Compiler-Based Timing (CT) For Extremely Fine-Grain Preemptive Parallelism

Supercomputing 2020

Souradip Ghosh, **Michael Cuevas**,
Peter Dinda, Simone Campanoni

Simone
Campanoni
Research
Group

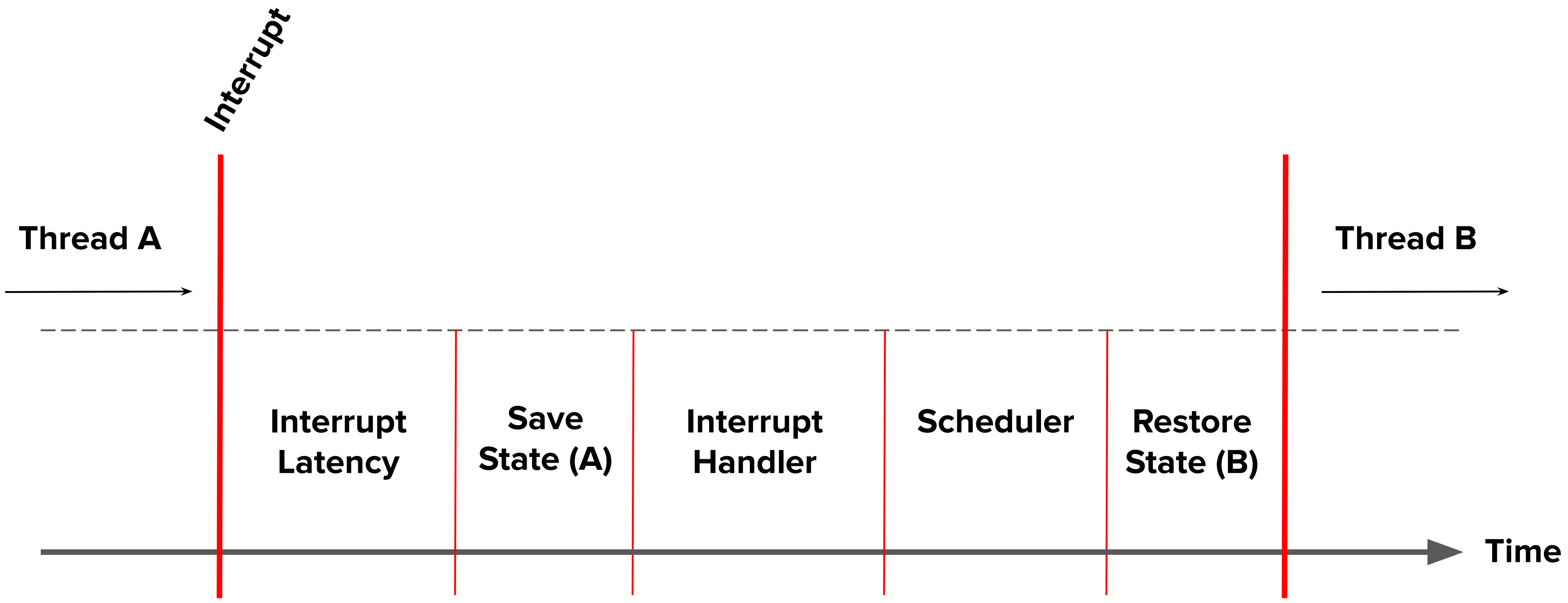
Compiler-Timing in a Nutshell

- **Timing** is **essential** --- notably in **preemptive threads**
- Threads are a useful abstraction for parallel programs, but they utilize **hardware timing** which incurs **high overheads**
- CT introduces a **fully software** approach to timing --- which can be coupled with **lightweight** multitasking mechanisms
- We achieve timing with **6x lower overhead** than hardware timing, which allows for **4x smaller granularity** than preemptive threads

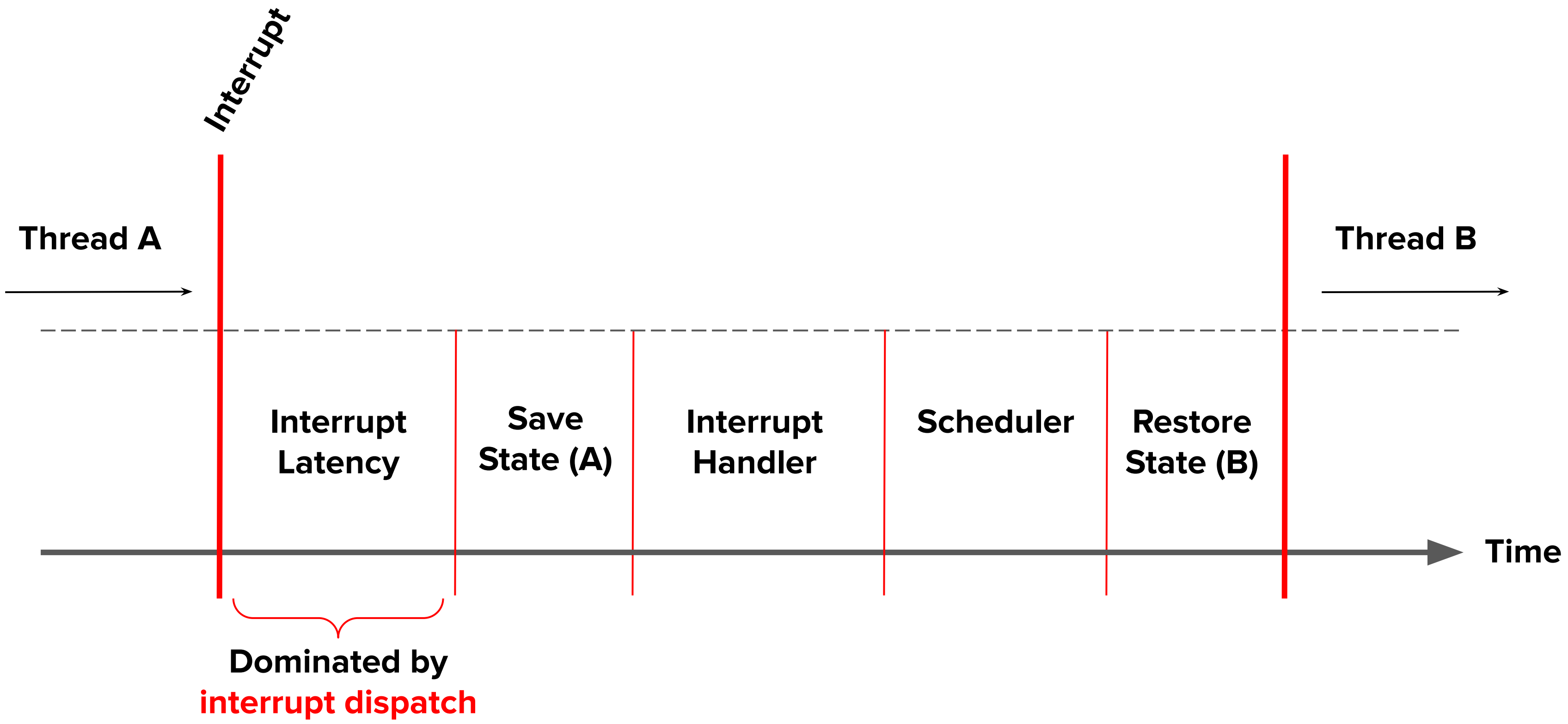
Background: *Parallelism and Its Limits*

- **Fine-grained** parallelism is increasingly necessary to fully utilize many cores
- **Preemptive threading** (i.e. threads) is a convenient **abstraction** for parallel programmers and language implementations
- Preemptive threading currently has **inherent limits** due to **hardware-based timing**

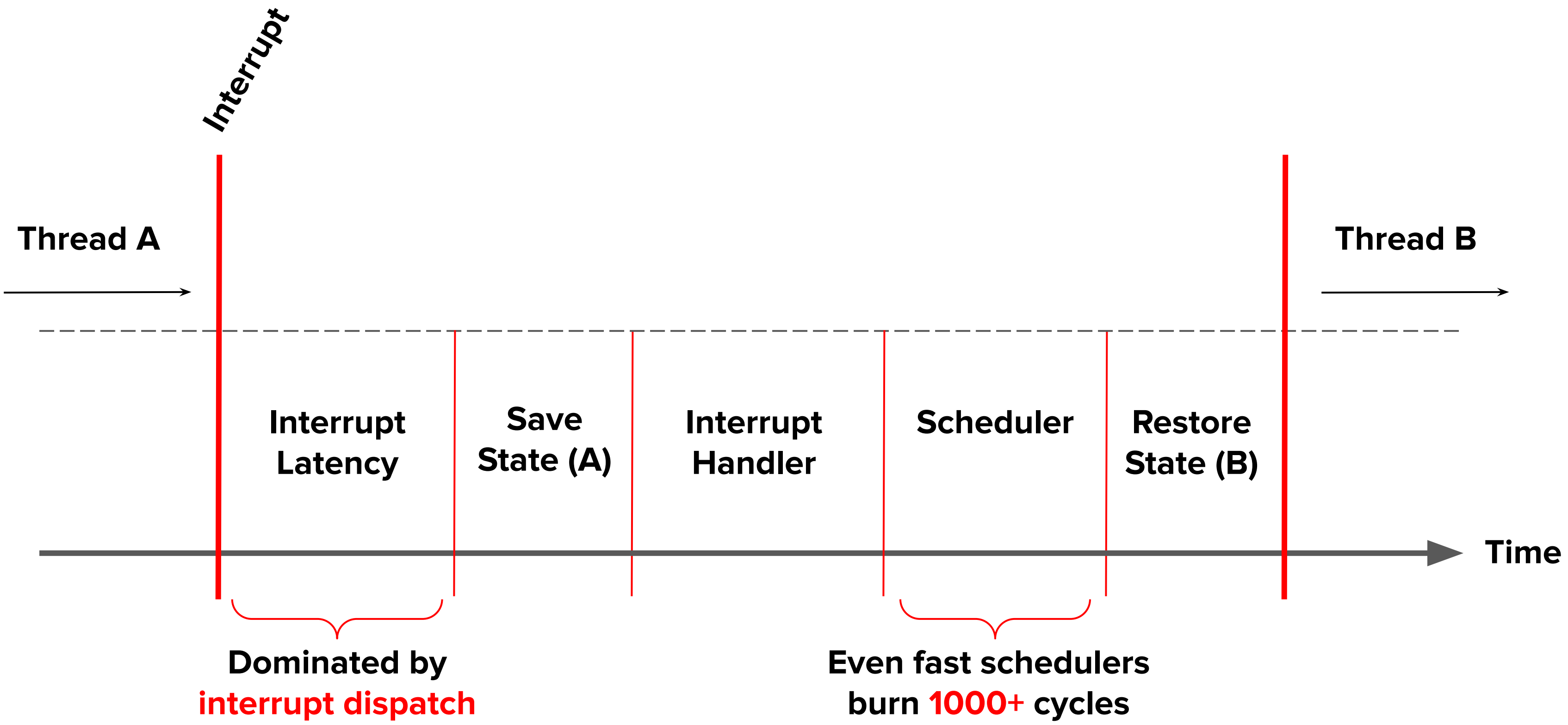
Background: *What are the **pitfalls** of preemption?*



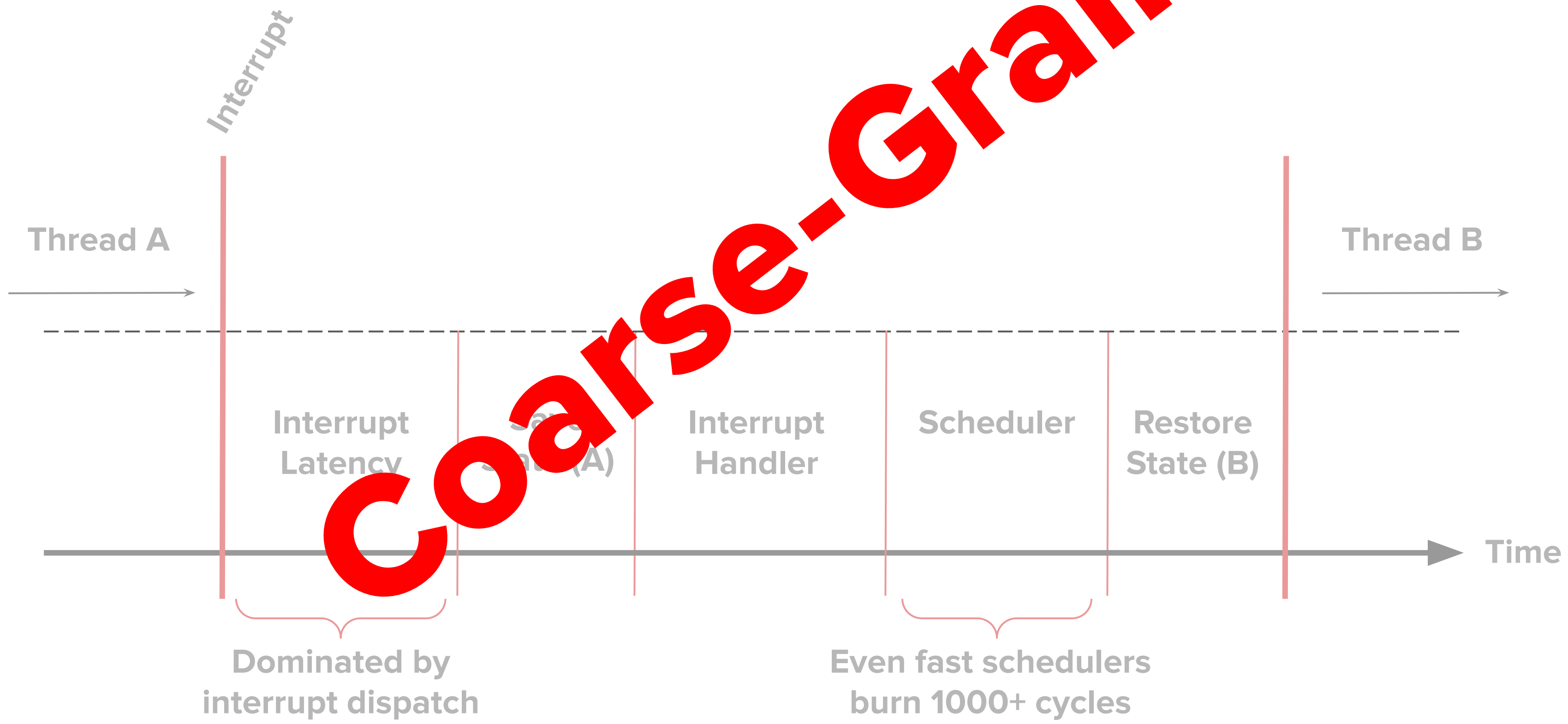
Background: *What are the **pitfalls** of preemption?*



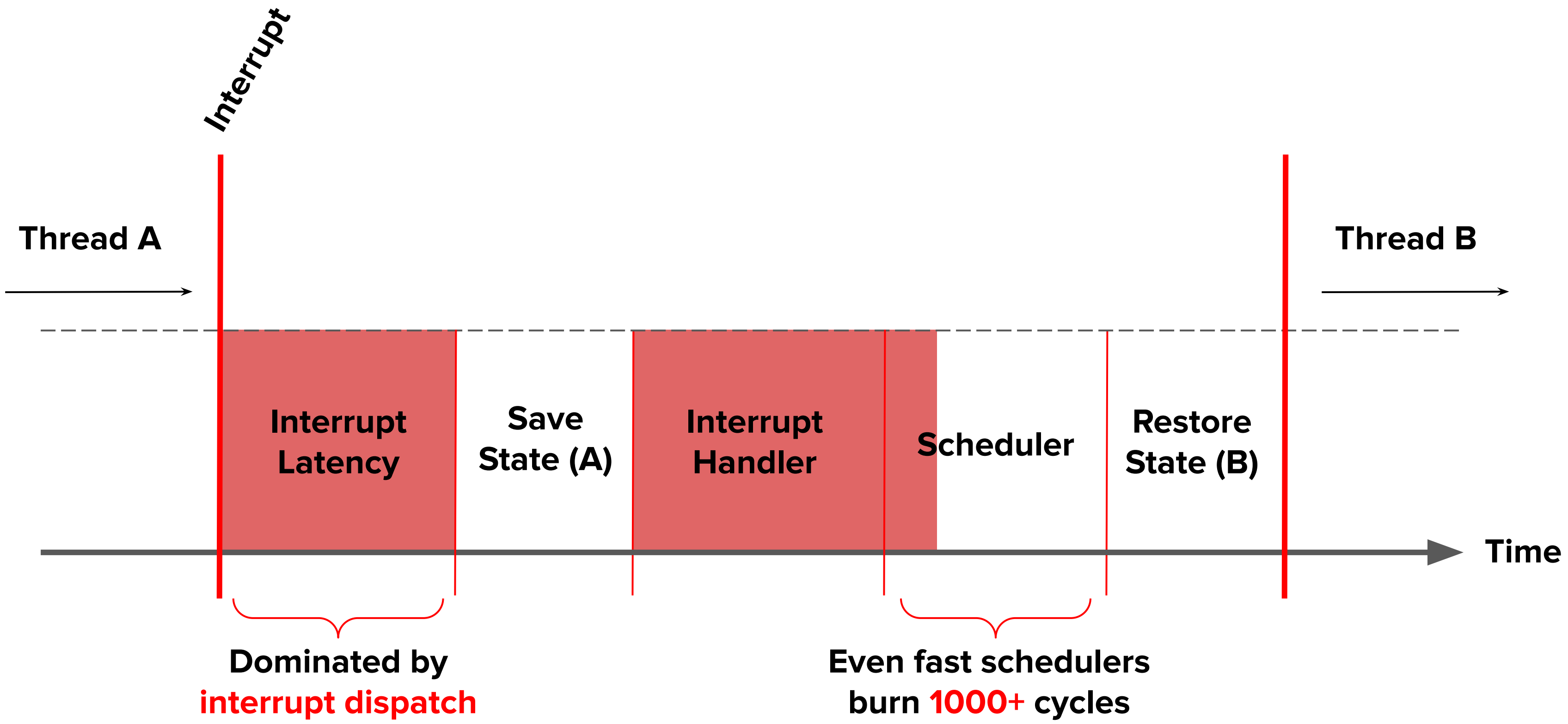
Background: *What are the **pitfalls** of preemption?*



Background: *What are the **pitfalls** of preemption?*



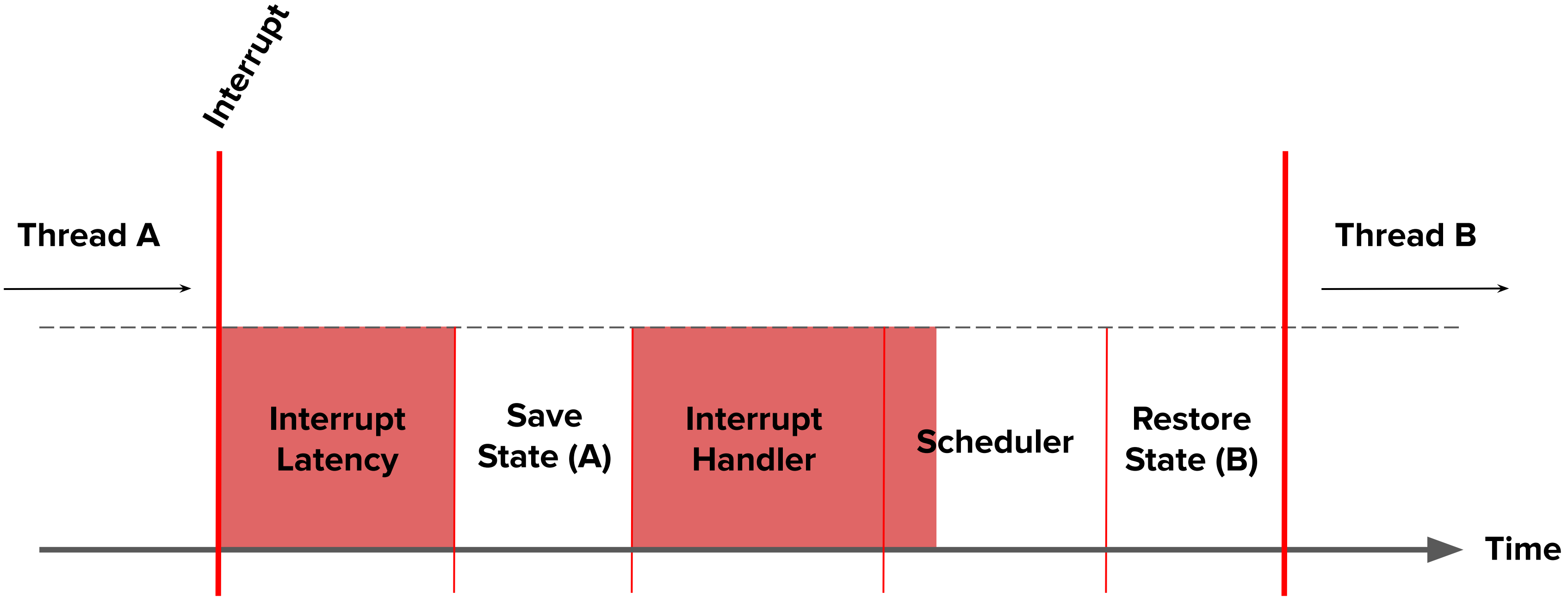
Background: *What are the **pitfalls** of preemption?*



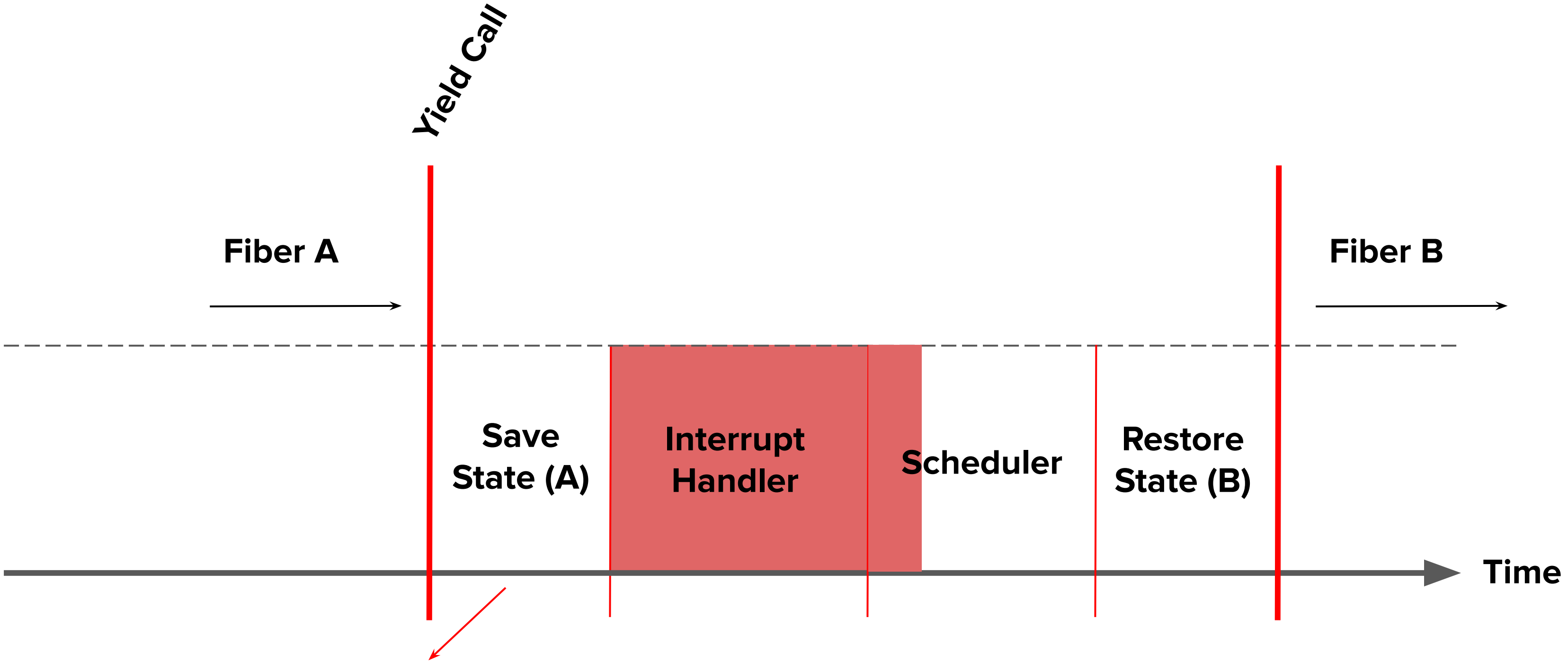
Fibers: *Motivation and Benefits*

- If preemption has inherent overhead, why not **eliminate it**?
- Fibers, or cooperatively scheduled threads, **cannot** be preempted
- Their context switch overhead is **much smaller**

Background: *What are the **pitfalls** of preemption?*

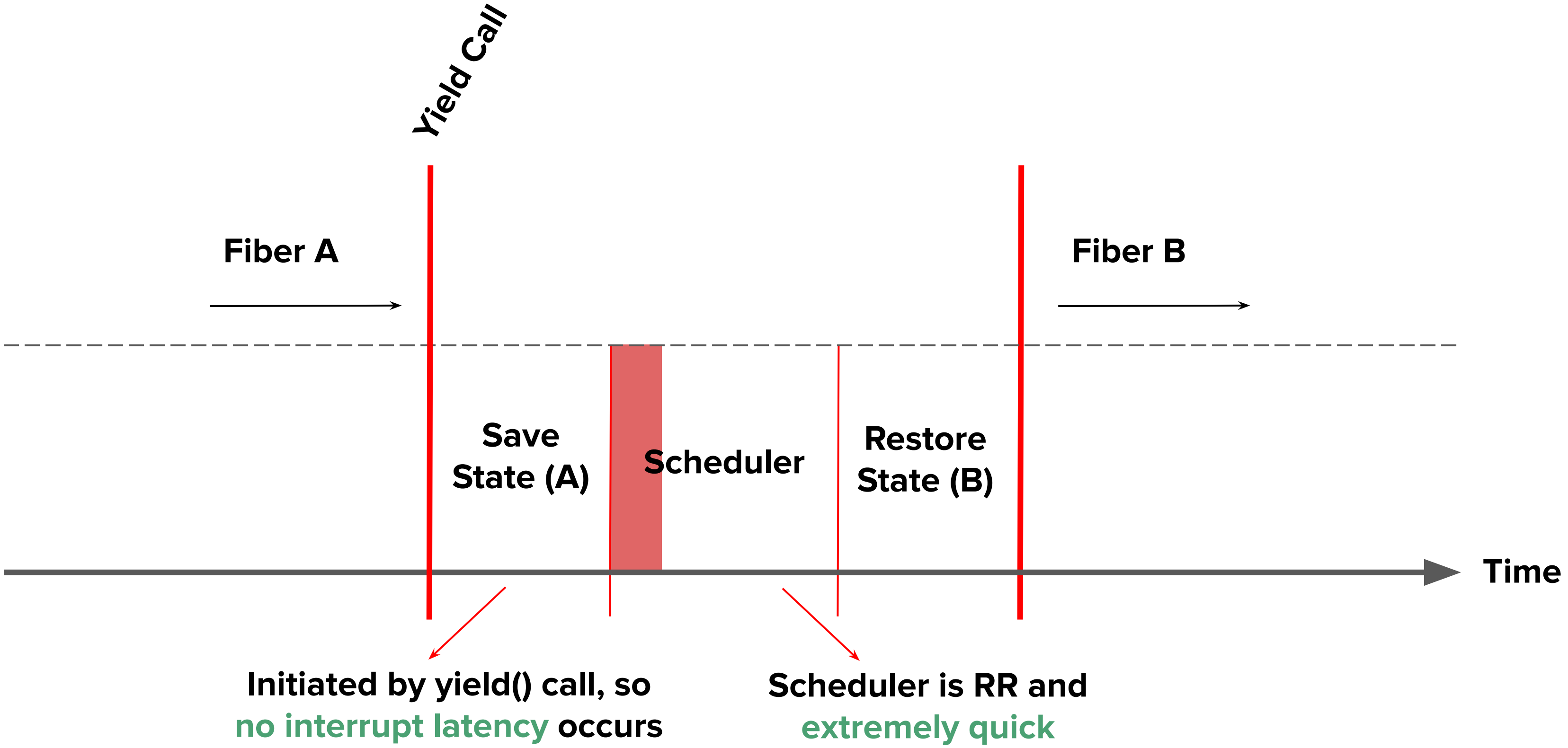


Fibers: *Motivation and Benefits*



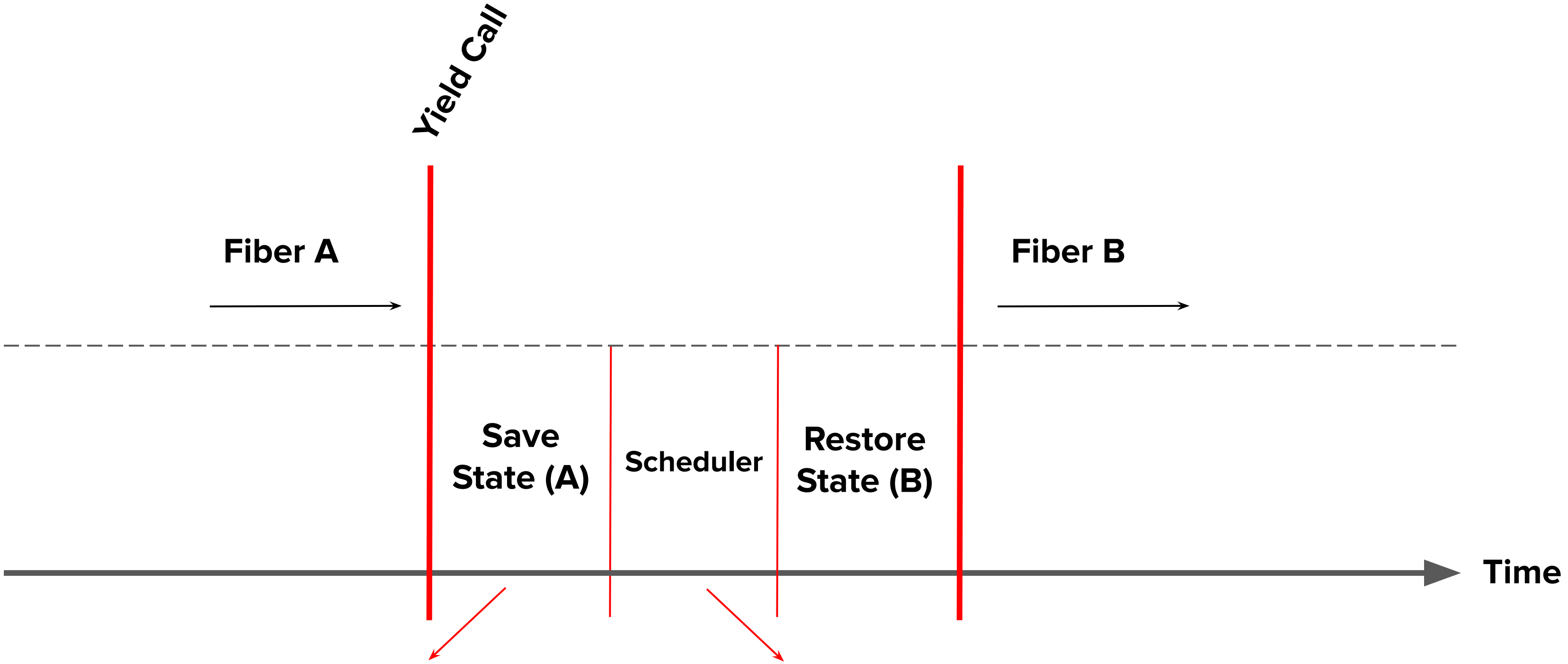
Initiated by yield() call, so no interrupt latency occurs

Fibers: *Motivation and Benefits*



Fibers: *Motivation and Benefits*

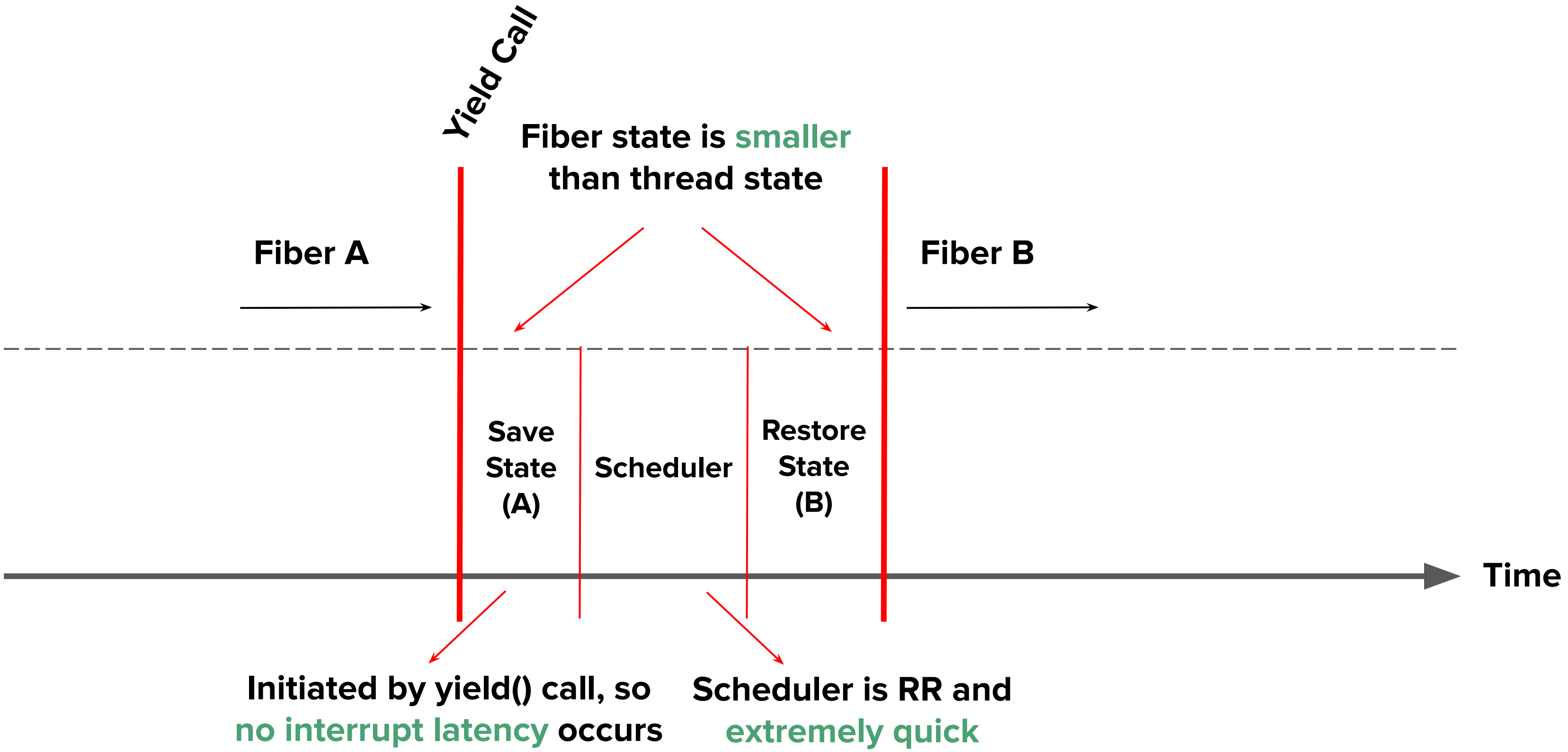
Recall - Nautilus is our custom kernel



Initiated by yield() call, so no interrupt latency occurs

Scheduler is RR and extremely quick

Fibers: *Motivation and Benefits*

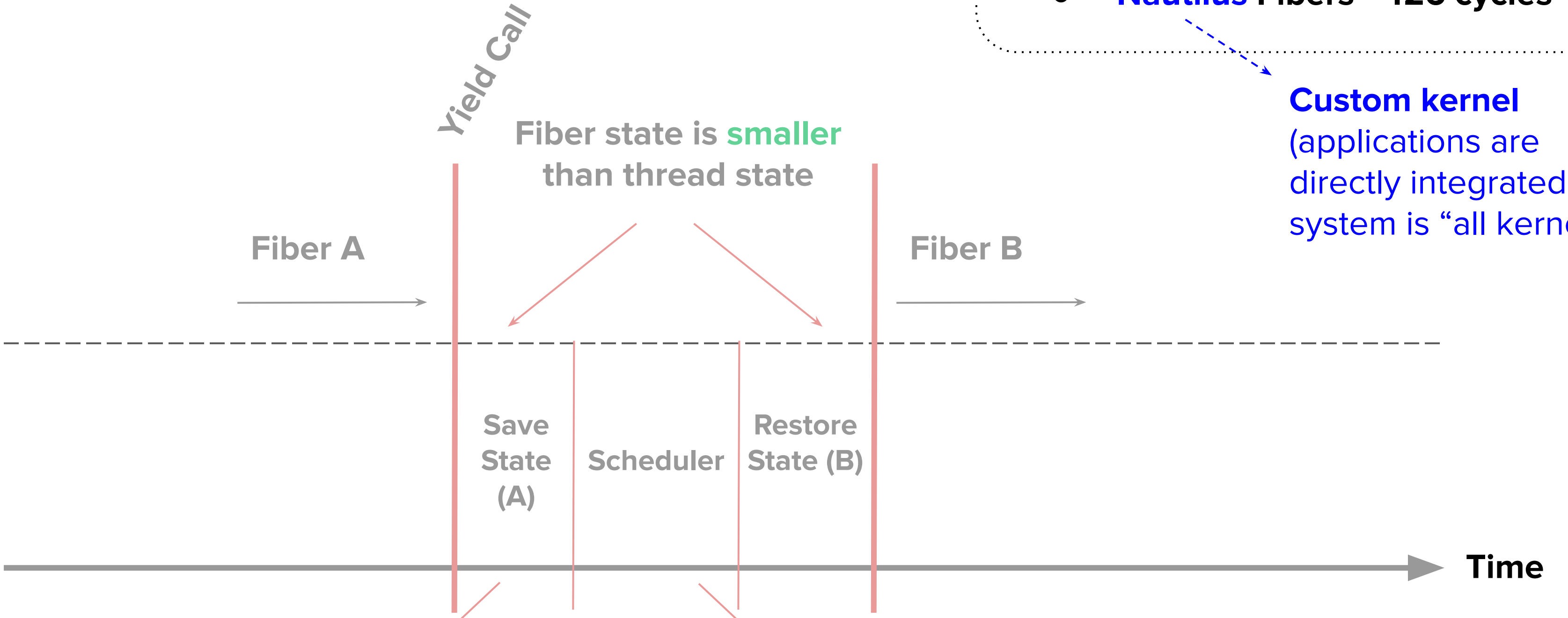


Fibers: *Motivation and Benefits*

Context Switch Costs:

- **Linux Thread - 4896 cycles**
- **Nautilus Thread - 2063 cycles**
- **Nautilus Fibers - 426 cycles**

Custom kernel
(applications are directly integrated --- system is "all kernel")



Initiated by yield() call, so no interrupt latency occurs

Scheduler is RR and extremely quick

Fibers: *Why don't we all switch to fibers?*

- Past experience shows they cause **major issues**

- Programmers struggle with using fibers
  - Pre-2000 MacOS and Windows

- **Relying on programmers** to not make mistakes is a **bad idea**

- One mistake can be **disastrous** for performance

It's established that **preemption is essential**

We have to find a way to **replace** high-overhead hardware **interrupts**

What if we **replace** the high-overhead hardware timing and preemption components with a **fully software** approach?

What if we **replace** the high-overhead hardware timing and preemption components with a **fully software** approach?

Our goal is to create a **low-overhead** preemptive mechanism that enables **fine granularity** control

What if we **replace** the high-overhead hardware timing and preemption components with a **fully software** approach?

How do we **drive** timing in software efficiently and accurately?

Introducing Compiler-Timing (CT)

- A **compiler-OS** codesign that replaces **hardware interrupts**
- **Compiler** --- Uses the middle-end of the compiler to inject callbacks into all code at a specified timing granularity
- **OS** --- Processes callbacks using a custom runtime to drive fast kernel fibers

Compiler: *Goals*

- The compiler needs a **concept of timing**
- **Select instructions** for **instrumentation** to achieve timing
- **Any injected callback** executed at runtime occurs at the specified timing granularity

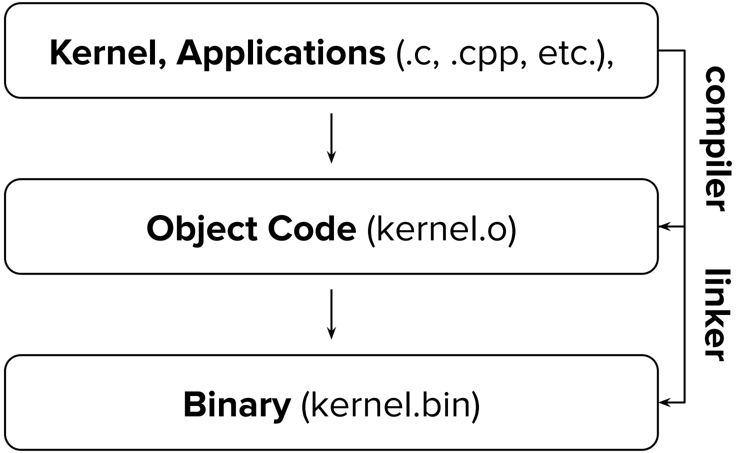
Compiler: *Novelties*

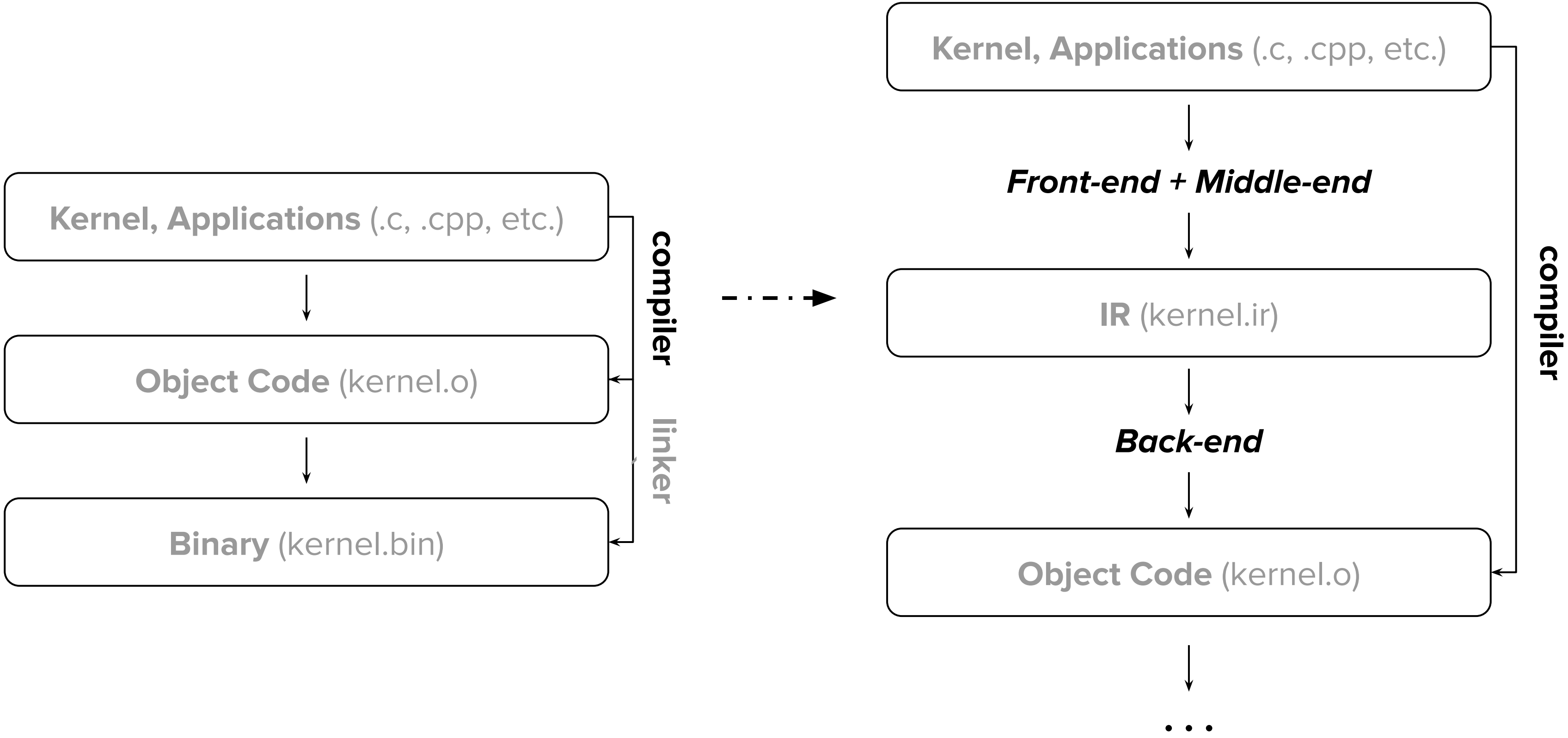
Timing is achieved in the IR

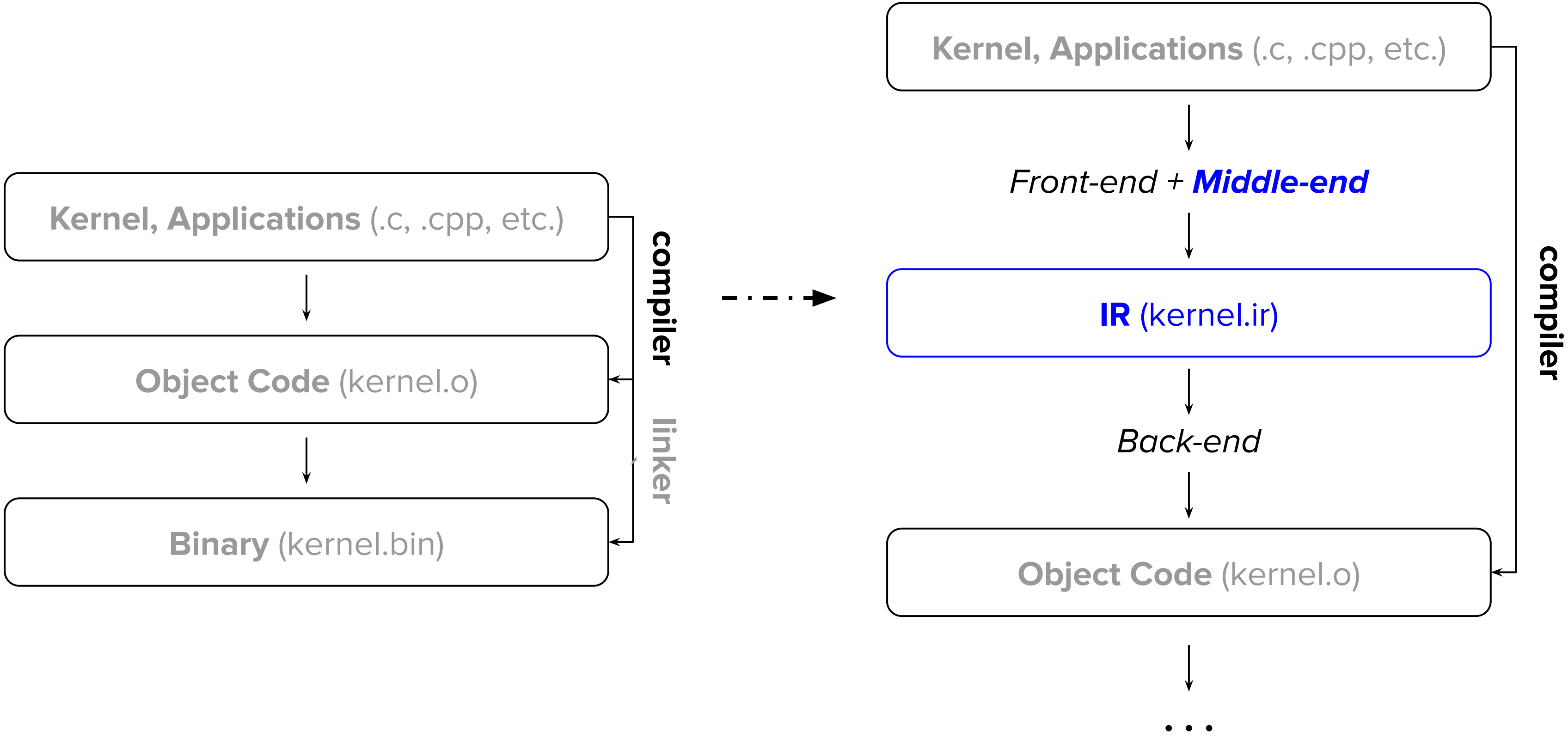
- Analysis/transformation spans the **whole** kernel
  - Whole kernel ➔ kernel + embedded applications

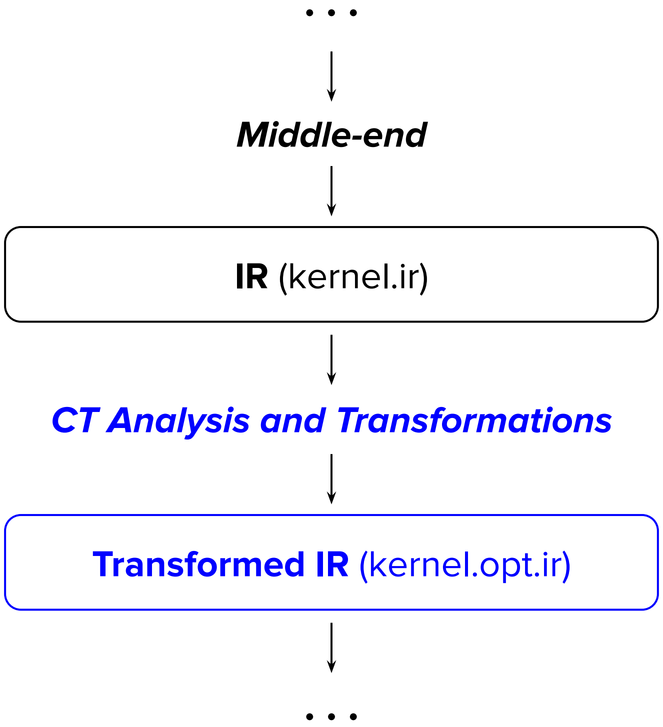
- Analysis of **IR instructions alone** is **sufficient** to achieve timing

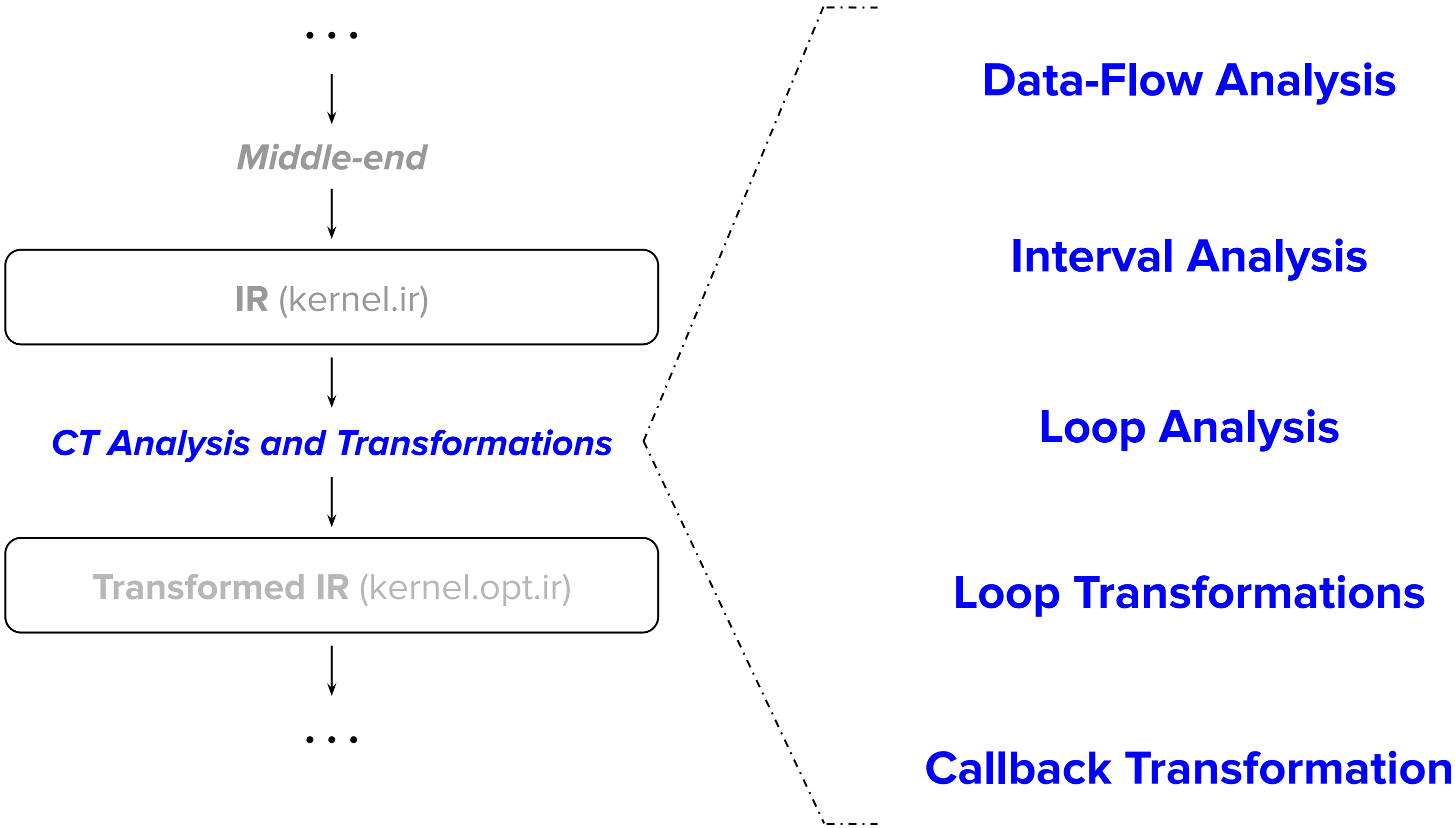
- Transformations can achieve periodic callbacks at runtime, **regardless of control-flow path**

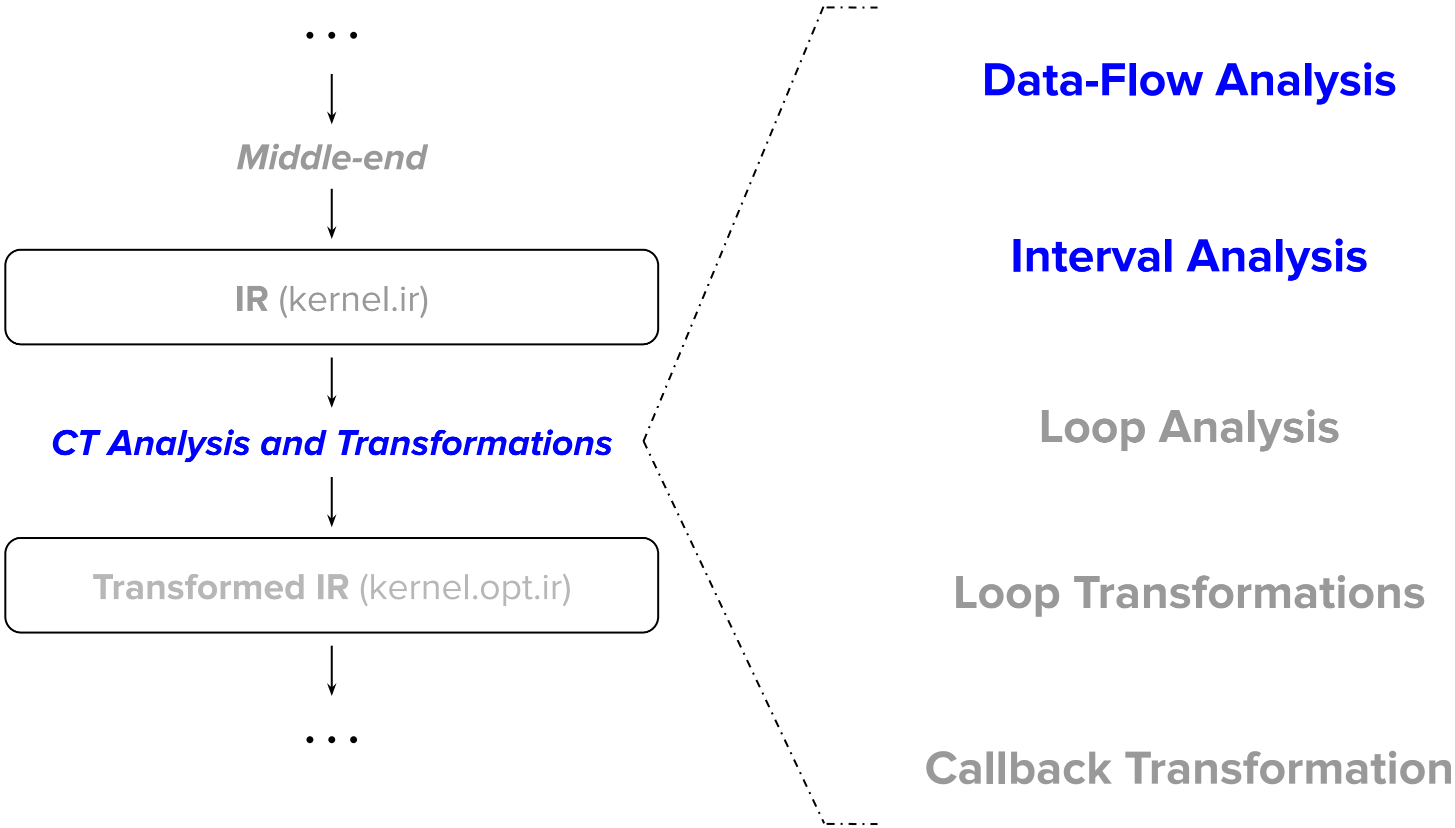

Kernel, Applications (.c, .cpp, etc.),
Object Code (kernel.o)
Binary (kernel.bin)
compiler
linker


Kernel, Applications (.c, .cpp, etc.)
Object Code (kernel.o)
Binary (kernel.bin)
compiler
linker
Kernel, Applications (.c, .cpp, etc.)
Front-end + Middle-end
IR (kernel.ir)
Back-end
Object Code (kernel.o)
...
compiler


Kernel, Applications (.c, .cpp, etc.)
Object Code (kernel.o)
Binary (kernel.bin)
compiler
linker
Kernel, Applications (.c, .cpp, etc.)
Front-end + Middle-end
IR (kernel.ir)
Back-end
Object Code (kernel.o)
compiler
. . .


. . .
Middle-end
IR (kernel.ir)
CT Analysis and Transformations
Transformed IR (kernel.opt.ir)
. . .


...
Middle-end
IR (kernel.ir)
CT Analysis and Transformations
Transformed IR (kernel.opt.ir)
...
Data-Flow Analysis
Interval Analysis
Loop Analysis
Loop Transformations
Callback Transformation


...
Middle-end
IR (kernel.ir)
CT Analysis and Transformations
Transformed IR (kernel.opt.ir)
...
Data-Flow Analysis
Interval Analysis
Loop Analysis
Loop Transformations
Callback Transformation

Compiler: *Timing in the IR*

- To have a conception of **timing** in the middle-end of a compiler, we **map IR instructions** to a clock-cycle **latency**
  - IR ➔ x86 mapping --- 1-to-1, 1-to-many, many-to-1, 1-to-none
  - Known/measured latencies of x86 instructions are applied to the IR

Compiler: *Timing in the IR*

- To have a conception of **timing** in the middle-end of a compiler, we **map IR instructions** to a clock-cycle **latency**
  - IR ➔ x86 mapping --- 1-to-1, 1-to-many, many-to-1, 1-to-none
  - Known/measured latencies of x86 instructions are applied to the IR

	Operands	µops	Unit	Latency
Arithmetic instructions				
ADD SUB	r,r/i	1	IP0/1	1

Compiler: *Timing in the IR*

- There are **known inaccuracies** to this approach --- but analyzing using this approach in the **aggregate is sufficient** for CT

Compiler: *Accumulated Latencies*

- To estimate timing in the IR **across code regions**, single-instruction latencies need to be **propagated**

Compiler: *Accumulated Latencies*

- To estimate timing in the IR **across code regions**, single-instruction latencies need to be **propagated**

- The **accumulated latency** of an IR instruction, $\mathbf{I}$, is a propagated aggregate of clock-cycle latencies from any given point through $\mathbf{I}$, calculated along all possible control-flows

Compiler: *Data Flow Analysis (DFA)*

- We utilize a simple, custom **DFA** to **calculate** the **accumulated latency** of any given instruction in the IR

Compiler: *Data Flow Analysis (DFA)*

- We utilize a simple, custom **DFA** to **calculate** the **accumulated latency** of any given instruction in the IR

- Mechanics:
  - Depends on **preceding instructions**' accumulated latencies
  - Complex control-flows are handled via simple **expectation**

Compiler: *Interval Analysis*

- Interval analysis **selects instructions** to instrument

Compiler: *Interval Analysis*

- Interval analysis **selects instructions** to instrument

- Mechanics:
  - **DFA is applied** starting from a heuristically chosen point
  - A code region is **selected** when propagating accumulated latency exceeds the specified granularity --- an **interval**
  - **Loops** are handled with transforms on top of interval analysis

- Ensures periodic run-time behavior **regardless of control-flow**

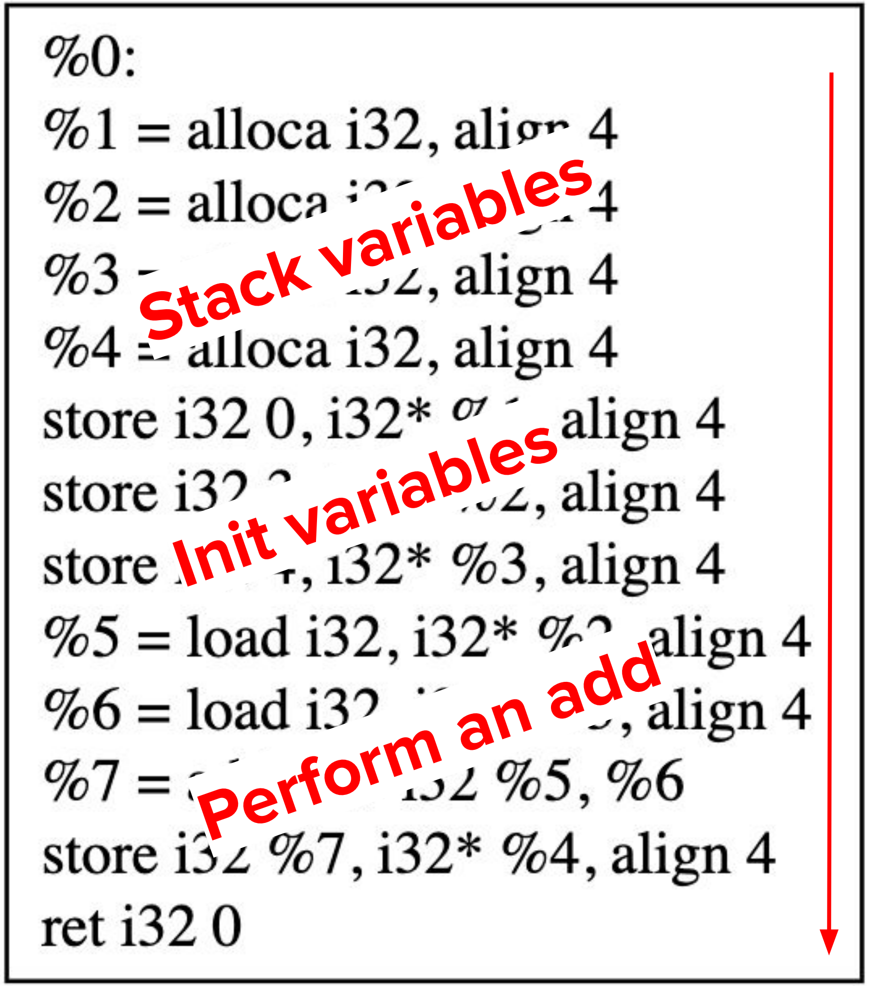
Compiler: *Inte*

- Interval analy … nt
- Mechanics:
  - **DFA is app** … nt
  - A code reg … ated latency exceeds th…
  - **Loops** are … nalysis
- Ensures peri… **f control-flow**

```
%0:
%1 = alloca i32, align 4
%2 = alloca i32, align 4
%3 = alloca i32, align 4
%4 = alloca i32, align 4
store i32 0, i32* %1, align 4
store i32 3, i32* %2, align 4
store i32 4, i32* %3, align 4
%5 = load i32, i32* %2, align 4
%6 = load i32, i32* %3, align 4
%7 = add nsw i32 %5, %6
store i32 %7, i32* %4, align 4
ret i32 0
```

Compiler: *Inte*

- Interval analy ent
- Mechanics:
  - **DFA is app** nt
  - A code reg ated latency exceeds th
  - **Loops** are l nalysis
- Ensures peri **f control-flow**



Compiler: *Inte*

- Interval analy ent
- Mechanics:
  - **DFA is app** nt
  - A code reg ated latency exceeds th
  - **Loops** are h nalysis
- Ensures peri **f control-flow**

```
%0:
%1 = alloca i32, align 4
%2 = alloca i32, align 4
%3 = alloca i32, align 4
%4 = alloca i32, align 4
store i32 0, i32* %1, align 4
store i32 3, i32* %2, align 4
store i32 4, i32* %3, align 4
%5 = load i32, i32* %2, align 4
%6 = load i32, i32* %3, align 4
%7 = add nsw i32 %5, %6
store i32 %7, i32* %4, align 4
ret i32 0
```

Compiler: *Inte*

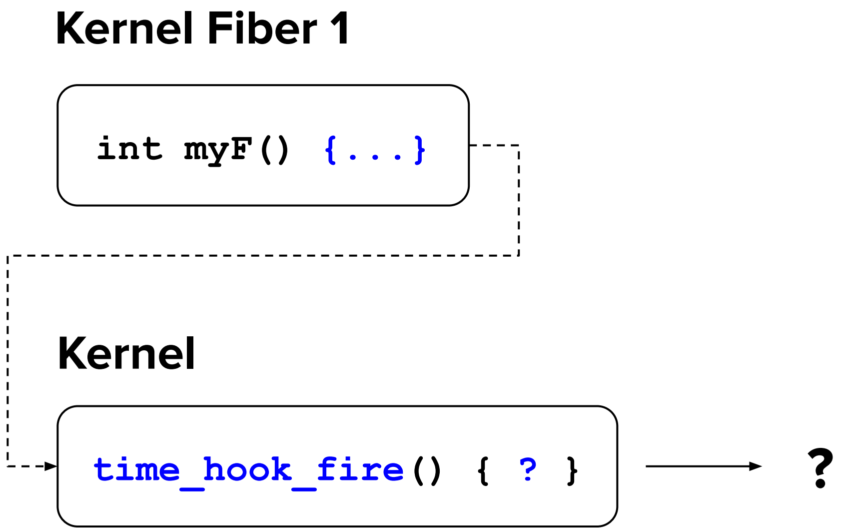
- Interval analy … ent
- Mechanics:
  - **DFA is app** … nt
  - A code reg … ated latency exceeds th
  - **Loops** are … nalysis
- Ensures peri … **f control-flow**

```
%0:
%1 = alloca i32, align 4
%2 = alloca i32, align 4
%3 = alloca i32, align 4
%4 = alloca i32, align 4
store i32 0, i32* %1, align 4
*store i32 3, i32* %2, align 4   9
store i32 4, i32* %3, align 4
%5 = load i32, i32* %2, align 4
%6 = load i32, i32* %3, align 4
%7 = add nsw i32 %5, %6
store i32 %7, i32* %4, align 4
ret i32 0
```

```
int myF()
{
    int foo = 42,
        bar = 24,
        baz = foo + bar;

    return baz;
}
```

CT Compiler →

```
int myF()
{
    int foo = 42,
        bar = 24,
        baz = foo + bar;

    time_hook_fire();

    return baz;
}
```



Kernel Fiber 1
int myF() {...}
Kernel
time_hook_fire() { ? }
?

Runtime: *Kernel Time-Hooking*

- We introduce a new **runtime** interface into the kernel: **time-hooking** --- that properly **handles injected callbacks**

- Goals:
  - **Manage deadlines**
  - **Trigger context switches**

Runtime: *Kernel Time-Hooking*

- `time_hook_fire()` is a "**pseudo**" **interrupt handler**
  - **Processes** deadline and **triggers** the context switch
  - Designed to **avoid overheads** and **scale**
  - **Fast** processing --- **150 - 200** clock cycles

Kernel Fiber 1

```
int myF() {...}
```

Only function call latency, not interrupt latency!

Time-Hook (Kernel)

```
time_hook_fire() {   }
```

```
/* Fast processing */
...
if (deadline_expired())
    yield();
```

Kernel Fiber 1

```
int myF() {...}
```

Only function call latency, not interrupt latency!

Time-Hook (Kernel)

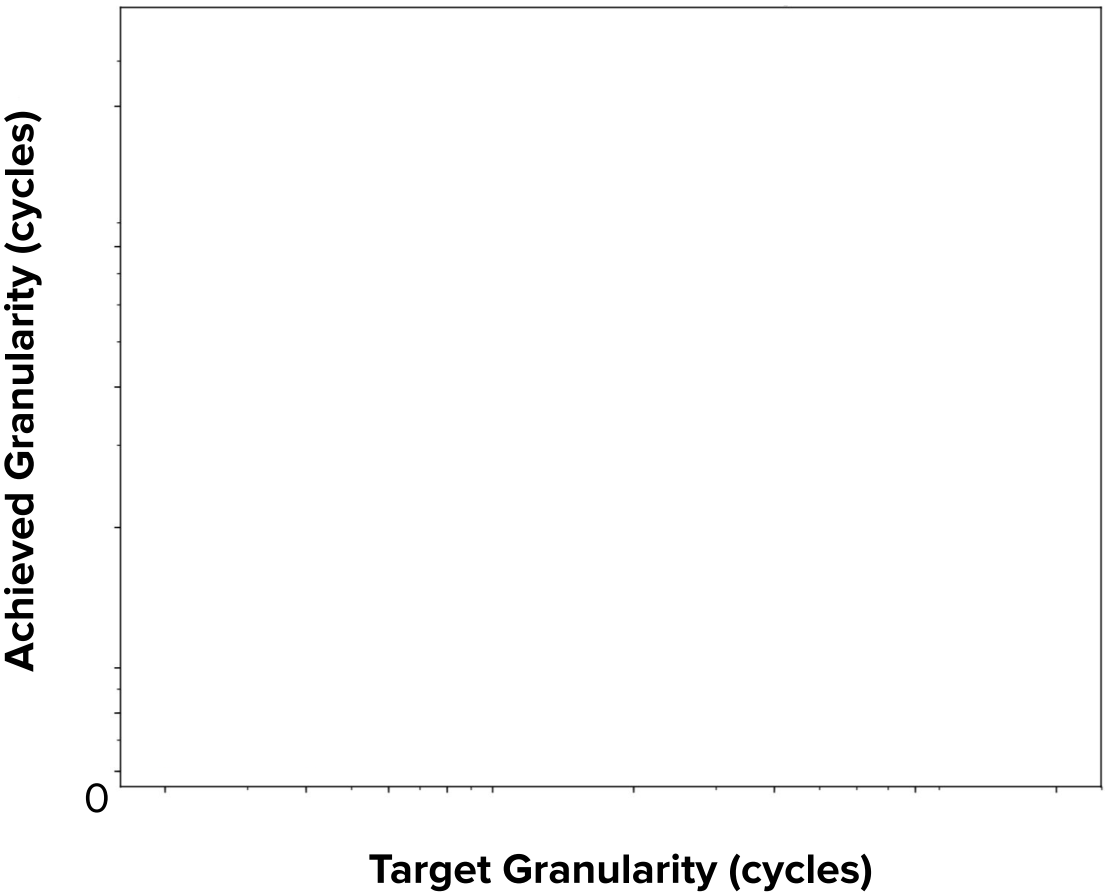
```
time_hook_fire() {  }
```

```
/* Fast processing */
...
if (deadline_expired())
    yield();
```

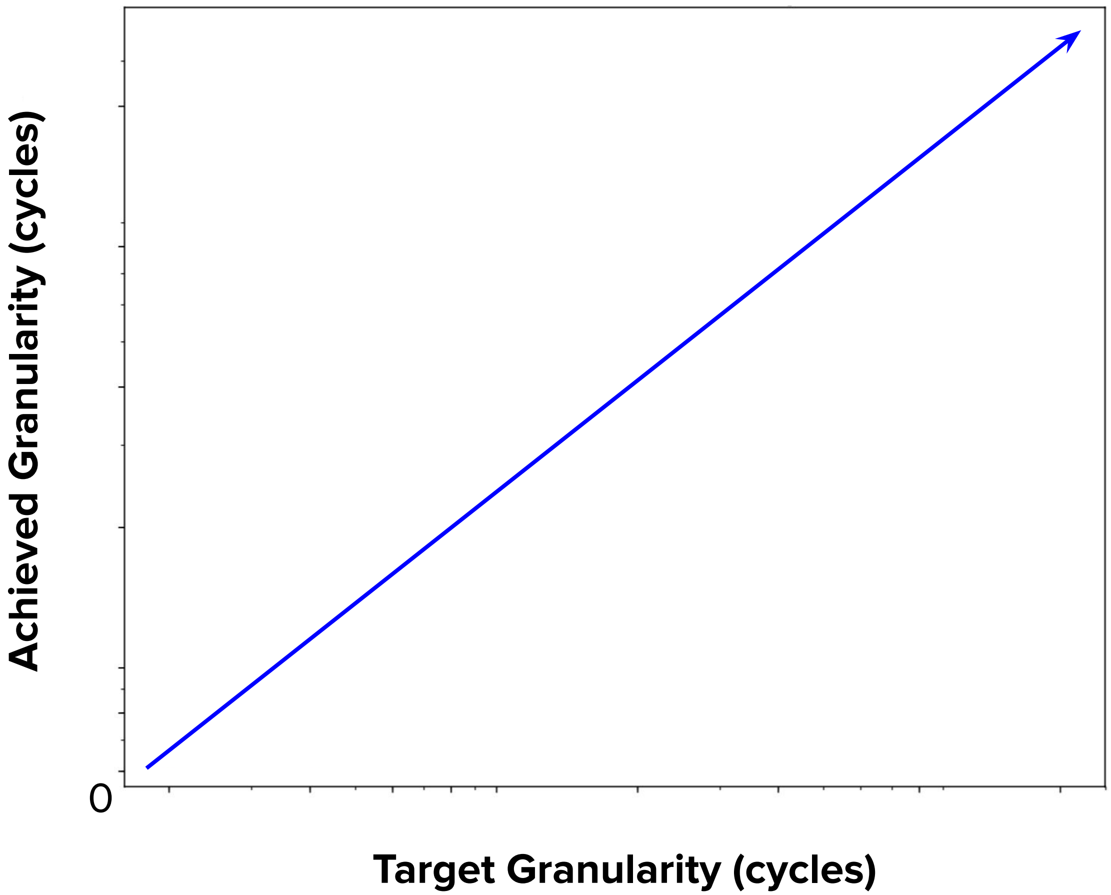
Kernel Fiber 2

```
int myF2() {...}
```

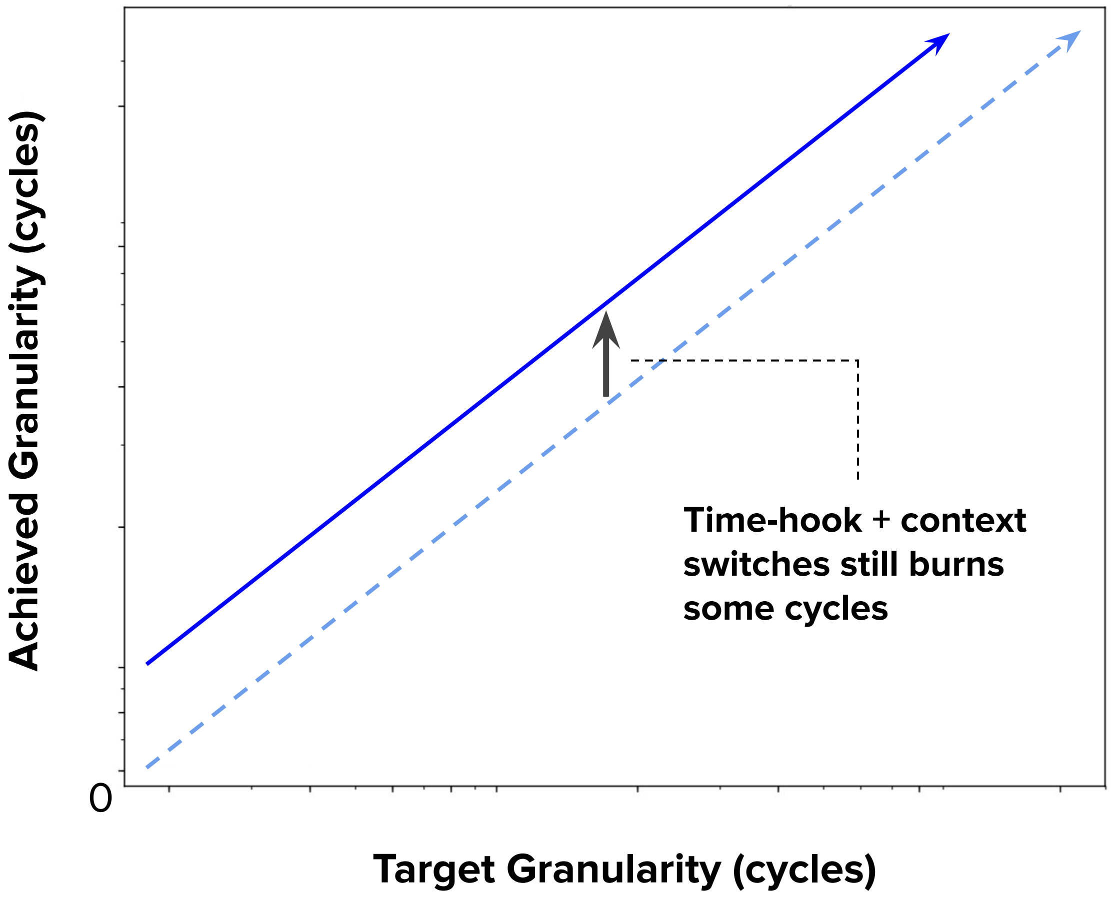
Accuracy of Compiler-Timing Driven Fibers per Context Switch



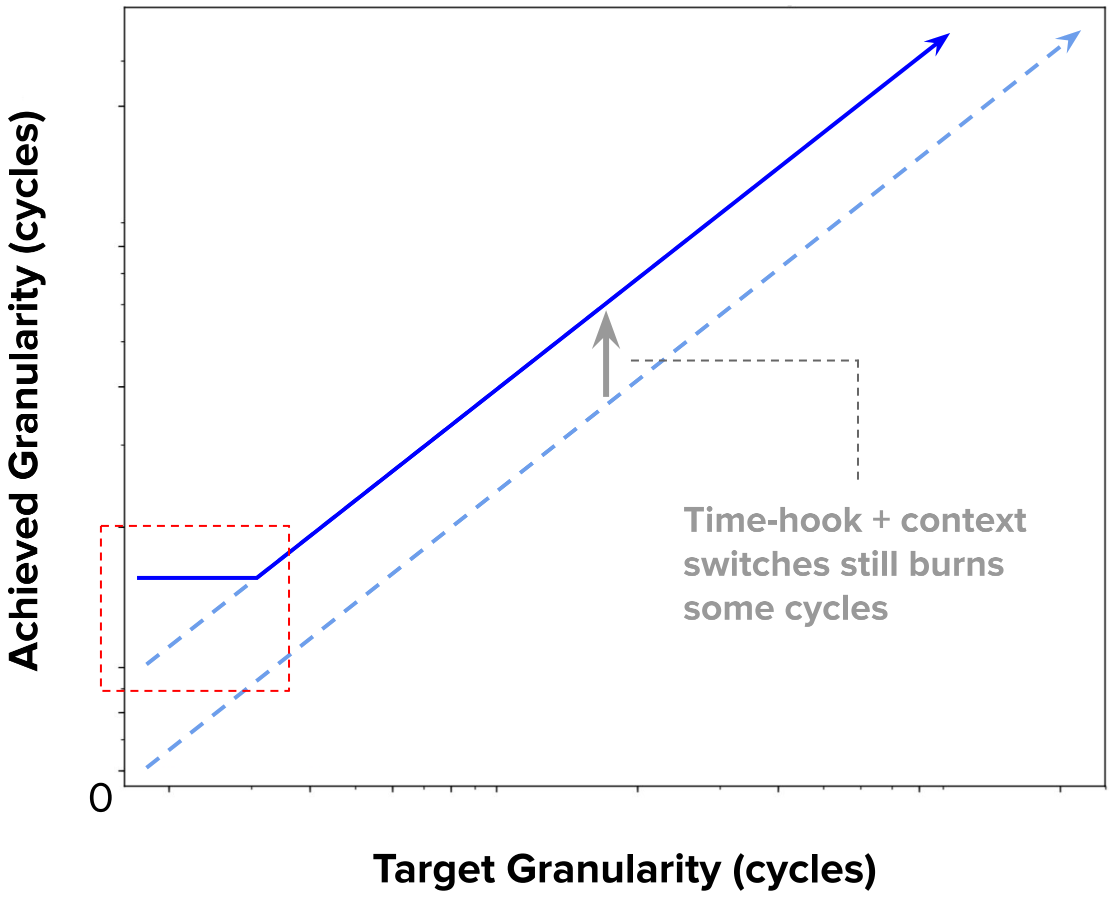
Accuracy of Compiler-Timing Driven Fibers per Context Switch



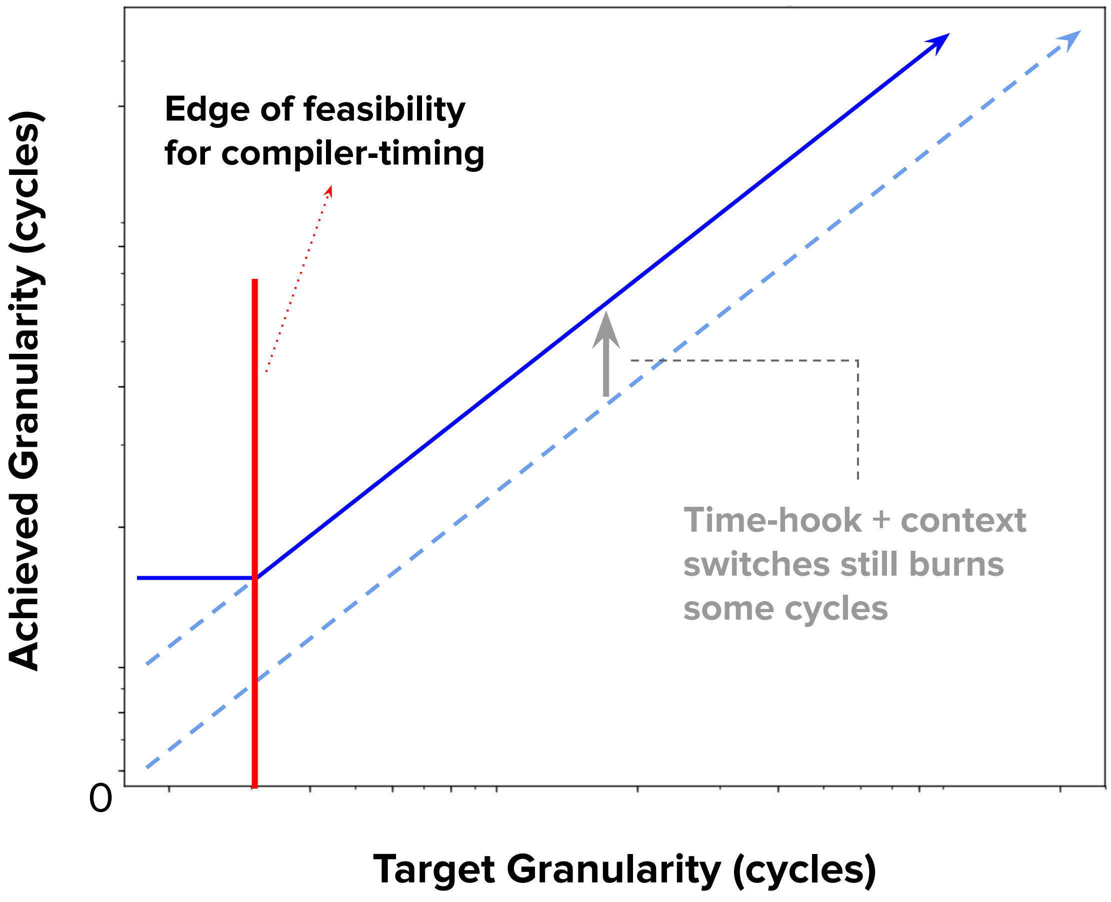
Accuracy of Compiler-Timing Driven Fibers per Context Switch



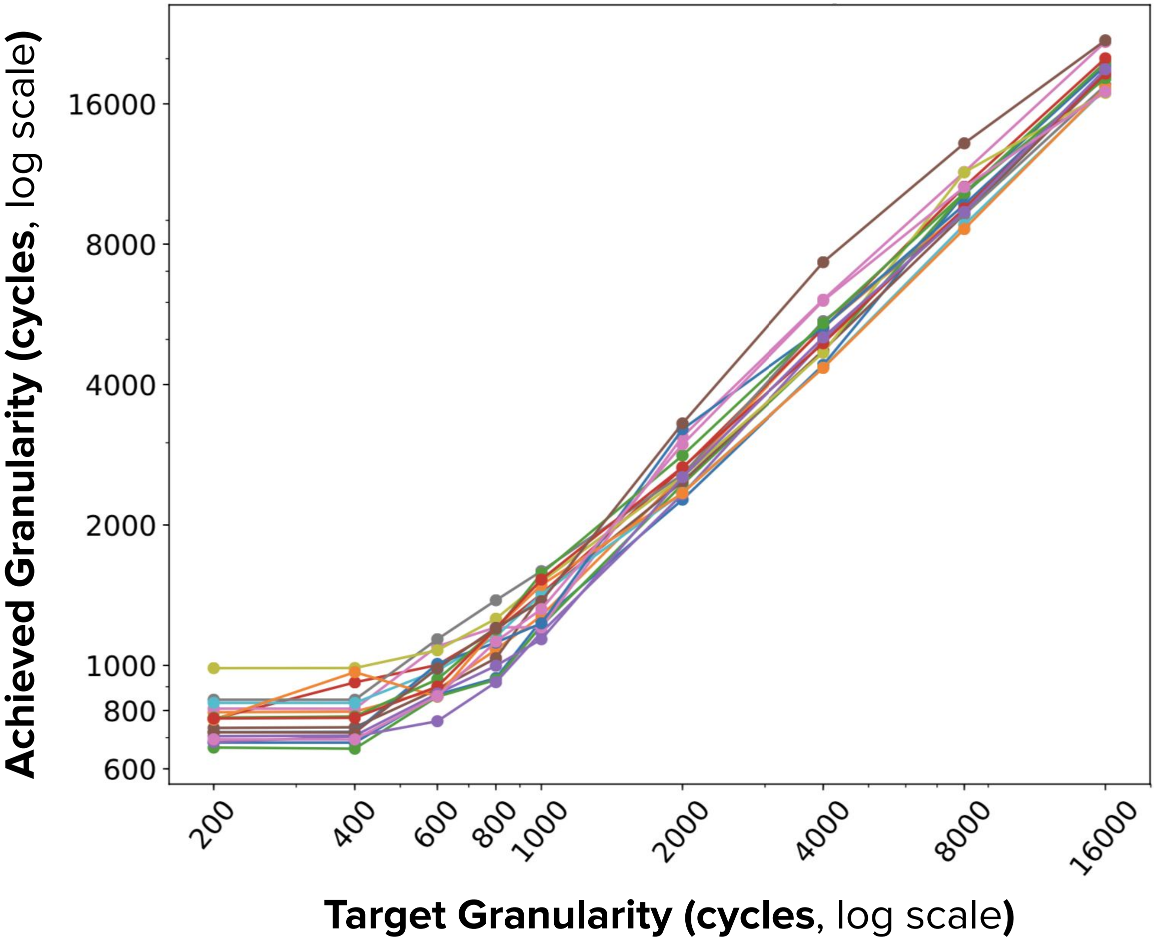
Accuracy of Compiler-Timing Driven Fibers per Context Switch



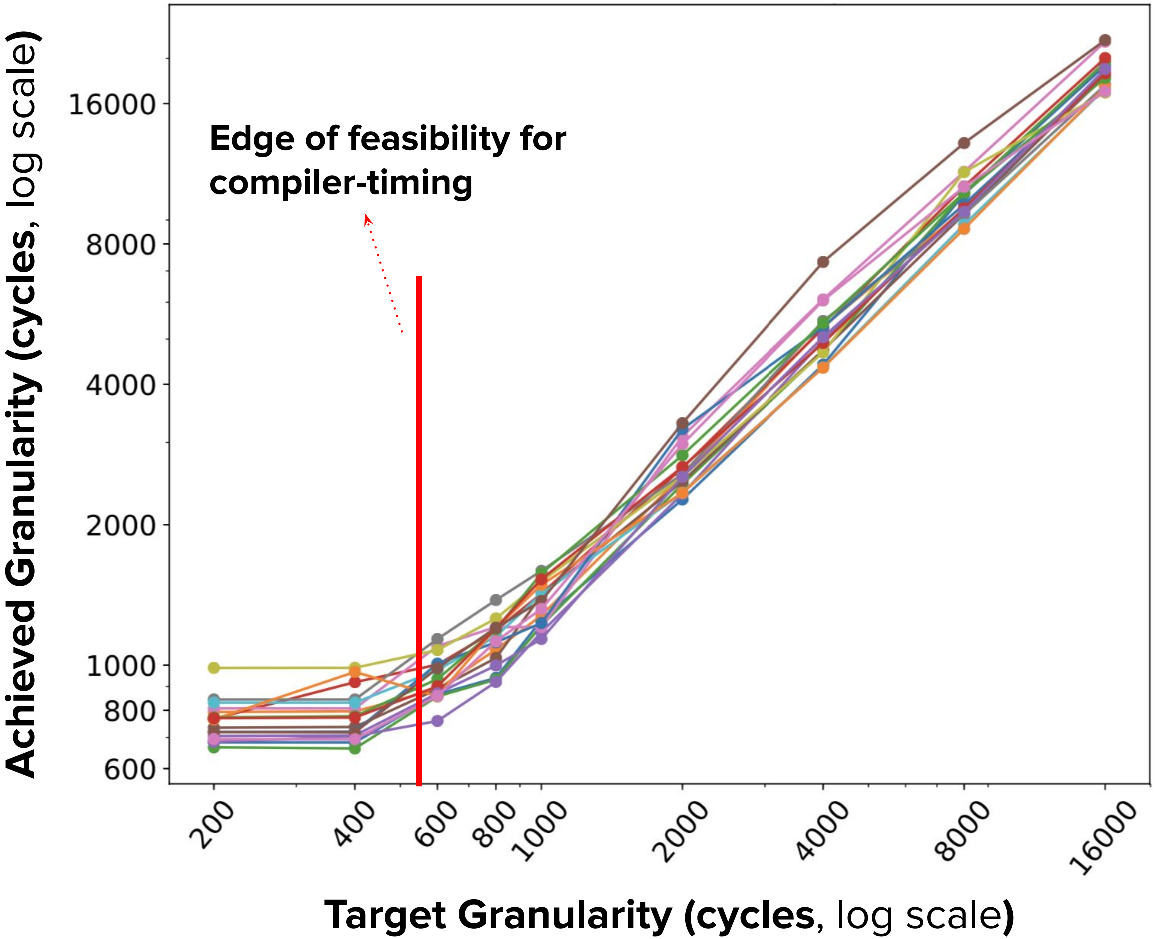
Accuracy of Compiler-Timing Driven Fibers per Context Switch



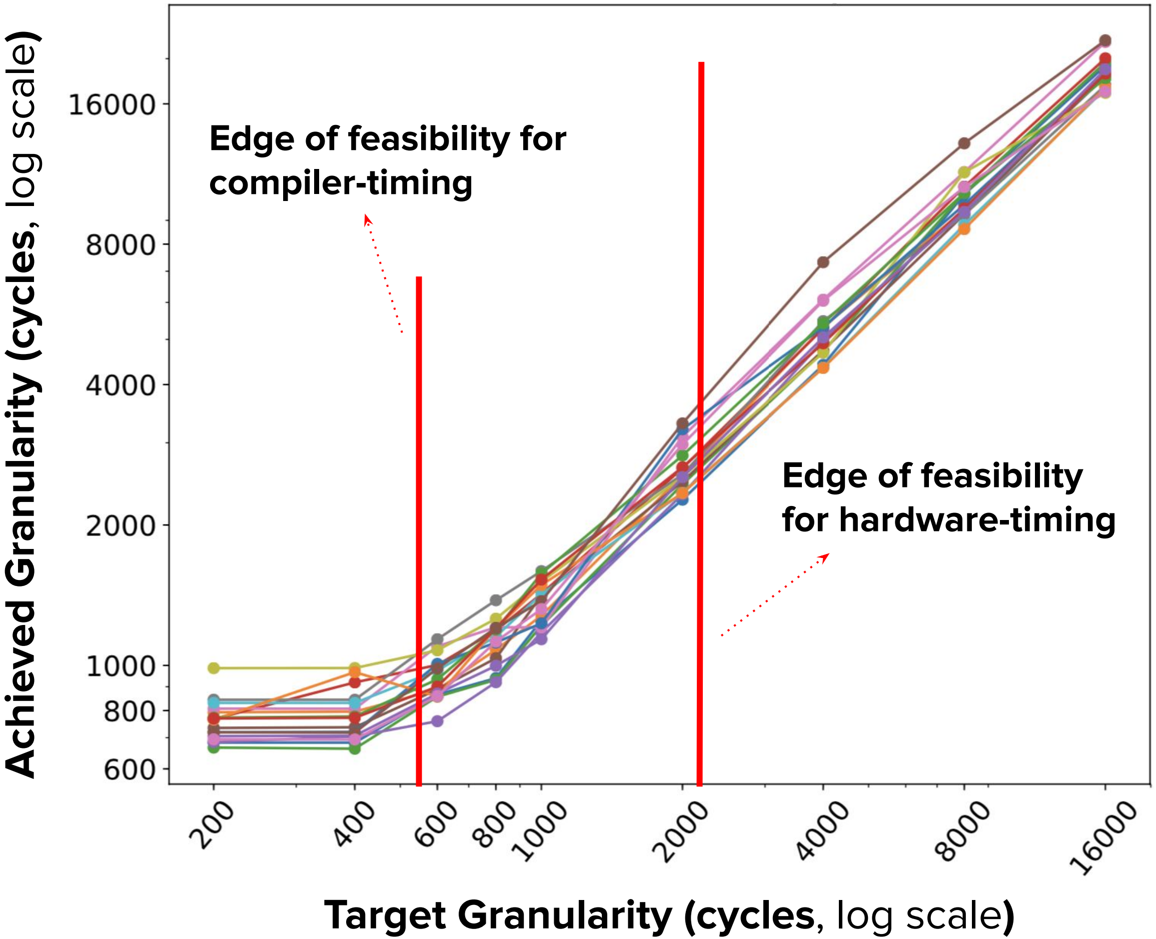
Accuracy of Compiler-Timing Driven Fibers per Context Switch



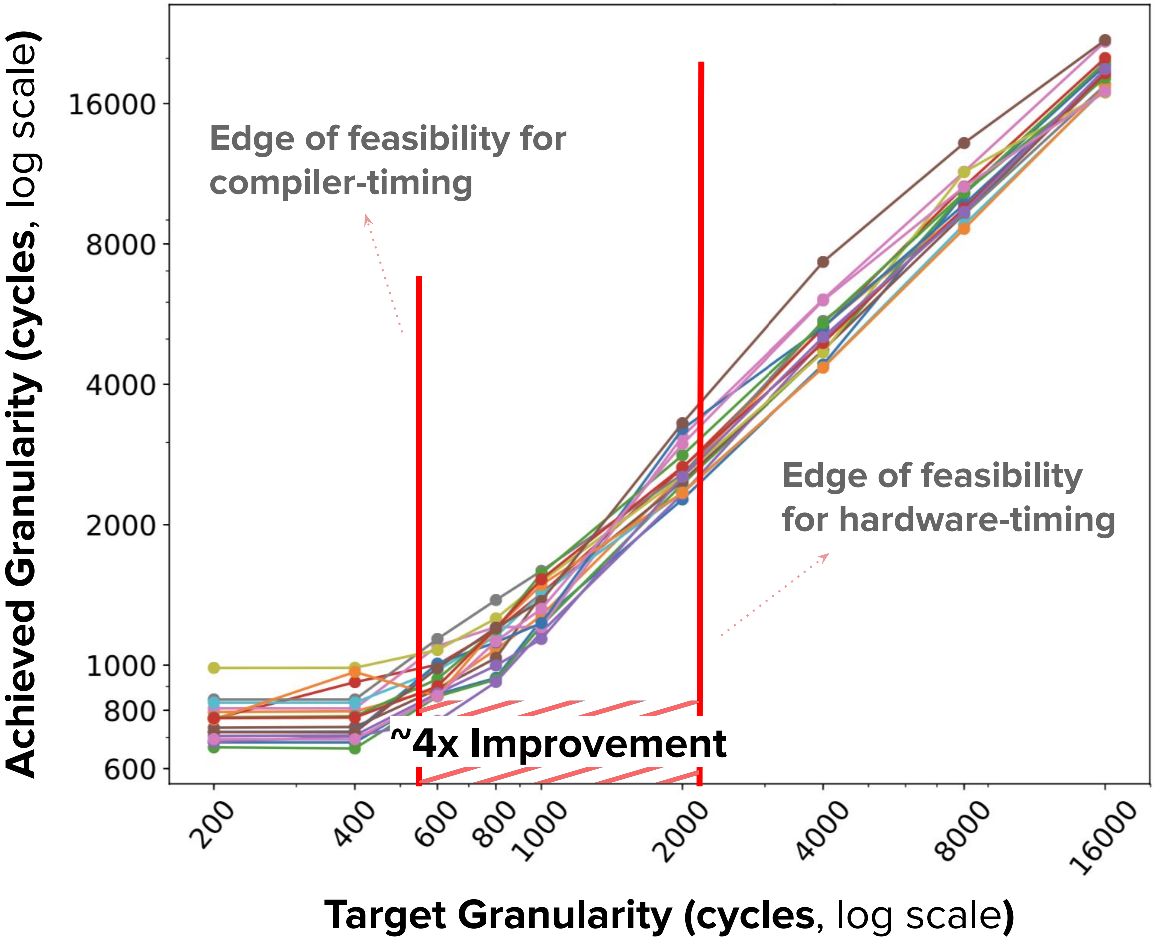
Accuracy of Compiler-Timing Driven Fibers per Context Switch



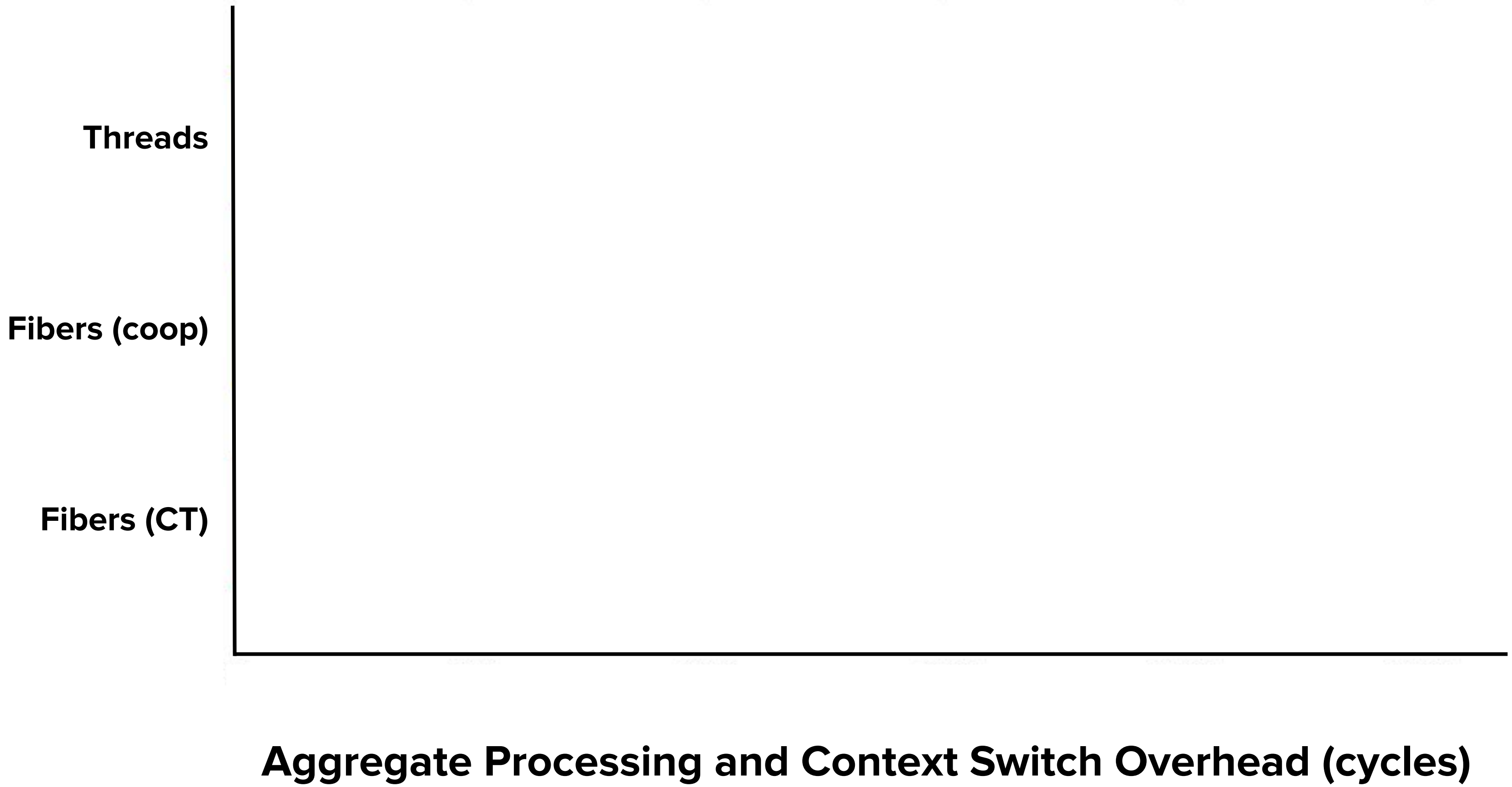
Accuracy of Compiler-Timing Driven Fibers per Context Switch

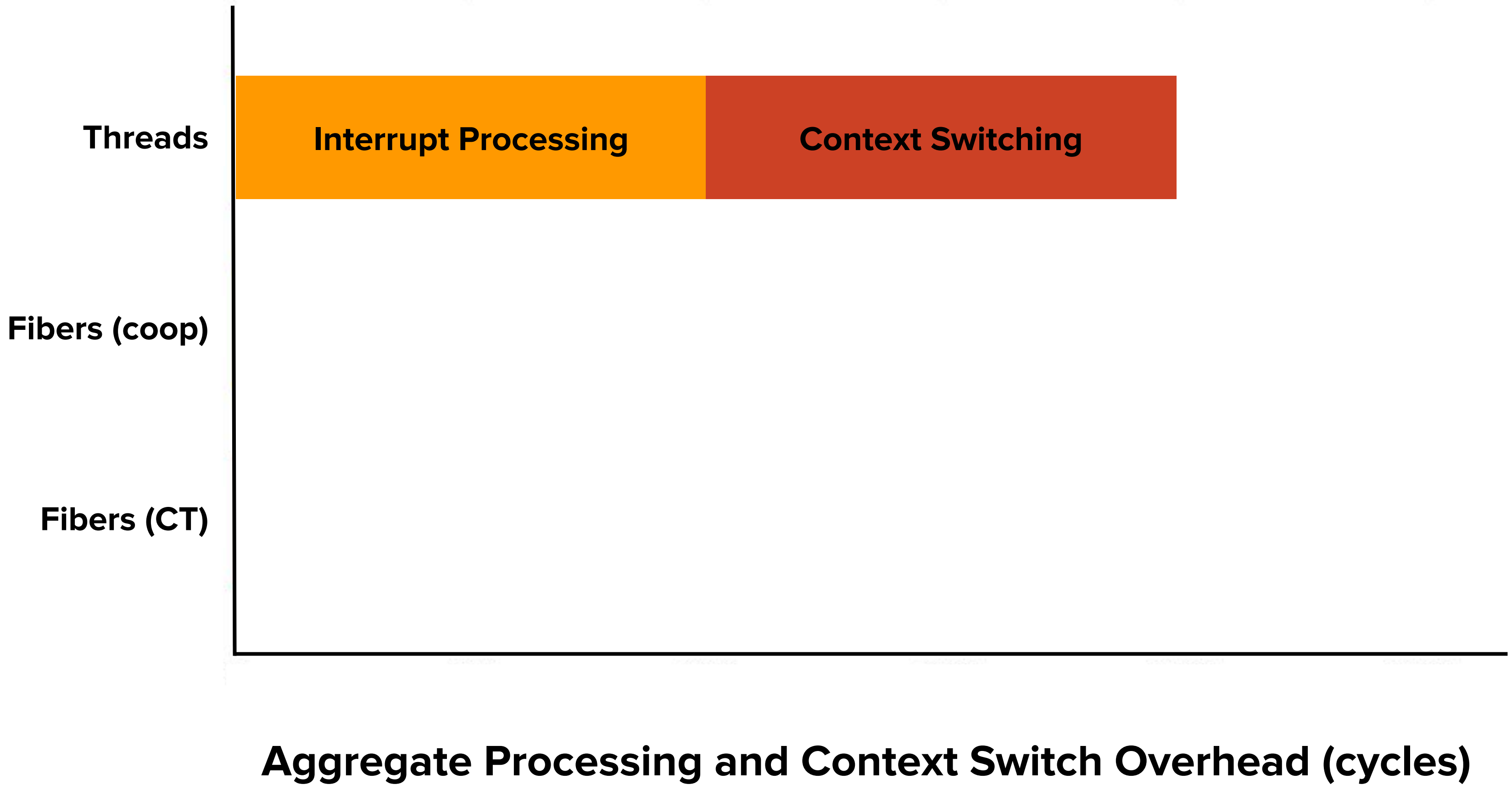


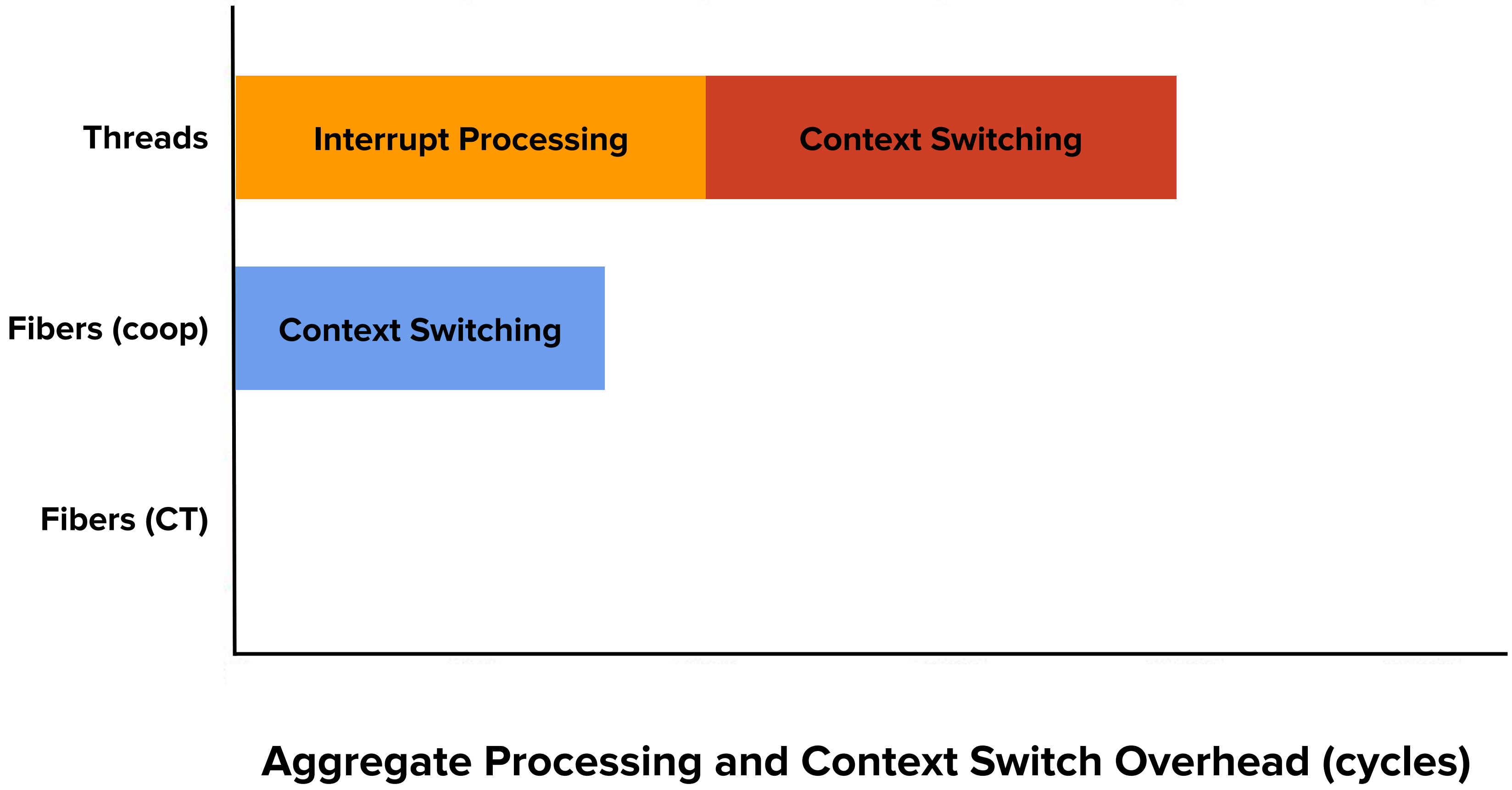
Accuracy of Compiler-Timing Driven Fibers per Context Switch

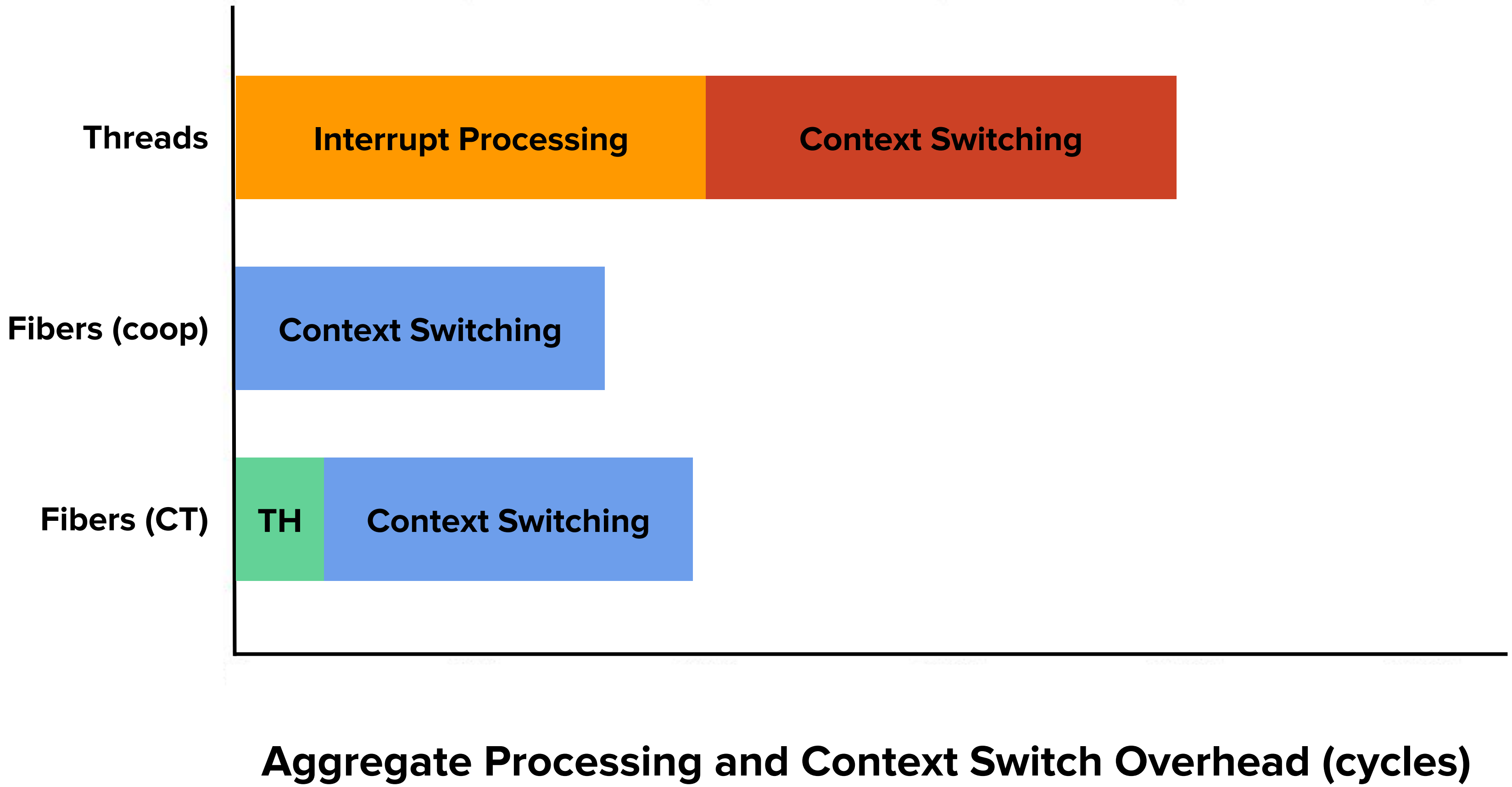


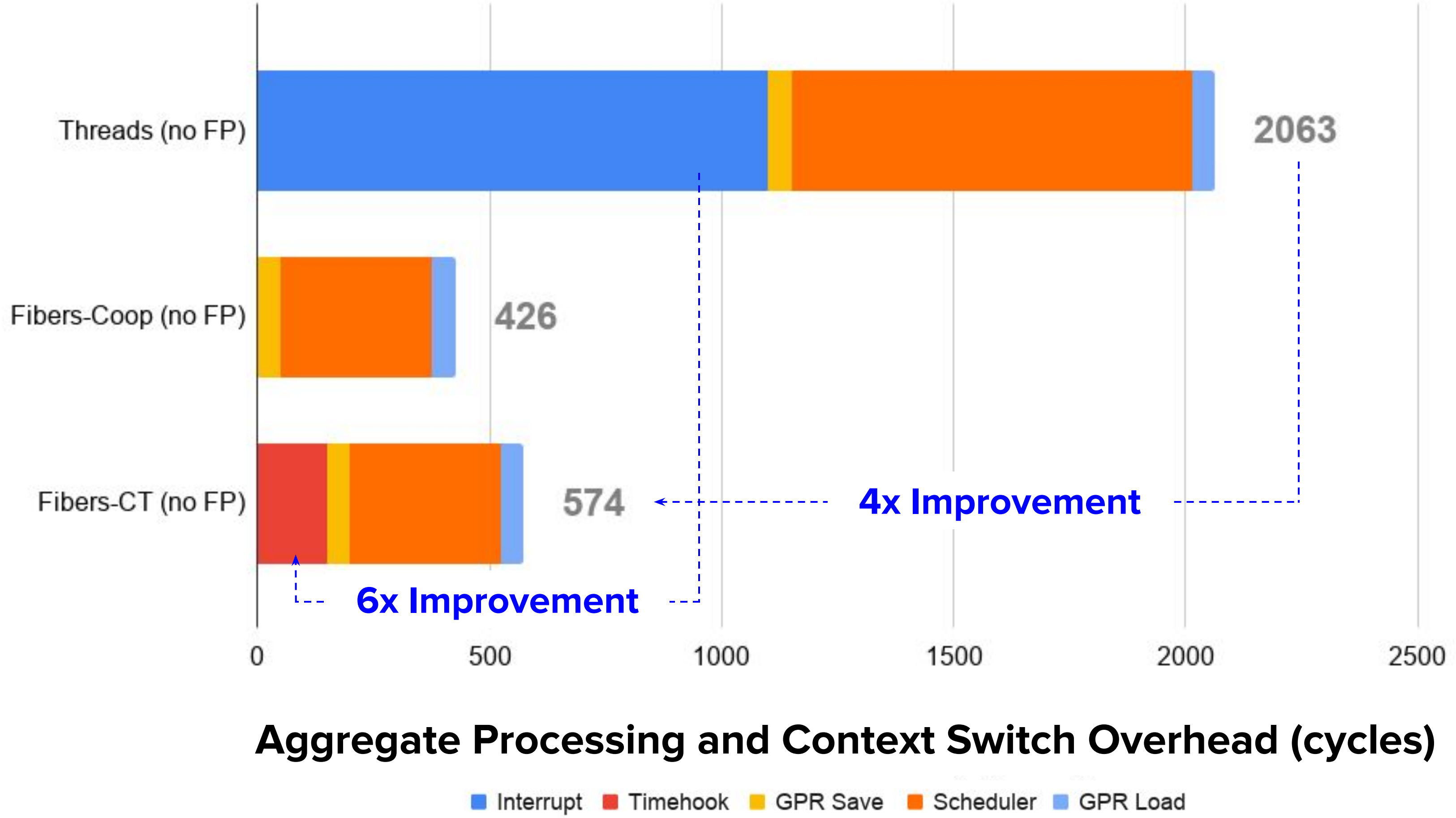
Aggregate Processing and Context Switch Overhead (cycles)


Threads
Fibers (coop)
Fibers (CT)
Aggregate Processing and Context Switch Overhead (cycles)


Threads
Fibers (coop)
Fibers (CT)
Interrupt Processing
Context Switching
Aggregate Processing and Context Switch Overhead (cycles)


Threads
Interrupt Processing
Context Switching
Fibers (coop)
Context Switching
Fibers (CT)
Aggregate Processing and Context Switch Overhead (cycles)


Threads
Interrupt Processing
Context Switching
Fibers (coop)
Context Switching
Fibers (CT)
TH
Context Switching
Aggregate Processing and Context Switch Overhead (cycles)


Threads (no FP)
Fibers-Coop (no FP)
Fibers-CT (no FP)
2063
426
574
4x Improvement
6x Improvement
0
500
1000
1500
2000
2500
Aggregate Processing and Context Switch Overhead (cycles)
Interrupt
Timehook
GPR Save
Scheduler
GPR Load

Compiler-Timing in a Nutshell

- **Timing** is **essential** --- notably in **preemptive threads**
- Threads are a useful abstraction for parallel programs, but they utilize **hardware timing** which incurs **high overheads**
- CT introduces a **fully software** approach to timing --- which can be coupled with **lightweight** multitasking mechanisms
- We achieve timing with **6x lower overhead** than hardware timing, which allows for **4x smaller granularity** than preemptive threads

Thanks for listening! Questions?

Authors:

- Souradip Ghosh: sgh@u.northwestern.edu, souradipghosh.com
- Michael Cuevas: cuevas@u.northwestern.edu, mcuevas.org
- Simone Campanoni: simonec@eecs.northwestern.edu, cutt.ly/simonec
- Peter Dinda: pdinda@northwestern.edu, pdinda.org

Interweaving Project: interweaving.org

Prescience Lab: presciencelab.org