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

I-Ting Angelina Lee
Washington University in St. Louis

FastCode @ HPEC
September 15, 2025

The Cilk Language

Cilk extends C/C++ with a small set of linguistic control constructs to support **task parallelism**.

```
int fib(int n) {
    if(n < 2) { return n; }
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
}
```

The named **child** function may execute in parallel with the continuation of its **parent**.

Control cannot pass this point until all spawned children have returned.

These keywords denote the **logical** parallelism of the computation and let an underlying scheduler automates scheduling and synchronization.

The Cilk Language, Continue

Cilk as well support **parallel loops** and **reducers**, a useful linguistic mechanism for avoiding **determinacy races [NM92,FL97]** in task-parallel code.

```
std::ofstream outf;
cilk::ostream_reducer<char> output(outf);
void print_numbers_to_file(int n) {
    outf.open("file.out");
    // Write to the ostream_reducer in parallel.
    cilk_for(int i = 0; i < n; ++i) {
        output << i << "\n";
    }
    outf.close();
}
```

Create an `ostream_reducer` that uses the output file stream.

reducer avoids determinacy race

same output as the sequential execution

Cilk's Performance Bound

Cilk uses a provably-efficient **work-stealing scheduler** to load-balance the computation.

Definition. T_P — execution time on P processors

T_1 — **work** T_∞ — **span** T_1 / T_∞ — **parallelism**

Theorem [BL94]. A work-stealing scheduler can achieve expected running time

$$T_P = T_1 / P + O(T_\infty)$$

on P processors.
$\Rightarrow$ **linear speedup** when $P \ll T_1 / T_\infty$

In Practice. Cilk's scheduler achieves execution time

$$T_P \approx T_1/P + T_\infty$$

on P processors.

OpenCilk

OpenCilk provides a new implementation of the Cilk language.

The **OpenCilk** system consists of a **compiler**, a **runtime-system library**, and a suite of **productivity tools**:

- **CilkSan**: a determinacy race detector
- **CilkScale**: a scalability analyzer and benchmarking tool

Talk Today: Who Should Use **OpenCilk** and Why

Who Should Use OpenCilk and Why

Who should use **OpenCilk**:

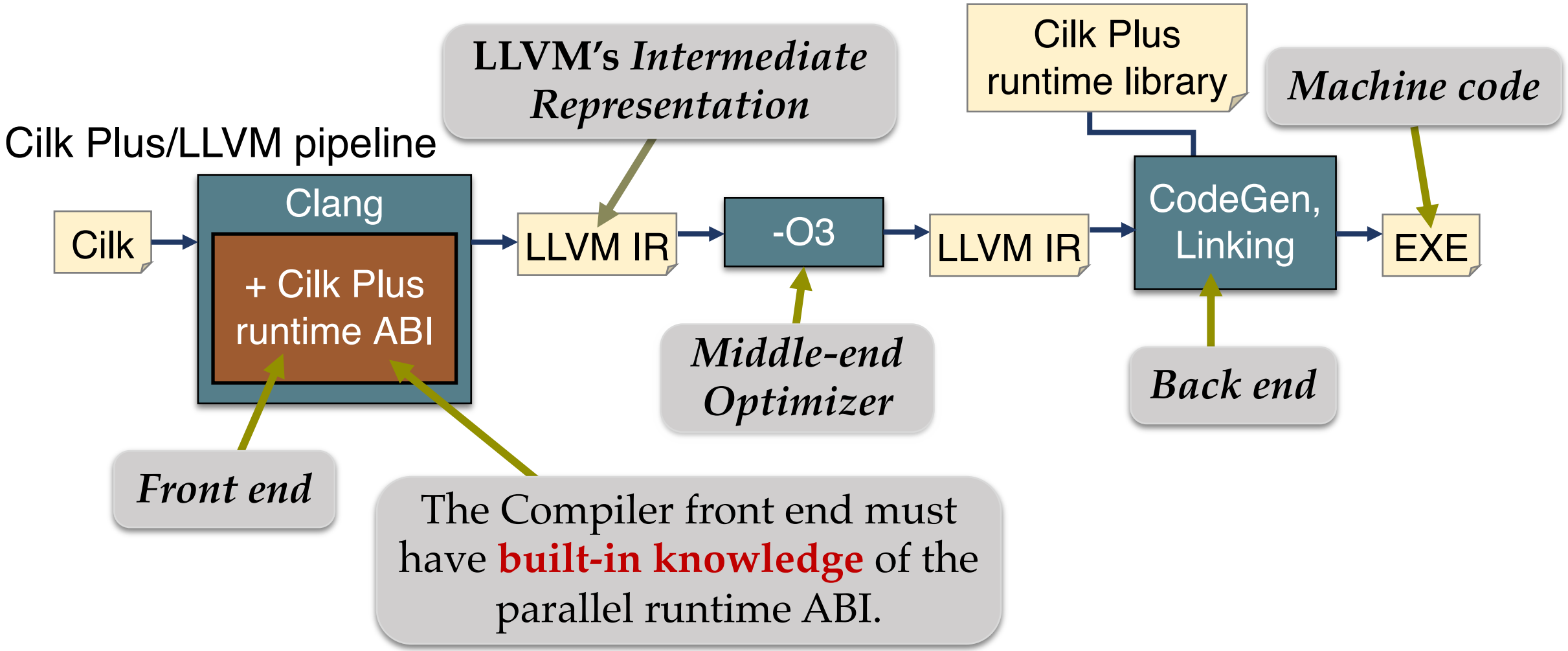
- **Educators** who teaches parallelism
- **Researchers** who wants to experiments with task-parallel platforms to:
  - build new **parallel language front end**
  - design and implement new **runtime features**
  - develop **tools** for task-parallel code
- **Application developers** who wants to write **fast task-parallel code** for multicore hardware

Why **OpenCilk**:

- The **linguistics** are simple and easy to understand.
- The runtime scheduler provides **provable execution time bounds**.
- The software infrastructure is **modular** and **extensible**.
- The overall system produces **fast code**.
- The system comes with a suite of **productivity tools**.

OpenCilk is Modular and Extensible.

The Traditional Way of Compiling Parallel Code



Other task-parallel systems, such as OpenMP or X10, use a similar design.

Example: The Cilk Plus ABI

The front end needs **ABI-specific** knowledge about **runtime data types** and **functions**.

Cilk Fibonacci code

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n - 1);
    y = fib(n - 2);
  }
  return x + y;
}
```

Clang

+ Cilk Plus runtime ABI

C pseudocode of LLVM IR

```
int fib(int n) {
  __cilkrts_stack_frame sf;
  if (n < 2) return n;
  int x, y;
  __cilkrts_enter_frame(&sf);
  if (!__builtin_setjmp(sf.ctx))
    __fib_helper(&x, n-1);
  y = fib(n-2);
  if (sf.flags & CILK_FRAME_UNSYNCHED)
    if (!__builtin_setjmp(sf.ctx))
      __cilkrts_sync(&sf);
  __cilkrts_leave_frame(&sf);
  return x + y;
}
void __fib_helper(int *x, int n) {
  __cilkrts_stack_frame sf;
  __cilkrts_enter_frame_helper(&sf);
  __cilkrts_detach(&sf);
  *x = fib(n);
  __cilkrts_leave_helper_frame(&sf);
}
```

Insert local *stack-frame variables.*

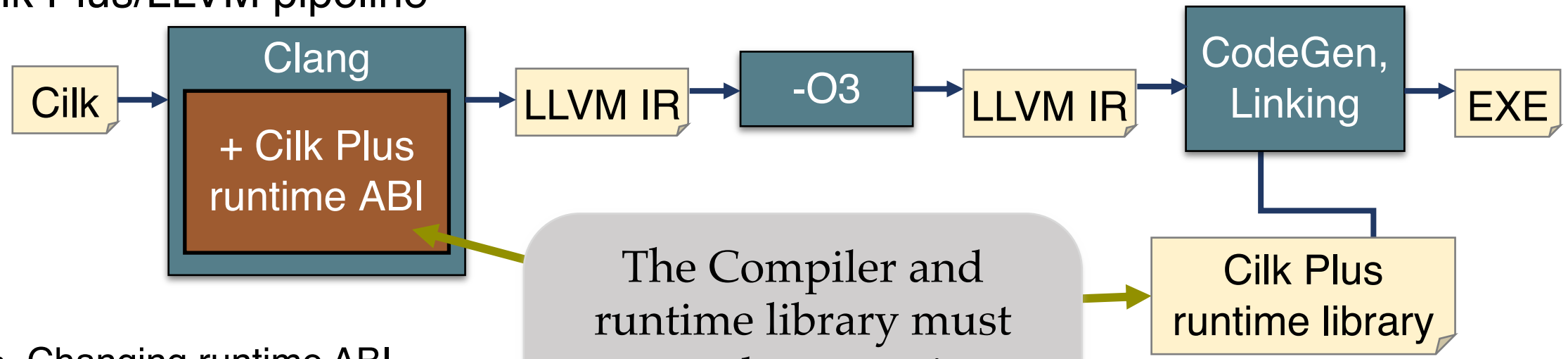
Create a *helper function.*

Call runtime functions and implement necessary control.

The OpenMP runtime ABI has similar complexities and is larger.

Problem: Hard to Modify Runtime ABI

Cilk Plus/LLVM pipeline

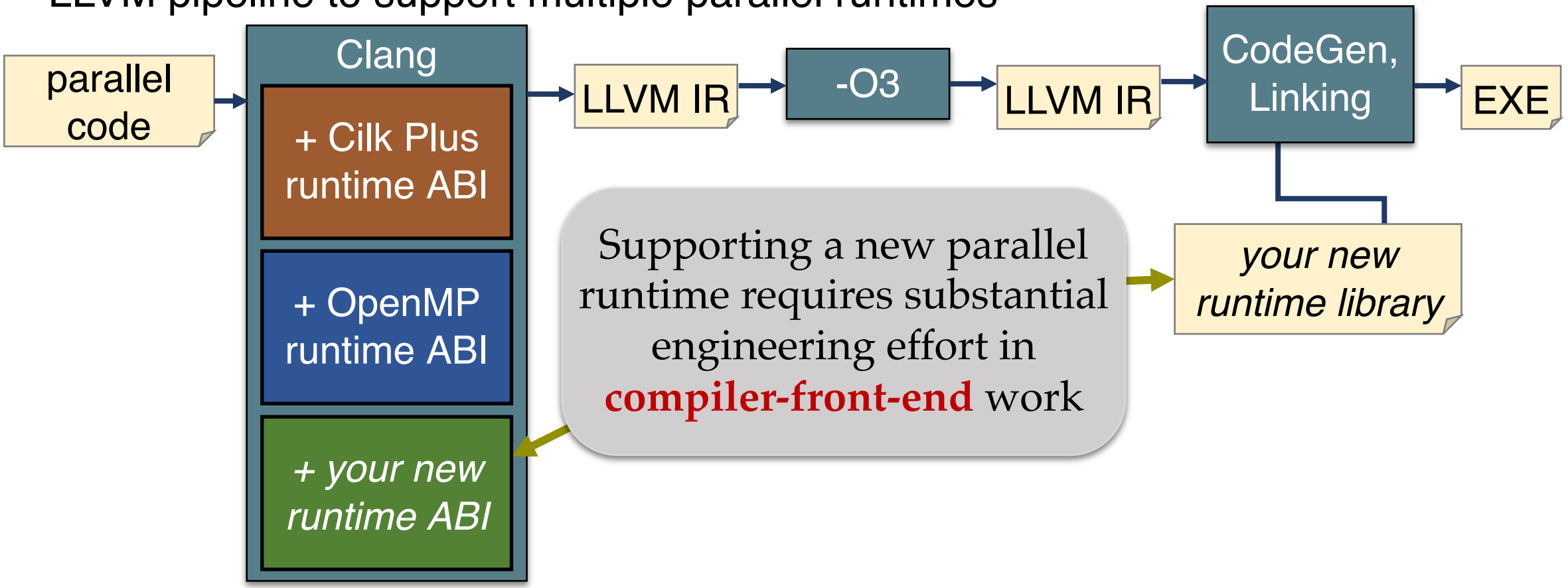


- Changing runtime ABI requires changing both the library and the compiler.
- Extending the ABI to add new runtime features or support for tools requires compiler work.

Other task-parallel systems, such as OpenMP or X10, use a similar design.

Problem: Hard to Develop New Parallel Runtime

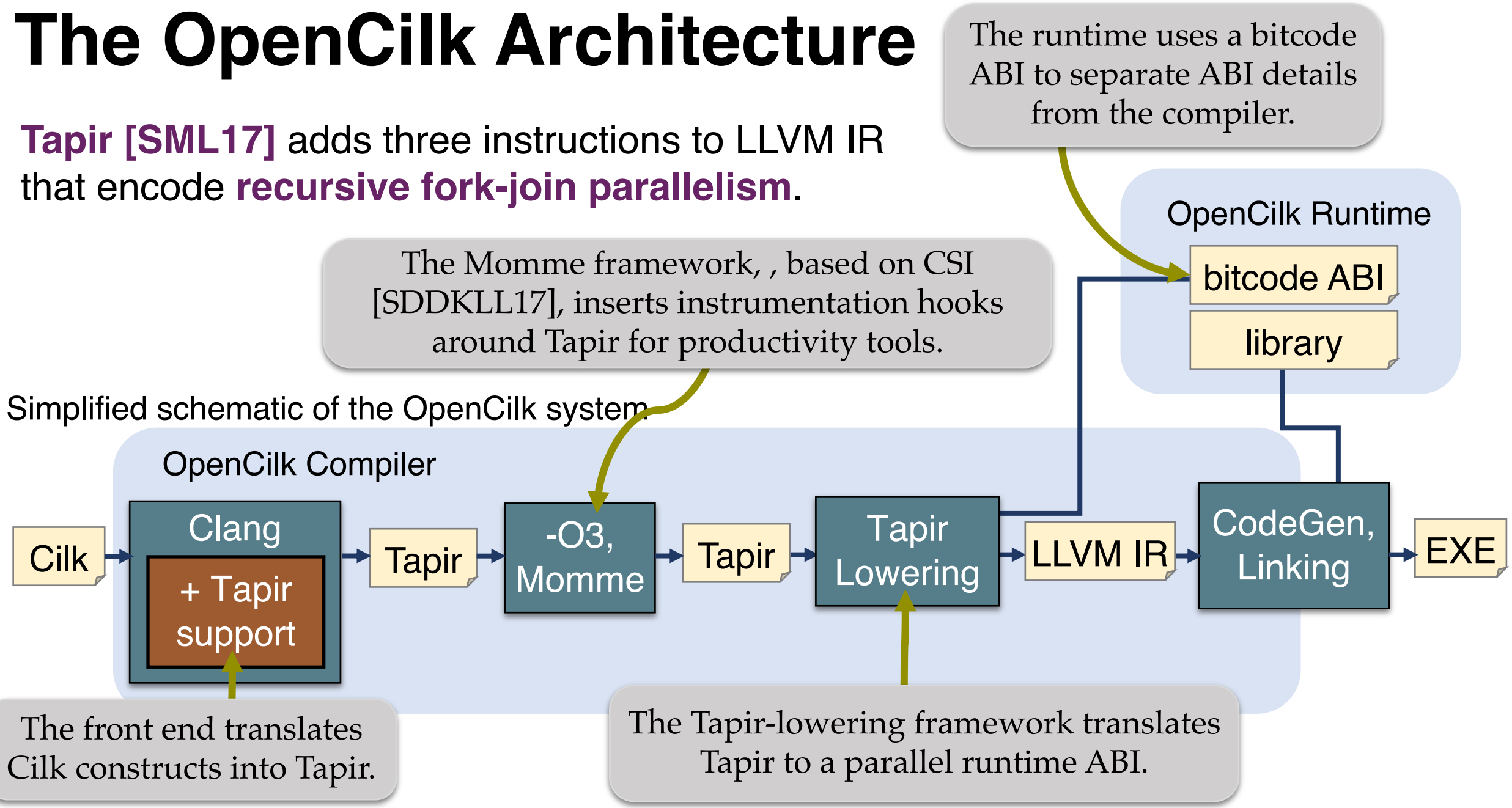
LLVM pipeline to support multiple parallel runtimes



Context: In LLVM 14, the Clang front end is approximately 1 million lines of code, substantially larger than the sources for many parallel-runtime libraries.

The OpenCilk Architecture

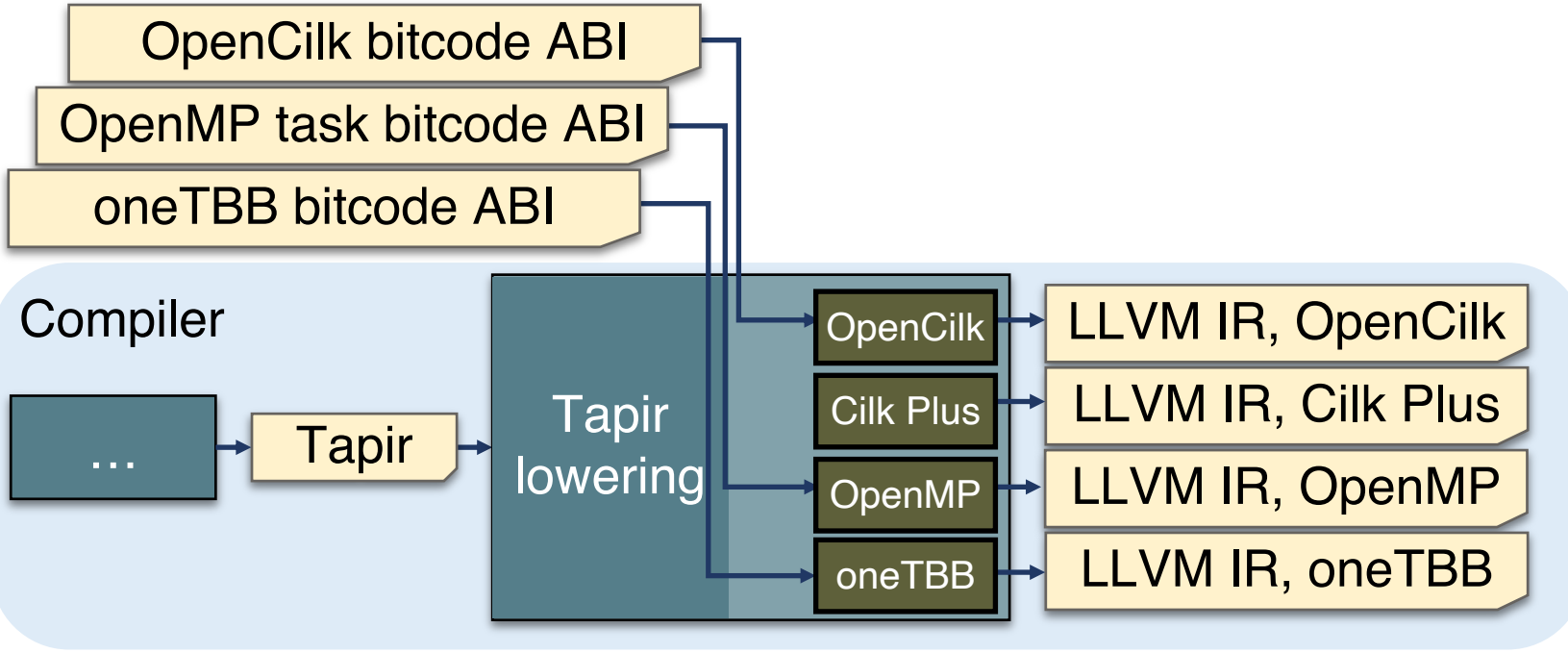
Tapir [SML17] adds three instructions to LLVM IR that encode **recursive fork-join parallelism**.



Case Study: Adding New Parallel Runtime Back-Ends

We extended OpenCilk to compile Cilk programs to **different** parallel runtime systems, including Cilk Plus, OpenMP tasks, and oneTBB.

Schematic of the Tapir-lowering framework



Runtimeback end	*Approx. new lines*
OpenCilk	1,680
Cilk Plus	1,900
OpenMP tasks	850
oneTBB	780

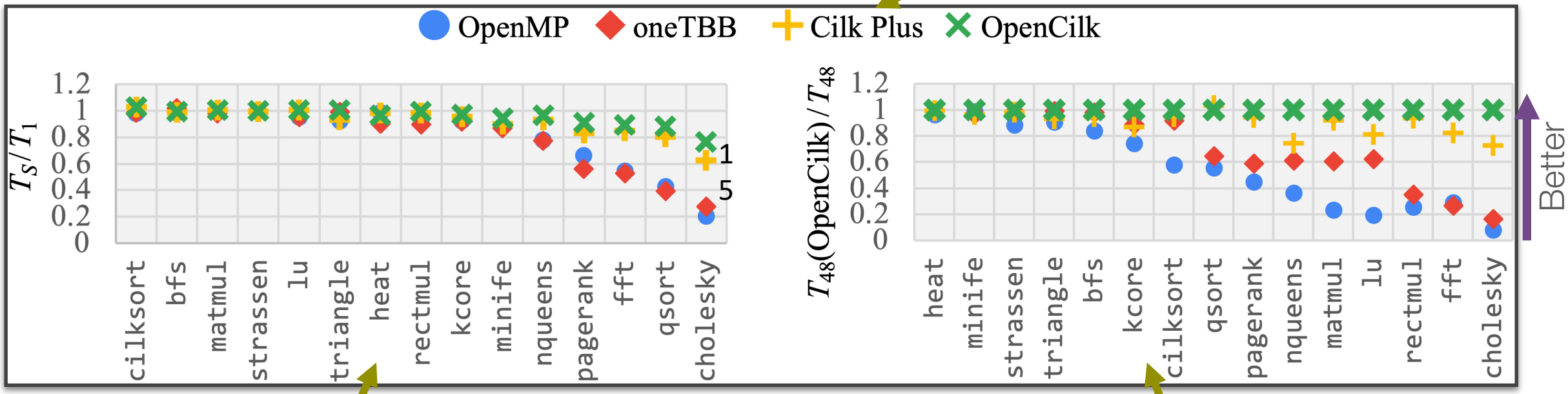
Each new runtime back end required fewer than **2000** new lines of code.

OpenCilk Produces Fast Code.

Performance of OpenCilk

OpenCilk produces **fast code**.

Comparable to the original Tapir/LLVM compiler



OpenCilk achieves high **work efficiency**.

OpenCilk **scales well** on parallel processors.

Machine: Amazon AWS c5.metal: 48 cores across 2 sockets clocked at 3 GHz, 192 GiB DRAM

OpenCilk's bitcode ABI made it **easy** to performance-engineer the runtime system.

Who Should Use OpenCilk and Why

Who should use **OpenCilk**:

- **Educators** who teaches parallelism
- **Researchers** who wants to experiments with task-parallel platforms to:
  - build new **parallel language front end**
  - design and implement new **runtime features**
  - develop **tools** for task-parallel code
- **Application developers** who wants to write **fast task-parallel code** for multicore hardware

Why **OpenCilk**:

- The **linguistics** are simple and easy to understand.
- The runtime scheduler provides **provable execution time bounds**.
- The software infrastructure is **modular** and **extensible**.
- The overall system produces **fast code**.
- The system comes with a suite of **productivity tools**.

Questions?

https://www.opencilk.org

Most content based on "OpenCilk: A Modular and Extensible Software Infrastructure for Fast Task-Parallel Code" by Tao B. Schardl and I-Ting Angelina Lee, published in PPoPP 2023.

Special thanks to the OpenCilk team —Tim Kaler, Alexandros-Stavros Iliopoulos, John Carr, Kyle Singer, Dorothy Curtis, Bruce Hoppe, and Charles E. Leiserson — and everyone who has contributed to and supported OpenCilk, including external contributors!