

LLVM'12 - European Conference, London

Reducing Dynamic Compilation Latency

Igor Böhm

PASTA
Processor Automated Synthesis
by iTerative Analysis
The University of Edinburgh

SYNOPSYS®
Predictable Success

LLVM'12 - European Conference, London

Concurrent and Parallel Dynamic Compilation

Igor Böhm

PASTA
Processor Automated Synthesis
by iTerative Analysis
The University of Edinburgh

SYNOPSYS®
Predictable Success

Dynamic Compilation

What do we want to improve?

Interp

Interpretation

Native

Native Code Execution

Interp

Native

Interp

Native

Time

Dynamic Compilation

What do we want to improve?

Interp

Interpretation

Native

Native Code Execution

Interp

Native

Interp

Native

Time

- initially code is interpreted

Dynamic Compilation

What do we want to improve?

Interp

Interpretation

Native

Native Code Execution

Interp

Native

Interp

Native

Time

- initially code is interpreted
- frequently executed code is compiled on-the-fly

Dynamic Compilation

What do we want to improve?

Interp

Interpretation

Native

Native Code Execution

Interp

Native

Interp

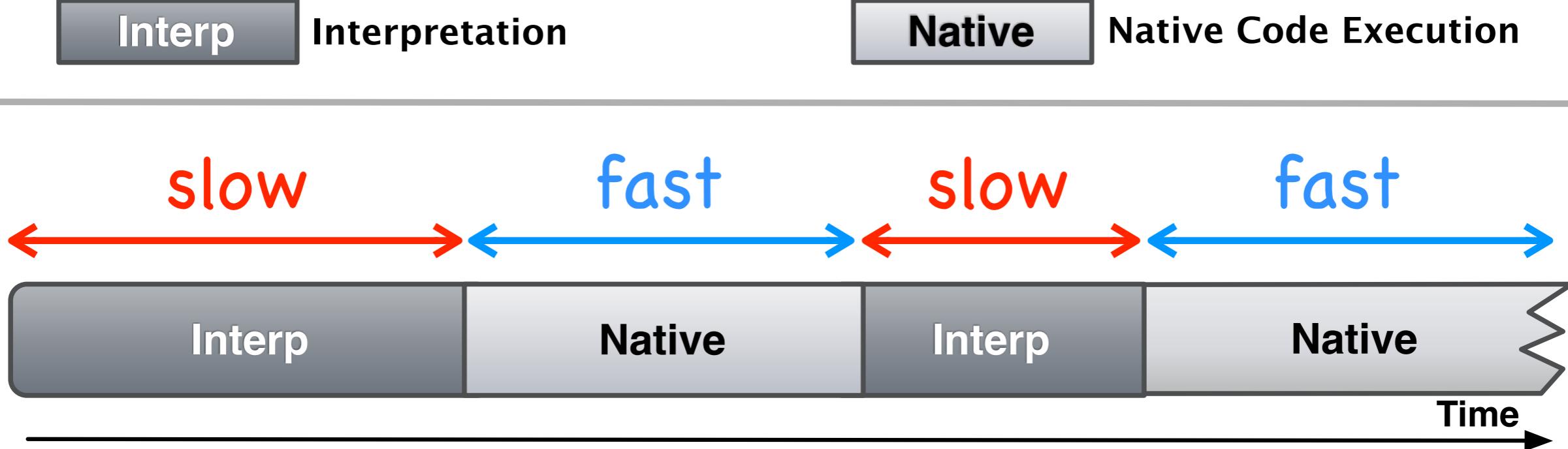
Native

Time

- initially code is interpreted
- frequently executed code is compiled on-the-fly
- switch from interpretive to native code execution as soon as dynamically compiled code is available

Dynamic Compilation

What do we want to improve?



- initially code is interpreted
- frequently executed code is compiled on-the-fly
- switch from interpretive to native code execution as soon as dynamically compiled code is available

Dynamic Compilation

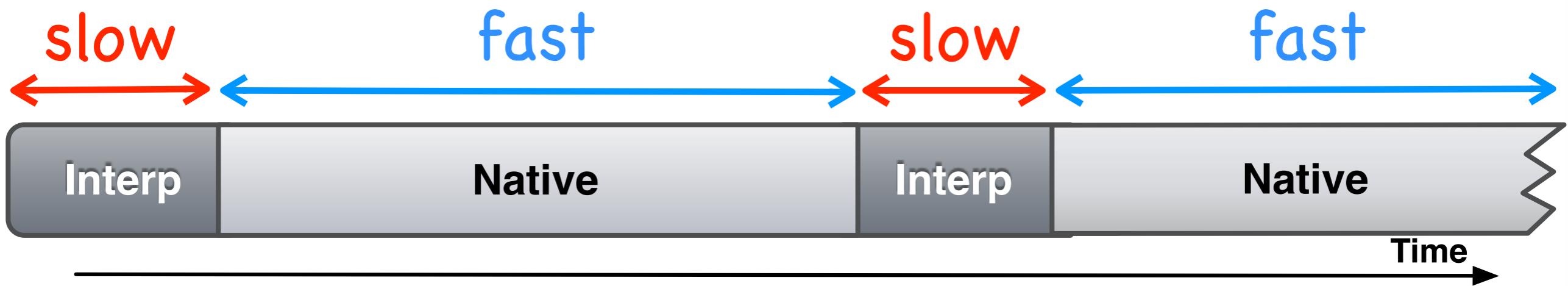
What do we want to improve?

Interp

Interpretation

Native

Native Code Execution



Earlier transition from interpretive to native execution

Interp

Interpretation

Compile

Dynamic Compilation

Profile

Interpretation with Profiling

Native

Native Code Execution

1

Dynamic Compilation using one Concurrent JIT Compiler**Main Thread**

Interp

Profile

Interp

Native

Interp

Profile

Interp

Profile

Native

Time

Compiler Thread

Thread 1

Compile

Compile

Compile

Time

Interp

Interpretation

Compile

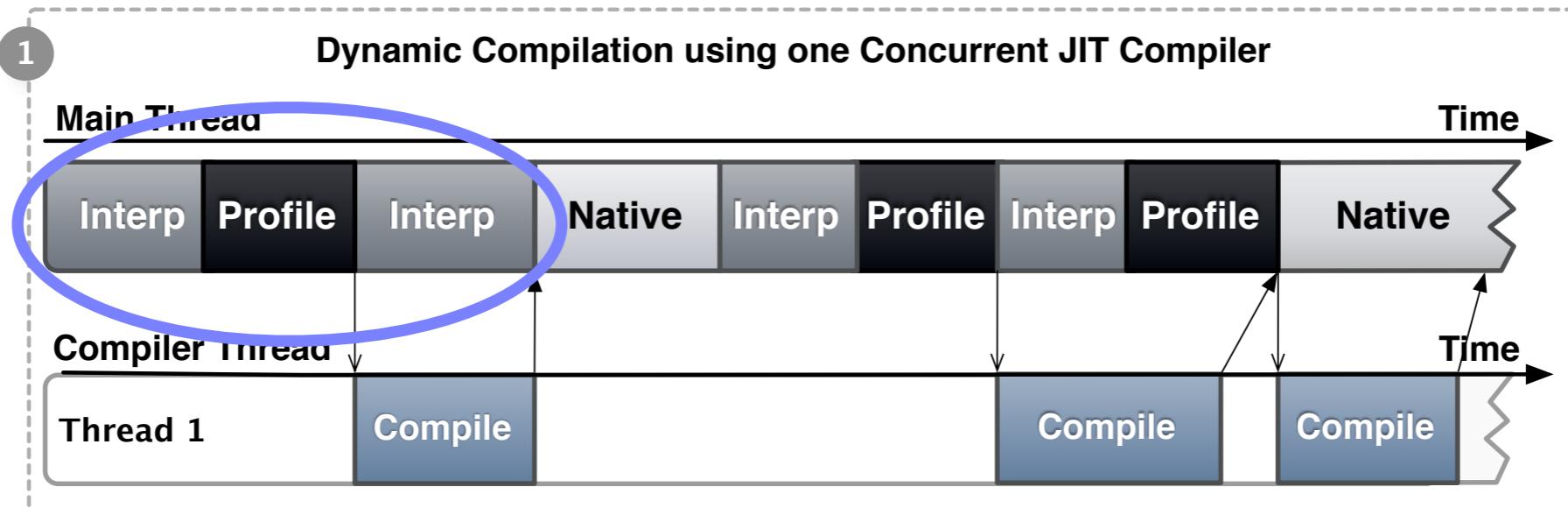
Dynamic Compilation

Profile

Interpretation with Profiling

Native

Native Code Execution



Interp

Interpretation

Compile

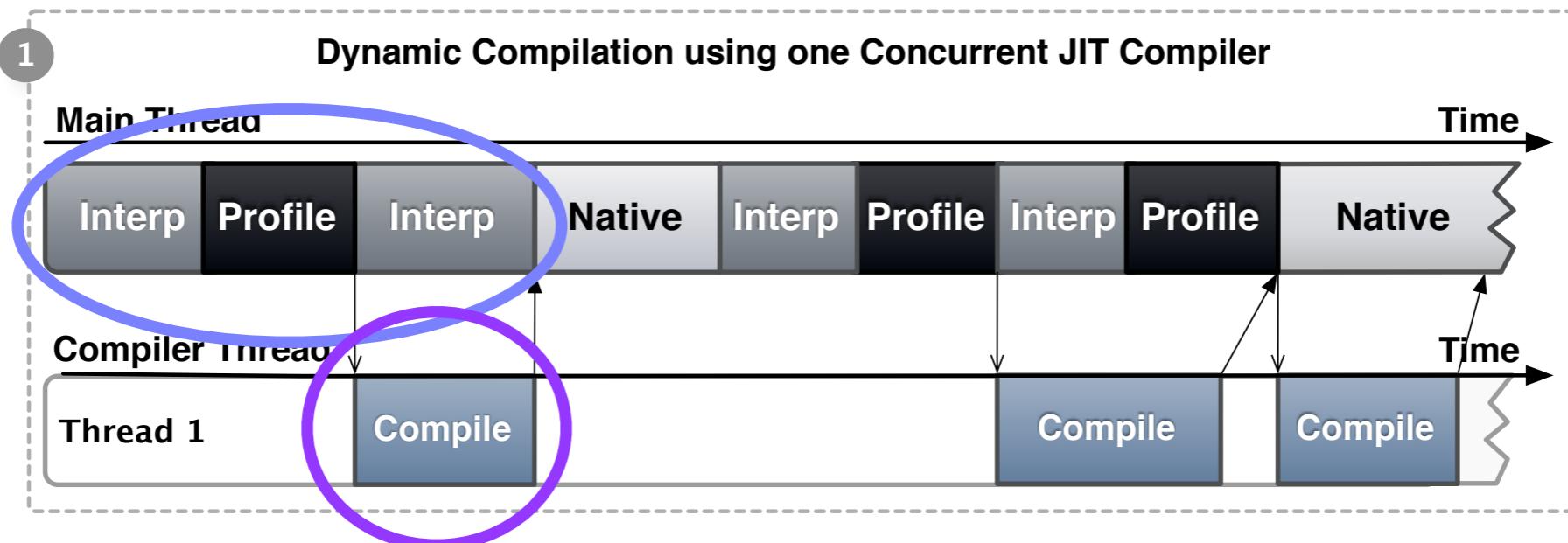
Dynamic Compilation

Profile

Interpretation with Profiling

Native

Native Code Execution



Interp Interpretation

Compile Dynamic Compilation

Profile Interpretation with Profiling

Native Native Code Execution

1

Dynamic Compilation using one Concurrent JIT Compiler

Main Thread

Time



Compiler Thread

Time

critical path

Thread 1 Compile

Compile

Compile

Interp Interpretation

Compile Dynamic Compilation

Profile Interpretation with Profiling

Native Native Code Execution

1

Dynamic Compilation using one Concurrent JIT Compiler

Main Thread

Time



Compiler Thread

critical path

Time



2

Dynamic Compilation using Concurrent and Parallel JIT Compiler Task Farm

Main Thread

Time



Compiler Thread

Thread 1 Compile Compile

Time

Compiler Thread

Compile Compile

Time

Compiler Thread

Compile

Time

Compiler Thread

Compile

Time

Compiler Thread

Compile Compile

Time

Interp Interpretation

Compile Dynamic Compilation

Profile Interpretation with Profiling

Native Native Code Execution

1

Dynamic Compilation using one Concurrent JIT Compiler

Main Thread

Time

Interp Profile Interp Native Interp Profile Interp Profile Native

Compiler Thread

critical path

Compile

Compile

2

Dynamic Compilation using Concurrent and Parallel JIT Compiler Task Farm

Main Thread

Time

Profile

Native

Profile

Native

Profile

Native

Compiler Thread

Thread 1 Compile Compile

Compile Compile

Compiler Thread

Thread 2 Compile

Compile

Compiler Thread

Thread 3 Compile

Compile Compile

Interp Interpretation

Compile Dynamic Compilation

Profile Interpretation with Profiling

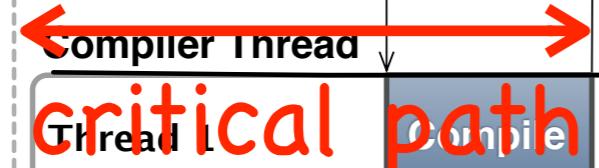
Native Native Code Execution

1

Dynamic Compilation using one Concurrent JIT Compiler

Main Thread

Time

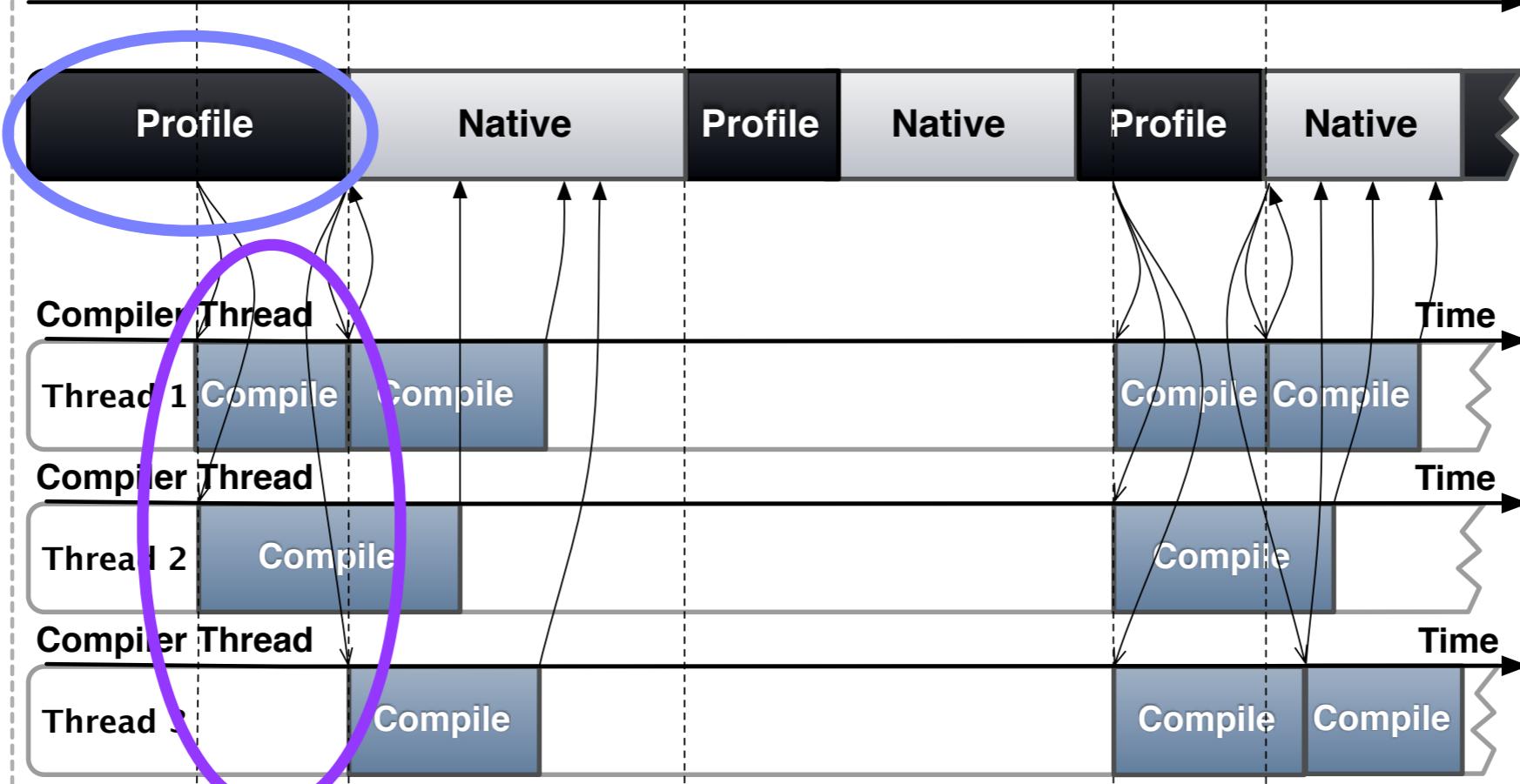


2

Dynamic Compilation using Concurrent and Parallel JIT Compiler Task Farm

Main Thread

Time



Interp Interpretation

Compile Dynamic Compilation

Profile Interpretation with Profiling

Native Native Code Execution

1

Dynamic Compilation using one Concurrent JIT Compiler

Main Thread

Time



Compiler Thread

critical path

Time

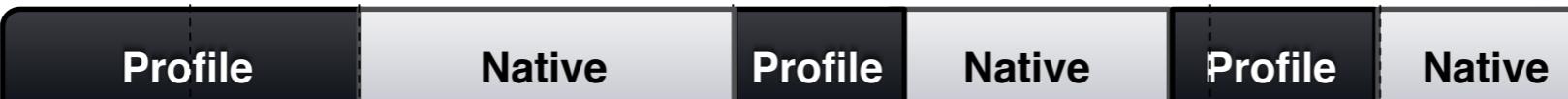


2

Dynamic Compilation using Concurrent and Parallel JIT Compiler Task Farm

Main Thread

Time



Compiler Thread

critical path

Time



Compiler Thread

Thread 2 Compile

Time

Compiler Thread

Compile

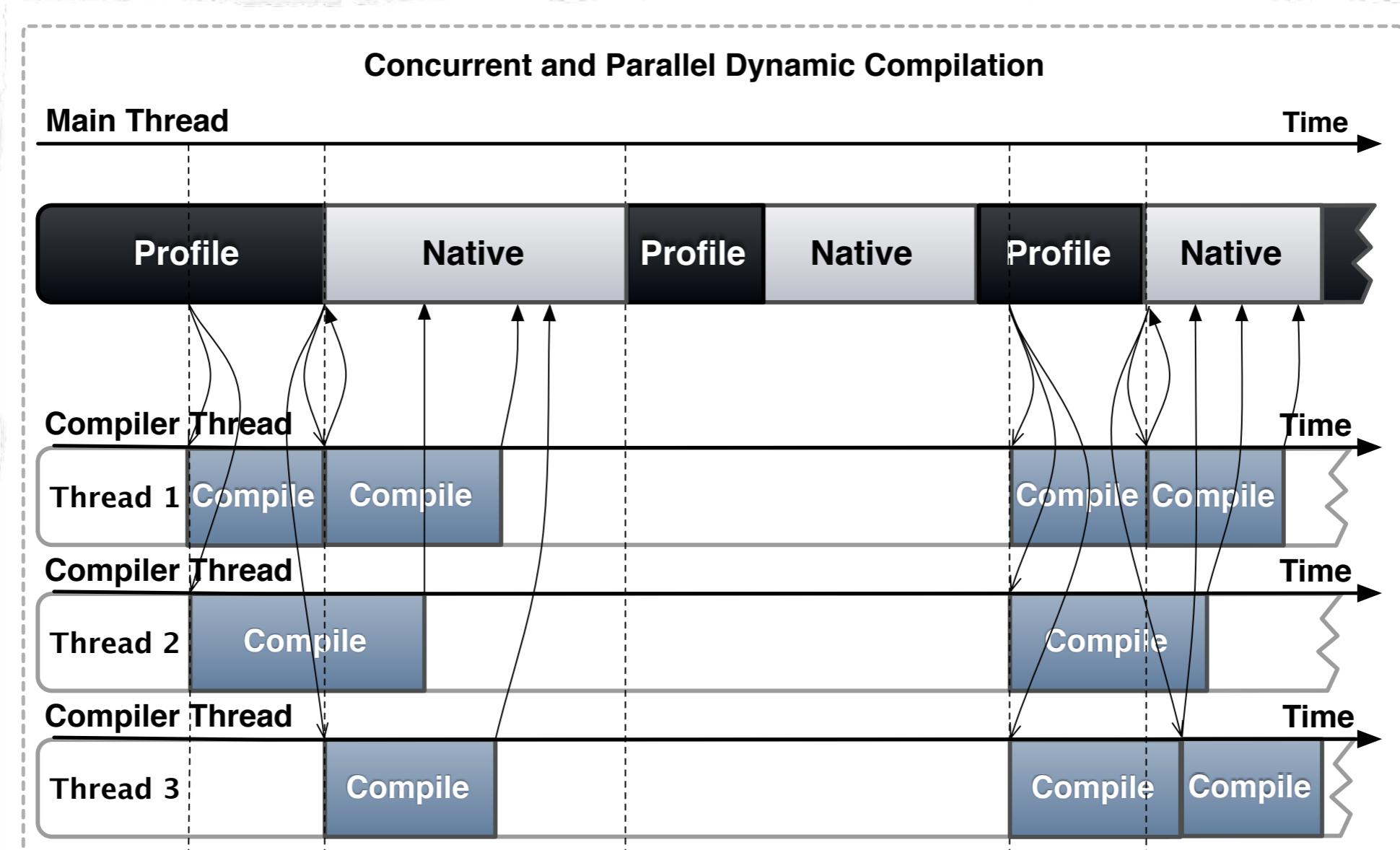
Time

Compiler Thread

Compile

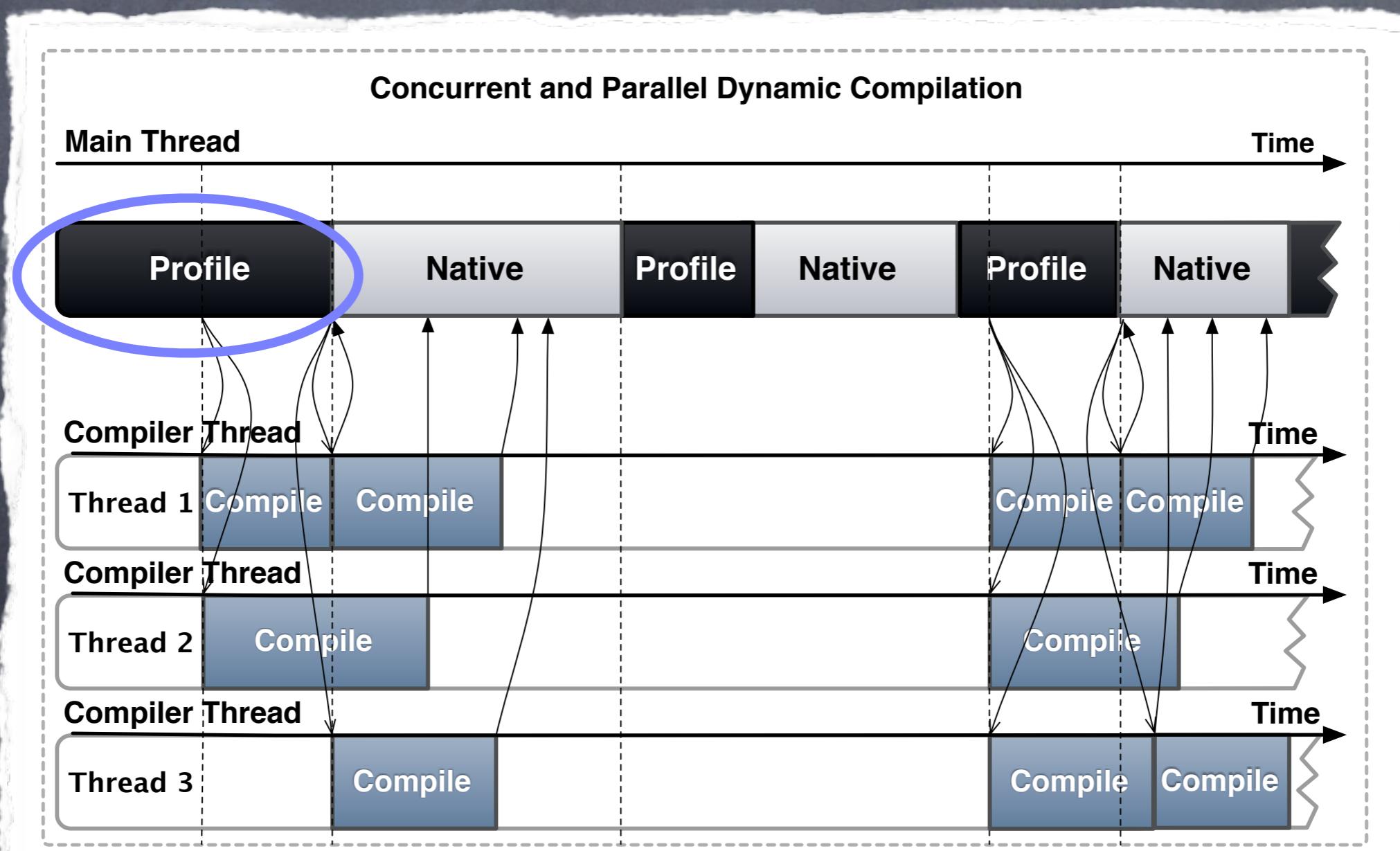
Time

Solution To Dynamic Compilation Latency Problem



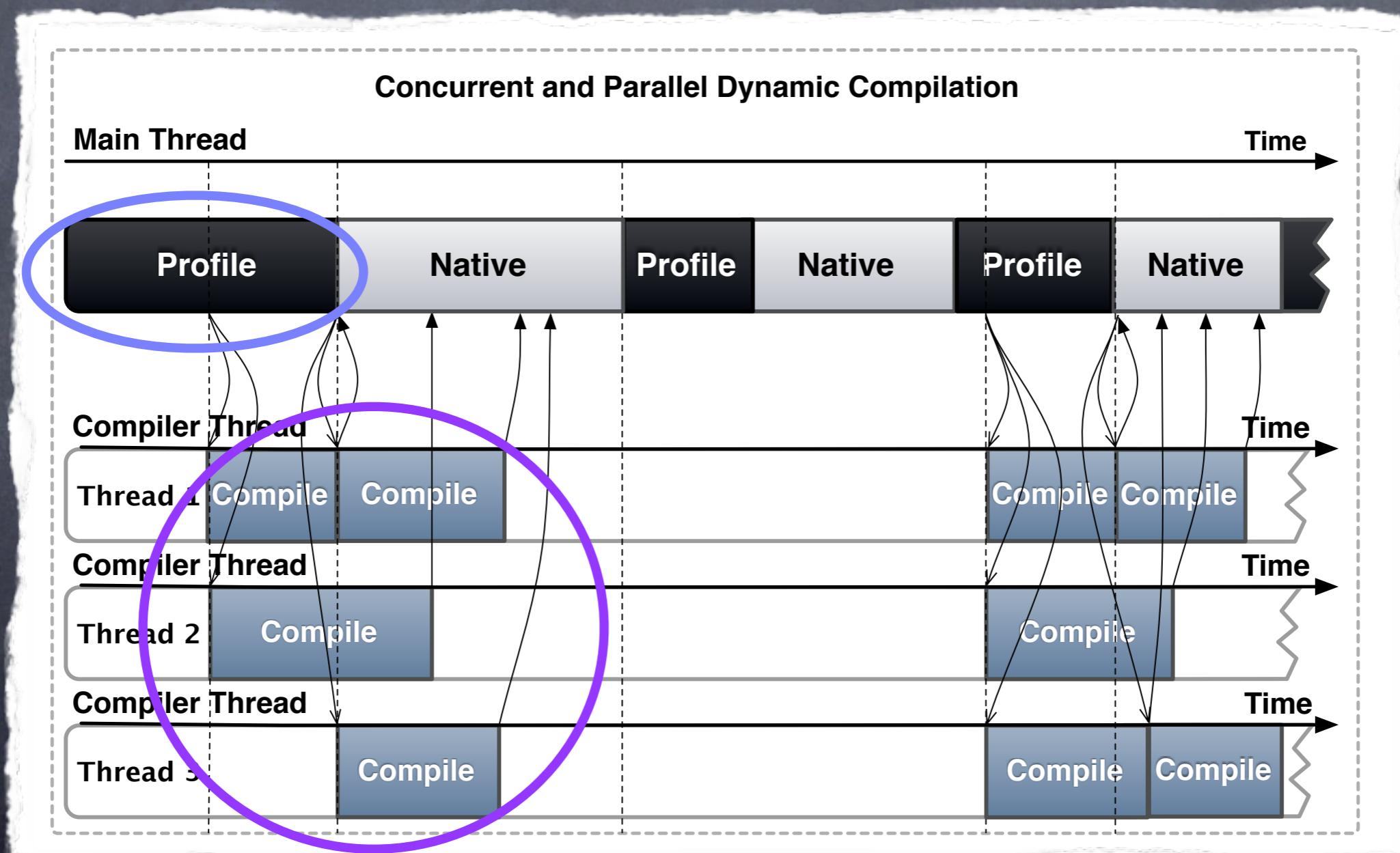
Solution To Dynamic Compilation Latency Problem

- improve code discovery/profiling



Solution To Dynamic Compilation Latency Problem

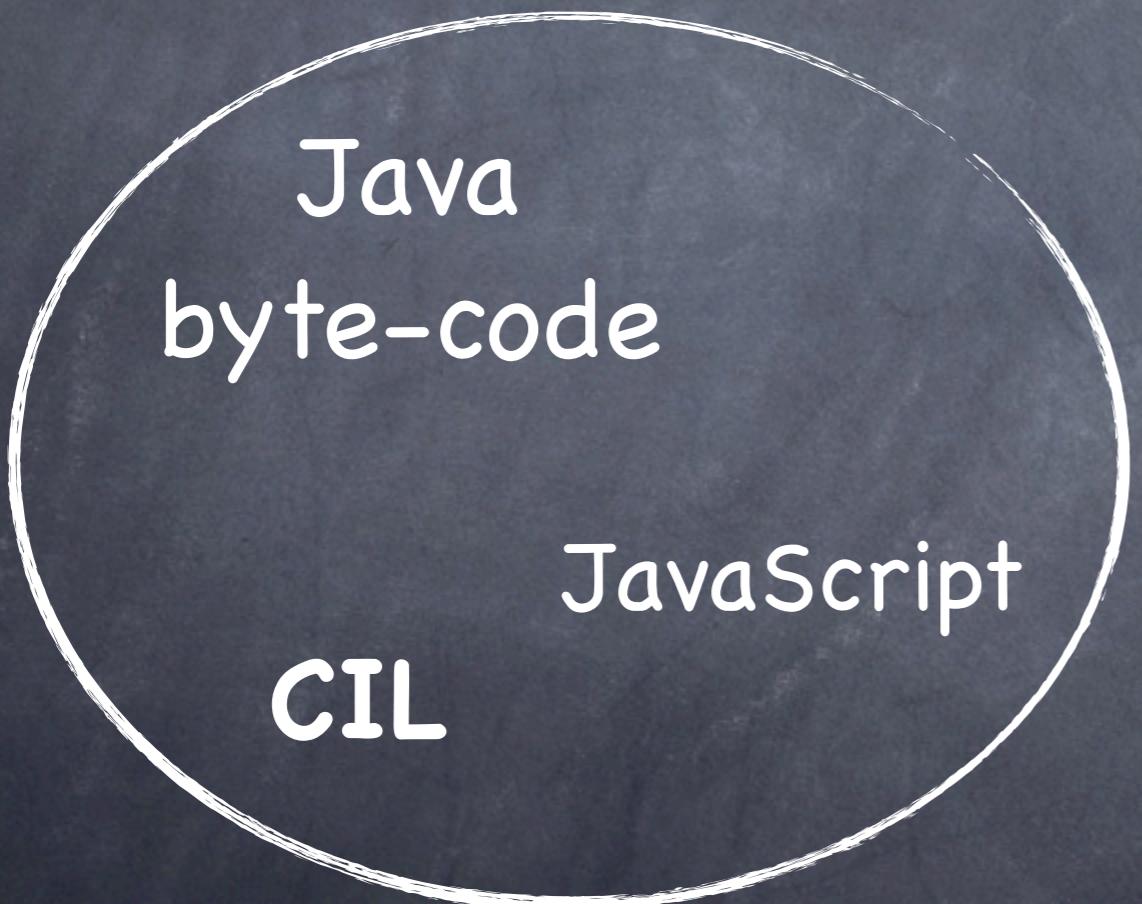
- improve code discovery/profiling
- improve dynamic compilation workload throughput



How hard can Code Discovery be?

How hard can Code Discovery be?

Static



How hard can Code Discovery be?

Static

Dynamic

Java
byte-code

JavaScript
CIL

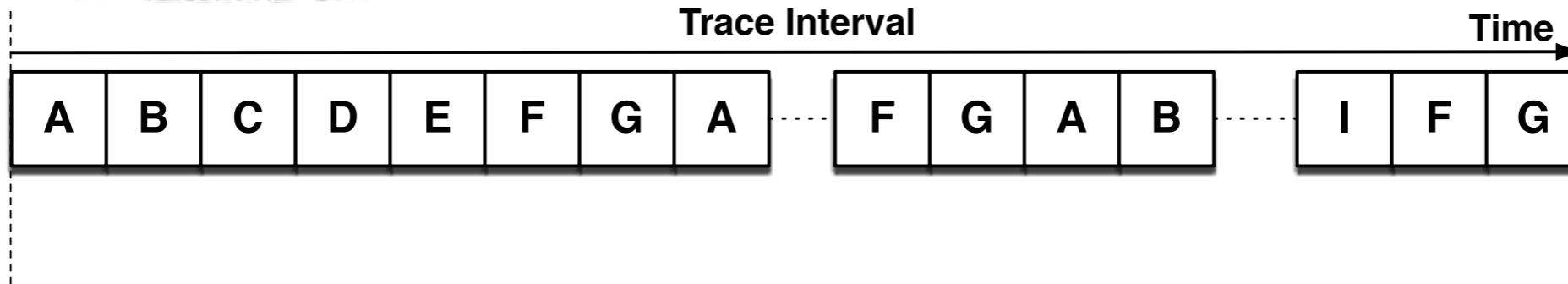
ARCompact
binary
x86
binary ARM
binary

How hard can Code Discovery be?

“A crucial problem in the decompilation or disassembly of computer programs is the identification of executable code, i.e. the separation of instructions from data. This problem, for most computer architectures, is equivalent to the Halting Problem and is therefore unsolvable in general.”

[Horspool and Marovac - 1980]

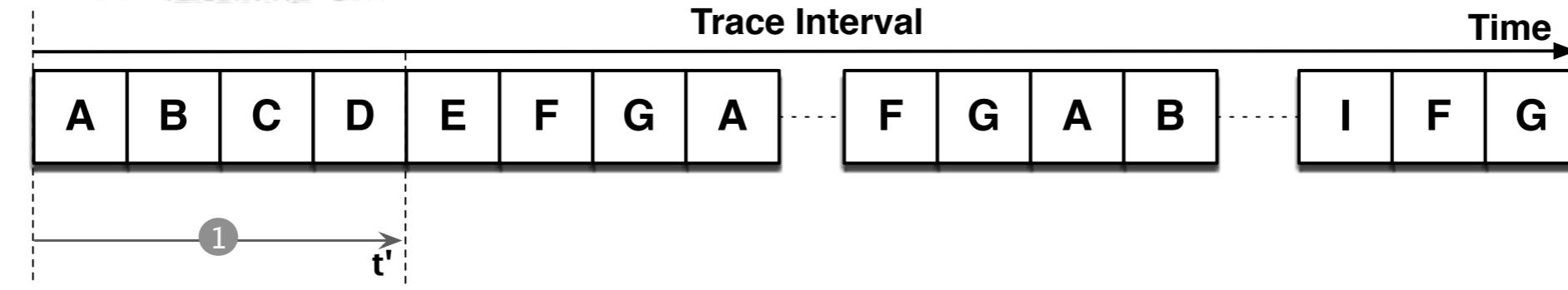
Incremental Code Discovery



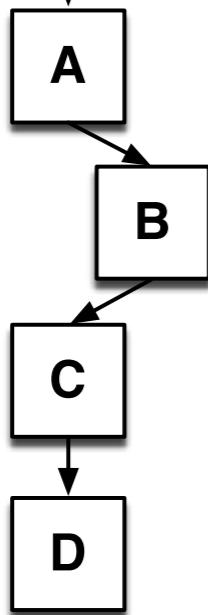
Sequence of interpreted basic blocks

- A** Basic Block
- ← CFG Edges
- ← Return to Interpreter

Incremental Code Discovery



1 Region after t'

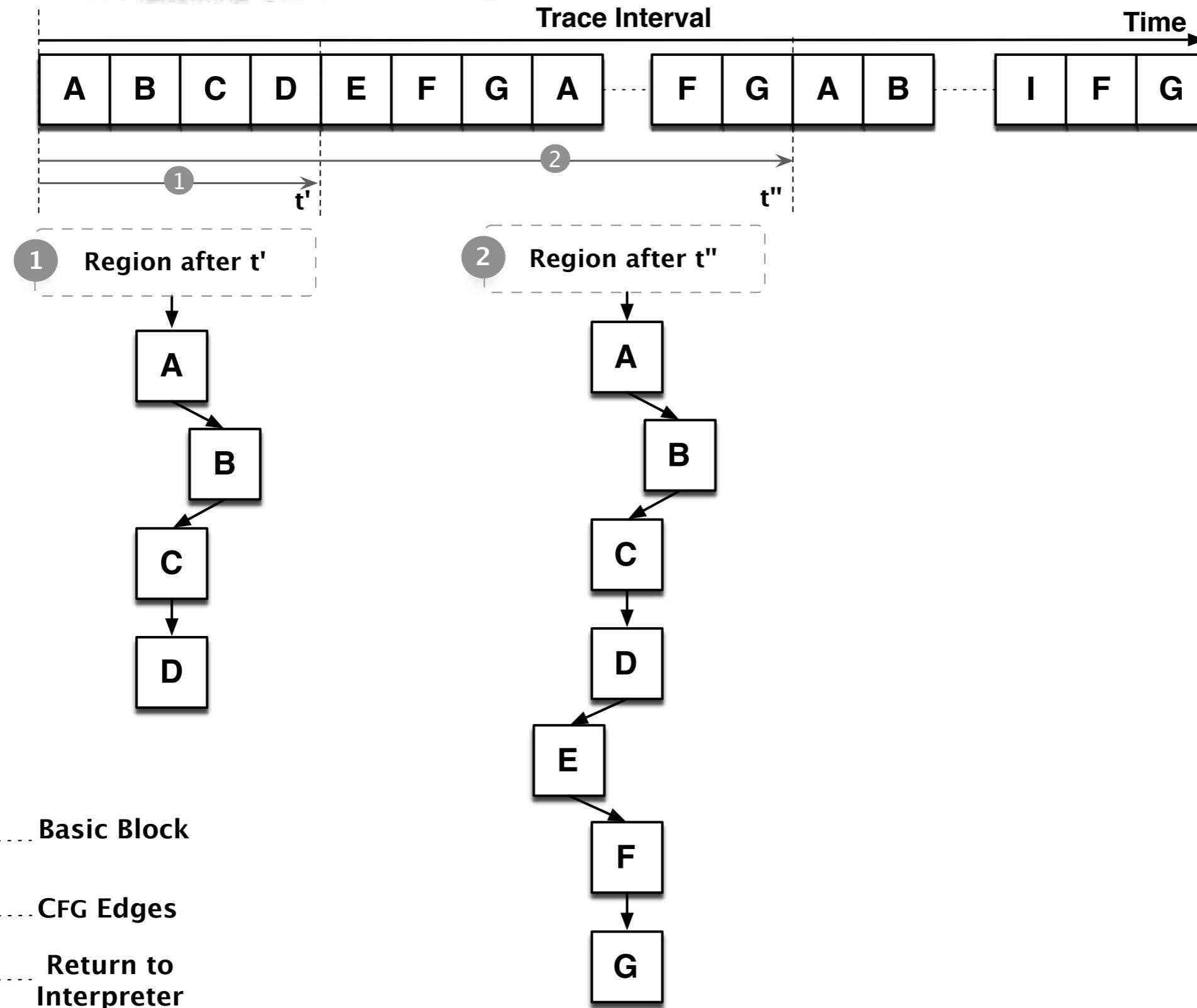


Basic Block

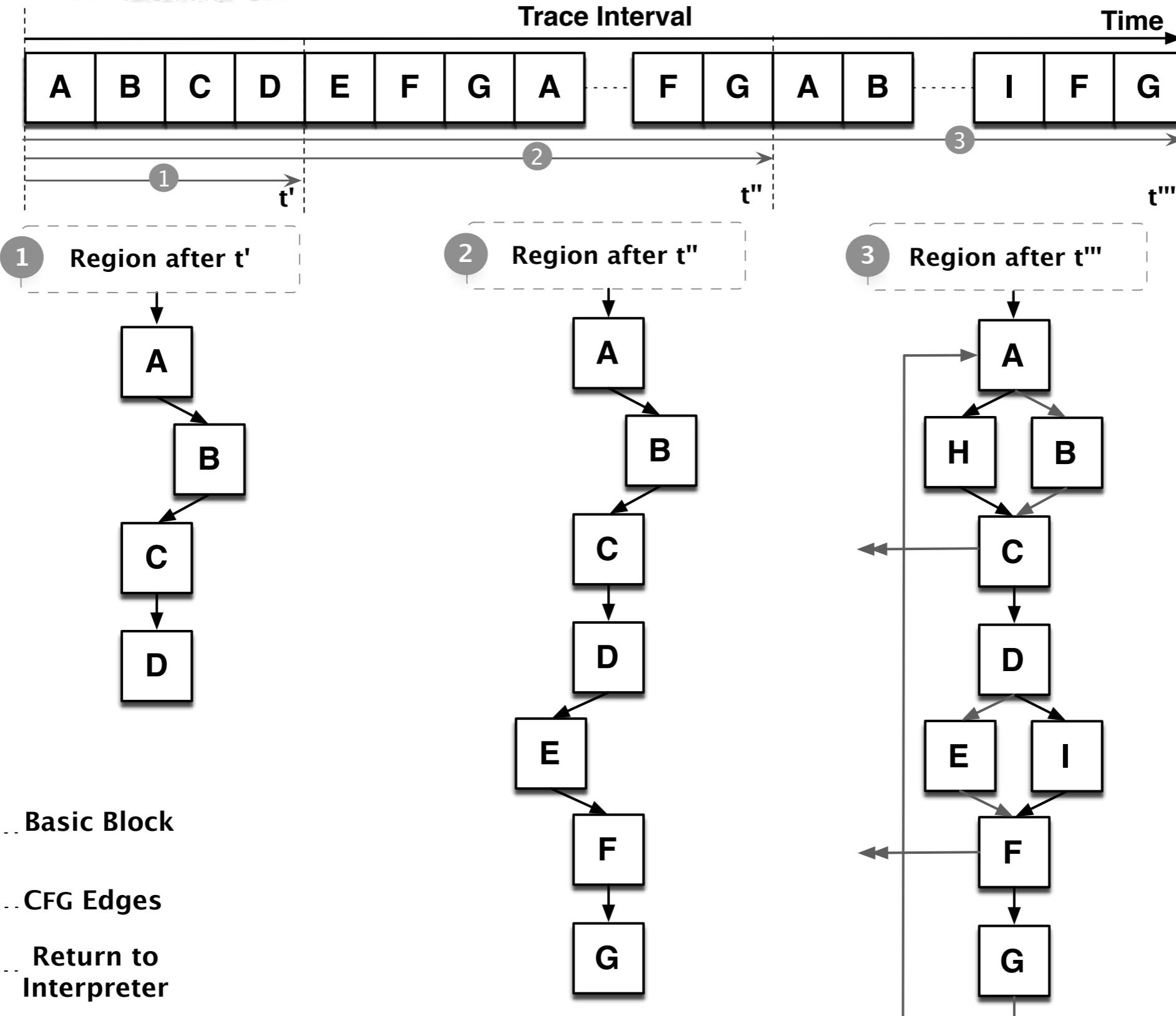
CFG Edges

Return to Interpreter

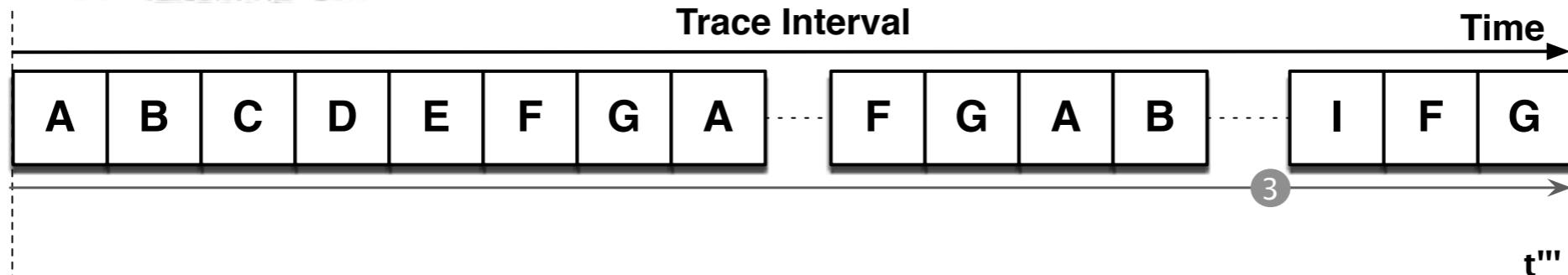
Incremental Code Discovery



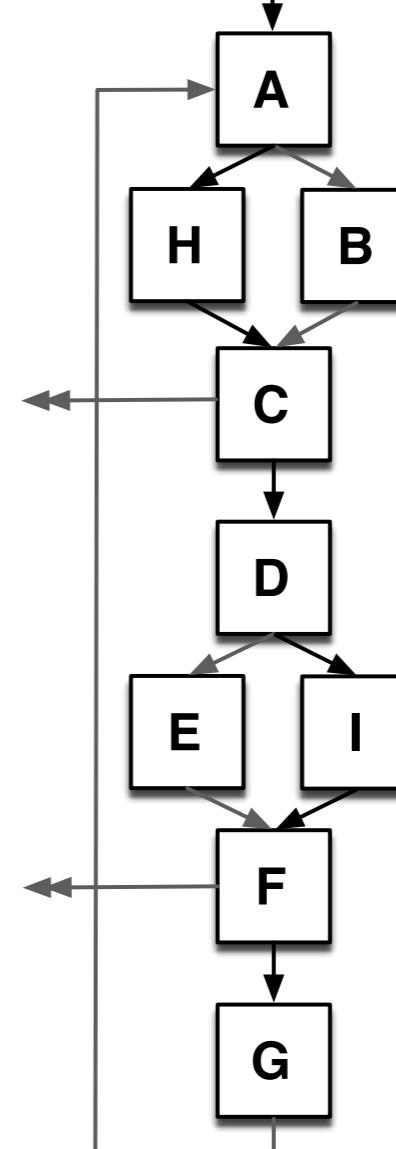
Incremental Code Discovery



Incremental Code Discovery



3 Region after t''

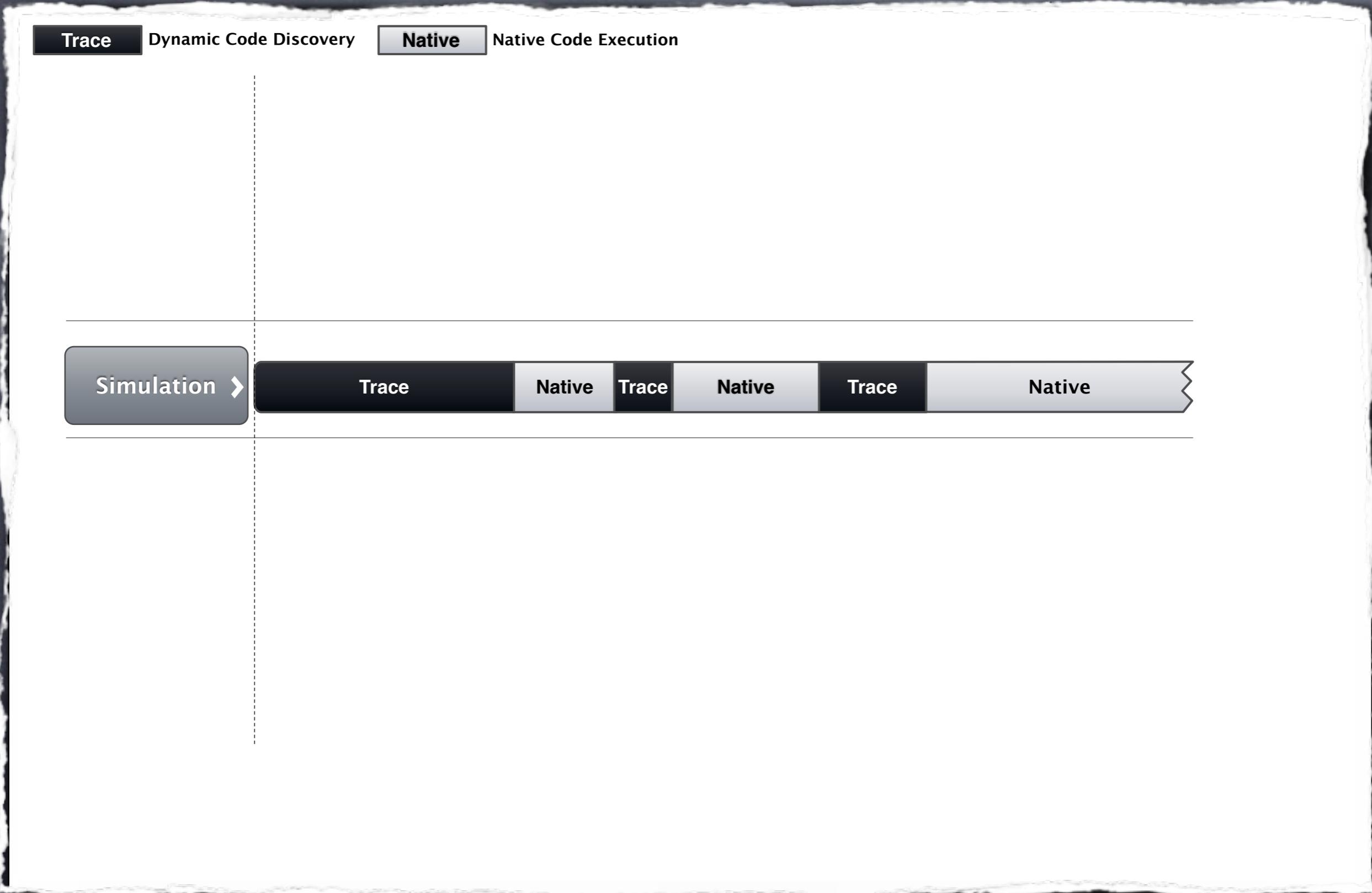


Region == Dynamic CFG

- Basic Block
- CFG Edges
- Return to Interpreter

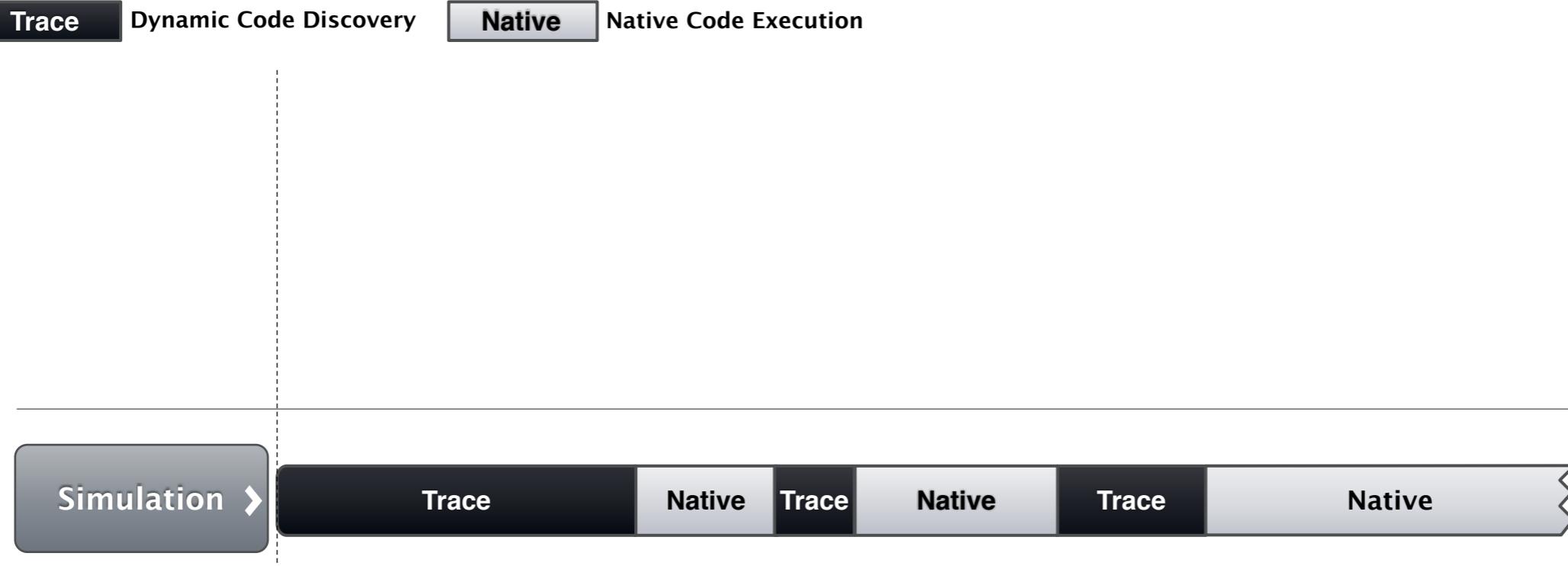
Concurrent and Parallel JIT Compilation in Action

(reducing the critical path)



Concurrent and Parallel JIT Compilation in Action

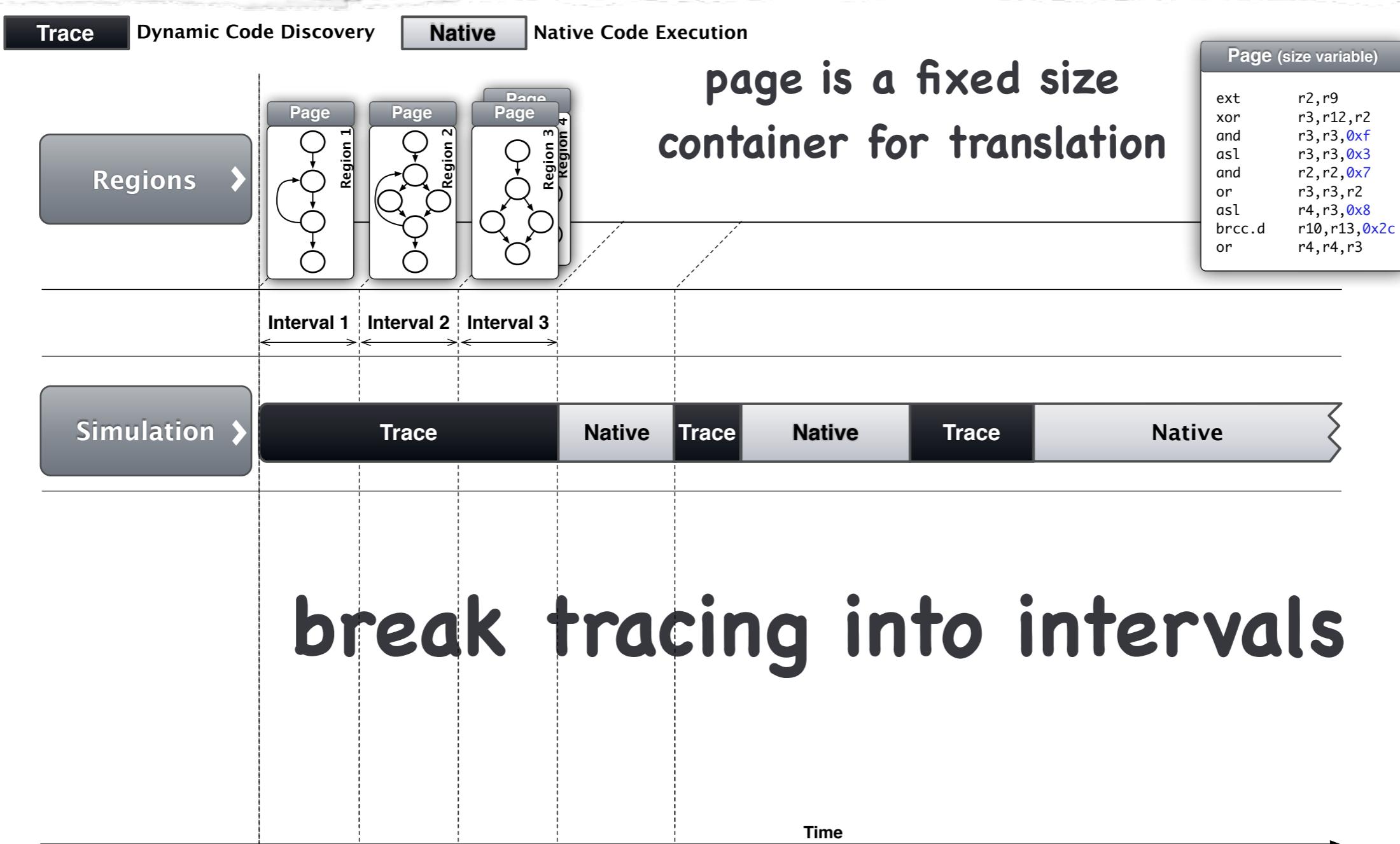
(reducing the critical path)



trace right from the start

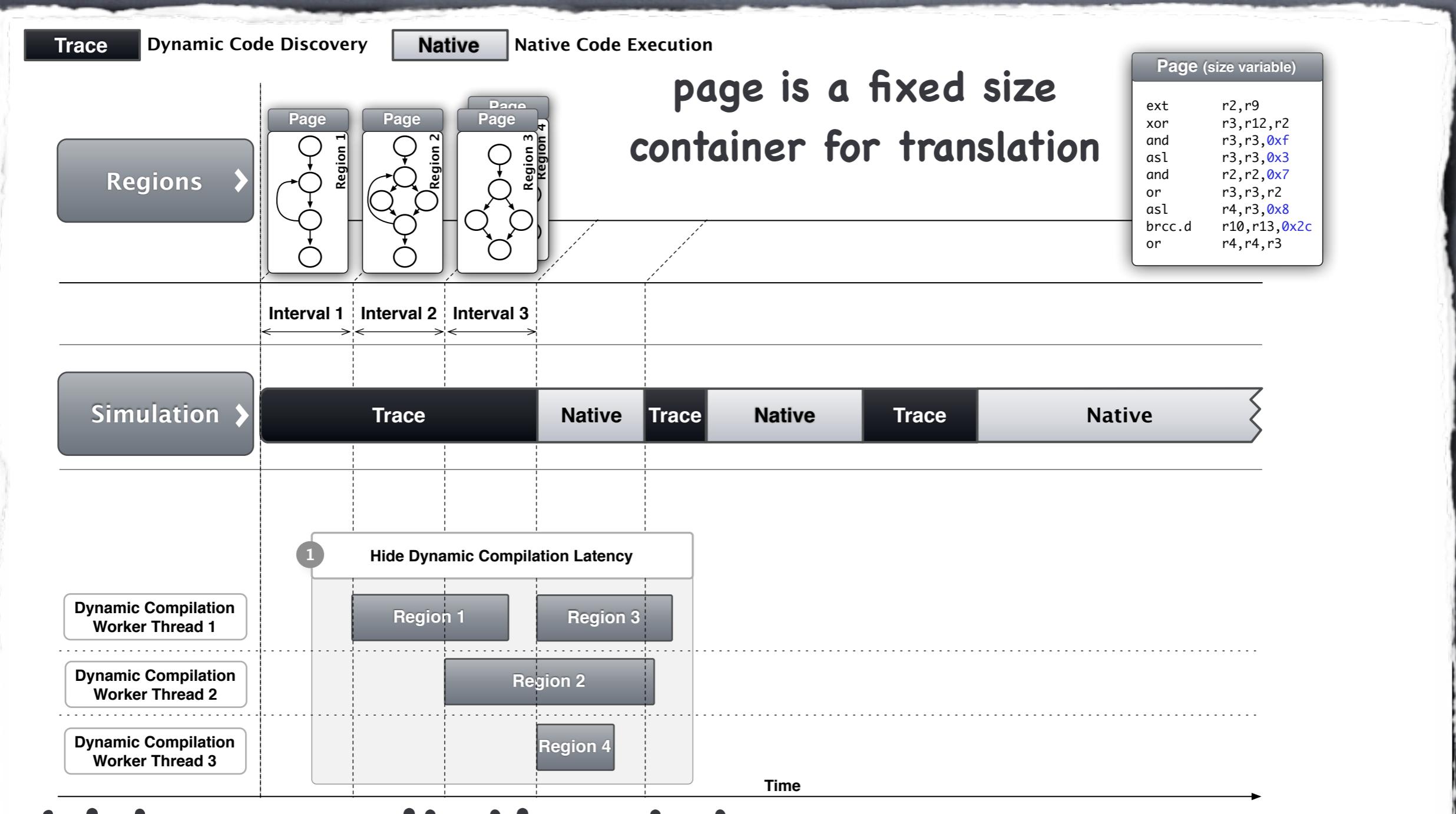
Concurrent and Parallel JIT Compilation in Action

(reducing the critical path)



Concurrent and Parallel JIT Compilation in Action

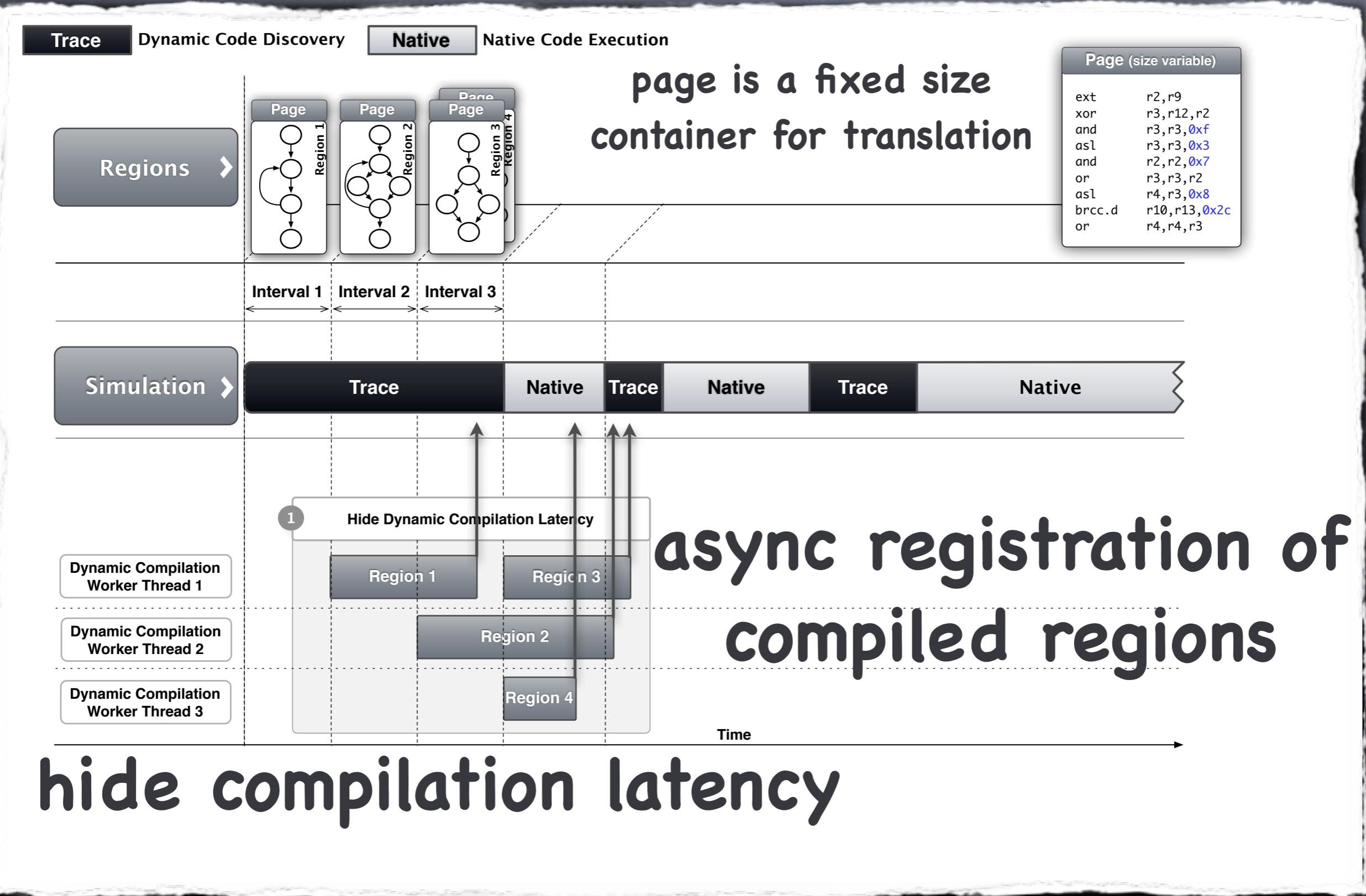
(reducing the critical path)



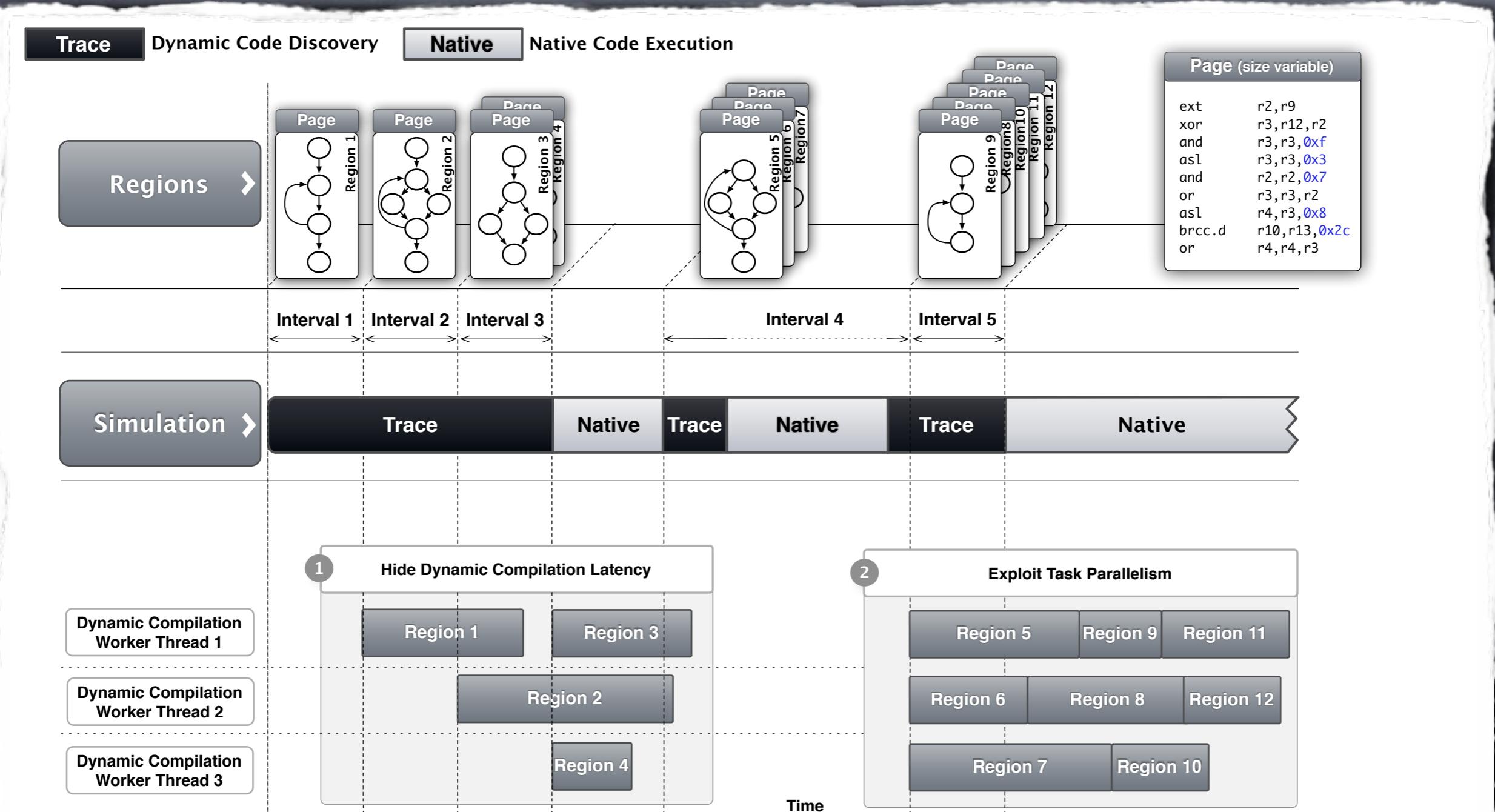
hide compilation latency

Concurrent and Parallel JIT Compilation in Action

(reducing the critical path)



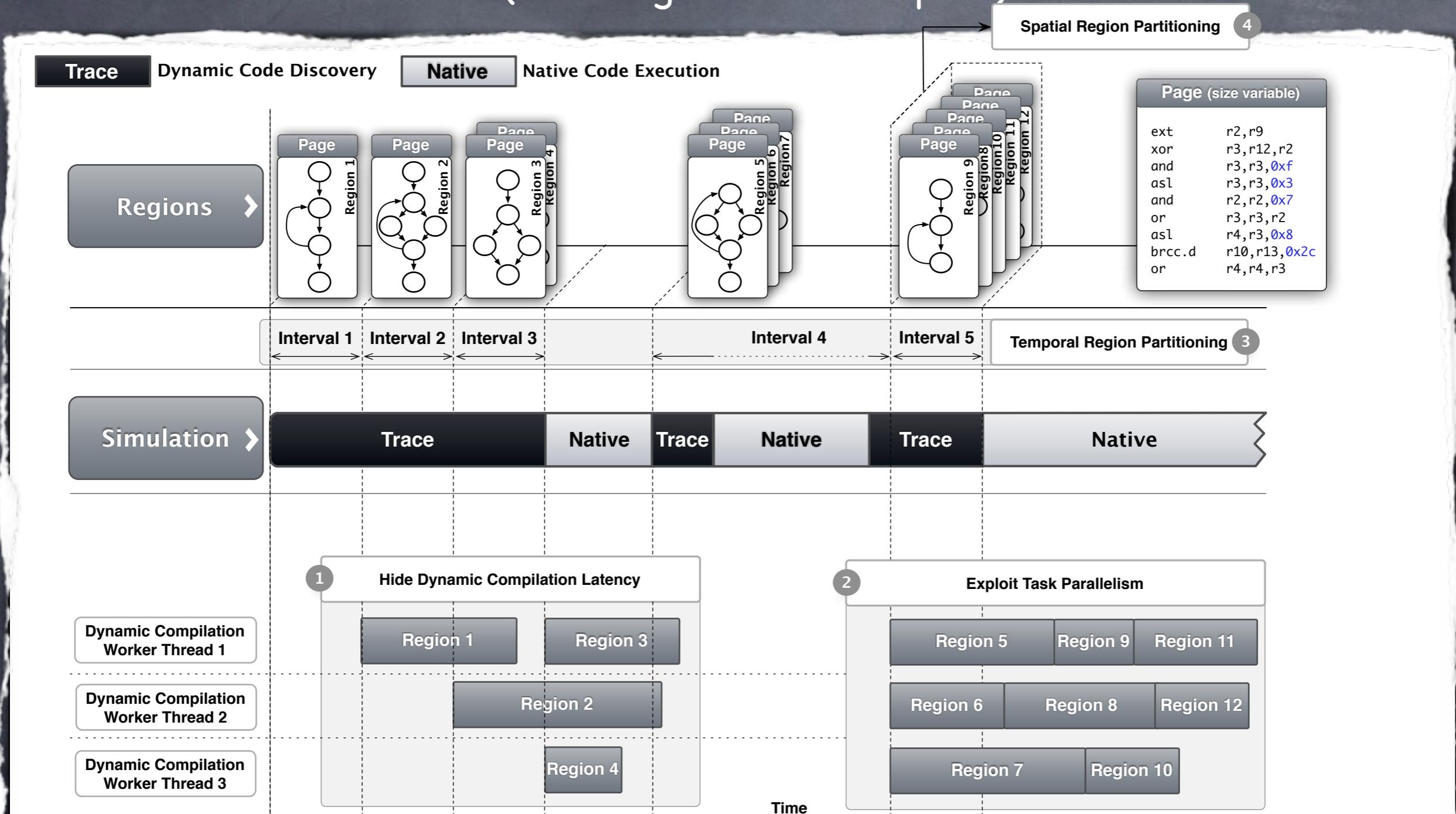
Concurrent and Parallel JIT Compilation in Action (reducing the critical path)



hide compilation latency
exploit task parallelism

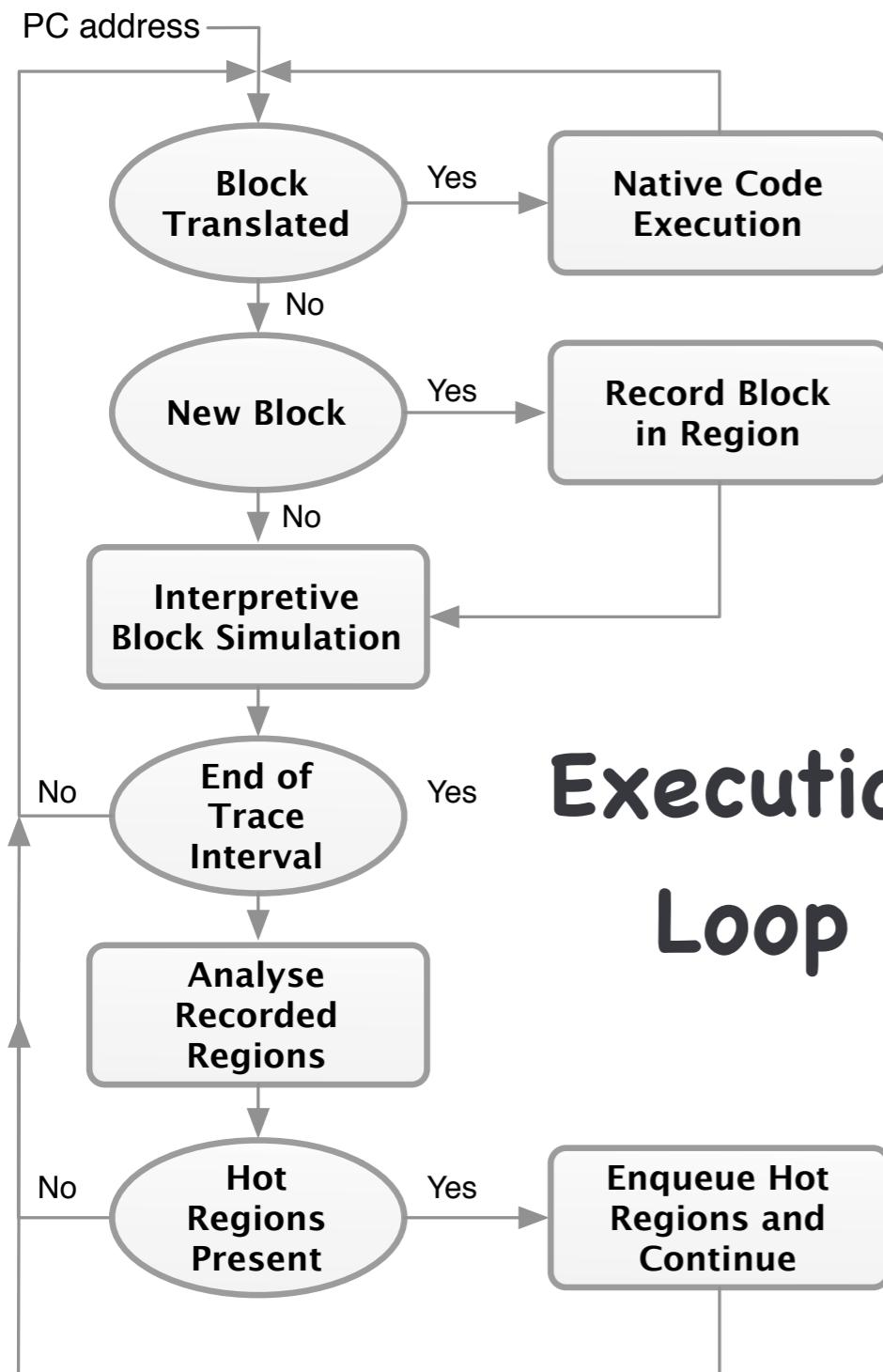
Concurrent and Parallel JIT Compilation in Action

(reducing the critical path)



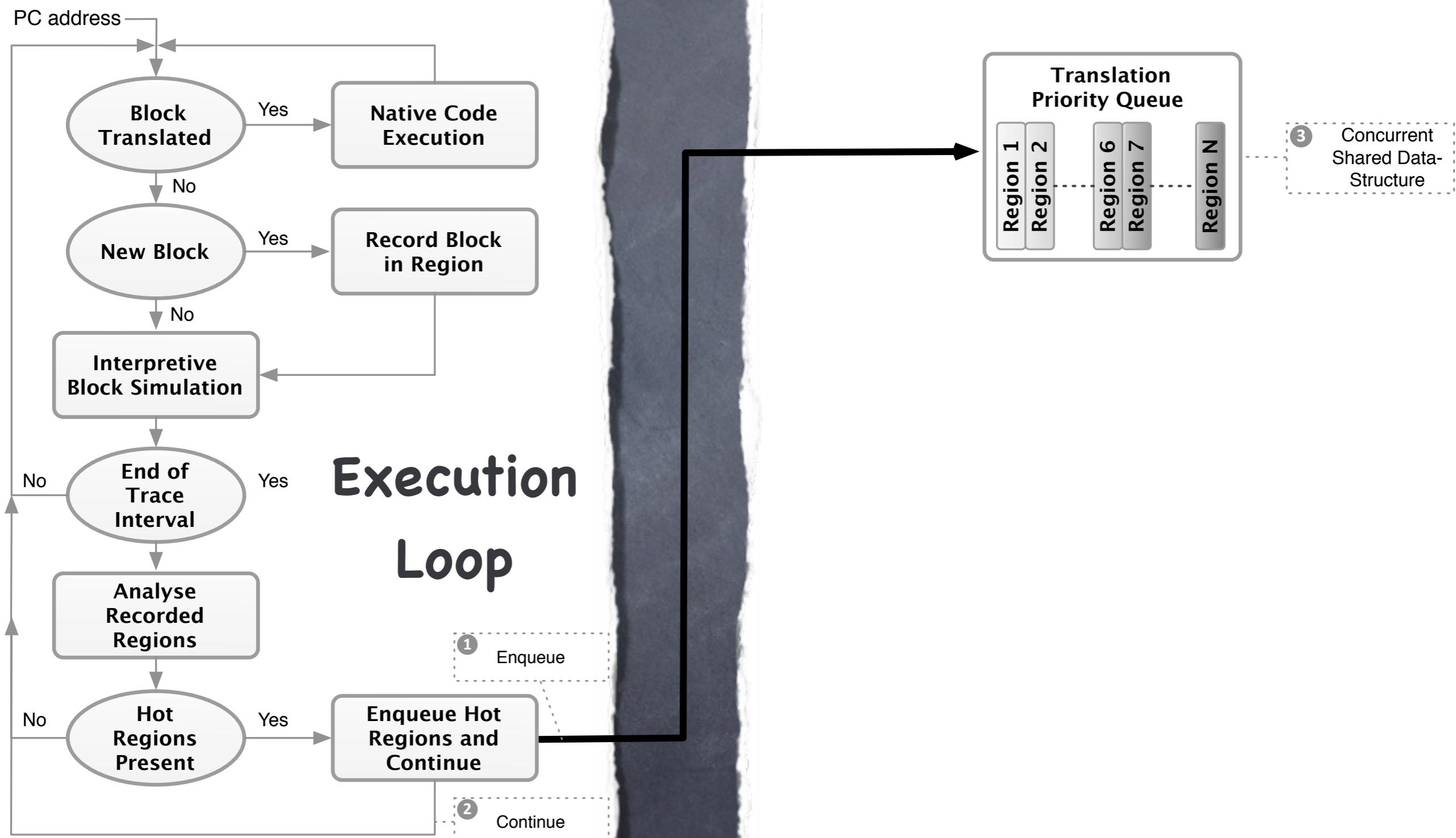
hide compilation latency
exploit task parallelism

Concurrent and Parallel JIT Compiler Design

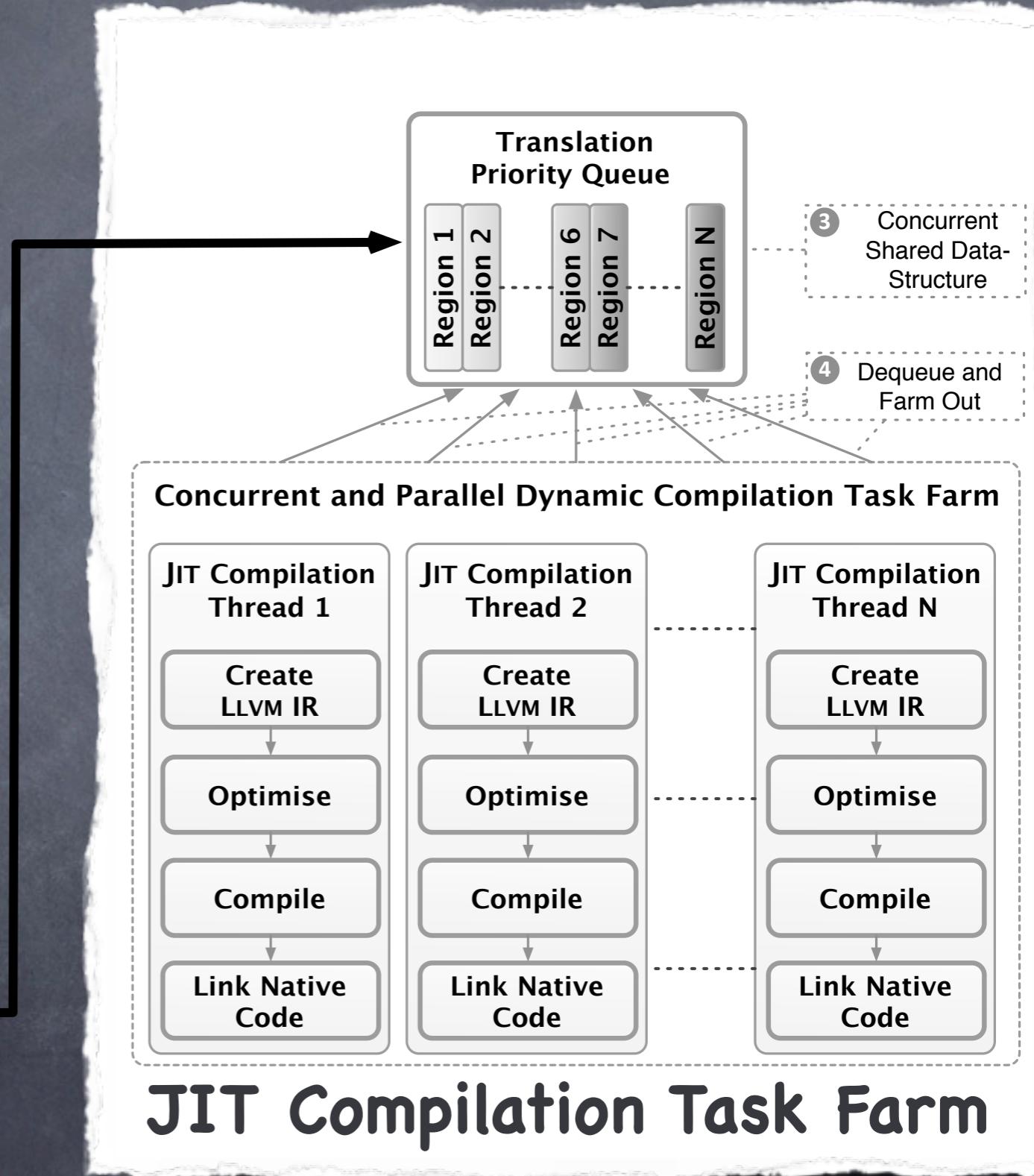
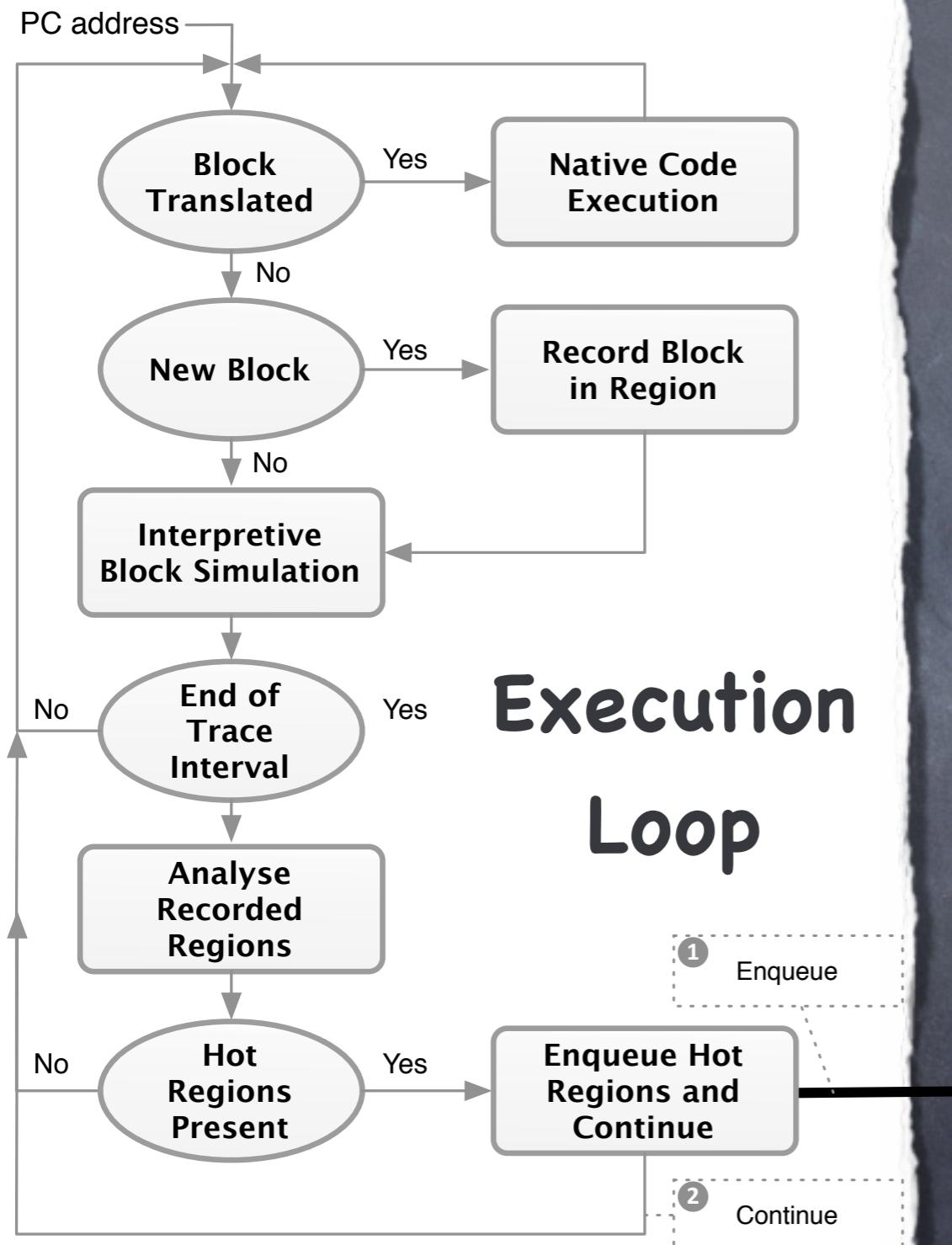


**Execution
Loop**

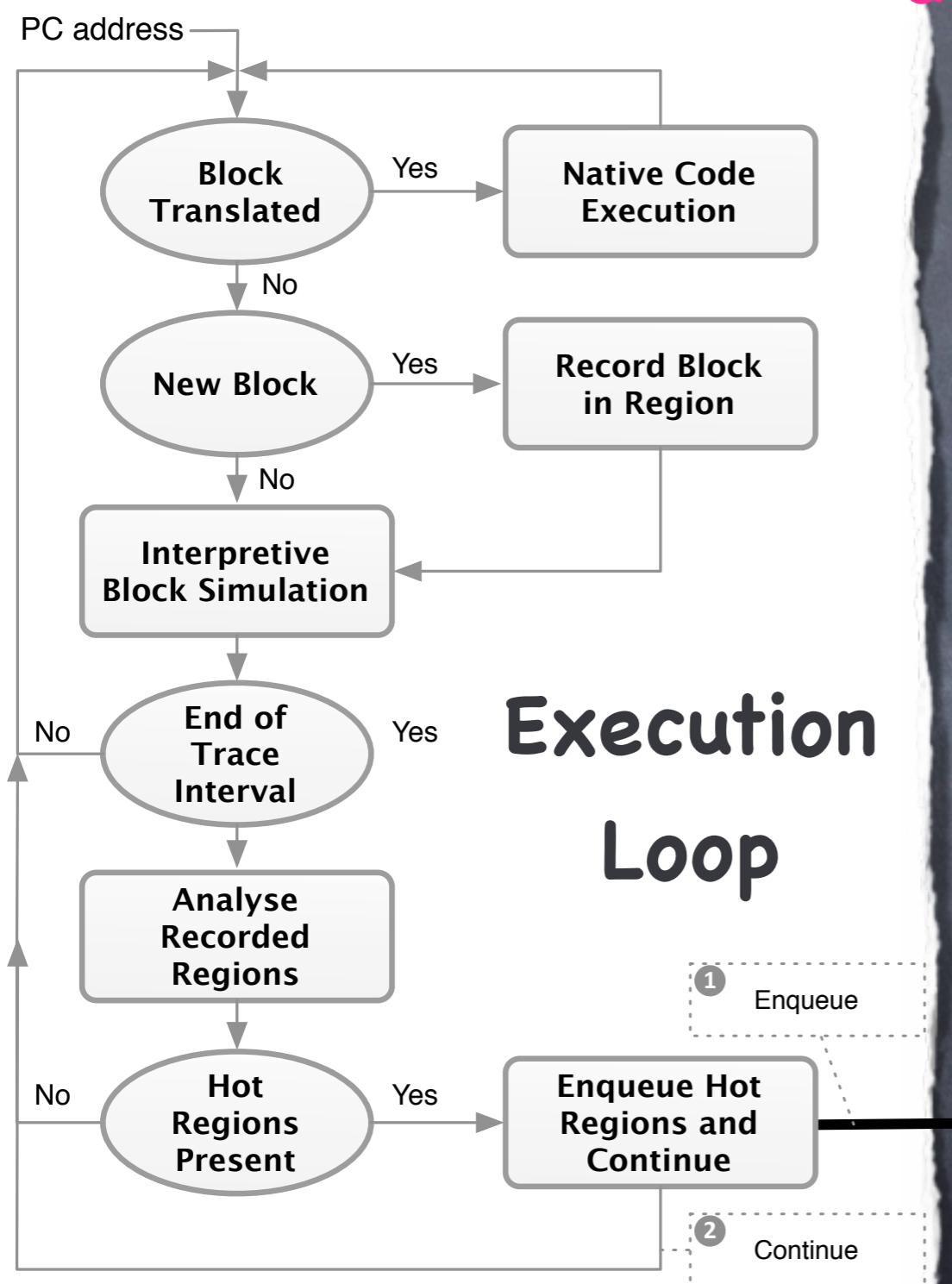
Concurrent and Parallel JIT Compiler Design



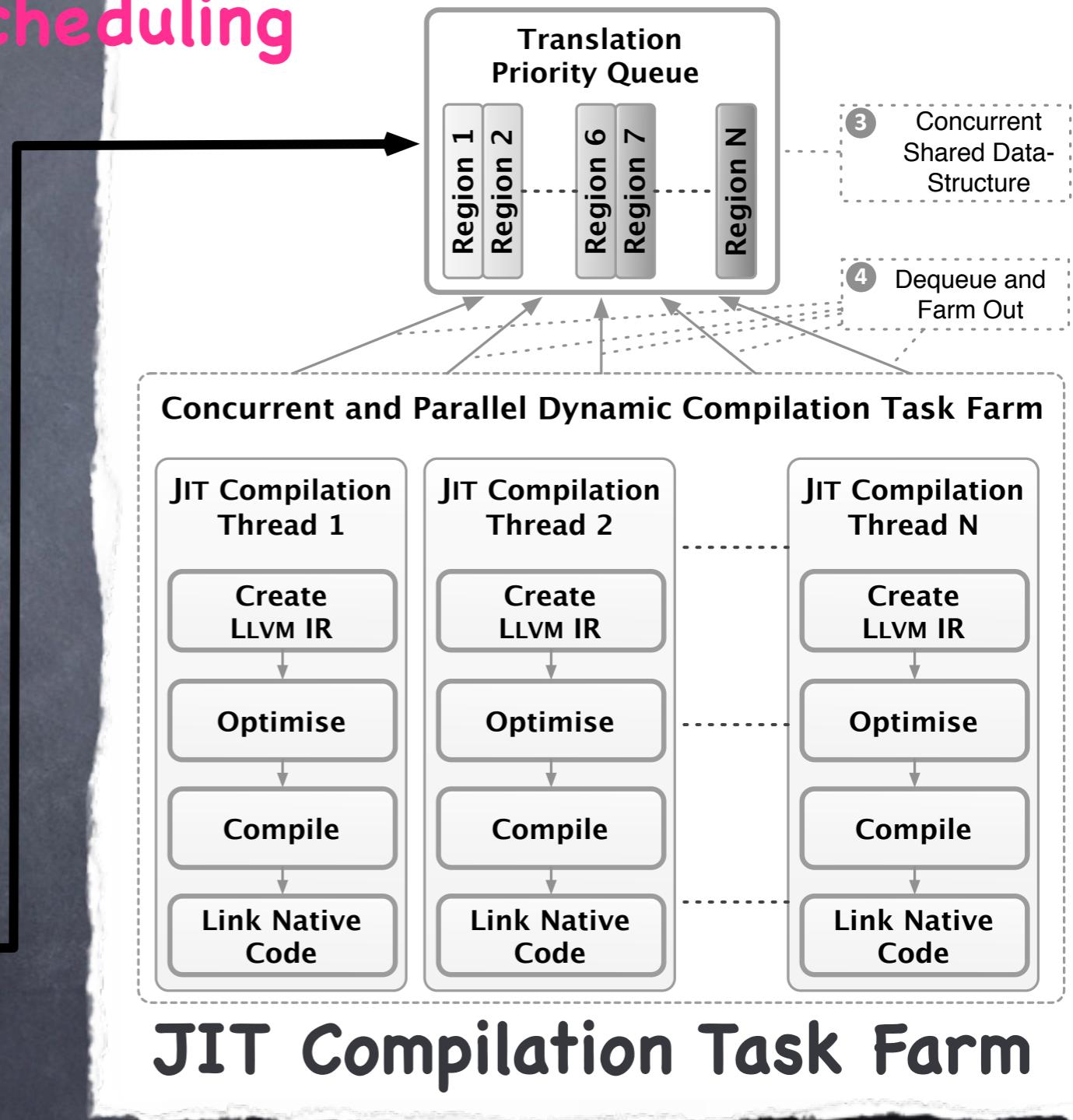
Concurrent and Parallel JIT Compiler Design



Concurrent and Parallel JIT Compiler Design



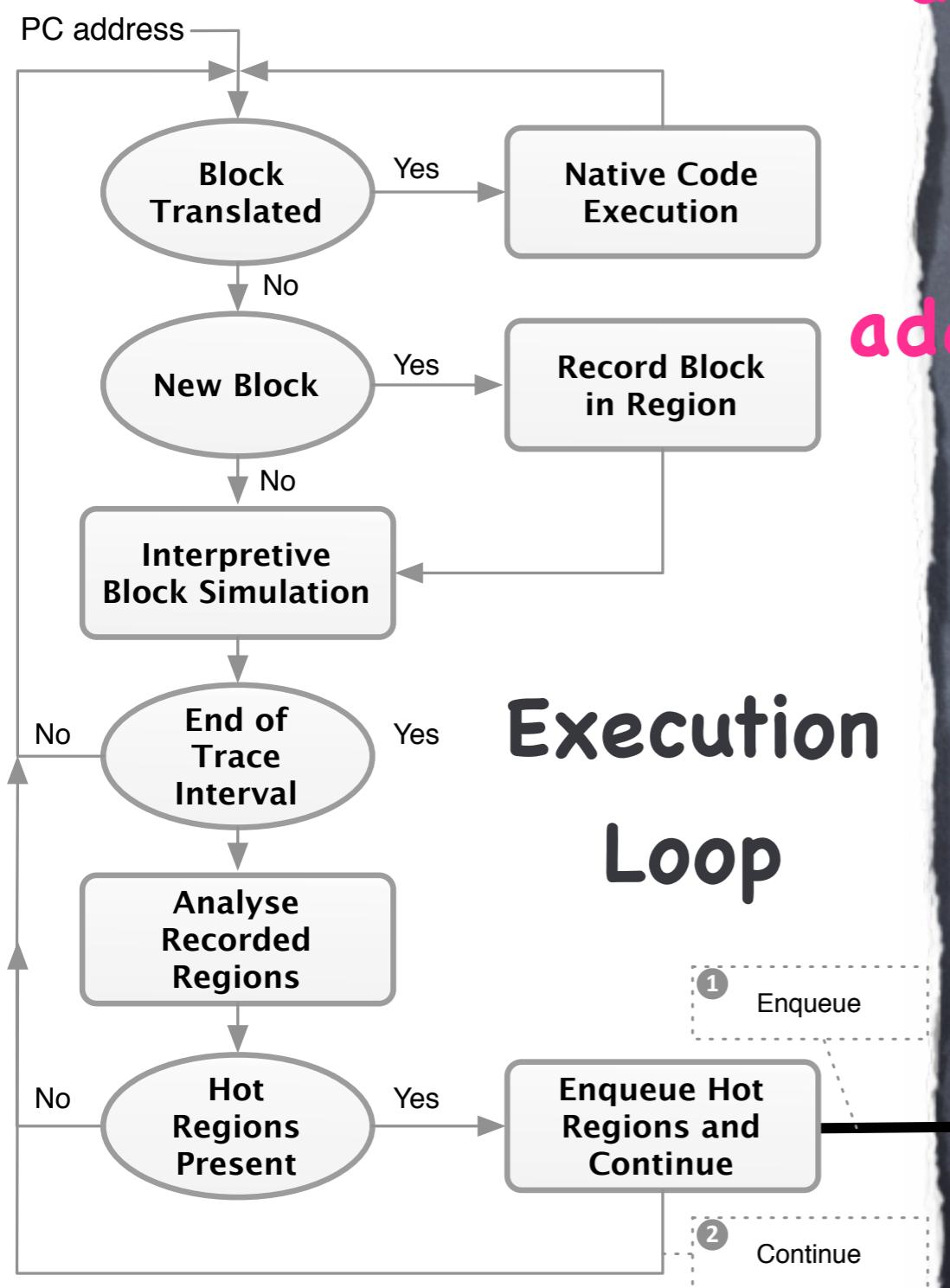
dynamic work
scheduling



Execution
Loop

JIT Compilation Task Farm

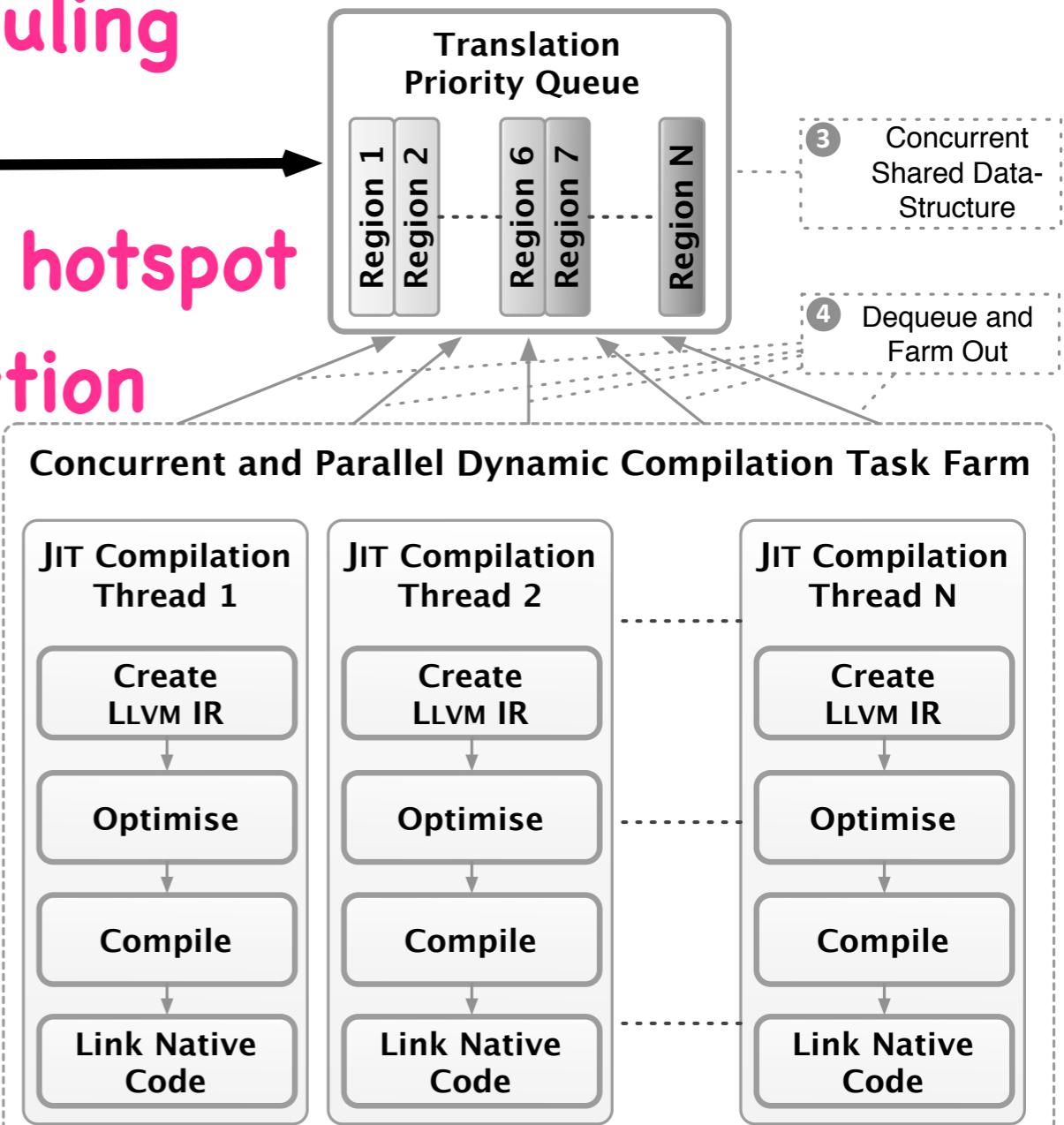
Concurrent and Parallel JIT Compiler Design



dynamic work
scheduling

adaptive hotspot
selection

Execution
Loop



JIT Compilation Task Farm

Concurrent and Parallel JIT Compiler Design Based on LLVM

- ⦿ Key Components:
 - ⦿ `llvm::LLVMContext` – owns and manages core ‘global’ data of LLVM’s core infrastructure
 - ⦿ `llvm::ExecutionEngine` – abstract, easy to use interface for implementation execution of LLVM modules
 - ⦿ state-of-the-art set of optimisation passes

Concurrent and Parallel JIT Compiler Design Based on LLVM

- Key Concepts:
 - dispatch of compilation units via thread-safe priority queue abstraction
 - each JIT compiler thread owns private `llvm::ExecutionEngine` instance enabling parallel JIT compilation without explicit synchronisation
 - asynchronous registration of compiled native code

Concurrent and Parallel JIT Compiler Design Based on LLVM

```
class JITThread : public Thread {  
private:  
    llvm::LLVMContext*      CTX_;   // per thread LLVMContext  
    llvm::Module*           MOD_;   // per thread main Module  
    llvm::ExecutionEngine*  ENG_;   // per thread ExecutionEngine  
    ...  
public:  
}  
}
```

Concurrent and Parallel JIT Compiler Design Based on LLVM

```
class JITThread : public Thread {
private:
    llvm::LLVMContext*      CTX_;   // per thread LLVMContext
    llvm::Module*            MOD_;   // per thread main Module
    llvm::ExecutionEngine*   ENG_;   // per thread ExecutionEngine
    ...
public:
    void create() {
        CTX_ = new llvm::LLVMContext();
        MOD_ = new llvm::Module("module", *CTX_);
        ENG_ = llvm::EngineBuilder(MOD_)
            .setEngineKind(llvm::EngineKind::JIT)
            .create();
        ...
    }
}
```

Concurrent and Parallel JIT Compiler Design Based on LLVM

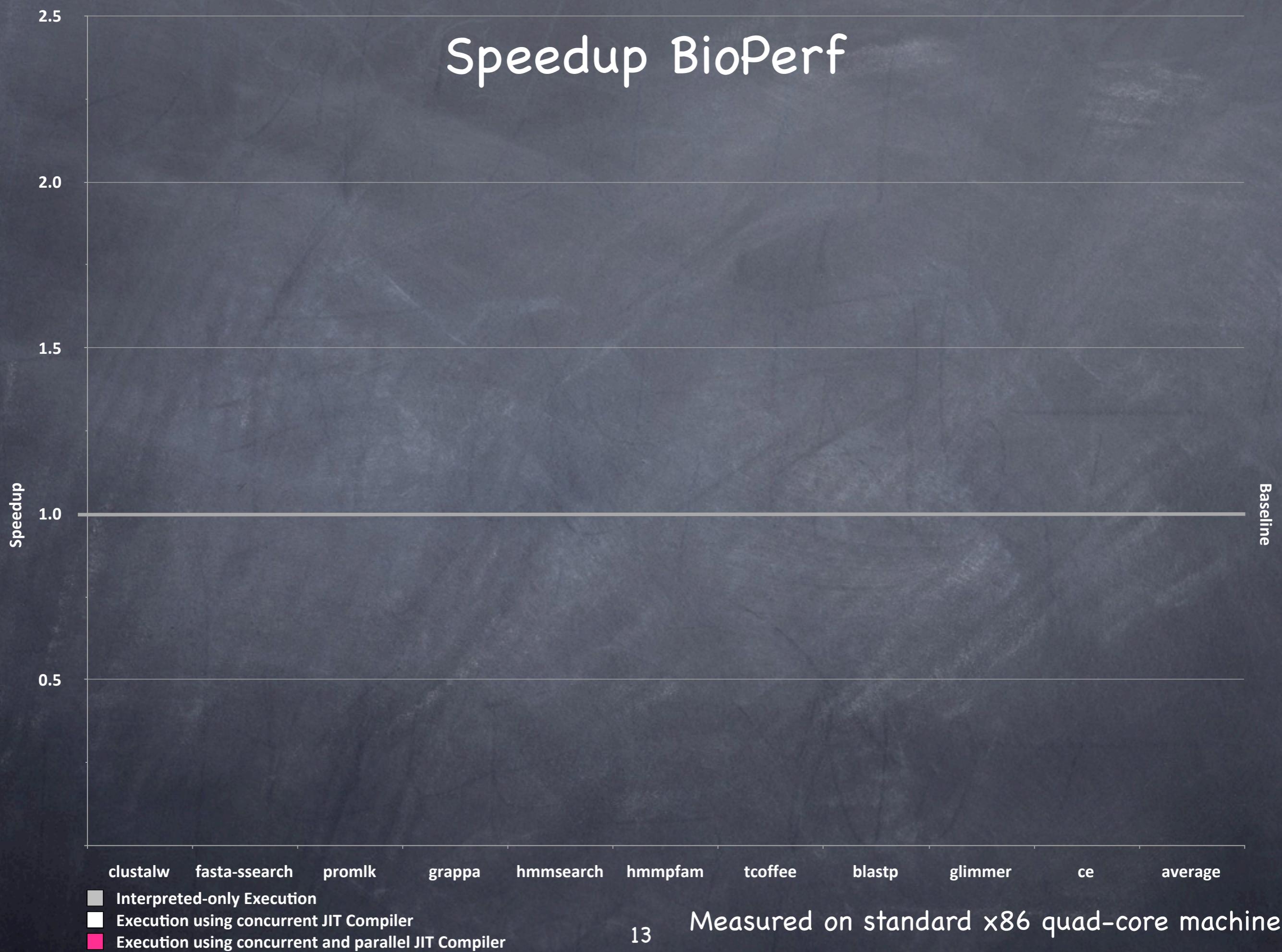
```
class JITThread : public Thread {
private:
    llvm::LLVMContext*      CTX_;   // per thread LLVMContext
    llvm::Module*            MOD_;   // per thread main Module
    llvm::ExecutionEngine*   ENG_;   // per thread ExecutionEngine
    ...
public:
    void create() {
        CTX_ = new llvm::LLVMContext();
        MOD_ = new llvm::Module("module", *CTX_);
        ENG_ = llvm::EngineBuilder(MOD_)
            .setEngineKind(llvm::EngineKind::JIT)
            .create();
        ...
    }

    void run() {
        for ( ; /* ever */ ; ) {
            queue.mutex.acquire();
            while (queue.empty()) {           // wait for work if queue is empty
                queue.condvar.wait(queue.mutex);
            }
            WorkUnit* u = queue.top();       // retrieve compilation unit
            queue.pop();
            queue.mutex.release();
            llvm::Function* f = Codegen(u); // generate IR
            void* native = ENG_->getPointerToFunction(f); // run JIT
            // register native translation for execution
            ...
        }
    }
}
```

Evaluation

- ⦿ Extensive evaluation using over 60 industry standard benchmarks built for ARCompact RISC platform:
 - ⦿ BioPERF
 - ⦿ SPEC CPU 2006
 - ⦿ EEMBC and CoreMark
- ⦿ Target Platform:
 - ⦿ ARCompact RISC ISA targeting ARC 700 processor
- ⦿ Simulation Platform:
 - ⦿ standard x86 Dell Intel Xeon quad-core machine

Speedup BioPerf



Speedup BioPerf

Speedup

2.5

2.0

1.5

1.0

0.5

0.34

0.11

0.06

0.47

0.94

0.81

0.68

0.44

0.19

0.08

Baseline

clustalw fasta-search promlk grappa hmmsearch hmmpfam tcoffee blastp glimmer ce average

- Interpreted-only Execution
- Execution using concurrent JIT Compiler
- Execution using concurrent and parallel JIT Compiler

Speedup BioPerf

Speedup

2.5

2.0

1.5

1.0

0.5

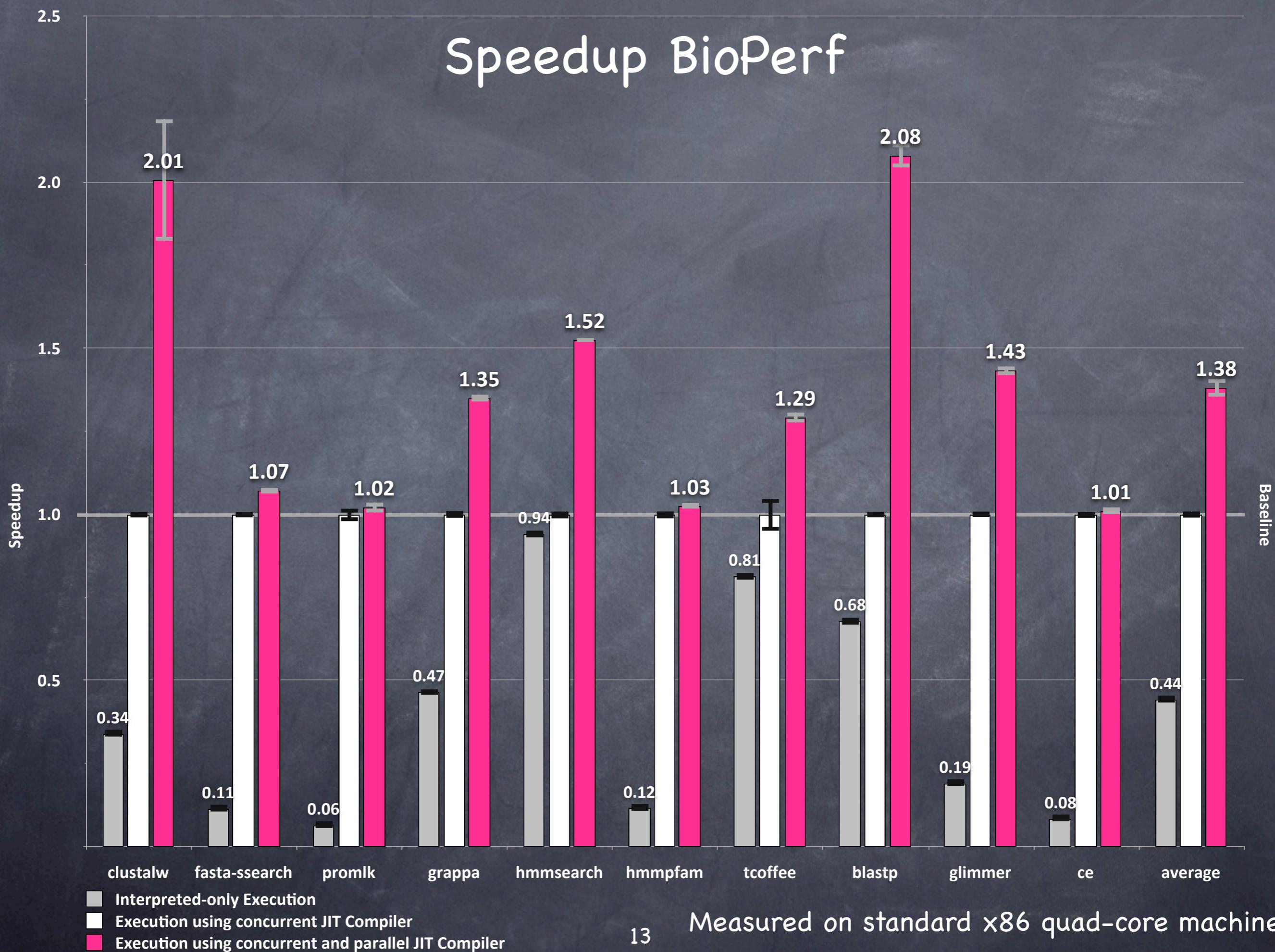
0.0

Baseline

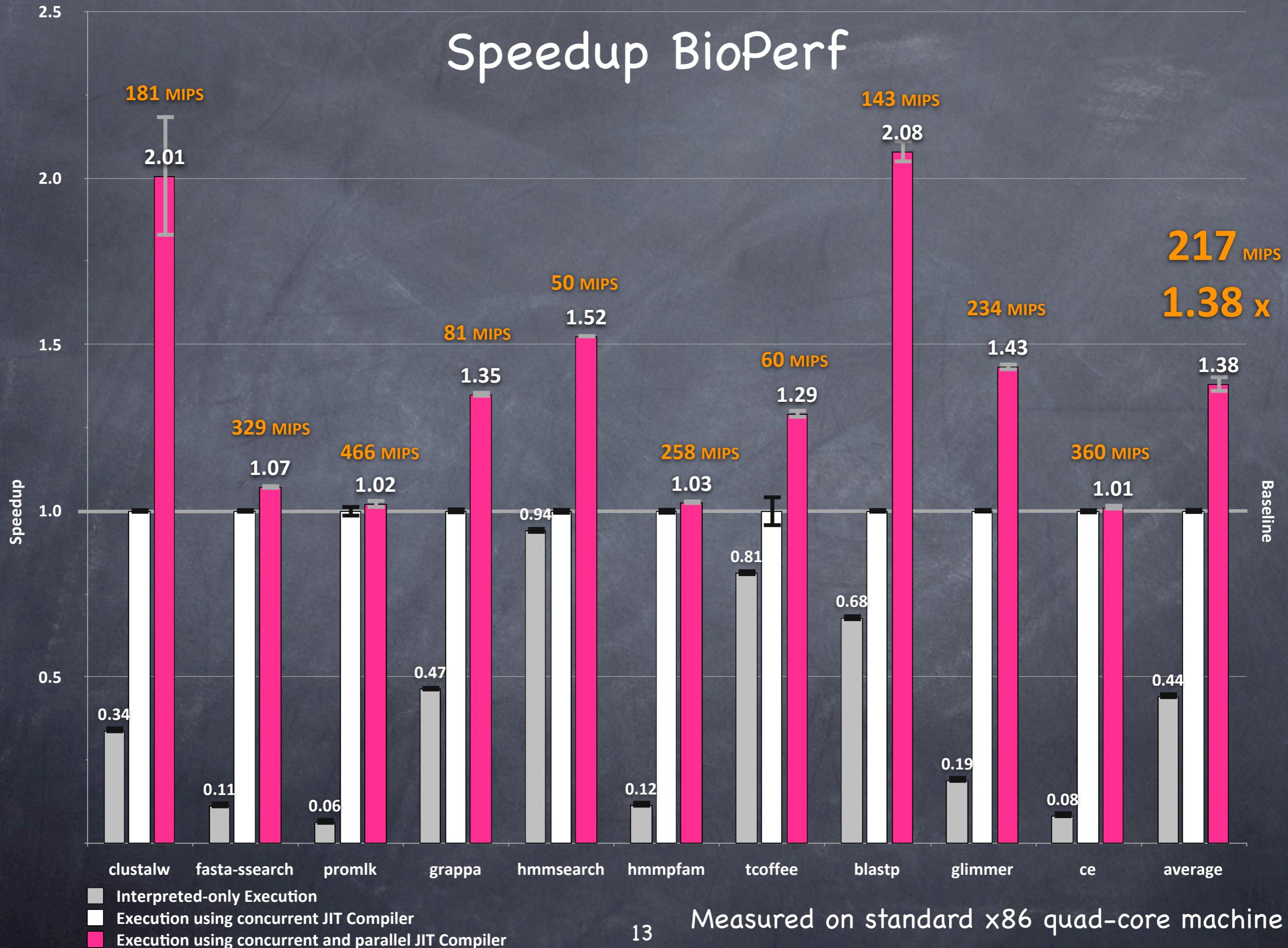
clustalw fasta-search promlk grappa hmmsearch hmmpfam tcoffee blastp glimmer ce average

- Interpreted-only Execution
- Execution using concurrent JIT Compiler
- Execution using concurrent and parallel JIT Compiler

Speedup BioPerf



Speedup BioPerf

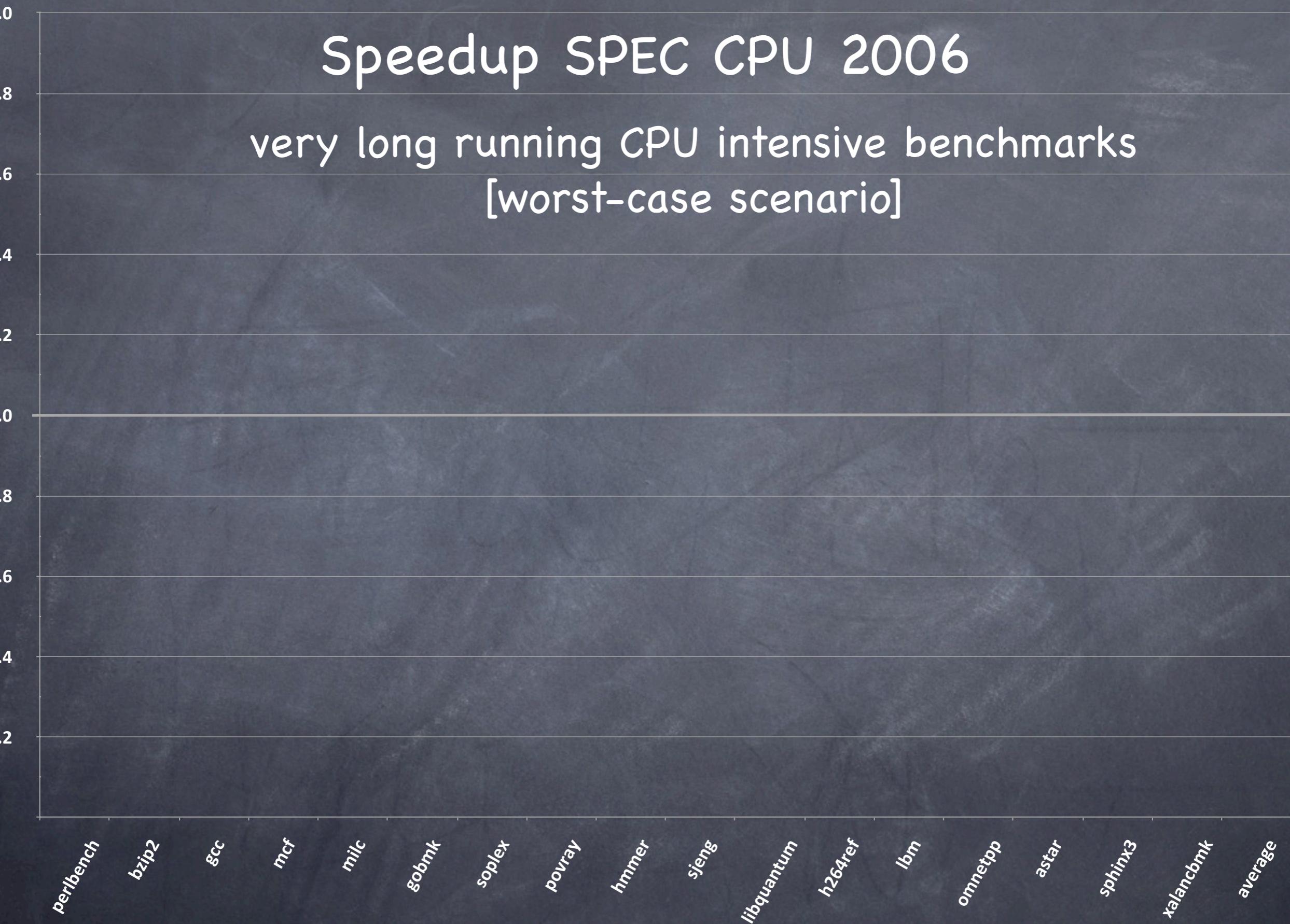


Speedup SPEC CPU 2006

very long running CPU intensive benchmarks
[worst-case scenario]

Speedup

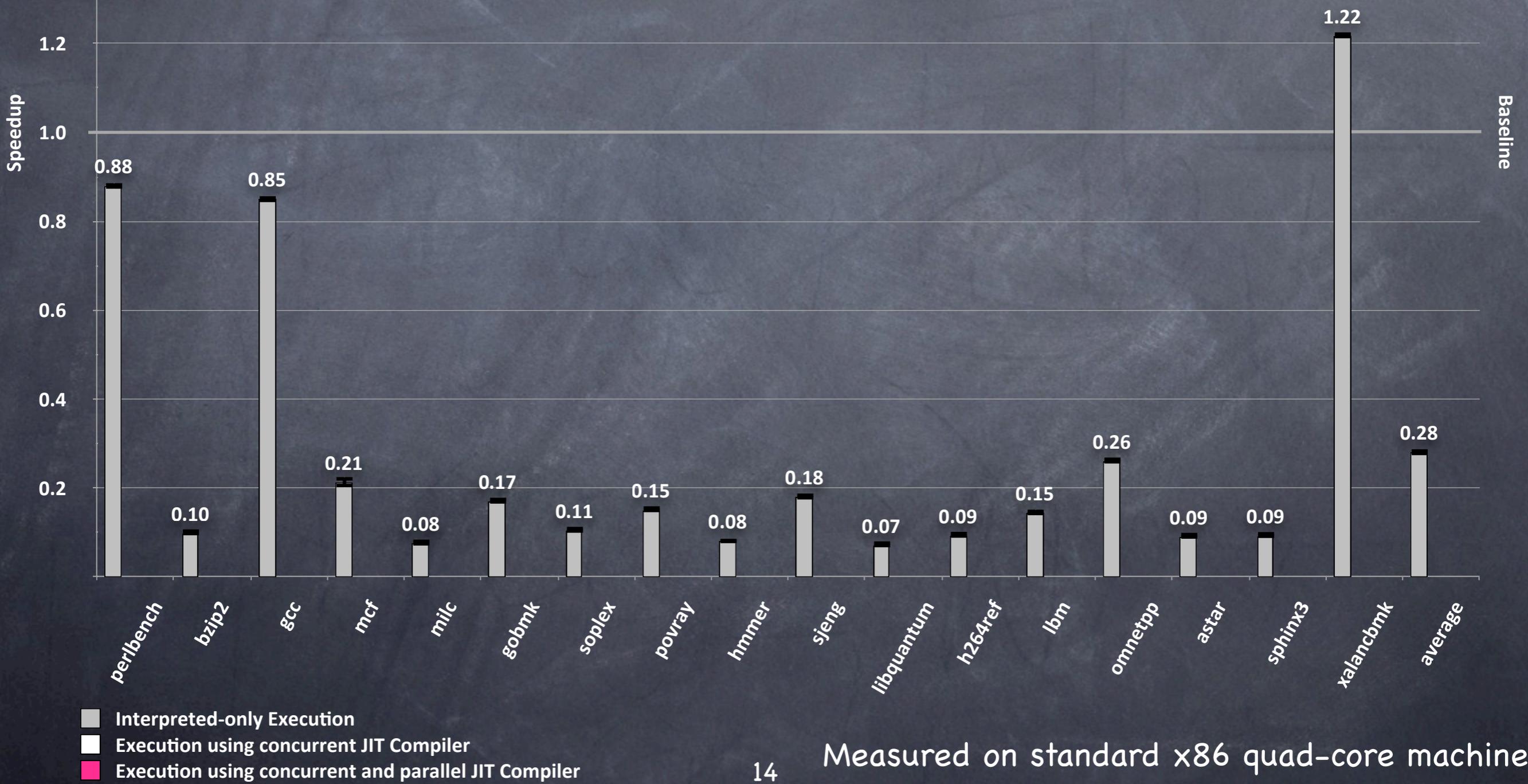
Baseline



- Interpreted-only Execution
- Execution using concurrent JIT Compiler
- Execution using concurrent and parallel JIT Compiler

Speedup SPEC CPU 2006

very long running CPU intensive benchmarks
[worst-case scenario]

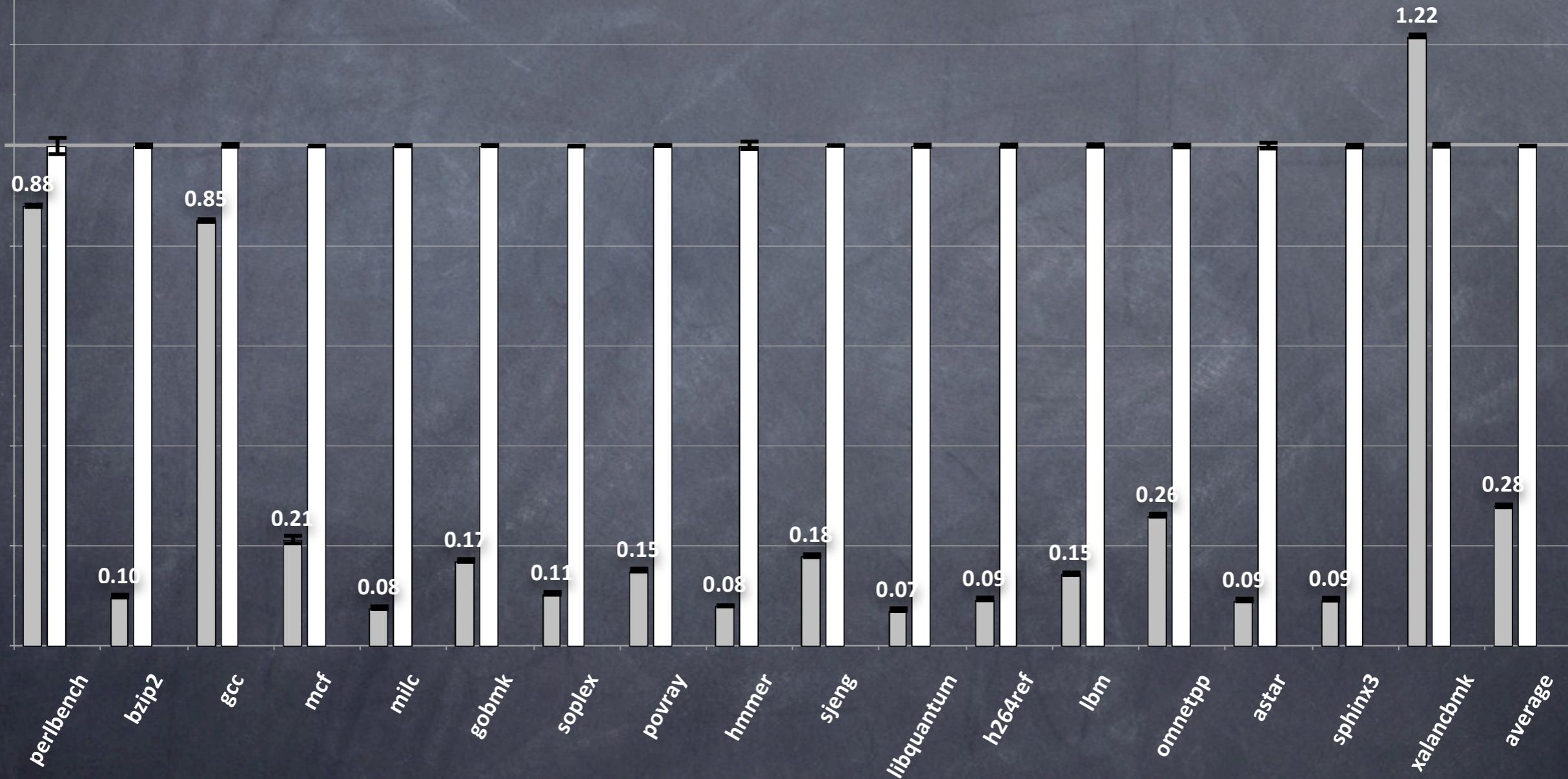


Speedup SPEC CPU 2006

very long running CPU intensive benchmarks
[worst-case scenario]

Speedup

Baseline

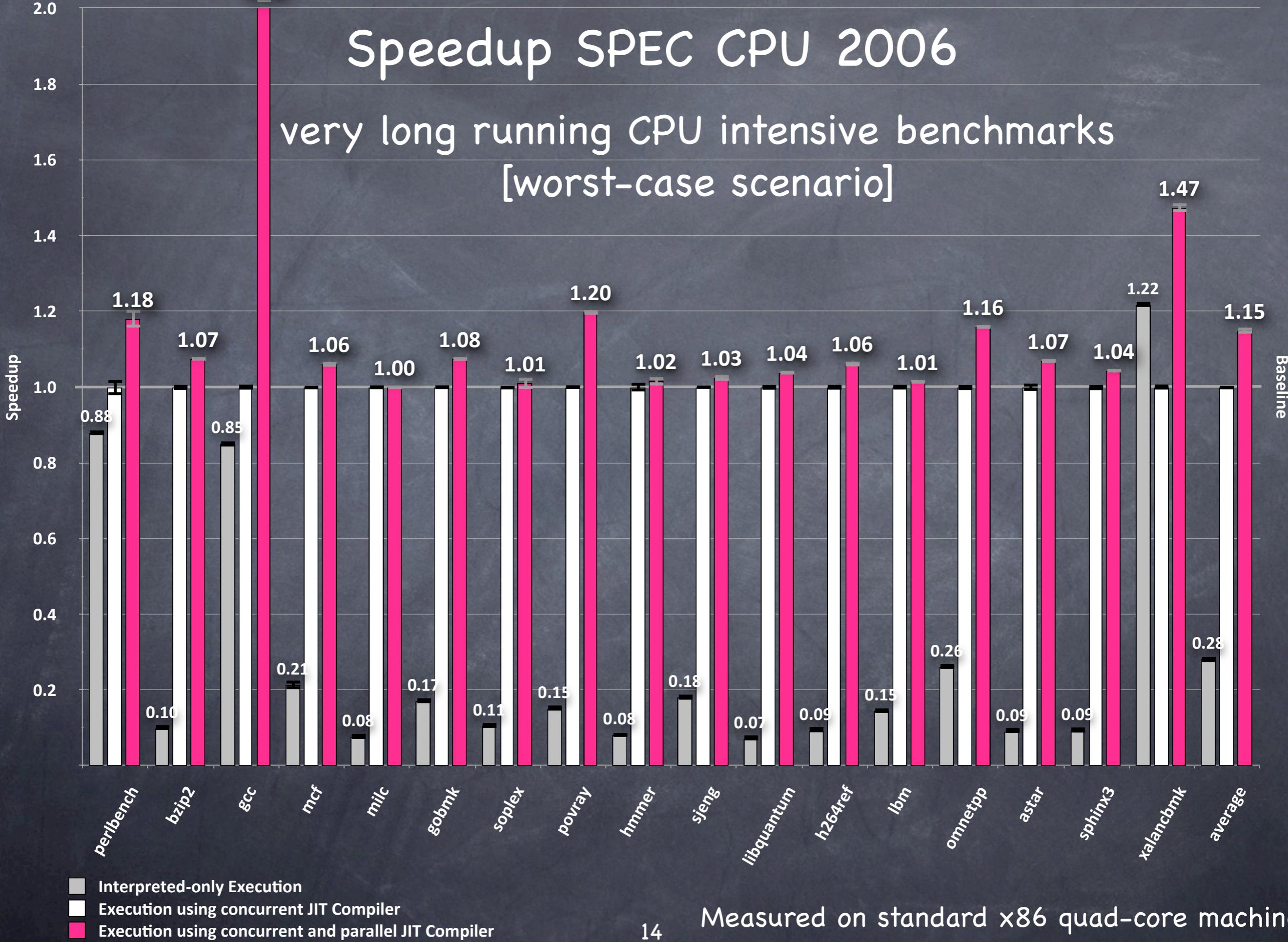


- Interpreted-only Execution
- Execution using concurrent JIT Compiler
- Execution using concurrent and parallel JIT Compiler

2.04

Speedup SPEC CPU 2006

very long running CPU intensive benchmarks
 [worst-case scenario]



- Interpreted-only Execution
- Execution using concurrent JIT Compiler
- Execution using concurrent and parallel JIT Compiler

2.04

Speedup SPEC CPU 2006

very long running CPU intensive benchmarks
[worst-case scenario]

24 MIPS

1.47

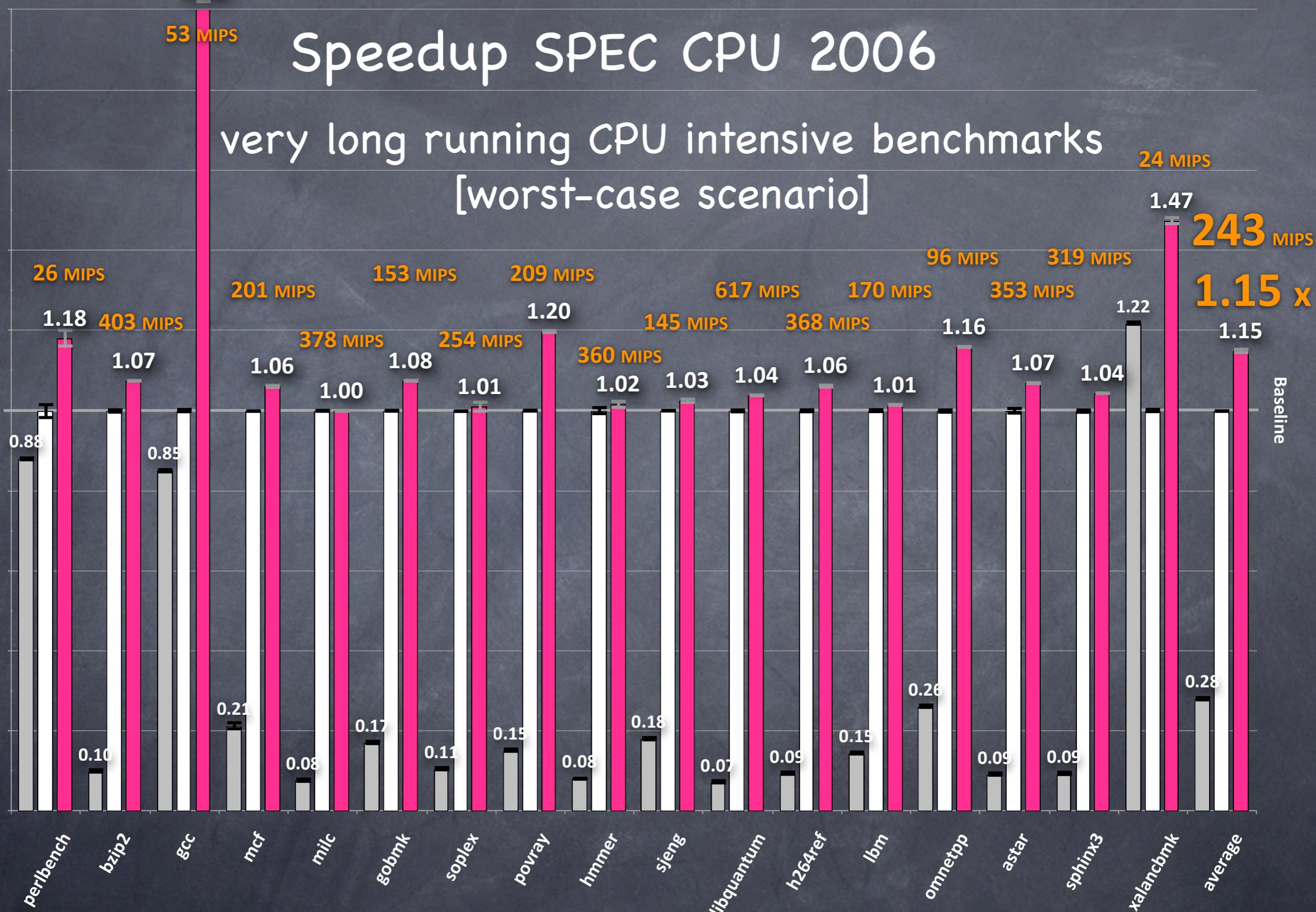
243 MIPS

1.15

1.15

Baseline

Speedup



Interpreted-only Execution

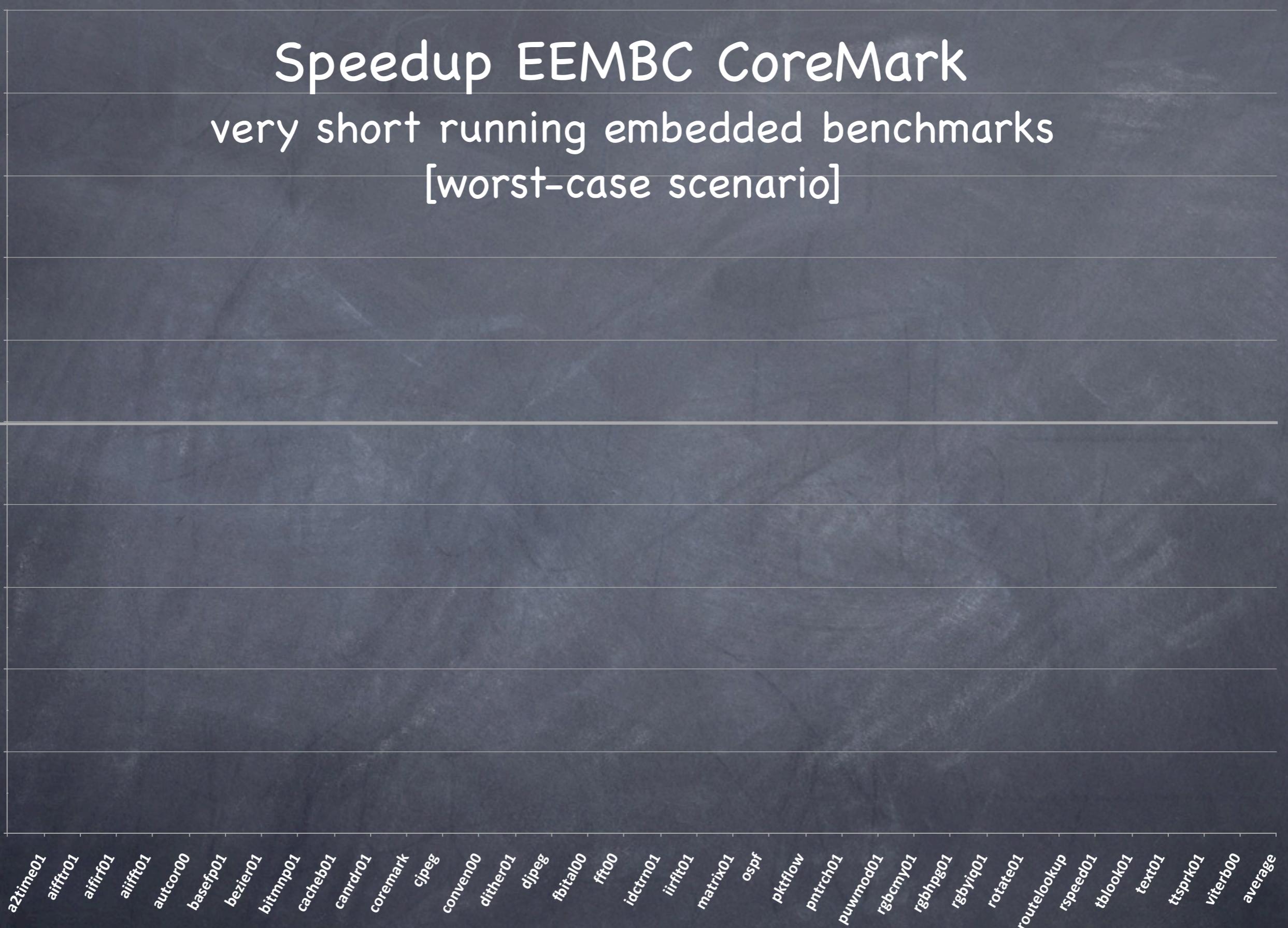
Execution using concurrent JIT Compiler

Execution using concurrent and parallel JIT Compiler

Speedup EEMBC CoreMark

very short running embedded benchmarks
[worst-case scenario]

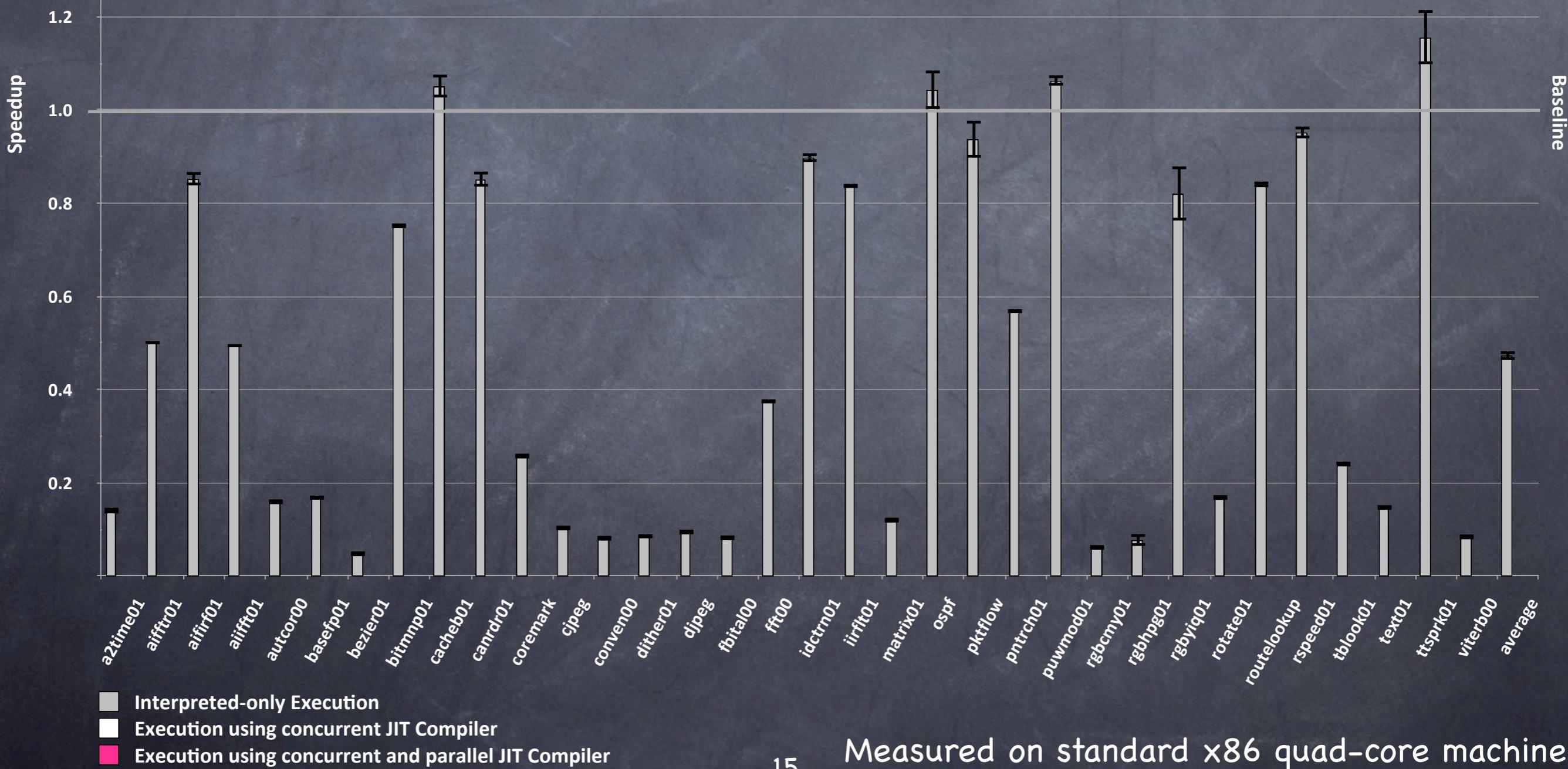
Speedup



- Interpreted-only Execution
- Execution using concurrent JIT Compiler
- Execution using concurrent and parallel JIT Compiler

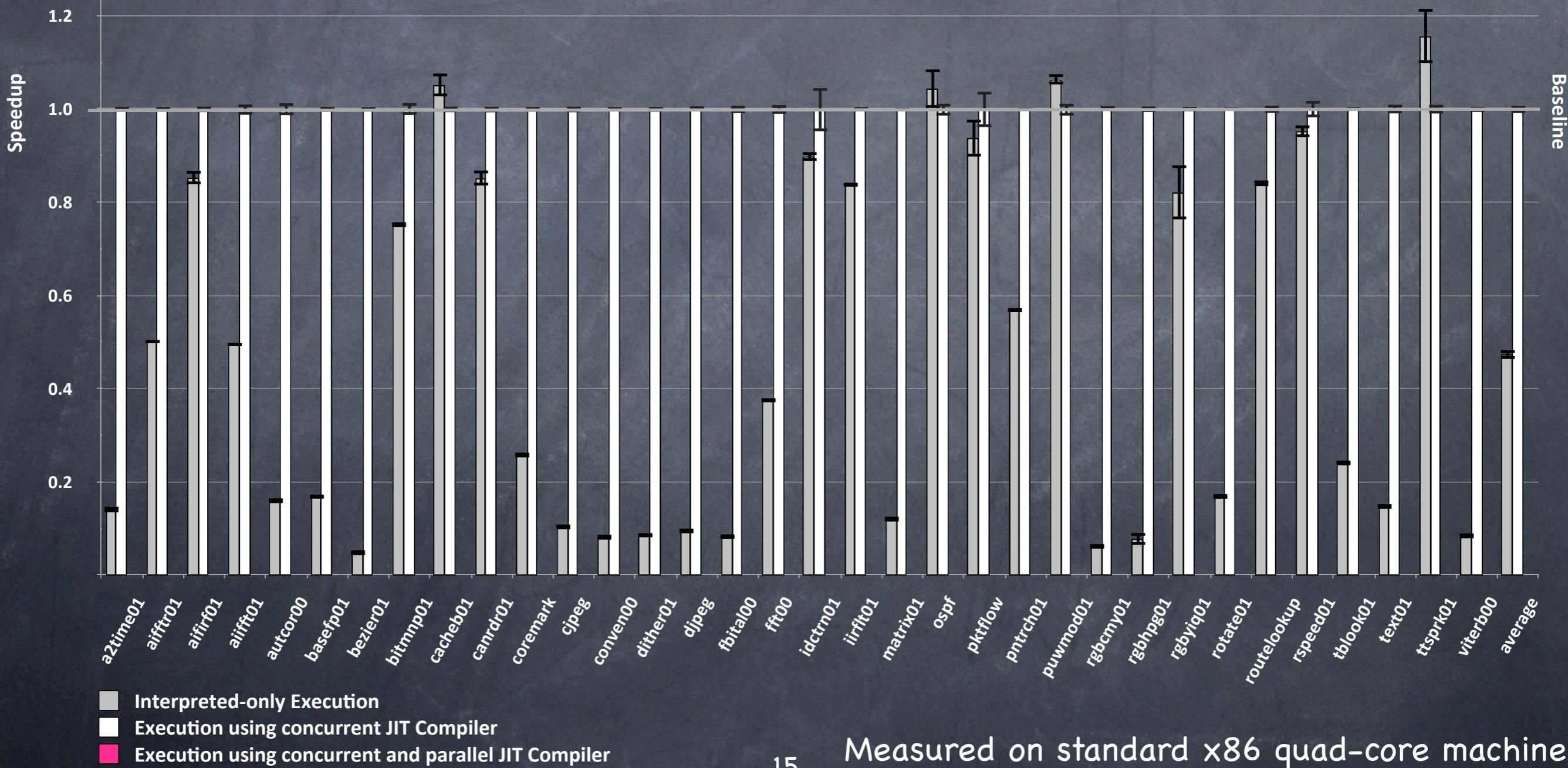
Speedup EEMBC CoreMark

very short running embedded benchmarks
[worst-case scenario]



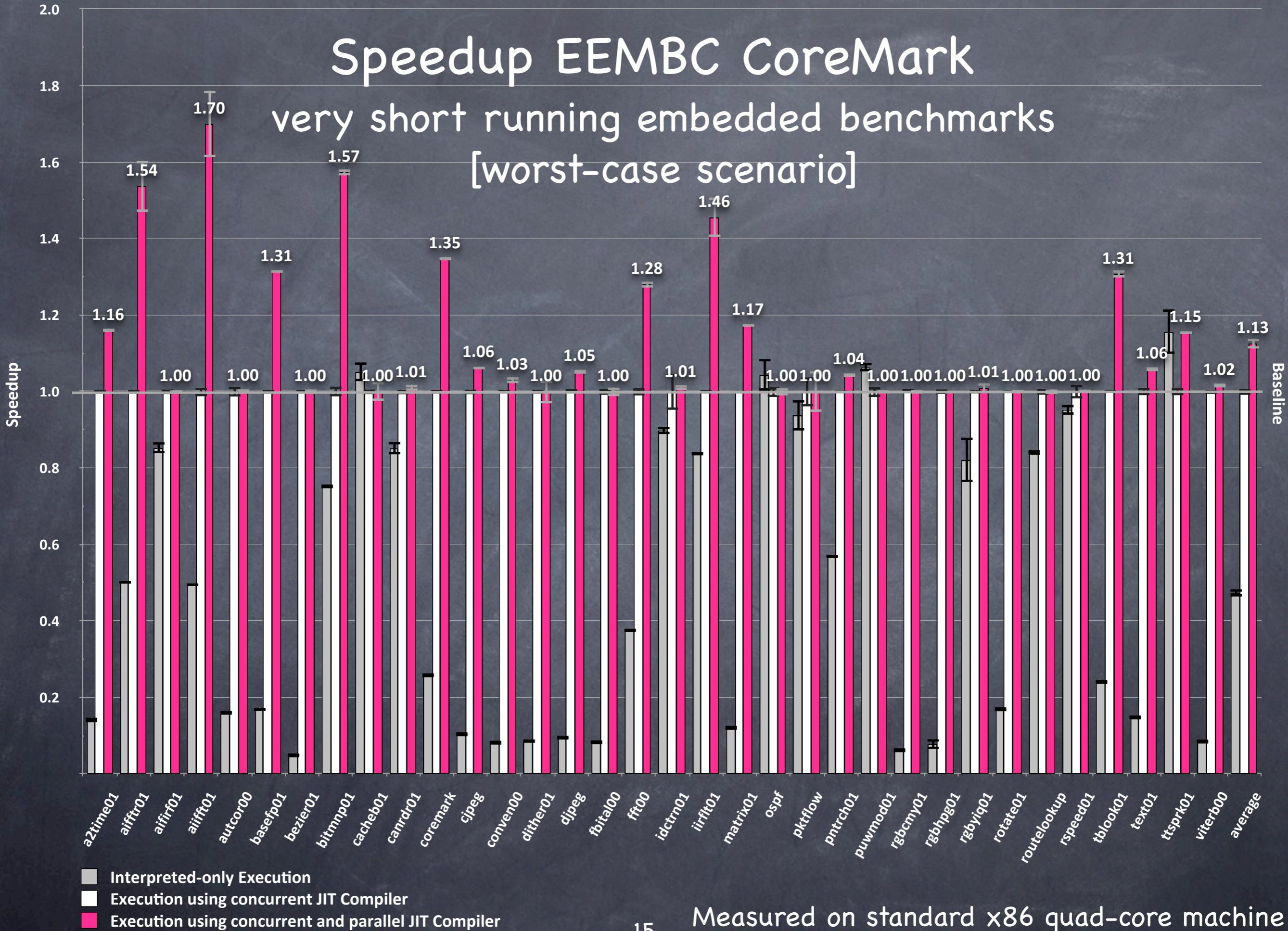
Speedup EEMBC CoreMark

very short running embedded benchmarks
[worst-case scenario]



Speedup EEMBC CoreMark

very short running embedded benchmarks
[worst-case scenario]



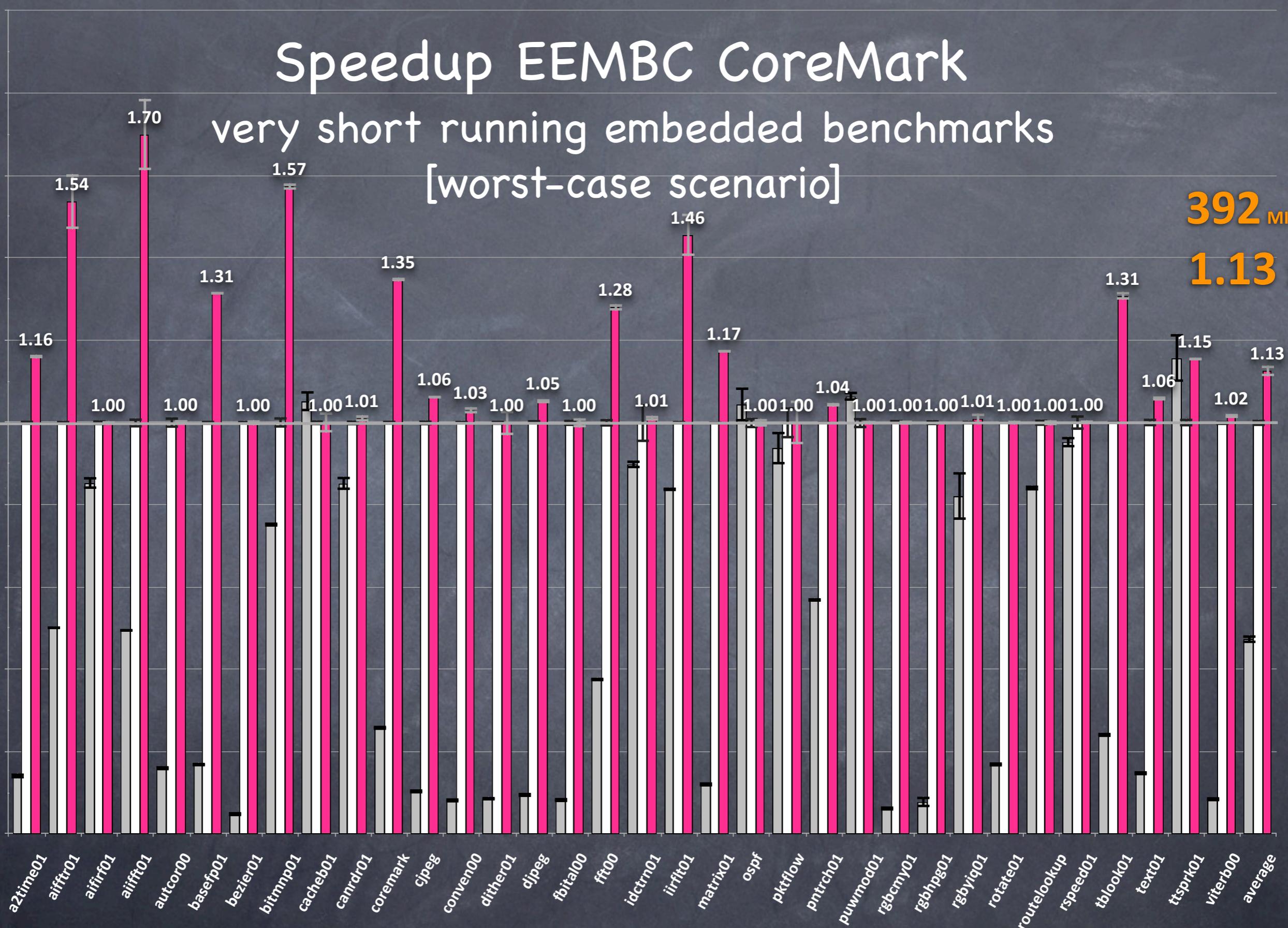
Speedup EEMBC CoreMark

very short running embedded benchmarks

392 MIPS

1.13 x

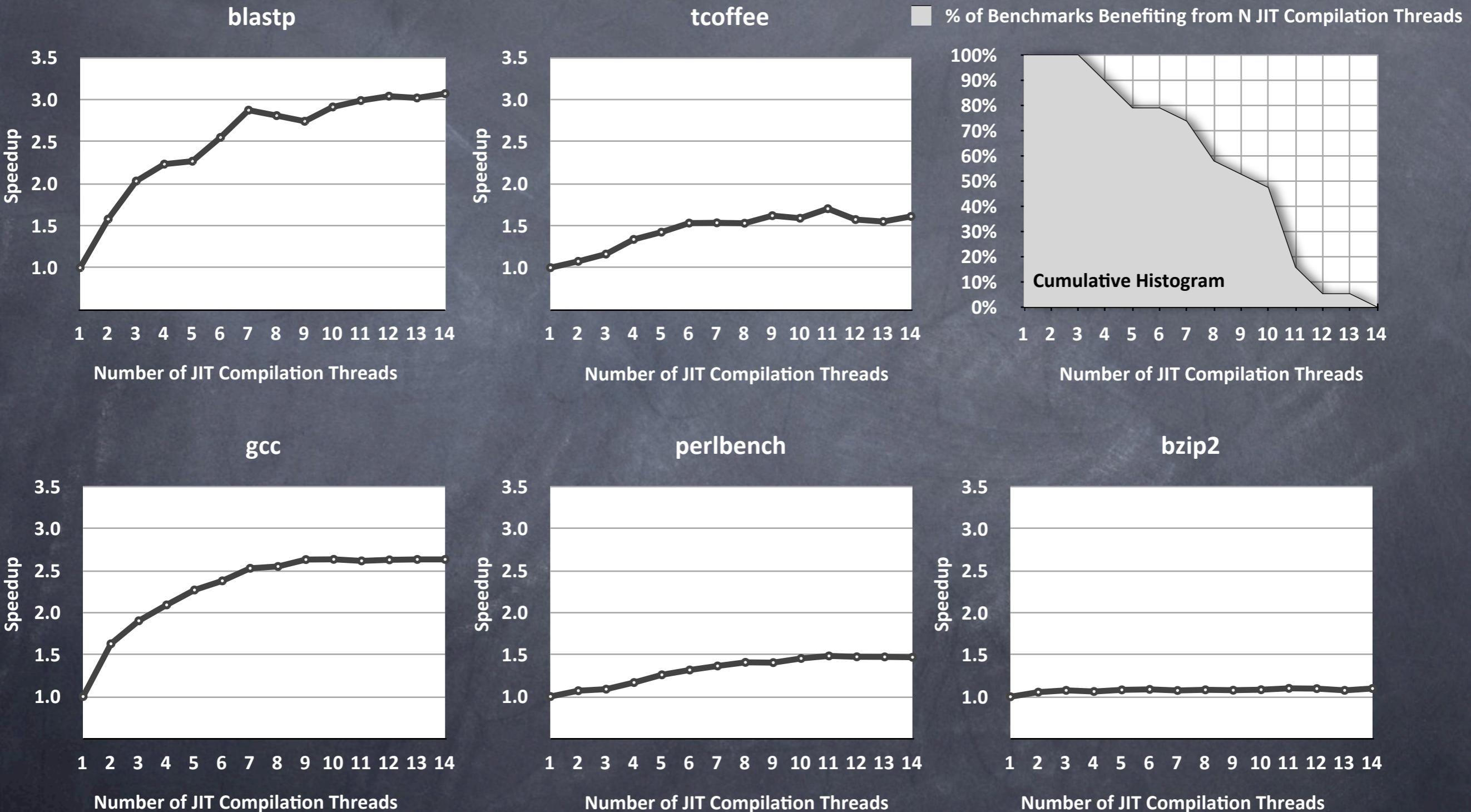
Baseline



- Interpreted-only Execution
 - Execution using concurrent JIT Compiler
 - Execution using concurrent and parallel JIT Compiler

How far does it scale?

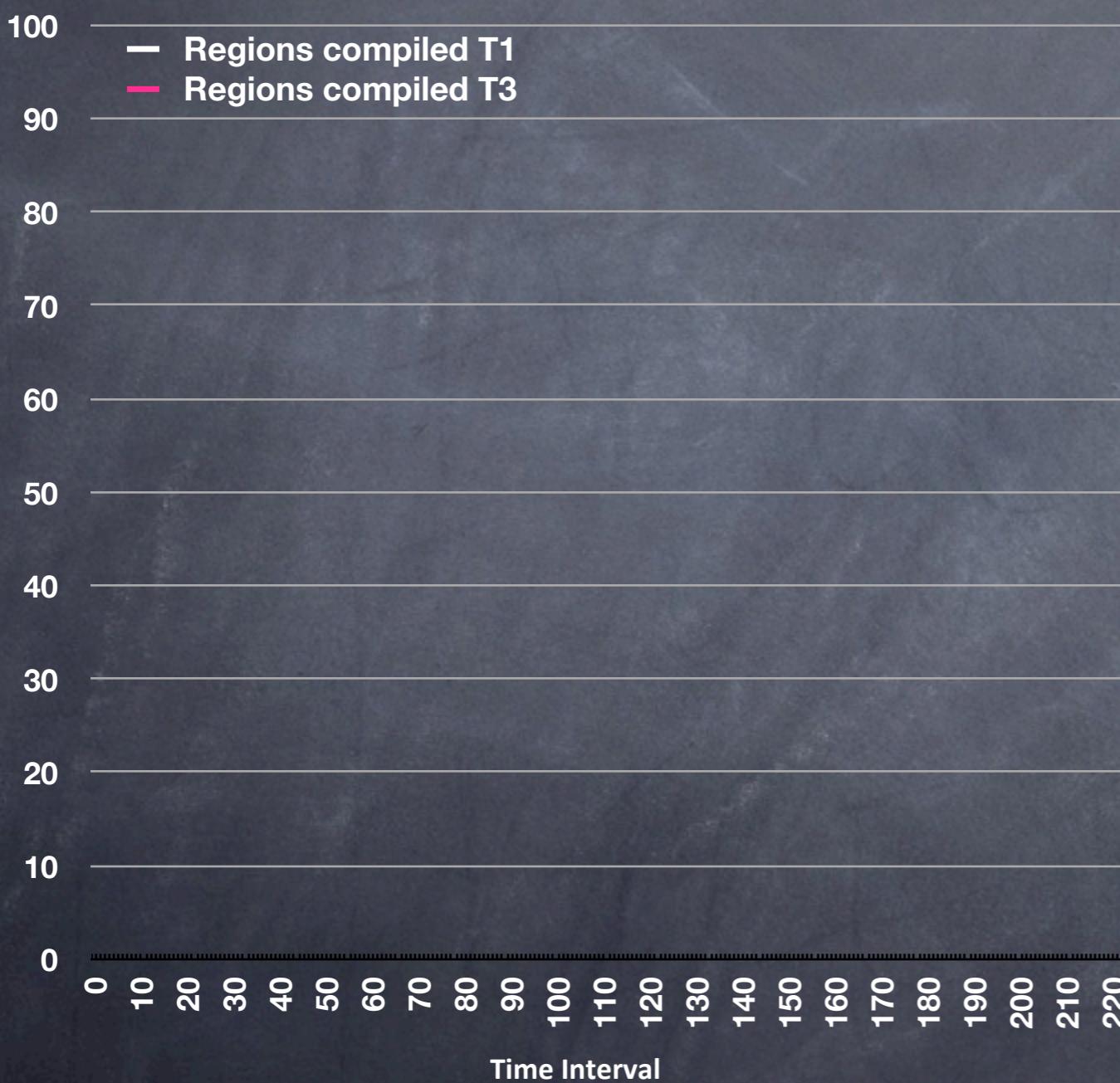
What is a sensible number of JIT compilation threads?



Effect of Concurrent and Parallel JIT Compilation on Throughput

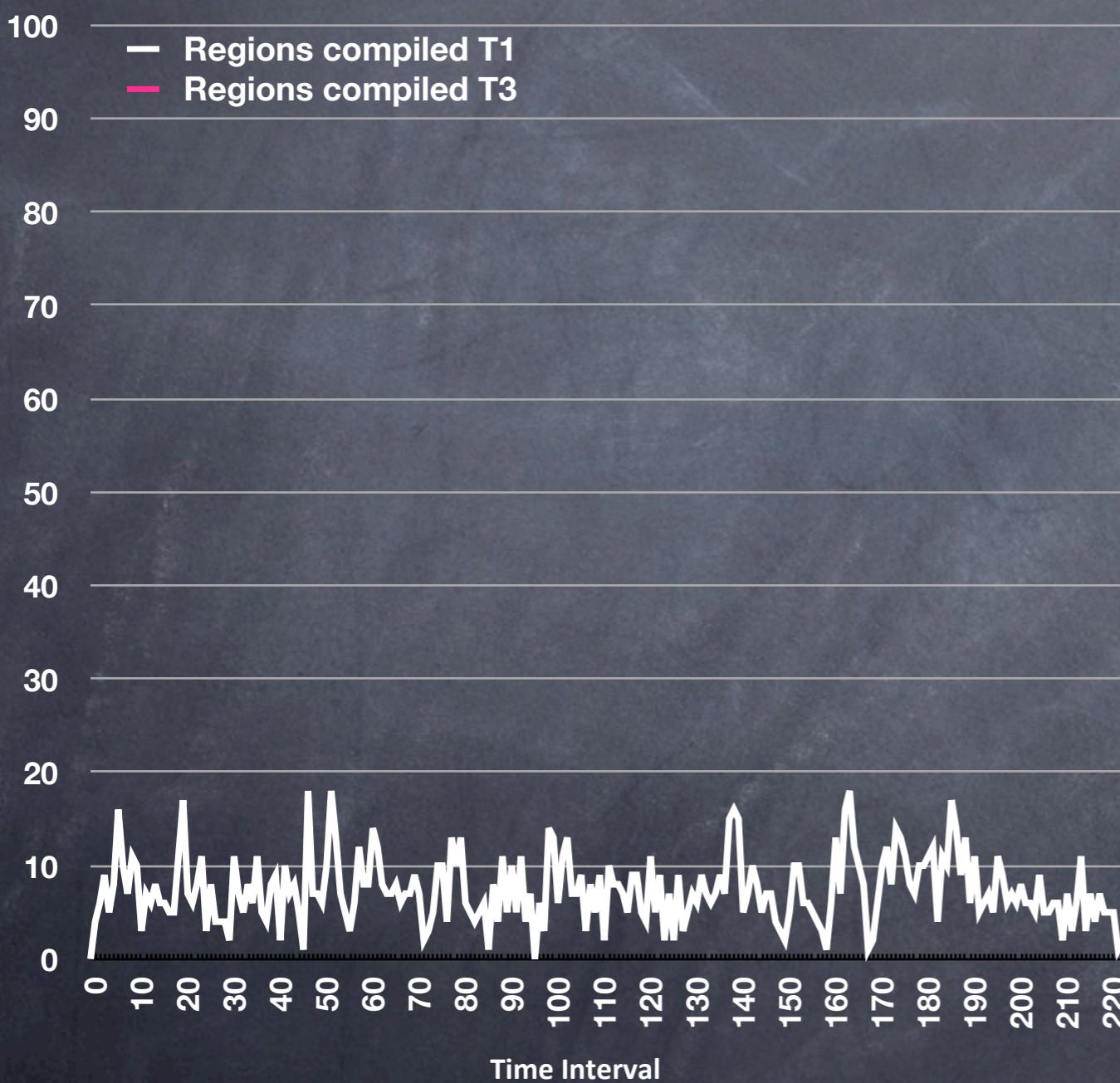
Effect of Concurrent and Parallel JIT Compilation on Throughput

403.gcc - Regions Compiled



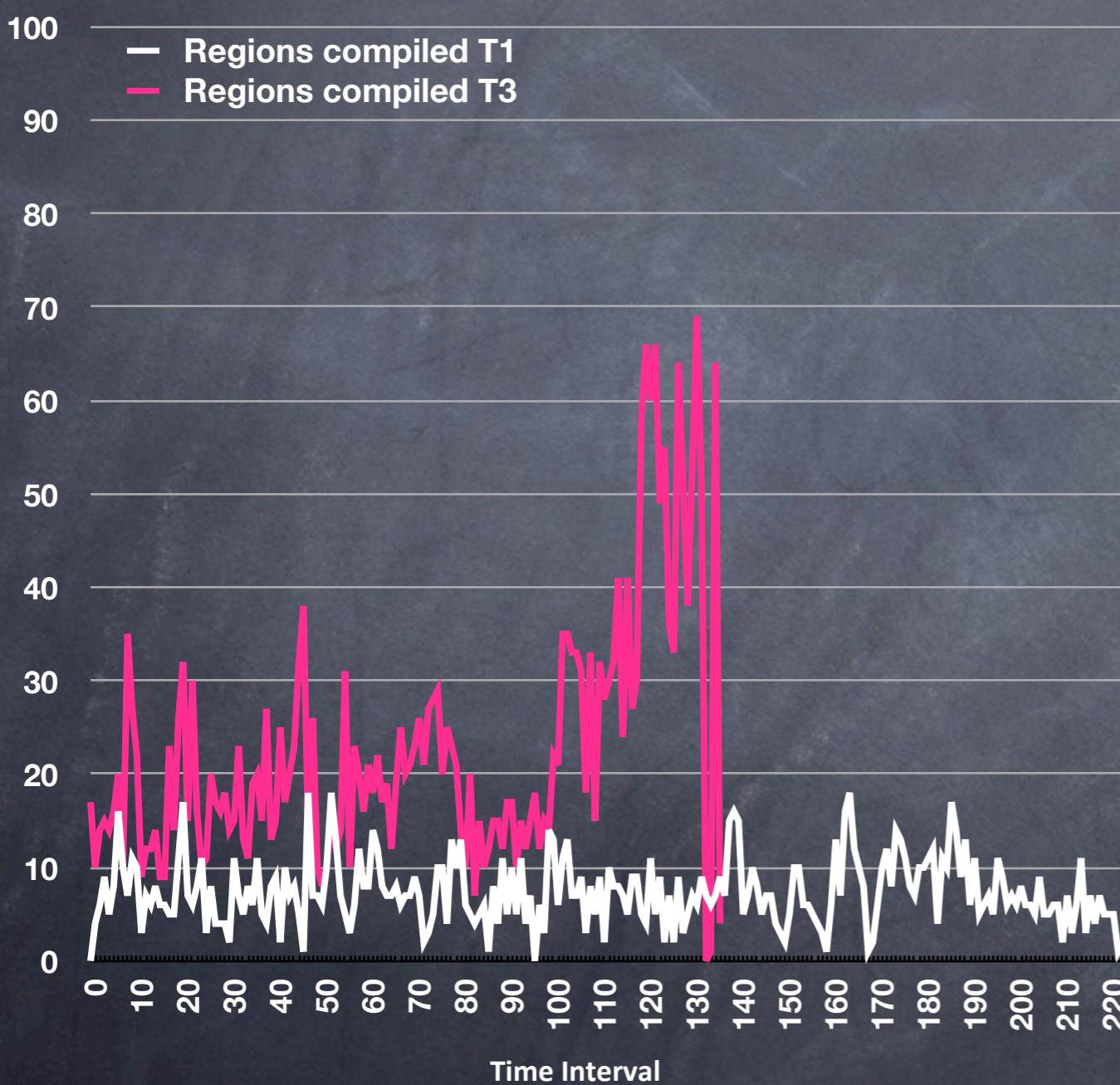
Effect of Concurrent and Parallel JIT Compilation on Throughput

403.gcc - Regions Compiled



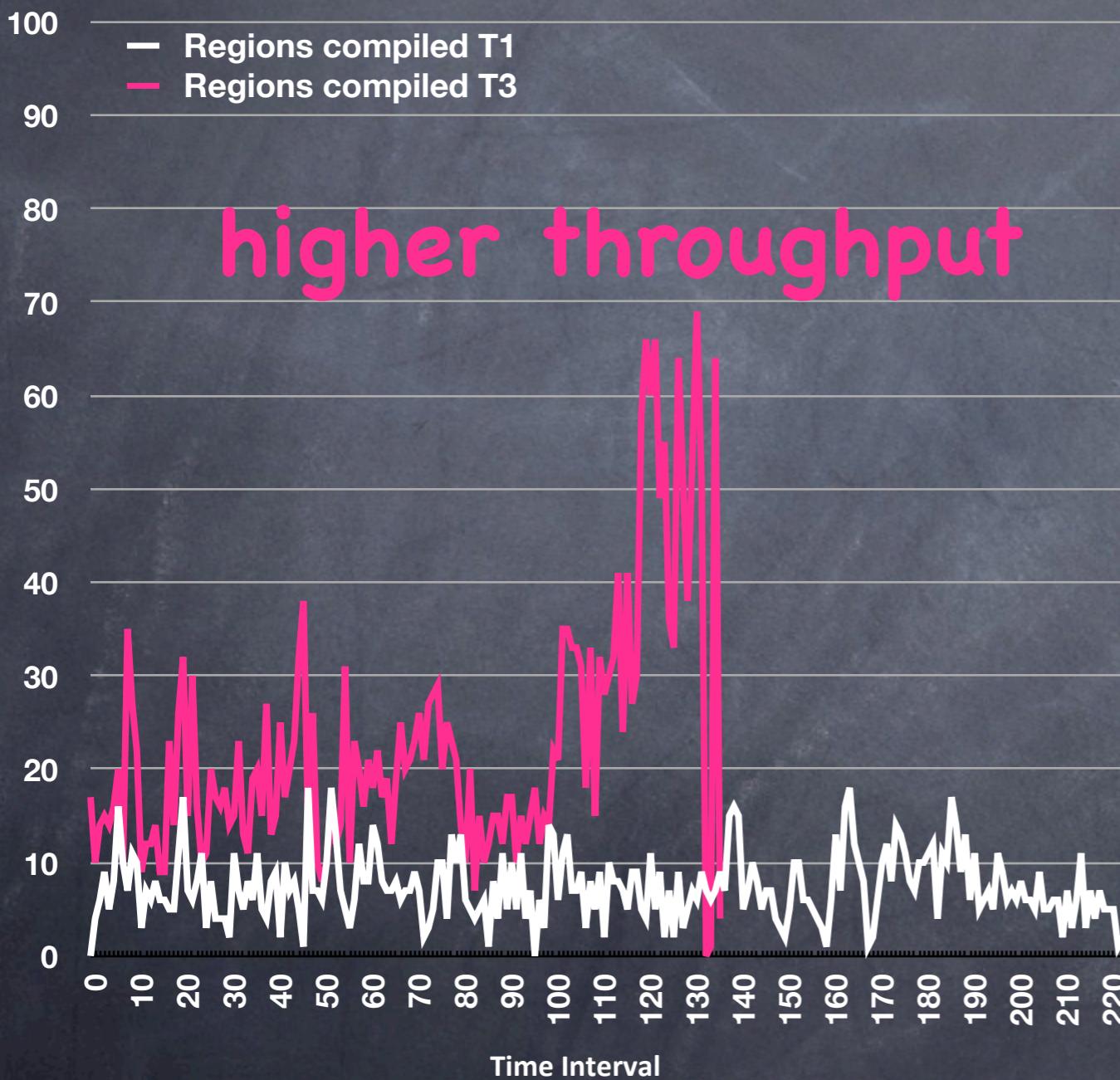
Effect of Concurrent and Parallel JIT Compilation on Throughput

403.gcc - Regions Compiled



Effect of Concurrent and Parallel JIT Compilation on Throughput

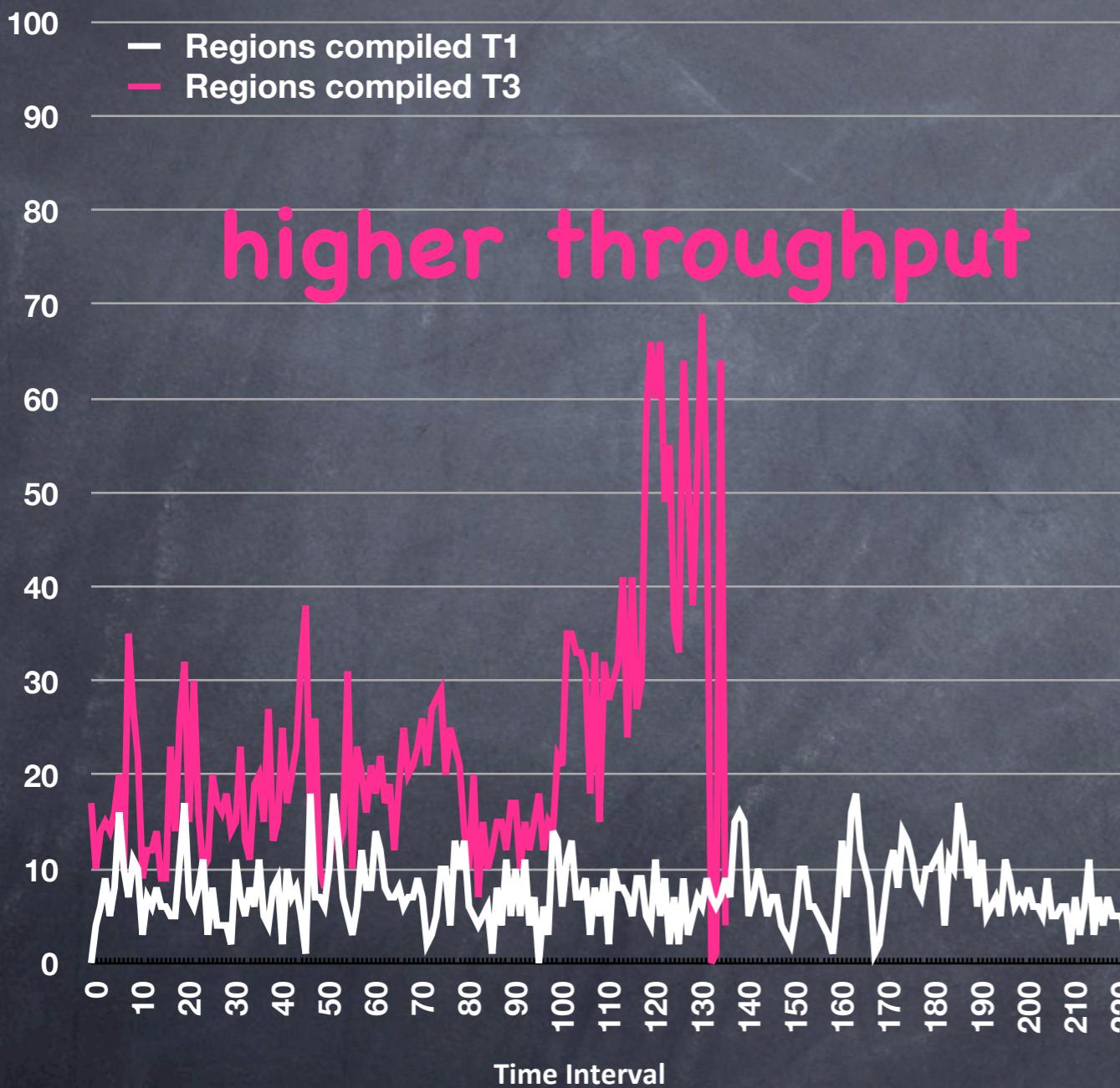
403.gcc - Regions Compiled



higher throughput

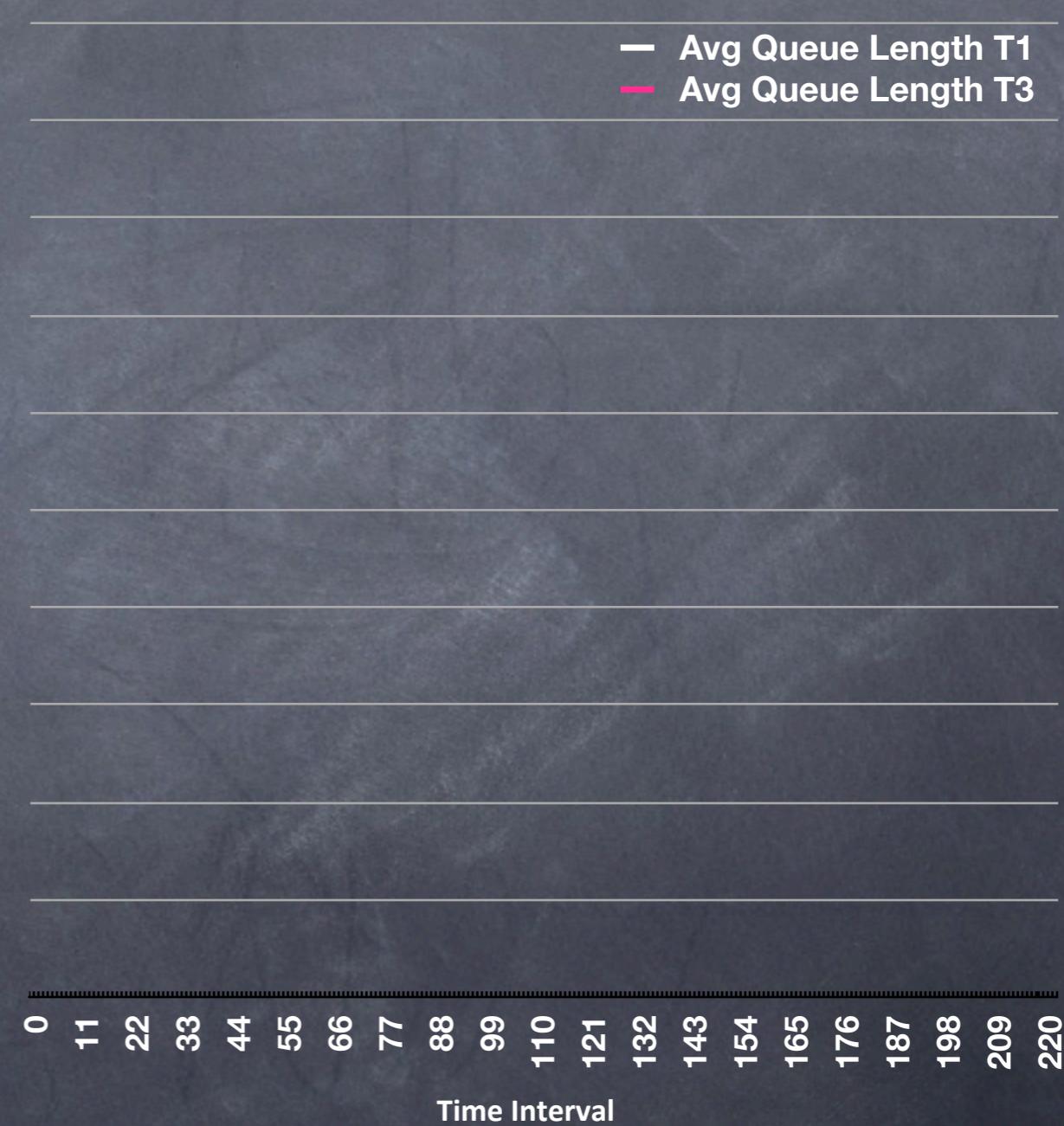
Effect of Concurrent and Parallel JIT Compilation on Throughput

403.gcc - Regions Compiled



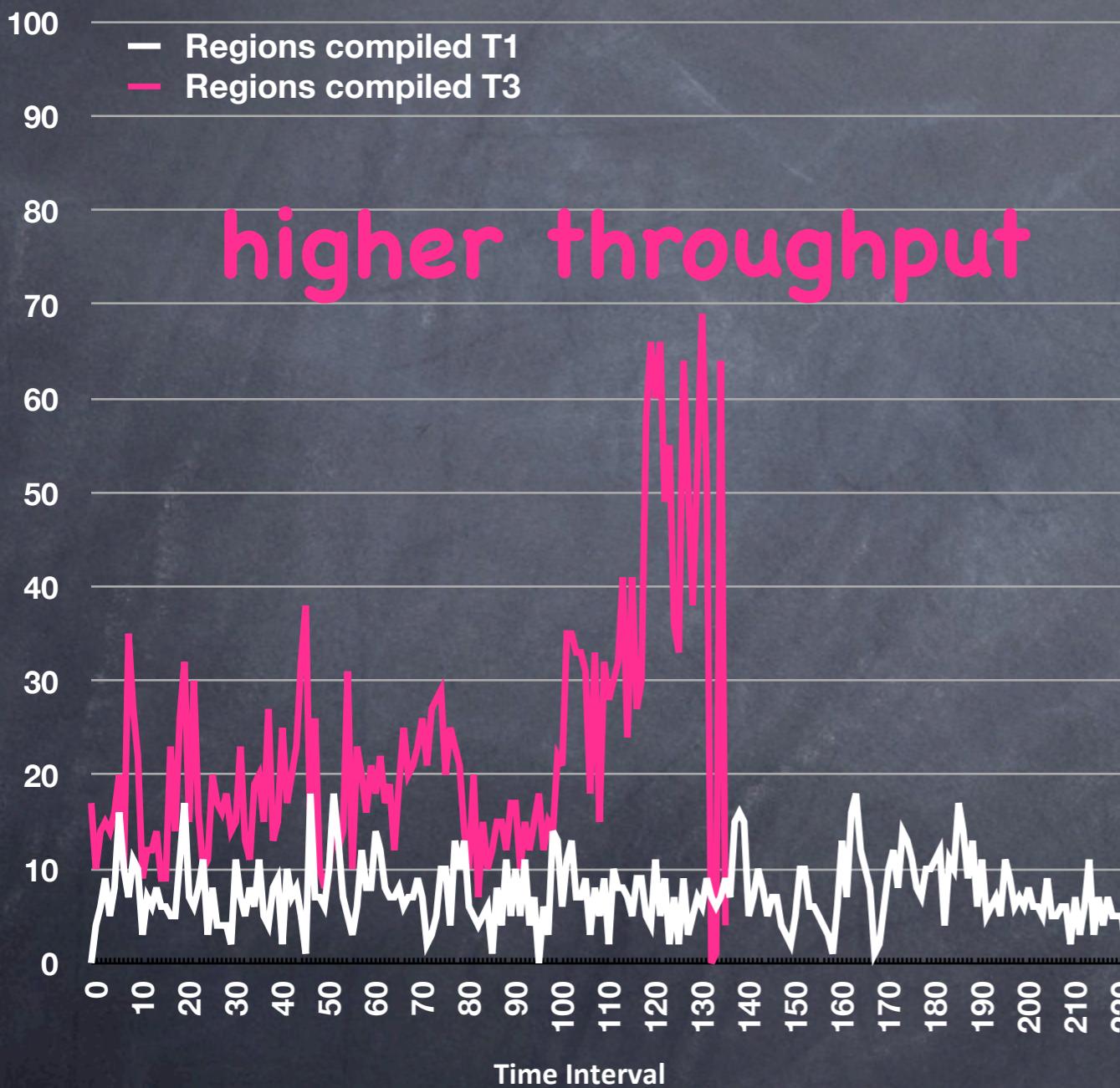
higher throughput

403.gcc - Average Queue Length



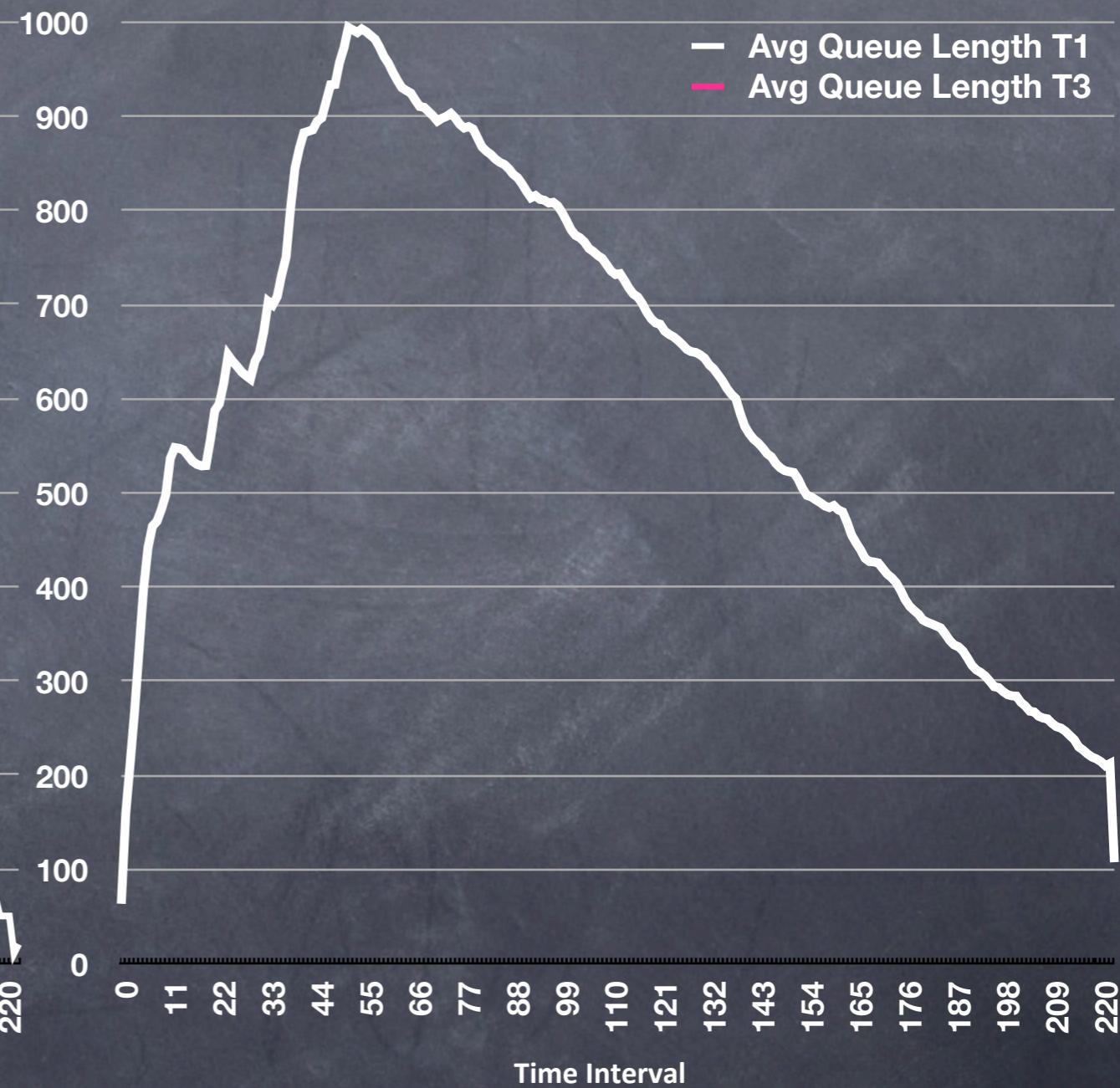
Effect of Concurrent and Parallel JIT Compilation on Throughput

403.gcc - Regions Compiled



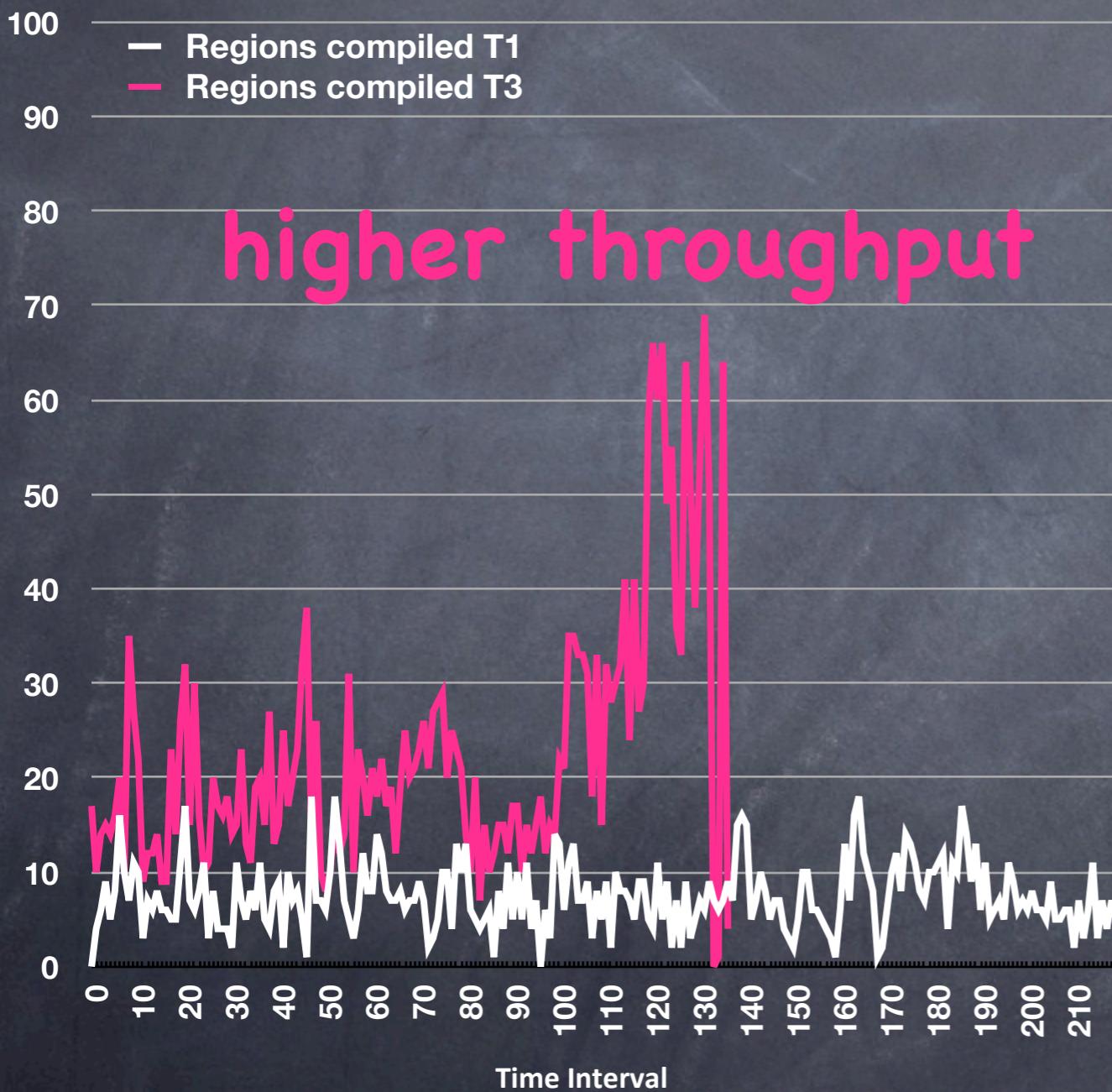
higher throughput

403.gcc - Average Queue Length



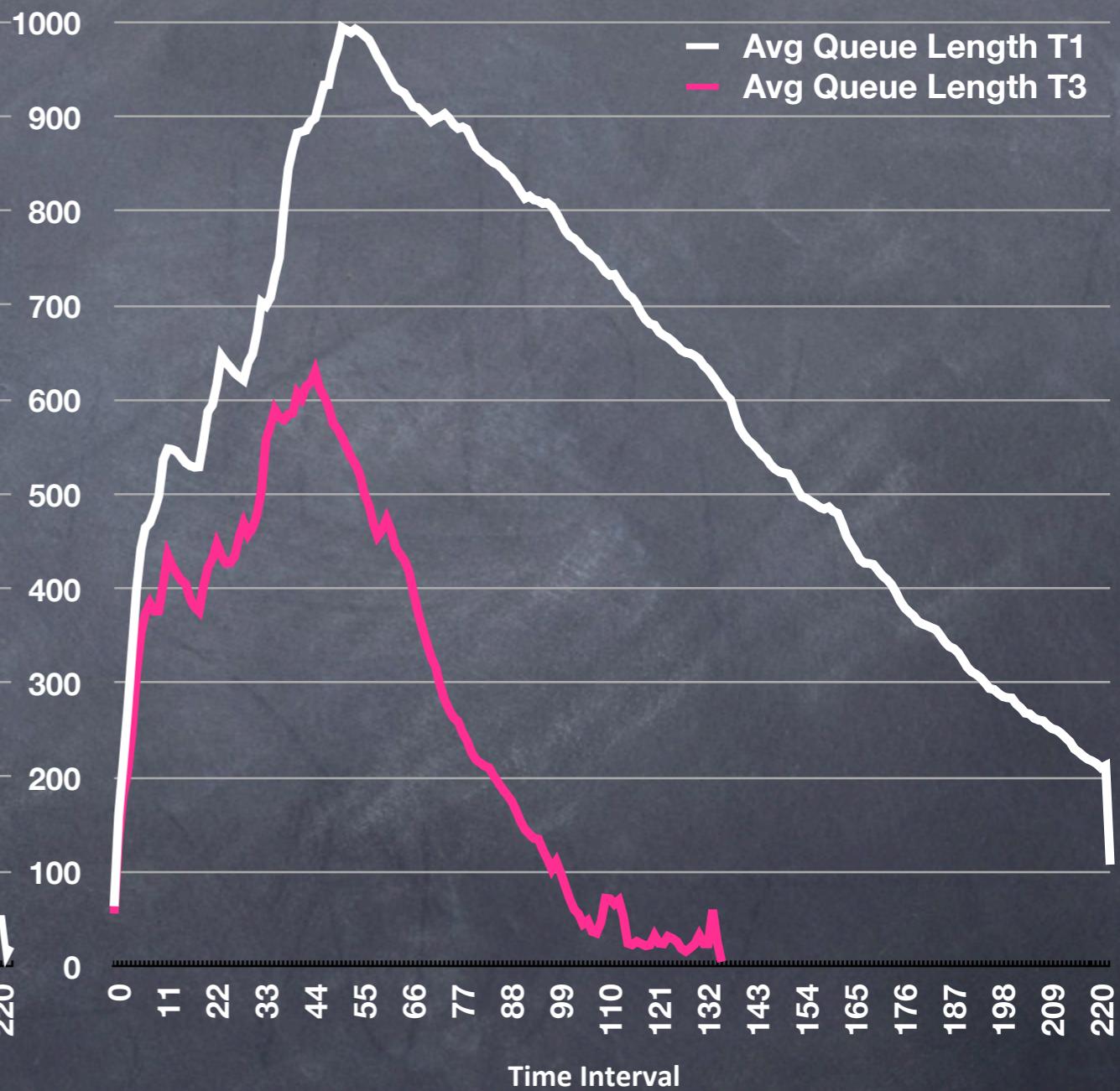
Effect of Concurrent and Parallel JIT Compilation on Throughput

403.gcc - Regions Compiled



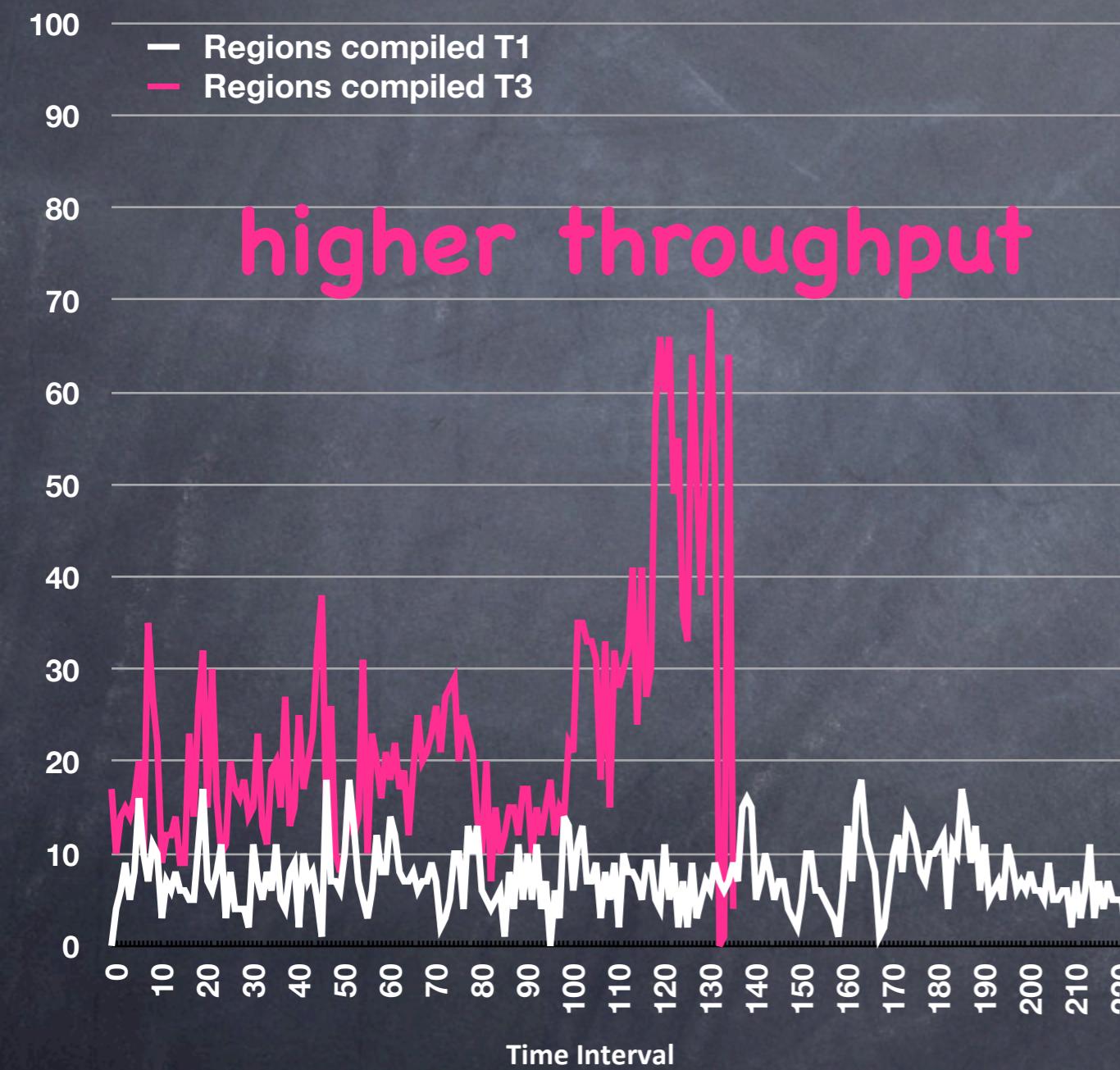
higher throughput

403.gcc - Average Queue Length



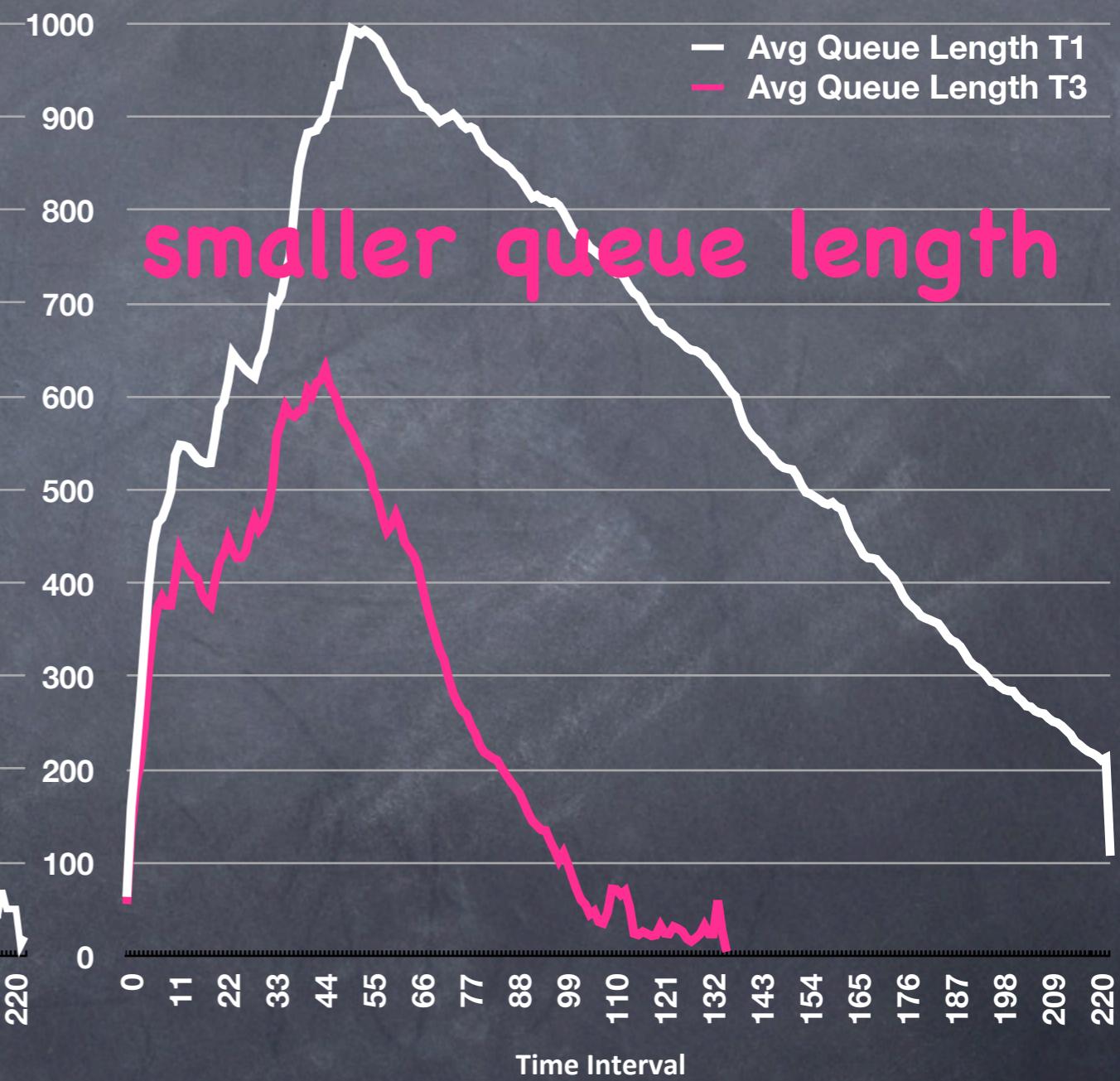
Effect of Concurrent and Parallel JIT Compilation on Throughput

403.gcc - Regions Compiled



higher throughput

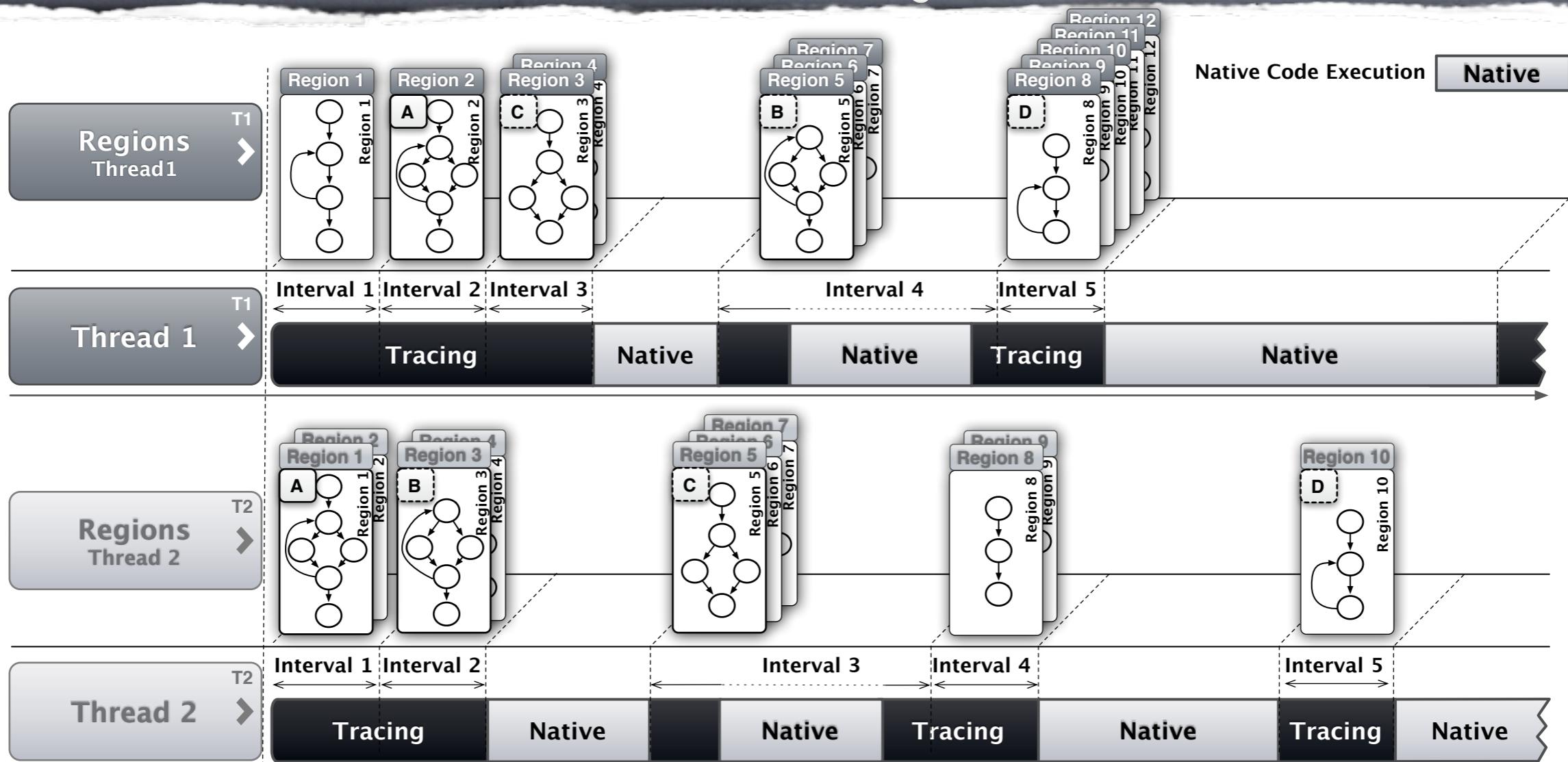
403.gcc - Average Queue Length



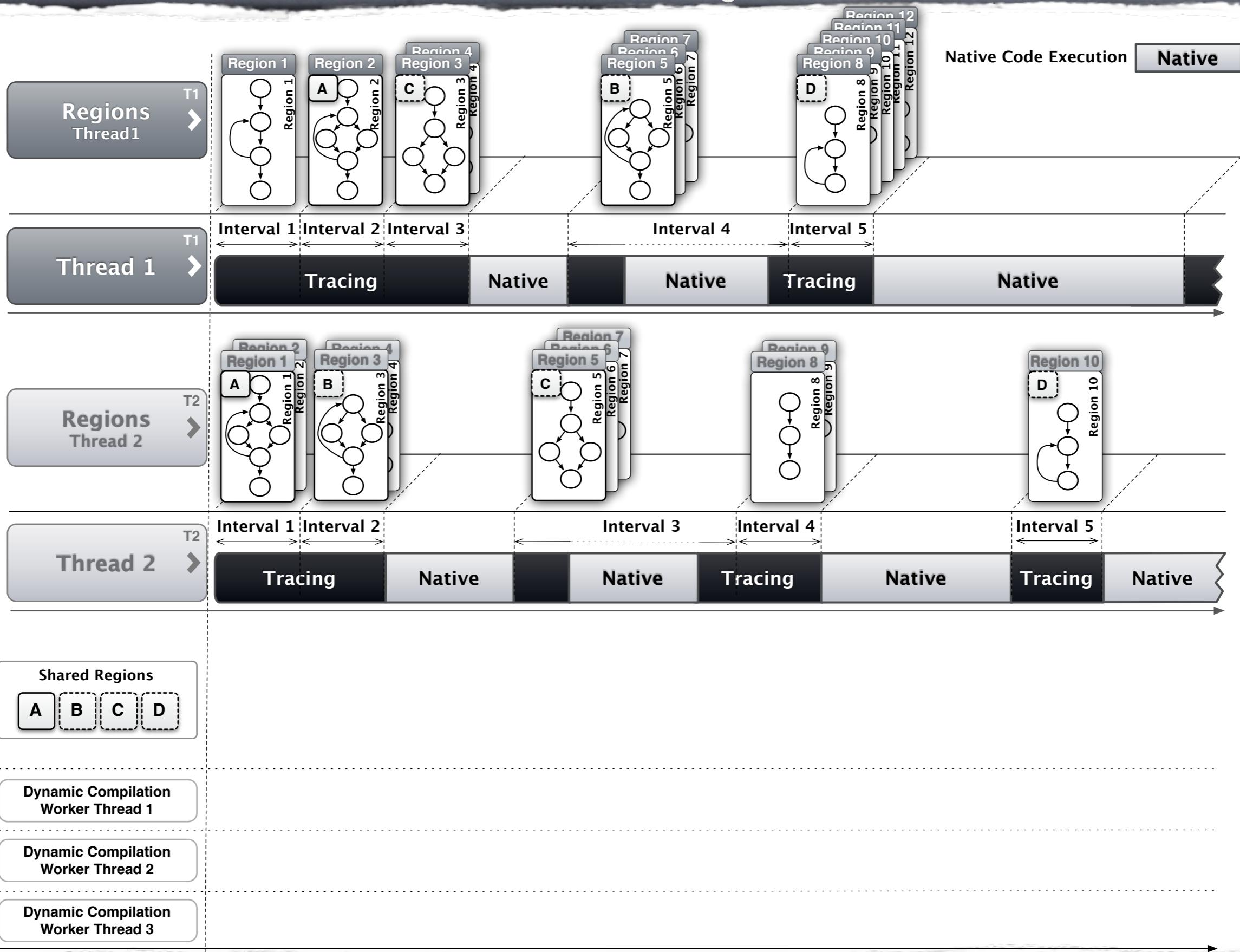
smaller queue length

Does this scale for
multi-threaded/core
applications?

Concurrent and Parallel JIT Compilation in Action (trace sharing)

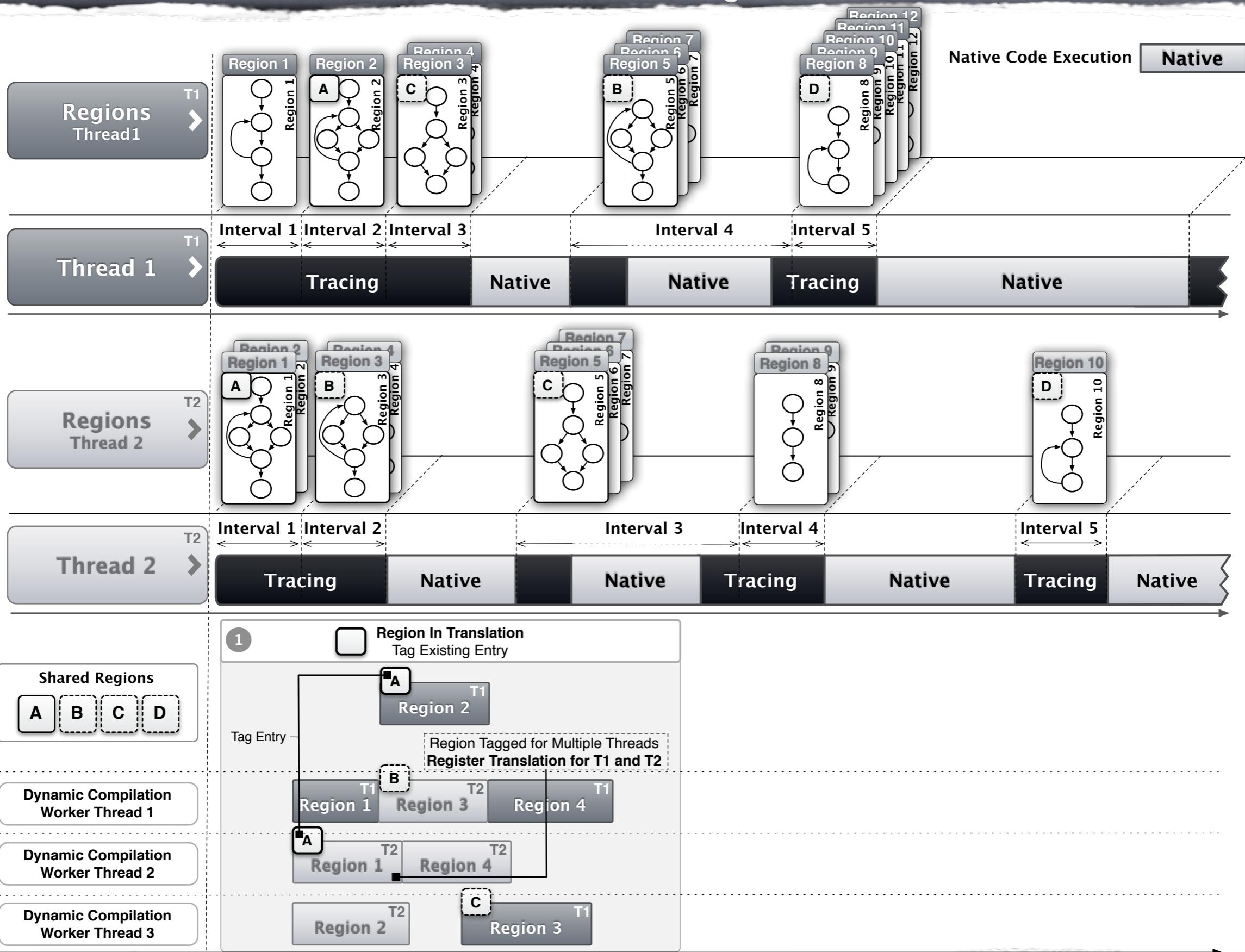


Concurrent and Parallel JIT Compilation in Action (trace sharing)



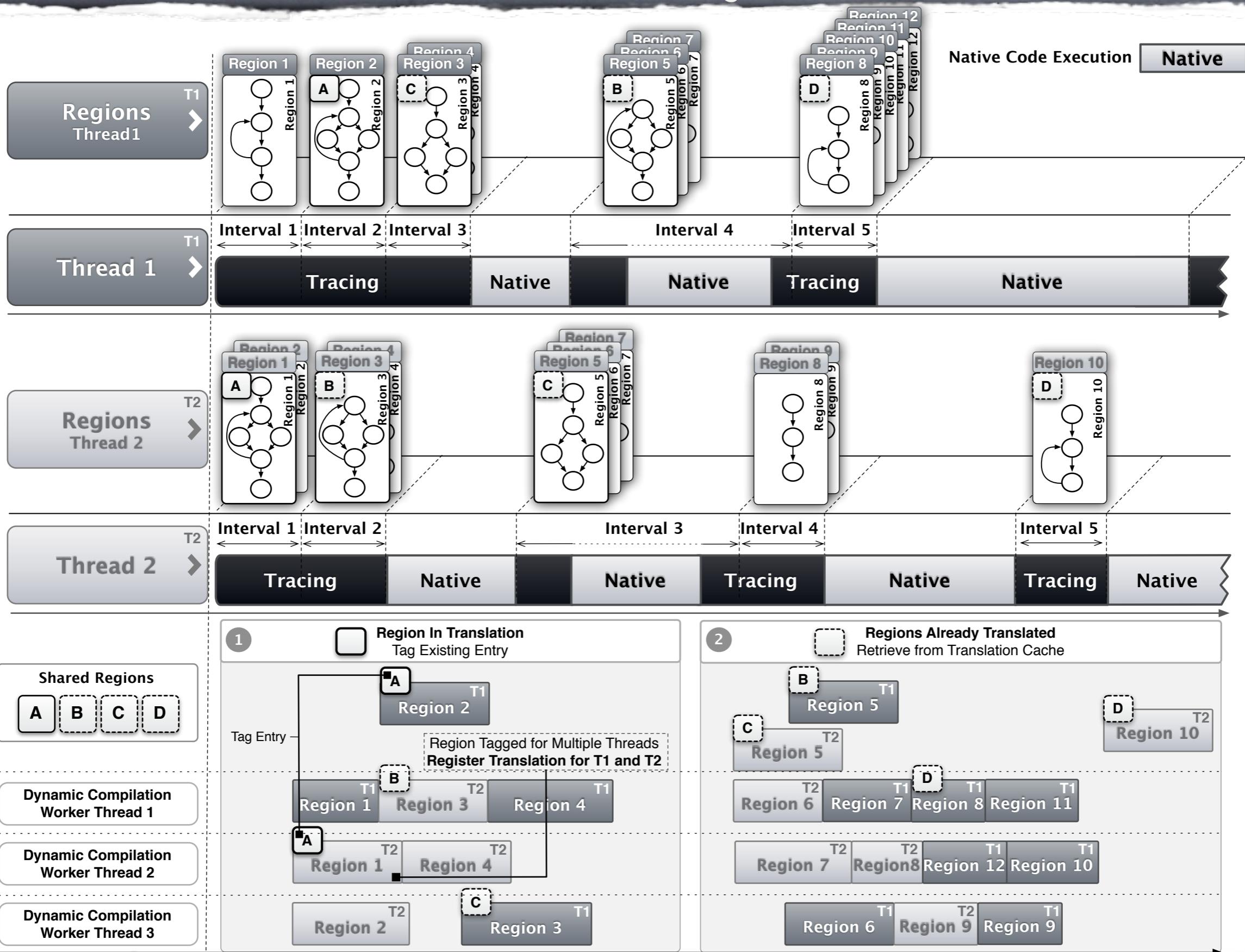
Concurrent and Parallel JIT Compilation in Action

(trace sharing)



Concurrent and Parallel JIT Compilation in Action

(trace sharing)



Conclusions

- ⦿ Novel interval based region code discovery scheme enables concurrent and parallel JIT compilation and is able to deliver:
 - ⦿ average reduction of execution time of 11.5% - and up to 51.9% across 60 industry standard benchmarks
- ⦿ we minimise JIT compilation overhead and effectively hide compilation latency by combining:
 - ⦿ light-weight interval based tracing
 - ⦿ dynamic work scheduling
 - ⦿ adaptive hotspot threshold selection
 - ⦿ concurrent and parallel JIT compilation

Demos

Demos

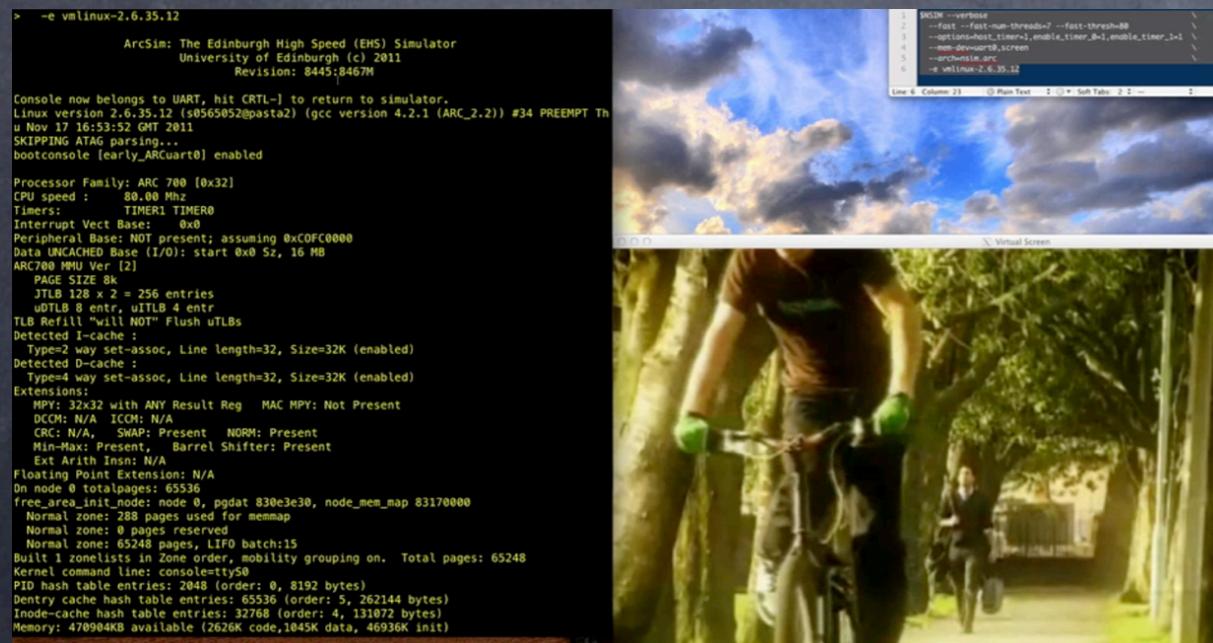
web▶m

Video Decoding
and Playback

Demos



Video Decoding and Playback



A screenshot of a video player window titled "Virtual Screen". Inside the window, a person is riding a bicycle on a path lined with trees under a blue sky with clouds. Above the video player, there is a terminal window showing command-line options for "EWSIM" followed by a list of kernel parameters for "vmlinux-2.6.35.12".

```
> -e vmlinux-2.6.35.12
  ArcSim: The Edinburgh High Speed (EHS) Simulator
  University of Edinburgh (c) 2011
  Revision: 8445:8467M

Console now belongs to UART, hit CTRL-[ to return to simulator.
Linux version 2.6.35.12 (s0565052@oasta2) (gcc version 4.2.1 (ARC_2.2)) #34 PREEMPT Th
u Nov 17 16:53:52 GMT 2011
SKIPPING ATAG parsing...
bootconsole [early_ARCuart0] enabled

Processor Family: ARC 700 [0x32]
CPU speed : 80.00 Mhz
Timers: TIMER1 TIMER0
Interrupt Vect Base: 0x0
Peripheral Base: NOT present; assuming 0xCOFC0000
Data UNCACHED Base (I/O): start 0x0 Sz, 16 MB
ARC700 MMU Ver [2]
PAGE SIZE 8k
JTLB 128 x 2 = 256 entries
uDTLB 8 entr, uITLB 4 entr
TLB Refill "will NOT" Flush uTLBs
Detected I-cache :
  Type2 way set-assoc, Line length=32, Size=32K (enabled)
Detected D-cache :
  Type4 way set-assoc, Line length=32, Size=32K (enabled)
Extensions:
  MPY: 32x32 with ANY Result Reg  MAC MPY: Not Present
  DCOM: N/A  ICOM: N/A
  CRC: N/A, SWAP: Present  NORM: Present
  Min-Max: Present, Barrel Shifter: Present
  Ext Arith Insn: N/A
Floating Point Extension: N/A
On node 0 totalpages: 65536
free_area_init_node: node 0, pgdat 830e3e30, node_mem_map 83170000
  Normal zone: 288 pages used for memmap
  Normal zone: 0 pages reserved
  Normal zone: 65248 pages, LIFO batch:15
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 65248
Kernel command line: console=ttyS0
PID hash table entries: 2048 (order: 0, 8192 bytes)
Dentry cache hash table entries: 65536 (order: 5, 262144 bytes)
Inode-cache hash table entries: 32768 (order: 4, 131072 bytes)
Memory: 470904KB available (2626K code,1045K data, 46936K init)
```

Full System OS Simulation

Thank You

State-of-the-Art Dynamic Compilation Strategies

Interp Interpretation

Profile Interpretation with Profiling

Compile Dynamic Compilation

Native Native Code Execution

Sequential Dynamic Compilation

Main Thread

Time →



State-of-the-Art Dynamic Compilation Strategies

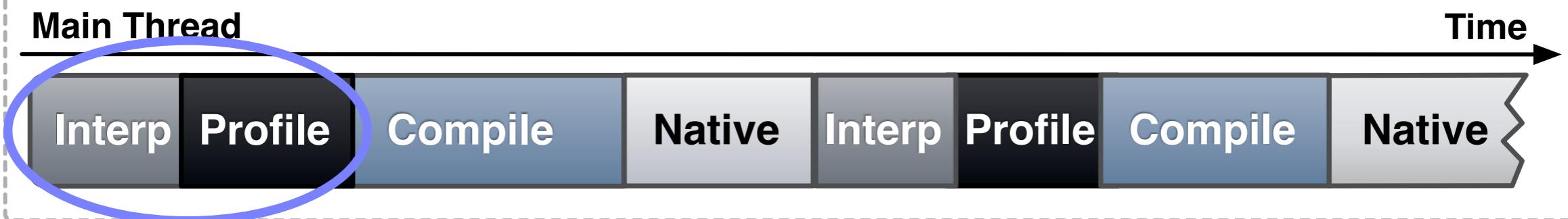
Interp Interpretation

Profile Interpretation with Profiling

Compile Dynamic Compilation

Native Native Code Execution

Sequential Dynamic Compilation



Code Discovery

State-of-the-Art Dynamic Compilation Strategies

Interp Interpretation

Profile Interpretation with Profiling

Compile Dynamic Compilation

Native Native Code Execution

Sequential Dynamic Compilation

Main Thread

Time →



Dynamic Compilation

State-of-the-Art Dynamic Compilation Strategies

Interp Interpretation

Profile Interpretation with Profiling

Compile Dynamic Compilation

Native Native Code Execution

Sequential Dynamic Compilation

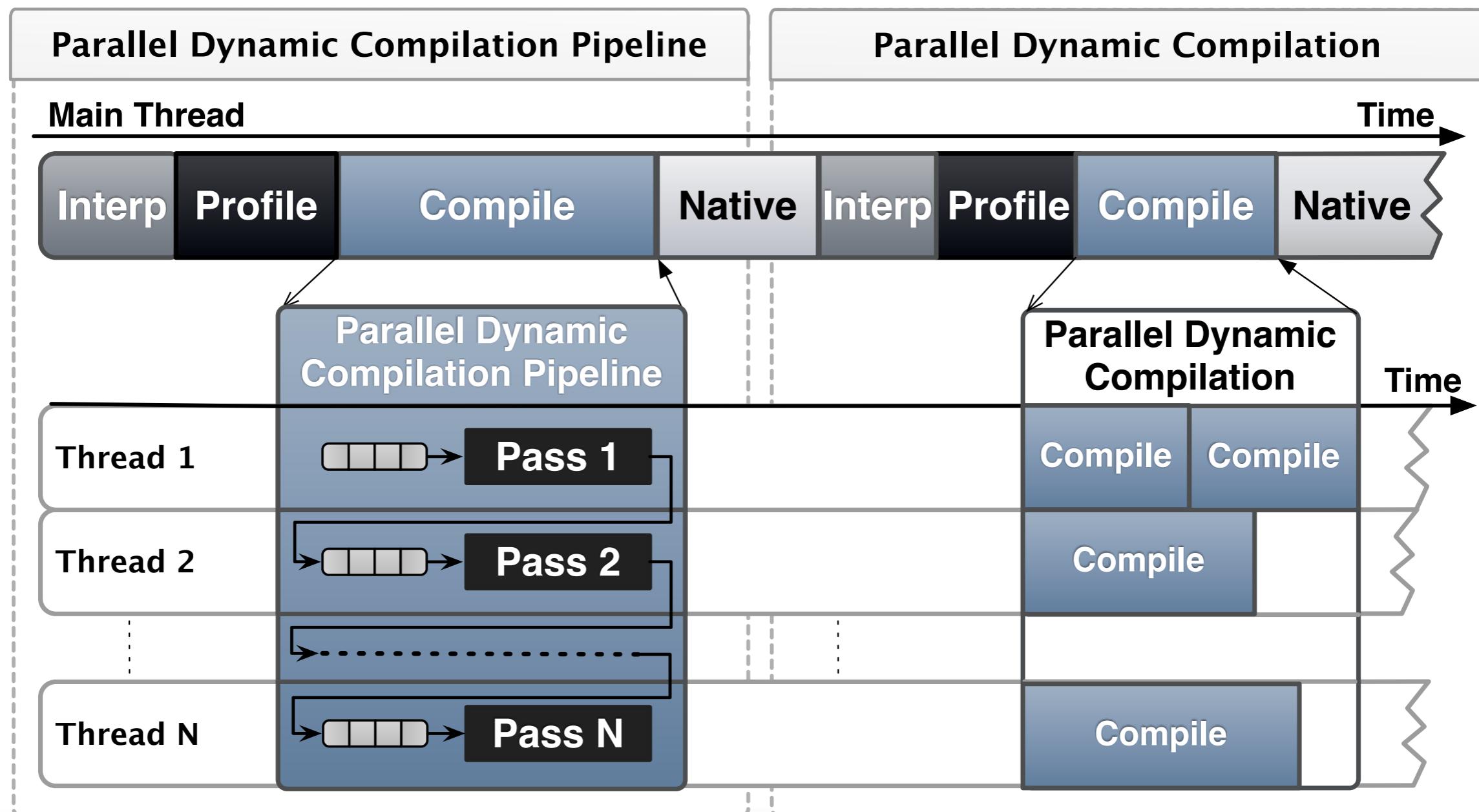
Main Thread

Time

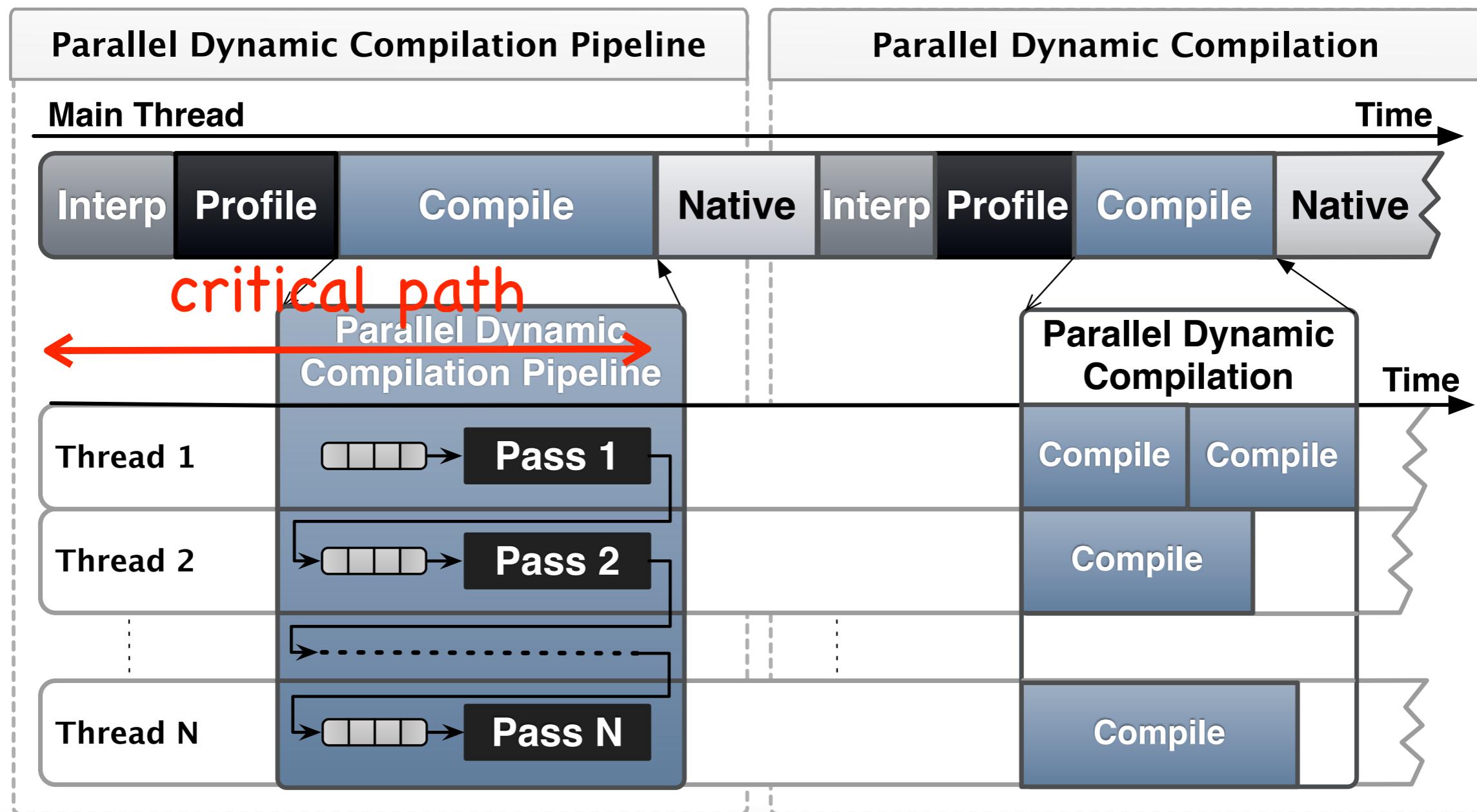


critical path

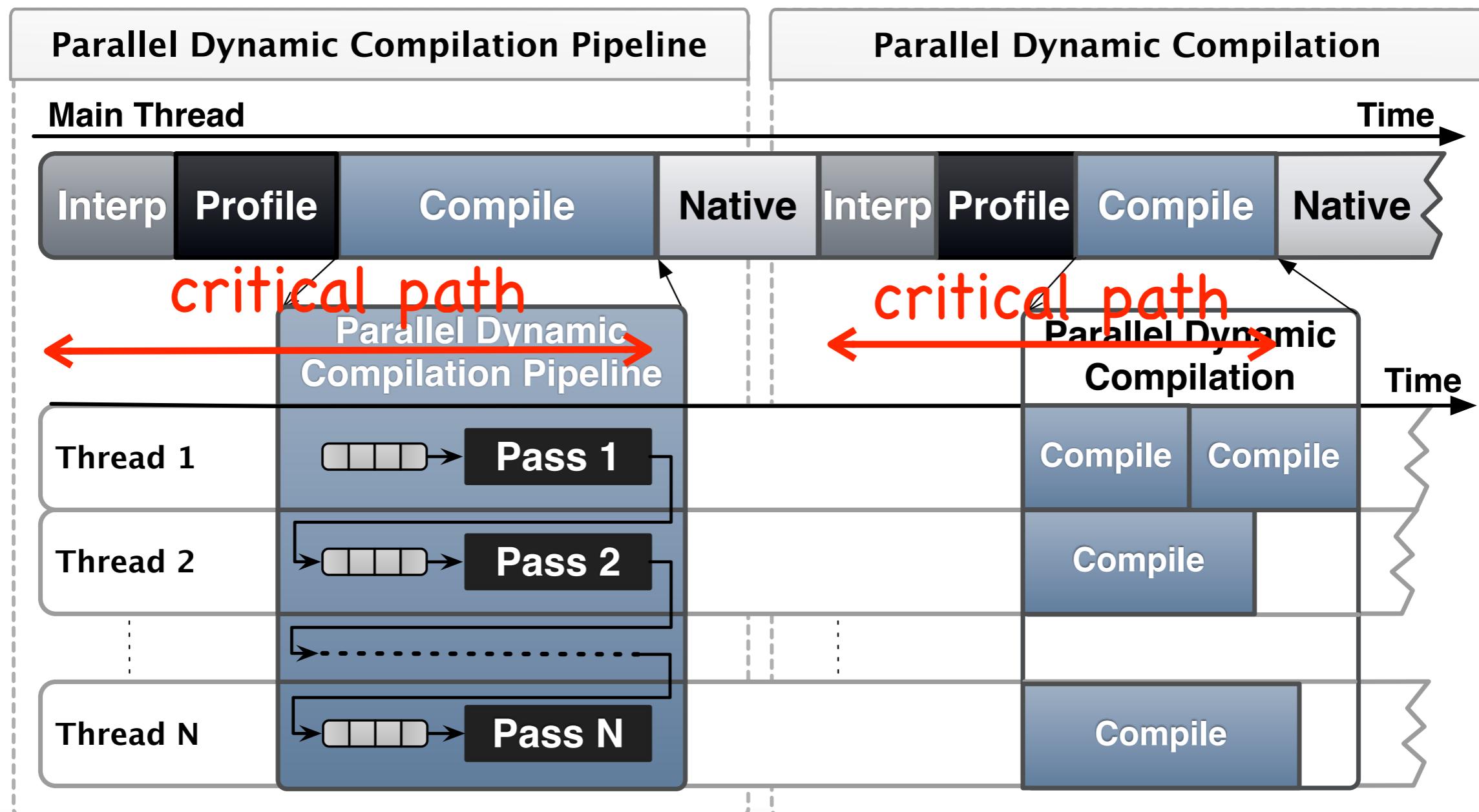
State-of-the-Art Dynamic Compilation Strategies



State-of-the-Art Dynamic Compilation Strategies



State-of-the-Art Dynamic Compilation Strategies



State-of-the-Art Dynamic Compilation Strategies

Interp Interpretation

Profile Interpretation with Profiling

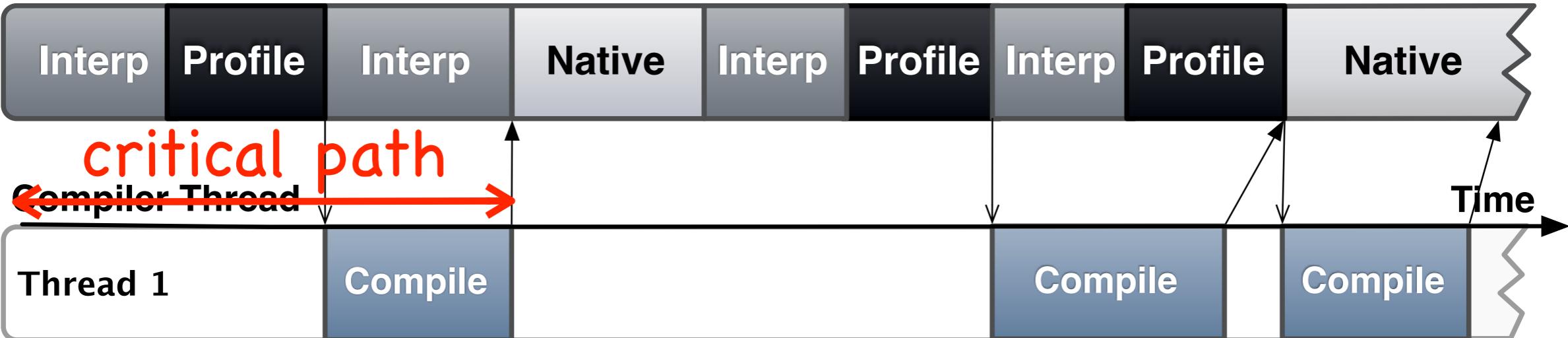
Compile Dynamic Compilation

Native Native Code Execution

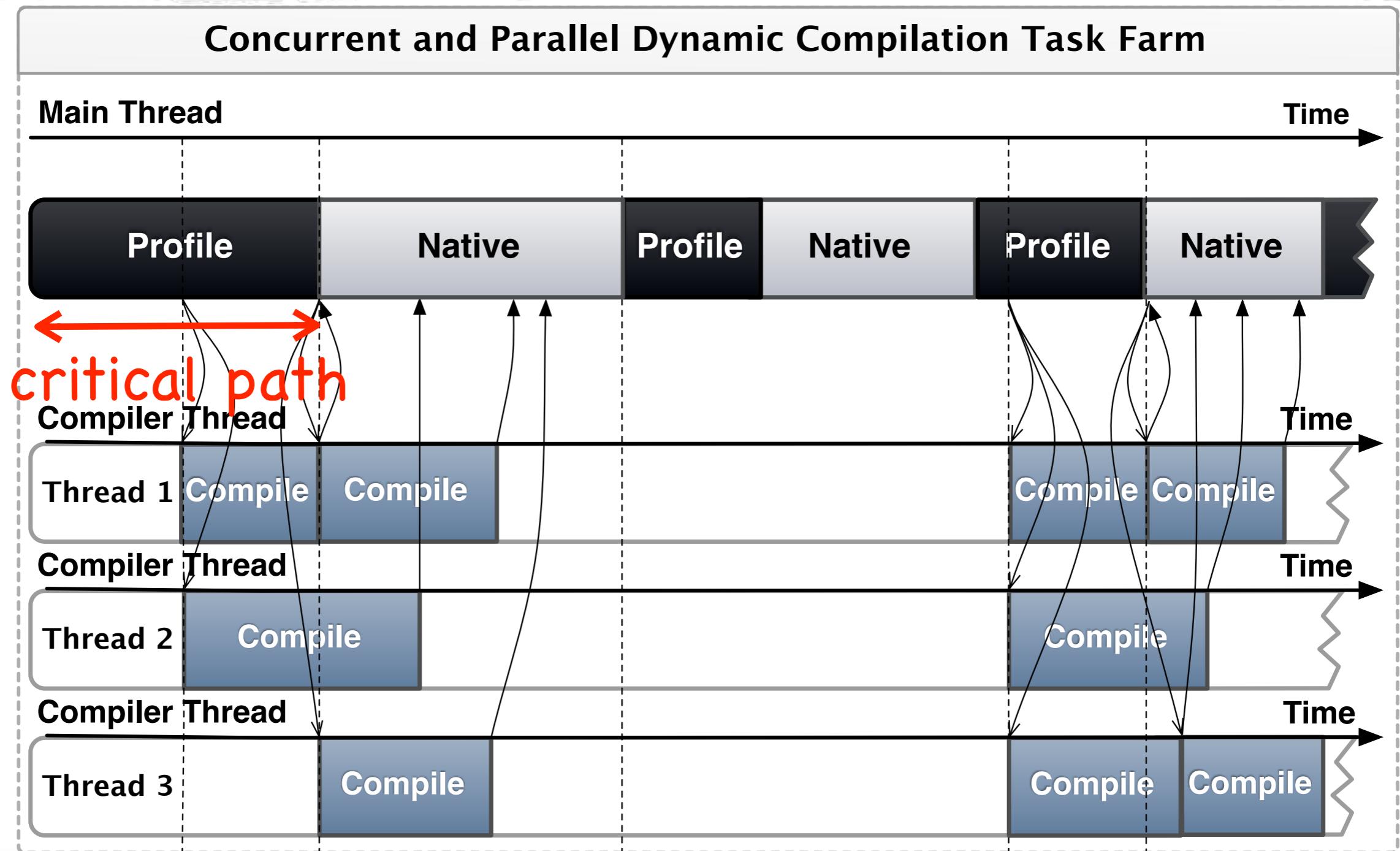
Concurrent Dynamic Compilation

Main Thread

Time



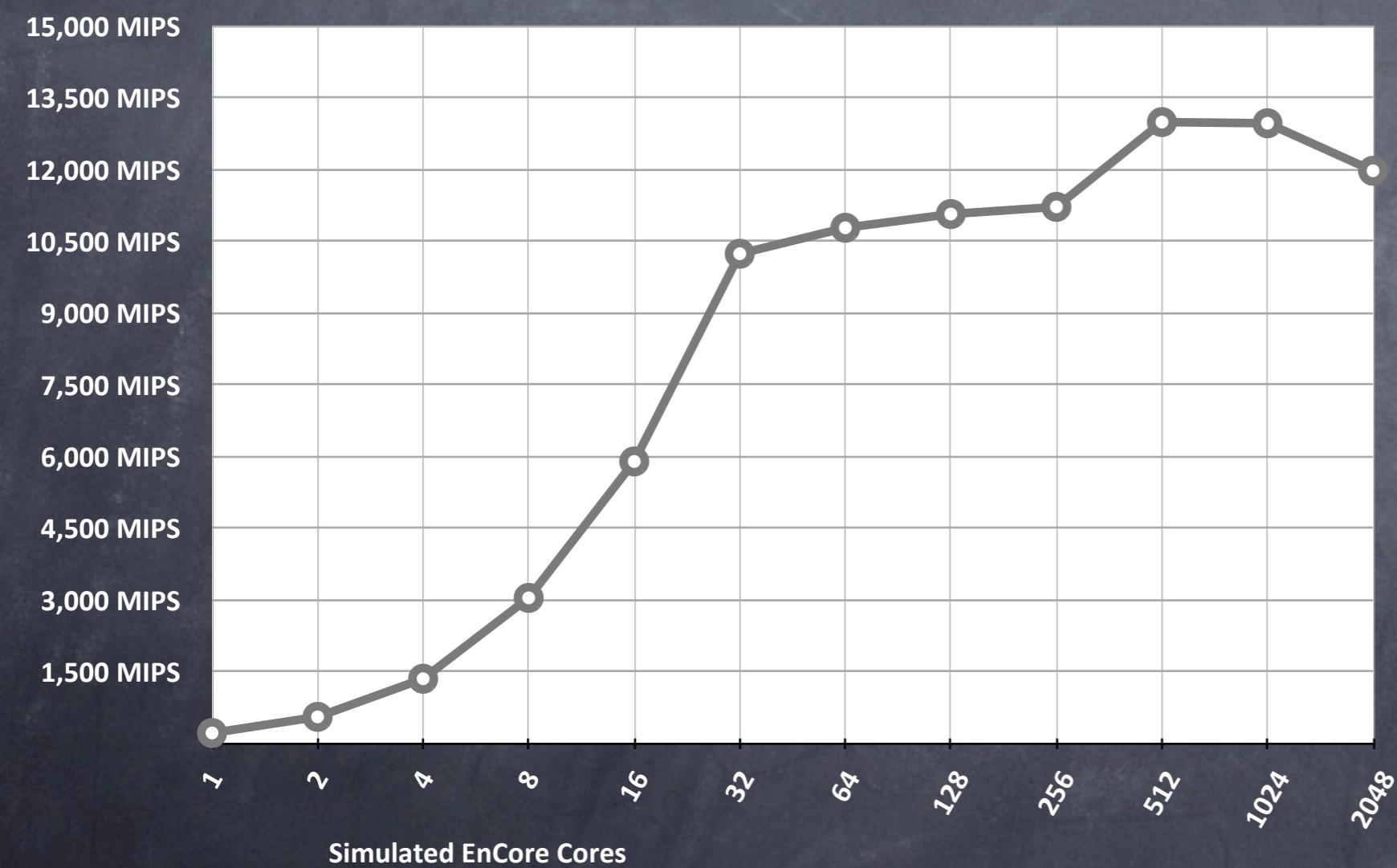
Concurrent and Parallel Dynamic Compilation



How does this perform for multi-threaded applications?

Simulator Scalability

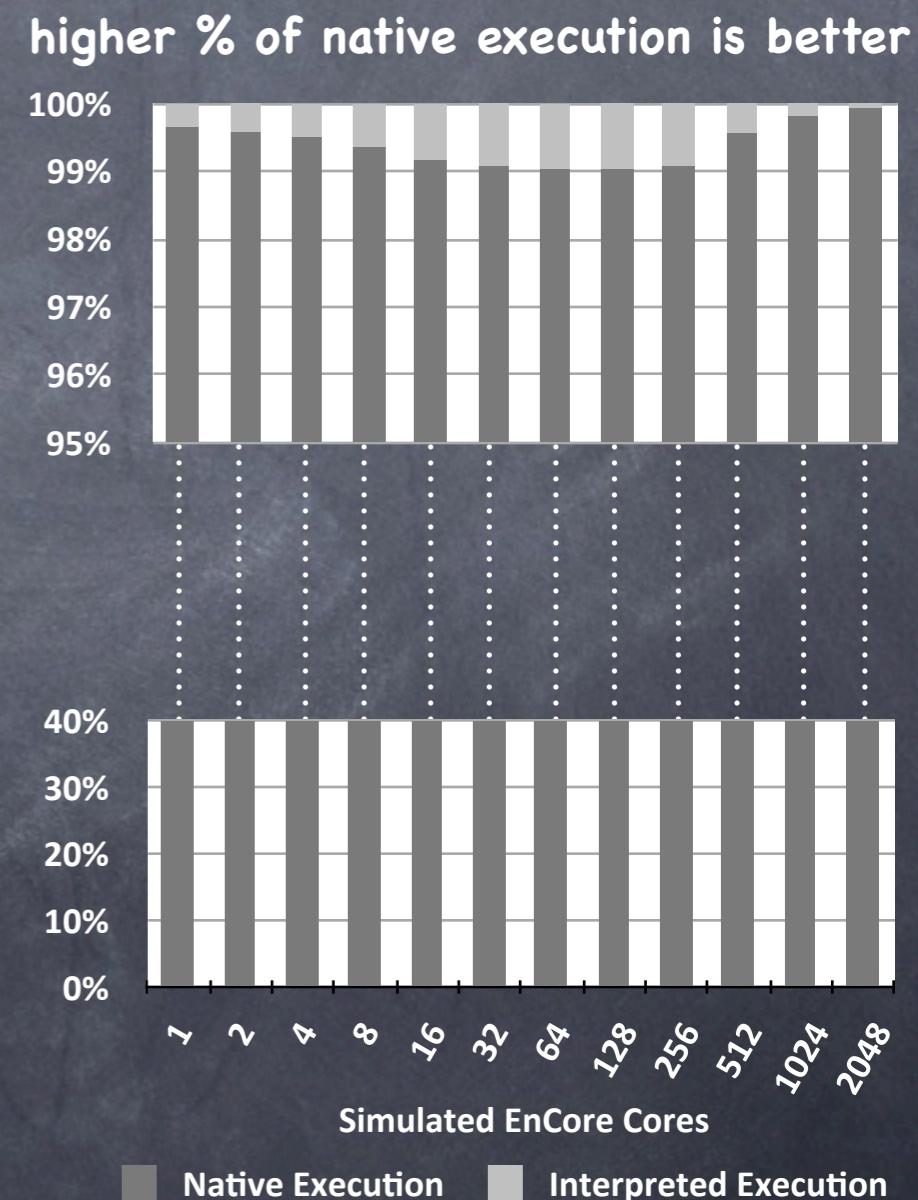
Simulation Rate



LU - Splash 2 Benchmark

Measured on a 32-core machine

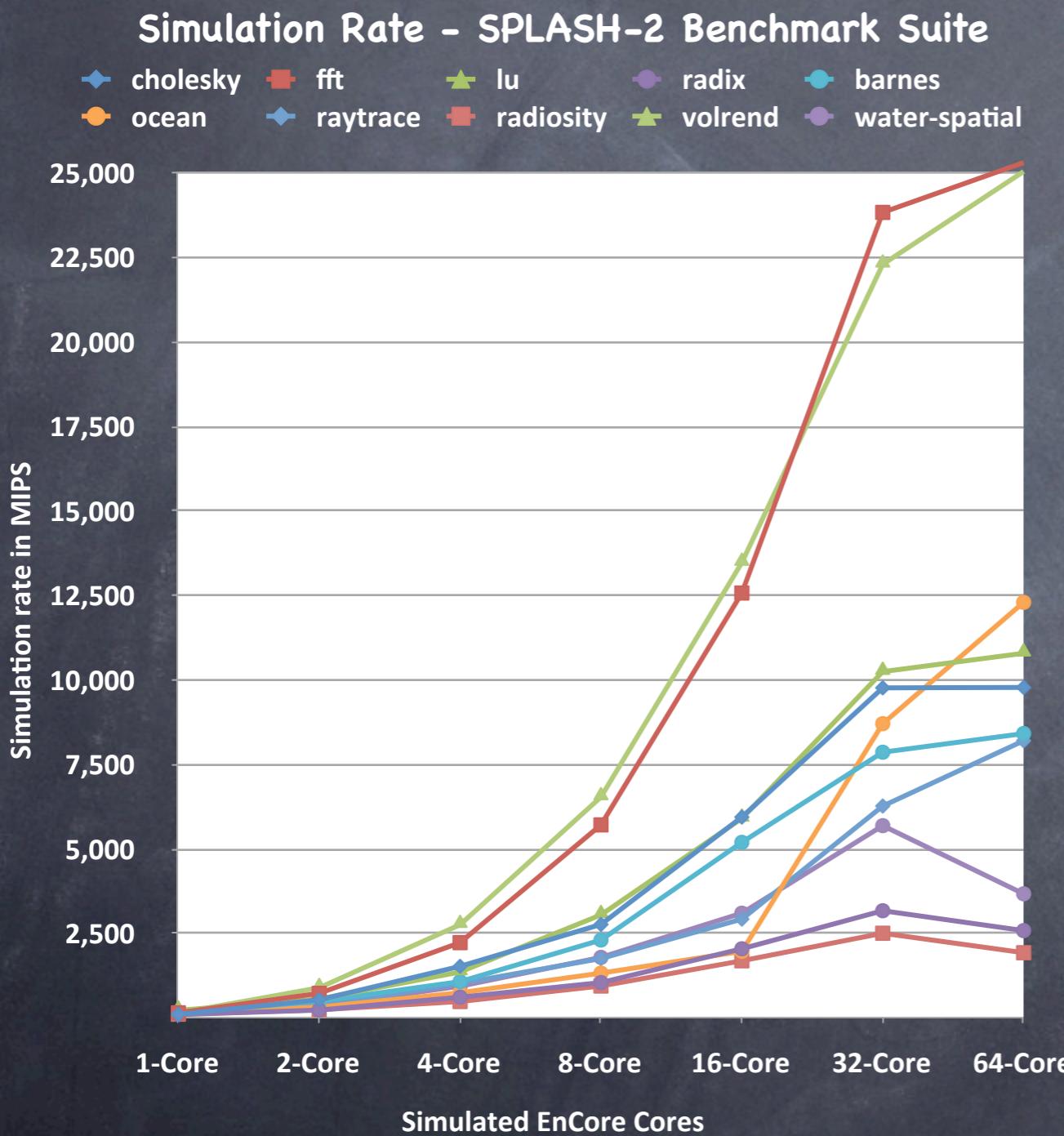
Interpreted vs. Native Execution



Native Execution Interpreted Execution

How does this perform for
multi-threaded applications?

How does this perform for multi-threaded applications?



How does this perform for multi-threaded applications?

