

# OpenCilk: Architecting a Task-Parallel Software Infrastructure for Modularity, Extensibility, and Performance

---

PMAM

February 26, 2023

Montreal, Canada



Tao B. Schardl  
MIT CSAIL

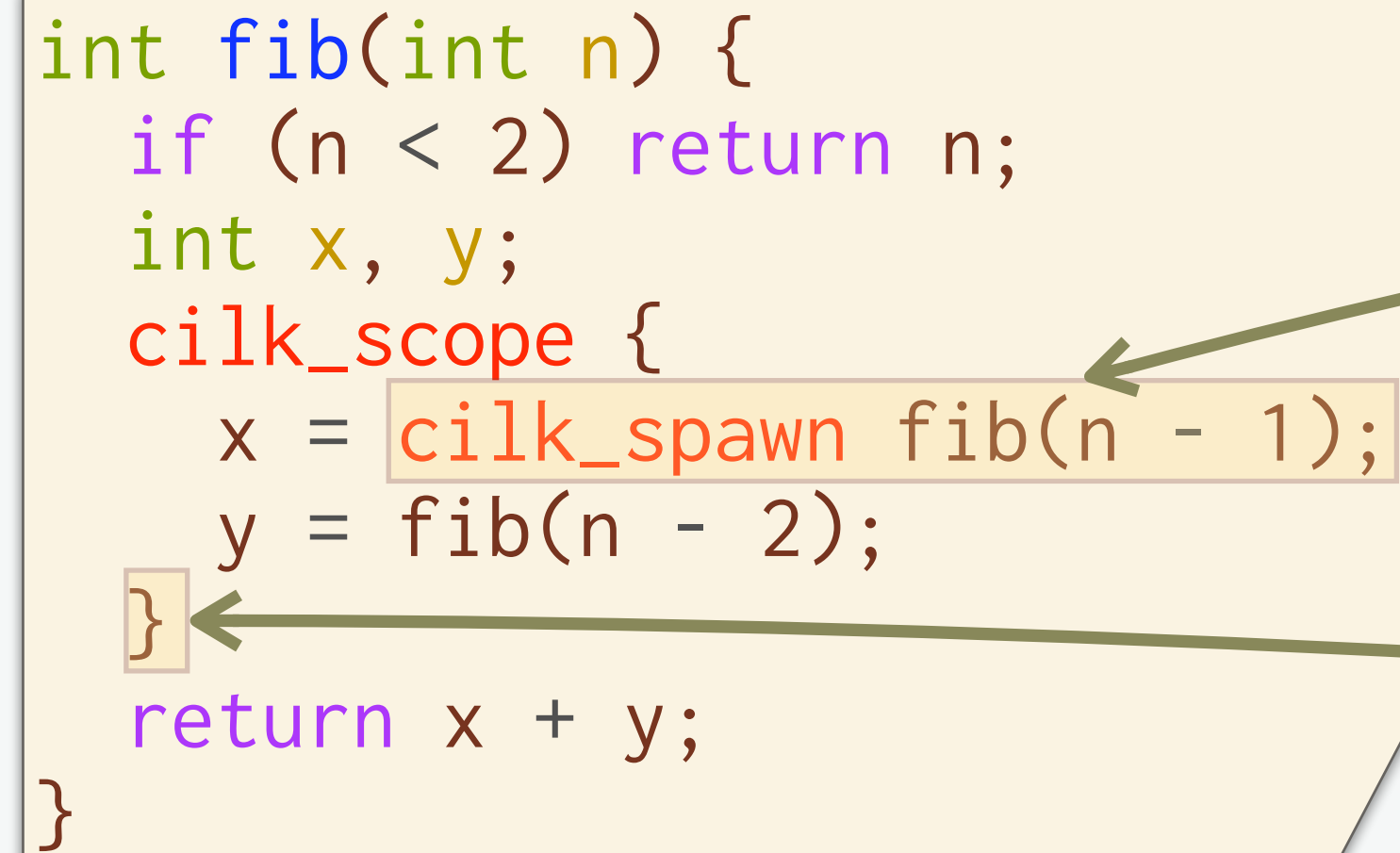


# Cilk task-parallel programming

OpenCilk provides a new implementation of the **Cilk** task-parallel programming platform.

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;  
}
```



The named **child function** is allowed to execute in parallel with the parent caller.

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

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

OpenCilk is largely compatible with its predecessor, Cilk Plus, but features an entirely **new design** and **implementation** that aims to cater to parallel-computing **researchers** and **teachers**.

# OpenCilk components

---

The OpenCilk system provides several **components**, including:

- A **compiler**, based on LLVM and Tapir,
- A streamlined and efficient work-stealing **runtime system**,
- A suite of provably good **productivity tools**, including a **race detector** and a **parallel-scalability analyzer**.

These components are **integrated**, but **modularized** to make it easy **modify** and **extend** OpenCilk without sacrificing **performance**.

**This talk:** OpenCilk's design and the rationale behind it.

# Example: Normalizing a vector using Cilk Plus (i.e., before OpenCilk)

Cilk code to normalize a vector

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

Allow all loop iterations  
to execute in **parallel**.

**Test:** Random vector,  $n = 64\text{M}$  **Machine:** Amazon AWS c4.8xlarge

*Running time of the original serial code:*  $T_S = 0.312\text{ s}$

*Running time on 18 cores:*  $T_{18} = 180.657\text{ s}$

*Running time on 1 core:*  $T_1 = 2600.287\text{ s}$

**Terrible work efficiency:**

$$T_S/T_1 = 0.312/2600$$

$$\sim 1/8300$$

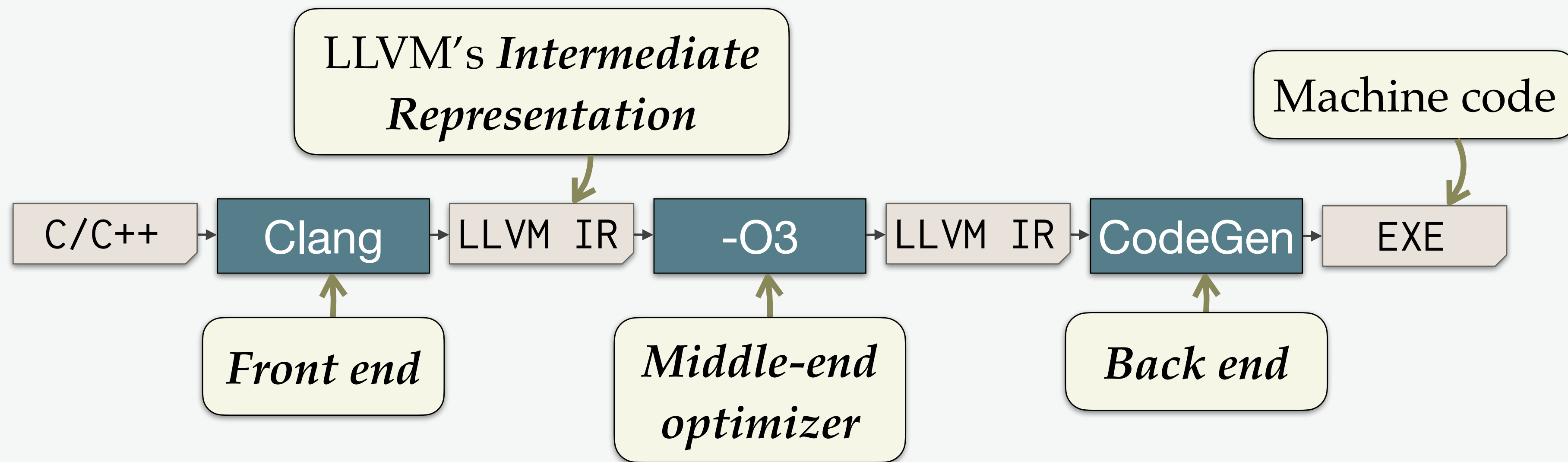
The story for OpenMP is similar, but more complicated.

Code compiled using GCC 6.2. The Cilk Plus/LLVM compiler produces worse results.

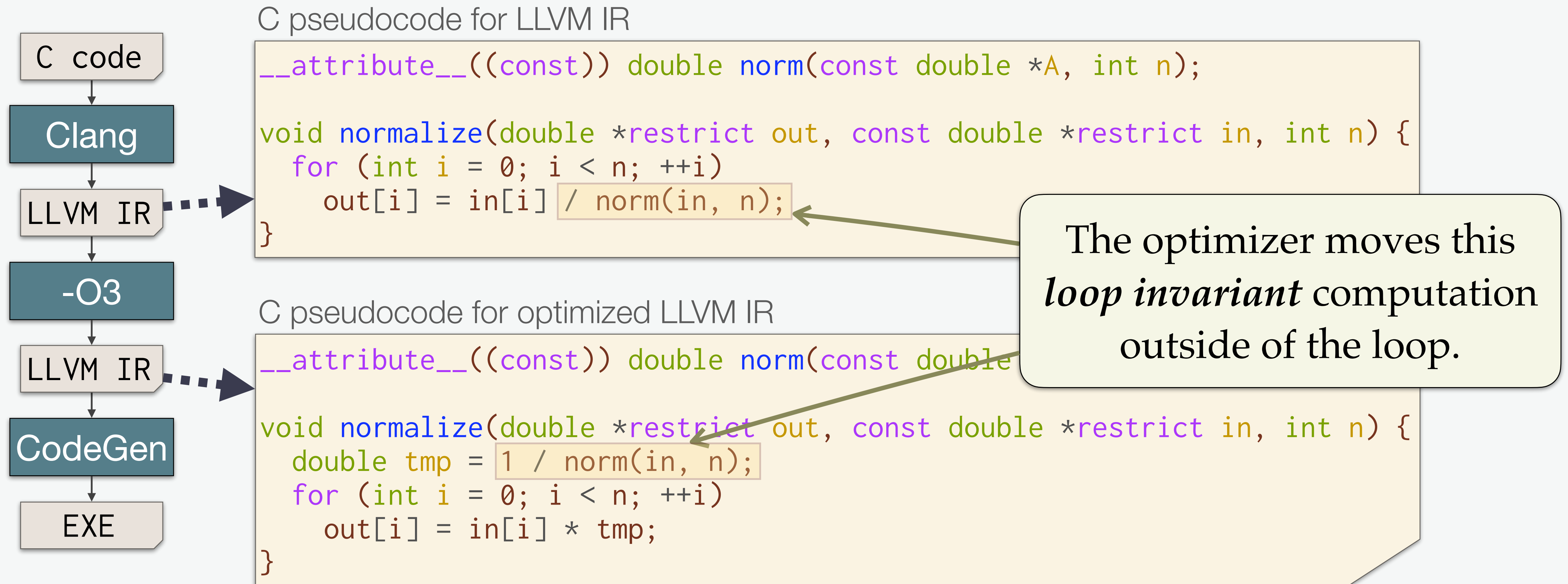


# The LLVM compiler pipeline

---



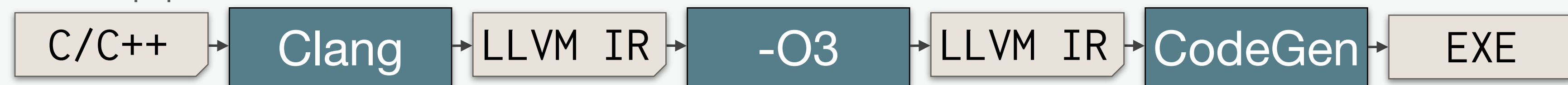
# Compiler optimization of serial code



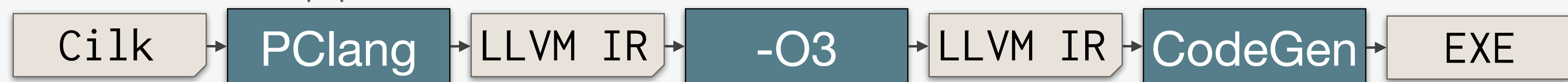
# The Cilk Plus/LLVM compiler pipeline

---

LLVM pipeline

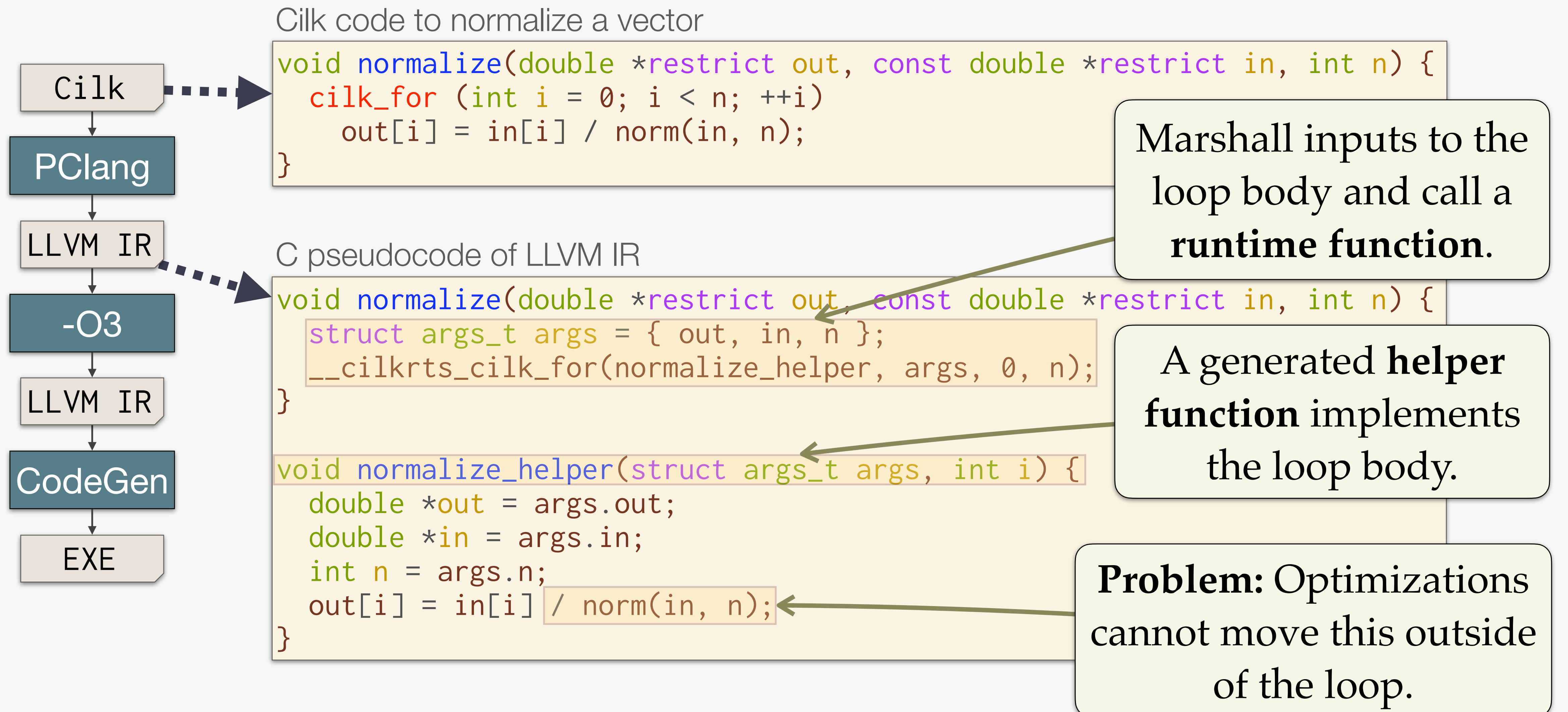


Cilk Plus/LLVM pipeline



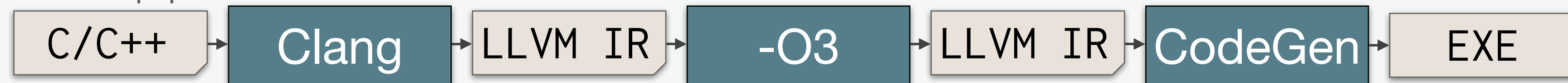
A modified front end  
handles all parallel  
language constructs.

# Compiling parallel code

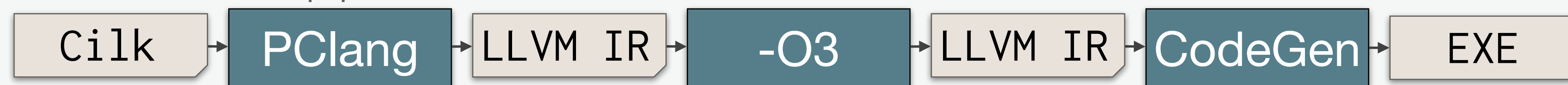


# Tapir: Fork-join parallelism within the compiler IR [SML17]

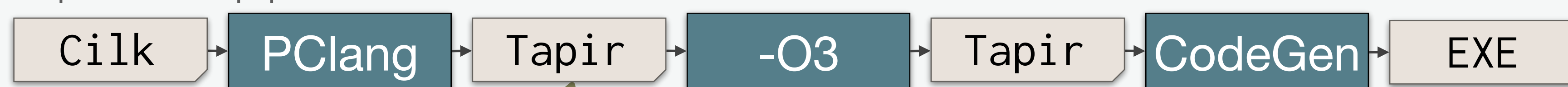
LLVM pipeline



Cilk Plus/LLVM pipeline



Tapir/LLVM pipeline



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

With **only minor changes**, LLVM's **existing optimizations and analyses** work on **parallel code**.



# Previous approaches to parallel IR's

---

- Parallel precedence graphs [SW91, SHW93]
- Parallel flow graphs [SG91, GS93]
- Concurrent SSA [LMP97, NUS98]
- Parallel program graphs [SS94, S98]
- “[LLVMdev][RFC] Parallelization metadata and intrinsics in LLVM (for OpenMP, etc.)”  
<http://lists.llvm.org/pipermail/llvm-dev/2012-August/052477.html>
- “[LLVMdev][RFC] Progress towards OpenMP support”  
<http://lists.llvm.org/pipermail/llvm-dev/2012-September/053326.html>
- LLVM Parallel Intermediate Representation: Design and Evaluation Using OpenSHMEM Communications [KJIAC15]
- LLVM Framework and IR Extensions for Parallelization, SIMD Vectorization and Offloading [TSSGMGZ16]
- HPIR [ZS11, BZS13]
- SPIRE [KJAI12]
- INSPIRE [JPTKF13]
- LLVM's parallel loop metadata

# Folk wisdom about parallel IR's

---

From “[LLVMdev] LLVM Parallel IR,” 2015:

- “[I]ntroducing [parallelism] into a so far ‘sequential’ IR will cause **severe breakage and headaches.**”
- “[P]arallelism is invasive by nature and would have to **influence most optimizations.**”
- “[It] is not an easy problem.”
- “[D]efining a parallel IR (with first-class parallelism) is a **research topic.**”

# Implementing Tapir in LLVM 6.0

<i>Compiler component</i>	<i>LLVM 6.0 (lines)</i>	<i>Tapir/LLVM (lines)</i>
Core middle-end	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

# Tapir: Fork-join parallelism within the compiler

Cilk code to normalize a vector

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

**Test:** Random vector,  $n = 64\text{M}$  **Machine:** Amazon AWS c4.8xlarge

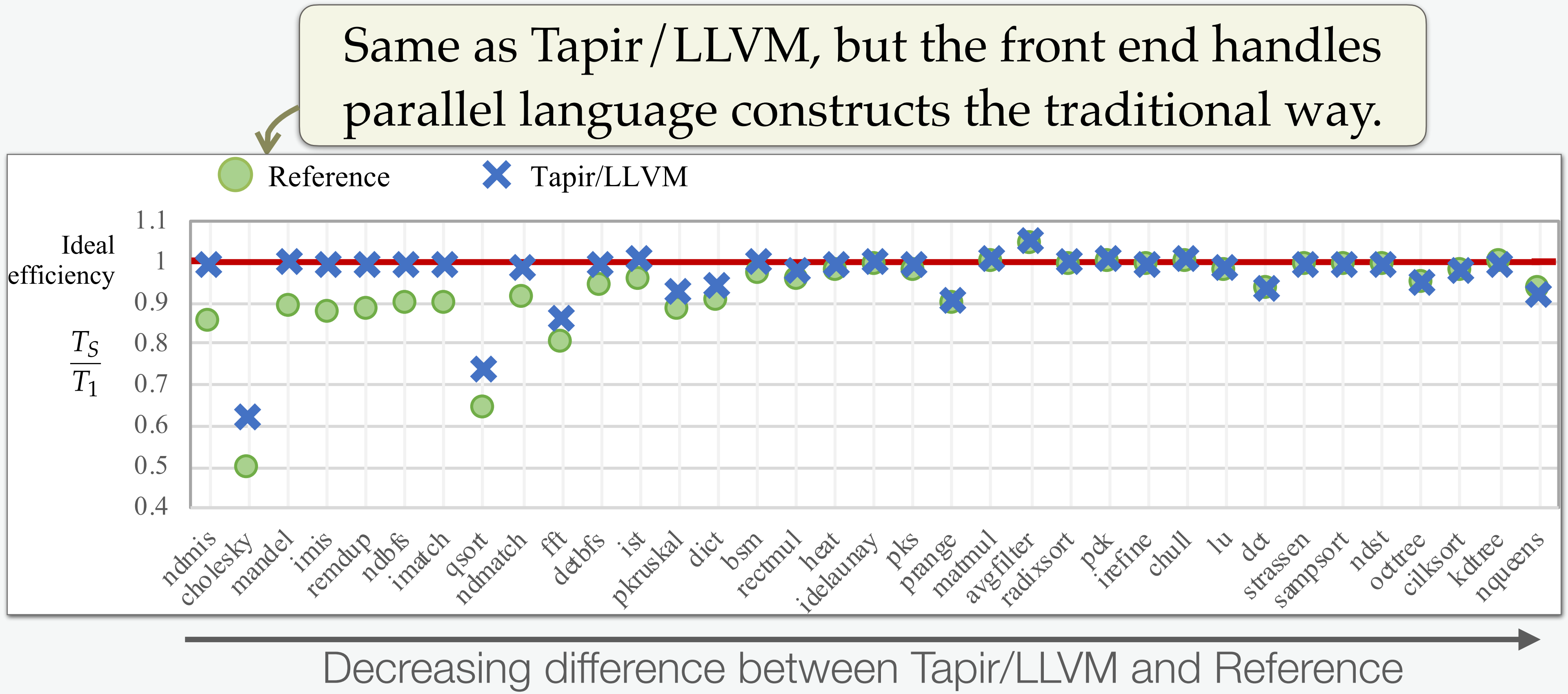
*Running time of the original serial code:*  $T_S = 0.312\text{ s}$

*Compiled with Tapir/LLVM, run on 1 core:*  $T_1 = 0.321\text{ s}$

*Compiled with Tapir/LLVM, run on 18 cores:*  $T_{18} = 0.081\text{ s}$

Great work  
efficiency:  
 $T_S/T_1 = 97\%$

# Work-efficiency improvement



**Machine:** Amazon AWS c4.8xlarge, with 18 cores clocked at 2.9 GHz, 60 GiB DRAM



# Outline

---

- Tapir: Embedding recursive fork-join parallelism into LLVM IR
- OpenCilk: A modular and extensible software infrastructure for fast task-parallel code
- Software performance engineering and the end of Moore's Law

# Outline

---

- Tapir: Embedding recursive fork-join parallelism into LLVM IR
- OpenCilk: A modular and extensible software infrastructure for fast task-parallel code
- Software performance engineering and the end of Moore's Law

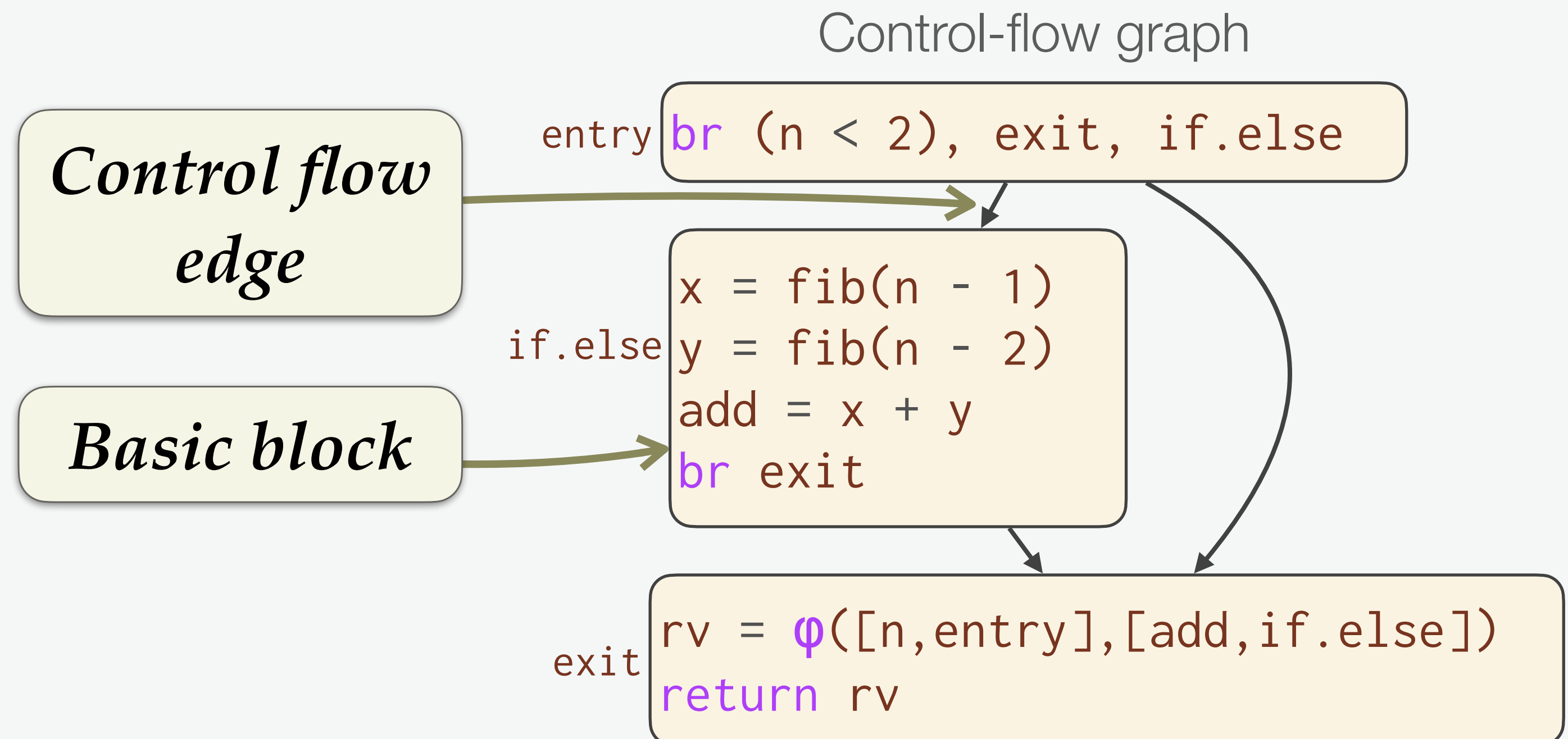
**Coauthors:** William S. Moses,  
Charles E. Leiserson

# Background on LLVM IR

LLVM represents each function as a **control-flow graph (CFG)**.

C code

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



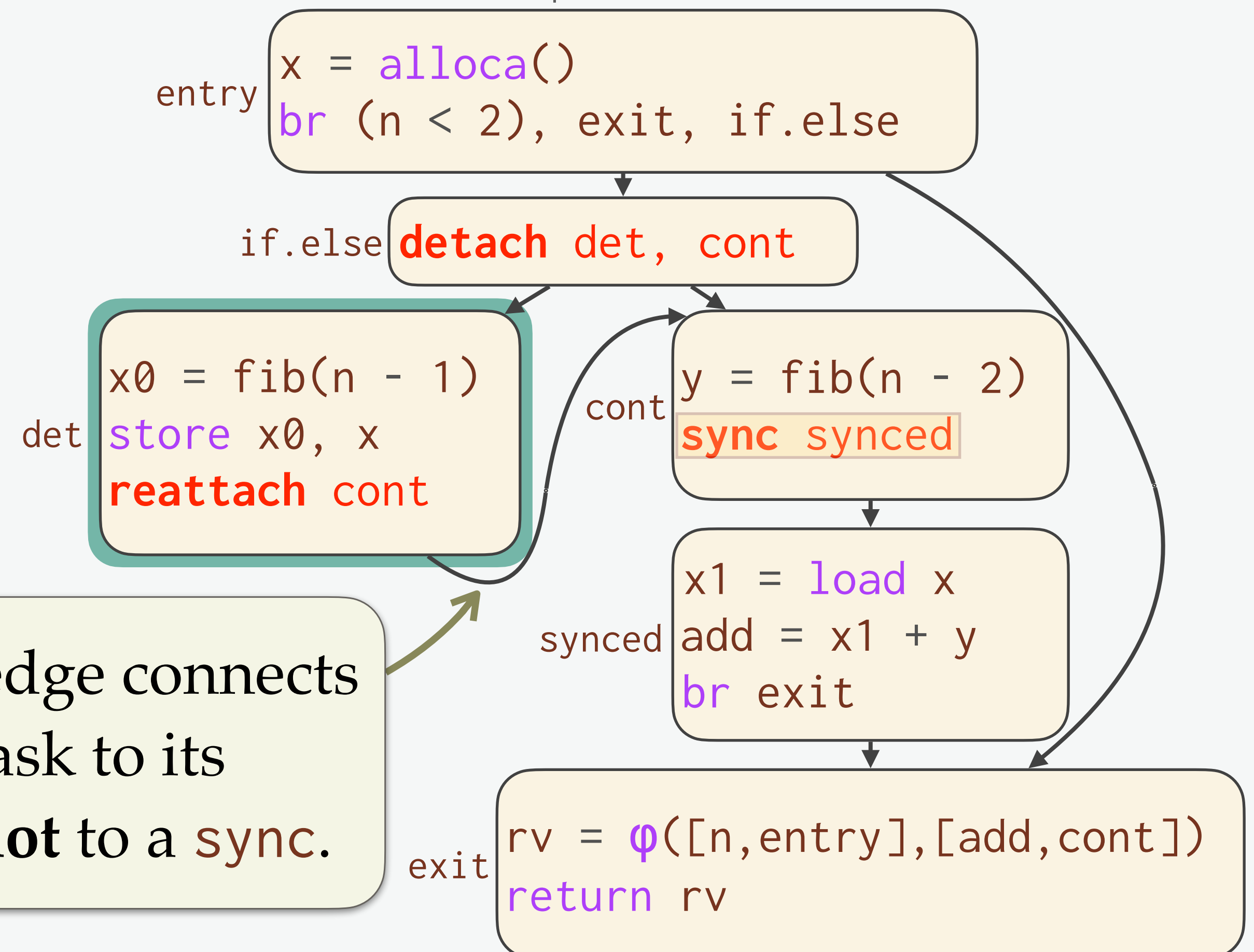
# Tapir's new LLVM IR instructions

Tapir's new instructions model parallel tasks **asymmetrically**.

Cilk Fibonacci code

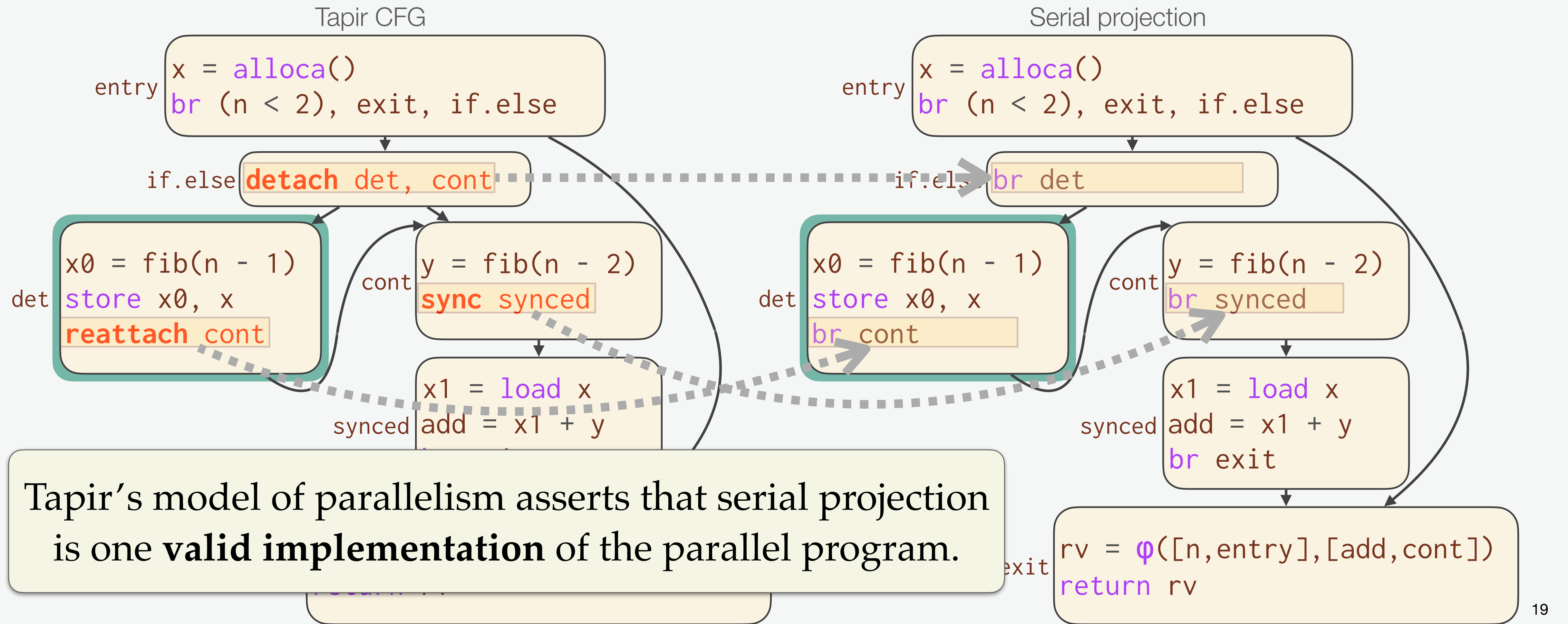
```
int fib(int n) {  
  if (n < 2) return n;  
  int x, y;  
  cilk_scope {  
    x = cilk_spawn fib(n - 1);  
    y = fib(n - 2);  
  }  
  return x + y;  
}
```

Tapir CFG



# The serial-projection property

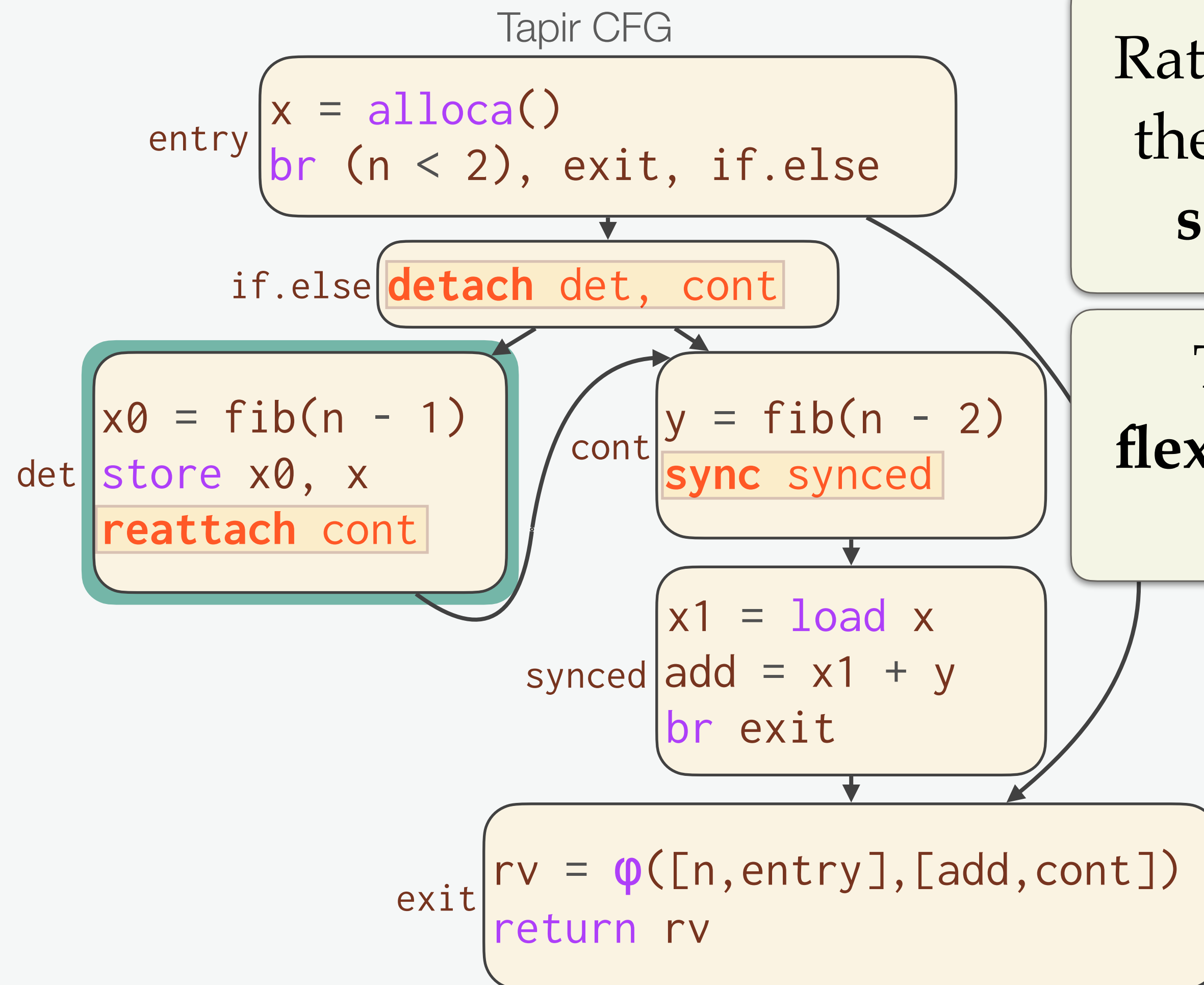
Tapir models the *serial projection* of the parallel program.





# Reasoning about a Tapir CFG

Intuitively, many compiler optimizations can reason about Tapir as a **minor change** to the serial projection.



Rather than struggle to analyze **concurrency**, the compiler can **understand** the program's **semantics** based on the serial projection.

The compiler and runtime system have **flexibility** to **choose** how to use the available parallelism.

**But not all parallel programs have a serial projection!**

# Focus of Tapir

---

- Shared-memory multicore programming
- Task parallelism
- Serial-projection property
- Simple execution model
- Extensible representation
- Deterministic debugging
- Effective compiler optimizations
- Simple performance model
- Work efficiency
- Parallel scalability
- Composable parallelism
- Parallelism, rather than concurrency

# Adoption of Tapir in parallel computing research and development

---

Tapir's focus has **enabled** its use in many novel parallel-programming settings.

- Margerm *et al.* (Simon Fraser University and Intel) developed *TAPAS* [MSGSP18], a **hardware synthesis tool** built on top of Tapir to synthesize parallel accelerators.
- Siddharth Samsi (MIT LL) and I developed *TapirXLA* [SS19], which integrates Tapir with TensorFlow's XLA compiler to **optimize machine-learning applications**.
- Shajii *et al.* developed the *Seq* language for **bioinformatics** [SNBBA19], which uses Tapir to compile and optimize parallel language constructs.
- Ying *et al.* developed the *T4* compiler [YJS20], based on Tapir, to compile sequential code for effective **speculative parallelization in hardware**.
- Lucata Corporation developed a back end to Tapir that targets their novel **in-memory-processing** architecture.

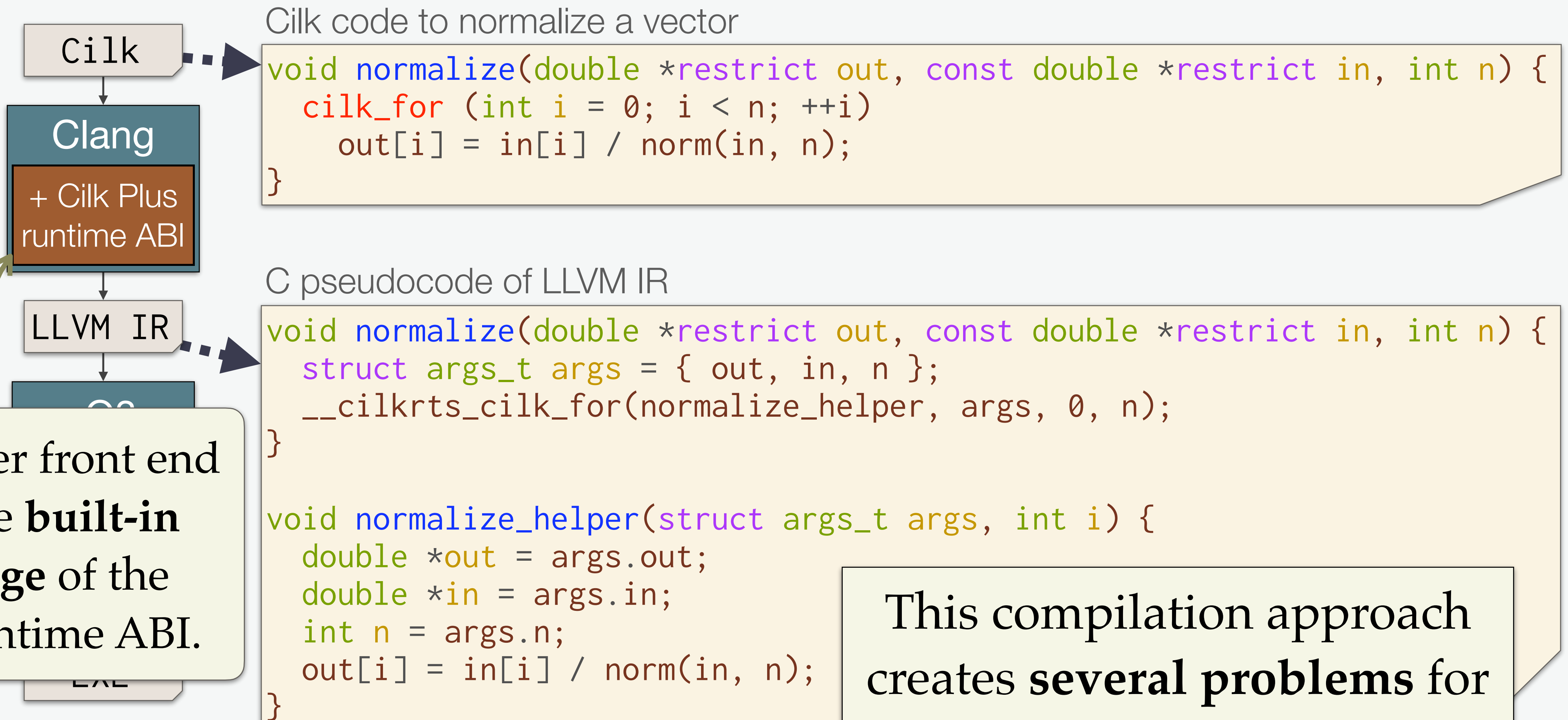
# Outline

---

- Tapir: Embedding recursive fork-join parallelism into LLVM IR
- OpenCilk: A modular and extensible software infrastructure for fast task-parallel code
- Software performance engineering and the end of Moore's Law

***Coauthor:*** I-Ting Angelina Lee

# Recap: Compiling parallel code the traditional way





# Example: Complexity of the Cilk Plus runtime ABI

The front-end code to implement a runtime ABI is **not simple**.

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.

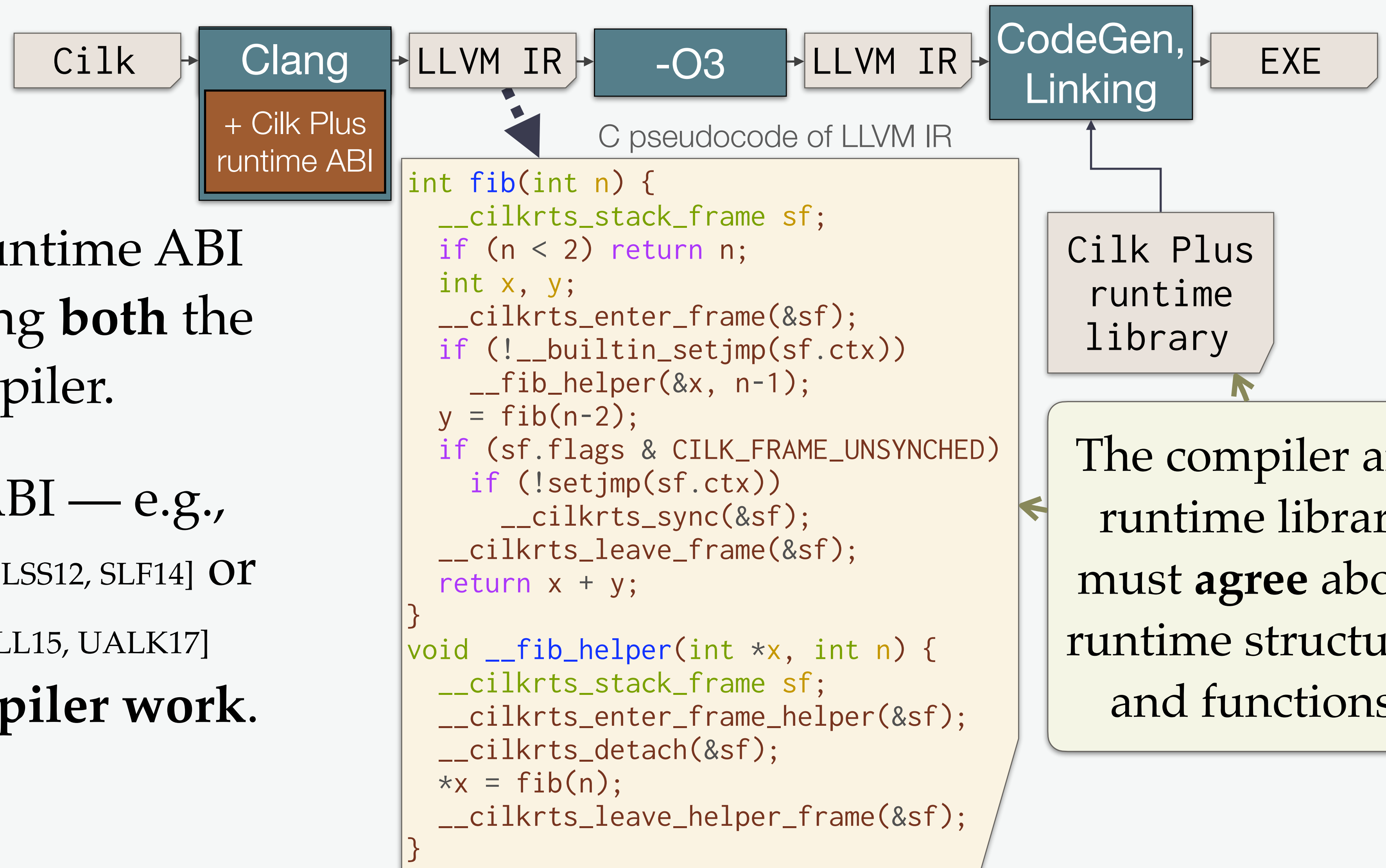
Call runtime functions  
and implement  
necessary control.

The OpenMP runtime ABI has similar complexities and is larger.

# Problem: Modifying the runtime ABI

The runtime ABI is **hard to modify**.

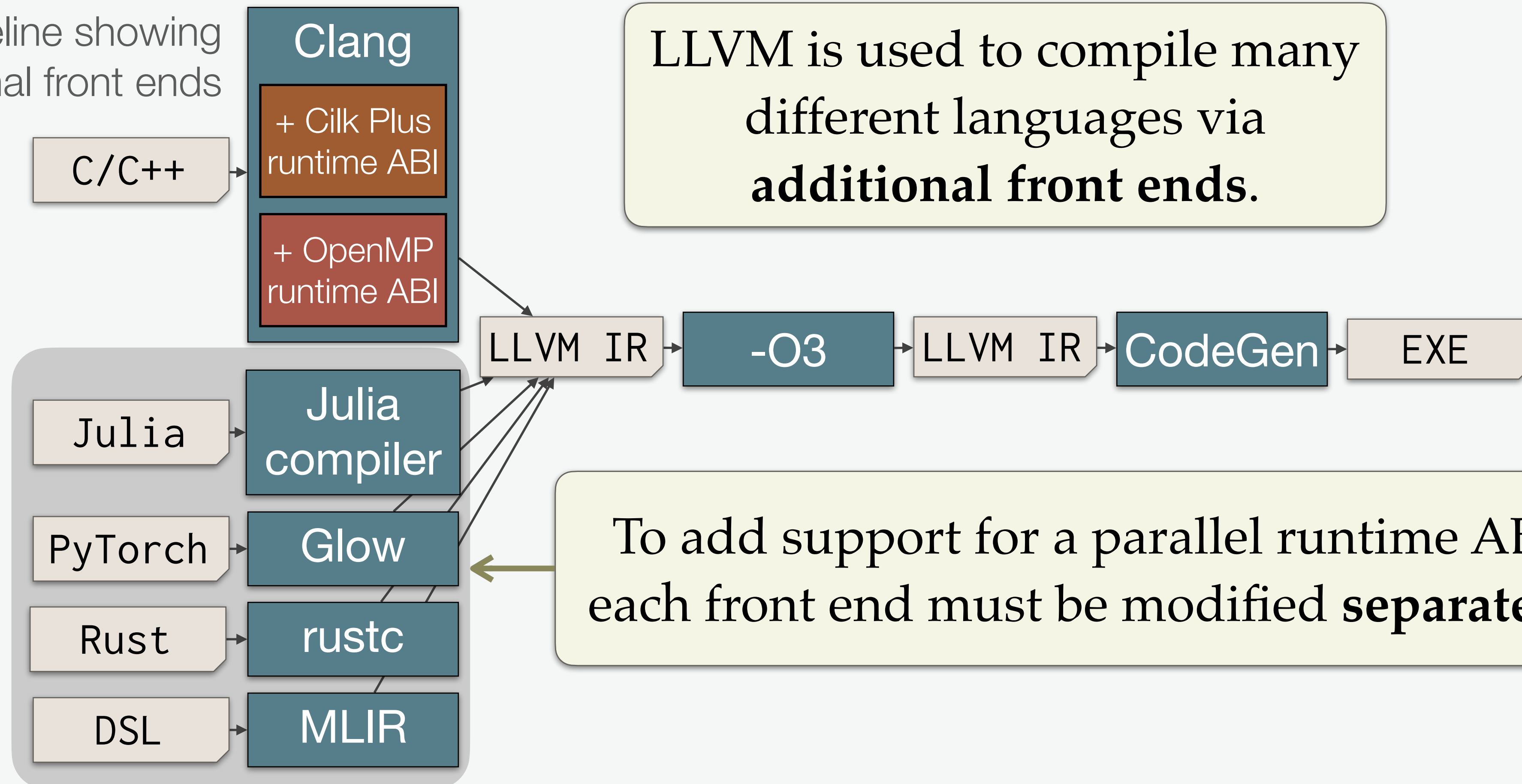
- Changing the runtime ABI requires changing **both** the library and compiler.
- Extending the ABI — e.g., to add DPRNG [LSS12, SLF14] or tool support [SKLLL15, UALK17] — requires **compiler work**.



# Problem: Hard to extend to new languages

Adding parallelism to a new language front end requires **independent engineering effort**.

LLVM pipeline showing  
additional front ends

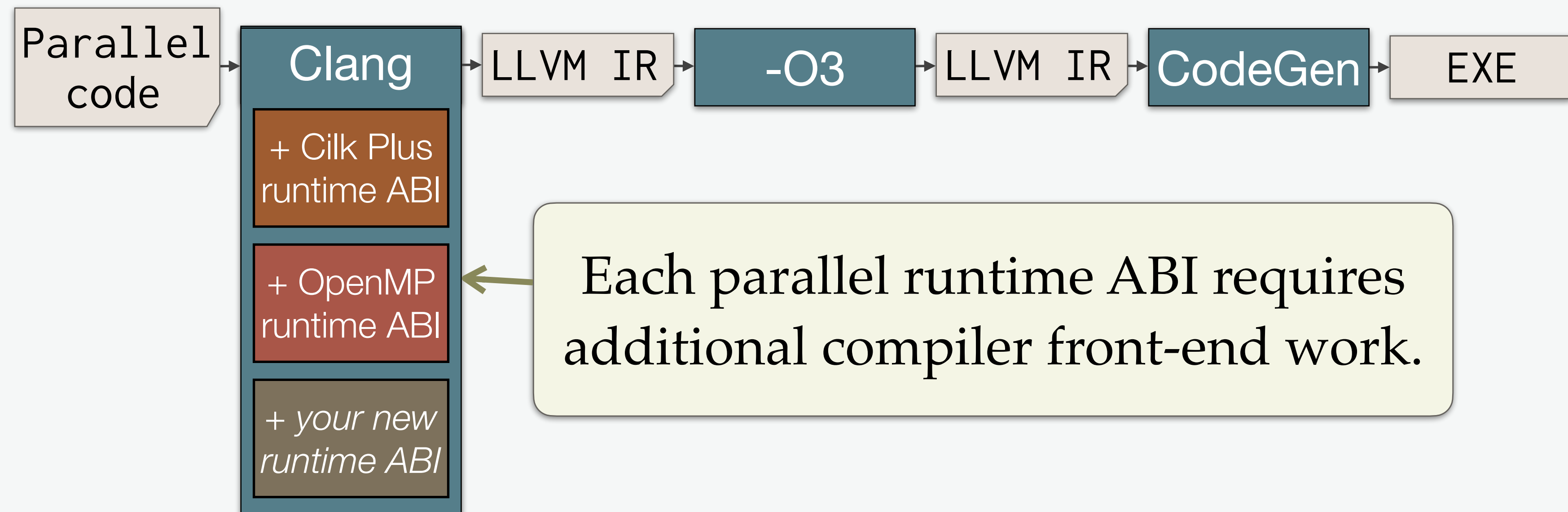


LLVM is used to compile many different languages via **additional front ends**.

To add support for a parallel runtime ABI, each front end must be modified **separately**!

# Problem: Hard to develop new parallel runtimes

Developing a new parallel runtime *back end* requires **substantial engineering effort** in the compiler front end.



Today, the Clang front end is approximately 1 million lines of code, substantially larger than the sources for many parallel-runtime libraries.

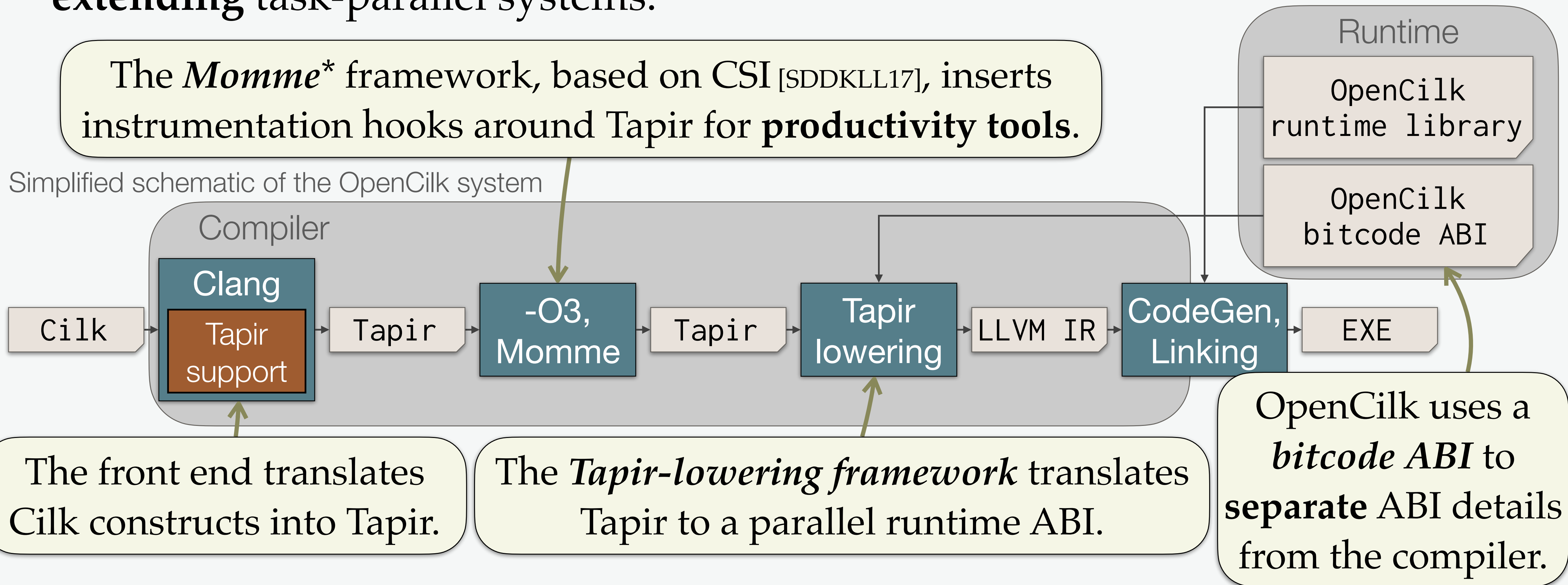


# OpenCilk architecture

OpenCilk uses Tapir and LLVM to address these issues with **modifying** and **extending** task-parallel systems.

The *Momme*\* framework, based on CSI [SDDKLL17], inserts instrumentation hooks around Tapir for **productivity tools**.

Simplified schematic of the OpenCilk system



\* Momme, in Japanese, is a unit used to measure the quality of silk fabrics.



# Case study: Making a new front end

We used OpenCilk to add `spawn`, `sync`, and `parallel_for` constructs to **Kaleidoscope**, a toy language used to teach LLVM internals.

Parallel Kaleidoscope Fibonacci code

```
def fib(n)
  if (n < 2) then n
  else
    var x, y in
      (spawn x = fib(n-1)) :
      y = fib(n-2) :
    sync
    (x + y);
```

We extended Kaleidoscope’s LLVM-based JIT compiler to use OpenCilk to **compile** and **execute** parallel tasks and use OpenCilk’s **productivity tools**.

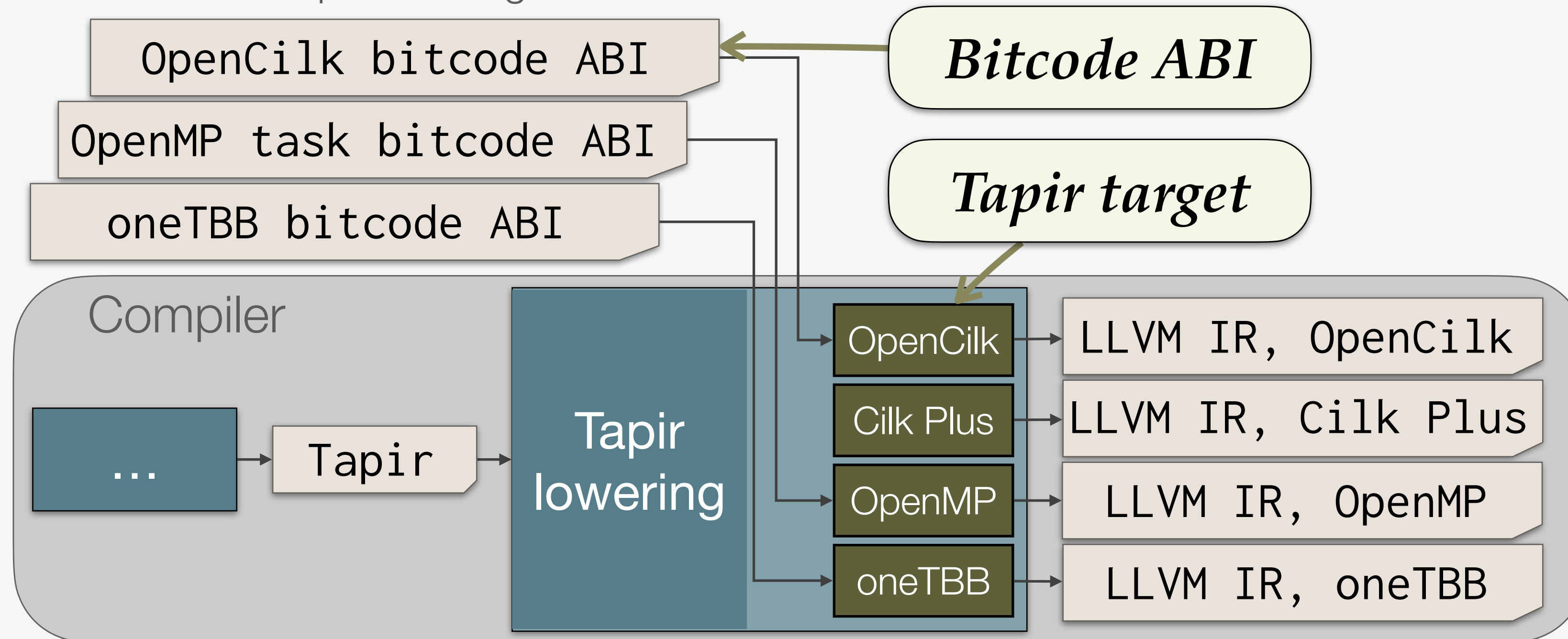
Implementing parallel constructs in Kaleidoscope

<i>Implementation task</i>	<i>Approx. new lines of code</i>
Parsing and Tapir generation	400
Invoke Tapir lowering and Momme	150
Link external libraries	100
Total	650

# 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 Tapir-lowering framework



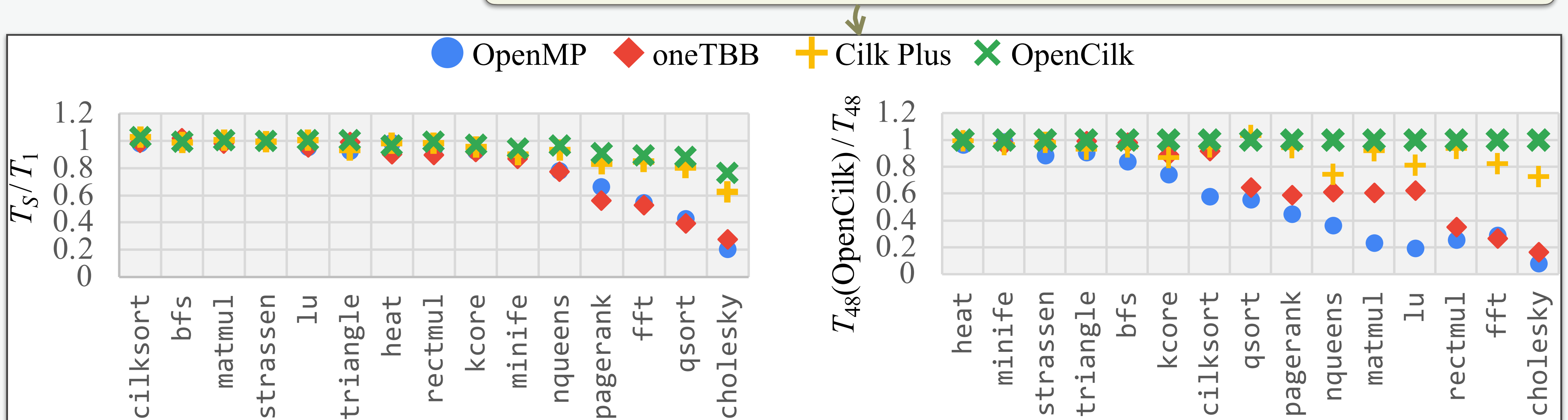
<i>Runtime back end</i>	<i>Approx. new lines</i>
OpenCilk	1,680
Cilk Plus	1,900
OpenMP tasks	850
oneTBB	780

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

# Performance of OpenCilk

OpenCilk produces **fast code** that consistently achieves high **work efficiency** and good **parallel scalability**.

Comparable to the original Tapir/LLVM runtime back end.



**Machine:** Amazon AWS c5.metal, with 48 cores clocked at 3 GHz, 192 GiB DRAM

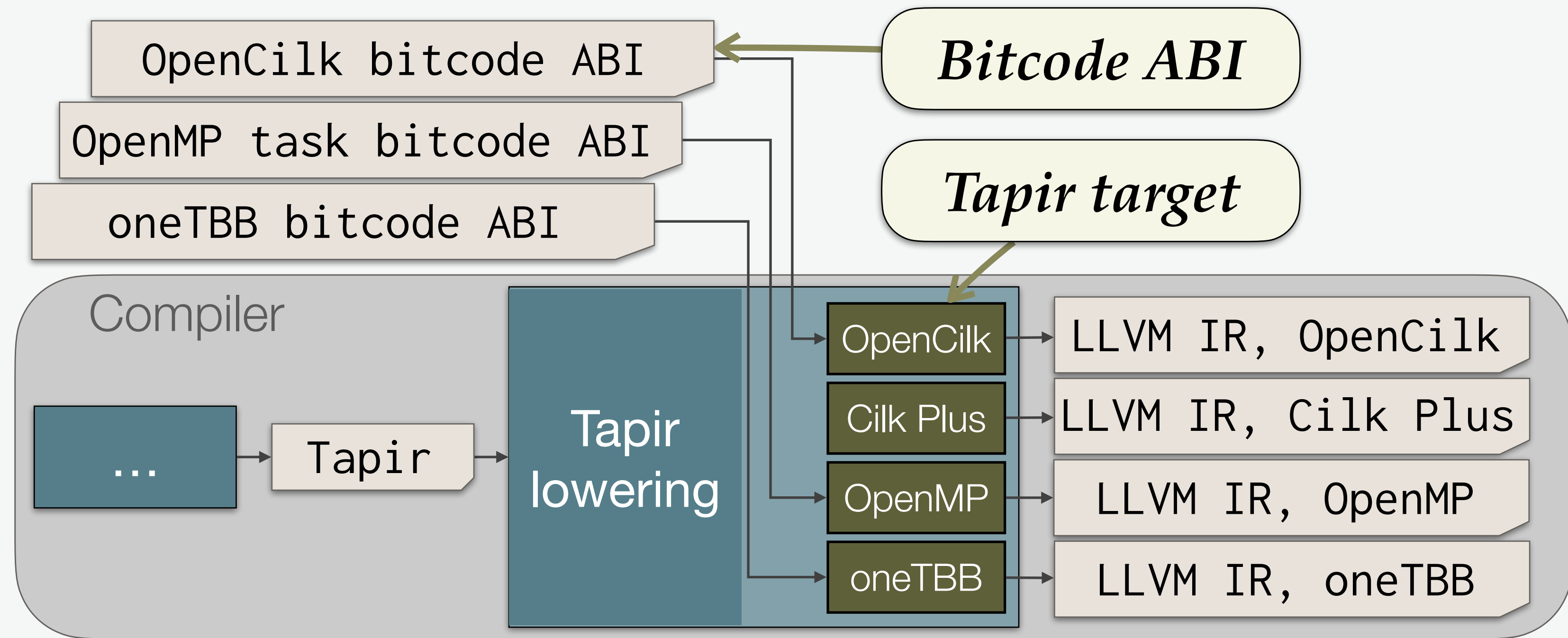
For more on OpenCilk's design...

Come to the PPoPP talk!

**Time:**

Tuesday, February 28 at  
10:00am.

**Room:** Montreal 4.





# Status of OpenCilk



- OpenCilk is completely open source and freely available online:  
<https://www.opencilk.org>
- The latest stable release is OpenCilk 2.0.1, which includes:
  - A **compiler**, based on LLVM 14.0.6, that implements Tapir,
  - A streamlined and fast work-stealing **runtime system**, and
  - Two productivity tools, built using Momme: A provably effective race detector — **Cilksan** — and a fast parallel-scalability analyzer — **Cilkscale**.
- OpenCilk features new linguistic and runtime support for reducer hyperobjects [FHLL09] and optimized and streamlined support for DPRNG'S [LSS12].
- OpenCilk's components are **integrated**, yet **modularized** to make it **easy** to **modify** and **extend** OpenCilk with new front ends, back ends, productivity tools, and more.



# Design goals of OpenCilk

---

- Support a **simple model of parallelism** with a **simple performance model** that is **easy to reason about** and **teach**.
- Enable **deterministic** parallel programming.
- Support **debugging** and **performance-analysis tools** that offer **mathematical guarantees** of their effectiveness.
- Ensure that all components are **integrated**.
- Make it **easy** for researchers and developers to **modify** and **extend** the system.
- Produce **high-performing** parallel code that is both work-efficient and achieves good parallel scalability, both in theory and in practice.

# Outline

---

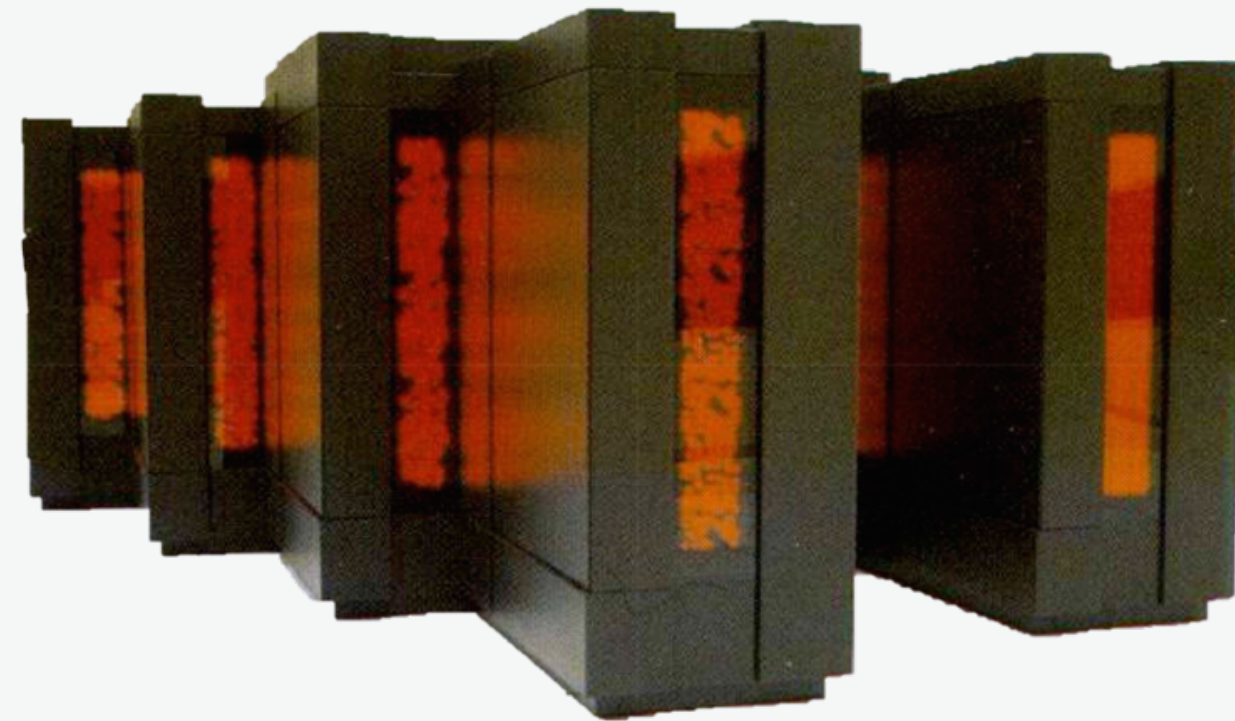
- Tapir: Embedding recursive fork-join parallelism into LLVM IR
- OpenCilk: A modular and extensible software infrastructure for fast task-parallel code
- Software performance engineering and the end of Moore's Law

**Coauthors:** Charles E. Leiserson, Neil C. Thompson,  
Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson,  
Daniel Sanchez

# The “popular” Moore’s Law

---

People often think of Moore’s Law as the trend of computing technology growing more powerful over time.



Connection Machine CM-5

- 60 GFLOPS in LINPACK
- \$47 million in 1993

≈



Apple 15" MacBook Pro

- 120 GFLOPS in LINPACK
- \$2799 in 2018

# The “real” Moore’s Law

This growth in computing performance has been driven by **semiconductor miniaturization**.

“There’s plenty of room at the bottom!” [F59]

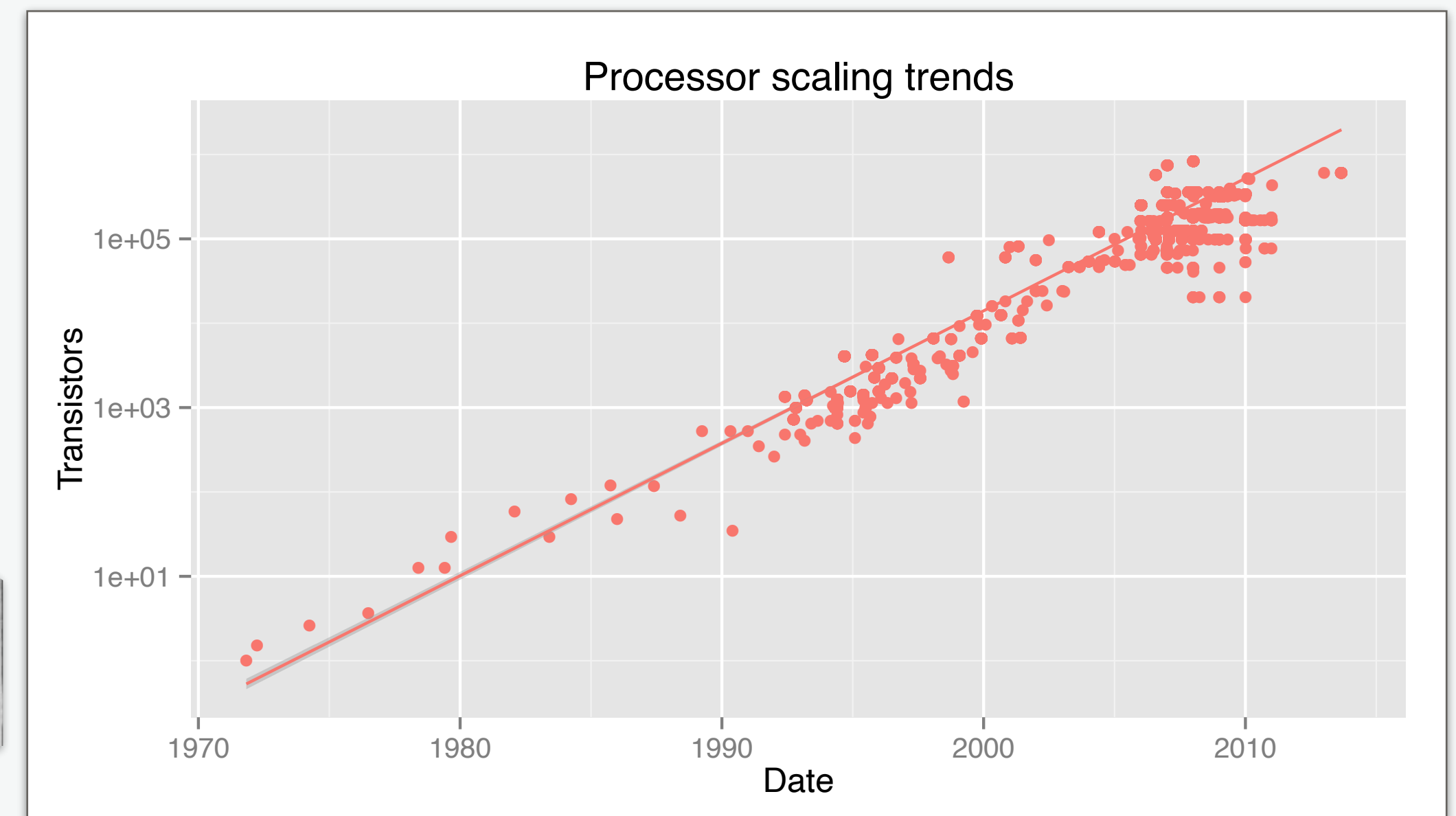


Richard Feynman

In 1965 and 1975, Gordon Moore predicted that the number of transistors on a semiconductor chip would **double every two years**.



Gordon Moore



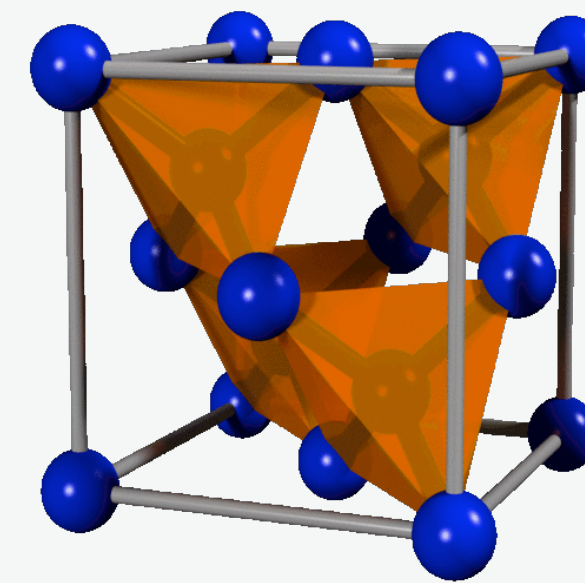
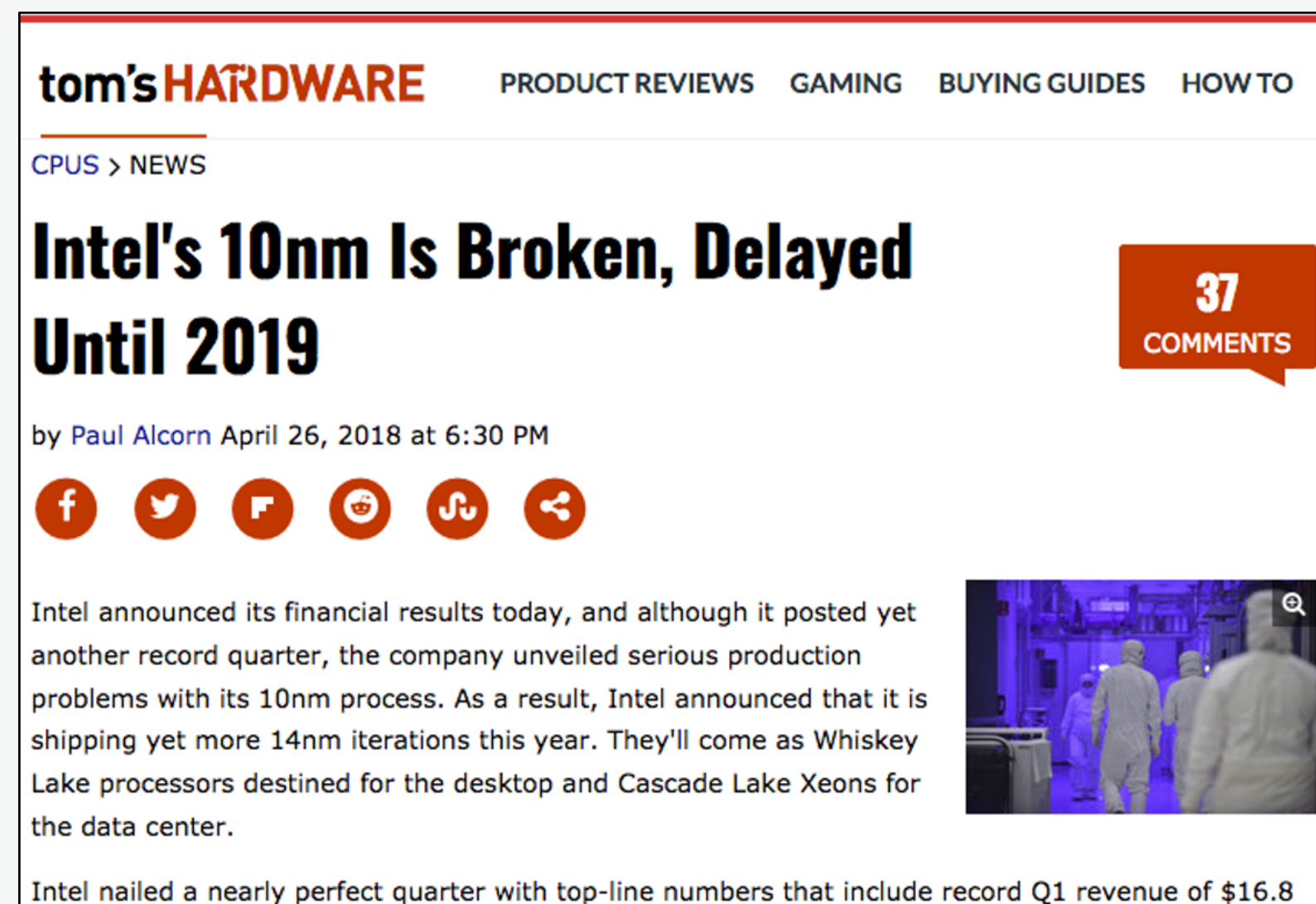


# The end is nigh!

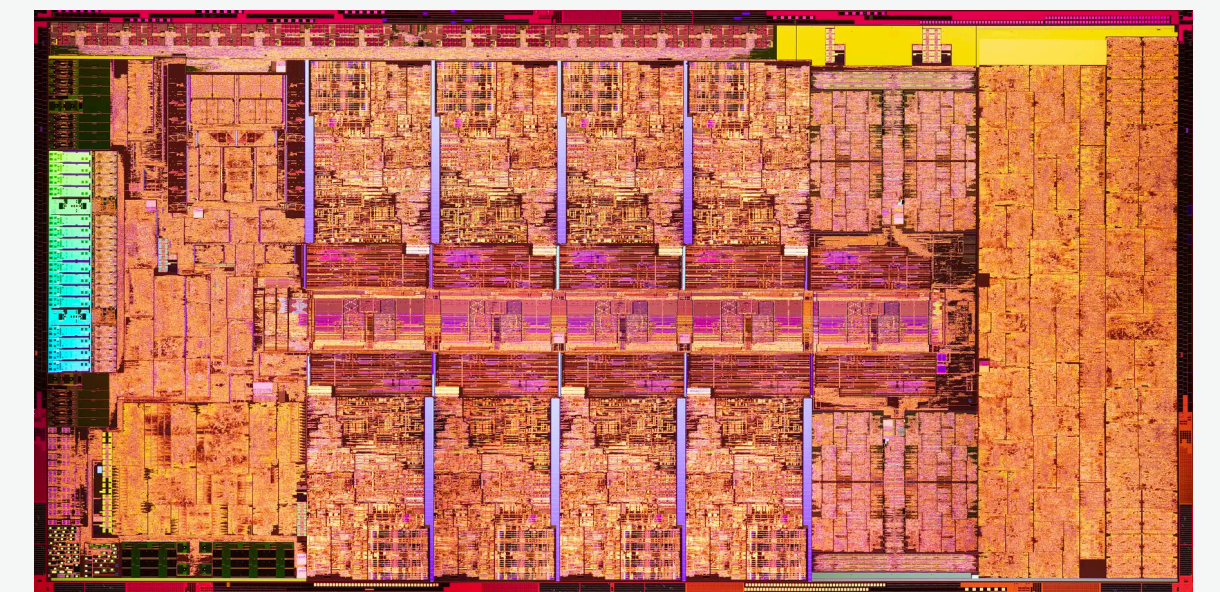
**Problem:** Semiconductor miniaturization is running out of steam.

**Example:** Intel's recent struggles with their 10nm process resulted in **significant delays**.

We're now reaching **physical limits** on miniaturization.



Silicon lattice constant:  
0.543 nanometers  
(5.43 angstroms)



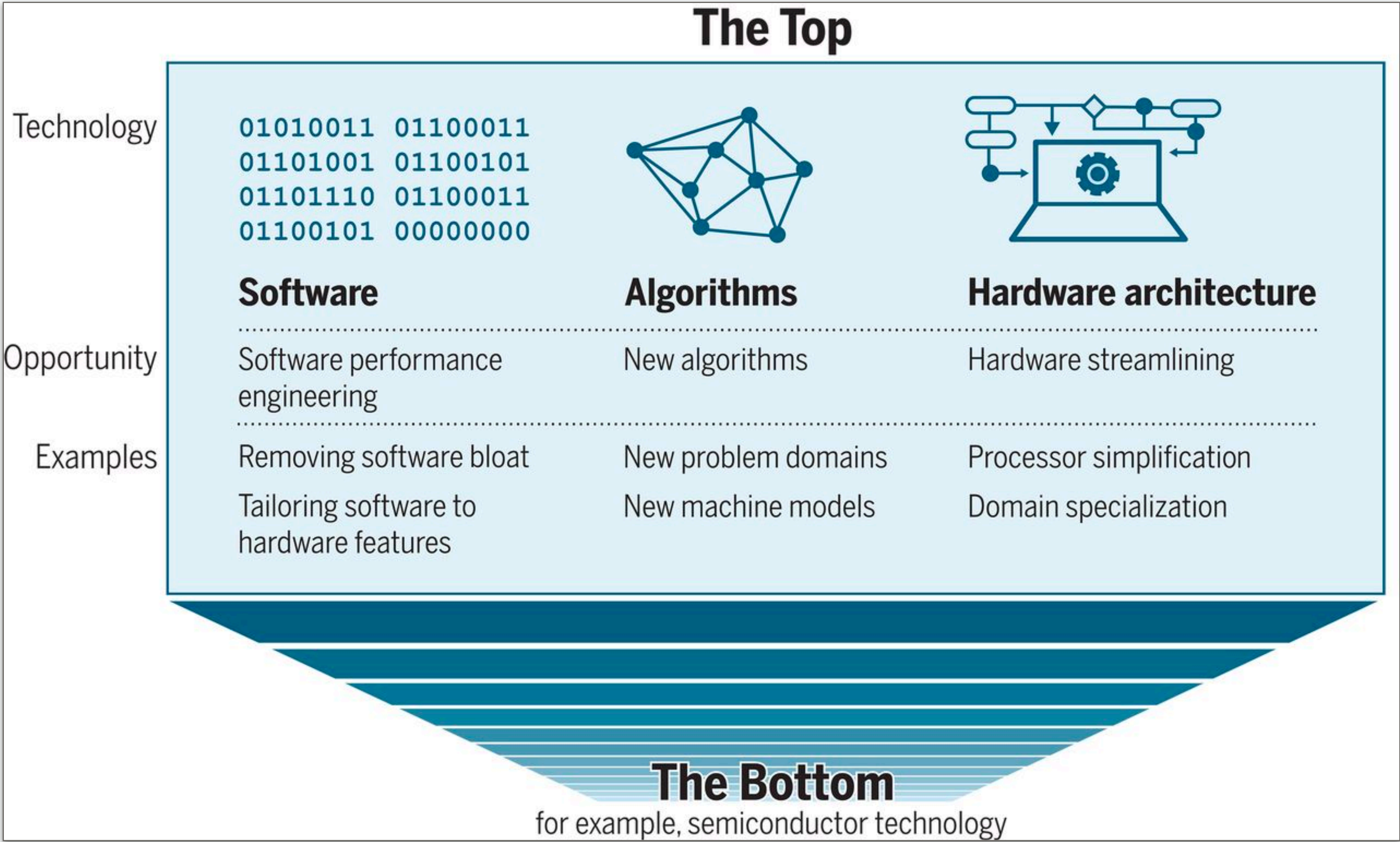
2021 Intel 10nm processor

## What will drive growth in computing performance after Moore's Law ends?



# There's plenty of room at the Top [LTEKLSS20]

We see substantial opportunities for growth in computing performance at the *Top* of the computing stack: **software, algorithms, and hardware architecture.**





# Opportunity in software

Considerable performance is available by addressing **software inefficiencies**.

**Example:**  
Multiply two  
4k-by-4k  
matrices

**Version 1:** Three  
nested loops in  
Python

**Machine:** Amazon  
AWS c4.8xlarge

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak
1	Python	25,552.48	0.005	1	—	0%
2	Java	2,372.68	0.058	11	10.8	0.01%
3	C	542.67	0.253	47	4.4	0.03%
4	Parallel loops	69.80	1.969	366	7.8	0.24%
5	Parallel divide- and-conquer	3.80	36.180	6,727	18.4	4.33%
6	+vectorization	1.10	124.914	23,224	3.5	14.96%
7	+AVX intrinsics	0.41	337.812	62,806	2.7	40.45%



# But software performance is complicated!

---

A modern multicore system contains:

- parallel-processing cores,
- vector units,
- caches,
- prefetchers,
- hyperthreading,
- dynamic frequency scaling,
- GPU's,
- and more!



2021 Intel 10nm processor

How can we enable **average programmers** to **contend** with this complexity and realize the performance gains from **writing fast code**?



# Science-based performance engineering

---

We need technologies that enable a **scientific approach** to software performance engineering.

- **Systems** one can reason about because they obey **simple mathematical properties**, such as monotonicity and composability.
- **Theories** of performance that are borne out in practice.
- **Diagnostic tools** for correctness and performance whose efficacy is **mathematically grounded**.
- **Reliable measurement** and **ubiquitous instrumentation**.

OpenCilk aims to provide these foundations and make it **easy** for programmers to write fast parallel code and educators to **teach** software performance engineering.

# Questions?

---



<https://www.opencilk.org>

Special thanks to the OpenCilk team — I-Ting Angelina Lee, Tim Kaler, Alexandros-Stavros Iliopoulos, John Carr, Dorothy Curtis, Bruce Hoppe, and Charles E. Leiserson — and everyone who has contributed to and supported OpenCilk.