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

TAO B. SCHARDL, WILLIAM S. MOSES, and CHARLES E. LEISERSON, MIT Computer Science and Artificial Intelligence Laboratory, USA

Tapir (pronounced TAY-per) is a compiler intermediate representation (IR) that embeds recursive fork-join parallelism, as supported by task-parallel programming platforms such as Cilk and OpenMP, into a mainstream compiler's 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 on and across parallel control constructs. Remedying this situation has generally been thought to require an extensive reworking of compiler analyses and code transformations to handle parallel semantics. Tapir leverages the "serial-projection property," which is commonly satisfied by task-parallel programs, to handle the semantics of these programs without an extensive rework of the compiler.

For recursive fork-join programs that satisfy the serial-projection property, Tapir enables effective compiler optimization of parallel programs with only minor changes to existing compiler analyses and code transformations. Tapir uses the serial-projection property to order logically parallel fine-grained tasks in the program's control-flow graph. This ordered representation of parallel tasks allows the compiler to optimize parallel codes effectively with only minor modifications. For example, to implement Tapir/LLVM, a prototype of Tapir in the LLVM compiler, we added or modified less than 3,000 lines of LLVM's half-million-line core middle-end functionality.

These changes sufficed to enable 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 parallel loops and Cilk's `cilk_spawn` keyword. Tapir also supports parallel optimizations, such as loop scheduling, which restructure the parallel control flow of the program. By making use of existing LLVM optimizations and new parallel optimizations, Tapir/LLVM can optimize recursive fork-join programs more effectively than traditional compilation methods. On a suite of 35 Cilk application benchmarks, Tapir/LLVM produces more efficient executables for 30 benchmarks, with faster 18-core running times for 26 of them, compared to a nearly identical compiler that compiles parallel linguistic constructs the traditional way.

CCS Concepts: • **Computing methodologies** → **Parallel programming languages**; • **Software and its engineering** → **Compilers**;

William S. Moses was supported in part by an MIT EECS SuperUROP. This research was supported in part by NSF Grants No. 1314547 and No. 1533644, in part by an 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 No. 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.

Authors' addresses: T. B. Schardl, W. S. Moses, and C. E. Leiserson, MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA, 02139, USA; emails: {neboat, wmoses, cel}@mit.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Association for Computing Machinery.

2329-4949/2019/12-ART19 \$15.00

<https://doi.org/10.1145/3365655>

Additional Key Words and Phrases: Cilk, compiling, control-flow graph, fork-join parallelism, LLVM, multi-core, OpenMP, optimization, parallel computing, serial-projection property, Tapir

ACM Reference format:

Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2019. Tapir: Embedding Recursive Fork-join Parallelism into LLVM’s Intermediate Representation. *ACM Trans. Parallel Comput.* 6, 4, Article 19 (December 2019), 33 pages.

<https://doi.org/10.1145/3365655>

1 INTRODUCTION

Mainstream compilers, such as GCC [104], ICC [48], and LLVM [58], now support **recursive fork-join parallelism** [30, 31, 46, 68] as expressed by task-parallel linguistic extensions including Cilk Plus [45] and OpenMP [6, 82].¹ Fork-join parallelism² allows subroutines to be spawned recursively in parallel and iterations of a parallel loop to execute concurrently. Conceptually, the execution of a fork-join program generates a series-parallel dag [25] of logically parallel fine-grained 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 [5, 13, 14, 29] to schedule and load-balance the computation among parallel worker threads. Fork-join programs provide serial semantics [29] and admit efficient tools to detect “determinacy races” or validate their absence [25, 26, 108].

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 7 and 8 in Figure 1(a), which indicates that iterations of the loop are free to execute in parallel. If the `cilk_for` keyword is replaced by an ordinary `for` keyword, then all of the compilers GCC 7.3.1, ICC 16.0.3, and Cilk Plus/LLVM 3.9.0 observe that the call to `norm` on line 8 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 1(b).

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 [53] or C++ [107], 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 representation (IR)**, which represents the control flow of the program as a more-or-less language-independent **control-flow graph (CFG)** [3, Section 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 1(a), for example, the front end translates the parallel loop in lines 7 and 8 into IR in two steps. (The OpenMP code in Figure 1(b) is handled similarly.) First, the loop body (line 8) is lifted into a helper

¹Unlike the other compilers, however, LLVM support for Cilk Plus is not in the main branch, but in a separate Cilk Plus/LLVM branch [46].

²Henceforth, for simplicity, when we say fork-join parallelism, we mean recursive fork-join parallelism.

a

```

01 __attribute__((const))
02 double norm(const double *A, int n);
03
04 void normalize(double *restrict out,
05               const double *restrict in,
06               int n) {
07     cilk_for (int i = 0; i < n; ++i)
08         out[i] = in[i] / norm(in, n);
09 }

```

b

```

10 __attribute__((const))
11 double norm(const double *A, int n);
12
13 void normalize(double *restrict out,
14               const double *restrict in,
15               int n) {
16     #pragma omp parallel for
17     for (int i = 0; i < n; ++i)
18         out[i] = in[i] / norm(in, n);
19 }

```

Fig. 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 7 and 8 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 8 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.

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 article aims to enable middle-end optimizations involving fork-join parallel control flow by embedding parallelism directly into the compiler IR, an endeavor that has historically proven to be challenging [64, 65]. For example, it is well documented [74] 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 [70, section entitled, “`llvm.loop.parallel_accesses`’ Metadata”], 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. LLVM allows compiler transformations to discard metadata without jeopardizing program semantics [70, section entitled “Metadata”]. Moreover a compiler transformation that moves memory accesses into a parallel loop, such as function inlining [3, p. 903], or that disrupts the loop’s structure, such as by completely unrolling the loop, risks serializing the loop from LLVM’s perspective.

Second, the compiler can use intrinsic functions to demark parallel tasks. (For examples, see References [66, 85, 113].) 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 [8, 113], SPIRE [54], and INSPIRE [52] representations, for instance, model parallel constructs using an alternative IR, such as one based on the program’s abstract syntax tree [3, Section 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

Fig. 2. Comparison between a traditional CFG with a symmetric representation for parallel tasks for parallel tasks. (a) The Cilk function `fib`, which computes Fibonacci numbers. The `cilk_spawn` on line 23 allows the two recursive calls to `fib` to execute in parallel, and the `cilk_sync` on line 25 waits for the spawned call to return. A serial execution of `fib` executes `fib(n-1)` before `fib(n-2)`. (b) A comparable `fib` implementation using OpenMP tasks. (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 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 orders parallel tasks. 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.

historically been criticized [67] 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 [102, 103], parallel flow graphs [34, 101], concurrent control-flow graphs [62, 80], and parallel program graphs [92, 93] represent parallel tasks as symmetric entities in a CFG. For the parallel `fib` function in Figures 2(a) and 2(b), for example, the parallel flow graph in Figure 2(c) illustrates how forked subcomputations might be represented symmetrically. Some of these approaches struggle to represent common parallel constructs, such as parallel loops [54, 62], while others exhibit problems when subjected to standard compiler analyses and transformations for serial programs [34, 56, 62, 91, 92, 102, 103]. 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 2(c), however, instructions in the `join` block must observe the values of `x` and `y` from both of its predecessors, as has been observed in Reference [62]. Parallel loops exacerbate this problem by allowing a dynamic number of tasks to join at the same basic block. Previous research [1, 91] has proposed solutions to these problems, including additional representations of the program and augmented analyses that account for interleavings of parallel instructions, but adopting these techniques into a mainstream compiler seems to require extensive changes to the existing codebase.

Serial projections

This article introduces Tapir, a compiler IR that augments an existing IR to encode logical parallelism but leverages the “serial-projection property” to overcome problems faced by previous approaches. Before describing Tapir itself, let us discuss this key property.

The Tapir approach is premised on the notion that parallel programs can usually be executed serially and that there is a normative serial execution order for tasks. For example, a `cilk_for` loop can be executed serially by treating the `cilk_for` keyword as an ordinary `for` keyword, and a simple OpenMP parallel loop can be executed serially by ignoring the OpenMP pragma directive. Similarly, a function call preceded by a `cilk_spawn` keyword, which causes the function to be invoked in parallel with the continuation in the parent, can be executed serially by ignoring the `cilk_spawn` keyword.

A *serial projection*³ of a parallel program captures this notion of executing parallel tasks in their normative serial order. Mathematically, a *projection* of a set A is an idempotent mapping $f : A \rightarrow A$, that is, for all $x \in A$, we have $f(f(x)) = f(x)$. A serial projection transforms a given program⁴ to replace parallel constructs with serial constructs. A serial projection satisfies the idempotency property, since a serial projection of a serial program produces the same serial program.

A parallel program obeys the *serial-projection property* if there exists a serial projection of the program that is a refinement [75, Ch. 1] of the original program, meaning that the execution of the serial projection produces one of the valid behaviors of the original program. In other words, the compiler and runtime system can assume that it is okay to execute parallel tasks serially, instead of in parallel. Intuitively, executing a program in parallel normally means that the instructions of two parallel tasks can interleave arbitrarily⁵ subject to other ordering constraints in the program. The serial-projection property says that it is always okay to execute all of the first task’s instructions before any of the second task’s instructions. Most task-parallel programming models support this property. Section 8 discusses models that do not.⁶

Serial projections vary in how drastically they modify program text. A serial projection of a Cilk program can employ simple transformation rules [29]: a `cilk_spawn` preceding a function call can simply be elided, leaving the function call in place, and a `cilk_sync` statement can be elided after `cilk_spawn` statements it might synchronize have been transformed. Other parallel constructs

³This notion was originally called the *serial elision* in early Cilk work [29], but this article adopts the more accurate mathematical term “serial projection.”

⁴Following convention for describing program transformations [110], this article describes a serial projection as being applied to a whole program. In general, a serial projection and the serial-projection property can be applied to a program fragment [17] that contains all parallel constructs that depend on each other semantically, e.g., all `cilk_spawn` statements that a `cilk_sync` statement would synchronize.

⁵For the purposes of this intuition, let us assume sequential consistency [57].

⁶The serial-projection property is not the same as ordered parallelism [50, 84], which Jeffrey et al. state as the requirement that parallel tasks “must follow a total or partial order.” In contrast, if a program that satisfies the serial-projection property, then there exists a total order of its tasks that is valid, although other orders might exist as well.

can have more complicated serial projections. For example, a serial projection of an OpenMP parallel loop with data-sharing attribute clauses [82, Section 2.15.3], such as `firstprivate` and `lastprivate`, might introduce new variable declarations and assignments within and outside the loop body.

The serial-projection property introduces an asymmetry into parallel tasks in that it imposes a linear order among the parallel tasks representing the order in which they would execute in a serial projection. In contrast to the Tapir approach, all prior approaches to embedding parallelism into the IR of a compiler, to our knowledge, involve treating parallel tasks symmetrically.

The Tapir approach

The Tapir compiler IR leverages the serial-projection property to enable effective compiler optimization of parallel programs with only minor changes to existing compiler analyses and code transformations. Tapir orders fork-join task parallelism in the program's CFG, based on its serial projection. Tapir adds three instructions—`detach`, `reattach`, and `sync`—to the IR of an ordinary serial compiler to express fork-join parallel control flow. Figure 2(d) illustrates the Tapir CFG for the `fib` function. As with the symmetric parallel flow graph in Figure 2(c), Tapir places the logically parallel recursive calls to `fib` in separate basic blocks. But these blocks do not join symmetrically at a synchronization point. Instead, one block connects to the other, reflecting the execution order of the serial projection.

The Tapir approach provides five advantages:

- (1) Introducing recursive fork-join parallelism into the compiler is relatively easy.
- (2) The IR is expressive and can represent fork-join parallel control constructs from different parallel-language extensions.
- (3) By leveraging the serial-projection property, Tapir allows the compiler to reason about parallel programs similarly to how it reasons about serial programs.
- (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.

(1) Ease of implementation

Tapir's ordered representation of logically parallel tasks makes it relatively simple to integrate Tapir into an existing compiler's intermediate representation such as LLVM IR [70]. Figure 3 documents the lines of code added, modified, or deleted to implement Tapir/LLVM, a prototype of Tapir in LLVM.

The breakdown of lines is as follows. The lines for "Core middle-end functionality" add Tapir's instructions to LLVM IR and adapt LLVM's analyses and optimizations to handle Tapir. These lines are partitioned into five components:

- "Base classes" implement programming utilities that are shared throughout LLVM's implementation.
- "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.
- "Memory behavior" controls how Tapir instructions interact with LLVM variables and memory operations, preventing the compiler from creating any races.
- The lines for "Other analyses" implement changes to other analyses in LLVM, including a simple cost analysis for Tapir's instructions.

Compiler component	LLVM 6.0	Tapir/LLVM
Core middle-end functionality	500,283	2,989
• Base classes	62,488	0
• Instructions	141,321	1,013
• Memory behavior	18,907	536
• Other analyses	84,348	17
• Optimizations	193,219	1,423
Regression tests	3,482,802	5,745
Parallelism lowering	0	5,780
Parallel-tool support	0	3,341
Other	1,856,877	285
Total	5,839,962	18,140

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

- “Optimizations” perform adjustments required for LLVM 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 parallel variants of standard optimizations, such as parallel tail-recursion elimination (described in Section 4).

“Regression tests” account for all regression tests of the new Tapir instructions. “Parallelism lowering” translates Tapir instructions into Cilk Plus runtime calls. “Parallel-tool support” allows Tapir programs to be debugged using custom tools, including a provably good race detector [25]. “Other” addresses a correctness issue in LLVM’s implementation of `setjmp` and implement useful features for the development environment.

As Figure 3 shows, the entirety of Tapir/LLVM was implemented with fewer than 20,000 source lines of code, as compared to LLVM’s roughly 6-million-line codebase. Of these lines, fewer than 3,000 lines of code were added or modified as part of core middle-end functionality to adapt LLVM’s existing compiler analyses and transformations to accommodate Tapir, as compared to over half a million lines in LLVM proper. Approximately 6,000 lines implement regression tests to ensure correctness, 6,000 lines implement lowering transformations in LLVM to generate parallel executables from Tapir, and just over 3,000 lines implement debugging tools.

The simplicity of the Tapir approach is further demonstrated by the small amount of work required to port Tapir to new versions of LLVM. Since the original publication [97] of this work, we have ported Tapir/LLVM to every new release of LLVM. Each port required only about 4 days of programmer effort to implement and test.

(2) Expressiveness of Tapir

Tapir can express a wide range of fork-join task-parallel programs by using its instructions in combination with existing compiler IR constructs. We discuss the expressiveness of both Tapir’s instructions for fork-join parallel control flow and Tapir as a whole.

Tapir’s instructions can express the logical fork-join parallel control flow supported by a variety of task-parallel linguistic constructs. The example from Figure 2 illustrates how Tapir can express the parallelism encoded by the `cilk_spawn` and `cilk_sync` linguistics from Cilk++ [63] and Cilk Plus [45], as well as the parallelism encoded by OpenMP `task` and `taskwait` clauses [6]. Similarly, Tapir can express the parallelism encoded by OpenMP tasks [82] and Habanero’s `async` and `finish` constructs, as recent research has demonstrated [18, 106]. Section 2 describes how Tapir

can also express parallel loops, including `cilk_for` loops and simple OpenMP parallel loops. Using these instructions, Tapir can directly handle internally deterministic programs [79] that only use fork-join linguistic constructs.

When Tapir’s instructions are used together with existing compiler IR constructs, Tapir can express fork-join programs that use other parallel linguistic constructs as well, including non-deterministic programs, as long as these programs obey the serial-projection property. Tapir can express fork-join OpenMP programs that use data-sharing attribute clauses [82, Section 2.15.3] such as `private`, `firstprivate`, and `lastprivate`. Tapir instructions are also compatible with existing compiler constructs for parallel reduction mechanisms [28, 47, 61, 73, 82, 88, 98, 111]. Concurrency mechanisms, such as mutual-exclusion locks, condition variables, and atomic operations, are compatible with Tapir instructions when they are used following certain patterns (see, for example, References [12, 99]). These and other features can be used with Tapir instructions, because the resulting program satisfies the serial-projection property. Section 8 discusses parallel programs that do not satisfy the serial-projection property.

Tapir makes minimal assumptions about the consistency of concurrent memory accesses [16, 86]. Tapir assumes that memory is shared among parallel tasks and that virtual-register state is local to each task. Instructions in Tapir exhibit a *determinacy race*⁷ [25] 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 [70, section entitled, “Atomic Memory Ordering Constraints”] to define whichever memory model they choose. For any targeted runtime system, Tapir relies on a correct implementation of lowering to implement the necessary synchronization, but Tapir is oblivious to how the given runtime system implements the synchronization.

(3) *Leveraging the serial-projection property*

Tapir makes use the serial-projection property to enable effective compiler analysis and optimization of parallel code. Section 3 discusses how the serial-projection property implies that a semantically valid execution of parallel tasks does not require strong forward-progress guarantees on the execution of logically parallel instructions. Intuitively, this property allows the compiler to reason about the semantics of a task separately from instructions that might execute in parallel with that task. The compiler can thus more easily discover operations that can be moved around parallel control flow without jeopardizing the program’s semantics. This property also allows the compiler to restructure a program’s parallel control flow, such as by combining tasks, reordering tasks, or rearranging task creation and synchronization. Section 4 describes several example optimizations on Tapir programs.

In contrast, parallel programs that do not satisfy the serial-projection property challenge compiler analysis and optimization. These constructs might require forward progress on all logically parallel instructions for an execution to be semantically valid. To handle these constructs, the compiler must reason about the forward progress of instructions that might execute in parallel, and common optimizations on parallel code, such as “sequentialization,” may be invalid [109]. But many fork-join programs do satisfy the serial-projection property and are thus amenable to effective compiler optimization based on simpler analysis.

⁷Determinacy races are also called general races [79] and are distinct from data races, which involve nonatomic accesses to critical regions.

Fig. 4. Comparison of the work efficiency of 35 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 projection 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 executables compiled by Tapir/LLVM and executables compiled by the reference compiler.

(4) Optimizations

In practice, we have found that Tapir enables a wide variety of standard compiler optimizations to work on 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 [77, Section 12.2], loop-invariant-code motion [77, Section 13.2], and tail-recursion elimination [77, Section 15.1], to work on parallel code, as well as new optimizations to the parallel control flow of the Tapir program, such as parallel-loop scheduling.

(5) Evaluation of Tapir/LLVM

The compiler optimizations that Tapir enables are effective in practice. We evaluated the Tapir approach by measuring the performance of 35 Cilk application benchmarks compiled using Tapir/LLVM. We compared the performance of these executables to those produced by a comparable reference compiler, which lowers parallel linguistic constructs directly into runtime calls, as mainstream compilers do today, but otherwise performs the same sequence of optimization passes as Tapir/LLVM. Section 6 describes our experimental setup in detail, including the designs of both Tapir/LLVM and the comparable reference compiler.

Figure 4 presents the results of comparing Tapir/LLVM and the reference compiler 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 multi-core 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 projection of each benchmark, denoted T_S , in which the benchmark is statically 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 . In theory, the maximum possible work efficiency is $T_S/T_1 = 1$, but in practice, quirky behaviors of the compiler and multi-core architecture can occasionally produce work efficiencies greater than 1. As Figure 4 shows, for nearly all benchmarks, the executables compiled using Tapir/LLVM achieve equal or higher

work efficiency than those compiled using the reference compiler. Moreover, for many benchmarks, Tapir/LLVM produces executables that achieve nearly optimal work efficiency. Section 6 elaborates on these experiments.

Contributions

This article makes the following research contributions:

- The design of a compiler IR that orders fork-join parallel tasks, which enables existing serial optimizations to operate on parallel code and also enables parallel optimizations.
- The implementation of Tapir/LLVM in the LLVM compiler by modifying less than 3,000 source lines of code, 0.6% of the half-million-line core middle-end functionality of the LLVM codebase.
- The implementation of parallel optimizations such as redundant 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 article 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, and Section 9 describes the current status of the Tapir/LLVM compiler.

2 TAPIR

This section describes how Tapir represents logically parallel tasks in the CFG of a program. We define Tapir's three new instructions and examine how these instructions model fork-join parallelism and parallel loops. We describe structural properties of Tapir. We briefly present informal semantics for Tapir, including the serial projection of a Tapir program. We examine how Tapir instructions interact with LLVM's `alloca` instruction [70, section entitled, "‘alloca’ Instruction"] for stack-memory allocation and with LLVM's static single-assignment (SSA) form [3, Section 6.2.4]. We describe two extensions to Tapir to support inlining of parallel control flow and integration with exception-handling code. Although we describe Tapir as an extension to LLVM IR [70], 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.

The final instruction in a basic block is a **terminator** instruction, which specifies how control flows out of the basic block.

Tapir instructions

Tapir extends LLVM IR with three instructions: `detach`, `reattach`, and `sync`. The `detach` and `reattach` instructions define the boundaries of logically parallel tasks, and the `sync` instruction

performs dynamic synchronization of 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` terminator instructions delineate the beginning and end of a parallel task, respectively. A `detach` instruction terminates the block a that contains it and takes two operands: a **detached** block b and a **continuation** block c . At runtime, the `detach` instruction **spawns** a new parallel task that starts 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 operand, inducing a **reattach** edge $(a', c) \in E$ in the CFG. At runtime, the `reattach` first terminates the parallel task containing it. The `reattach` then allows its continuation block to be executed if execution of that block has not yet begun. Together, a `detach` instruction and its associated `reattach` instructions demark the start and end of a parallel task, and the `detach` instruction grants permission to the task to execute in parallel with its continuation block.

Figure 2(d) from Section 1 illustrates an example usage of the `detach` and `reattach` instructions. In Figure 2(d), 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` create a parallel task that consists of the `det` block. These instructions thus allow the execution of the `det` block to occur in parallel with the execution of the `cont` block. As a result, the call to `fib(n-1)` in the `det` block is allowed to execute in parallel with the call to `fib(n-2)` in the `cont` block.

The `sync` instruction provides dynamic control synchronization of parallel tasks.⁸ When a function invocation reaches a `sync` instruction, control is not allowed to pass the `sync` until all parallel tasks spawned by the function invocation and all descendants of those tasks have completed. In the Tapir CFG illustrated in Figure 2(d), for example, when program execution reaches the `sync` instruction in the `cont` block, control may not pass the `sync` until execution of the `det` block is complete. In general, a `sync` instruction is scoped to the function or parallel task that contains it. Execution of a parallel task cannot pass a `sync` instruction until all of its descendant parallel tasks have completed.

The `sync` instruction can be used and implemented in a flexible manner. As Figure 2(d) illustrates, unlike `reattach` instructions, `sync` instructions are not explicitly associated with `detach` instructions. In fact, `sync` instructions can be executed within conditionals to support dynamic task synchronization. A `sync` instruction j **syncs** a `detach` instruction i if i and j are executed within the same ancestor function or task and the CFG detached by i is not guaranteed to have completed when j executes. The `sync` instruction also permits a variety of runtime-system implementations. A processing unit, such as a worker thread managed by the runtime system, can simply wait when it encounters a `sync` instruction. Alternatively, a processing unit can suspend execution of the current function or parallel task at the `sync` and find other computation to execute.

⁸Unlike `detach` and `reattach`, which are terminators, the `sync` instruction, as exemplified in this article, is not. The latest version of Tapir/LLVM reverses this decision and treats `sync` as a terminator. The analysis and optimization passes in the compiler turn out to be much simpler because of this design decision.

Fig. 5. Two example Tapir CFGs for parallel loops. (a) The Tapir CFG for the parallel loop in Figure 1, using a format similar to the CFGs in Figure 2. (b) The Tapir CFG for a simple parallel implementation of $n \times n$ matrix multiplication that uses three nested loops, where the two outer loops are parallel. Edge labels are dropped from this subfigure for compactness.

Parallel loops in Tapir

Tapir supports a simple representation of parallel loops. Figure 5(a) illustrates Tapir’s default representation of the parallel loops from Figure 1. As Figure 5(a) shows, Tapir can represent a parallel loop in the CFG as an ordinary loop, where the **head** block repeatedly spawns the loop body—which is the single block **body** for the loops in Figure 1—and both the **head** block and loop body share the same continuation. Finally, the **exit** block executes a **sync** to dynamically synchronize all spawned executions of the loop body. 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, Section 4 describes how separate optimization passes in Tapir/LLVM make use of the serial-projection property to transform this parallel-loop representation into a divide-and-conquer form that exhibits good performance and scalability.

In general, the parallel task created by **detach** and **reattach** can encompass many basic blocks. Furthermore, the blocks within a parallel task may themselves contain **detach** and **reattach** instructions, allowing Tapir to express nested parallel tasks. Figure 5(b) illustrates an example Tapir CFG for a parallel matrix-multiplication subroutine that uses nested parallel loops. In this example, the nested parallel loops create nested parallel tasks. The **detach** instruction in basic block **ihead** and **reattach** instruction in block **jexit** delineate a parallel task composed of the basic blocks **jhead**, **kloop**, **kexit**, **jinc**, and **jexit**. Within that parallel task, the **detach** instruction in block **jhead** and the **reattach** instruction in block **kexit** delineate a nested parallel task composed of the basic blocks **kloop** and **kexit**. The **sync** instruction in block **exit** waits for the completion of all nested parallel tasks spawned by the execution of these two outer parallel loops.

Optionally, a **sync** instruction can be included inside of a task. For example, in Figure 5(b), a **sync** instruction can be added at the beginning of the basic block **jexit**. In this case, the **sync** in **jexit** would synchronize all of the spawned iterations of the inner parallel loop—that is, the loop consisting of the blocks **jhead**, **kloop**, **kexit**, and **jinc**—that execute within one iteration of the outer parallel loop. This nested **sync** would not synchronize across different iterations of the outer parallel loop.

Structural invariants of Tapir CFGs

The compiler ensures that the **detach** and **reattach** instructions in a CFG obey several structural properties. We say that 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 , which means that the detach edge of i **dominates** j .
- (3) Every path starting from the detached block of i reaches 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 also passes through a **reattach** instruction j' that reattaches i' .
- (5) Every cycle containing the detach edge of a **detach** instruction i passes through a **reattach** instruction that reattaches i .
- (6) The continuation block of j does not contain any ϕ instructions [3, Sec 6.2.4].

These invariants imply that a **detach** instruction i with detached block b and the set of **reattach** instructions R that reattach i define a single-entry sub-CFG $G' = (V', E', b)$, called the **detached sub-CFG**. The vertex set $V' \subset V$ contains all basic blocks on paths from b to a **reattach** instruction $j \in R$. The edge set $E' \subset E$ contains all edges (u, v) where $u, v \in V'$ and $(u, v) \in E$. The detached block b is the entry point of G' .

Informal semantics of Tapir

Semantically, a parallel task resembles a function whose CFG is a detached sub-CFG nested within another CFG, specifically, the CFG of the parent function or parent parallel task. The **detach** instruction thus resembles an ordinary function call, except that the **detach** allows the detached sub-CFG to execute in parallel with its continuation. The **reattach** instructions that terminate a parallel task are analogous to return statements.

Tapir instructions encode a serial projection of the program. For a **detach** instruction and the **reattach** instructions that reattach it, the serial projection they encode replaces the **detach** instruction with an unconditional branch to its detached block, and it replaces each **reattach** instruction with an unconditional branch to its continuation block. This serial projection implies that the parallel task delineated by the **detach** and **reattach** instructions can be executed to completion before the continuation block of the **detach** begins execution. The serial projection of a **sync** instruction elides the **sync** and performs a serial projection on all **detach** instructions that the **sync** instruction might sync. Tapir instructions thus make it easy for the compiler to analyze a serial projection of the program.

By encoding a serial projection, Tapir instructions order fork-join parallelism in the CFG. The CFG detached by a **detach** instruction is connected by reattach edges to the continuation block of the **detach**, even though the detached sub-CFG and continuation block can execute in parallel. For

example, in Figure 2(d), the reattach edge between **det** and **cont** breaks the symmetry between them. 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.

The dynamic execution of a Tapir program can be modeled as a tree of “parallel contexts,” which is analogous to the function-invocation tree for an ordinary serial program. Each **parallel context** in the tree models the dynamic execution of a function or a parallel task. 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 parallel context is destroyed and control returns to the parent’s context.

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 2(d) 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.

Stack-memory allocation and static single-assignment form

Tapir generalizes LLVM’s **alloca** instruction [70, section entitled, “**alloca**’ Instruction”] for stack-memory allocation to support parallel contexts. An **alloca** instruction in Tapir allocates memory local to the currently executing parallel context. This memory is automatically reclaimed when the parallel context terminates. The execution of a Tapir program organizes parallel-context-local memory into a **cactus stack** [76], which allows this memory to be accessed by descendant parallel contexts, but not by ancestors.

LLVM’s static single-assignment (SSA) form [3, Section 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** [3, 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 implement 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.

Extensions to Tapir

We describe two extensions to Tapir’s three instructions: sync regions and exceptional **detach** instructions. Strictly speaking, one can implement Tapir in a CFG-based compiler IR without implementing these extensions. But we found in practice—specifically, when implementing Tapir in LLVM IR—that these features facilitate effective compiler analysis and optimization.

Sync regions address an issue with inlining functions [3, p. 903] that contain Tapir instructions. Consider a simple function that spawns function `foo`, then calls function `bar`, and finally spawns function `baz`. Suppose further that `bar` spawns and synchronizes parallel tasks. If the call to `bar` is inlined, then any inlined `sync` instructions from `bar` will synchronize the execution of `foo` before the spawn of `baz`. As a result, inlining the call to `bar` can serialize the parallel control flow in its caller. Such serialization does not jeopardize the correctness of the parallel code, because Tapir always permits serialization of parallel tasks. But this serialization can affect parallel scalability and performance.

Sync regions support inlining of Tapir instructions without affecting parallel control flow. A **sync region** defines a tag shared by Tapir instructions. Each Tapir instruction takes a sync region as an additional operand, where **reattach** instructions take the same sync region as the **detach** instruction they reattach. A **sync** instruction with a given sync region only synchronizes tasks spawned by **detach** instructions with the same sync region. Although each sync region can be used by Tapir instructions within the same function or detached sub-CFG, different Tapir instructions within the same function or detached sub-CFG can be tagged with different sync regions. Sync regions can thus preserve parallel control flow after function inlining. In the previous example, by ensuring that the Tapir instructions in `bar` use a distinct sync region from those in its caller, any **sync** instruction inlined from `bar` will not mistakenly synchronize tasks spawned in the caller.

Exceptional **detach** instructions integrate Tapir with the explicit representation of exception-handling code in LLVM IR [69]. A function call in LLVM IR can be specified with two return destinations: a normal destination, where control flows if the call returns normally, and an **unwind** destination, where control flows if the call throws an exception. In Tapir, a parallel task might throw an exception to its parent parallel context. But the exception-handling code for both the parallel task and its parent appears in the same CFG, leading to ambiguity about which exception-handling code must be executed in the parallel task and which must be executed in its parent.

To model the event where a parallel task throws an exception to its parent, Tapir supports an **exceptional detach instruction**, which takes an **unwind continuation** as an additional operand. The unwind continuation specifies the exception-handling code in the parent parallel context to handle an exception thrown by the task. A **reattach** instruction in the parallel task can direct control flow to the unwind continuation of the **detach** it reattaches, to model throwing an exception from the parallel task.

Different parallel runtime systems can specify different code to run when a parallel task throws an exception. Different variants of Cilk specify how exceptions thrown from parallel tasks are handled [20, 63], while the OpenMP runtime system requires each parallel task to handle any exceptions it may throw [82, p. 49]. Tapir's exceptional **detach** instruction supports a variety of runtime specifications by differentiating exception-handling code in a parallel task from that in its parent context.

3 ANALYSIS PASSES

This section describes how LLVM's analysis passes can be adapted to operate on Tapir programs. We first discuss the ramifications of the serial-projection property and constraints on how Tapir programs can be safely transformed. Implementing these constraints on LLVM optimization passes primarily involves adapting standard compiler analyses—specifically alias analysis [3, Ch. 12], dominator analysis [3, Ch. 9], and data-flow analysis [3, Ch. 9]—to accommodate Tapir's instructions. We describe how each of these analyses was minimally modified to support Tapir.

Leveraging the serial-projection property

To analyze a parallel program, Tapir takes advantage of the serial-projection property and, in particular, what the property implies about the forward-progress requirements for a semantically valid execution of parallel tasks. The serial-projection property guarantees that parallel tasks can be executed correctly according to the serial projection. Hence any task in the program can be executed to completion before any forward progress is made on instructions that (i) are logically in parallel with that task, and (ii) would execute after that task in the serial-projection order. Parallel programs supported by task-parallel programming platforms typically satisfy the serial-projection property. In fact, many such programs satisfy a stronger property, namely, that the execution of a parallel task does not require forward progress in the execution of any logically parallel instructions.

The serial-projection property does not preclude other semantically valid executions of parallel tasks, nor does it prohibit determinacy races. If the program contains a determinacy race, then other valid executions of the program can produce behaviors that differ from the behavior of a serial projection. The range of possible program behaviors can depend on the memory consistency model [16, 86]. Hence, the serial-projection property does not require a particular memory consistency model, such as sequential consistency [57].

Not all parallel programs satisfy the serial-projection property. Some programs that employ arbitrary concurrency mechanisms between parallel tasks require all participating tasks to make forward progress in any semantically valid execution. To optimize these programs, compilers must reason about these concurrency mechanisms, which can be intractable in general. In addition, common optimizations on parallel code, such as “sequentialization,” may be invalid on these programs [109].

Tapir takes advantage of the serial-projection property to facilitate compiler analysis and optimization of parallel code. Section 2 described how Tapir uses the serial-projection property to order the program’s parallel control flow. In addition, the serial-projection property allows the compiler to understand the semantics of a parallel task without deeply analyzing instructions that might execute logically in parallel with that task. Standard analysis passes can instead analyze a function containing fork-join parallelism based on the control flow expressed in the structure of the CFG containing Tapir instructions, as this section now describes.

Constraints on transformations

To be correct, a code transformation on a Tapir program must produce a program that satisfies the serial-projection property 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 raced-on memory location. If a Tapir program contains no determinacy races, then it is guaranteed to exhibit the same semantics as its serial projection. For a Tapir program that does contain a determinacy race, Tapir imposes no requirements on the possible outcomes of the race except for the serial-projection property. Tapir instead composes with the memory consistency model defined by the compiler, such as LLVM’s memory model [70, section entitled, “Memory Model for Concurrent Operations”], allowing that memory model to define the possible outcomes of concurrent accesses.

To avoid introducing new behaviors, code transformations must not create determinacy races, although they can eliminate determinacy races. Many existing serial optimizations can be made 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.

Alias analysis

LLVM uses alias analysis [3, 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 Tapir instructions as if they access memory. For example, consider a load or a store instruction k . 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 parallel task spawned 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 parallel task spawned by i . To handle Case 1, i is treated as if it were a function call that accesses all memory locations accessed in the parallel task. 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 memory fence that prevents movement of memory instructions. With these modifications, existing rules in LLVM that restrict reordering of loads and stores properly restrict memory reordering around Tapir instructions.

Dominator analysis

Optimization passes determine what values are available to an instruction in part by using dominator analysis [3, 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 , then an optimization pass might assume that the value produced by i is always available when j executes.

Tapir's ordered representation of parallelism 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 projection 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 [3, 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 CFGs, data-flow analyses must be adapted specifically to handle reattach edges. Because Tapir 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.

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. Let E_R denote the set of reattach edges in E . 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 [71] to optimize parallel control flow with little to no modification. It also enables new optimization passes that specifically target Tapir's fork-join constructs. This section describes how Tapir enables these existing and novel optimization passes.

Figure 6 presents a breakdown of the lines of code changed in LLVM's optimization passes to accommodate Tapir instructions. As the figure shows, most optimization passes required no modification to accommodate Tapir. For the optimization passes that were modified, the figure shows that the necessary modifications are minimal in size.

The remainder of this section discusses specific optimizations. Common-subexpression elimination [77, Section 12.2] serves as one example of the ~ 60 optimization passes that require no changes and "just work" with the additional Tapir instructions. Loop-invariant code motion [77, Section 13.2], and tail-recursion elimination [77, Section 15.1] are two passes that require modification to work effectively on parallel code. Parallel-loop scheduling serves as an example of a new optimization pass. We discuss two new minor passes that optimize Tapir CFGs as well.

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 $(\text{low} + \text{high})/2$ in Figure 7(a) is computed in lines 41 and 42. Tapir/LLVM performs CSE on this code, producing code equivalent to that in Figure 7(b). Existing mainstream compilers that support fork-join parallelism do not eliminate this common subexpression, however, and they compute $(\text{low} + \text{high})/2$ 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.

Optimization	LLVM 6.0	Tapir/LLVM
Shared utilities	31,497	748
CFG simplification	6,395	236
Function inlining	3,844	94
Global value numbering	5,767	77
Instruction combining	30,763	24
Jump threading	2,704	16
Loop-invariant-code motion	1,626	28
Loop rotation	746	8
Loop simplification	942	6
Loop unrolling	3,783	11
Loop vectorization	8,943	10
Scalar replacement of aggregates	4,586	41
Sparse conditional constant propagation	2,106	18
Tail-recursion elimination	919	106
The other ~60 optimizations	88,598	0
Total	193,219	1,423

Fig. 6. Breakdown of the lines of code added, modified, or deleted in the Tapir/LLVM prototype to adapt the optimization passes in LLVM 6.0 to handle Tapir. Optimizations are identified using LLVM terminology. The row “Shared utilities” accounts for utility functions shared among all of LLVM’s optimization passes.

a	b
<pre> 38 void search(int low, int high) { 39 if (low == high) search_base(low); 40 else { 41 cilk_spawn search(low, (low+high)/2); 42 search((low+high)/2 + 1, high); 43 cilk_sync; 44 } } </pre>	<pre> 45 void search(int low, int high) { 46 if (low == high) search_base(low); 47 else { 48 int mid = (low+high)/2; 49 cilk_spawn search(low, mid); 50 search(mid + 1, high); 51 cilk_sync; 52 } } </pre>

Fig. 7. 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 41 and 42 of the original version is computed only once and stored in the variable `mid` in line 48 of the optimized version.

Loop-invariant code motion

The loop-invariant code motion (LICM) optimization [77, Section 13.2] aims to move **loop-invariant** computations—which run on every iteration of the loop, compute the same value on every iteration, and have no side effects—out of loop bodies. LICM is responsible, for example, for moving the call to `norm` in the parallel loop in Figure 1(a) 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. To find computations to move outside of the loop, the LICM pass looks for computations in the loop body that it can prove are loop-invariant. To find such computations, LICM looks for instructions in the loop body that dominate the exiting block of the loop, for example, block `inc` in Figure 5(a). (The block labeled `exit` is the exit of the function, not the loop exit.) This analysis fails for parallel loops

in Tapir, however. Consider the CFG illustrated in Figure 5(a), which models the parallel loop in Figure 1(a). As Figure 5(a) shows, the continue edge that bypasses the loop body ensures that no blocks in the loop body ever dominate the exit block as they would in the serial projection of the loop.

Tapir/LLVM modifies LLVM's LICM pass to handle a parallel loop by analyzing the serial projection 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(a), this modification is implemented by finding blocks in the loop body that dominate the predecessors of the loop exit. The modification required changing 53 lines in LLVM: 28 lines of LLVM's LICM pass and 25 lines of shared utility functions.

Tail-recursion elimination

Tail-recursion elimination (TRE) [77, Section 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, as well as synchronization overheads on spawning functions.

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. The TRE optimization can affect the parallel control flow of a Tapir program by changing the set of spawned tasks each `sync` instruction syncs, but the optimization does not introduce new determinacy races and it preserves the serial-projection property. This modification to LLVM's TRE pass required changing 106 lines.

To see why these `sync` instructions can be safely ignored, consider Figure 8, which illustrates how Tapir/LLVM's TRE pass operates on the `pqsort` function, a parallel version of Hoare's quick-sort algorithm [40]. The original tail-recursive code is shown in Figure 8(a). Figure 8(b) illustrates the result of simply inlining the tail-recursive call. For the inlined code, all `return` statements are replaced with branches to the `join` label. Because there is a `cilk_sync` at the start of `join`, the `cilk_sync` on line 75 can be eliminated. The resulting code contains a tail-recursive call on line 74, on which this inlining and `cilk_sync`-elimination process can be repeated. Because this process can be applied 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, as Figure 8(c) illustrates.

Parallel-loop scheduling and lowering

As discussed in Section 2, Tapir effectively represents a parallel loop as a serial loop where each iteration spawns the loop body. 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 not be efficient. 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 Reference [72, Section 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 in 2,600 lines as part of Tapir/LLVM's 5,800-line lowering pass, which translates `detach`, `reattach`, and `sync` instructions into appropriate Cilk Plus runtime calls [44]. Tapir/LLVM contains two implementations of this parallel-loop-scheduling pass. One

Fig. 8. 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 58 has been replaced by one round of inlining. (c) A version `pqsort` where tail-recursion elimination has removed the recursive tail call on line 58.

implementation uses the Cilk Plus runtime calls for `cilk_for` loops [44, Section 10.7] to schedule parallel loops. The other runs as an independent pass and directly emits an appropriate Tapir function that performs parallel recursive divide-and-conquer. Although the second implementation has advantages over the first, we chose to use the first implementation to evaluate Tapir/LLVM in this article, to perform fair performance comparisons between Tapir/LLVM and compilers that lower parallel constructs in their front end.

Additional parallel optimization passes

Tapir/LLVM implements two minor parallel optimization passes: puny-task elimination and redundant-synchronization elimination. **Puny-task elimination** replaces detached sub-CFGs that perform little or no work with their serial projection. If the runtime overhead of creating a parallel task outweighs the work in the task, then the task might as well be run serially. **Redundant-synchronization elimination** identifies and eliminates `sync` instructions that cannot sync any `detach` instructions. Both of these optimization passes are implemented in 236 lines of code by augmenting LLVM's pass for simplifying CFGs.

5 AUXILIARY SOFTWARE

This section describes auxiliary software that we developed to exercise, test, and debug Tapir/LLVM. Although this work focuses on the middle end of the compiler, we implemented a front end, called PClang, that translates Cilk Plus codes and emits Tapir instructions. In addition, we developed compiler instrumentation that allows the compiler to interface to a provably good race detector to verify the correctness of the Tapir/LLVM implementation. Because of its provable

guarantees, this race detector allows us to run regression tests to either certify that Tapir/LLVM introduces no determinacy races or pinpoint racing instructions if they exist.

To create the PClang front end, we modified of the Clang C/C++ front end for LLVM to translate Cilk Plus codes to Tapir. PClang is compatible with Cilk Plus, minus its vector notation, and it supports existing Cilk Plus applications and libraries. PClang also supports modern programming technologies, including debugging symbols and Google’s Sanitizer tool suite, as well as modern C++ features, including C++ exceptions and the C++17 standard.

We augmented Tapir/LLVM in several ways to test the correctness of the implementation. We modified LLVM’s internal verification pass to check that Tapir’s structural invariants are also maintained. 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 [25], is guaranteed to find a determinacy race if and only if one exists in the program execution. The verification pass and determinacy-race detector helped us locate and fix bugs in Tapir/LLVM, both within the modifications for Tapir and within the underlying LLVM codebase. Tapir/LLVM now passes all tests in LLVM’s regression test suites and correctly compiles a large suite of parallel test programs. We have also added a suite of regression tests for Tapir instructions and code transformations to LLVM’s regression-test suite.

We extended the CSI framework for compiler-inserted instrumentation [95] to instrument Tapir instructions. This framework has allowed rapid development of efficient dynamic-analysis tools for Tapir, such as scalability analyzers [38] and scalability profilers [96], to further assess the correctness and optimization capabilities of Tapir/LLVM.

6 EVALUATION

To evaluate the effectiveness of the Tapir approach, we evaluated Tapir/LLVM on 35 application benchmarks implemented using dialects of Cilk. On 22 benchmarks, Tapir/LLVM outperforms GCC version 7.3.1 in both 1-core and 18-core executions. On 2 benchmarks, GCC fails to compile correct code. On the remaining 11 benchmarks, GCC outperforms Tapir/LLVM, but on these benchmarks GCC also outperforms LLVM on serial code by similar margins. But these comparisons, which show that Tapir/LLVM is “real,” do not substitute for a scientific evaluation of the Tapir approach, because these compilers differ in more ways than just where they lower parallel constructs. The experiments in this section attempt to validate the Tapir approach with an apples-to-apples comparison.

Experimental setup

To evaluate the Tapir approach, we implemented an additional compiler, called “Reference,” to serve as a scientific control to minimize the effects of other implementation details in the compiler. The design and implementation of Reference are as close to identical to Tapir/LLVM as we could muster, except for where lowering occurs. Figure 9 illustrates the compilation pipelines for the standard Clang/LLVM serial compiler, 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 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. The Lower pass translates all the Tapir-specific constructs to LLVM IR. The second `-O3` pass performs optimizations on the LLVM IR. Finally, the CodeGen back-end lowers LLVM IR to machine code.

Fig. 9. 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 third pipeline, called Reference, models how mainstream compilers work today, where parallel constructs are transformed into runtime calls before any optimization can take place. As Figure 9 illustrates, the only difference between Reference and Tapir/LLVM is that the RefPClang front end for Reference immediately lowers parallel constructs into LLVM IR, as mainstream compilers do today. We ensured that the RefPClang and the Lower pass produce the exactly same implementation of the Cilk Plus ABI specification [44]. Although Reference lowers the parallel constructs early, two iterations of **-O3** optimization and an invocation of the Lower pass are included to ensure that the Tapir/LLVM gains no advantage from the additional passes. Serial code passes through both pipelines identically.

We found that running the extra passes of **-O3** and **Lower** on ordinary LLVM IR can have a minor effect on performance in practice. We tested a variation of the Clang/LLVM pipeline that incorporates the extra passes of **-O3** and **Lower** to match the structure of the Tapir/LLVM and Reference pipelines. Across all 35 benchmarks tested, the executables produced by Clang/LLVM varied in performance by up to 1%, based on whether or not these additional **Lower** and second **-O3** passes were included in the pipeline. For the evaluation of Tapir/LLVM, we chose to include the additional passes of **-O3** and **Lower** in the Clang/LLVM pipeline, to ensure that all three pipelines perform the same sequence of passes and that no pipeline gains an unfair advantage from running more or fewer passes.

Benchmarking

To benchmark the compiler pipelines, we assembled a collection of 35 benchmark programs taken from the MIT Cilk benchmark suite [29], Intel Cilk code samples [49], and the CMU

<i>Suite</i>	<i>Benchmark</i>	<i>Description</i>
Cilk	cholesky	Cholesky decomposition
	cilksort	Parallel merge sort
	fft	Fast Fourier transform
	heat	Heat diffusion stencil
	lu	LU decomposition
	matmul	Square matrix multiplication
	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
	bsm	Black-Scholes model
	dct	Discrete cosine transform
PBBS	detbfs	BFS, deterministic algorithm
	ndbfs	BFS, nondeterministic algorithm
	sampsort	Parallel sample sort
	chull	Convex hull
	idelaunay	Incremental Delaunay triangulation
	irefine	Incremental Delaunay refinement
	dict	Parallel dictionary
	radixsort	Parallel radix sort
	imis	MIS, incremental algorithm
	ndmis	MIS, nondeterministic algorithm
	imatch	Maximal matching, incremental algorithm
	ndmatch	Maximal matching, nondeterministic algorithm
	pkruskal	Parallel Kruskal's algorithm
	pck	Parallel n -body simulation
	octtree	Nearest-neighbors search
	kdtree	Sparse-triangle intersection
	remdup	Remove duplicates
	ist	Spanning tree, incremental algorithm
	ndst	Spanning tree, nondeterministic algorithm
	pks	Parallel suffix-array construction, Karkkainen and Sanders algorithm
	prange	Parallel suffix-array construction, range algorithm

Fig. 10. Descriptions of the 35 benchmarks used to evaluate Tapir/LLVM. These benchmarks were taken from the MIT Cilk benchmark suite [29], Intel Cilk Plus example programs [49], and the CMU Problem-based Benchmark Suite (PBBS) [100]. “MIS” denotes the computation of a maximal independent set of a graph. “BFS” denotes the breadth-first search of a graph.

Problem-based Benchmark Suite [100]. Figure 10 describes the suite of benchmarks tested. All benchmarks were compiled with `-O3` optimization. In addition, the benchmarks from the Problem-based Benchmark Suite, which are written in C++, were compiled with `-ffast-math` and C++ exceptions enabled.

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 projection [29] of each benchmark through the same sequence of middle-end passes. Each running time is the minimum of 5 runs on an Amazon AWS c4.8xlarge spot instance, which is a

		cholesky	fft	nqueens	qsort	rectmul	strassen	lu
T_S		2.545	8.256	2.484	4.356	8.841	9.075	11.867
T_1	Reference	5.055	10.192	2.652	6.725	9.205	9.087	12.046
	Tapir/LLVM	4.084	9.536	2.680	5.861	8.998	9.082	12.019
T_{18}	Reference	0.396	0.660	0.162	0.623	0.584	1.035	0.762
	Tapir/LLVM	0.341	0.610	0.162	0.579	0.557	1.022	0.770
$\frac{T_S}{T_1}$	Reference	0.503	0.810	0.937	0.648	0.960	0.999	0.985
$\frac{T_1}{T_{18}}$	Tapir/LLVM	0.623	0.866	0.927	0.743	0.982	0.999	0.987
$\frac{T_S}{T_{18}}$	Reference	6.426	12.509	15.333	6.992	15.139	8.768	15.573
$\frac{T_{18}}{T_{18}}$	Tapir/LLVM	7.463	13.534	15.333	7.523	15.873	8.880	15.412
		heat	cilkSORT	matmul	avgfilter	mandel	bsm	dct
T_S		6.958	7.966	5.517	0.359	22.429	101.073	4.745
T_1	Reference	7.098	8.126	5.512	0.342	25.052	103.506	5.072
	Tapir/LLVM	7.005	8.137	5.475	0.340	22.261	100.394	5.068
T_{18}	Reference	1.380	0.552	0.347	0.161	1.582	6.263	0.409
	Tapir/LLVM	1.385	0.545	0.341	0.160	1.387	6.040	0.413
$\frac{T_S}{T_1}$	Reference	0.980	0.980	1.001	1.050	0.895	0.976	0.936
$\frac{T_1}{T_{18}}$	Tapir/LLVM	0.993	0.979	1.008	1.056	1.008	1.007	0.936
$\frac{T_S}{T_{18}}$	Reference	5.042	14.431	15.899	2.230	14.178	16.138	11.601
$\frac{T_{18}}{T_{18}}$	Tapir/LLVM	5.024	14.617	16.179	2.244	16.171	16.734	11.489

Fig. 11. Comparison between executables compiled from the Cilk-5 and Intel benchmark suites using Reference and Tapir/LLVM. Each column refers to a different parallel benchmark described in Figure 10. Rows are labeled “Reference” or “Tapir/LLVM” to identify which compiler pipeline was used to compile the executable. Each measured running time is the minimum over five executions, measured in seconds. The row labeled T_S gives the running time of the executable compiled from the serial projection of each benchmark. This row is not labeled “Reference” or “Tapir/LLVM,” because both pipelines are identical for these serial codes. 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 T_S and T_1 rows. The pair of rows labeled T_S/T_{18} gives the parallel speedup of each compiled executable on 18 cores, derived from the T_S and T_{18} rows.

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, and each core has a 32 KiB private L1-data-cache and a 256 KiB private L2-cache. The system was “quiesced” to permit careful performance measurements by turning off Turbo Boost, address-space layout randomization (ASLR), dynamic voltage and frequency scaling (DVFS), hyperthreading, extraneous interrupts, and so on.

Overall performance

Figures 11 and 12 present the results of the benchmarking tests. For the T_S row, Reference and Tapir/LLVM produce identical executables when compiling the serial projection of a benchmark. The T_1 rows show that Tapir/LLVM produces executables with lower work than Reference on 30 of the benchmarks. For 12 of the benchmarks, moreover, Tapir/LLVM produces executables that are more than 5% more work efficient than executables produced by Reference. The T_S/T_1 rows elaborate on the results in the T_1 rows to show that Tapir/LLVM produces executables with nearly optimal work efficiency (over 99%) on 23 of the benchmarks, whereas Reference does so on only 11. Although work efficiency over 100% is not possible in theory, the heuristics that drive compiler

		detbfs	ndbfs	samsort	chull	idelaunay	irefine	dict
T_S		34.466	22.766	17.529	10.403	59.700	68.300	2.680
T_1	Reference	36.566	25.233	17.564	10.359	60.099	68.700	2.942
	Tapir/LLVM	34.600	22.766	17.562	10.325	59.350	68.350	2.828
T_{18}	Reference	2.296	1.633	1.126	0.961	4.115	4.765	0.189
	Tapir/LLVM	2.203	1.510	1.123	0.973	4.095	4.740	0.182
$\frac{T_S}{T_1}$	Reference	0.943	0.902	0.998	1.004	0.993	0.994	0.911
$\frac{T_1}{T_{18}}$	Tapir/LLVM	0.996	1.000	0.998	1.008	1.006	0.999	0.948
$\frac{T_S}{T_{18}}$	Reference	15.011	13.941	15.568	10.825	14.508	14.333	14.180
$\frac{T_{18}}{T_{18}}$	Tapir/LLVM	15.645	15.077	15.609	10.692	14.579	14.409	14.725
		radixsort	imis	ndmis	imatch	ndmatch	pkruskal	pck
T_S		2.022	3.596	5.476	12.833	7.659	51.366	54.066
T_1	Reference	2.032	4.076	6.409	14.233	8.380	57.933	53.833
	Tapir/LLVM	2.020	3.616	5.510	12.866	7.726	55.166	53.533
T_{18}	Reference	0.193	0.328	0.413	0.935	0.562	3.910	3.470
	Tapir/LLVM	0.192	0.317	0.383	0.872	0.529	4.096	3.470
$\frac{T_S}{T_1}$	Reference	0.995	0.882	0.854	0.902	0.914	0.887	1.004
$\frac{T_1}{T_{18}}$	Tapir/LLVM	1.001	0.994	0.994	0.997	0.991	0.931	1.010
$\frac{T_S}{T_{18}}$	Reference	10.477	10.963	13.259	13.725	13.628	13.137	15.581
$\frac{T_{18}}{T_{18}}$	Tapir/LLVM	10.531	11.344	14.298	14.717	14.478	12.541	15.581
		octtree	kdtree	remdup	ist	ndst	pks	prange
T_S		15.210	4.970	3.664	27.466	31.033	31.633	26.099
T_1	Reference	15.976	4.956	4.126	28.666	31.200	32.133	29.049
	Tapir/LLVM	15.986	4.976	3.670	27.233	31.200	31.766	28.800
T_{18}	Reference	1.082	0.324	0.285	1.956	9.570	2.460	1.918
	Tapir/LLVM	1.137	0.324	0.248	1.880	9.243	2.448	1.915
$\frac{T_S}{T_1}$	Reference	0.952	1.003	0.888	0.958	0.995	0.984	0.898
$\frac{T_1}{T_{18}}$	Tapir/LLVM	0.951	0.999	0.998	1.009	0.995	0.996	0.906
$\frac{T_S}{T_{18}}$	Reference	14.057	15.340	12.856	14.042	3.243	12.859	13.607
$\frac{T_{18}}{T_{18}}$	Tapir/LLVM	13.377	15.340	14.774	14.610	3.357	12.922	13.629

Fig. 12. Comparison between executables compiled from the Problem-based Benchmark Suite [100] using Reference and using Tapir/LLVM. Results were derived and formatted similarly to Figure 11.

optimizations and quirky behaviors of multicore architecture, such as caching and memory effects, can lead to such work efficiencies in practice. The T_{18} and T_S/T_{18} rows show that Tapir/LLVM generally produces executables with similar or better parallel speedups than those produced by Reference.

We found 5 benchmarks where Tapir/LLVM did not produce more work-efficient executables than Reference, all of which demonstrate less than a 1% difference between their work relative to Tapir/LLVM or Reference. The biggest slowdown created from Tapir/LLVM's compilation occurs on nqueens. For this benchmark, heuristics in LLVM's optimization passes caused Tapir/LLVM, but not Reference, to vectorize a serial loop in a function that contains `cilk_spawn` statements. It turns out that vectorizing that serial loop leads to a small performance degradation on the benchmark. For the other four benchmarks, the performance difference seemed to occur due to measurement noise. For example, we found that changing the operating-system shell used to run these benchmarks could invert the work-efficiency comparisons.

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 [4, 90]. Joisha et al. [51] 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 recursive-fork-join parallel programs that support the serial-projection property. 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 optimizations 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. [7, 8] 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 produces a program that satisfies the serial-projection property and does not introduce determinacy races. Compared with general concurrency mechanisms, well-structured parallelism seems to offer a less onerous path to performance.

Khalidi et al. [55] 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 [54] 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, Tapir's strong assumption of a fork-join programming model that supports the serial-projection property and compiles to a flexible multicore architecture seems to provide both performance and simplicity. Applying Tapir to other parallel-computing architectures, such as distributed systems, remains an open research question.

In contrast with much of the work just referenced, Chatarasi et al. [19] focus, as Tapir does, on fork-join programs that support the serial-projection property. 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

Significant interest has emerged in applying and extending Tapir to a broader range of parallel programming models, for example, to handle non-fork-join task-parallel linguistic constructs, to target specialized hardware architectures such as GPUs, and to accommodate distributed-memory programs. One key concern is that not all of these parallel models satisfy the serial-projection property that Tapir leverages, which raises the question, how can one tell in general whether a parallel program satisfies the serial-projection property? This question is simple to answer for programs that only use task-parallel constructs that can be shown to support a serial projection.

But a program that uses arbitrary concurrency mechanisms might or might not support the serial-projection property, depending on how those mechanisms are used.

To address this problem, we advocate, as many have (see, for example, Reference [33]), separating the concerns of parallelism from those of concurrency. From our point of view, parallelism is concerned with the physically simultaneous execution of instructions, specifically, to speed up the execution of a program. In contrast, concurrency is concerned with interactions among separate agents, such as threads. Although these concepts are related—after all, concurrency mechanisms are the building blocks for synchronization in most parallel-programming runtime systems—they are also distinct. Concurrency can exist even between distinct processes running on a single-core machine, whose instructions cannot execute simultaneously. Moreover, many task-parallel programming platforms support deterministic parallel programming [9–11, 15, 21, 22, 24, 25, 32, 37, 42, 81, 83, 105, 112], in which the programmer observes no nondeterministic behavior due to concurrency.

To separate these concerns in the context of multicore programming, we draw a distinction between task parallelism and threading. Threading supports arbitrary uses of concurrency mechanisms. As a result, a program with multiple threads might not satisfy the serial-projection property: two interacting threads might each require that the other make forward progress to properly implement a concurrency protocol. But this generality comes at a cost. Threads are hard to reason about [41, 59], because, in general, the programmer must consider all possible interleavings of operations. Moreover, threads are not generally amenable to compiler optimizations (see, for example, Reference [109]). In contrast, task parallelism generally supports the serial-projection property, and in particular, allows parallel tasks to execute with limited forward-progress guarantees on logically parallel instructions, as described in Section 3. Concurrency mechanisms cannot be used in parallel tasks in a manner that violates the serial-projection property: Doing so results in incorrect code. As a result, task parallelism simplifies testing, debugging, and formal verification, and as our work on Tapir demonstrates, task-parallel programs are amenable to both standard compiler optimizations and optimizations that modify parallel control flow.

We believe the parallel-programming community should strive to develop parallel-programming models that distinguish parallelism from concurrency. These models might still support threads for concurrent programming while still providing the benefits of task parallelism for programs that do not require arbitrary concurrency mechanisms. To support this distinction, one appealing approach aims to allow individual threads to be parallelized using task parallelism without compromising their ability to act as threads with respect to the other threads in the system. This approach has been proposed in various settings (see, for example, References [35, 36, 89]). Indeed, many distributed applications adopt a similar model by employing message-passing to coordinate a computation across message-passing nodes of a computing cluster—which are analogous to threads—and task parallelism to parallelize the computation on a single node [87]. These distributed applications demonstrate the potential performance benefits of this approach [87].

Two open research questions specific to Tapir emerge from this perspective on parallel-programming models.

First, how should Tapir handle parallel constructs that do not satisfy the serial-projection property? Even if a program employs task parallelism satisfying the serial-projection property to parallelize an individual thread, when compiling such a program, Tapir must correctly compile concurrency mechanisms amidst task-parallel constructs. Indeed, we have used Tapir/LLVM to compile and optimize a version of the distributed miniFE program [39] that uses a mixture of Cilk and MPI [27]. It seems that Tapir/LLVM handles such programs correctly, because the underlying mainstream LLVM compiler has already been designed to correctly compile serial-program

fragments that contain calls to Pthread library routines [43], MPI routines, and other concurrency mechanisms. It remains an open problem to formalize this intuition.

Second, how can Tapir be extended to model task-parallel programming constructs beyond recursive fork-join? Many task-parallel programming environments support non-fork-join models of parallelism, including parallel reductions [28, 47, 61, 73, 82, 88, 98, 111], pipeline parallelism [23, 60, 78], and task graphs [2, 88]. We anticipate seeing more such task-parallel constructs emerge in the future, because these constructs expand the set of programmers who can employ well-structured task parallelism, rather than arbitrary concurrency, to parallelize applications. Although we have verified that Tapir/LLVM correctly compiles some of these programs—specifically, programs that use parallel reductions and programs that use concurrency mechanisms that satisfy the serial-projection property [12, 99]—none of these constructs are directly modeled by Tapir’s instructions. As a result, the compiler misses optimization opportunities involving these parallel constructs. One direction of future research seeks to extend Tapir to support effective compiler analysis and optimization on these non-fork-join task-parallel constructs.

We believe these two questions should be addressed, at least in part, by developing a formal semantics for Tapir. A formal semantics would give us a mathematical foundation for understanding parallel-programming models and proving the validity of parallel-program transformations. We have begun an exploration of this question [94, Ch. 10] by formalizing the relationship between a Tapir program and its serial projection. But much work remains to be done, and we view this research direction as a high priority, because a technology as promising as Tapir should stand on a strong theoretical foundation.

9 EPILOGUE

Since the conference publication [97] of an early version of this work, we have continued to enhance the capabilities of Tapir/LLVM and improve its robustness. Tapir/LLVM, its PClang front end, its determinacy-race detector, and its suite of tools have been battle-tested by hundreds of undergraduate and graduate students over three years at MIT, Carnegie Mellon University, and Washington University in St. Louis. We are continuing to make Tapir/LLVM more accessible, such as by making it easier to download and install. We are also preparing a new open-source implementation of the Cilk concurrency platform, called *Open Cilk* (see <http://cilkhub.org/>), based on the Tapir/LLVM compiler.

ACKNOWLEDGMENTS

Many thanks to our master bug-finder Tim Kaler of MIT, who also helped us with cloud technology for evaluating Tapir/LLVM. Thanks to I-Ting Angelina Lee of Washington University in St. Louis and George Stelle of Los Alamos National Lab for their help finding bugs in Tapir/LLVM. Thanks to Johannes Doerfert, Simon Moll, and Sebastian Hack of Saarland University, Germany, for their feedback and insights on the design and semantics of Tapir. Thanks to Hal Finkel of Argonne National Lab for discussions about the serial-projection property. Thanks to Jiahao Li and Dougie Kogut of MIT for discussions related to extensions to Tapir. Thanks to Julian Shun of MIT for his assistance with application benchmarks. Thanks to Larry Hardesty of the MIT News Office for asking questions that helped us to simplify Figure 3. Shahin Kamali, formerly of MIT and now of University of Manitoba, and Bradley Kuszmaul, formerly of MIT and now of Oracle, were involved in many helpful discussions. Thanks to Genghis Chau, Louis Golowich, Phil Sun, and Helen Xu for their work adapting various parallel tools to operate with Tapir/LLVM. We are grateful to the students and staff of the Fall 2016 and 2017 offerings of MIT class 6.172/6.871 *Performance Evaluation of Software Systems* for their patience in using the Tapir/LLVM compiler throughout

the semester and for reporting bugs. We thank the PPoPP 2017 and TOPC reviewers for their feedback.

REFERENCES

- [1] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. 2007. May-happen-in-parallel analysis of X10 programs. In *Proceedings of PPoPP*. 183–193.
- [2] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. 2010. Executing task graphs using work-stealing. In *Proceedings of IPDPS*. 1–12.
- [3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.
- [4] Jonathan Aldrich, Craig Chambers, EminGun Sirer, and Susan Eggers. 1999. Static analyses for eliminating unnecessary synchronization from Java programs. In *Static Analysis*, Agostino Cortesi and Gilberto Filé (Eds.). Lecture Notes in Computer Science, Vol. 1694. 19–38.
- [5] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of SPAA*. 119–129.
- [6] E. Ayguade, N. Copt, A. Duran, J. Hoeflinger, Yuan Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and Guansong Zhang. 2009. The design of OpenMP tasks. *IEEE Trans. Parallel Distrib. Syst.* 20, 3 (2009), 404–418.
- [7] Rajkishore Barik and Vivek Sarkar. 2009. Interprocedural load elimination for dynamic optimization of parallel programs. In *Proceedings of PACT*. 41–52.
- [8] Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. 2013. Interprocedural strength reduction of critical sections in explicitly-parallel programs. In *Proceedings of PACT*. 29–40.
- [9] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of ASPLOS*.
- [10] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009. Grace: Safe multithreaded programming for C/C++. In *Proceedings of OOPSLA*. 81–96.
- [11] Guy E. Blelloch. 1996. Programming parallel algorithms. *Commun. ACM* 39, 3 (Mar. 1996).
- [12] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *Proceedings of PPoPP*. 181–192.
- [13] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An efficient multithreaded runtime system. *J. Parallel Distrib. Comput.* 37, 1 (1996), 55–69.
- [14] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (1999), 720–748.
- [15] Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. 2009. Parallel programming must be deterministic by default. In *Proceedings of HotPar*.
- [16] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *Proceedings of PLDI*. 68–78.
- [17] Luca Cardelli. 1997. Program fragments, linking, and modularization. In *Proceedings of POPL*. 266–277.
- [18] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: The new adventures of old X10. In *Proceedings of PPPJ*. 51–61.
- [19] Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar. 2015. Polyhedral optimizations of explicitly parallel programs. In *Proceedings of PACT*. 213–226.
- [20] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. 2008. Programming with exceptions in JCilk. *Sci. Comput. Program.* 63, 2 (Dec. 2008), 147–171.
- [21] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic shared memory multiprocessor. In *Proceedings of ASPLOS*. 85–96.
- [22] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. 2011. RCDC: A relaxed consistency deterministic computer. In *Proceedings of ASPLOS*. 67–78.
- [23] Wei Du, Renato Ferreira, and Gagan Agrawal. 2003. Compiler support for exploiting coarse-grained pipelined parallelism. In *Proceedings of SC*. 8–21.
- [24] Mingdong Feng and Charles E. Leiserson. 1997. Efficient detection of determinacy races in Cilk programs. In *Proceedings of SPAA*.
- [25] Mingdong Feng and Charles E. Leiserson. 1999. Efficient detection of determinacy races in Cilk programs. *Theory Comput. Syst.* 32, 3 (1999), 301–326.
- [26] Jeremy T. Fineman and Charles E. Leiserson. 2011. Race detectors for Cilk and Cilk++ programs. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). 1706–1719.
- [27] The MPI Forum. 1993. MPI: A message passing interface. In *Proceedings of Supercomputing*. 878–883.

- [28] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. 2009. Reducers and other Cilk++ hyperobjects. In *Proceedings of SPAA*. 79–90.
- [29] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of PLDI*. 212–223.
- [30] GCC Team. 2014. GCC 4.9 Release Series Changes, New Features, and Fixes. Retrieved from <https://gcc.gnu.org/gcc-4.9/changes.html>.
- [31] GCC Team. 2015. GOMP—An OpenMP Implementation for GCC. Retrieved from <https://gcc.gnu.org/projects/gomp/>.
- [32] P. B. Gibbons. 1989. A more practical PRAM model. In *Proceedings of SPAA*. 158–168.
- [33] Dan Grossman and Ruth E. Anderson. 2012. Introducing parallelism and concurrency in the data structures course. In *Proceedings of the Technical Symposium on Computer Science Education (SIGCSE '12)*. ACM, New York, NY, 505–510.
- [34] Dirk Grunwald and Harini Srinivasan. 1993. Data flow equations for explicitly parallel programs. In *Proceedings of PPoPP*. 159–168.
- [35] Pablo Halpern. 2012. *Strict Fork-Join Parallelism*. Technical Report N3409. Intel Corporation.
- [36] Pablo Halpern and Charles E. Leiserson. 2013. *Thread-Local Storage in X-Parallel Computations*. Technical Report N3556. Intel Corporation and MIT.
- [37] Robert H. Halstead, Jr. 1985. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.* 7, 4 (Oct. 1985), 501–538.
- [38] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. 2010. The Cilkview scalability analyzer. In *Proceedings of SPAA*.
- [39] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. 2009. *Improving Performance via Mini-applications*. Technical Report SAND2009-5574. Sandia National Laboratories.
- [40] C. A. R. Hoare. 1961. Algorithm 64: Quicksort. *Commun. ACM* 4, 7 (1961), 321.
- [41] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, and M. V. Zelkowitz. 2005. Parallel programmer productivity: A case study of novice parallel programmers. In *Proceedings of SC*.
- [42] Derek R. Hower, Polina Dudnik, Mark D. Hill, and David A. Wood. 2011. Calvin: Deterministic or not? Free will to choose. In *Proceedings of HPCA*. 333–334.
- [43] Institute of Electrical and Electronic Engineers. [n.d.]. Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]. *IEEE Standard 1003.1, 1996 Edition*.
- [44] Intel Corporation. 2010. *Intel Cilk Plus Application Binary Interface Specification*. Document Number: 324512-001US. Retrieved from https://software.intel.com/sites/products/cilk-plus/cilk_plus_abi.pdf.
- [45] Intel Corporation. 2010. *Intel Cilk Plus Language Specification*. Document Number: 324396-001US. Retrieved from http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf.
- [46] Intel Corporation. 2013. Cilk Plus/LLVM. Retrieved from <http://cilkplus.github.io/>.
- [47] Intel Corporation. 2013. *Intel Cilk Plus Language Extension Specification, Version 1.2*. Intel Corporation. Retrieved from https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm.
- [48] Intel Corporation. 2015. *Intel C++ Compiler 16.0 User and Reference Guide*.
- [49] Intel Corporation. 2018. Intel Cilk Plus Samples. Retrieved from <https://software.intel.com/en-us/code-samples/intel-compiler/all-samples-and-downloads>.
- [50] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A scalable architecture for ordered parallelism. In *Proceedings of MICRO*. 228–241.
- [51] Pramod G. Joisha, Robert S. Schreiber, Prithviraj Banerjee, Hans J. Boehm, and Dhruva R. Chakrabarti. 2011. A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code. In *Proceedings of POPL*. 623–636.
- [52] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. 2013. INSPIRE: The Insieme parallel intermediate representation. In *Proceedings of PACT*. 7–18.
- [53] Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language (2nd ed.)*. Prentice Hall, Inc.
- [54] D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin. 2012. *SPIRE, a Sequential to Parallel Intermediate Representation Extension*. Technical Report. Technical Report CRI/A-487, MINES ParisTech.
- [55] Dounia Khaldi, Pierre Jouvelot, François Irigoin, Corinne Ancourt, and Barbara Chapman. 2015. LLVM parallel intermediate representation: Design and evaluation using OpenSHMEM communications. In *Proceedings of LLVM*. 2:1–2:8.
- [56] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. 1996. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Program. Lang. Syst.* 18, 3 (May 1996), 268–299.
- [57] Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.* C-28, 9 (Sept. 1979), 690–691.

- [58] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of CGO*. 75–87.
- [59] Edward A. Lee. 2006. The problem with threads. *Computer* 39 (2006), 33–42.
- [60] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim Sukha. 2015. On-the-fly pipeline parallelism. *Trans. Parallel Comput.* 2, 3, Article 17 (2015), 42 pages.
- [61] I-Ting Angelina Lee, Aamir Shafi, and Charles E. Leiserson. 2012. Memory-mapping support for reducer hyperobjects. In *Proceedings of SPAA*. 287–297.
- [62] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. 1997. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Proceedings of LCPC*. 114–130.
- [63] Charles E. Leiserson. 2010. The Cilk++ concurrency platform. *J. Supercomput.* 51, 3 (2010), 244–257.
- [64] LLVM Developer List. 2012. [LLVMdev] [RFC] SPIR Provisional Specification Is Now Available in the Khronos Website. Retrieved from <http://lists.llvm.org/pipermail/llvm-dev/2012-September/053293.html>.
- [65] LLVM Developer List. 2012. [LLVMdev] [RFC] OpenMP Representation in LLVM IR. Retrieved from <http://lists.llvm.org/pipermail/llvm-dev/2012-September/053861.html>.
- [66] LLVM Developer List. 2012. [LLVMdev] [RFC] Parallelization Metadata and Intrinsics in LLVM (for OpenMP, Etc.). Retrieved from <http://lists.llvm.org/pipermail/llvm-dev/2012-September/053792.html>.
- [67] LLVM Developer List. 2015. [LLVMdev] LLVM Parallel IR. Retrieved from <http://lists.llvm.org/pipermail/llvm-dev/2015-March/083314.html>.
- [68] LLVM Project. 2015. OpenMP: Support for the OpenMP Language. Retrieved from <http://openmp.llvm.org/>.
- [69] LLVM Project. 2018. *Exception Handling in LLVM*. Retrieved from <https://llvm.org/docs/ExceptionHandling.html>.
- [70] LLVM Project. 2018. *LLVM Language Reference Manual*. Retrieved from <http://llvm.org/docs/LangRef.html>.
- [71] LLVM Project. 2018. *LLVM's Analysis and Transform Passes*. Retrieved from <http://llvm.org/docs/Passes.html>.
- [72] Michael McCool, Arch D. Robison, and James Reinders. 2012. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science.
- [73] Don McCrady. 2008. Avoiding Contention Using Combinable Objects. *Microsoft Developer Network blog post*. Retrieved from <http://blogs.msdn.com/nativeconcurrency/archive/2008/09/25/avoiding-contention-using-combinable-objects.aspx>.
- [74] Samuel P. Midkiff and David A. Padua. 1990. Issues in the optimization of parallel programs. In *Proceedings of ICPP*. 105–113.
- [75] Carroll Morgan. 1994. *Programming from Specifications* (2nd ed.). Prentice Hall International (UK) Ltd.
- [76] Joel Moses. 1970. *The Function of FUNCTION in LISP or Why the FUNARG Problem Should be Called the Environment Problem*. Technical Report memo AI-199. Massachusetts Institute of Technology Artificial Intelligence Laboratory.
- [77] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.
- [78] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. 2009. Analytical modeling of pipeline parallelism. In *Proceedings of PACT*. 281–290.
- [79] Robert H. B. Netzer and Barton P. Miller. 1992. What are race conditions? *ACM Lett. Program. Lang. Syst.* 1, 1 (1992), 74–88.
- [80] Diego Novillo, Ron Unrau, and Jonathan Schaeffer. 1998. Concurrent SSA form in the presence of mutual exclusion. In *Proceedings of ICPP*. 356–364.
- [81] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: Efficient deterministic multithreading in software. In *Proceedings of ASPLOS*. 97–108.
- [82] OpenMP Architecture Review Board. 2015. *OpenMP Application Program Interface, Version 4.5*. Retrieved from <http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>.
- [83] Suhas S. Patil. 1970. Closure properties of interconnections of determinate systems. In *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, Jack B. Dennis (Ed.). ACM.
- [84] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of parallelism in algorithms. In *Proceedings of ACM PLDI*.
- [85] Antoniu Pop and Albert Cohen. 2010. Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers. In *Proceedings of CPC*.
- [86] William Pugh. 1999. Fixing the Java memory model. In *Proceedings of JAVA*. 89–98.
- [87] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. 2009. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Proceedings of PDP*. 427–436.
- [88] James Reinders. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc.
- [89] Arch D. Robison and Ralph E. Johnson. 2010. Three layer cake for shared-memory programming. In *Proceedings of ParaPLoP*. 5:1–5:8.

- [90] Erik Ruf. 2000. Effective synchronization removal for Java. In *Proceedings of PLDI*. 208–218.
- [91] Radu Rugina and Martin C. Rinard. 2003. Pointer analysis for structured parallel programs. *ACM Trans. Program. Lang. Syst.* 25, 1 (Jan. 2003), 70–116.
- [92] Vivek Sarkar. 1998. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *Proceedings of LCPC*. 94–113.
- [93] Vivek Sarkar and Barbara Simons. 1994. Parallel program graphs and their classification. In *Proceedings of LCPC*. 633–655.
- [94] Tao B. Schardl. 2016. *Performance Engineering of Multicore Software: Developing a Science of Fast Code for the Post-Moore Era*. Ph.D. Dissertation. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA.
- [95] Tao B. Schardl, Tyler Denniston, Damon Doucet, Bradley C. Kuszmaul, I-Ting Angelina Lee, and Charles E. Leiserson. 2017. The CSI framework for compiler-inserted program instrumentation. *Proc. ACM Meas. Anal. Comput. Syst.* 1, 2 (Dec. 2017).
- [96] Tao B. Schardl, Bradley C. Kuszmaul, I-Ting Angelina Lee, William M. Leiserson, and Charles E. Leiserson. 2015. The Cilkprof scalability profiler. In *Proceedings of SPAA*. 89–100.
- [97] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2017. Tapir: Embedding fork-join parallelism into LLVM’s intermediate representation. In *Proceedings of PPoPP*. 249–265.
- [98] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. 2009. Phaser accumulators: A new reduction construct for dynamic parallelism. In *Proceedings of IPDPS*.
- [99] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. 2013. Reducing contention through priority updates. In *Proceedings of SPAA*. 152–163.
- [100] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyröla, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: The problem-based benchmark suite. In *Proceedings of SPAA*. 68–70.
- [101] Harini Srinivasan and Dirk Grunwald. 1991. *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.
- [102] Harini Srinivasan, James Hook, and Michael Wolfe. 1993. Static single assignment for explicitly parallel programs. In *Proceedings of POPL*. 260–272.
- [103] Harini Srinivasan and Michael Wolfe. 1991. Analyzing programs with explicit parallelism. In *Proceedings of LCPC*. 405–419.
- [104] Richard M. Stallman and the GCC Developer Community. 2016. *Using the GNU Compiler Collection (for GCC version 6.1.0)*. Free Software Foundation.
- [105] Guy L. Steele Jr. 1990. Making asynchronous parallelism safe for the world. In *Proceedings of POPL*. 218–231.
- [106] George Stelle, William S. Moses, Stephen L. Olivier, and Patrick McCormick. 2017. OpenMPIR: Implementing openmp tasks with tapir. In *Proceedings of LLVM-HPC*.
- [107] Bjarne Stroustrup. 2013. *The C++ Programming Language* (4th ed.). Addison-Wesley.
- [108] Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. 2016. Provably good and practically efficient parallel race detection for fork-join programs. In *Proceedings of SPAA*. 83–94.
- [109] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *Proceedings of POPL*. 209–220.
- [110] Eelco Visser. 2001. A survey of rewriting strategies in program transformation systems. *Electr. Notes Theoret. Comput. Sci.* 57 (2001), 109–143.
- [111] Martin Wimmer. 2013. Wait-free hyperobjects for task-parallel programming systems. In *Proceedings of IPDPS*. 803–812.
- [112] Jie Yu and Satish Narayanasamy. 2009. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of ISCA*. 325–336.
- [113] Jisheng Zhao and Vivek Sarkar. 2011. Intermediate language extensions for parallelism. In *Proceedings of SPLASH*. 329–340.

Received April 2018; revised January 2019; accepted April 2019