}(a)$ for each predecessor block a of b:

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

To handle Tapir CFG's, data-flow analyses must be adapted specifically to handle reattach edges. Because Tapir's asymmetric representation propagates virtual registers and memory state differently across a reattach edge, the modifications to LLVM data analyses consider registers and memory separately.

For variables stored in shared memory, the standard data-flow equations remain unchanged. Thus, LLVM need not be modified to handle them for Tapir.

a

```
34 void search(int low, int high) {
35   if (low == high) search_base(low);
36   else {
37     cilk_spawn search(low, (low+high)/2);
38     search((low+high)/2 + 1, high);
39     cilk_sync;
40 } }
```

b

```
41 void search(int low, int high) {
42   if (low == high) search_base(low);
43   else {
44     int mid = (low+high)/2;
45     cilk_spawn search(low, mid);
46     search(mid + 1, high);
47     cilk_sync;
48 } }
```

Figure 6. Example of common-subexpression elimination on a Cilk program. **a** The function `search`, which uses parallel divide-and-conquer to apply the function `search_base` to every integer in the closed interval [`low`, `high`]. **b** An optimized version of `search`, where the common subexpression `(low+high)/2` in lines 37 and 38 of the original version is computed only once and stored in the variable `mid` in line 44 of the optimized version.

For register variables, however, LLVM's data-flow analyses must be modified to exclude the values in registers from an immediate predecessor a of a basic block b if the edge $(a, b) \in E$ is a reattach edge. Denote the set of reattach edges in E by E_R. For a Tapir CFG, forward data-flow analyses define $\text{IN}(b)$ for register variables as

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

that is, they ignore predecessors across a reattach edge. With this change, Tapir/LLVM correctly propagates register variables through the CFG, never allowing register values in a basic block to use register values set in a logically parallel detached sub-CFG.

4. Optimization passes

Tapir enables LLVM's existing optimization passes [42] to work across parallel control flow. It also enables new optimization passes that specifically target Tapir's fork-join parallel constructs. This section discusses four representative optimizations. Common-subexpression elimination [48, Sec. 12.2] illustrates an optimization pass that "just works" with the additional Tapir instructions. Loop-invariant code motion [48, Sec. 13.2], and tail-recursion elimination [48, Sec. 15.1] were the only two out of LLVM's roughly 80 optimization passes that required any modification to work effectively on parallel code. Parallel-loop scheduling serves as an example of a new optimization pass.

Common-subexpression elimination

The common-subexpression elimination (CSE) optimization identifies redundant calculations and transforms the code so

that they are only computed once. For example, the expression `(low+high)/2` in Figure 6a is computed in both line 37 and line 38. Tapir/LLVM performs CSE on this code, producing code equivalent to that in Figure 6b. Existing mainstream compilers that support fork-join parallelism do not eliminate this common subexpression, however, and they compute `(low+high)/2` twice. Tapir/LLVM can perform CSE across either a continue edge, as in the example, or a detach edge. Like the vast majority of optimization passes in Tapir/LLVM, CSE "just works" on Tapir code without any modifications to LLVM's CSE pass.

Loop-invariant code motion

The loop-invariant code motion (LICM) optimization [48, Sec. 13.2] aims to move computations out of loop bodies if they compute the same value on every iteration of the loop. LICM is responsible, for example, for moving the call to `norm` in the parallel loop in Figure 1a outside of the loop, as described in Section 1. By adapting LICM to handle parallel loops, Tapir/LLVM reduces the asymptotic serial running time of this parallel loop from $\Theta(n^2)$ to $\Theta(n)$.

Tapir/LLVM requires a minor change to LLVM's LICM pass to handle parallel loops. Consider the CFG illustrated in Figure 5, which models the parallel loops in Figure 1. For the serial elision of the loop, which would have a similar graph structure except with the continue edge missing, LLVM attempts to find candidate computations to move outside the loop by looking for instructions in the basic blocks of the loop body that dominate the exit block of the loop, such as the block `inc` in Figure 5. (The block labeled `exit` is the exit of the function, not the loop exit.) For a parallel loop, however, this analysis fails to identify any code to move due to the existence of the continue edge. As Figure 5 shows, with the continue edge, blocks in the loop body can never dominate the exit block `inc` as they could for the serial elision.

Tapir/LLVM modifies LLVM's LICM pass to handle a parallel loop by analyzing the serial elision of the loop, which essentially means ignoring continue edges. For simple parallel loop structures with a single continue edge, such as that shown in Figure 5, this modification is implemented by finding blocks in the loop body that dominate the predecessors of the loop exit. The modification required changing only 25 lines of LLVM's LICM pass.

Tail-recursion elimination

Tail-recursion elimination (TRE) [48, Sec. 15.1] aims to replace a recursive call at the end of a function with a branch to the start of the function. By eliminating these recursive tail calls, TRE can avoid function-call overheads and reduce the stack space they consume. This optimization can especially benefit fork-join parallel programs, as many parallel runtime systems impose additional setup and cleanup overhead on a spawned function.

LLVM's existing TRE pass can perform the TRE optimization on Tapir programs with just a minor modification. Specifically, the modified TRE pass ignores `sync` instructions after the tail-recursive call. Further, if TRE is applied and ignores a `sync` instruction, it must then insert a `sync` instruction before any remaining returns. This modification to LLVM's TRE pass required changing only 68 lines.

To see why these `sync` instructions can be safely ignored, consider Figure 7, which illustrates how Tapir/LLVM's TRE pass operates on the `pqsort` function, a parallel version of Hoare's quicksort algorithm [20]. The original tail-recursive code is shown in Figure 7a. Figure 7b illustrates the result of simply inlining the tail-recursive call. For the inlined code, all `return` statements are replaced with branches to the `join`

a

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

b

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

c

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

Figure 7. Example of tail-recursion elimination on a parallel quicksort program. **a** The Cilk function `pqsort` sorts an array of integers in the range specified by the `start` and `end` pointers. **b** A version of `pqsort` where the recursive tail call on line 54 has been replaced by one round of inlining. **c** A version `pqsort` where tail-recursion elimination has removed the recursive tail call on line 54.

label. Because there is a `cilk_sync` at the start of `join`, the `cilk_sync` on line 71 can be eliminated. call an arbitrary number of times, TRE can safely ignore a `cilk_sync` instruction after the final tail-recursive call, assuming that it inserts a `cilk_sync` instruction before all remaining returns.

Parallel-loop scheduling and lowering

As discussed above and in Section 2, Tapir effectively represents a parallel loop as a serial loop over a body that is spawned every iteration. Depending on the number of iterations of the loop and the amount of work inside each loop, however, statically scheduling loop iterations in this way may be inefficient. For a parallel loop with a large number of iterations, for instance, it is faster to schedule the iterations in a recursive divide-and-conquer fashion, which produces more parallelism (see [46, Sec. 8.3]. For parallel loops with few iterations, however, the additional function calls required to perform the parallel divide-and-conquer can make the loop run slower than simply spawning off the iterations.

Tapir/LLVM implements a parallel optimization pass that schedules the iterations of a parallel loop using recursive divide-and-conquer, but only if that loop contains sufficiently many iterations. This pass is implemented as part of Tapir/LLVM's 3800-line lowering pass, which translates `detach`, `reattach`, and `sync` instructions into appropriate Cilk Plus runtime calls [22]. In particular, Tapir/LLVM uses the Cilk Plus runtime calls for `cilk_for` loops [22, Sec 10.7] to schedule parallel loops. Although we could have separated parallel-loop scheduling from lowering, we chose to combine these two passes so that we could perform fair comparisons between Tapir/LLVM and compilers that lower parallel constructs in their front end. We plan to separate the parallel-loop-scheduling and lowering passes in a future version of Tapir/LLVM.

Other optimization passes

Tapir/LLVM implements two minor parallel optimization passes: unnecessary-synchronization elimination and puny-task elimination. ***Unnecessary-synchronization elimination*** identifies and eliminates `sync` instructions that could not possibly sync a detached sub-CFG. ***Puny-task elimination*** serializes detached sub-CFG's that perform little or no work. If the runtime overhead of creating a parallel task outweighs the work in the task, the task might as well be run serially. Both of these optimization passes were implemented in 52 lines of code by augmenting LLVM's SimplifyCFG pass.

5. Auxiliary software

This section describes auxiliary software that we developed to exercise and test Tapir/LLVM. Although our research focuses on the middle end of the compiler, we implemented a front end for Cilk Plus. In addition, we developed compiler instrumentation that allows the compiler to interface to a race detector to verify the correctness of the Tapir/LLVM implementation.

To create the front end, we created a modification of the Clang front end called PClang, which translates Cilk Plus codes to Tapir. We also created a version of Clang that can handle some OpenMP codes. PClang handles most of the fork-join control constructs specified by the Cilk Plus programming model, and specifically, enough to run all the benchmarks described in Section 6.

We augmented Tapir/LLVM in two ways to test the correctness of the implementation. First, we modified LLVM's internal verification pass to check that Tapir's invariants are also maintained. Second, we added an instrumentation pass to Tapir/LLVM to allow parallel executables to be tested for determinacy races using a provably good determinacy race detector. This race detector, based on the SP-bags algorithm [14], is guaranteed to find a determinacy race if an only if one exists in the program execution. The verification pass and race detector helped us locate and fix bugs in Tapir/LLVM, both within our code and within the underlying LLVM codebase. Tapir/LLVM now passes all tests in LLVM's regression test suites and correctly compiles our own suite of parallel test programs.

The instrumentation pass has proved useful for supporting other dynamic-analysis tools based on Tapir/LLVM. Genghis Chau of MIT adapted the Cilkprof scalability profiler [61] to use Tapir/LLVM and this instrumentation in order to build an integrated development environment with always-on race detection and scalability profiling facilities.

6. Evaluation

To evaluate the effectiveness of the Tapir approach, we evaluated Tapir/LLVM on 20 benchmarks. The experiments support the contention that Tapir's approach of embedding parallelism in the IR is superior to lowering parallelism in the compiler front end. We could not simply run Tapir/LLVM against another compiler, such as Cilk Plus/LLVM [23], which lowers parallelism in the front end, because Cilk Plus/LLVM and Tapir/LLVM differ in more ways than just where they lower parallel constructs. Consequently, to perform an apples-to-apples comparison of these two approaches, we implemented a compiler called "Reference," which is as close to identical to Tapir/LLVM as we could muster, except for where lowering occurs. Figure 8 illustrates the compilation pipelines for Clang/LLVM, Tapir/LLVM, and Reference.

The first pipeline, Clang/LLVM, has the traditional three-phase structure. The Clang front-end takes serial C/C++ code and emits LLVM IR. The `-O3` middle-end optimizes the IR, and the CodeGen back-end lowers LLVM IR to machine code for a particular hardware platform.

The second pipeline shows how Tapir/LLVM is organized. The PClang front end takes parallel Cilk Plus code as input and emits Tapir. The middle-end now consists of three steps: `-O3` optimization, a Lower pass to lower Tapir to LLVM IR, and another pass at `-O3` optimization. The first `-O3` pass performs optimizations on the Tapir representation,

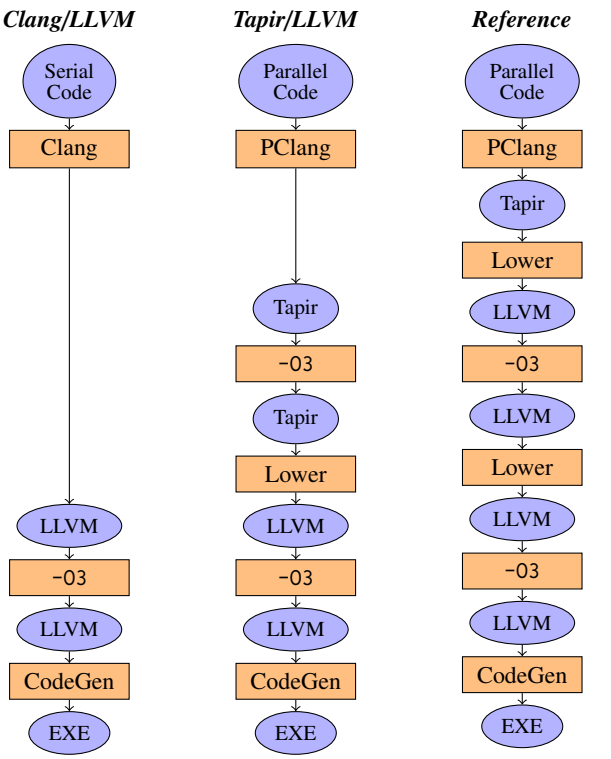

Figure 8. The compilation pipelines for Clang/LLVM, Tapir/LLVM, and Reference. Each block represents a compiler transformation, and each oval designates the format of the code at that point in the pipeline.

the lowering pass translates all the Tapir-specific constructs to LLVM IR, and the second `-O3` pass performs optimizations on the LLVM IR. Finally, the CodeGen back end lowers LLVM IR to machine code.

The third pipeline, called Reference, models how mainstream compilers work today, where parallel constructs are transformed into runtime calls before any optimization can take place. The only difference between Reference and Tapir/LLVM is that the Tapir code emitted by the PClang front end is immediately lowered to LLVM IR before the rest of the Tapir pipeline is invoked. (The second Lower pass in the Reference pipeline therefore has no effect.) Although Reference lowers the parallel constructs early, two iterations of `-O3` are included to ensure that the Tapir/LLVM gains no advantage from optimizing twice. Although one might think that a second pass of `-O3` would be redundant, it is not. For example, a simple matrix-multiplication code runs 13% faster after two rounds of optimization compared to just one. And although most benchmarks run faster after two `-O3` passes, some actually run slower. Thus, we implemented Reference with the same passes as Tapir/LLVM, except for the initial Lower pass in Reference. This difference only affects parallel code. Serial code passes through both pipelines identically.

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

Figure 9. Descriptions of the 20 benchmarks used to evaluate Tapir/LLVM. These benchmarks were taken from the MIT Cilk benchmark suite [16], Intel Cilk Plus example programs [25], and the CMU Problem-Based Benchmark Suite [63]. "MIS" denotes the computation of a maximal independent set of a graph. "BFS" denotes the breadth-first search of a graph.

Benchmarking

To benchmark the compiler pipelines, we assembled a collection of benchmark programs taken from the MIT Cilk benchmark suite [16], Intel Cilk code samples [25], and the CMU Problem-Based Benchmark Suite [63]. From these collections, we selected stable programs that tend to exhibit little performance difference when the number or order of optimization passes is changed. Figure 9 describes the suite of benchmarks tested.

We compiled each program in our benchmark suite with both Tapir/LLVM and Reference, and we ran them on both 1 and 18 cores of our test machine. Additionally, we compiled the serial elision of each benchmark with each compiler. Each running time is the minimum of 10 runs on an Amazon AWS `c4.8xlarge` spot instance, which is a dual-socket Intel Xeon E5-2666 v3 system with a total of 60 GiB of memory. Each Xeon is a 2.9 GHz 18-core CPU with a shared 25 MiB L3-cache. Each core has a 32 KiB private L1-data-cache and a 256 KiB private L2-cache. The system was "quiesced" to permit careful measurements by turning off Turbo Boost, dvfs, hyperthreading, extraneous interrupts, etc.

Overall performance

The results of our tests are given in Figure 10. For the first pair of rows, Reference and Tapir/LLVM produce essentially identical executables when compiling the serial elision of a benchmark. Differences in running times in these rows

		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

Figure 10. Comparison between executables compiled using Reference and using Tapir/LLVM. Each column refers to a different parallel benchmark described in Figure 9. Rows labeled "Ref." describe executables compiled using Reference, and rows labeled "Tapir" describe executables compiled using Tapir/LLVM. Each measured running time is the minimum over 10 executions, measured in seconds. The pair of rows labeled T_S gives the running time of the executable compiled from the serial elision of each benchmark. The pair of rows labeled T_1 gives the work of each benchmark. The pair of rows labeled T_{18} gives the 18-core running time of each benchmark. The pair of rows labeled T_S/T_1 gives the work efficiency of each compiled benchmark, derived from the first and second pairs of rows. The pair of rows labeled T_S/T_{18} gives the parallel speedup of each compiled executable on 18 cores, derived from the first and third pairs of rows.

are due to system noise. The second pair of rows shows that Tapir/LLVM produces executables with better work than Reference on 15 of the benchmarks. Of the remaining 5 benchmarks, 4 demonstrate less than a 1% difference between their work relative to Tapir/LLVM or Reference. The fourth pair of rows elaborates on the results in the second pair to show that Tapir/LLVM produces executables with nearly optimal work efficiency (within 1%) on 12 of the benchmarks, whereas Reference does so on only 2. The third and fifth pairs of row show that Tapir/LLVM generally produces executables with similar or better parallel speedups than those produced by Reference.

The biggest slowdown created from Tapir/LLVM's compilation occurs on Cholesky, for which the executable produced by Tapir/LLVM has 4% more work than that produced by Reference. In investigating this benchmark, we found that LLVM runs a handful of optimizations on each function before the middle-end optimization and lowering passes in either Tapir/LLVM or Reference. Although these early optimizations have little effect on most programs, they reduce the work of the Reference-compiled Cholesky executable by approximately 20%. Although we experimented with several ways to implement lowering in Reference before these early optimizations, the resulting compilers consistently exhibited bugs on other benchmarks in the suite. In our final design for Reference, we placed the initial lowering pass as early as we could muster while still ensuring that Reference could compile all benchmarks correctly.

7. Related work

This section describes related work in representing parallelism in a compiler IR and in analyzing and optimizing parallel programs.

Various prior research explores compiler optimizations on unstructured parallel threads. For example, some researchers have explored how to find and remove unnecessary synchronization in Java programs [3, 57]. Joisha *et al.* [26] present a technique to detect instructions that are unaffected by parallel threads and can be safely optimized across unstructured parallel control flow. In contrast, our work on Tapir focuses on compiler optimizations for structured parallelism, namely fork-join parallel programs with serial semantics. Although fork-join parallelism may be more restricted than unstructured parallel threads, Tapir demonstrates that many of the optimizations for serial code easily extend to fork-join parallelism. Enabling similar optimiza-

tions for unstructured parallel threads appears to be a much harder problem.

Some previous work on compiler optimizations for fork-join parallel programs evaluate which instructions can safely execute in parallel [1] based on concurrency mechanisms supported by a particular memory model. For example, Barik *et al.* [6, 7] use interprocedural analysis to perform various optimizations affecting critical sections of X10 and Habanero-Java programs. Rather than dealing with the complexities of general concurrency mechanisms, Tapir enables compiler optimizations for an easy-to-understand situation: when the optimization respects the serial semantics of the program and does not introduce determinacy races. Compared with general concurrency mechanisms, well-structured parallelism seems to offer a less onerous path to performance.

Khaldi *et al*. [30] modify LLVM IR to support OpenSHMEM parallel programs with the aim of achieving performance in modern network interconnects that support efficient data transfers for partitioned global address spaces (PGAS). Based on the SPIRE methodology [29] for representing parallel code, they augment functions, basic blocks, instructions, identifiers, and types in LLVM IR with execution, synchronization, scheduling, and memory-layout information. In contrast, Tapir models fork-join parallelism for shared-memory multicores, a conceptually simpler context than PGAS systems, and extends LLVM IR minimally using only three instructions. Once again, the Tapir's strong assumption of a fork-join programming model with serial semantics that compiles to a flexible multicore architecture seems to provide both performance and simplicity, albeit at the cost of scalability to huge cluster-based supercomputers that lack strong memory-consistency guarantees.

In contrast with much of the work referenced above, Chatarasi *et al*. [12] focus, as Tapir does, on fork-join programs with serial semantics. Specifically, they examine polyhedral optimizations on OpenMP programs with serial semantics. By combining dependency and happens-before analyses, they manage to enable traditional polyhedral optimizers to work on parallel loops, much as Tapir enables common middle-end compiler optimizations to work on parallel code.

8. Conclusion

To conclude, we would like to leave the reader with three interesting considerations regarding the nature of asymmetry in parallelism, the future of parallel optimizations, and extensions of Tapir-like systems to other models of parallel programming.

Reasoning about logically parallel tasks asymmetrically based on serial semantics can sometimes simplify the understanding of a parallel program's behavior. When a task is spawned to execute in parallel with another, it is natural to reason about the logically parallel tasks as symmetric, because their instructions can execute in any relative order. For parallel programs with serial semantics, however, it is always valid to execute the program on a single processor, which asymmetrically executes one parallel task to completion before starting the other. Serial semantics encourage an asymmetric representation of parallel control flow that is similar enough to its serial elision that most common analyses and transformations for serial programs work on parallel constructs with little or no modification. In particular, serial semantics enables common optimizations on parallel code that can be invalid under other models of parallelism [71].

One of the great benefits of Tapir is that its strategy for representing parallelism makes it easy to write optimization passes specifically for parallel code. Section 4 briefly mentioned some parallel optimization passes we implemented, including parallel-loop scheduling and unnecessary-sync elimination. In addition to helping close the performance gap between serial and parallel versions of code, we hope that the introduction of Tapir will encourage the development and implementation of many more parallel-optimization passes.

Finally, Tapir allows fork-join parallel programs to benefit from both serial and parallel optimizations. Moving forwards, it is natural to wonder whether other models of parallelism, such as pipeline parallelism [13, 33, 49] or data-graph computations [43–45, 51, 52, 62, 64], can take advantage of the Tapir approach.

Acknowledgments

William S. Moses was supported in part by an MIT EECS SuperUROP. This research was supported in part by NSF Grants 1314547 and 1533644, in part by a MIT CSAIL grant from Foxconn, and in part by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior/ Interior Business Center (DoI/IBC) contract number D16PC00002. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/IBC, or the U.S. Government.

Many thanks to our master bug-finder Tim Kaler of MIT, who also helped us with cloud technology for the artifact evaluation. Shahin Kamali of MIT and Bradley Kuszmaul, formerly of MIT and now of Oracle, were involved in many helpful discussions. Thanks to Johannes Doerfert and Simon Moll of Saarland University, Germany, for their feedback and insights. Thanks to Larry Hardesty of the MIT News Office for asking questions that helped us to simplify Figure 3. We are grateful to the students and staff of the Fall 2016 MIT class 6.172/6.871 *Performance Evaluation of Software Systems* for their patience in using the Tapir/LLVM compiler throughout the semester and reporting bugs. We thank the reviewers for their feedback.

References

[1] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *PPoPP*, pages 183–193, 2007.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, second edition, 2006.

[3] J. Aldrich, C. Chambers, E. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In A. Cortesi and G. Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 19–38. 1999.

[4] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA*, pages 119–129, 1998.

[5] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[6] R. Barik and V. Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *PACT*, pages 41–52, 2009.

[7] R. Barik, J. Zhao, and V. Sarkar. Interprocedural strength reduction of critical sections in explicitly-parallel programs. In *PACT*, pages 29–40, 2013.

[8] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5): 720–748, 1999.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[10] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, pages 68–78, 2008.

[11] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: the new adventures of old X10. In *PPPJ*, pages 51–61, 2011.

[12] P. Chatarasi, J. Shirako, and V. Sarkar. Polyhedral optimizations of explicitly parallel programs. In *PACT*, pages 213–226, 2015.

[13] W. Du, R. Ferreira, and G. Agrawal. Compiler support for exploiting coarse-grained pipelined parallelism. In *SC*, pages 8–21, 2003.

[14] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[15] J. T. Fineman and C. E. Leiserson. Race detectors for Cilk and Cilk++ programs. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1706–1719. 2011.

[16] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[17] GCC Team. GCC 4.9 release series changes, new features, and fixes. Available at `https://gcc.gnu.org/gcc-4.9/changes.html`, 2014.

[18] GCC Team. GOMP — an OpenMP implementation for GCC. Available at `https://gcc.gnu.org/projects/gomp/`, 2015.

[19] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *PPoPP*, pages 159–168, 1993.

[20] C. A. R. Hoare. Algorithm 64: Quicksort. *CACM*, 4(7):321, 1961. ISSN 0001-0782.

[21] Intel Corporation. *Intel Cilk Plus Language Specification*, 2010. Document Number: 324396-001US. Available from `http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf`.

[22] Intel Corporation. *Intel Cilk Plus Application Binary Interface Specification*, 2010. Document Number: 324512-001US. Available from `https://software.intel.com/sites/products/cilk-plus/cilk_plus_abi.pdf`.

[23] Intel Corporation. Cilk Plus/LLVM. Available from `http://cilkplus.github.io/`, 2013.

[24] Intel Corporation. *Intel C++ Compiler 16.0 User and Reference Guide*, 2015.

[25] Intel Corporation. Intel Cilk Plus samples. Available from `https://software.intel.com/en-us/code-samples/intel-compiler/intel-compiler-features/intelcilkplus`, 2016.

[26] P. G. Joisha, R. S. Schreiber, P. Banerjee, H. J. Boehm, and D. R. Chakrabarti. A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code. In *POPL*, pages 623–636, 2011.

[27] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer. INSPIRE: The Insieme parallel intermediate representation. In *PACT*, pages 7–18, 2013.

[28] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., second edition, 1988.

[29] D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin. SPIRE, a sequential to parallel intermediate representation extension. Technical report, Technical Report CRI/A-487, MINES ParisTech, 2012.

[30] D. Khaldi, P. Jouvelot, F. Irigoin, C. Ancourt, and B. Chapman. LLVM parallel intermediate representation: Design and evaluation using OpenSHMEM communications. In *LLVM*, pages 2:1–2:8, 2015.

[31] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM TOPLAS*, pages 268–299, 1996.

[32] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, pages 75–87, 2004.

[33] I.-T. A. Lee, C. E. Leiserson, T. B. Schardl, Z. Zhang, and J. Sukha. On-the-fly pipeline parallelism. *ACM TOPC*, 2(3): 17:1–17:42, 2015.

[34] J. Lee, S. P. Midkiff, and D. A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *LCPC*, pages 114–130, 1997.

[35] C. E. Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51(3):244–257, 2010.

[36] LLVM Developer List. LLVMdev discussions on Intel OpenMP proposal. Available from `http:`

//lists.llvm.org/pipermail/llvm-dev/2012-September/053861.html, September 2012.

[37] LLVM Developer List. LLVMdev Parallelization metadata and intrinsics in LLVM (for OpenMP, etc.). Available from http://lists.llvm.org/pipermail/llvm-dev/2012-September/053792.html, September 2012.

[38] LLVM Developer List. LLVMdev discussions on OpenCL SPIR proposal. Available from http://lists.llvm.org/pipermail/llvm-dev/2012-September/053293.html, September 2012.

[39] LLVM Developer List. LLVMdev discussions on parallel IR. Available from http://lists.llvm.org/pipermail/llvm-dev/2015-March/083314.html, March 2015.

[40] LLVM Project. OpenMP®: Support for the OpenMP language. Available at http://openmp.llvm.org/, 2015.

[41] LLVM Project. *LLVM Language Reference Manual*, 2015. Available from http://llvm.org/docs/LangRef.html.

[42] LLVM Project. *LLVM's Analysis and Transform Passes*, 2015. Available from http://llvm.org/docs/Passes.html.

[43] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI*, pages 340–349, 2010.

[44] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, pages 716–727, 2012.

[45] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[46] M. McCool, A. D. Robison, and J. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science, 2012.

[47] S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In *ICPP*, pages 105–113, 1990.

[48] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[49] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval. Analytical modeling of pipeline parallelism. In *PACT*, pages 281–290, 2009.

[50] R. H. B. Netzer and B. P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1): 74–88, 1992.

[51] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471, 2013.

[52] D. Nguyen, A. Lenharth, and K. Pingali. Deterministic Galois: On-demand, portable and parameterless. In *ASPLOS*, pages 499–512, 2014.

[53] D. Novillo, R. Unrau, and J. Schaeffer. Concurrent SSA form in the presence of mutual exclusion. In *ICPP*, pages 356–364, 1998.

[54] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 4.0*, July 2013. Available from http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[55] A. Pop and A. Cohen. Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers. In *CPC*, 2010. URL https://hal.inria.fr/inria-00551518.

[56] W. Pugh. Fixing the Java memory model. In *JAVA*, pages 89–98, 1999.

[57] E. Ruf. Effective synchronization removal for Java. In *PLDI*, pages 208–218, 2000.

[58] R. Rugina and M. C. Rinard. Pointer analysis for structured parallel programs. *TOPLAS*, pages 70–116, Jan. 2003.

[59] V. Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *LCPC*, pages 94–113, 1998.

[60] V. Sarkar and B. Simons. Parallel program graphs and their classification. In *LCPC*, pages 633–655, 1994.

[61] T. B. Schardl, B. C. Kuszmaul, I.-T. A. Lee, W. M. Leiserson, and C. E. Leiserson. The Cilkprof scalability profiler. In *SPAA*, pages 89–100, 2015.

[62] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

[63] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *SPAA*, pages 68–70, 2012.

[64] J. Shun, L. Dhulipala, and G. E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *DCC*, pages 403–412, 2015.

[65] H. Srinivasan and D. Grunwald. An efficient construction of parallel static single assignment form for structured parallel programs. Technical report, Technical Report CU-CS-564-91, University of Colorado at Boulder, 1991.

[66] H. Srinivasan and M. Wolfe. Analyzing programs with explicit parallelism. In *LCPC*, pages 405–419, 1991.

[67] H. Srinivasan, J. Hook, and M. Wolfe. Static single assignment for explicitly parallel programs. In *POPL*, pages 260–272, 1993.

[68] R. M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection (for GCC version 6.1.0)*. Free Software Foundation, 2016.

[69] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 4th edition, 2013.

[70] R. Utterback, K. Agrawal, J. T. Fineman, and I.-T. A. Lee. Provably good and practically efficient parallel race detection for fork-join programs. In *SPAA*, pages 83–94, 2016.

[71] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL*, pages 209–220, 2015.

[72] J. Zhao and V. Sarkar. Intermediate language extensions for parallelism. In *SPLASH*, pages 329–340, 2011.

A. Artifact description

A.1 Abstract

This guide describes how to set up Tapir/LLVM and how to download and run our suite of application benchmarks. In particular, this guide focuses on setting up and running three software components:

- the Tapir/LLVM compiler,
- the PClang front end to Tapir/LLVM, and
- the suite of 20 Cilk application benchmarks described in Figure 9.

We provide instructions to download and build Tapir/LLVM and PClang. We also provide instructions to download the application benchmark suite and run the Tapir/LLVM compiler on that suite.

We have built and tested Tapir/LLVM, PClang, and the test suite on an `x86_64` shared-memory multicore machine running Linux. We provide instructions for obtaining Tapir/LLVM and PClang from our GitHub repositories and setting up the compiler on such a machine. Due to the complexity of the LLVM compiler on which Tapir/LLVM is based, building Tapir/LLVM requires significant computational resources: approximately 50 GiB of disk, 12 GiB of RAM, and anywhere from a few minutes to a couple of hours, depending on the machine. We also provide instructions for obtaining a copy of our test suite from a tarball.

We provide scripts to run all application benchmarks and evaluate the performance results to produce a table of Tapir/LLVM results similar to those in Figure 10. Running all of the tests takes approximately 7 hours. Because we are continuing to develop the Tapir/LLVM compiler, the performance results will not precisely match those in Figure 10.

A.2 Description

Check-list (artifact meta information):

- **Program:** The application benchmark suite includes the 20 Cilk programs described in Figure 9. Packaged with this suite are scripts for compiling and running the tests using the Tapir/LLVM compiler and PClang front end. This suite is publicly available through a tarball, which is approximately 12 GiB in size.
- **Compilation:** Our main software artifact is the Tapir/LLVM compiler. The source code for both Tapir/LLVM and its associated PClang front end are publicly available from GitHub. Due to the complexity of the underlying LLVM and Clang codebases, building Tapir/LLVM and PClang from source requires significant computational resources, even though the changes to implement Tapir/LLVM amount to only about 6000 lines of code.
- **Data set:** The application benchmark suite includes all necessary data sets to run the applications.
- **Run-time environment:** Both Tapir/LLVM and the benchmark suite employ the Intel Cilk Plus runtime system, which is available on several modern Linux distributions via a mainstream package. The evaluation also employs several Linux commands which are similarly available through mainstream packages. We have built and tested Tapir/LLVM on Ubuntu 16.04 and Fedora 24. Root accesses is not required to install Tapir/LLVM, PClang, or the application benchmark suite, but it is required to install the Intel Cilk Plus runtime system and auxiliary Linux commands.
- **Hardware:** We have built and tested Tapir/LLVM on an `x86_64` system. A shared-memory multicore system is required to execute the compiled benchmarks in parallel.
- **Output:** Running the benchmark tests produces a table of performance results similar to Figure 10. These performance results are likely to vary between machines. Moreover, because we are continuing to develop Tapir/LLVM and PClang, these performance results will not perfectly match the values in Figure 10.
- **Experiment workflow:** We provide scripts with the application benchmark suite to run the tests and aggregate the results. The total running time of this experiment is approximately 7 hours on an AWS `c4.8xlarge` instance, but can vary between machines.
- **Experiment customization:** You can run separate sets of tests within the benchmark suite. You can also write your own programs to compile using Tapir/LLVM and PClang. The PClang front end supports a variant of the Cilk programming language and produces Tapir code that can be processed and optimized by Tapir/LLVM.
- **Publicly available?:** Tapir/LLVM and PClang are publicly available on GitHub. The test suite is publicly available via a tarball.

How delivered: You can download and build Tapir/LLVM and PClang from our GitHub repositories, and you can download the application benchmark test suite from a tarball. Although Tapir represents a relatively small modification to the LLVM compiler, it inherits a good deal of complexity from the underlying LLVM software infrastructure. The instructions below aim to streamline the process of building and testing Tapir/LLVM in spite of this complexity.

Hardware dependencies: Tapir/LLVM, PClang, and the test suite have been built and tested to run on `x86_64`. A shared-memory multicore system is needed to evaluate the parallel performance of programs compiled using Tapir/LLVM.

Software dependencies: We have tested Tapir/LLVM, PClang, and the test suite on Ubuntu 16.04 and on Fedora 24. Building Tapir/LLVM and PClang requires `cmake` and a C/C++ compiler. The test script uses `taskset` and `numactl` to quiesce the system for running each benchmark, specifically, to mitigate the effects of hyperthreading and NUMA on the execution of the tests. The test script also uses `bc` to calculate derived values from the measured running times of the tests.

Datasets: The tarball of application benchmarks includes all necessary data sets to execute these applications.

A.3 Building Tapir/LLVM from source

This section describes how to download the source code for Tapir/LLVM and PClang from GitHub and build them. These instructions assume you are building Tapir/LLVM on an `x86_64` system running Linux.

System requirements. Building Tapir/LLVM and PClang involves building the LLVM and Clang systems that they extend. Because of the size of the underlying LLVM and Clang codebases, you need a relatively powerful machine in order to build the compiler in a timely fashion. Approximately 50 GiB of disk space and 12 GiB of memory are needed to compile LLVM and Clang. A fresh build of LLVM and Clang can take substantial time to complete, e.g., approximately an hour on one processor of an AWS `c4.8xlarge` instance. The build script will attempt to use parallel processors to speed up compilation. See `http://llvm.org/docs/CMake.html` for more information on building LLVM and Clang.

1. Install the requisite software to build Tapir/LLVM and PClang, namely, `cmake`, `gcc`, and `git`.
2. Download the sources of Tapir/LLVM and PClang from GitHub:

   ```
   $ git clone --recursive \
   > https://github.com/wsmoses/Tapir-Meta.git
   ```

   The source is approximately 800 MiB in size.
3. Compile Tapir/LLVM and PClang:

   ```
   $ cd Tapir-Meta/
   $ bash ./build.sh
   ```

   This script will build Tapir/LLVM and PClang and store the compiled binaries in `Tapir-Meta/tapir/build`. If the build succeeds, the final line of output will be `Installation successful`.
4. Set up your environment variables to use Tapir/LLVM and PClang:

   ```
   $ source ./setup-env.sh
   ```

   This script will add the `Tapir-Meta/tapir/build/bin/` subdirectory to your path, so that the `clang` command will refer to Tapir/LLVM and PClang.

A.4 Running the benchmark suite

This section describes how you can download the application benchmark suite described in Figure 9 and test Tapir/LLVM on these benchmarks.

1. Install the requisite software to download and run the tests, namely, `bc`, `libcilkrts`, `numactl`, `python`, `taskset`, and `wget`.
2. Download the tarball containing the application benchmark suite and unpack it:

   ```
   $ wget http://tinyurl.com/TapirLLVMTesting -O testing.tar
   $ tar -xvf testing.tar
   ```

   This tarball is approximately 12 GiB in size. Unpacking the tarball creates the `testing/` subdirectory of the current working directory that contains the application benchmark suite.
3. Run the test script:

   ```
   $ cd testing
   $ ./test.sh
   ```

   The test script takes approximately 7 hours to run. The script compiles each benchmark in the test suite twice using Tapir/LLVM: once as a parallel program, and once as the program's serial elision. All compilations use optimization level `-O3`. The test script runs each compiled executable 10 times using 1 worker thread and 10 times using 18 worker threads.

A.5 Evaluation and expected result

Once the test script finishes running, the results can be summarized into a table similar to Figure 10 as follows:

```
$ ./results.sh > results.csv
```

This command will produce `results.csv`, a table of tab-separated values that contains the minimum running time from each set of 10 runs of a particular executable on a particular worker count. The table also contains derived work-efficiency and parallel speedup values for each benchmark program.

Because these results are performance measurements, they are likely to vary from run to run and from system to system. Moreover, we are continuing to develop the Tapir/LLVM compiler and PClang, meaning that your results will not pecisely match those in Figure 10.

A.6 Experiment customization

We provide scripts for running individual sets of tests within the test suite. The `testCilk.sh`, `testIntel.sh`, and `testPBBS.sh` scripts recompiles and reruns all tests from the MIT Cilk benchmark suite, the set of Intel Cilk Plus example programs, and the CMU Problem-Based Benchmark Suite, respectively. Running any of these scripts updates the performance results for the designated subset of tests. Rerunning the `results.sh` script produces a new table with the latest test results from each subset.

You can write your own programs and compile them using PClang and Tapir/LLVM. The PClang front end is *not* a fully featured Cilk front end, however. For more information on the source language parsed by PClang, please see `testing/PClang-README.txt`.