OpenCilk: A Modular and Extensible Software Infrastructure for Fast Task-Parallel Code

Tao B. Schardl
MIT CSAIL
neboat@mit.edu

I-Ting Angelina Lee
Washington University in St. Louis
angelee@wustl.edu

Abstract

This paper presents OpenCilk, an open-source software infrastructure for task-parallel programming that allows for substantial code reuse and easy exploration of design choices in language abstraction, compilation strategy, runtime mechanism, and productivity-tool development.

The OpenCilk infrastructure consists of three main components: a compiler designed to compile fork-join task-parallel code, an efficient work-stealing runtime scheduler, and a productivity-tool development framework based on compiler instrumentation designed for fork-join parallel computations. OpenCilk is *modular* — modifying one component for the most part does not necessitate modifications to the other components — and *easy to extend* — its construction naturally encourages code reuse. Despite being modular and easy to extend, OpenCilk produces *high-performing* code.

We investigated OpenCilk's modularity, extensibility, and performance through several case studies, including a study to extend OpenCilk to support multiple runtime systems, including Cilk Plus, OpenMP, and oneTBB. OpenCilk's design enables rapid prototyping of new compiler back ends to target different parallel-runtime ABIs. Each back end required fewer than 2000 new lines of code. We examined the OpenCilk runtime's performance empirically on 15 benchmark Cilk programs and found that it outperforms the other runtimes by a geometric mean of 4%–26% on 1 core and 10%–120% on 48 cores.

Keywords: bitcode, Cilk, compiling, compiler-inserted instrumentation, fork-join parallelism, OpenCilk, OpenMP, oneTBB, parallel computing, parallel runtime system, productivity tools, Tapir, task parallelism

1 Introduction

The widespread deployment of multicore platforms — from personal computers to mobile devices to supercomputers — has led to an increased interest in finding simple approaches to programming them. ***Task parallelism***[1] is a popular approach where the platform provides high-level programming abstractions to express the *logical parallelism* of the computation and allows an underlying runtime system to automate load-balancing and synchronization. Task parallelism continues to draw significant attention from the research community to explore different language abstractions (e.g., [11, 34, 35, 53, 68, 69, 77]), compilation techniques (e.g., [8, 9, 25, 51, 64, 78, 83]), scheduling algorithms (e.g., [5, 6, 52, 70–73]), runtime mechanisms (e.g., [4, 31, 44, 47, 49, 67, 76]), and productivity tools (e.g., [28, 33, 60, 63, 79–82, 84–87]). Task parallelism has been adopted into mainstream compilers [3, 30, 38, 48] and mainstream programming languages, such as via `task::spawn` in the Tokio runtime in Rust [1] and `goroutine` in Go [2].

Although task parallelism is becoming mainstream, its software development tool chain remains rudimentary. Today's compilers for serial languages consist of three phases: 1) a ***front end*** that parses and type-checks the input program and translates it into an abstract target-independent language, called an ***intermediate representation (IR)***; 2) a ***middle end*** that performs analyses and optimizations on the IR; and 3) a ***back end*** that performs translates the optimized IR into target-specific machine code. Serial control constructs are represented in the IR, and the middle end accounts for their semantics when performing optimizations. In contrast, existing task-parallel platforms (e.g., [7, 15, 16, 23, 29, 46, 54]) support parallel control constructs as syntactic sugar for opaque function calls into a runtime library. The parallel constructs are exclusively processed at the front end, where each parallel construct is transformed into a sequence of instructions interleaved with calls into a corresponding runtime system, which is typically linked as a separate library.

This approach has several downsides. First, the compilation process is tightly coupled to both a specific parallel language and the application binary interface (ABI) of a specific runtime system. Changing this ABI generally requires changing both the runtime system and the compiler, whose codebase is often 100× larger than the runtime system's. As a result, the compiler and runtime system act like a monolithic piece of software that is difficult to modify and extend. Second, this approach only supports minimal code reuse. A compiler-runtime combination for a C/C++-based task-parallel platform is difficult to port to a different language front end or a different runtime system. Finally, this approach can harm performance. The compiler is largely oblivious to parallel semantics, and calls into the runtime system are

[1] In the literature, task parallelism has also been referred to as cooperative threading, dynamic multithreading, or fine-grained parallelism.

PPoPP '23, February 25–March 1, 2023, Montreal, QC, Canada
2023. ACM ISBN 979-8-4007-0015-6/23/02.
https://doi.org/10.1145/3572848.3577509

treated as having unknown side effects that block compiler optimization. Efforts to reenable compiler optimizations end up tied to a particular compiler-runtime ABI (e.g., [25]).

OpenCilk

This work aims to improve the software infrastructure for task parallelism to enable substantial code reuse and easy exploration of design choices in language abstraction, compilation strategy, runtime scheduling, and productivity-tool development. This paper introduces ***OpenCilk***, an open-source software infrastructure[2] that consists of three main components: a compiler designed to compile fork-join task-parallel code, an efficient runtime system that implements randomized work stealing [12], and the ***Momme***[3] tool-development framework that provides compiler instrumentation designed specifically for fork-join parallel programs.

OpenCilk focuses on satisfying three design goals:

Modularity: Modifying one component of OpenCilk does not necessitate changes to other components.

Extensibility: OpenCilk's architecture naturally encourages code reuse.

Performance: OpenCilk produces high-performing code.

OpenCilk builds upon three existing technologies — the LLVM compiler [41, 42], Tapir [64], and CSI [62] — that each fall short of OpenCilk's design goals. LLVM provides a mainstream modular compiler infrastructure that produces high-performing code, but provides limited support for parallelism. Tapir extends LLVM's IR to express fork-join parallelism for high-performing parallel code. But the Tapir/LLVM compiler — the original implementation of Tapir in LLVM — has limited modularity and extensibility. For example, Tapir/LLVM only implements the Cilk Plus runtime ABI, and extending Tapir/LLVM to support other parallel runtimes requires substantial, error-prone compiler development work. CSI provides a flexible and powerful framework that allows tools to use compiler-inserted program instrumentation while being developed outside of the compiler codebase, but it does not provide general support for task parallelism. In addition, CSI relies on link-time optimization (LTO) [74] to produce high-performing code. Finally, CSI lacks support for stack-allocated tool data structures, meaning a tool written with CSI cannot insert tool-specific data into the function frames of the program-under-test. Such a feature can simplify the tool development and lead to better performing tools.

To overcome their limitations, OpenCilk combines these technologies and builds upon them in three key ways.

First, the OpenCilk compiler builds upon Tapir/LLVM and introduces a ***Tapir-lowering infrastructure*** that provides a general framework to transform Tapir to fit the ABIs for different parallel runtime systems. This infrastructure allows a particular runtime-system's ABI to be implemented as a simple plugin, called a ***Tapir target***. Because the Tapir-lowering infrastructure operates on Tapir's constructs, rather than the program's source code, it allows task-parallel programs to use different parallel runtime systems interchangeably, regardless of their source task-parallel language.

Second, OpenCilk introduces ***bitcode ABIs***, a mechanism for implementing the ABI of a component that promotes modularity and extensibility. A bitcode ABI allows a component, such as the OpenCilk runtime system or a tool in the OpenCilk tool suite, to implement all of its ABI definitions in LLVM bitcode. The ABI therefore does not need to be hard-coded into the OpenCilk compiler, unlike the traditional design of a compiler-runtime ABI. As a result, many sophisticated changes to component ABIs can be implemented without any change to the OpenCilk compiler. In addition, bitcode ABIs ensure that separating the compiler-runtime ABI from the compiler codebase does not harm performance.

Third, the Momme tool-development framework extends CSI in several ways to support efficient tools for task-parallel programs. Momme allows tools to instrument a task-parallel program based on Tapir's representation of parallelism. As a result, Momme supports tools that analyze the logical parallelism of a program-under-test, regardless of the program's source language or parallel runtime system. In addition, Momme can take a bitcode ABI for a tool as input. Momme uses the bitcode ABI both to optimize and tailor the tool's instrumentation without requiring LTO and to allow tools to insert custom data structures into the function frames of the program-under-test.

Evaluation of OpenCilk

This paper demonstrates the modularity and extensibility of OpenCilk through six case studies.

In one case study, we ported OpenCilk to support additional parallel runtime ABIs by adding new Tapir targets for the OpenMP runtime [59][4] and Intel oneAPI Threading Building Blocks (oneTBB) [39]. The Tapir-lowering infrastructure and bitcode ABIs allowed for rapid prototyping. Adding the OpenMP and oneTBB Tapir targets required only about 850 and 750 new lines of code, respectively.

In another case study, we extended OpenCilk to support deterministic parallel random-number generators (DPRNGs) [47]. In the Cilk Plus runtime, adding efficient support for DPRNGs requires modifying the runtime's ABI, which in turn requires changing the compiler [47, 80]. In contrast, OpenCilk's design allows us add DPRNG support to the OpenCilk runtime system by changing only the runtime system's codebase. Furthermore, this modularity does not sacrifice performance. The OpenCilk runtime system, modified to provide two mechanisms to support DPRNGs, runs

[2]The OpenCilk software infrastructure is freely available at https://github.com/OpenCilk.

[3]Momme, in Japanese, is a unit traditionally used to measure the quality of silk fabrics.

[4]Specifically, the OpenMP runtime distributed with LLVM 14.

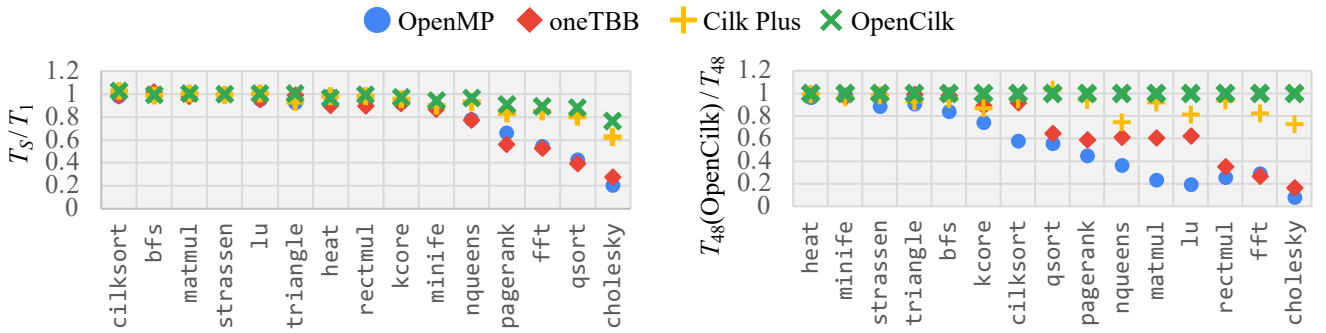

Figure 1. Performance comparison of the OpenMP, oneTBB, Cilk Plus, and OpenCilk runtime systems on 15 benchmark Cilk programs. The left plot presents the work efficiency T_S/T_1 of each executable, where T_S is the running time of the serial projection. The right plot presents each executable's 48-core speedup T_S/T_{48} divided by the 48-core speedup of the corresponding executable that uses the OpenCilk runtime. Higher is better for both plots.

a geometric mean of 5% faster than the Cilk Plus runtime system, which only provides one such mechanism.

Finally, to show that OpenCilk produces high-performing code, we used OpenCilk to compile 15 C/C++-based fork-join parallel programs to run on either the OpenCilk runtime, the OpenMP runtime [59], the oneTBB runtime [39], or the Cilk Plus runtime [36], and we compared the performance of the resulting executables. Figure 1 shows the results.[5] In the figure, the left plot shows each executable's ***work efficiency***, T_S/T_1, where T_S is the running time of the program's ***serial projection*** — in which all parallel control constructs are removed from the program — and T_1 is the running time of parallel executable on 1 core. The right plot shows each executable's 48-core speedup, T_S/T_{48}, divided by the 48-core speedup of the corresponding executable that uses the OpenCilk runtime. As the figure shows, the OpenCilk runtime system consistently provides work efficiency and speedup that is comparable to or better than that of the other runtime systems. In particular, on 48 cores, the OpenCilk runtime runs a geometric mean of approximately 10% faster than Cilk Plus, 50% faster than oneTBB, and 120% faster than OpenMP.

In summary, this paper contributes the following:

- Section 2 introduces OpenCilk, a open-source software infrastructure for fork-join parallelism that includes a compiler, a runtime system, and a tool-development framework. OpenCilk employs several novel design features, including a Tapir-lowering infrastructure and bitcode ABIs, that make it modular, easy to extend, and fast.
- Section 3 shows that OpenCilk is easier to extend than the conventional approach. Through several case studies, we illustrate how the design of OpenCilk makes it easy to add new language abstractions, runtime mechanisms, and fast productivity tools.
- Section 4 demonstrates that OpenCilk produces high-performing code in practice and empirically evaluates runtime extensions built using the OpenCilk infrastructure that are discussed in Section 3.

[5] Appendix A presents the running-time data.

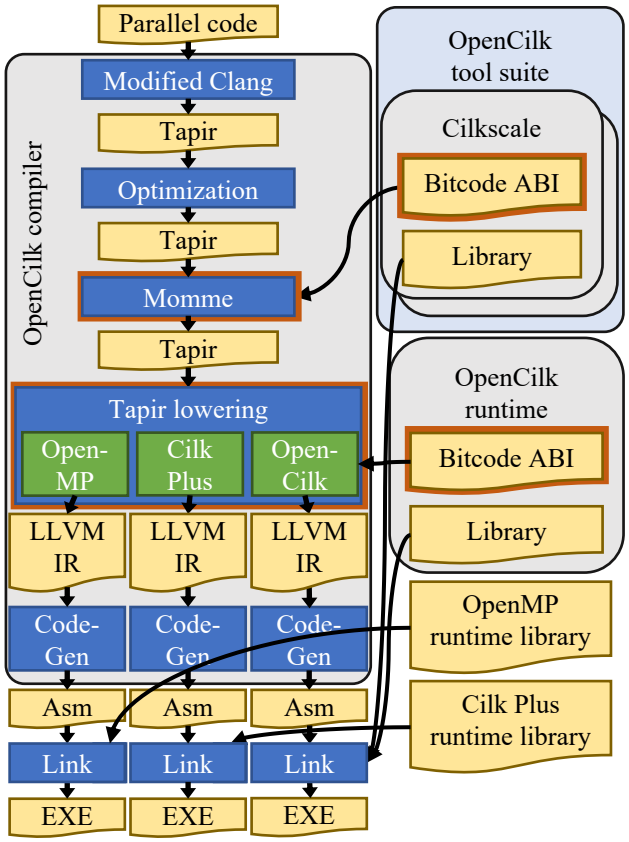

Figure 2. Diagram of the OpenCilk system architecture. Rounded rectangles distinguish the OpenCilk compiler, runtime system, and tool suite. Rectangles describe compiler and linker stages. The rectangles within the "Tapir lowering" rectangle represent Tapir targets. Document shapes represent code in some form. Highlighting indicates key aspects of OpenCilk discussed in Section 2.

2 OpenCilk system architecture

This section describes the OpenCilk system architecture and the features that make it easy to modify and extend. OpenCilk includes a compiler, a work-stealing runtime system [12],

and a framework for developing program-analysis tools, including a race detector and a scalability analyzer. Figure 2 illustrates the OpenCilk system architecture, with key aspects highlighted. The OpenCilk compiler introduces a unifying Tapir-lowering framework, to compile task-parallel programs to different parallel runtime systems, and the Momme framework, to support the development of program-analysis tools. Figure 2 also shows how the OpenCilk runtime system and OpenCilk tools are implemented as combinations of conventional libraries and bitcode ABIs, to make them easy to modify and extend without sacrificing performance.

Preliminaries

OpenCilk supports (recursive) fork-join parallelism, in which a function can ***spawn*** a ***child*** task, thereby allowing it to be executed in parallel with the continuation of its ***parent***. A function can also perform a ***sync*** to wait for all of its spawned child tasks to complete. Tasks can themselves spawn and sync subtasks. For convenience, we refer to any function that performs a spawn as a ***spawning function***.

Currently, the OpenCilk compiler supports C/C++-based fork-join parallel code written using Cilk syntax [29] and multiple runtime back ends, including the OpenCilk runtime, the Intel Cilk Plus [36] runtime, the OpenMP runtime in LLVM 14 [59], and Intel's oneTBB runtime [39]. The runtime back ends for OpenMP and oneTBB utilize only a subset of the OpenMP and oneTBB runtime systems necessary to parallelize fork-join code. This section focuses on the OpenCilk runtime back end when discussing OpenCilk's architecture. Section 3 discusses the other back ends.

The OpenCilk compiler is based on Tapir/LLVM [64], which extends LLVM's IR to encode fork-join parallelism in a manner that is independent of language front ends and parallel runtime systems. LLVM IR represents a function as a ***control-flow graph (CFG)*** (V, E, v_0), where each vertex $v \in V$ represents a ***basic block*** — a sequence of instructions where control flow must enter through the first instruction and leave from the last instruction, which is called a ***terminator*** — and edges E denote control-flow dependencies between basic blocks. Vertex $v_0 \in V$ denotes the function's single entry point. Tapir adds three terminators to LLVM IR to express spawning and synchronization of tasks. Tapir represents each spawned task as a subCFG, and it guarantees that all tasks are well nested. Each basic block is ***contained*** in at most one task t, meaning it is included in the subCFG of t and not the subCFG of a task that t spawns.

Tapir-lowering infrastructure

The OpenCilk compiler implements a Tapir-lowering infrastructure to make it easy to lower Tapir constructs to different parallel-runtime ABIs. Although the ABIs for parallel runtime systems differ substantially, many ABIs, including those for OpenCilk, Cilk Plus [36], Habanero [7, 15], and OpenMP [55] share several properties in common. In particular, these runtimes all require a spawned task to eventually be encoded as a function in the binary, so that, for example, the runtime system can refer to a task internally using a function pointer. Tapir, however, encodes spawned tasks directly within the CFGs of spawning functions, in order to facilitate compiler optimizations [64]. To lower Tapir constructs, the Tapir-lowering infrastructure provides a general facility to efficiently create a separate function for each spawned task.

Algorithm 1: Lower Tapir to a parallel-runtime ABI according to a given Tapir target.

Data: A CFG for a function F and a Tapir target, Target.
Result: A set of functions such that each parallel task in F is encoded in a separate function and all Tapir instructions in F are transformed according to the parallel-runtime ABI encoded by Target.

```
1  T ← ComputeTaskTree(F);
2  H ← ∅, I ← ∅;
3  for t ∈ T in post order do
4      if children(t) ≠ ∅ then
5          Target.ApplySpawnerABI(t);
6          for c ∈ children(t) do
7              Target.ApplySpawnABI(t, H[c]);
8          for y ∈ syncs(t) do
9              Target.ApplySyncABI(y);
10     I[t] ← ComputeInputs(t, I);
11     h ← CreateSpawnHelper(t, I[t]);
12     H[t] ← (h, spawnsite(t), I[t]);
```

Algorithm 1 presents pseudocode for the high-level algorithm that the Tapir-lowering infrastructure implements. The algorithm operates on a function F, in the form of a CFG with Tapir instructions, and a Tapir target, which encodes compiler-internal operations for implementing the ABI of a particular runtime system. The OpenCilk compiler also implements a separate, but substantively similar, algorithm that lowers parallel loops in particular.

Let us walk through Algorithm 1, temporarily ignoring the use of the Tapir target on lines 4–9. Line 1 calls `ComputeTaskTree` to compute a ***task tree*** for F, in which each vertex denotes a task in F — where the root denotes F itself — and edges indicate that one task can spawn another. `ComputeTaskTree` also associates each basic block in F with the task that contains it. Lines 3–12 traverse the task tree in post-order. For each task t, lines 4–9 invoke the Tapir target to process the ***spawn site*** of each child task c of t — the location in the CFG where c is spawned — and each sync in t. Line 10 calls `ComputeInputs` to compute the set $I[t]$ of inputs to t. Line 11 calls `CreateSpawnHelper` to create a new ***spawn helper*** function h for t. Line 12 records h, the spawn site of t, and $I[t]$ for the Tapir target to process later.

Although Algorithm 1 is simple, the `ComputeTaskTree`, `ComputeInputs`, and `CreateSpawnHelper` subroutines encapsulate significant complexity. These routines handle various intricacies that arise in real programs, such as identifying exception-handling code with different tasks, handling

a

```
01 int fib(int n) {
02   if (n < 2) return n;
03   int x, y;
04   fib_helper(&x, n-1);
05   y = fib(n-2);
06   sync;
07   return x + y;
08 }
09 void fib_helper(int *x, int n) { *x = fib(n); }
```

b

```
10 int fib(int n) {
11   __cilkrts_stack_frame sf;
12   __cilkrts_enter_frame(&sf);
13   if (n < 2) {
14     __cilkrts_leave_frame(&sf);
15     return n;
16   }
17   int x, y;
18   if (__cilkrts_prepare_spawn(&sf))
19     fib_helper(&x, n-1);
20   y = fib(n-2);
21   __cilkrts_sync(&sf);
22   __cilkrts_leave_frame(&sf);
23   return x + y;
24 }
25 void fib_helper(int *x, int n) {
26   __cilkrts_stack_frame sf;
27   __cilkrts_enter_helper_frame(&sf);
28   __cilkrts_detach(&sf);
29   *x = fib(n);
30   __cilkrts_leave_helper_frame(&sf);
31 }
```

Figure 3. C-pseudocode illustrating how the OpenCilk Tapir target operates on a parallel OpenCilk program to compute Fibonacci numbers. **a** C-pseudocode for the result of Tapir lowering before invoking the Tapir target. **b** C-pseudocode illustrating the code from (a) transformed by the Tapir target.

function and parameter attributes, and maintaining debug information. The Tapir-lowering infrastructure aims to minimize the need for Tapir targets to handle these complexities themselves. Algorithm 1 also helps to ensure that these subroutines are efficient by performing a constant number of passes over all basic blocks to create spawn helpers for all tasks, including nested tasks, in a function.

Tapir targets

To implement a particular parallel-runtime ABI, Algorithm 1 takes a Tapir target as an argument. The Tapir target defines a set of routines, `ApplySpawnerABI`, `ApplySpawnABI`, and `ApplySyncABI`, to modify the CFGs of spawning and spawn-helper functions according to an ABI.

Figure 3 illustrates, in C-pseudocode, a simple example of how the Tapir-lowering infrastructure uses the OpenCilk Tapir target. Conceptually, Figure 3(a) illustrates a spawning function at the point of line 4 in Algorithm 1, after the algorithm has created a spawn helper for the spawned task (line 9). The Tapir-lowering infrastructure uses the OpenCilk Tapir target to generate Figure 3(b) from Figure 3(a) as follows. `ApplySpawnerABI` inserts lines 11, 12, 14, and 22 in Figure 3(b). `ApplySpawnABI` inserts lines 26, 27, 28, and 30 and transforms line 4 into lines 18–19. `ApplySyncABI` transforms line 6 into line 21.

Some runtime ABIs require ABI functions to be inlined into their callers for correctness or performance. To support such ABIs and generally optimize runtime-ABI code, the Tapir-lowering infrastructure runs a few simple optimizations after running Algorithm 1 on all functions, including function inlining, constant propagation, common-subexpression elimination, and dead-code elimination.

Bitcode ABIs

Although the Tapir-lowering infrastructure can flexibly lower parallel tasks to different runtime ABIs, it can still require changes to the OpenCilk compiler's internals (i.e., Tapir targets) in order to change a runtime ABI. For example, some runtime-system ABIs add a local ***stack-frame variable*** to every spawning function and spawn helper. To insert those variables, the corresponding Tapir targets must know the variable's type definition. Therefore, changing the contents or layout of the stack-frame variable in the runtime system requires changing the Tapir target as well.

To relax this requirement, OpenCilk introduces bitcode ABIs for the runtime system and tools to interact with the compiler. Figure 2 illustrates that the OpenCilk runtime system consists of a standard library as well as a ***bitcode-ABI file***, which is an LLVM bitcode file — a binary representation of LLVM IR — that encodes definitions of the OpenCilk compiler-runtime ABI. This bitcode-ABI file includes the type definition of the OpenCilk runtime's stack-frame variable as well as definitions of runtime ABI functions (e.g., `__cilkrts_stack_frame` and functions whose names start with `__cilkrts_` in Figure 3(b)). When Tapir lowering runs, the OpenCilk Tapir target reads in and internalizes the OpenCilk runtime's bitcode-ABI file before any of its routines are called, allowing the OpenCilk Tapir target to get the stack-frame variable's type definition of from the bitcode-ABI file before it inserts any local variables of that type.

By using a bitcode ABI, the OpenCilk Tapir target can implement the OpenCilk runtime ABI without requiring the ABI to be hard-coded into the Tapir target. One can thus change the OpenCilk runtime system, including its ABI, without needing to modify the OpenCilk compiler. As long as the modified runtime system uses the same names for ABI functions and types at the same places within the lowered program, the modified runtime can be used without any changes to the compiler. Moreover, this approach does not sacrifice performance, as the OpenCilk Tapir target reads in and internalizes the OpenCilk runtime's bitcode-ABI file before it performs runtime-specific lowering, which allows the runtime data declarations and function invocations to be inlined. Section 3 discusses how the bitcode ABI greatly simplifies changing the OpenCilk runtime system.

Tool development

OpenCilk provides the Momme tool-development framework, which supports productivity tools including the ***Cilksan*** race detector and the ***Cilkscale*** scalability analyzer. Momme is based on the CSI framework [62], which allows tools that use compiler instrumentation to be developed outside of the compiler. Momme adds an instrumentation stage to the OpenCilk compiler that inserts generic function calls, known as ***(instrumentation) hooks***, throughout the program-under-test that tools can implement to instrument a task-parallel program.

Momme builds upon CSI in three key ways to make it easy to develop fast productivity tools in a modular fashion.

First, Momme provides hooks for Tapir's IR constructs, allowing tools to instrument spawns and syncs in a fork-join parallel program. By instrumenting Tapir's IR constructs, tools can analyze parallel control flow independently of what parallel runtime system the program uses. OpenCilk thus allows tools and parallel runtime systems to be modified independently without breaking each other's functionality.

Second, Momme can optionally accept a bitcode ABI for a tool. This capability provides several advantages. Momme can use a bitcode ABI to tailor the hooks it inserts into a program-under-test, as discussed in Section 3. Momme can also use a tool's bitcode ABI to optimize the inserted instrumentation. In contrast, CSI relies on link-time optimization (LTO) to provide similar capabilities.[6]

Finally, Momme allows tools to optionally define, within a bitcode ABI, a structure type for a variable that Momme will insert into every function on behalf of the tool. Momme thereby allows tools to use storage allocated on the program's call stack to achieve better performance. Similarly to the Tapir-lowering infrastructure, Momme runs a few simple optimizations after inserting instrumentation to facilitate optimizing that instrumentation. Section 3 explores this capability in the context of Cilkscale.

3 Case studies

This section presents six case studies that explore OpenCilk's modularity and extensibility. We describe our experience using the Tapir-lowering infrastructure to allow the OpenCilk compiler to target different parallel runtime ABIs. We examine how to add support for DPRNGs [47, 75] to the OpenCilk runtime system. We discuss our experience using OpenCilk to add parallel language constructs to Kaleidoscope, a simple language that is used to teach LLVM internals [58]. We examine how to add a synchronizing lexical scope to the OpenCilk front end. We study how bitcode ABIs support tool development in the context of Cilkscale and Cilksan.

[6]Schardl et al. describe an alternative design for CSI that uses compile-time optimization (CTO) instead of LTO, which is similar to how the Momme uses bitcode-ABI files. However, Schardl et al. did not implement the CTO strategy, nor did they explore additional capabilities that Momme supports.

Supporting multiple runtime systems

We used the Tapir-lowering infrastructure to allow the OpenCilk compiler to target four different parallel-runtime ABIs, i.e., for the OpenCilk runtime, the Cilk Plus runtime, the OpenMP runtime [59] in LLVM 14 (which supports OpenMP 4.5 [54]) and oneTBB [39]. We implemented a distinct Tapir target for each runtime ABI. These targets allow OpenCilk to compile Cilk programs to use any of these runtime systems.

The Tapir-lowering infrastructure simplifies the effort to implement these Tapir targets substantially. The OpenCilk Tapir target is approximately 1300 lines of code within the OpenCilk compiler (plus 380 lines in a bitcode-ABI file), while the Cilk Plus Tapir target is approximately 1900 lines of code, and the OpenMP and oneTBB Tapir targets are each less than 700 lines of code (plus 80–150 lines in bitcode-ABI files). In contrast, the Tapir-lowering infrastructure is significantly larger, approximately 5800 lines of code.

Although the Cilk Plus target resembles the OpenCilk target — because they share similar runtime ABIs — it does not use a bitcode-ABI file. Instead, the Cilk Plus target hard-codes its runtime ABI, in order to insert stack-frame variables and inline runtime functions into the compiled code. For this reason, the Cilk Plus target contains more lines of code than the OpenCilk target, which illustrates how using a bitcode-ABI file can simplify runtime-back-end implementation. Moreover, even minor changes to the Cilk Plus runtime ABI can necessitate compiler changes, which is not the case for the OpenCilk runtime. Finally, we observe that the use of a bitcode-ABI file does not hurt performance: the OpenCilk runtime obtains comparable or better work efficiency than Cilk Plus (Figure 1) despite sharing a similar runtime ABI.

Our implementations of the OpenMP and oneTBB Tapir targets use subsets of the corresponding runtime-scheduling libraries. The OpenMP target uses the OpenMP runtime functions and data structures that implement `omp task` and `omp taskwait` pragmas [54]. The oneTBB target creates a local `task_group` in each spawning function. To implement a spawn, the oneTBB target creates a C++ lambda to execute the spawn helper function and calls `run` on the local `task_group` to execute that lambda. To implement a sync, the oneTBB target calls `wait` on the local `task_group`.

To simplify their implementations, both the OpenMP and oneTBB Tapir targets use bitcode-ABI files. The OpenMP runtime requires initializing specific data structures and invoking a sequence of functions to spawn a task. The bitcode ABI allows us to encode these protocols in a small C source file rather than hard-code them into the compiler. For oneTBB, the `task_group.run` function takes the spawned task as a C++ function object, e.g., a C++ lambda. Our implementation punts the construction and invocation of the C++ lambda to the bitcode-ABI file, thereby avoiding the need to deal with the C++ calling convention for lambda expressions, such as data packaging and name mangling, at the LLVM IR level.

Supporting DPRNGs

To explore how the OpenCilk architecture can support research in task-parallel runtime systems, we examined how to extend OpenCilk to support deterministic parallel random-number generators (DPRNGs) [47]. A DPRNG generates pseudorandom numbers in a parallel program deterministically, regardless of how it is scheduled. To support DPRNGs for Cilk programs, Leiserson et al. introduced a runtime-system extension that deterministically assigns a unique label, called a ***pedigree***, to each ***strand*** of the computation, that is, each sequence of executed instructions that contain no parallel control. By hashing pedigrees, DPRNG library can generate pseudorandom numbers for different strands deterministically in parallel. Leiserson et al. showed how the Cilk Plus runtime system can maintain pedigrees efficiently, and pedigrees were incorporated into Cilk Plus.

Cilk Plus's pedigree implementation affects its compiler-runtime ABI. Similarly to the OpenCilk runtime ABI, the Cilk Plus runtime ABI adds a stack-frame structure to each spawning function and spawn helper. To support pedigrees, Cilk Plus augments this stack-frame structure and the functions in its runtime ABI that use this structure. Therefore, modifying this support for DPRNGs — e.g., to build a DPRNG into the runtime system that implements Steele et al.'s proposed optimizations [75] — requires changing both the compiler and runtime for Cilk Plus.

We leveraged the OpenCilk runtime system's bitcode ABI to develop ***OpenCilk-DPRNG***, an extension to OpenCilk that adds both support for pedigrees, based on the algorithm by of Leiserson et al. [47], and a ***builtin DPRNG*** that incorporates some of Steele et al.'s optimizations [75]. Although these changes affect the OpenCilk runtime ABI, the bitcode ABI allows these changes to be encapsulated within the OpenCilk runtime codebase. Section 4 presents our empirical evaluation of OpenCilk-DPRNG and finds that it is generally faster than the Cilk Plus runtime, despite maintaining both pedigrees and a builtin DPRNG. Hence, bitcode ABIs provide modularity without harming performance.

Parallelizing Kaleidoscope

We explored how OpenCilk can be used to add support for fork-join parallelism to Kaleidoscope, a simple language used in a standard LLVM tutorial [58]. Kaleidoscope is a procedural language with functions, conditionals, loops, and basic math operations. The LLVM tutorial implements a read-eval-print loop for Kaleidoscope that uses an LLVM-based JIT compiler to compile and execute code.

To explore the extensibility of OpenCilk, we modified the Kaleidoscope tutorial code to add fork-join parallel language constructs to Kaleidoscope, including constructs to spawn and sync tasks and a parallel-loop construct. OpenCilk allowed us to add these constructs to Kaleidoscope by adding just a few hundred lines of code to the Kaleidoscope codebase. We added approximately 400 lines of code to parse the new language constructs and generate Tapir for them. We added approximately 150 lines to modify the JIT compiler to invoke the Tapir-lowering infrastructure with the OpenCilk Tapir target and, optionally, to invoke Momme. We added approximately 100 lines to modify the JIT compiler to link external libraries. These changes sufficed to allow a programmer to write fork-join parallel Kaleidoscope code, compile and run it using OpenCilk, and optionally use Cilksan and Cilkscale to analyze its parallel execution. No changes were needed to the OpenCilk runtime system or any tools.

Adding synchronizing lexical scopes

We explored how the OpenCilk architecture can support a `cilk_scope` construct, which defines lexical scope that synchronizes spawned subroutines, similar to the `finish` statement in X10/Habanero [7, 15]. Intuitively, to support a `cilk_scope`, the OpenCilk compiler's front end inserts a Tapir sync instruction at each point in the IR that corresponds with the end of the `cilk_scope`. But complications arise when a `cilk_scope` synchronizes a subset of the spawned children of a function. In particular, the OpenCilk runtime system only supports synchronizing spawned subcomputations at the granularity of functions in the binary. To support this case, we modified the OpenCilk compiler to place such a `cilk_scope` in its own function.

We added support for the `cilk_scope` construct using a few changes to the OpenCilk compiler and no changes to the OpenCilk runtime system or any tools. We added two ***outline-scope*** intrinsic functions (intrinsics for short) that delineate the start and ends of a `cilk_scope`. Because these intrinsics have simple serial semantics, most compiler analyses and code-transformations ignore them, meaning these intrinsics have minimal impact on compiler optimization. We augmented the Tapir-lowering infrastructure to recognize outline-scope intrinsics and generate functions for the subCFGs they delineate.

Optimizing Cilkscale

To study the performance impact of using bitcode ABIs, we examined how Cilkscale could be implemented with and without a bitcode ABI. Cilkscale implements a parallel version of the Cilkview algorithm [33] to analyze the ***parallelism*** — potential for parallel speedup — of a fork-join parallel program. To implement this algorithm, Cilkscale maintains state variables for each spawned function. Cilkscale uses the instrumentation hooks Momme inserts around Tapir instructions to update these variables whenever the program-under-test performs a spawn or sync.

We developed ***Cilkscale-bc***, which implements this algorithm efficiently using a bitcode ABI. Cilkscale-bc defines a tool-specific stack structure to store state variables for each function frame. Momme uses this bitcode ABI to allocate

a tool structure in each function frame, thereby allowing Cilkscale-bc to maintain the necessary state variables directly on the call stack of the program-under-test.

We compared Cilkscale-bc to ***Cilkscale-lib***, which implements Cilkscale as a standard library that maintains the state variables in a separate shadow stack on the heap. Empirically, we observed that the benchmarks run a geometric mean of 3% faster when using Cilkscale-bc compared to using Cilkscale-lib, and 6 of the benchmarks running at least 5% faster serially and in parallel. Figure 9 in Appendix A shows the full performance results.

Making Cilksan more flexible

We explored another use of bitcode ABIs in the context of Cilksan. To instrument all memory accesses in a program-under-test, Cilksan uses hooks for compiler intrinsics that encode architecture-specific operations, such as vector scatter and gather operations. Such intrinsics are likely to proliferate as architectures evolve, which creates a challenge for Cilksan to keep up with this ever-growing set of intrinsics.

To gracefully handle cases where Cilksan does not implement a hook for an intrinsic, Cilksan implements a ***placeholder hook***, which emits a warning to inform the user that an intrinsic is not instrumented, but allows race-detection to continue on the rest of the program. To insert calls to the placeholder hook correctly, Momme must recognize when the program-under-test uses an intrinsic that Cilksan does not instrument. Although one can achieve this functionality using weak symbols [14, p. 680], support for weak symbols varies, for example, between macOS and Linux and between static and dynamic libraries. Accommodating this varying support for weak symbols across systems would complicate the implementation of Momme and Cilksan.

Cilksan uses a bitcode ABI to achieve this capability of weak symbols in a system-agnostic fashion. Cilksan's bitcode ABI identifies all of the instrumentation hooks that Cilksan defines. When Momme instruments a program-under-test, it checks whether Cilksan's bitcode ABI defines a hook for each intrinsic it encounters and inserts a call to either the corresponding hook or the placeholder hook. Because Momme implements this mechanism within the OpenCilk compiler, it works across different systems and types of linking.

4 Empirical evaluation

This section presents our empirical evaluation of OpenCilk. Previous work demonstrated the performance advantages of compiling with Tapir [64]. This evaluation compares the four runtime back ends that OpenCilk supports and compares OpenCilk-DPRNG to Cilk Plus.

Experimental setup

All tests were run on an AWS `c5.metal` instance running Fedora 36, which contains two 3GHz Intel Xeon Platinum 8275CL processor chips with 24 processor cores per chip and 192GB of main memory. Each processor core contains private 32KB L1 instruction and data caches and a private 1MB L2 cache. All processor cores on a chip share a 35.75MB L3 cache. To reduce performance variability, Intel Turbo Boost, Hyperthreading, and Linux NUMA balancing were disabled. All benchmarks were compiled with `-O3` optimizations using the OpenCilk compiler, which is based on LLVM 14.0.6, and linked with TCMalloc [18]. All benchmarks were run on 1 core (T_1), 24 cores on 1 chip (T_{24}), and all 48 cores (T_{48}). All running times were measured in seconds and aggregated as the median of 20 runs.

Benchmark	Input	Description
`cholesky`	4k×4k matrix, 8k nnz	Cholesky decomposition
`cilksort`	80M elements	Parallel merge sort
`fft`	20M elements	Fast Fourier transform
`heat`	4k×4k grid, 200 steps	Heat diffusion stencil
`lu`	4k×4k matrix	LU decomposition
`matmul`	2k×2k matrix	Square matmul
`nqueens`	$n = 13$	n-Queens solver
`qsort`	50M elements	Hoare quicksort
`rectmul`	4k×4k matrix, 4k×2k matrix	Rectangular matmul
`strassen`	4k×4k matrix	Strassen matmul
`bfs`	`com-Orkut`	Breadth-first search
`kcore`	`com-Orkut`	k-Core decomposition
`triangle`	`com-Orkut`	Triangle counting
`pagerank`	`com-Orkut`	PageRank algorithm
`minife`	150×150×150 box	Finite elements
`fibrng`	$n = 40$	Sum F_n random numbers
`pi`	100M samples	Monte Carlo π estimation
`mis`	`com-Orkut`	Maximal independent set
`sf`	`com-Orkut`	Spanning forest

Figure 4. Benchmark programs used in evaluation. The middle line separates non-randomized benchmarks (above) from randomized benchmarks (below). The abbreviations "nnz" and "matmul" stand for "number of nonzeros" and "matrix multiplication," respectively. The `com-Orkut` graph contains 3M vertices and 117M edges. The notation F_n denotes the nth Fibonacci number.

To evaluate performance, we assembled a suite of Cilk benchmark programs from a variety of sources, including the MIT Cilk benchmark suite [29], the Graph Based Benchmark Suite [24], and a finite element miniapplication [56]. These benchmarks use a variety of fork-join parallel programming patterns and exhibit different performance properties, such as different compute intensities and parallelism. Figure 4 describes the benchmark suite.

Runtime-system performance comparison

We compared the performance of the OpenCilk, Cilk Plus, oneTBB, and LLVM OpenMP runtime systems. We used the OpenCilk compiler to compile each of the 15 non-randomized benchmarks to use different runtime systems via different Tapir targets. Barring one exception, all benchmarks were compiled identically until Tapir lowering, when different

Tapir targets were used to lower Tapir constructs to respective runtime ABIs.

We slightly modified how the OpenCilk compiler compiles executables that use the OpenMP target, in order to improve their parallel performance. To verify our OpenMP and oneTBB Tapir targets, we checked the performance of some programs using those targets against versions implemented using OpenMP directives or oneTBB directly. In doing so, we discovered a compiler optimization affecting `cholesky` that improves performance when using OpenCilk, Cilk Plus, or oneTBB but makes performance worse when using OpenMP. The optimization removes `spawn` statements that are immediately followed by a `sync`, because such `spawn` statements incur scheduling overhead without increasing parallelism. But this optimization turns out to substantially hurt the parallel performance of `cholesky` when using OpenMP. We therefore disabled this optimization when compiling the benchmarks to use the OpenMP target.

Figure 5 presents the performance of the benchmarks using the different runtime systems. As Figure 5 shows, the OpenCilk runtime system almost always matches or exceeds the performance of the other runtime systems. In the few cases where another runtime system performs better, the performance differences between executables are small and often within the noise. On 1 core, the OpenCilk runtime runs a geometric mean of 4% faster than the Cilk Plus runtime and 26% faster than the LLVM OpenMP and oneTBB runtimes.[7] The OpenCilk runtime shows greater improvements on 48 cores, which corresponds to running the executables on both NUMA nodes. On 48 cores, the OpenCilk runtime runs a geometric mean of 1.11×, 1.56×, and 2.22× faster than the Cilk Plus, oneTBB, and OpenMP runtimes, respectively.

We speculate that OpenCilk's performance advantage over OpenMP and oneTBB comes in part from the differences in runtime ABIs and in part from the scheduling policy. First, the OpenCilk runtime ABI enables better work efficiency than the OpenMP and oneTBB runtime ABIs. Whereas the OpenCilk and Cilk Plus runtime ABIs pass arguments to a spawned task as ordinary function arguments, the OpenMP and oneTBB ABIs require the compiler to marshal that data into a separate structure. Avoiding such marshalling and unmarshalling of data leads to better work efficiency. Second, whereas OpenCilk and Cilk Plus implement continuation stealing, OpenMP and oneTBB implement child-stealing, which changes the locality behavior of a program's parallel execution compared to that of its serial projection. Suppose a function `F` spawns another function `G`. Whereas OpenCilk and Cilk Plus execute `G` upon encountering the `spawn`, OpenMP and oneTBB marshal `G`'s arguments, mark `G` as available for stealing, and then execute the continuation of `F` before executing `G`. This change in execution order changes the data locality of the program, which can hurt the performance of a program with a well-optimized serial projection. In addition, when combined with the marshalling of data, child stealing can exacerbate locality issues of parallel executions, as the thread that marshals `G`'s arguments is likely not the thread that executes `G` during a parallel execution.

We found that hardware-counter measurements support our speculation. On benchmarks where OpenCilk exhibits better work efficiency, running on OpenCilk generally leads to lower instruction counts, branch miss rates, and cache miss rates compared to running on OpenMP or oneTBB, even during parallel runs. On benchmarks where the runtimes demonstrate similar work efficiency but OpenMP and oneTBB exhibit worse parallel execution times, we find that the parallel executions under OpenMP and oneTBB incur more last-level cache references and misses.

We have identified two sources of the improved performance of the OpenCilk runtime system compared to Cilk Plus. First, the OpenCilk runtime system implements several optimizations for modern multicore systems. The OpenCilk runtime optimizes the data structures and functions both in its ABI, in accordance with the work-first principle [29], and in its work-stealing implementation, to mitigate the performance impact of memory accesses of threads who are not executing work. The OpenCilk runtime's use of a bitcode ABI enabled us to rapidly develop and test these optimizations. Second, the Cilk Plus runtime implements pedigrees by default, to support DPRNGs [47], whereas the OpenCilk runtime does not. We separately evaluate the performance impact of adding DPRNG support to OpenCilk.

Performance of DPRNG support

We studied the performance of OpenCilk-DPRNG — which adds DPRNG support to the OpenCilk runtime — both on the non-randomized benchmark programs and on the 4 randomized parallel benchmarks in Figure 4, which generate pseudorandom numbers in parallel. Figures 7 and 8 in Appendix A present the running-time results.

On the non-randomized benchmarks, OpenCilk-DPRNG runs a geometric mean of 2% slower than the unmodified OpenCilk runtime and approximately 5% faster than Cilk Plus. Compared to the unmodified OpenCilk runtime, OpenCilk-DPRNG exhibits a maximum slowdown of 18% on `cholesky`, whose running time is particularly sensitive to scheduling overheads. Meanwhile, OpenCilk-DPRNG runs a geometric mean of 2% faster than Cilk Plus on 1 core. OpenCilk-DPRNG therefore exhibits both higher work efficiency and better parallel scalability than Cilk Plus. Notably, while Cilk Plus only implements pedigrees, OpenCilk-DPRNG runs faster than Cilk Plus while implementing both pedigrees and a builtin DPRNG.

We also evaluated the performance of OpenCilk-DPRNG on 4 benchmarks that generate pseudorandom numbers in

[7] These speedups are computed as the geometric mean over all benchmarks of $T_1(\texttt{R})/T_1(\texttt{OpenCilk})$, where $T_1(\texttt{R})$ denotes the 1-core running time of an executable that uses runtime system `R`.

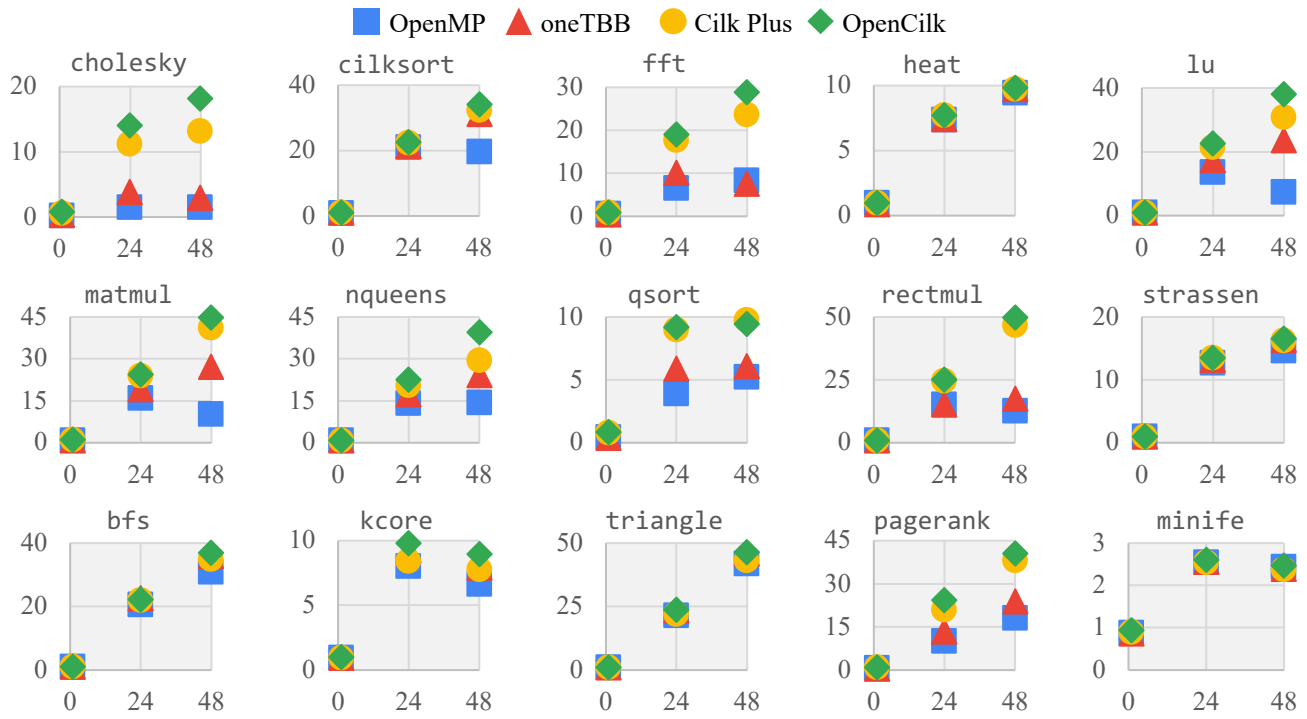

Figure 5. Performance of 15 benchmark Cilk programs compiled and run using the OpenMP, oneTBB, Cilk Plus, and OpenCilk runtime back ends. In each plot, the x-axis denotes the number P of processor cores, and the y-axis denotes speedup T_S/T_P over the running time T_S of the serial projection. Figure 6 in Appendix A presents the running-time data for these plots.

parallel. We compared three DPRNG implementations on these programs: 1. Cilk Plus with DotMix, a DPRNG library that uses pedigrees [47]; 2. OpenCilk-DPRNG with DorMix; and 3. OpenCilk-DPRNG using its builtin DPRNG. Overall, when using the DotMix DPRNG library, these benchmarks run a geometric mean of 3% faster when using OpenCilk-DPRNG compared to Cilk Plus. We also find these benchmarks run 85% faster when using the builtin DPRNG instead of DotMix. The `fibrng` and `pi` benchmarks show the largest performance improvements, because their running times are dominated by the time to generate pseudorandom numbers.

5 Related work

Languages and libraries for task parallelism. Mainstream support for task parallelism has been explored primarily through two interfaces: libraries (e.g., [10, 20, 26, 43, 45, 61]) and languages (e.g., [7, 15, 16, 23, 29, 46, 55]). Library solutions avoid the need for compiler support and are thus simpler to modify and extend. But previous work has found that supporting parallelism through a library can jeopardize the ability to write simple, correct, and efficient parallel programs [13, 32]. In contrast, language solutions can leverage the compiler to ensure that parallel semantics are understood and preserved through program optimizations. But existing implementations require details of a particular runtime system to be embedded within the compiler codebase, making them more difficult to modify or extend. OpenCilk provides language and compiler support for task parallelism while capturing the benefits of library solutions in being easier to modify and extend. By employing bitcode ABIs, OpenCilk minimizes the amount of runtime-specific information that must be hard-coded into the compiler's codebase.

Extensions to Tapir. The original work on Tapir implemented a single front end and back end (i.e., Cilk Plus). Subsequent research has explored alternative front ends and back ends to Tapir/LLVM. Stelle et al. developed a front end for OpenMP's `task` and `taskwait` pragmas [54]. Schardl and Samsi modified TensorFlow's XLA compiler to use Tapir/LLVM and thus enable parallel tasks in TensorFlow applications to run on the Cilk Plus runtime [65]. Shajii et al. introduce the Seq language for bioinformatics, which uses Tapir/LLVM to compile and run parallel tasks on a custom OpenMP back end for Tapir/LLVM [66]. Margerm et al. developed TAPAS, a hardware-synthesis tool that generates parallel accelerators for task-parallel programs expressed using Tapir. OpenCilk can benefit these works by allowing new front ends to run on multiple runtime back ends and new back ends to be implemented more easily, using the Tapir-lowering framework.

Uses of LLVM bitcode. LLVM uses bitcode files to support link-time optimization (LTO) [40, 57]. GraalVM [22] uses LLVM bitcode to execute programs written in different source languages on the GraalVM runtime. NVIDIA distributes the `libdevice` library as part of CUDA to provide NVVM bitcode, which is based on LLVM bitcode, for common functions for NVIDIA GPU's [21]. When compiling

a CUDA program, the compiler links in `libdevice` early, allowing it to optimize these functions and their uses. Compared to prior work, OpenCilk's use of bitcode is more targeted: to simplify and relax how an ABI is defined inside the compiler. The OpenCilk compiler reads a bitcode file to internalize definitions of a parallel-runtime ABI before implementing that ABI in a compiled program. Therefore, bitcode-ABI files not only provide compiler-optimization opportunities, but they allow the ABIs of parallel runtime systems and tools to be changed without modifying the compiler's codebase and without harming performance.

Tool-development infrastructure for parallel code. OpenMP 5.0 [55] provides the OMPT and OMPD interfaces to enable the development of tools for analyzing and debugging OpenMP programs. OMPT supports tools that are linked directly with a target OpenMP program [27], whereas OMPD supports tools that run as separate processes. Both OMPT and OMPD are defined in terms of the OpenMP standard and are tied to conformant implementations of OpenMP. Cilk Plus supported low-overhead tool annotations [37] that augment the binary of a compiled Cilk Plus program to identify points of interest, such as around a `cilk_spawn` or `cilk_sync`. These annotations were necessary because the ABIs for these constructs is more complex than simple runtime-function calls. The Pin binary-instrumentation framework [50] was modified to recognize these annotations, allowing tools including Cilkview [33] and Cilkscreen [19] to instrument a Cilk Plus program using binary instrumentation. Although the low-overhead tool annotation mechanism was general purpose, its use for Cilk Plus programs was closely tied to the Cilk Plus ABI. In addition, problems arose with tools when Pin and the compiler were not updated at the same time (e.g., [17]). In contrast, Momme is integrated into the OpenCilk compiler and its instrumentation is not tied to a particular parallel language or runtime ABI.

6 Conclusion

The design of OpenCilk takes inspiration from software tool chains for sequential programming. As higher-level language abstractions for serial programming evolved, compilers evolved from monolithic pieces of software, built to handle a single language and a single target platform, to flexible and modular designs that handle multiple languages and target platforms. This modular design means that extending the compiler to support a new language or target platform requires linear, not quadratic, amount of implementation work. Moreover, the compiler middle end provides a standard interface that enables rapid development with substantial code reuse and therefore advancement of compiler analyses and optimizations.

We see parallel programming going through a similar evolution. The abstraction has evolved, from manually manipulating operating-system threads to implement custom scheduling and load balancing, to using higher-level task-parallel control constructs to express an application's logical parallelism. The way we support compilation of the parallel abstractions should similarly evolve. The design of OpenCilk aims to provide a modular software infrastructure for task-parallel programming, similar to what today's compilers provide for serial programming.

OpenCilk has primarily focused on fork-join parallelism. For future work, we would like to add support for other more flexible forms of task parallelism, such as pipelining and futures. These parallel constructs support likely requires new parallel primitives within OpenCilk compiler's IR to accommodate the semantics of these other constructs.

Acknowledgments

Thanks to Tim Kaler of MIT for his help developing the bitcode-ABI functionality as well as the pedigree and builtin-DPRNG support in OpenCilk. Thanks to William S. Moses of MIT and George Stelle and Pat McCormick of Los Alamos National Lab, for their help developing and debugging the Tapir-lowering infrastructure and the OpenMP Tapir target. Thanks to John Carr for his help developing and debugging the OpenCilk compiler and runtime system. Thanks to our master bug-finder Brian Wheatman of Johns Hopkins University. Thanks to Charles E. Leiserson and Dorothy Curtis of MIT and Bradley Kuszmaul of Google for many useful discussions related to OpenCilk.

This research was supported in part by NSF Grants CCF-1910568 and CCF-1943456, Los Alamos National Laboratory subcontract 531711, the United States Air Force Research Laboratory, under Cooperative Agreement Number FA8750-19-2-1000, and the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003965.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force, the U.S. Government, or any agency thereof. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

References

[1] 2022. Rust Documentation: Crate tokio. urlhttps://docs.rs/tokio/latest/tokio/. Accessed in August 2022.

[2] 2022. A Tour of Go: Goroutines. urlhttps://go.dev/tour/concurrency/1. Accessed in August 2022.

[3] GCC 4.9. 2014. GCC 4.9 Release Series Changes, New Features, and Fixes. Available at https://gcc.gnu.org/gcc-4.9/changes.html.

[4] Umut A. Acar, Vitaly Aksenov, Arthur Charguéraud, and Mike Rainey. 2019. Provably and Practically Efficient Granularity Control. In *PPoPP*. ACM, 214—-228.

[5] Umut A. Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski. 2018. Heartbeat Scheduling: Provable Efficiency for Nested Parallelism. In *PLDI*. ACM, 769—-782.

[6] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling Parallel Programs by Work Stealing with Private Deques. In *PPoPP*. 219—-228.

[7] Rajkishore Barik, Zoran Budimlić, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. 2009. The Habanero Multicore Software Research Project. In *OOPSLA*. 735–736.

[8] Rajkishore Barik and Vivek Sarkar. 2009. Interprocedural Load Elimination for Dynamic Optimization of Parallel Programs. In *PACT*. 41–52.

[9] Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. 2013. Interprocedural Strength Reduction of Critical Sections in Explicitly-Parallel Programs. In *PACT*. 29–40.

[10] David A. Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, and Thomas RW Scogland. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *P3HPC*. 71–81.

[11] Guy E. Blelloch and Margaret Reid-Miller. 1997. Pipelining with futures. In *SPAA*. ACM, 249–259.

[12] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *JACM* 46, 5 (1999), 720–748.

[13] Hans-J. Boehm. 2005. Threads Cannot Be Implemented as a Library. *SIGPLAN Not.* 40, 6 (jun 2005), 261–268.

[14] Randal E. Bryant and David R. O'Hallaron. 2015. *Computer Systems: A Programmer's Perspective* (3rd ed.). Pearson, USA.

[15] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: the new adventures of old X10. In *PPPJ*. 51–61.

[16] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *OOPSLA*. 519–538.

[17] Intel Comunities. 2012. Problems with cilkprof. Available from https://community.intel.com/t5/Software-Archive/Problems-using-cilkprof/m-p/741804.

[18] Google Corporation. 2022. TCMalloc. https://google.github.io/tcmalloc/. Accessed August 2022.

[19] Intel Corporation. 2011. Intel Cilk Plus Software Development Kit. Available at http://software.intel.com/en-us/articles/intel-cilk-plus-software-development-kit/.

[20] Microsoft Corporation. 2021. Parallel Patterns Library (PPL). Available from https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl.

[21] NVIDIA Corporation. 2022. Libdevice User's Guide. https://docs.nvidia.com/cuda/libdevice-users-guide/index.html. Accessed August 2022.

[22] Oracle Corporation. 2022. GraalVM. https://www.graalvm.org/. Accessed August 2022.

[23] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. 2008. Programming with exceptions in JCilk. *Science of Computer Programming* 63, 2 (Dec. 2008), 147–171.

[24] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2021. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. *ACM TOPC* 8, 1, Article 4 (apr 2021), 70 pages.

[25] Johannes Doerfert and Hal Finkel. 2018. Compiler Optimizations for OpenMP. In *Evolving OpenMP for Evolving Architectures*. 113–127.

[26] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *JPDC* 74, 12 (2014), 3202–3216.

[27] Alexandre E. Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. 2013. OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis. In *OpenMP in the Era of Low Power Devices and Accelerators*. 171–185.

[28] Mingdong Feng and Charles E. Leiserson. 1999. Efficient Detection of Determinacy Races in Cilk Programs. *Theory of Computing Systems* 32, 3 (1999), 301–326.

[29] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*. 212–223.

[30] GCC Team. 2015. GOMP — An OpenMP Implementation for GCC. Available at https://gcc.gnu.org/projects/gomp/.

[31] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. 2009. Work-first and Help-first Scheduling Policies for Async-finish Task Parallelism. In *IPDPS*. 1–12.

[32] Pablo Halpern. 2013. *Considering a Fork-Join Parallelism Library*. Technical Report N3557. Intel Corporation. Available from https://isocpp.org/files/papers/n3557.pdf.

[33] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. 2010. The Cilkview Scalability Analyzer. In *SPAA*. 145–156.

[34] Shams Imam and Vivek Sarkar. 2014. Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. In *ECOOP*. 618–643.

[35] Shams M. Imam and Vivek Sarkar. 2012. Integrating Task Parallelism with Actors. In *OOPSLA*. 753–772.

[36] Intel 2012. Intel® Cilk™ Plus Language Extension Specification. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1665.htm. Accessed in August 2022.

[37] Intel Corporation. 2011. Intrinsics for Low Overhead Tool Annotations. Document Number 326357-001US. Available from http://web.archive.org/web/20150908043715/https://www.cilkplus.org/sites/default/files/open_specifications/LowOverheadAnnotations.pdf.

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

[39] Intel Corporation 2023. *Intel(R) oneAPI Threading Building Blocks (oneTBB) Documentation*. Intel Corporation. Available from https://www.intel.com/content/www/us/en/develop/documentation/onetbb-documentation/top.html.

[40] Teresa Johnson, Mehdi Amini, and Xinliang David Li. 2017. ThinLTO: Scalable and incremental LTO. In *CGO*. 111–121.

[41] Chris Lattner. 2002. *LLVM: An Infrastructure for Multi-Stage Optimization*. Master's thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL. *See* `http://llvm.cs.uiuc.edu`..

[42] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. 75.

[43] Doug Lea. 2000. A Java fork/join framework. In *ACM 2000 Conference on Java Grande*. 36–43.

[44] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. 2010. Using Memory Mapping to Support Cactus Stacks in Work-Stealing Runtime Systems. In *PACT*. ACM, 411–420.

[45] Daan Leijen and Judd Hall. 2007. Optimize Managed Code For Multi-Core Machines. *MSDN Magazine* (2007). Available from http://msdn.microsoft.com/magazine/.

[46] Charles E. Leiserson. 2010. The Cilk++ Concurrency Platform. *J. Supercomputing* 51, 3 (2010), 244–257.

[47] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. 2012. Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms. In *PPoPP*.

[48] LLVM Project. 2015. OpenMP: Support for the OpenMP Language. Available at http://openmp.llvm.org/.

[49] Li Lu, Weixing Ji, and Michael L. Scott. 2014. Dynamic Enforcement of Determinism in a Parallel Scripting Language. In *PLDI*. 519–529.

[50] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*. 190–200.

[51] Steven Margerm, Amirali Sharifian, Apala Guha, Arrvindh Shriraman, and Gilles Pokam. 2018. TAPAS: Generating Parallel Accelerators from Parallel Programs. In *MICRO*. 245–257.

[52] Stefan K. Muller and Umut A. Acar. 2016. Latency-Hiding Work Stealing: Scheduling Interacting Parallel Computations with Work Stealing. In *SPAA*. 71–82.

[53] Stefan K. Muller, Umut A. Acar, and Robert Harper. 2018. Competitive Parallelism: Getting Your Priorities Right. In *ICFP*. 95:1–95:30.

[54] OpenMP 4.5 2015. *OpenMP Application Program Interface, Version 4.5.*

[55] OpenMP 5.0 2018. *OpenMP API 5.0 Specification.*

[56] Mantevo Organization. 2022. miniFE Finite Element Mini-Application. Available from https://github.com/Mantevo/miniFE. Accessed August 2022.

[57] LLVM Project. 2022. LLVM Link Time Optimization: Design and Implementation. https://llvm.org/docs/LinkTimeOptimization.html. Accessed August 2022.

[58] LLVM Project. 2022. My First Language Frontend with LLVM Tutorial. https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html. Accessed August 2022.

[59] LLVM Project. 2023. Clang 14.0.0 documentation, OpenMP Support. https://releases.llvm.org/14.0.0/tools/clang/docs/OpenMPSupport.html. Accessed January 2023.

[60] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *PLDI*. 531–542.

[61] James Reinders. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* O'Reilly Media, Inc.

[62] 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. *ACM POMACS* 1, 2, Article 43 (Dec. 2017), 25 pages.

[63] Tao B. Schardl, Bradley C. Kuszmaul, I-Ting Angelina Lee, William M. Leiserson, and Charles E. Leiserson. 2015. The Cilkprof Scalability Profiler. In *SPAA*. 89–100.

[64] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2019. Tapir: Embedding Recursive Fork-Join Parallelism into LLVM's Intermediate Representation. *ACM TOPC* 6, 4, Article 19 (Dec. 2019), 33 pages.

[65] Tao B. Schardl and Siddharth Samsi. 2019. TapirXLA: Embedding Fork-Join Parallelism into the XLA Compiler in TensorFlow Using Tapir. In *HPEC*. 1–8.

[66] Ariya Shajii, Ibrahim Numanagić, Riyadh Baghdadi, Bonnie Berger, and Saman Amarasinghe. 2019. Seq: A High-Performance Language for Bioinformatics. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 125 (oct 2019), 29 pages.

[67] Shumpei Shiina and Kenjiro Taura. 2019. Almost Deterministic Work Stealing. In *SC*.

[68] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. 2008. Phasers: A Unified Deadlock-free Construct for Collective and Point-to-point Synchronization. In *ICS*. 277–288.

[69] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. 2009. Phaser accumulators: A new reduction construct for dynamic parallelism. In *IPDPS*. 1–12.

[70] Kyle Singer, Kunal Agrawal, and I-Ting Angelina Lee. 2020. Scheduling I/O Latency-Hiding Futures in Task-Parallel Platforms. In *APoCS*. 147–161.

[71] Kyle Singer, Noah Goldstein, Stefan K. Muller, Kunal Agrawal, I-Ting Angelina Lee, and Umut A. Acar. 2020. Priority Scheduling for Interactive Applications. In *SPAA*. 465–477.

[72] Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. 2019. Proactive Work Stealing for Futures. In *PPoPP*. 257–271.

[73] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. 2009. Beyond Nested Parallelism: Tight Bounds on Work-stealing Overheads for Parallel Futures. In *SPAA*. 91–100.

[74] Amitabh Srivastava and David W. Wall. 1992. *A Practical System for Intermodule Code Optimization at Link-Time.* Technical Report 92/6. Digital Western Research Laboratory.

[75] Guy L. Steele, Doug Lea, and Christine H. Flood. 2014. Fast Splittable Pseudorandom Number Generators. In *OOPSLA*, Vol. 49. 453—-472.

[76] Olivier Tardieu, Haichuan Wang, and Haibo Lin. 2012. A Work-stealing Scheduler for X10's Task Parallelism with Suspension. In *PPoPP*. 267–276.

[77] Sağnak Taşırlar and Vivek Sarkar. 2011. Data-Driven Tasks and Their Implementation. In *ICPP*. 652–661.

[78] Peter Thoman, Peter Zangerl, and Thomas Fahringer. 2017. Task-Parallel Runtime System Optimization Using Static Compiler Analysis. In *CF*. 201––210.

[79] Robert Utterback, Kunal Agrawal, Jeremy Fineman, and I-Ting Angelina Lee. 2016. Provably Good and Practically Efficient Parallel Race Detection for Fork-Join Programs. In *SPAA*. 83–94.

[80] Robert Utterback, Kunal Agrawal, I-Ting Angelina Lee, and Milind Kulkarni. 2017. Processor-Oblivious Record and Replay. In *PPoPP*. 145–161.

[81] Yifan Xu, I-Ting Angelina Lee, and Kunal Agrawal. 2018. Efficient Parallel Determinacy Race Detection for Two-dimensional Dags. In *PPoPP*. 368–380.

[82] Yifan Xu, Kyle Singer, and I-Ting Angelina Lee. 2020. Parallel Determinacy Race Detection for Futures. In *PPoPP*. 217––231.

[83] Victor A. Ying, Mark C. Jeffrey, and Daniel Sanchez. 2020. T4: Compiling Sequential Code for Effective Speculative Parallelization in Hardware. In *ISCA*. 159–172.

[84] Adarsh Yoga and Santosh Nagarakatte. 2016. Atomicity Violation Checker for Task Parallel Programs. In *CGO*. 239–249.

[85] Adarsh Yoga and Santosh Nagarakatte. 2017. A Fast Causal Profiler for Task Parallel Programs. In *ESEC/FSE*. 15—-26.

[86] Adarsh Yoga and Santosh Nagarakatte. 2019. Parallelism-Centric What-If and Differential Analyses. In *PLDI*. 485—-501.

[87] Adarsh Yoga, Santosh Nagarakatte, and Aarti Gupta. 2016. Parallel Data Race Detection for Task Parallel Programs with Locks. In *FSE*. 833—-845.

A Artifact description

This appendix describes how to setup OpenCilk and how to download and run the application benchmarks for the empirical evaluation described in this paper. The OpenCilk source code used for this paper, including its LLVM-based compiler, runtime system, and productivity tools, are available at https://doi.org/10.5281/zenodo.7510908.

For convenience, OpenCilk and the application benchmarks described in Figure 4 are distributed via a Docker image for x86-64 processors. This appendix provides a summary overview of how to use this Docker image to replicate the empirical evaluation. The source code for the application benchmarks, along with detailed instructions for using the Docker image, are available at https://github.com/neboat/opencilk-ppopp-23-ae/. See the `README.md` file at https://github.com/neboat/opencilk-ppopp-23-ae/ for detailed instructions on using this Docker image to run the empirical evaluation.

Running-time results

The steps described in this appendix to replicate the empirical evaluation produce four CSV files. These files contain aggregated running times for the experiments described in Sections 3 and 4. Figures 6, 7, 8 and 9 show the running-time data underlying the empirical results described in this paper.

- Figure 6 presents the running-time data for the non-randomized benchmarks when compiled to use either the OpenCilk runtime system, the Cilk Plus runtime system, the OpenMP runtime system in LLVM 14, or oneTBB. These running times are used to generate the plots in Figures 1 and 5.
- Figure 7 presents the running-time data for the performance comparison, described in Section 4, between OpenCilk-DPRNG and Cilk Plus on the 15 non-randomized benchmarks.
- Figure 8 presents the running-time data for the performance comparison, described in Section 4, on the 4 randomized benchmarks between Cilk Plus with the DotMix DPRNG library [47], OpenCilk-DPRNG with DotMix, and OpenCilk-DPRNG using its builtin DPRNG.
- Figure 9 presents the running-time data for Cilkscale-bc and Cilkscale-lib on the 15 non-randomized benchmarks. These running times underlie the performance comparison, described in Section 3, between these two implementations of Cilkscale.

Although we cannot guarantee that you can exactly replicate the running times presented in these figures — due to differences in computer hardware and system configuration — you can follows these steps to observe the general functionality of OpenCilk and similar overall performance trends.

Summary usage guide

The following steps overview how to setup and use the Docker image to run the experiments described in this paper. These instructions are intended to be run on an x86-64 multicore machine running a contemporary version of Linux with Docker installed.

Step 1: Get the image. Download and run the Docker image as follows:

```
$ wget https://people.csail.mit.edu/neboat/\
> opencilk-ppopp-23/docker-opencilk-ppopp-23.tar.gz
$ docker load -i docker-opencilk-ppopp-23.tar.gz
$ docker run -it opencilk-ppopp-23 /bin/bash
```

Step 2: Run the tests. Enter the test directory inside the Docker container and run the empirical tests using the `run_tests.py` script:

```
$ cd /usr/local/src/ppopp-23-ae
$ python3 ./run_tests.py
```

Running `run_tests.py` without any command-line arguments will run all benchmark programs 10 times each and run the parallel executables only on the number of CPU cores on the system (excluding hyperthreads and SMT). Although this default behavior does not match the experimental setup (i.e., number of trials and CPU counts) described in Section 4, it runs the empirical tests more quickly in a manner that nonetheless supports evaluation of OpenCilk. On a system with at least 48 cores, you can pass the command-line flags `-t 20 -c 1,24,48` to `run_tests.py` to run all benchmark programs with experimental setup described in Section 4. For more information on accepted command-line flags, run `run_tests.py` with the `-h` flag.

Step 3: Examine the CSV files. Exit the Docker container, and then use the `docker cp` command to copy the CSV files of aggregate results out of the Docker container. Examine the CSV files in a spreadsheet program.

When the `run_tests.py` script is run to perform all experiments, it generates four CSV files containing aggregated performance results. Each CSV file is named with a ***run tag*** that identifies the year, month, day, hour, and minute of the corresponding invocation of `run_tests.py`. On completion, `run_tests.py` reports the run tag for its run.

These CSV files correspond with Figures 6, 7, 8 and 9 as follows, where `<tag>` denotes a run tag:

- The `baseline-<tag>.csv` file contains running-time results that correspond with Figure 6.
- The `pedigrees-<tag>.csv` file contains running-time results that correspond with Figure 7.
- The `dprng-<tag>.csv` file contains running-time results that correspond with Figure 8.
- The `cilkscale-compare-<tag>.csv` file contains running-time results that correspond with Figure 9.

As in their corresponding figures, the running times reported in these CSV files are measured in seconds and aggregated as the median of the runs of that executable.

In these CSV files, rows identify different programs, and columns generally identify a system (e.g., Cilk Plus or OpenCilk) and CPU core count. The column heading

		OpenMP			oneTBB			Cilk Plus			OpenCilk		
Benchmark	T_S	T_1	T_{24}	T_{48}	T_1	T_{24}	T_{48}	T_1	T_{24}	T_{48}	T_1	T_{24}	T_{48}
cholesky	2.745	13.343	1.788	1.898	9.962	0.709	0.926	4.406	0.244	0.209	**3.592**	**0.195**	**0.151**
cilksort	9.483	9.701	0.450	0.481	9.608	0.442	0.303	9.232	0.425	0.293	**9.230**	**0.420**	**0.277**
fft	5.563	10.209	0.845	0.666	10.534	0.547	0.726	6.563	0.314	0.235	**6.220**	**0.293**	**0.193**
heat	5.776	5.859	0.781	0.612	6.393	0.776	0.592	**5.887**	**0.750**	0.594	5.981	**0.750**	**0.589**
lu	12.114	12.726	0.888	1.636	12.727	0.693	0.513	12.083	0.565	0.392	**12.047**	**0.537**	**0.318**
matmul	6.645	6.715	0.413	0.642	6.770	0.345	0.245	**6.599**	0.277	0.161	6.617	**0.271**	**0.148**
nqueens	3.212	4.117	0.226	0.223	4.144	0.186	0.133	3.455	0.156	0.109	**3.326**	**0.142**	**0.081**
qsort	5.300	12.350	1.362	1.009	13.450	0.901	0.871	6.604	0.590	**0.543**	**6.014**	**0.576**	0.559
rectmul	10.375	11.081	0.680	0.823	11.565	0.679	0.591	10.567	0.423	0.222	**10.471**	**0.414**	**0.208**
strassen	9.027	9.057	0.713	0.615	9.091	0.682	0.551	9.096	0.670	0.555	**9.029**	**0.666**	**0.544**
bfs	0.641	0.650	0.031	0.021	**0.630**	**0.028**	0.018	0.648	0.029	0.018	0.646	0.029	**0.017**
kcore	9.788	10.613	1.225	1.473	10.661	1.058	1.234	10.227	1.166	1.263	**10.079**	**0.999**	**1.094**
triangle	139.815	151.200	6.495	3.338	140.940	5.945	3.050	150.129	6.337	3.239	**139.269**	**5.886**	**3.014**
pagerank	54.926	83.279	5.496	3.055	97.648	4.100	2.316	65.835	2.615	1.439	**60.357**	**2.259**	**1.356**
minife	42.623	48.251	16.743	17.544	49.019	16.712	17.878	47.410	16.726	17.922	**45.070**	**16.353**	**17.287**

Figure 6. Running times of 15 non-randomized benchmarks when compiled and run using the OpenMP, oneTBB, Cilk Plus, and OpenCilk runtime back ends. The T_S column presents the running time of each benchmark's serial projection. The best running time among the runtime back ends is bold-faced. The plots in Figures 1 and 5 are derived from these running times.

		fibrng	pi	mis	sf
T_1	Cilk Plus, DotMix	22.211	14.224	3.915	2.821
	OpenCilk-DPRNG, DotMix	22.874	14.985	3.863	**2.734**
	OpenCilk-DPRNG, Builtin	**6.293**	**4.652**	**3.701**	2.779
T_{24}	Cilk Plus, DotMix	0.942	0.695	0.172	0.126
	OpenCilk-DPRNG, DotMix	0.949	0.636	0.171	**0.117**
	OpenCilk-DPRNG, Builtin	**0.265**	**0.197**	**0.163**	0.120
T_{48}	Cilk Plus, DotMix	0.483	0.421	0.105	0.073
	OpenCilk-DPRNG, DotMix	0.475	0.334	0.105	**0.067**
	OpenCilk-DPRNG, Builtin	**0.133**	**0.100**	**0.097**	0.068

Figure 8. Performance of four randomized benchmark programs using Cilk Plus and the DotMix DPRNG, OpenCilk-DPRNG and the DotMix DPRNG, and OpenCilk-DPRNG and its builtin DPRNG.

	Cilkscale-lib			Cilkscale-bc		
Benchmark	T_1	T_{24}	T_{48}	T_1	T_{24}	T_{48}
cholesky	23.147	1.063	0.656	**21.108**	**0.964**	**0.597**
cilksort	**9.451**	**0.426**	**0.281**	9.644	0.434	0.283
fft	17.301	0.750	0.423	**16.282**	**0.704**	**0.399**
heat	6.032	0.766	0.610	**5.953**	**0.756**	**0.605**
lu	13.955	0.638	0.416	**13.775**	**0.628**	**0.405**
matmul	6.897	**0.279**	0.155	**6.871**	**0.279**	**0.154**
nqueens	**6.197**	**0.262**	**0.142**	6.208	**0.262**	**0.142**
qsort	23.747	1.247	0.784	**21.892**	**1.160**	**0.738**
rectmul	13.381	0.529	0.268	**13.053**	**0.517**	**0.261**
strassen	**9.310**	**0.676**	0.551	9.398	0.680	**0.545**
bfs	0.730	0.033	0.020	**0.656**	**0.029**	**0.018**
kcore	11.424	1.441	1.734	**11.020**	**1.344**	**1.603**
triangle	143.129	6.045	3.097	**140.633**	**5.943**	**3.044**
pagerank	123.439	5.174	2.675	**116.610**	**4.789**	**2.461**
minife	47.333	17.089	18.419	**46.093**	**16.745**	**18.074**

Figure 9. Performance of two implementations of Cilkscale, Cilkscale-lib and Cilkscale-bc.The better running time among the two libraries is bold-faced.

Benchmark	T_1		T_{24}		T_{48}	
cholesky	3.963	(1.10×)	0.214	(1.10×)	0.177	(1.18×)
cilksort	9.234	(1.00×)	0.419	(1.00×)	0.277	(1.00×)
fft	6.409	(1.03×)	0.301	(1.03×)	0.204	(1.06×)
heat	5.786	(0.97×)	0.755	(1.01×)	0.601	(1.02×)
lu	12.117	(1.01×)	0.539	(1.00×)	0.329	(1.03×)
matmul	6.655	(1.01×)	0.269	(0.99×)	0.149	(1.01×)
nqueens	3.361	(1.01×)	0.144	(1.01×)	0.083	(1.02×)
qsort	6.401	(1.06×)	0.583	(1.01×)	0.551	(0.98×)
rectmul	10.533	(1.01×)	0.413	(1.00×)	0.211	(1.01×)
strassen	9.107	(1.01×)	0.664	(1.00×)	0.541	(0.99×)
bfs	0.645	(1.00×)	0.029	(1.00×)	0.017	(1.00×)
kcore	10.031	(1.00×)	1.011	(1.01×)	1.112	(1.02×)
triangle	140.529	(1.01×)	5.941	(1.01×)	3.040	(1.01×)
pagerank	61.412	(1.02×)	2.414	(1.07×)	1.402	(1.03×)
minife	45.075	(1.00×)	16.332	(1.00×)	17.312	(1.00×)

Figure 7. Running times of non-randomized benchmarks when run using OpenCilk-DPRNG . Numbers in parentheses present the slowdown of the benchmark on a given core count compared to the OpenCilk runtime with no DPRNG support (shown in Figure 6).

`serial` 1 denotes executions on 1 CPU core of the serial projection of the Cilk program. In `dprng-<tag>.csv`, the column headings also identify the DPRNG used, i.e., DotMix or OpenCilk-DPRNG's builtin DPRNG. In `cilkscale-compare-<tag>.csv`, the keywords `cilkscale` and `cilkscale-bitcode` in the column headings indicate that the programs were compiled and run with the Cilkscale-lib or Cilkscale-bc, respectively.