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

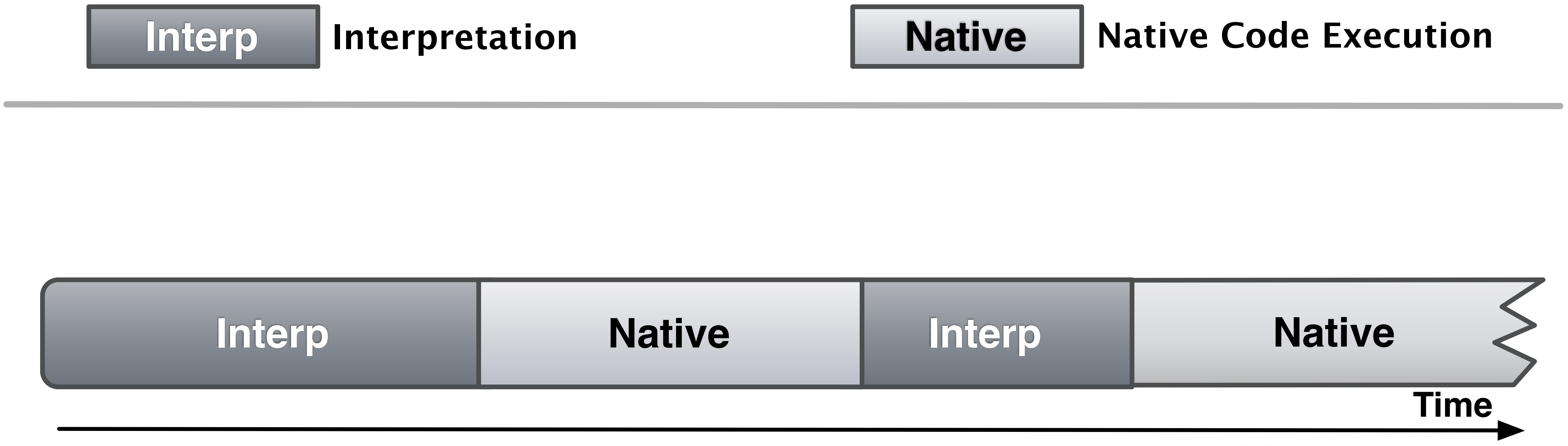
The University of Edinburgh

SYNOPSYS®

Predictable Success

# Dynamic Compilation

## What do we want to improve?



Interp Interpretation

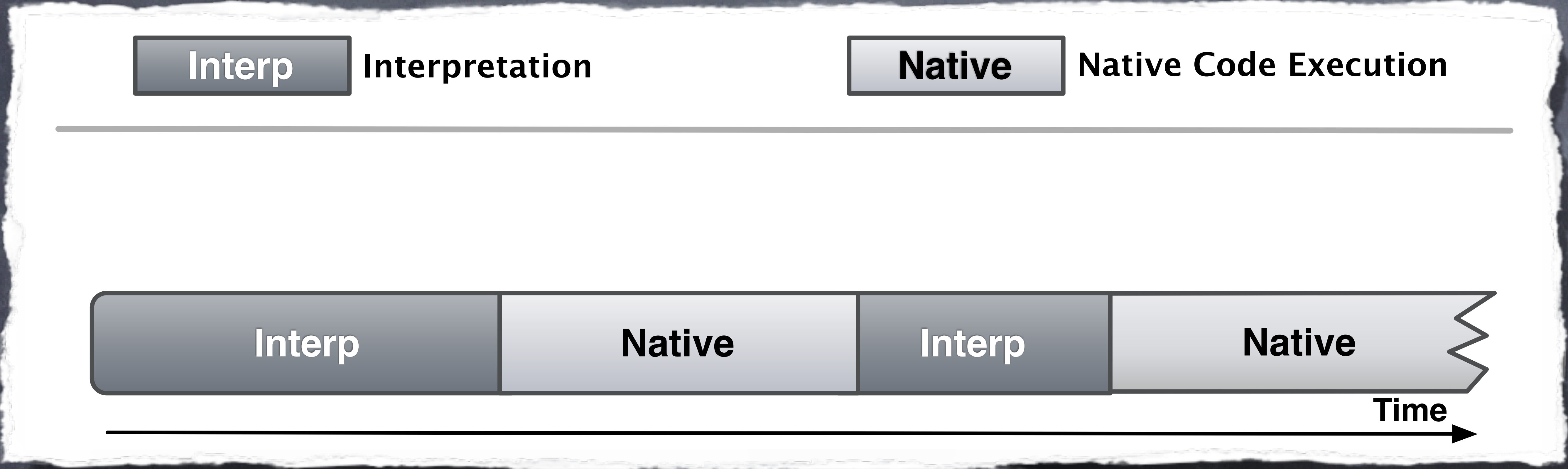
Native Native Code Execution

Interp | Native | Interp | Native

Time

# Dynamic Compilation

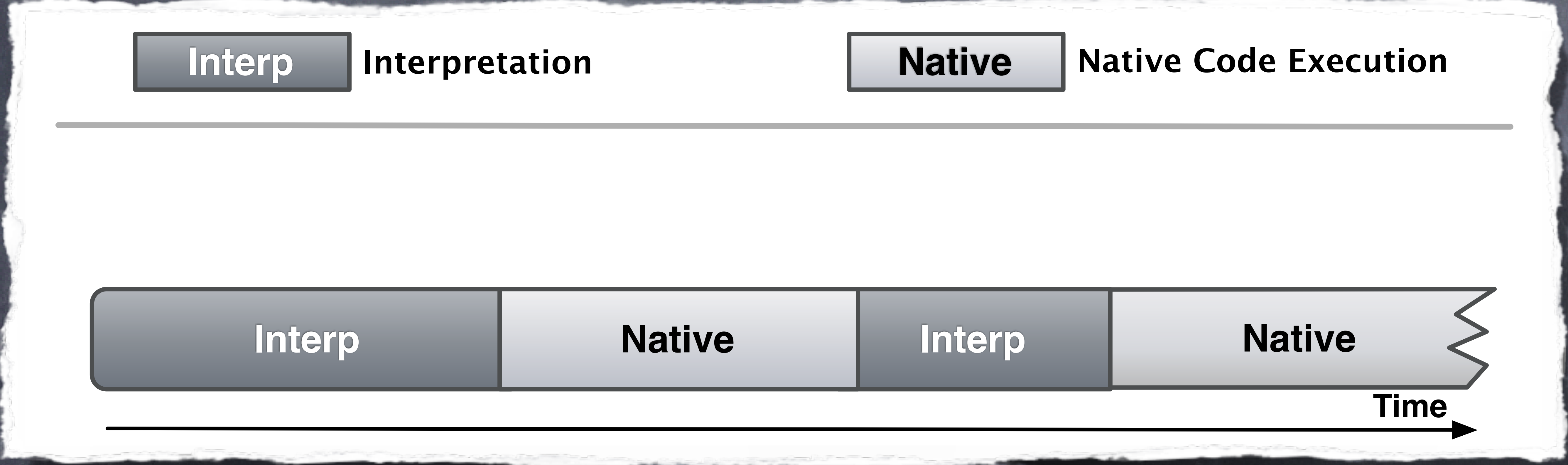
## What do we want to improve?



- initially code is interpreted

# Dynamic Compilation

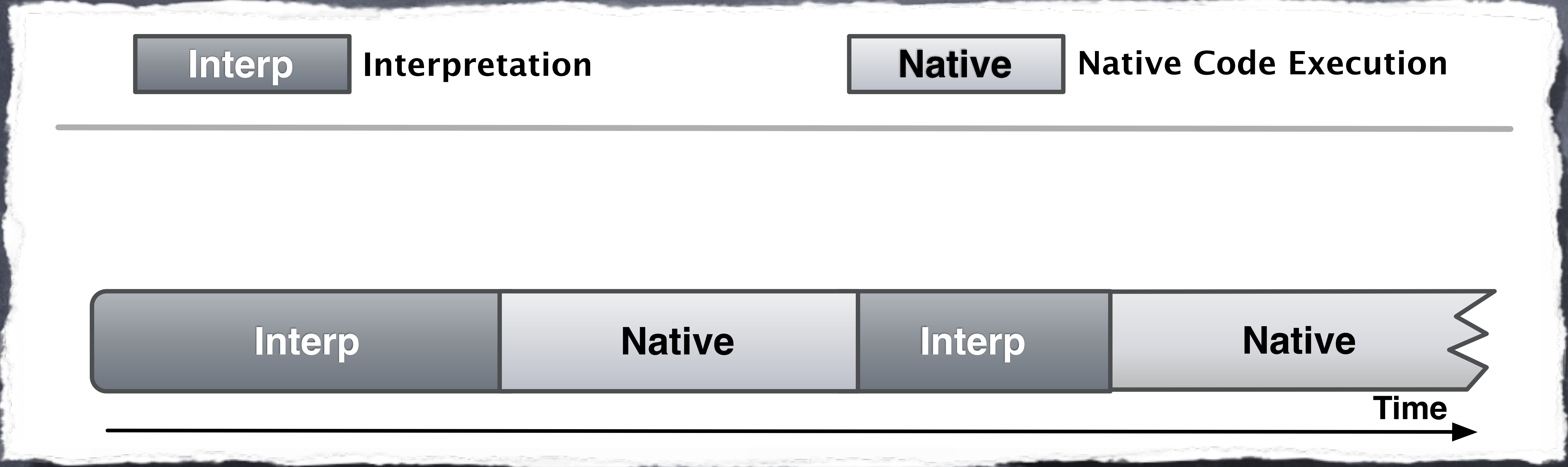
## What do we want to improve?



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

# Dynamic Compilation

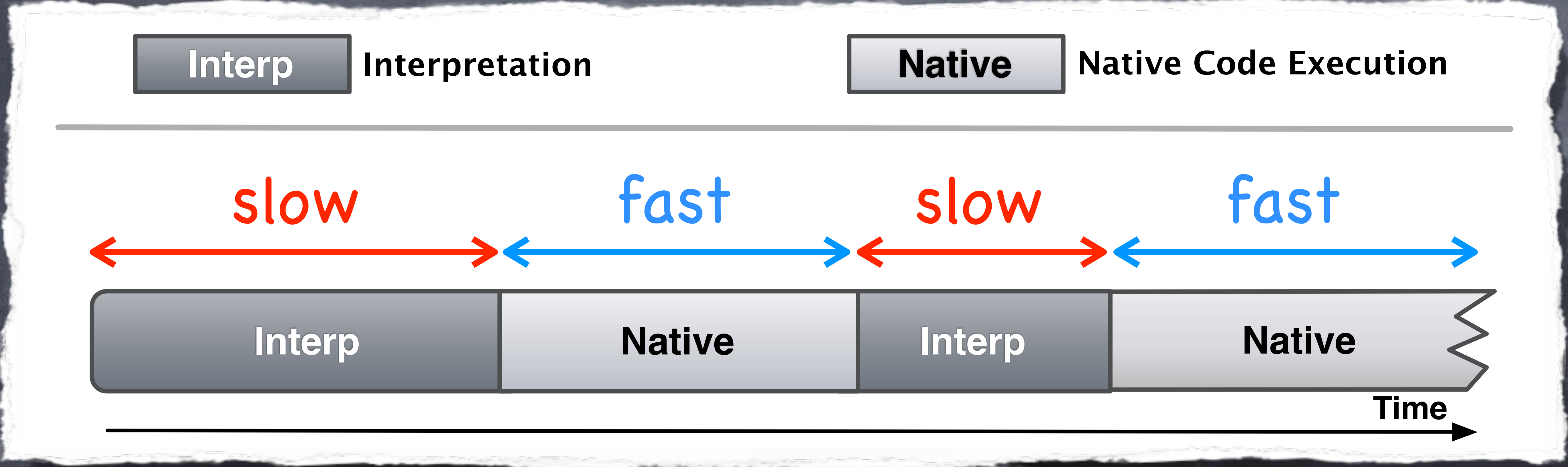
## What do we want to improve?



- 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

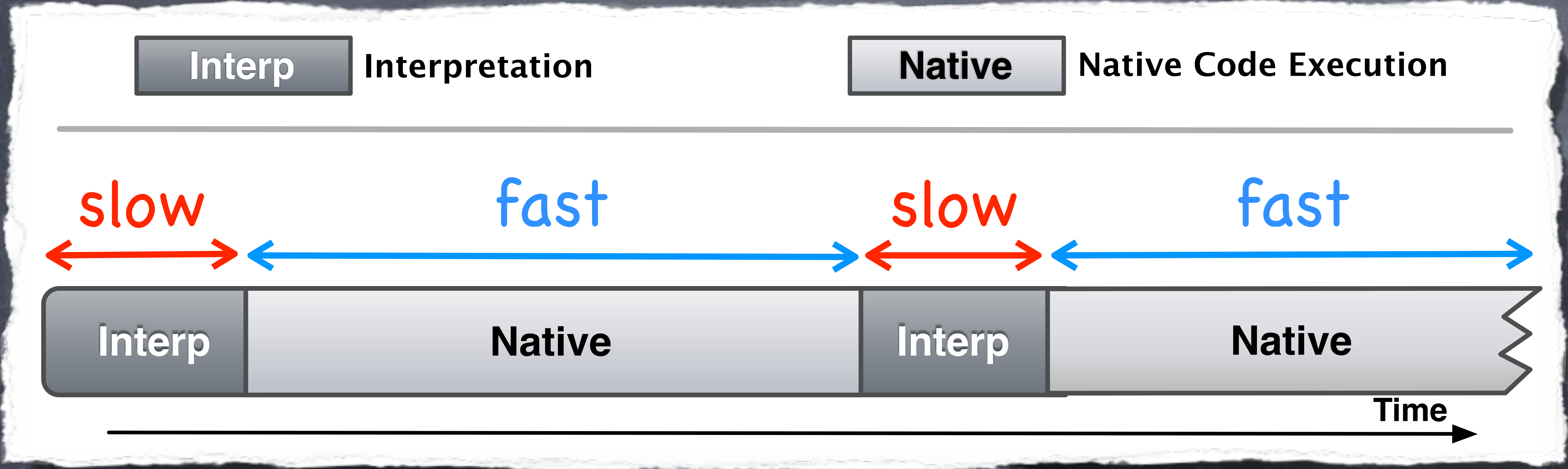
## What do we want to improve?



- 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

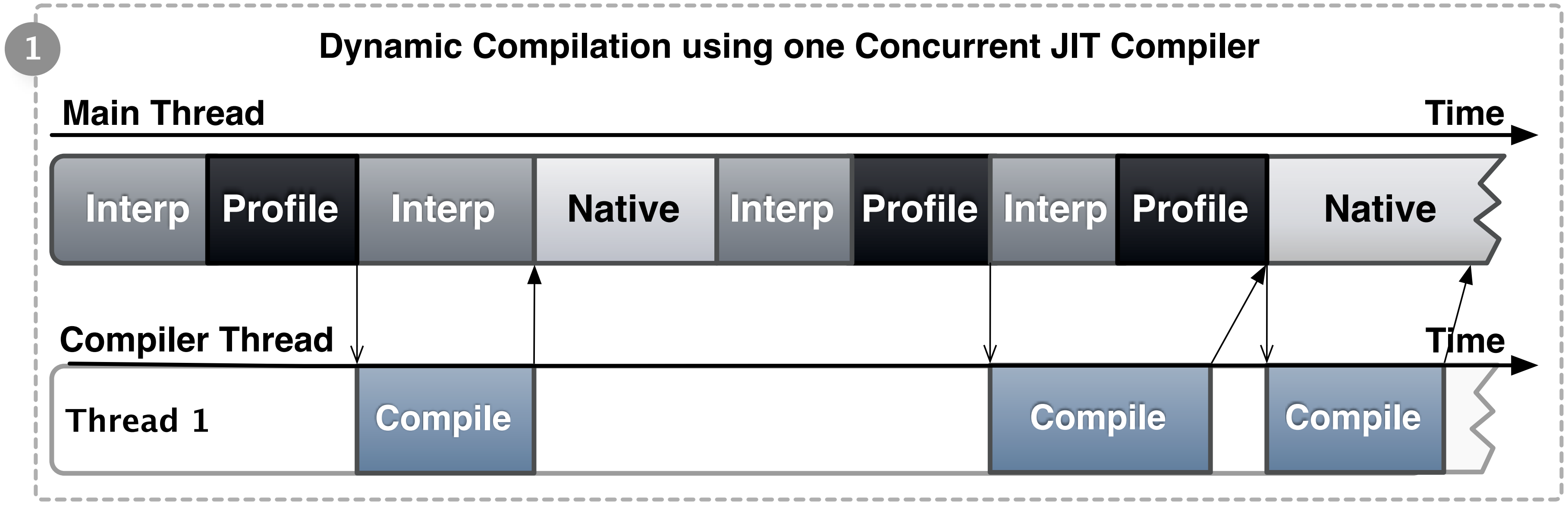
# Dynamic Compilation

## What do we want to improve?

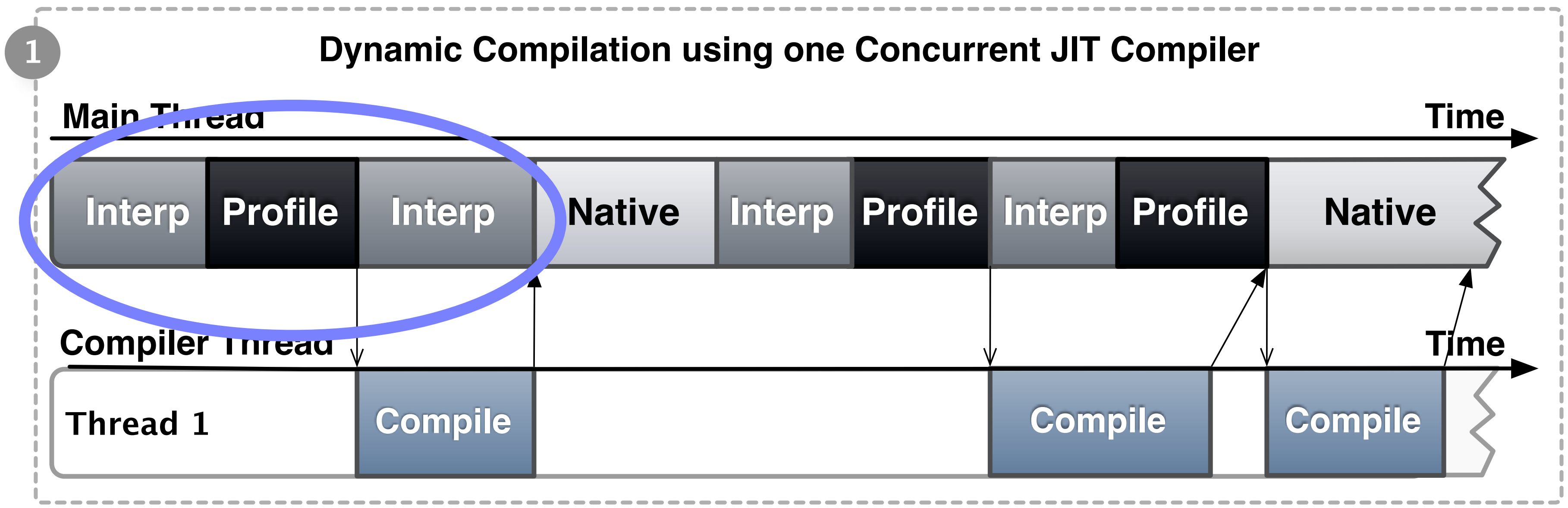


Earlier transition from interpretive to native execution

| | | | |
|---|---|---|---|
| Interp | Interpretation | Compile | Dynamic Compilation |
| Profile | Interpretation with Profiling | Native | Native Code Execution |

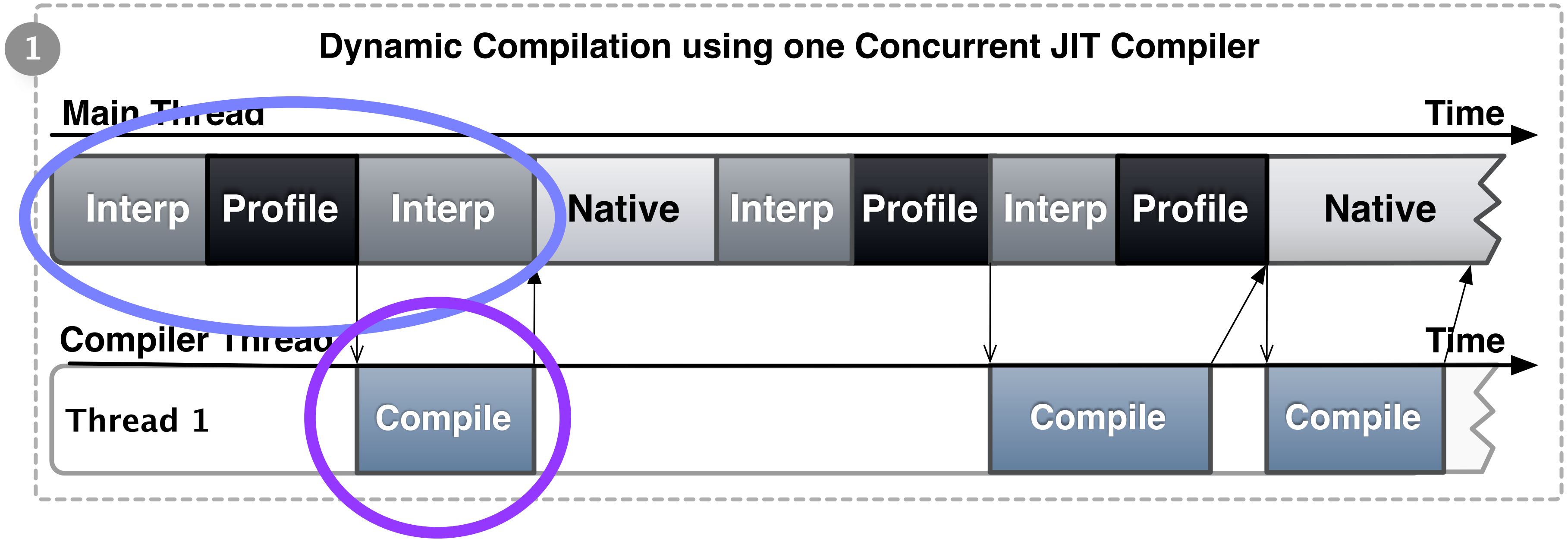


| | |
|---|---|
| Interp | Interpretation |
| Profile | Interpretation with Profiling |
| Compile | Dynamic Compilation |
| Native | Native Code Execution |



1 Dynamic Compilation using one Concurrent JIT Compiler

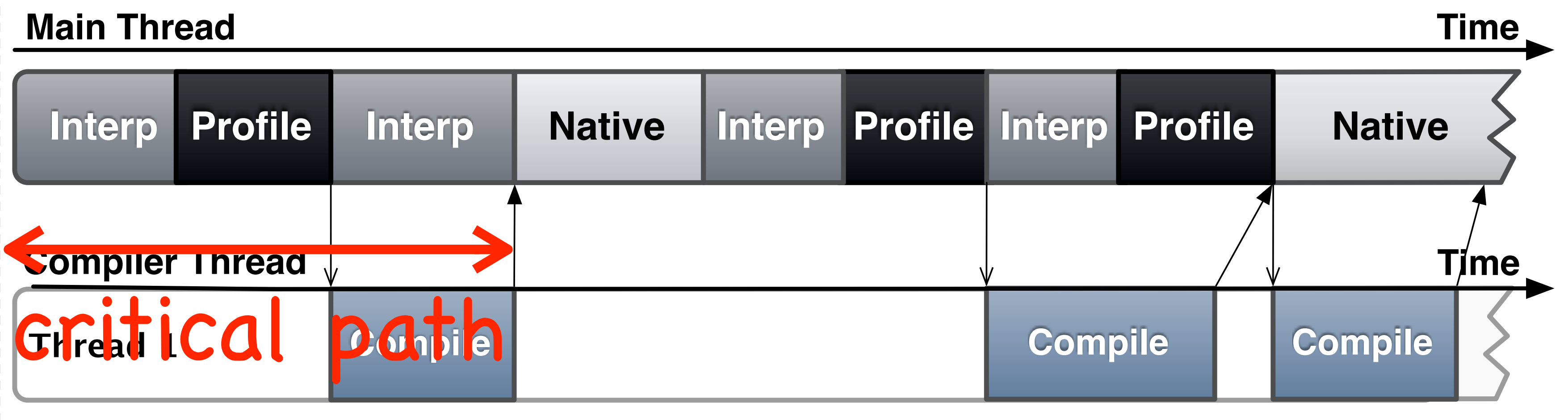
| | | | |
|---|---|---|---|
| Interp | Interpretation | Compile | Dynamic Compilation |
| Profile | Interpretation with Profiling | Native | Native Code Execution |



| | | | |
|---|---|---|---|
| Interp | Interpretation | Compile | Dynamic Compilation |
| Profile | Interpretation with Profiling | Native | Native Code Execution |

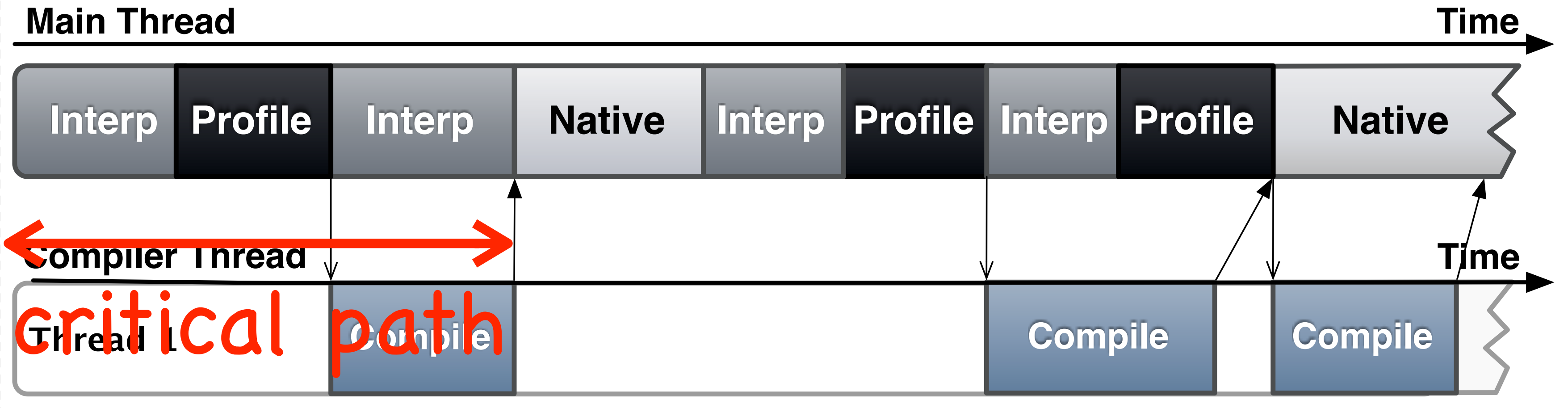
1

## Dynamic Compilation using one Concurrent JIT Compiler

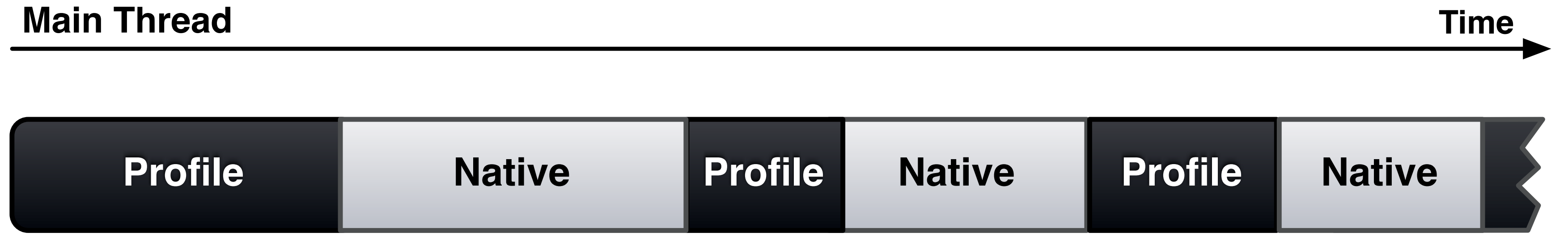


| | |
|---|---|
| Interp | Interpretation |
| Profile | Interpretation with Profiling |
| Compile | Dynamic Compilation |
| Native | Native Code Execution |

1 **Dynamic Compilation using one Concurrent JIT Compiler**

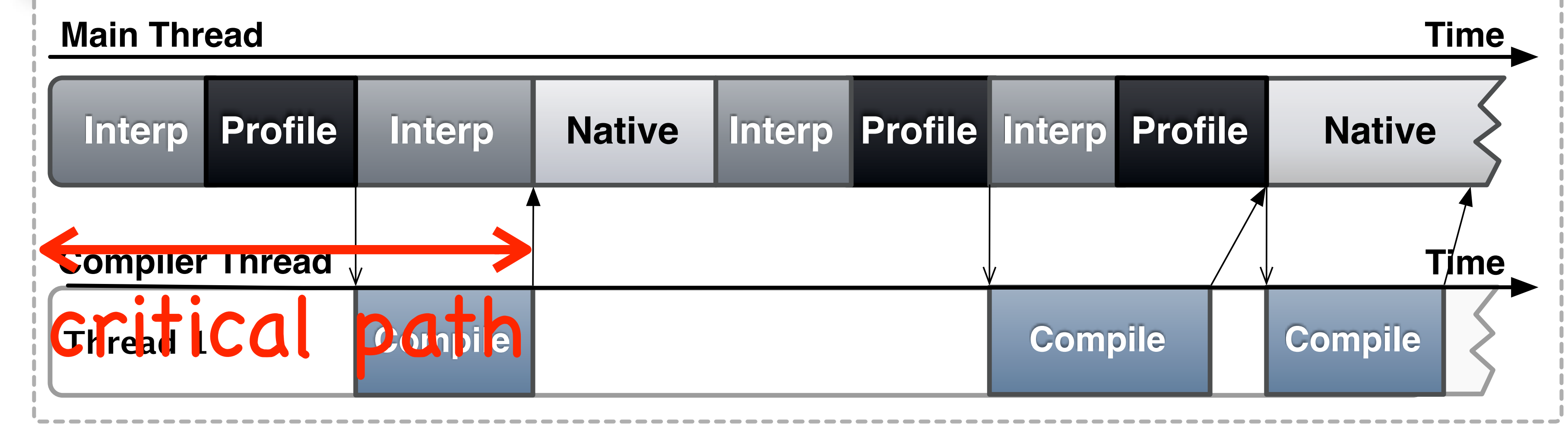


2 **Dynamic Compilation using Concurrent and Parallel JIT Compiler Task Farm**

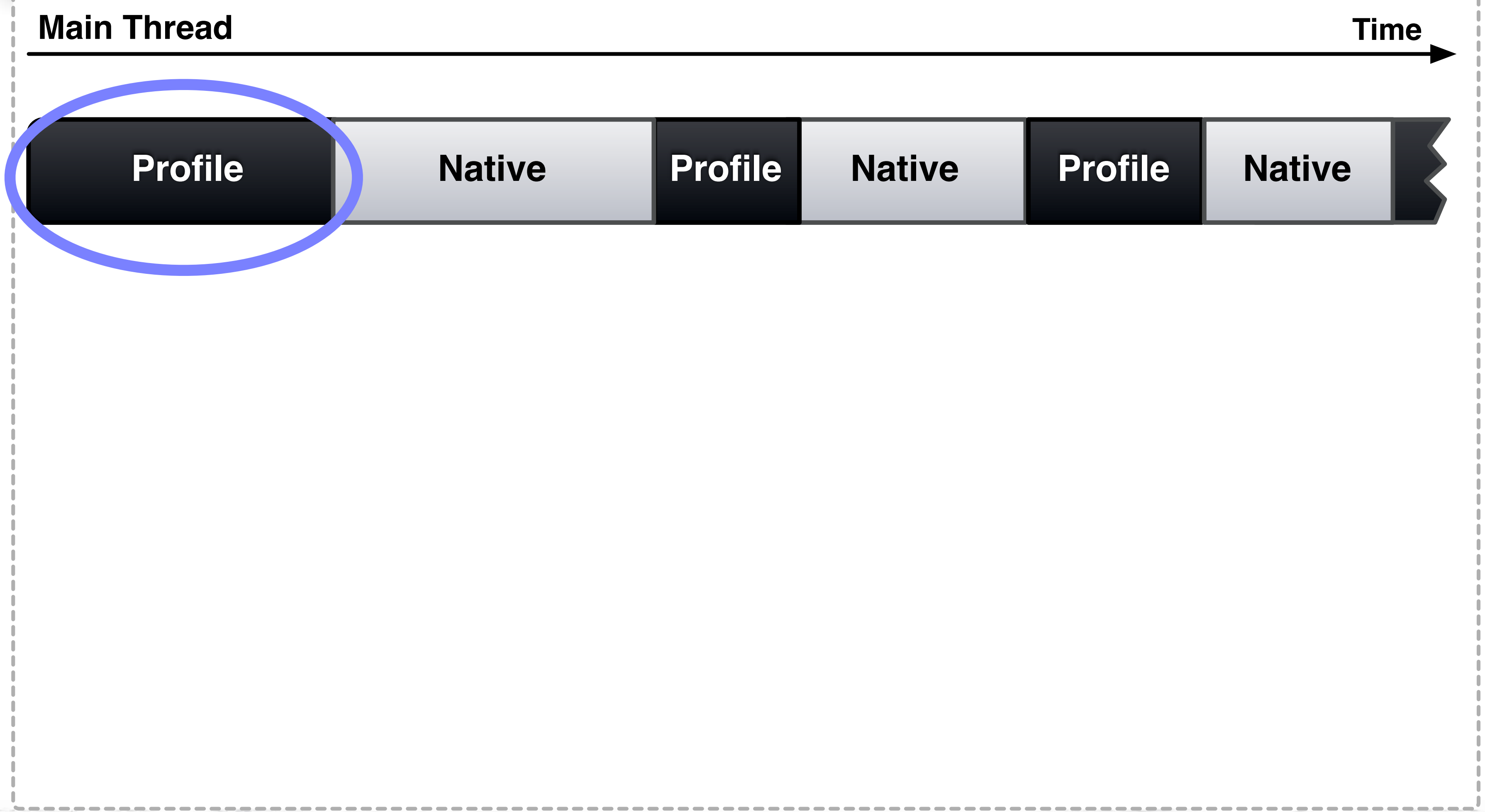


| | | | |
|---|---|---|---|
| Interp | Interpretation | Compile | Dynamic Compilation |
| Profile | Interpretation with Profiling | Native | Native Code Execution |

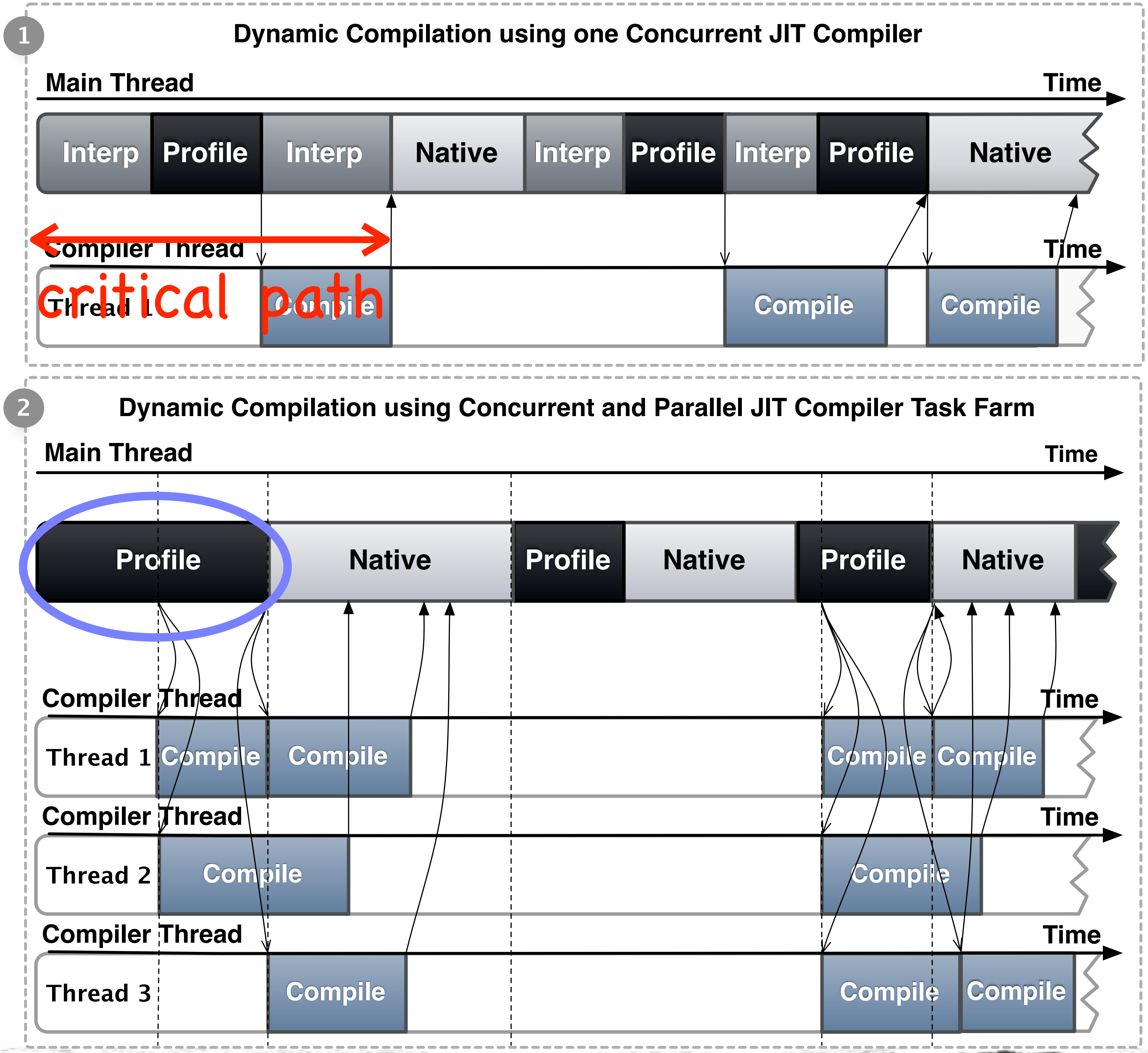
1 Dynamic Compilation using one Concurrent JIT Compiler



2 Dynamic Compilation using Concurrent and Parallel JIT Compiler Task Farm

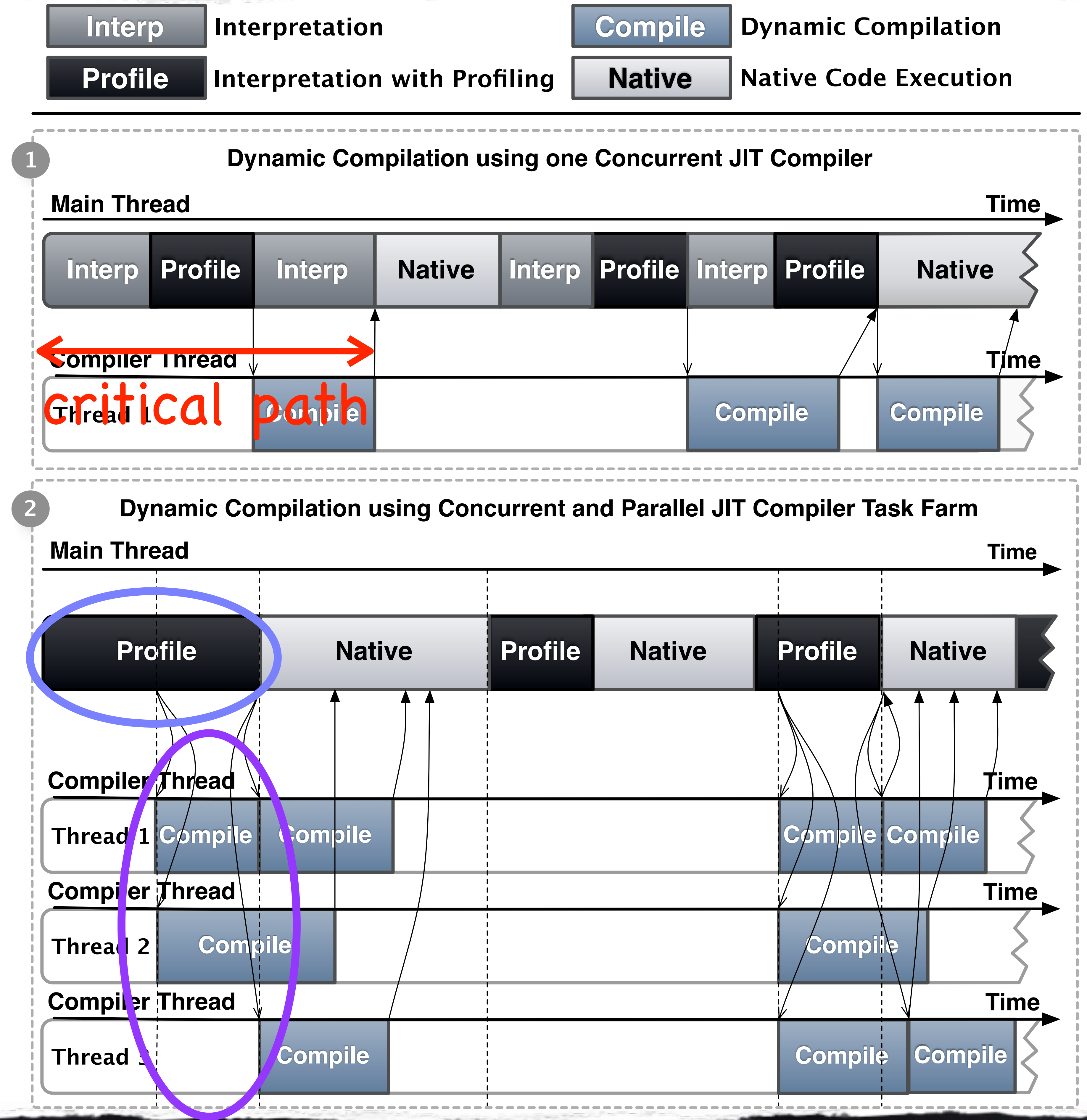


| | |
|---|---|
| Interp | Interpretation |
| Profile | Interpretation with Profiling |
| Compile | Dynamic Compilation |
| Native | Native Code Execution |



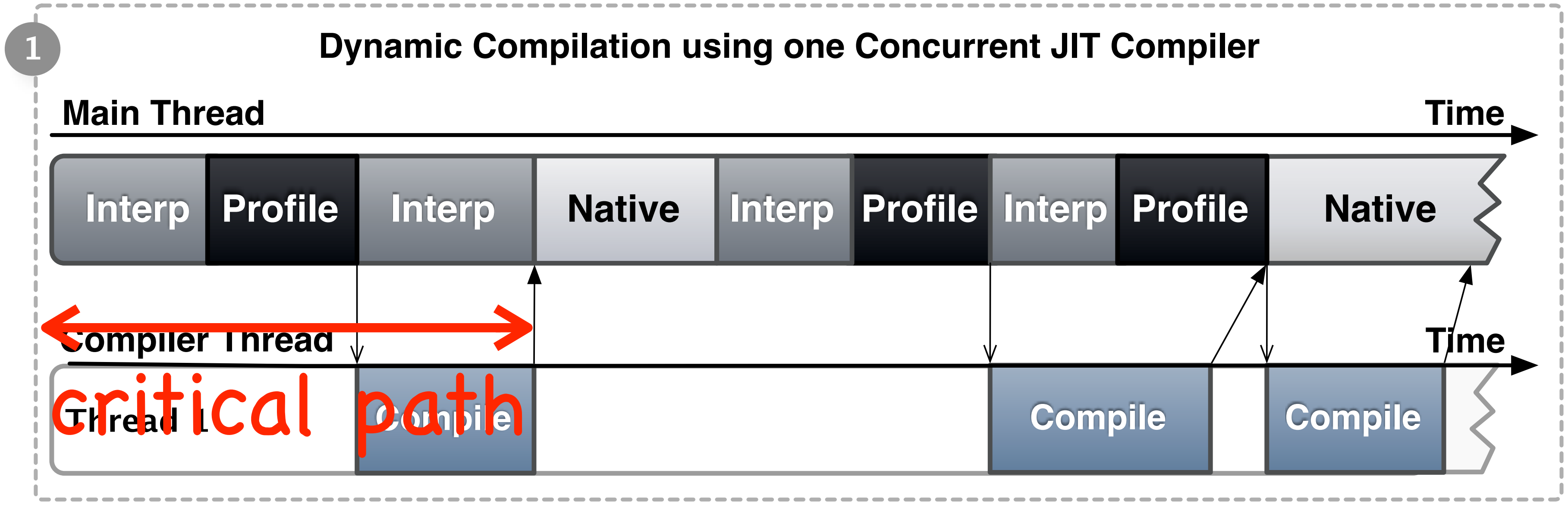
1 Dynamic Compilation using one Concurrent JIT Compiler

2 Dynamic Compilation using Concurrent and Parallel JIT Compiler Task Farm



Interp Interpretation
Profile Interpretation with Profiling
Compile Dynamic Compilation
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
Time
Thread 1
Compile
Compile
Compile
critical path
2
Dynamic Compilation using Concurrent and Parallel JIT Compiler Task Farm
Main Thread
Time
Profile
Native
Profile
Native
Profile
Native
Compiler Thread
Time
Thread 1
Compile
Compile
Compile
Compile
Compiler Thread
Time
Thread 2
Compile
Compile
Compiler Thread
Time
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

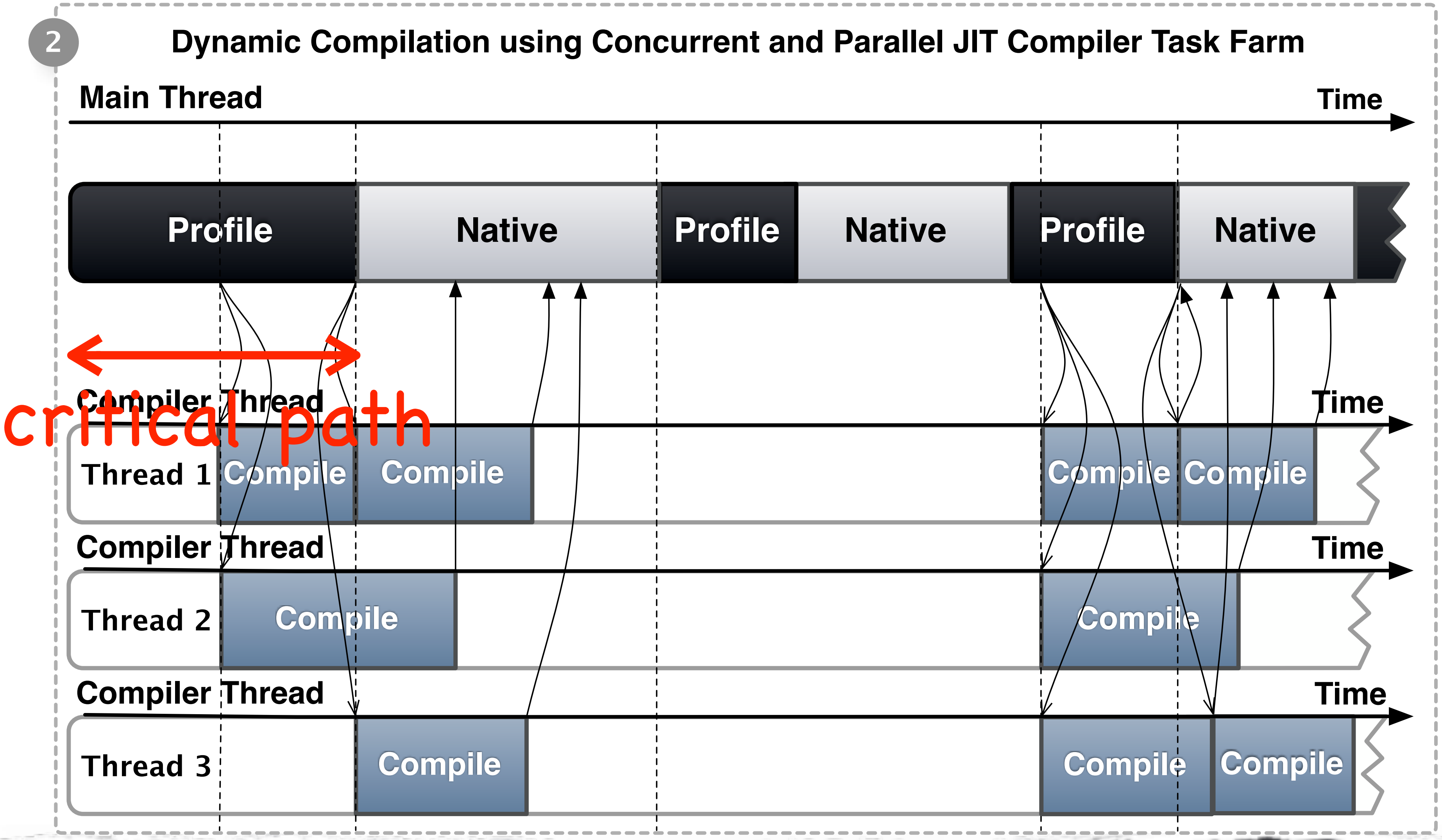
Interp Profile Interp Native Interp Profile Interp Profile Native

Compiler Thread

Time

Thread 1 Compile Compile Compile

critical path



2 Dynamic Compilation using Concurrent and Parallel JIT Compiler Task Farm

Main Thread

Time

Profile Native Profile Native Profile Native

Compiler Thread

Time

Thread 1 Compile Compile Compile Compile

Compiler Thread

Time

Thread 2 Compile Compile

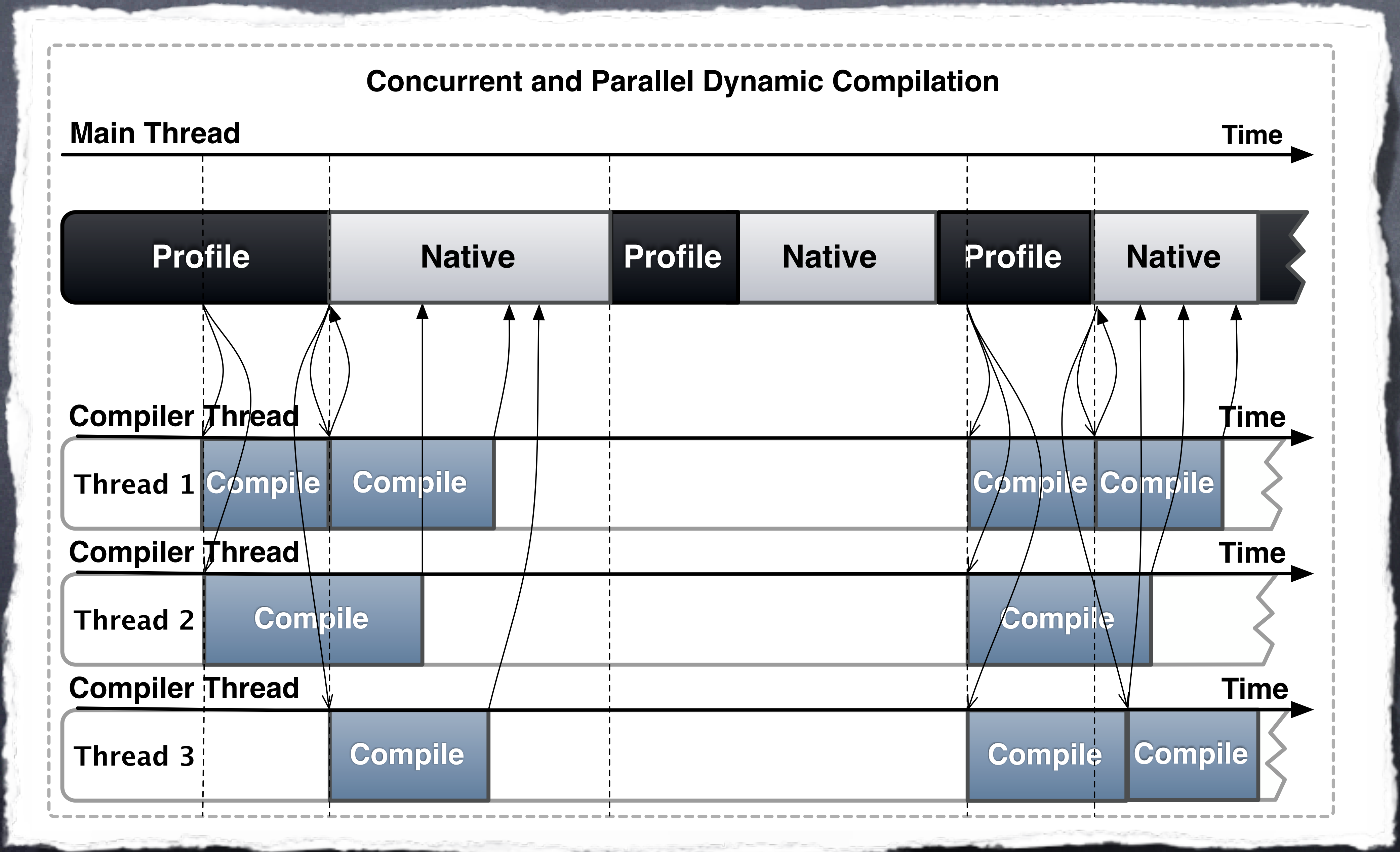
Compiler Thread

Time

Thread 3 Compile Compile Compile

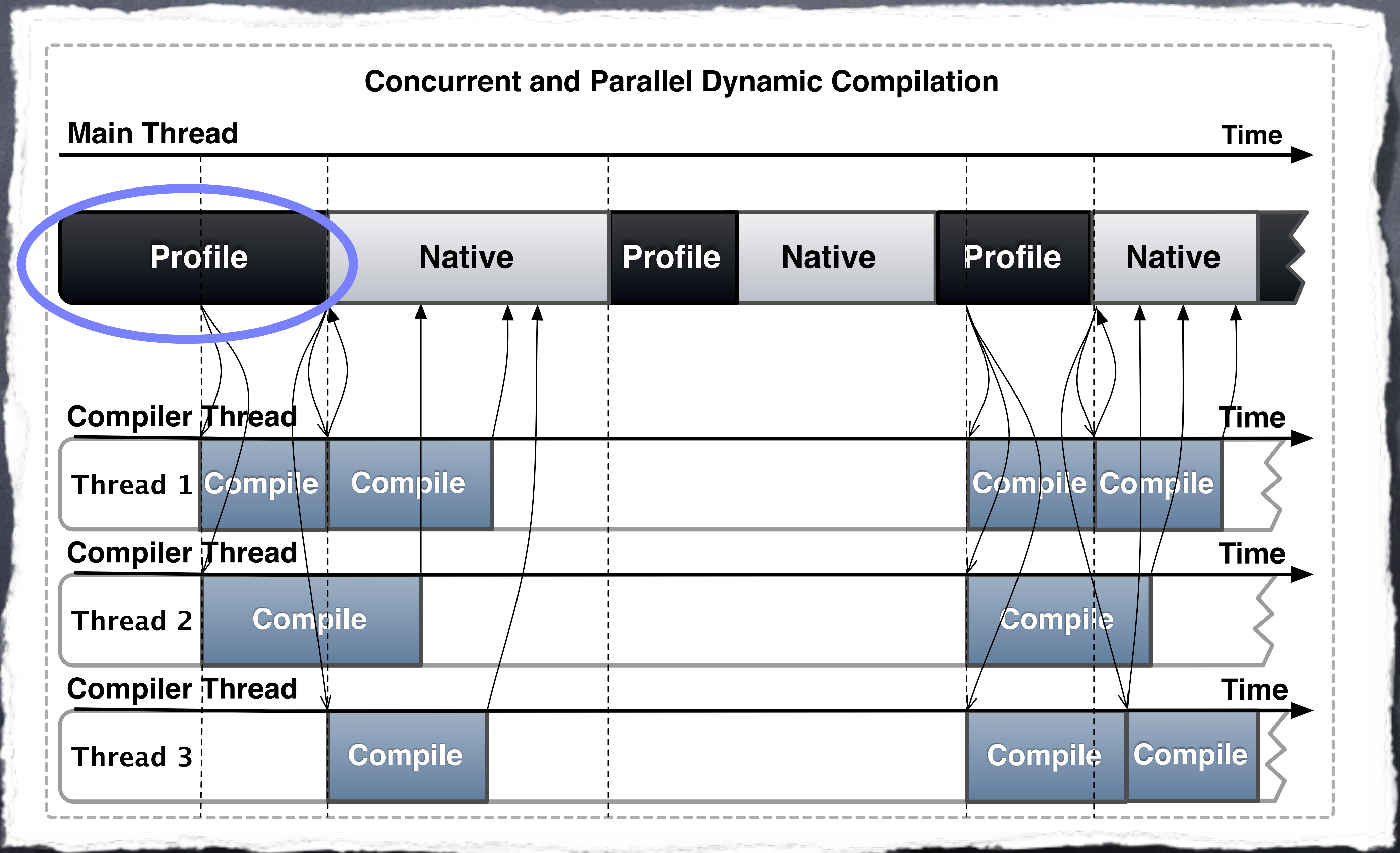
critical path

# Solution To Dynamic Compilation Latency Problem



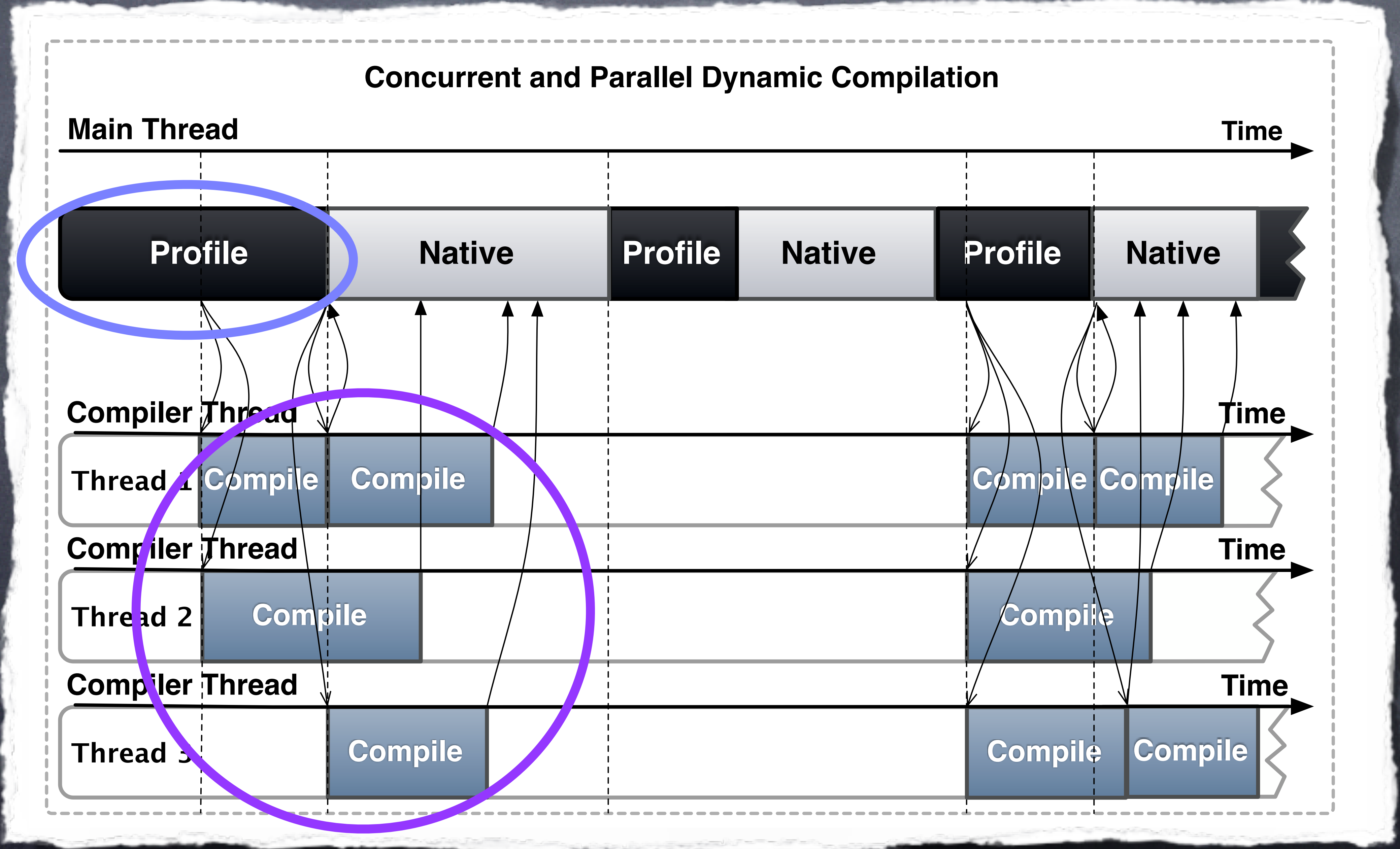
# Solution To Dynamic Compilation Latency Problem

- improve **code discovery/profiling**



# Solution To Dynamic Compilation Latency Problem

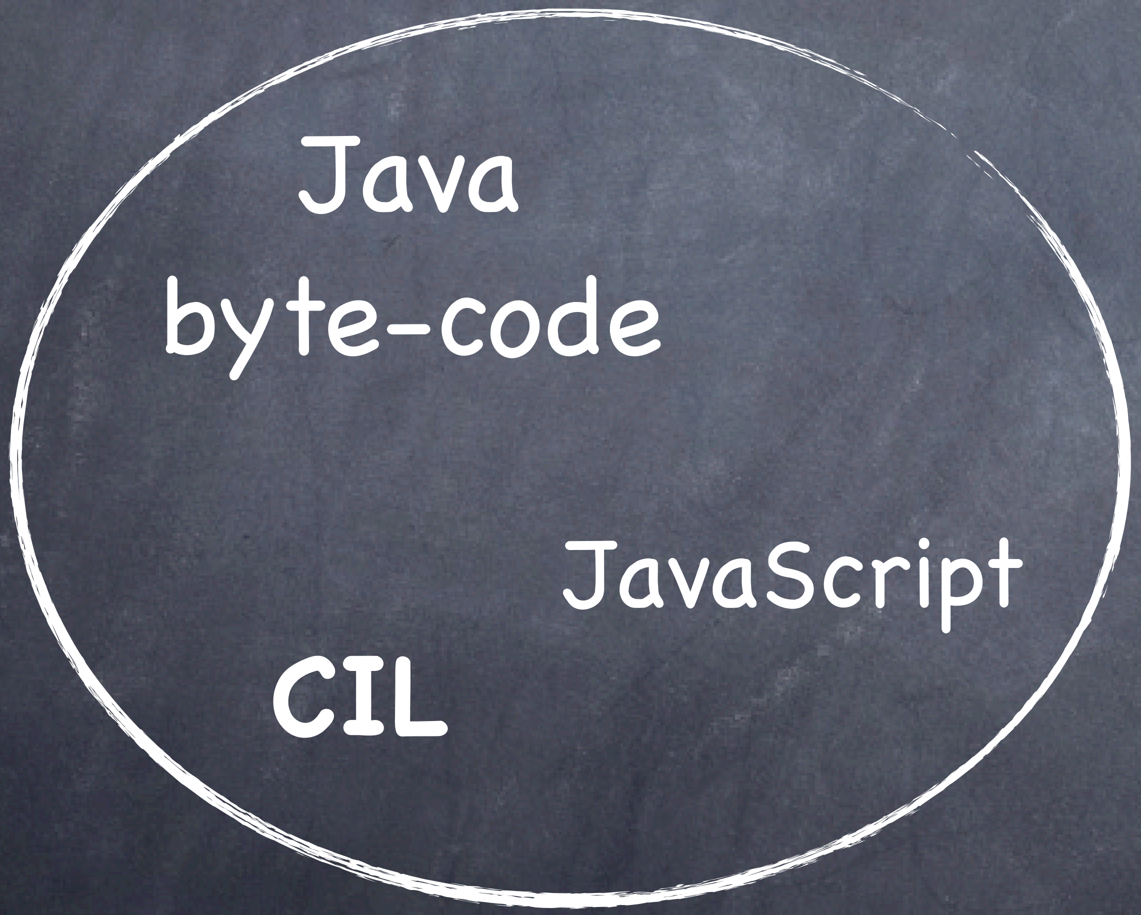
- improve **code discovery/profiling**
- improve dynamic compilation workload throughput



# How hard can Code Discovery be?

# How hard can Code Discovery be?

Static

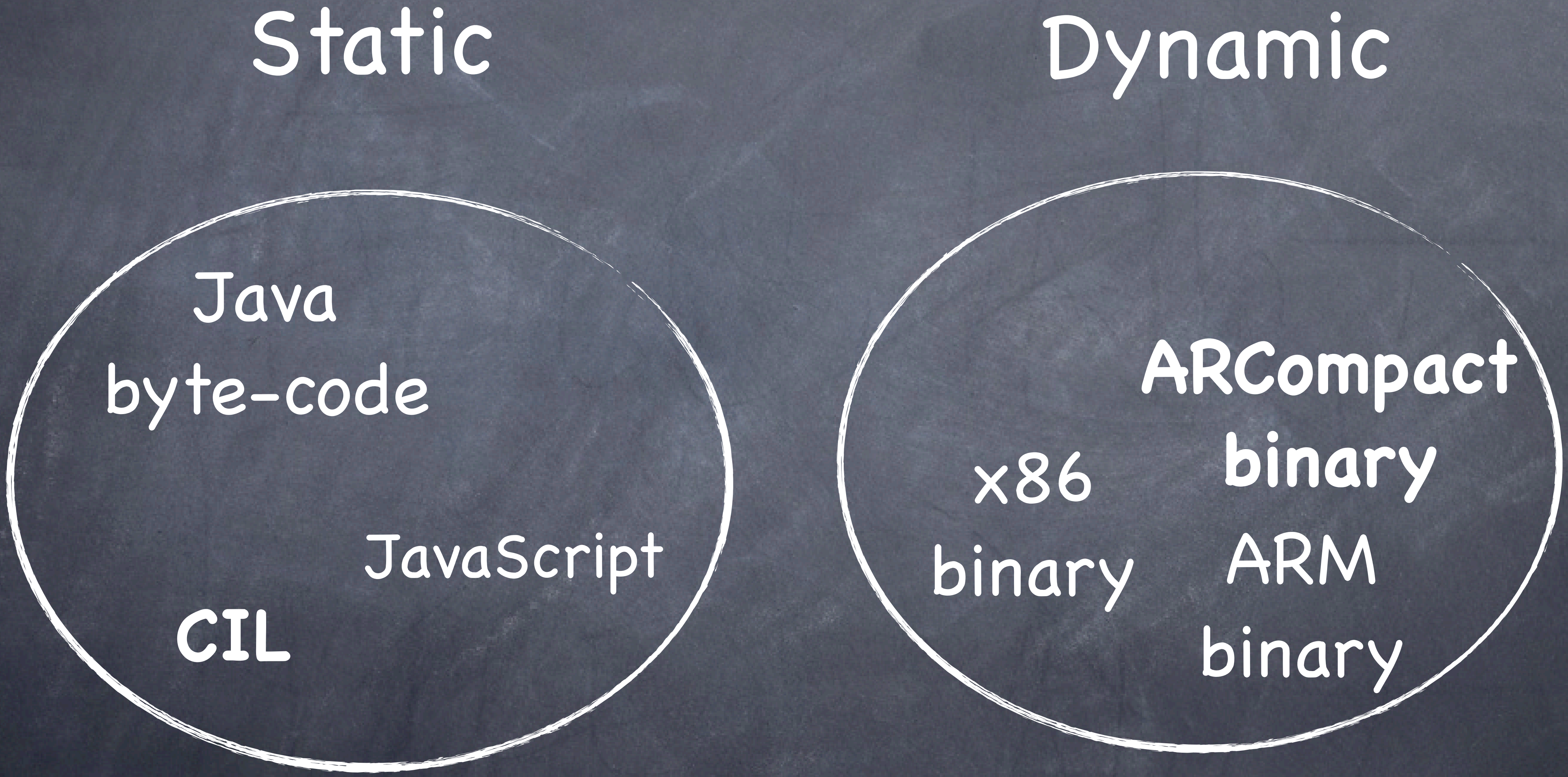


Java byte-code

JavaScript

CIL

# How hard can Code Discovery be?



Static

Java byte-code

JavaScript

**CIL**

Dynamic

**ARCompact binary**

x86 binary

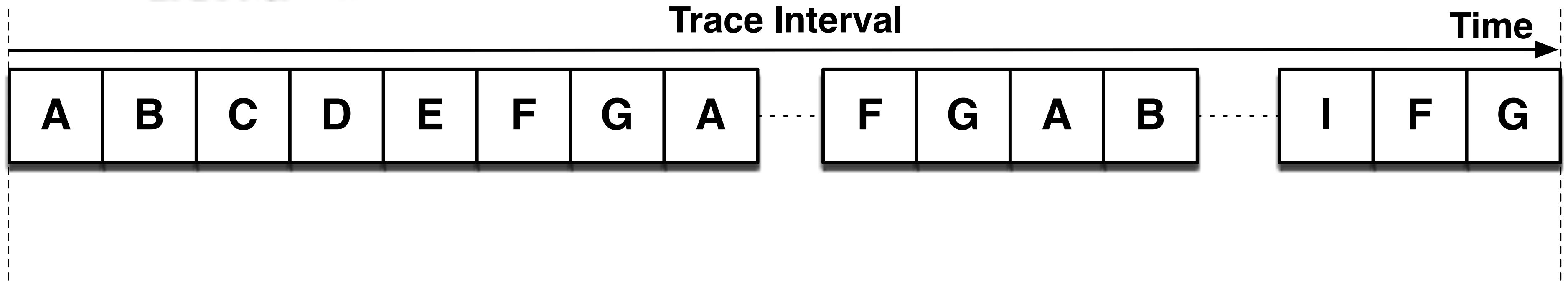
ARM binary

# How hard can Code Discovery be?

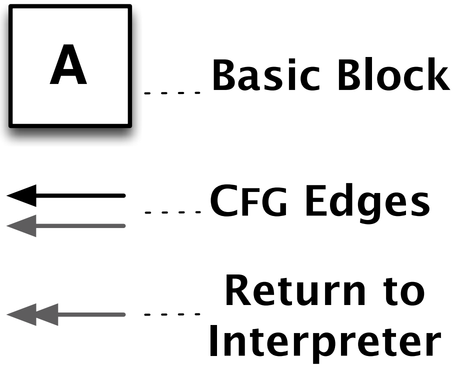
"A crucial problem in the decompilation or disassembly of computer programs is the identification of executable code, i.e. the separation of instructions from data. This problem, for most computer architectures, is equivalent to the Halting Problem and is therefore unsolvable in general."

[Horspool and Marovac - 1980]

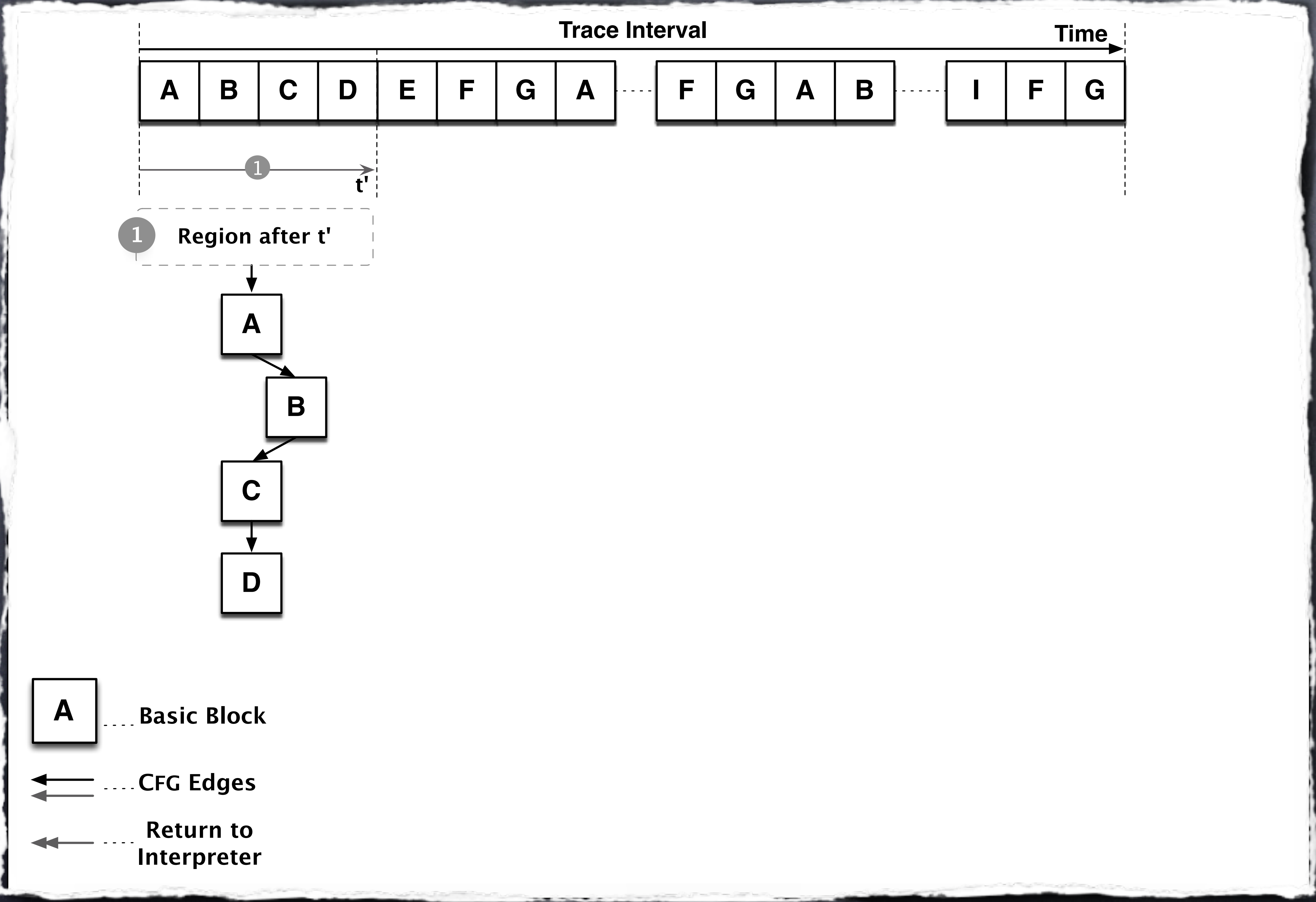
# Incremental Code Discovery



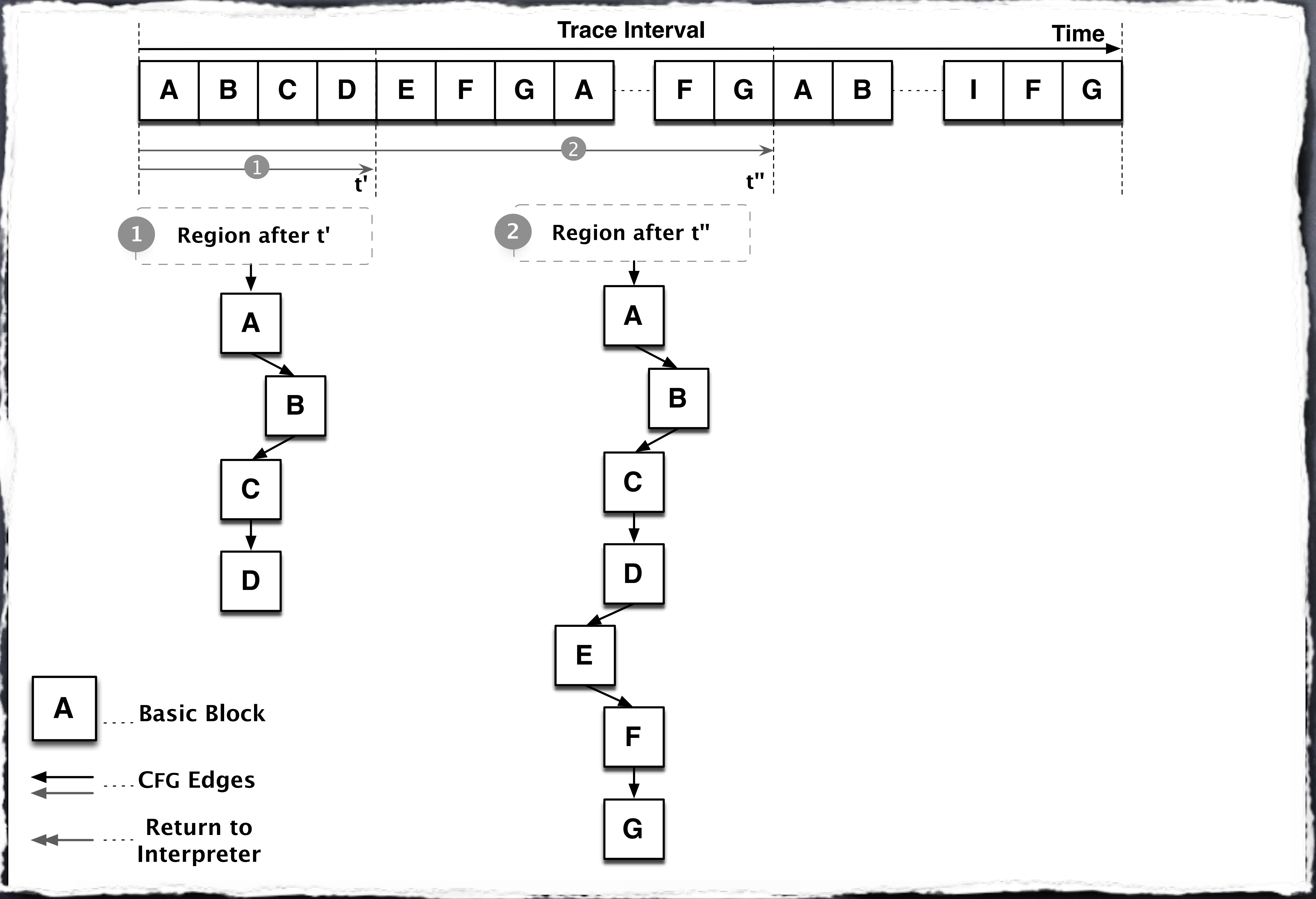
**Sequence of interpreted basic blocks**



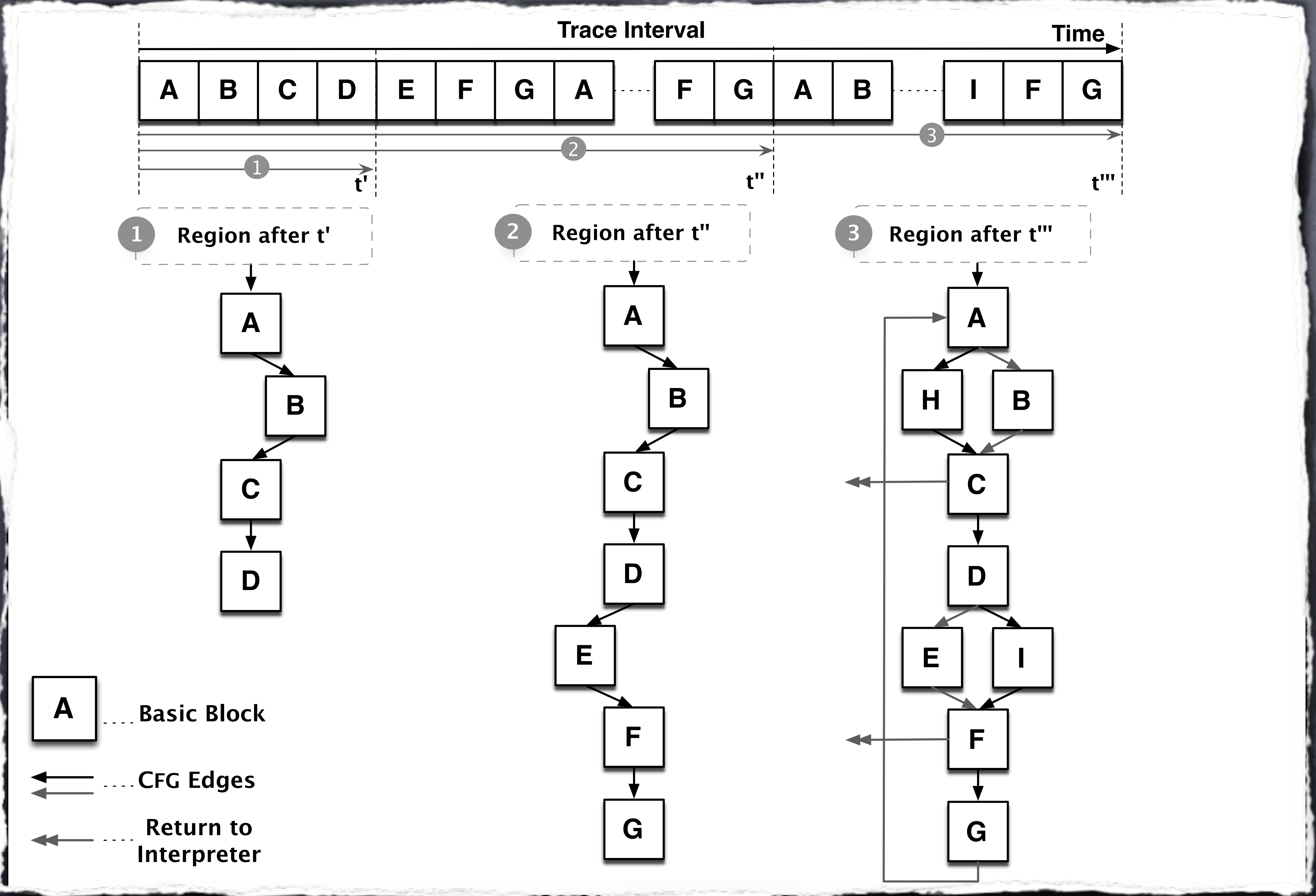
# Incremental Code Discovery



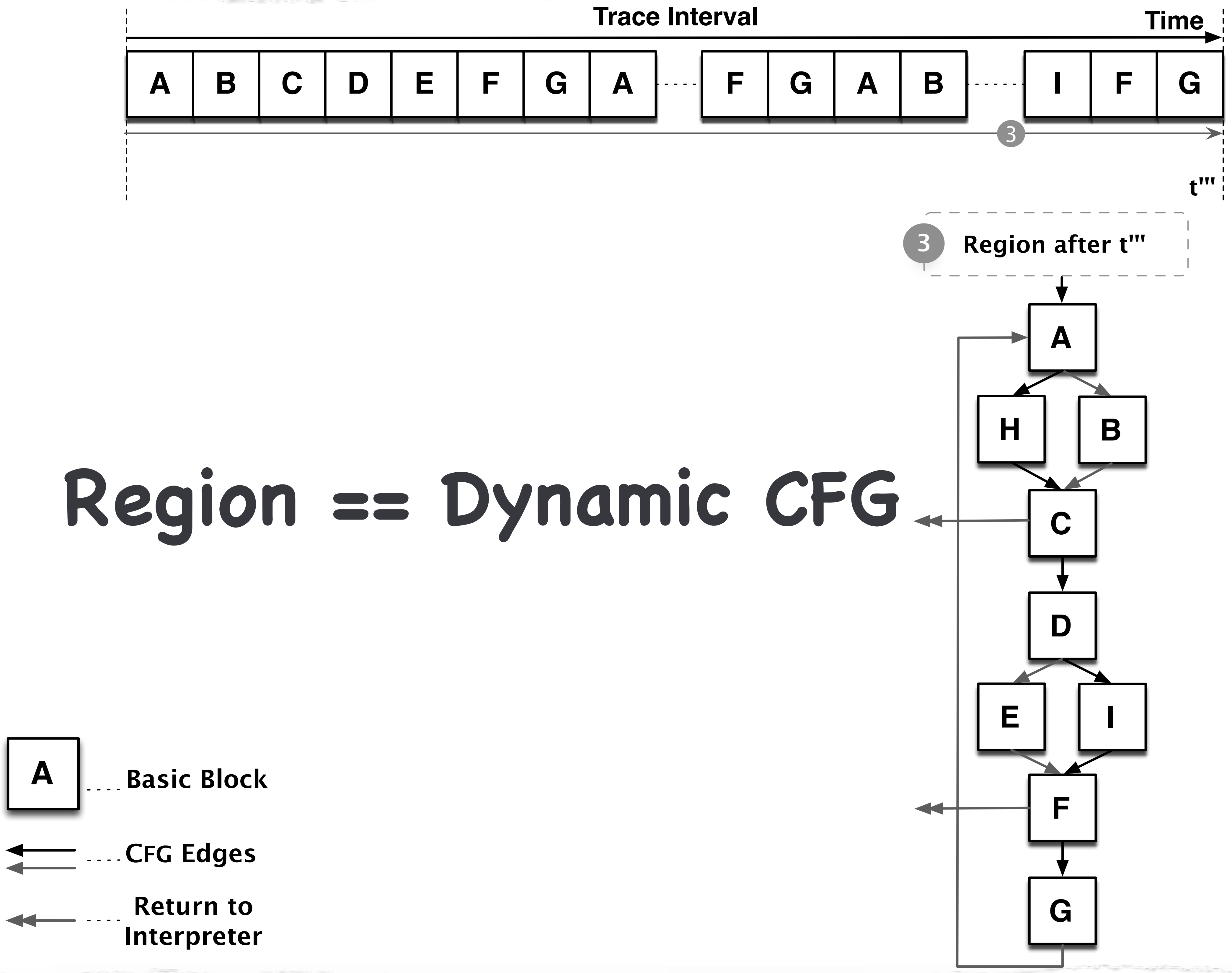
# Incremental Code Discovery



# Incremental Code Discovery



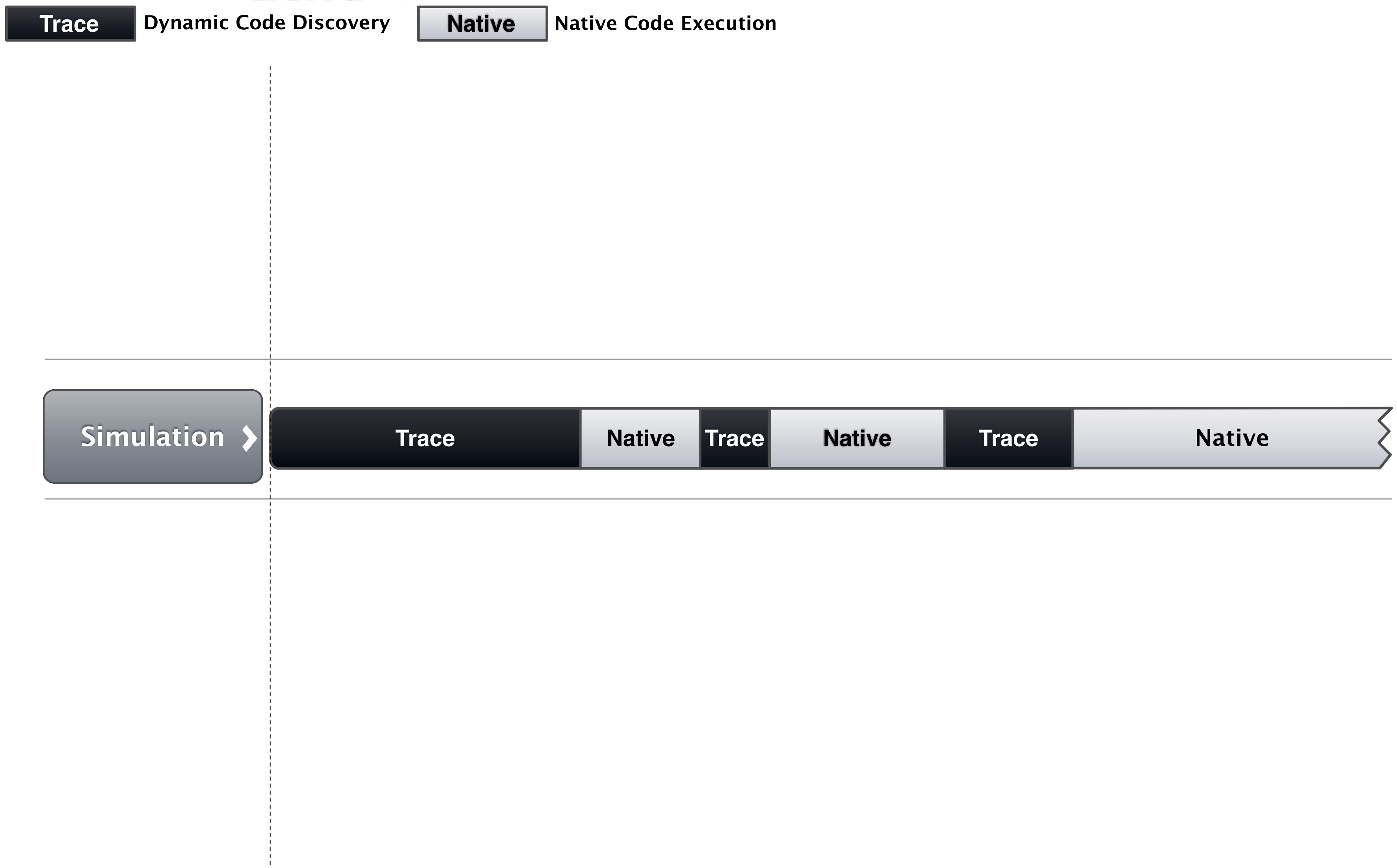
# Incremental Code Discovery



**Region == Dynamic CFG**

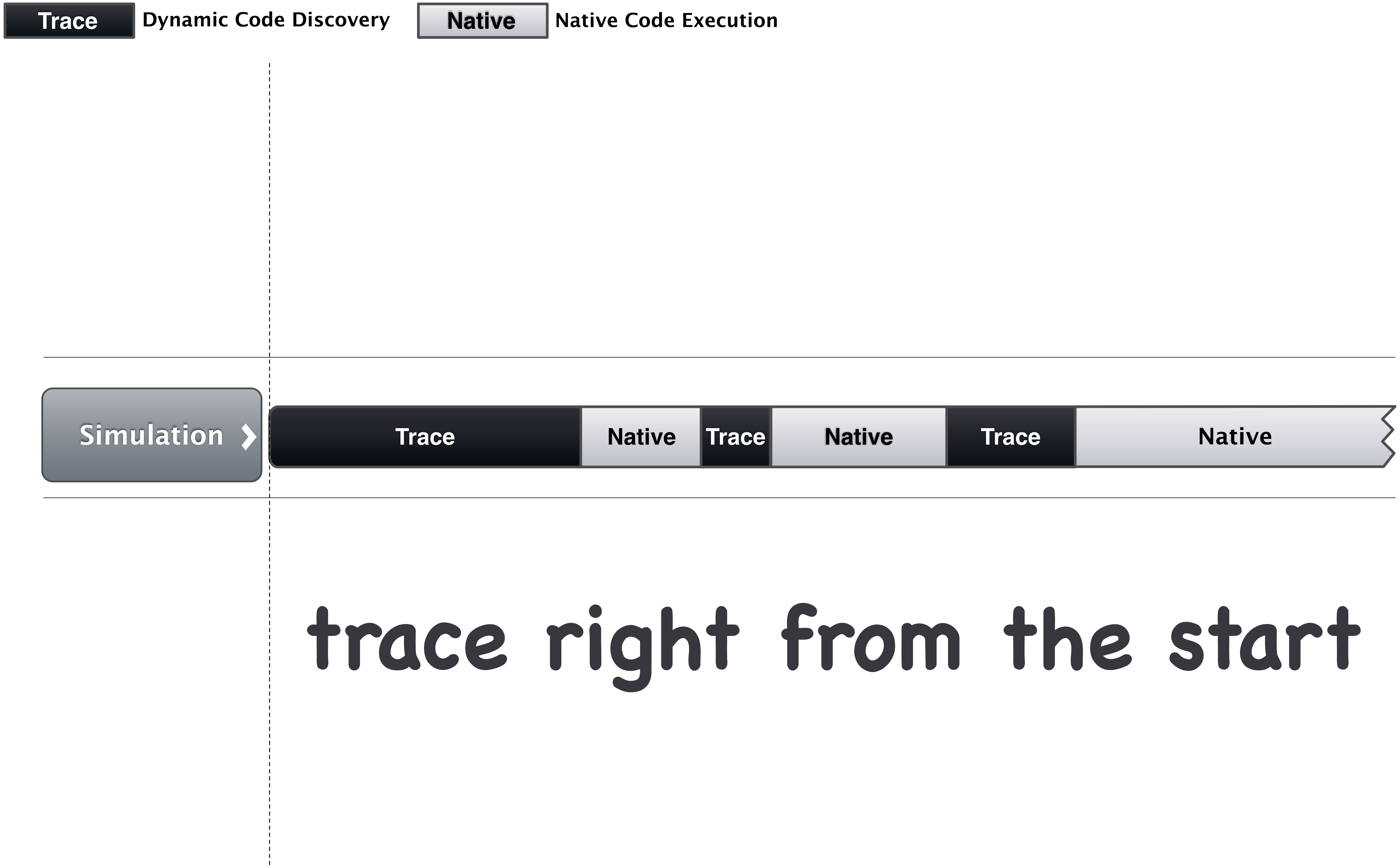
# Concurrent and Parallel JIT Compilation in Action

(reducing the critical path)



# Concurrent and Parallel JIT Compilation in Action

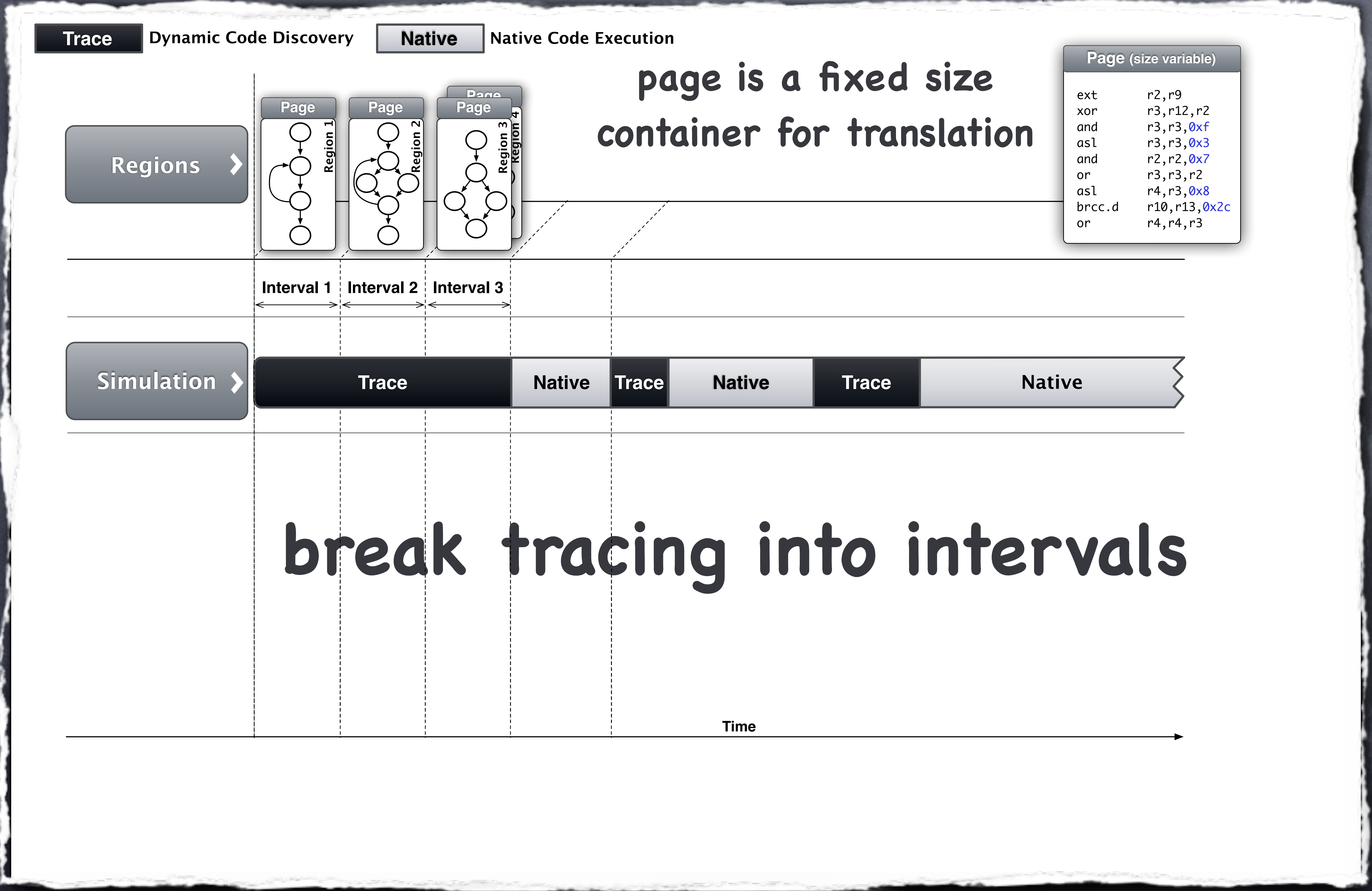
(reducing the critical path)



**trace right from the start**

# Concurrent and Parallel JIT Compilation in Action

(reducing the critical path)



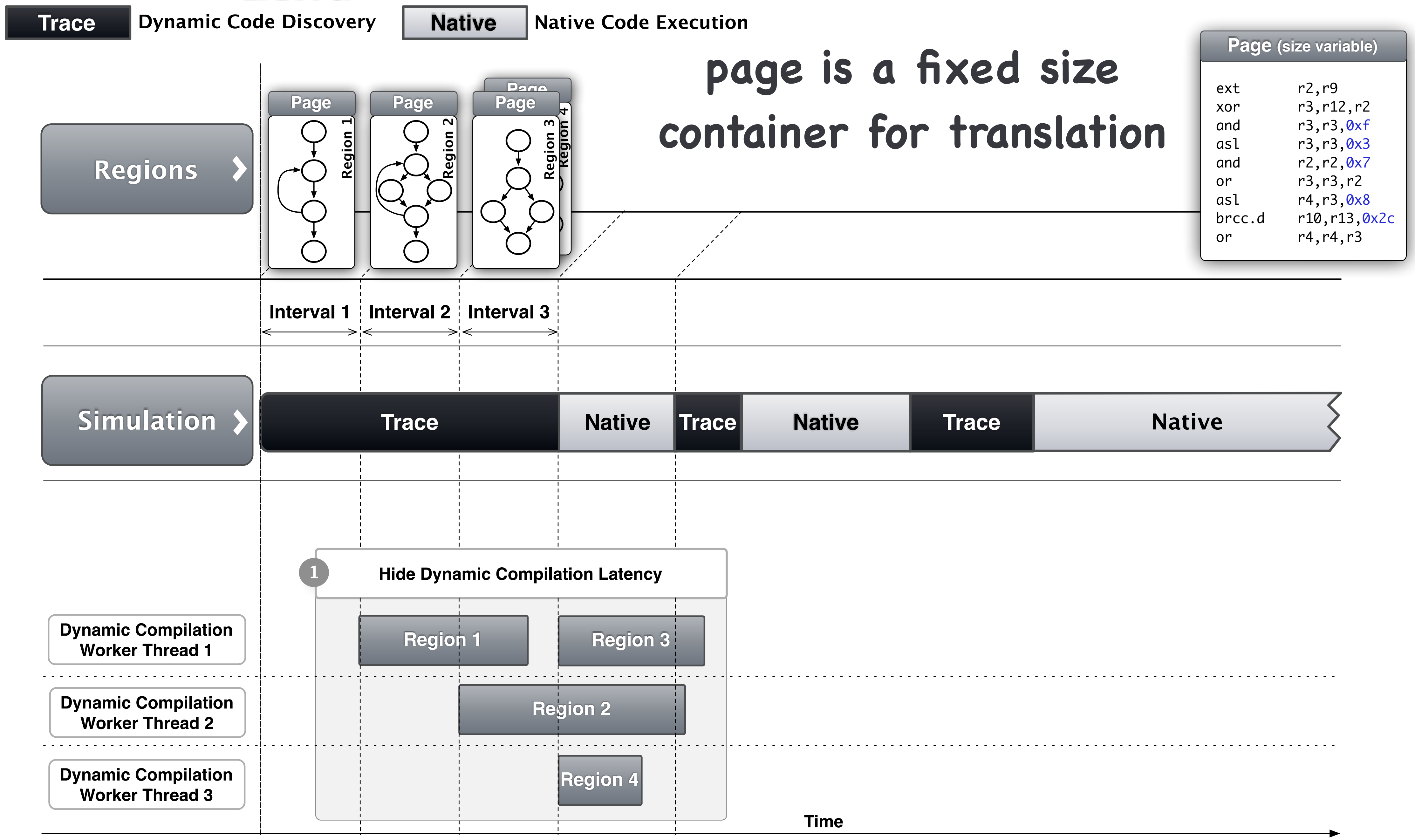
page is a fixed size container for translation

```
ext     r2,r9
xor     r3,r12,r2
and     r3,r3,0xf
asl     r3,r3,0x3
and     r2,r2,0x7
or      r3,r3,r2
asl     r4,r3,0x8
brcc.d  r10,r13,0x2c
or      r4,r4,r3
```

break tracing into intervals

# Concurrent and Parallel JIT Compilation in Action

(reducing the critical path)



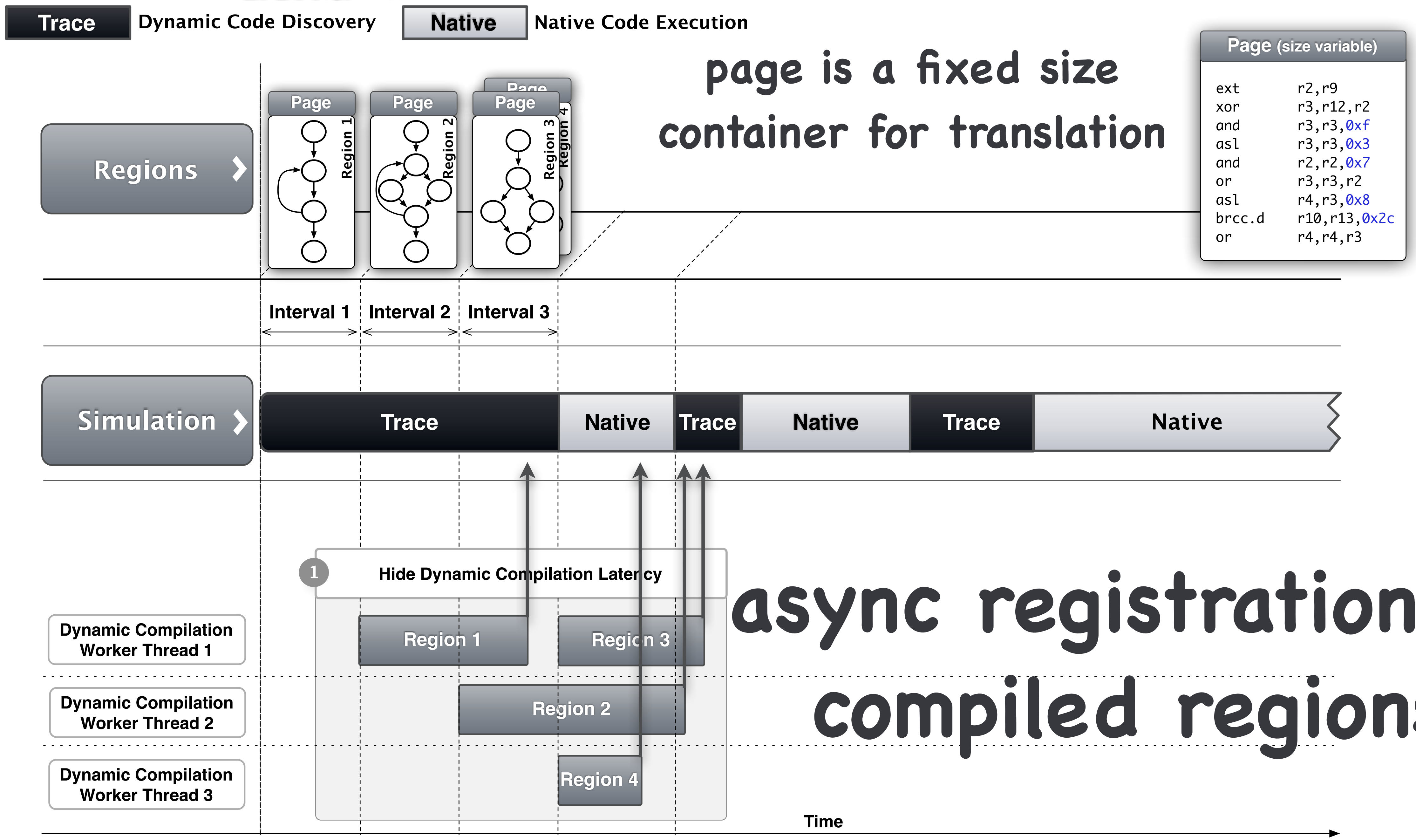
```
ext       r2,r9
xor       r3,r12,r2
and       r3,r3,0xf
asl       r3,r3,0x3
and       r2,r2,0x7
or        r3,r3,r2
asl       r4,r3,0x8
brcc.d    r10,r13,0x2c
or        r4,r4,r3
```

page is a fixed size container for translation

hide compilation latency

# Concurrent and Parallel JIT Compilation in Action

(reducing the critical path)



```
ext       r2,r9
xor       r3,r12,r2
and       r3,r3,0xf
asl       r3,r3,0x3
and       r2,r2,0x7
or        r3,r3,r2
asl       r4,r3,0x8
brcc.d    r10,r13,0x2c
or        r4,r4,r3
```

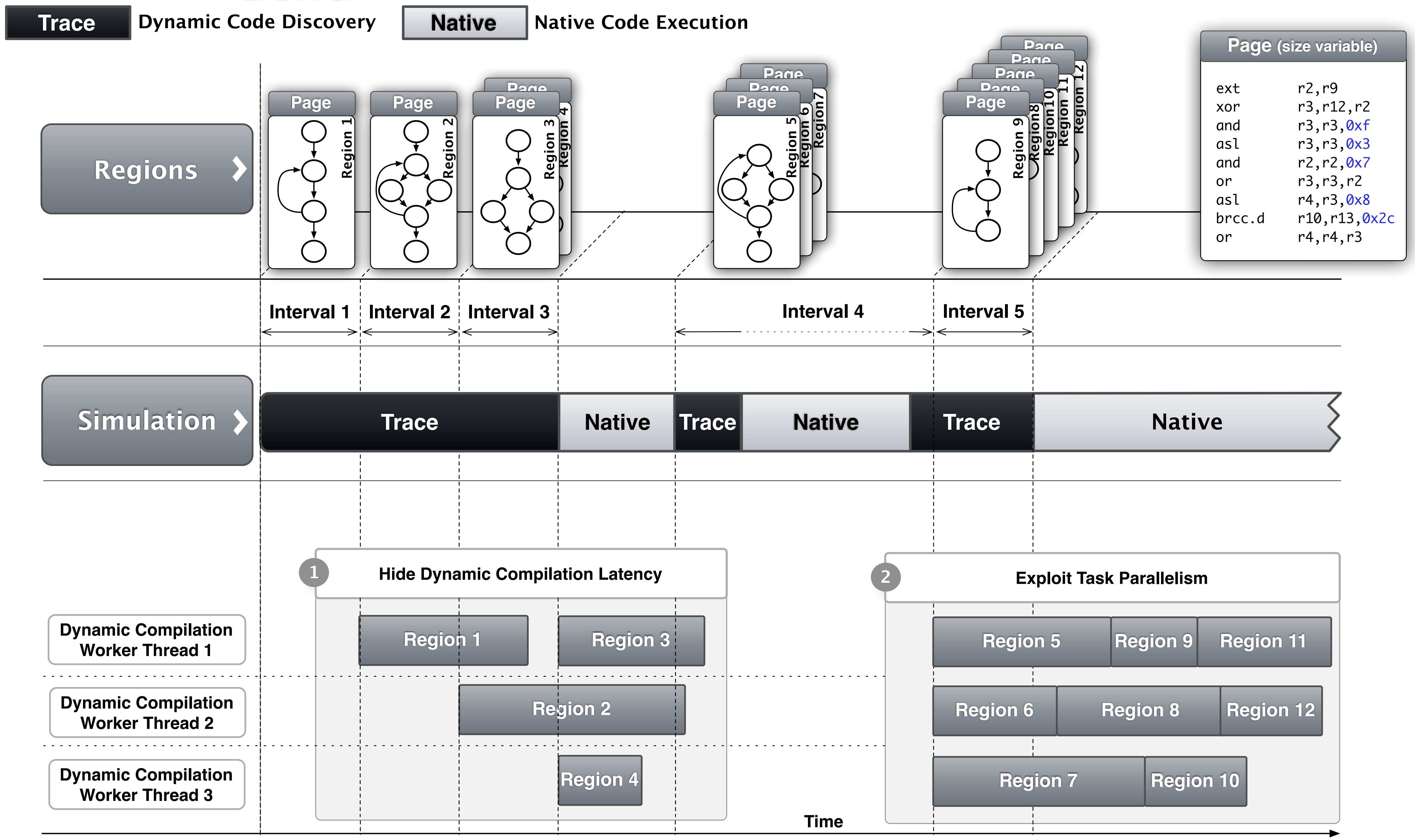
page is a fixed size container for translation

async registration of compiled regions

hide compilation latency

# Concurrent and Parallel JIT Compilation in Action

## (reducing the critical path)



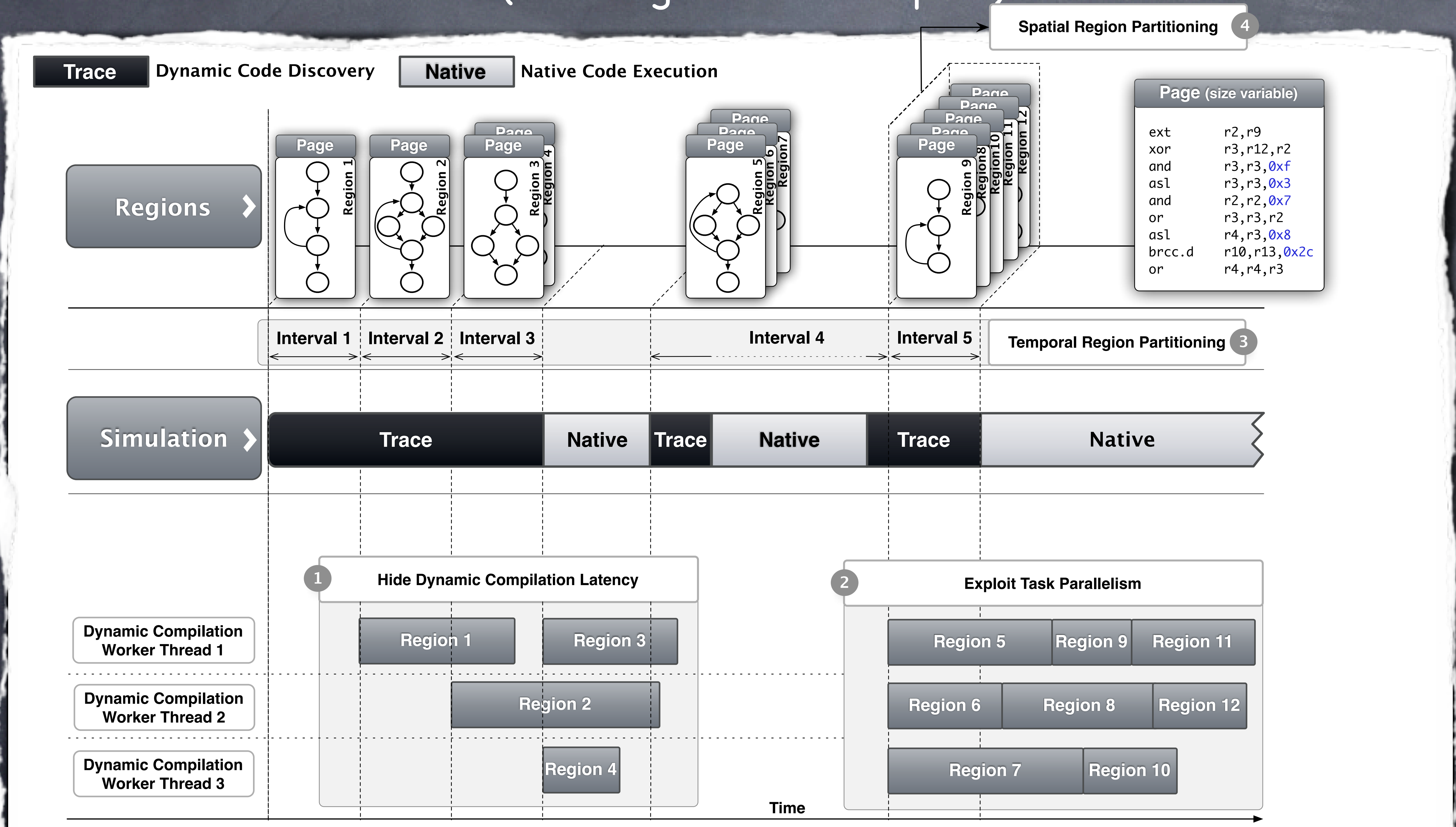
```
ext     r2,r9
xor     r3,r12,r2
and     r3,r3,0xf
asl     r3,r3,0x3
and     r2,r2,0x7
or      r3,r3,r2
asl     r4,r3,0x8
brcc.d  r10,r13,0x2c
or      r4,r4,r3
```

**hide compilation latency**

**exploit task parallelism**

# Concurrent and Parallel JIT Compilation in Action

(reducing the critical path)

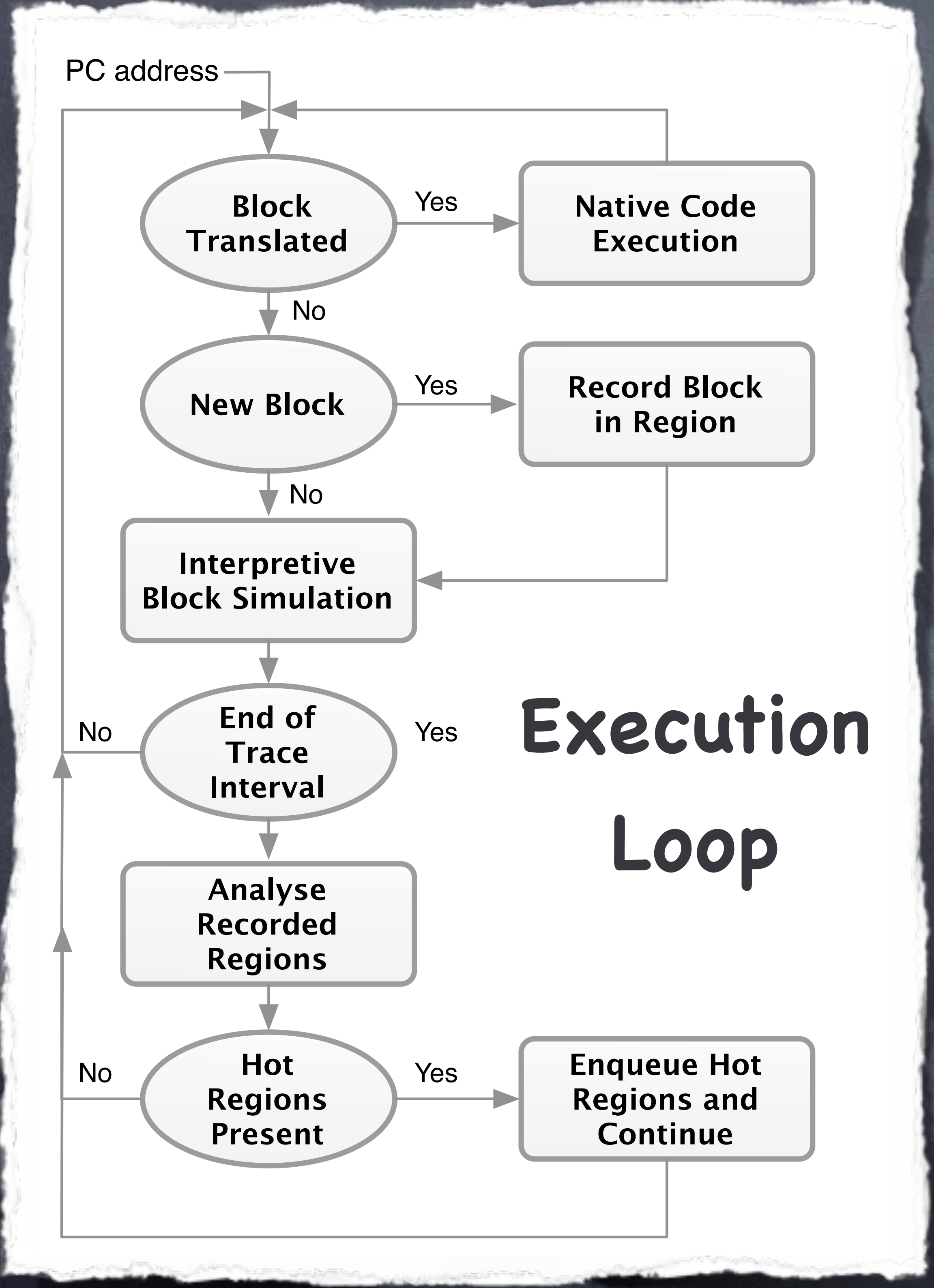


```
ext       r2,r9
xor       r3,r12,r2
and       r3,r3,0xf
asl       r3,r3,0x3
and       r2,r2,0x7
or        r3,r3,r2
asl       r4,r3,0x8
brcc.d    r10,r13,0x2c
or        r4,r4,r3
```

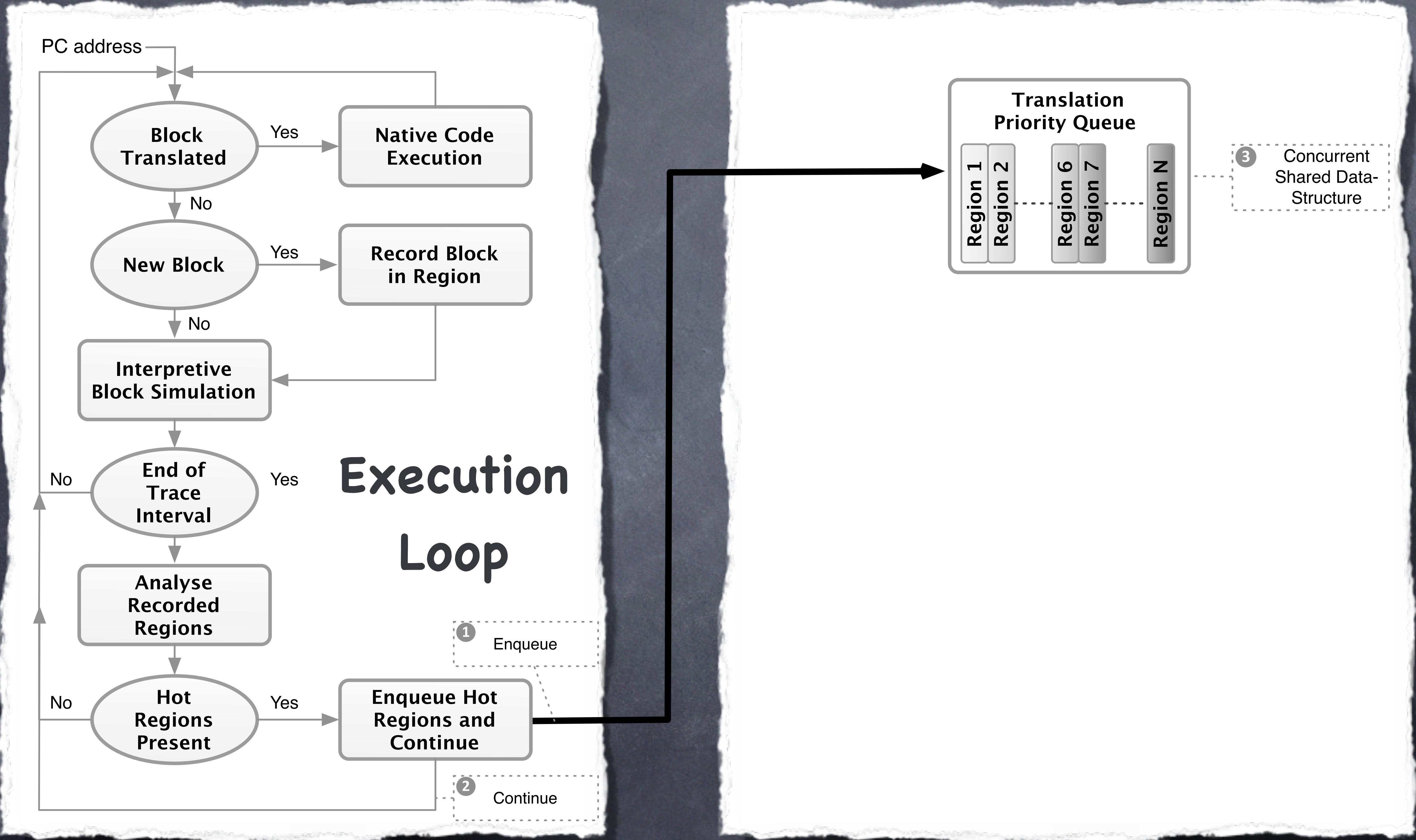
## hide compilation latency

## exploit task parallelism

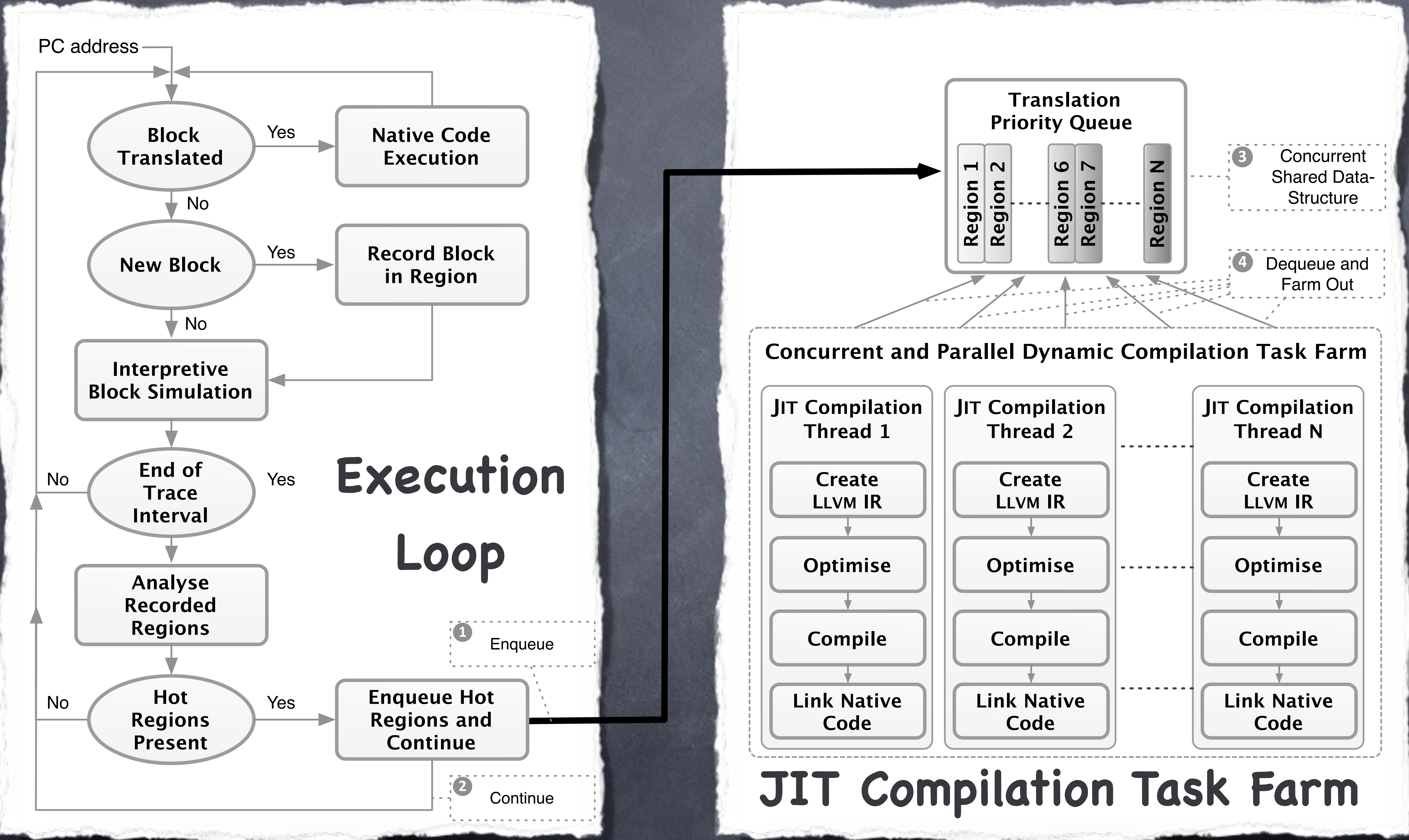
# Concurrent and Parallel JIT Compiler Design



# Concurrent and Parallel JIT Compiler Design



# Concurrent and Parallel JIT Compiler Design



PC address

Block Translated

Yes

Native Code Execution

No

New Block

Yes

Record Block in Region

No

Interpretive Block Simulation

No

End of Trace Interval

Yes

Analyse Recorded Regions

No

Hot Regions Present

Yes

Enqueue Hot Regions and Continue

1 Enqueue

2 Continue

Execution Loop

Translation Priority Queue

Region 1

Region 2

Region 6

Region 7

Region N

3 Concurrent Shared Data-Structure

4 Dequeue and Farm Out

Concurrent and Parallel Dynamic Compilation Task Farm

JIT Compilation Thread 1

Create LLVM IR

Optimise

Compile

Link Native Code

JIT Compilation Thread 2

Create LLVM IR

Optimise

Compile

Link Native Code

JIT Compilation Thread N

Create LLVM IR

Optimise

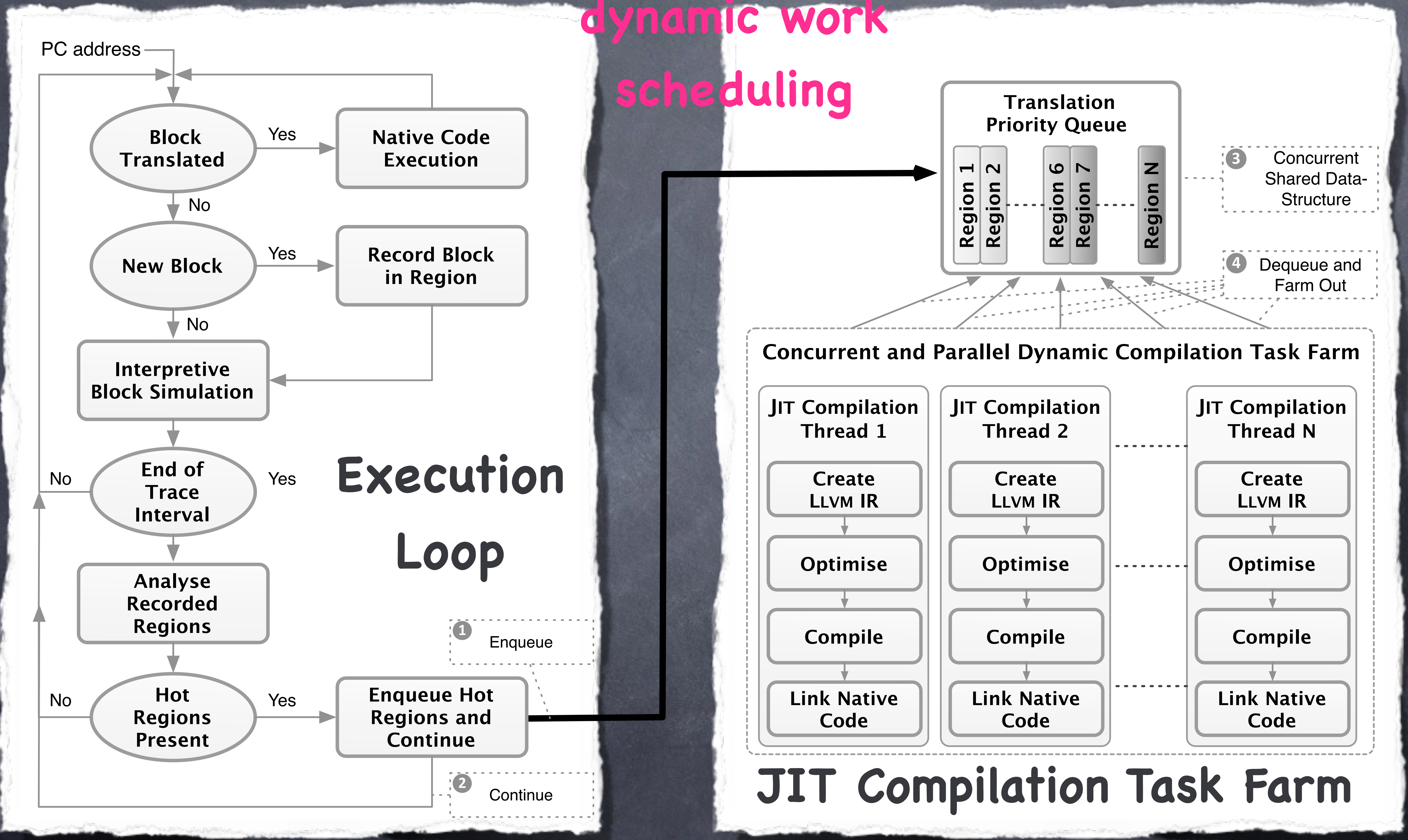
Compile

Link Native Code

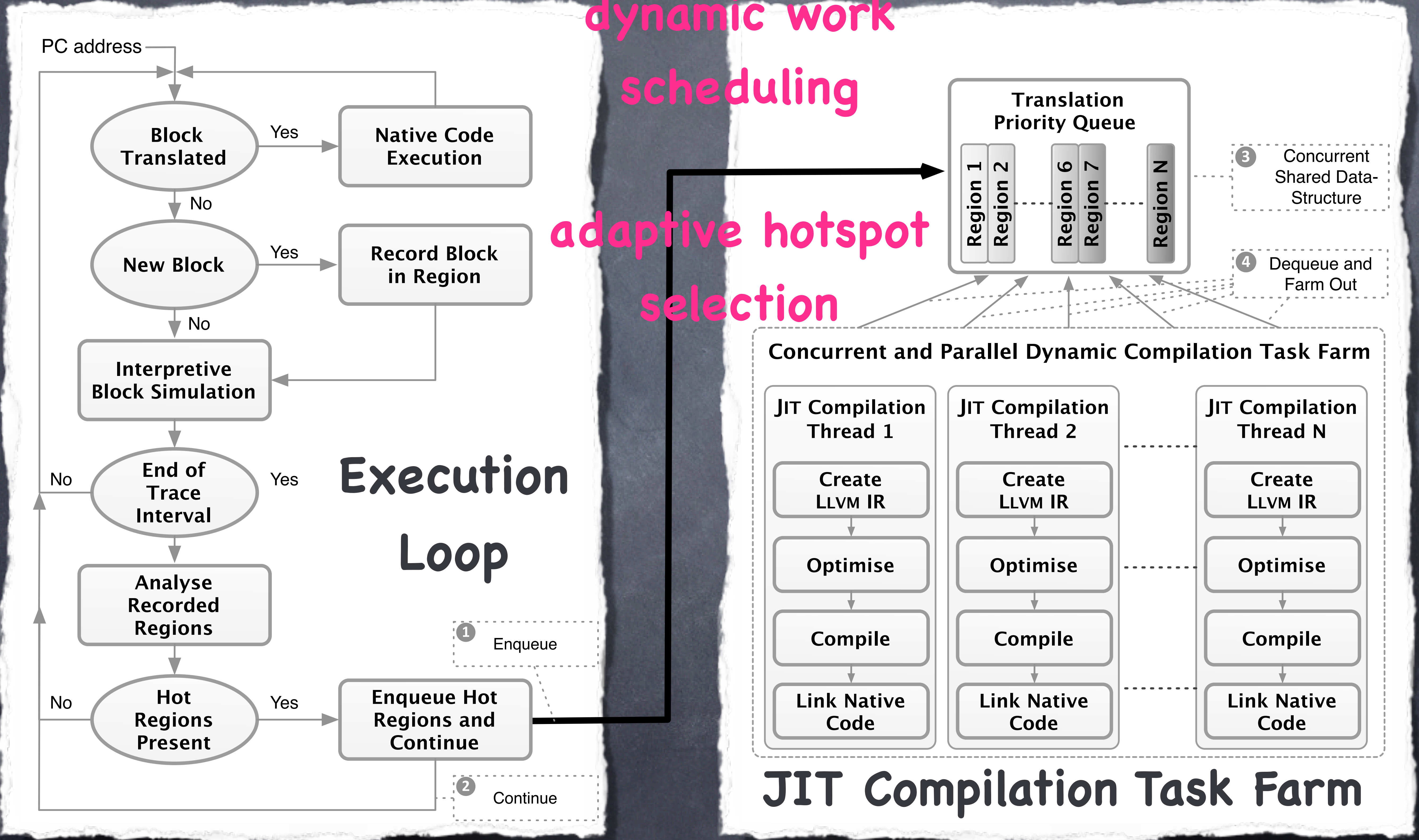
JIT Compilation Task Farm

# Concurrent and Parallel JIT Compiler Design

dynamic work scheduling



# Concurrent and Parallel JIT Compiler Design



dynamic work scheduling

adaptive hotspot selection

# 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();
    ...
  }



}
```

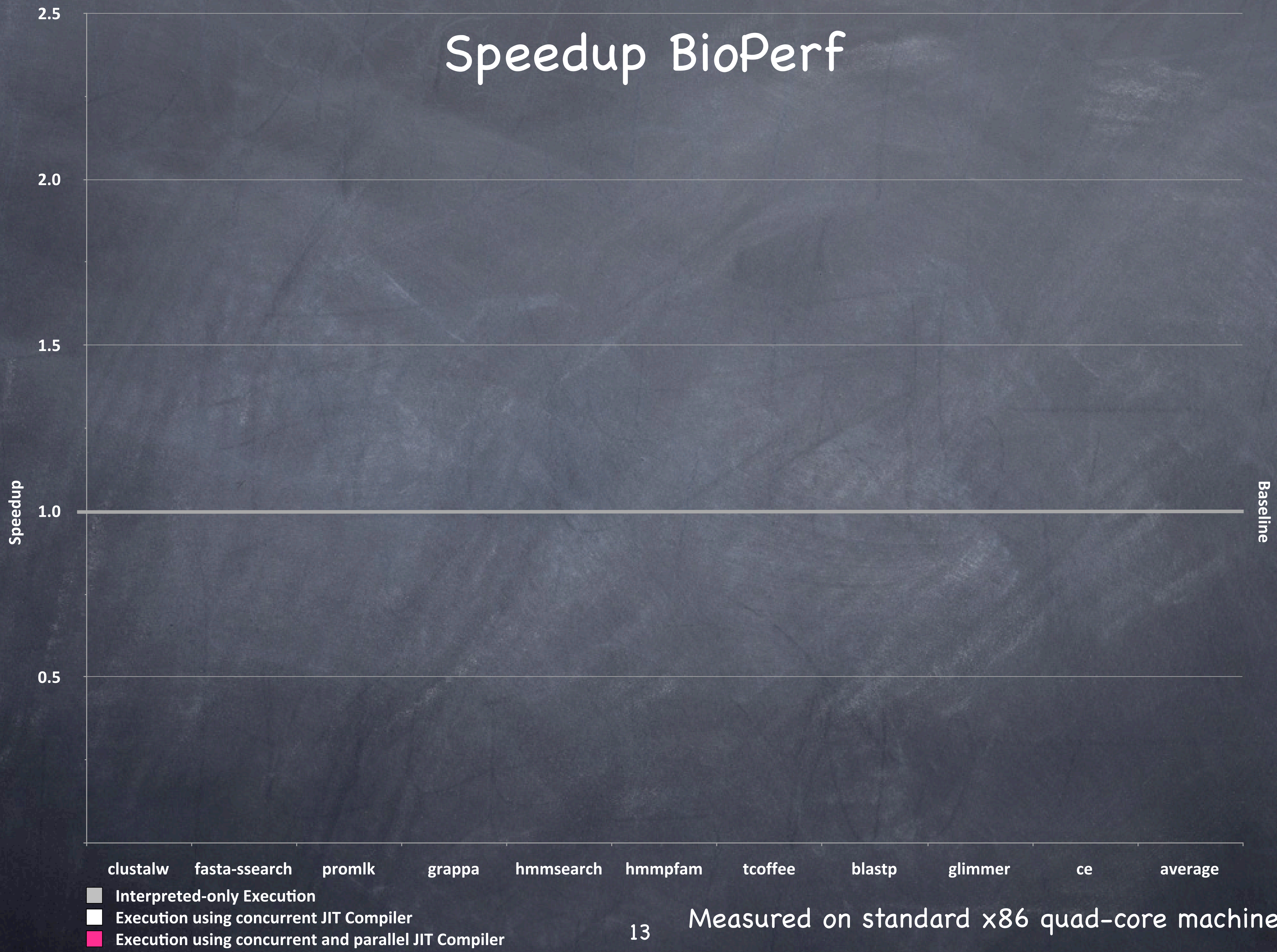
# Concurrent and Parallel JIT Compiler Design Based on LLVM

```
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

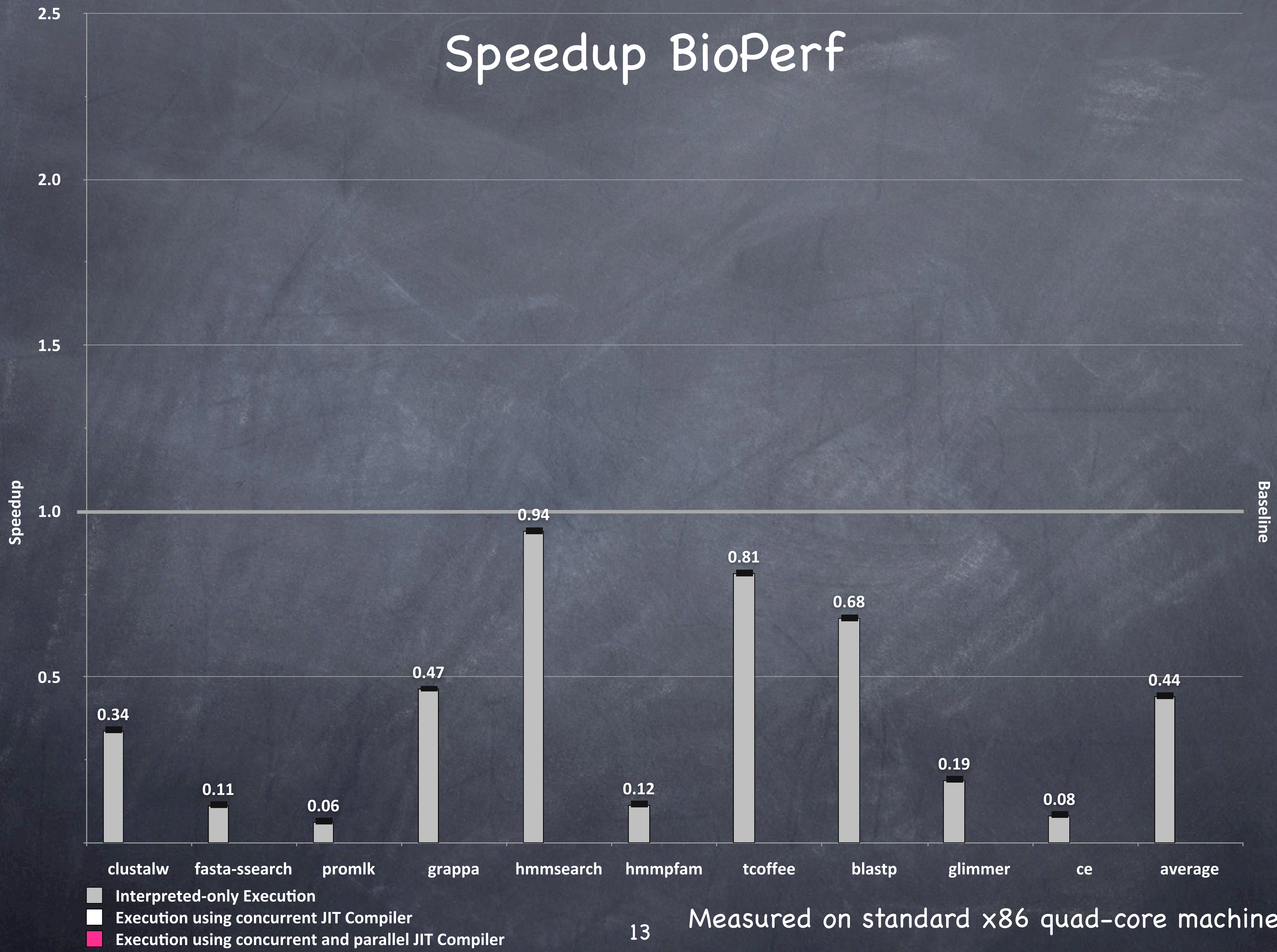
- Extensive evaluation using over 60 industry standard benchmarks built for ARCompact RISC platform:
  - BioPERF
  - SPEC CPU 2006
  - EEMBC and CoreMark
- Target Platform:
  - ARCompact RISC ISA targeting ARC 700 processor
- Simulation Platform:
  - standard x86 Dell Intel Xeon quad-core machine

# Speedup BioPerf



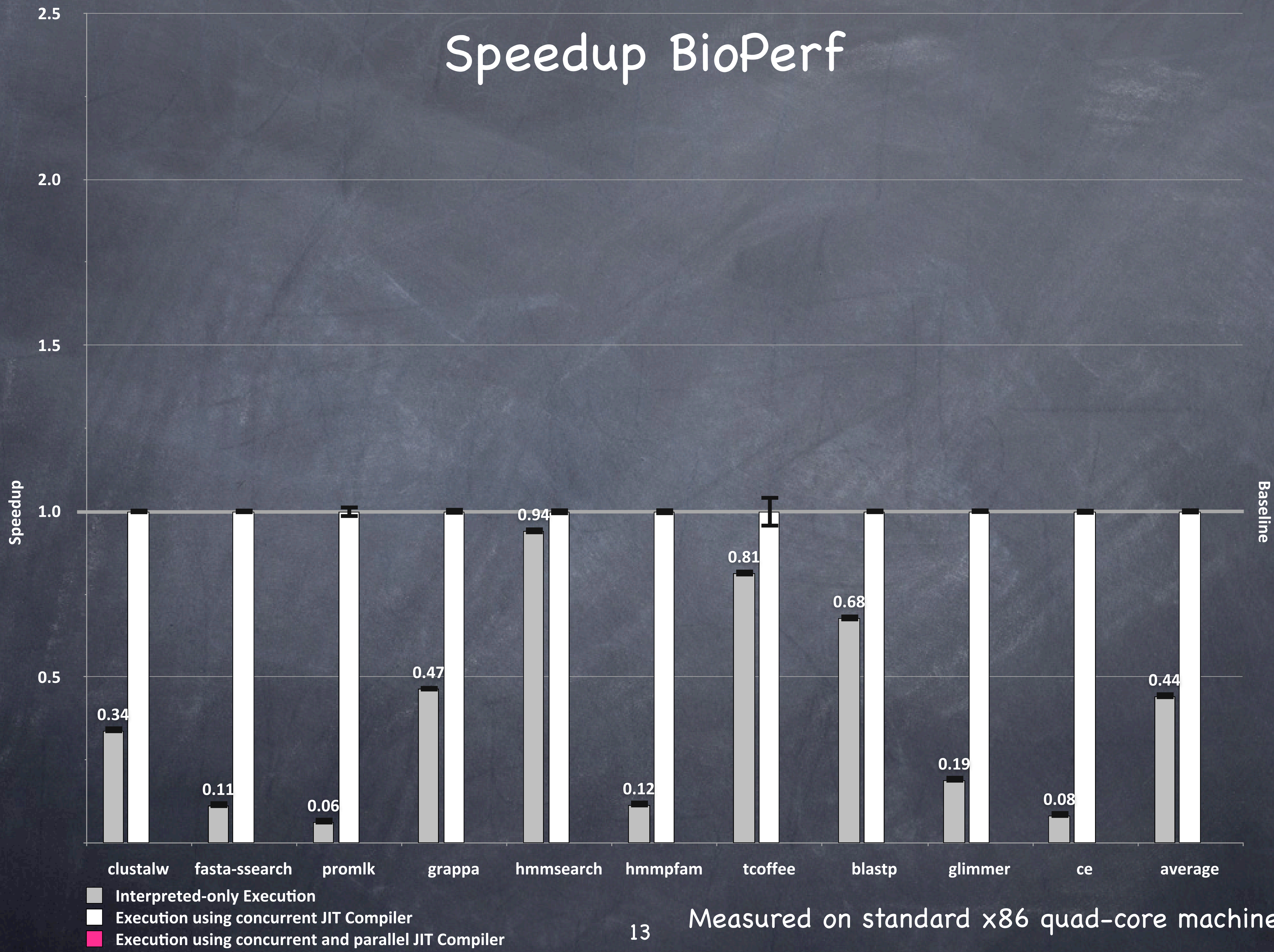
Measured on standard x86 quad-core machine

# Speedup BioPerf



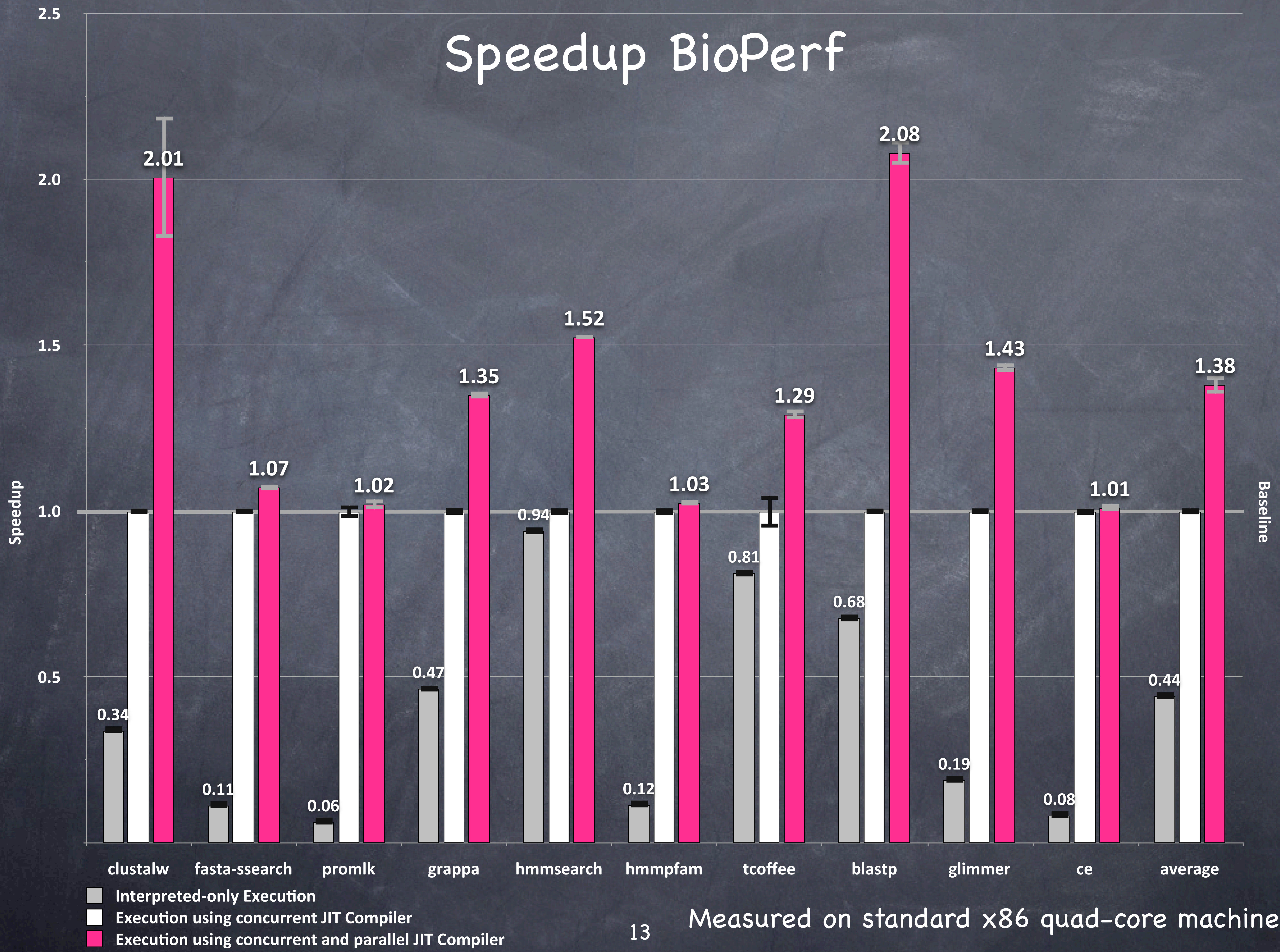
Measured on standard x86 quad-core machine

# Speedup BioPerf



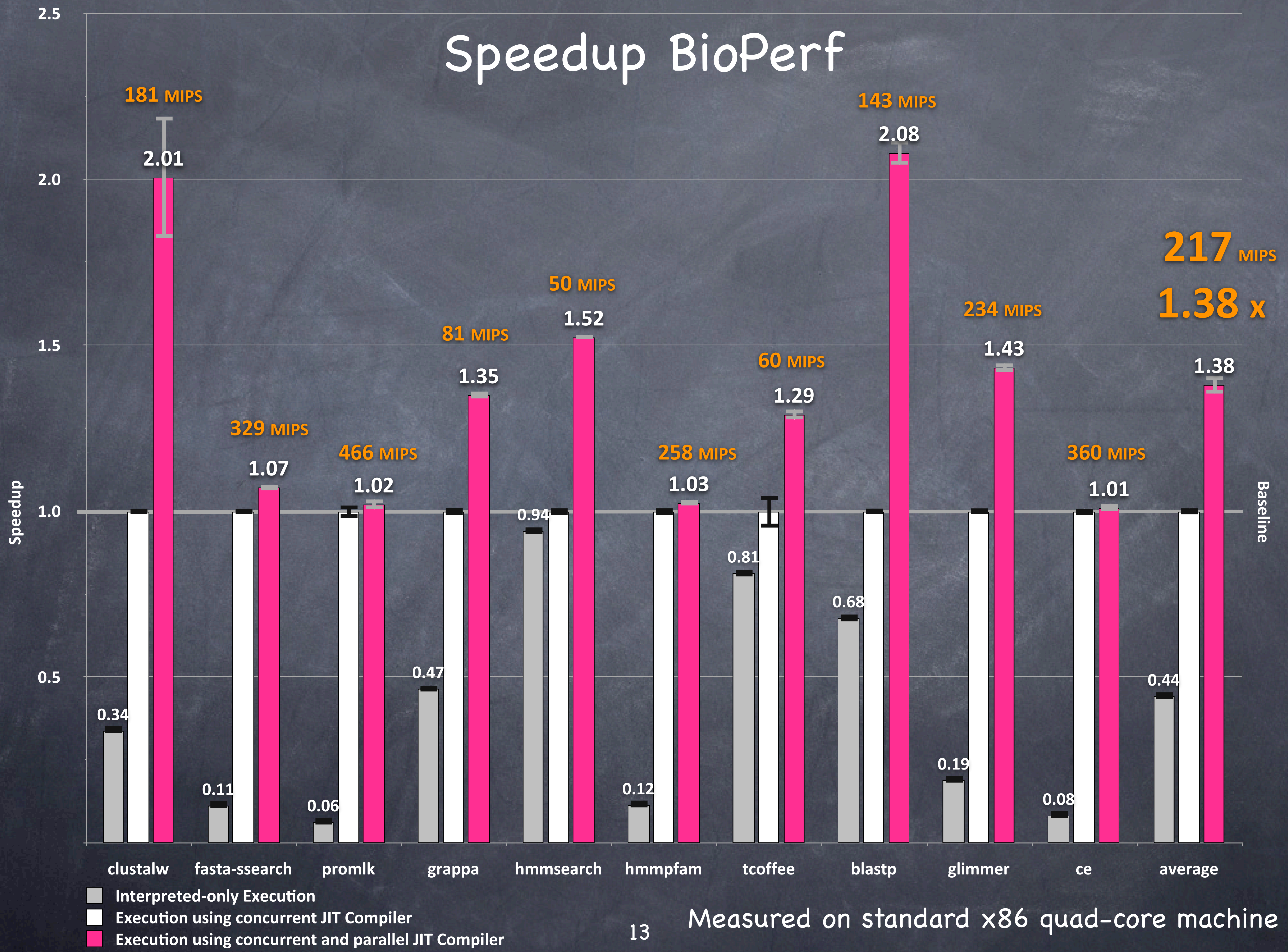
Measured on standard x86 quad-core machine

# Speedup BioPerf



Measured on standard x86 quad-core machine

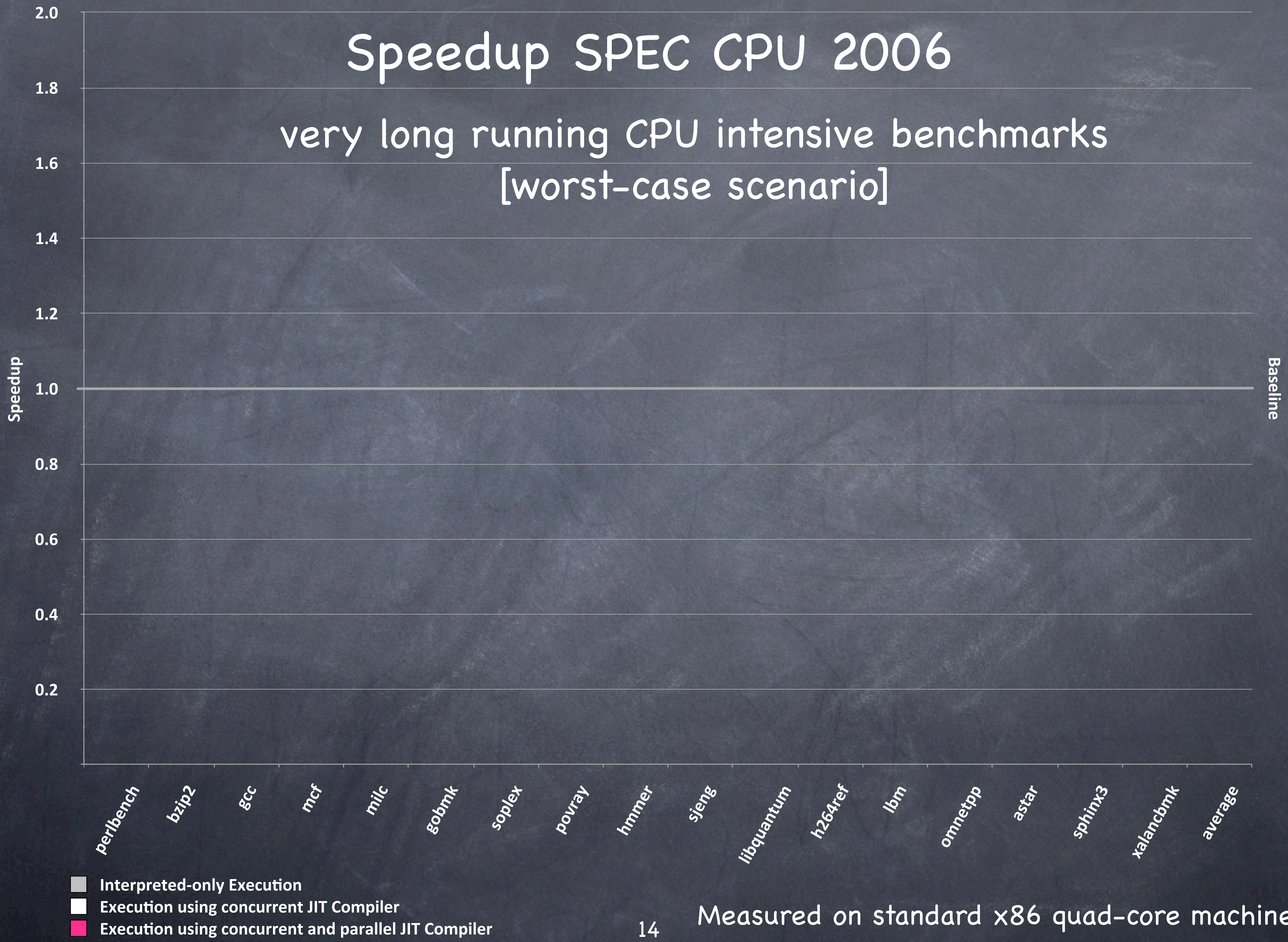
# Speedup BioPerf



Measured on standard x86 quad-core machine

# Speedup SPEC CPU 2006

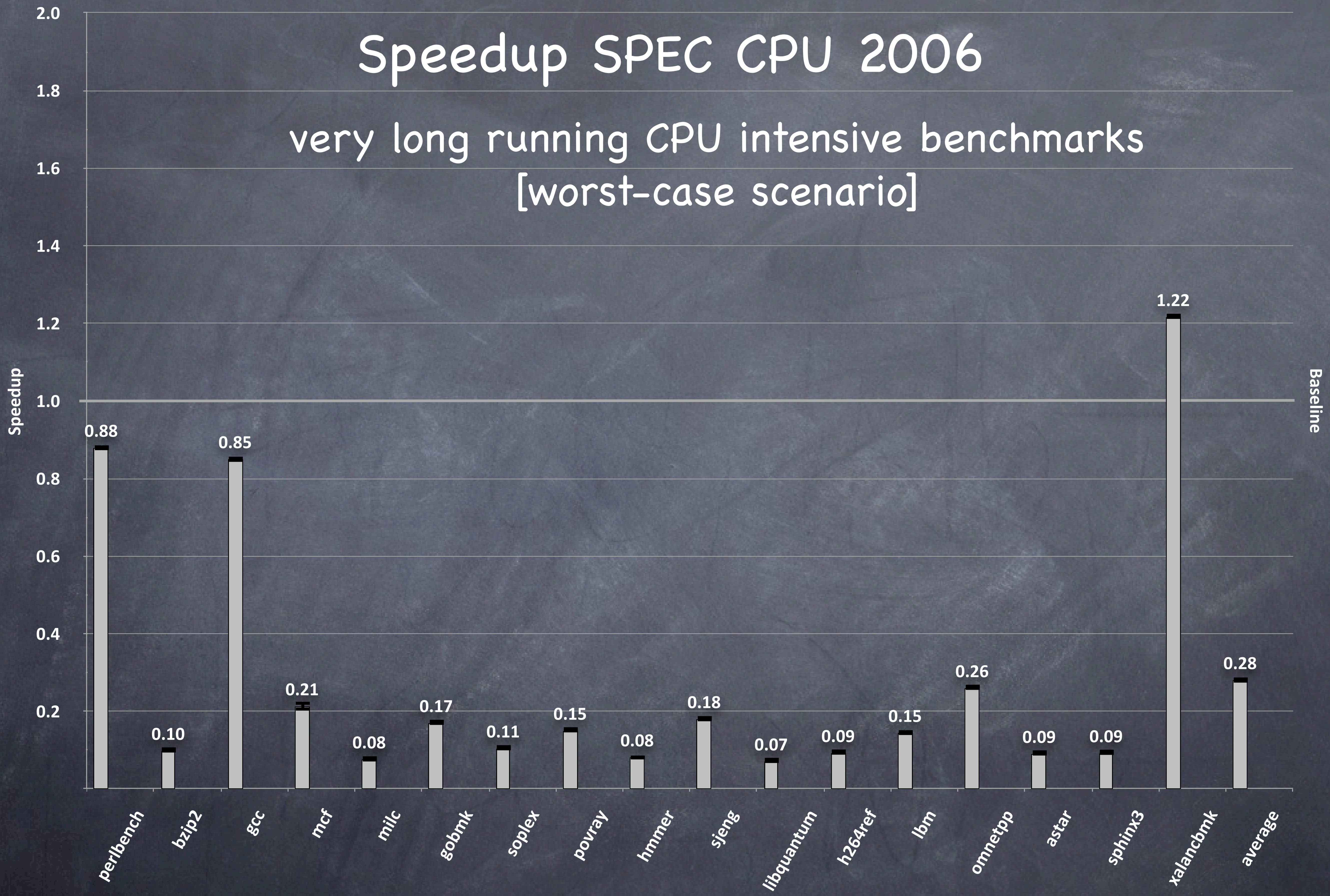
very long running CPU intensive benchmarks
[worst-case scenario]



Measured on standard x86 quad-core machine

# Speedup SPEC CPU 2006

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

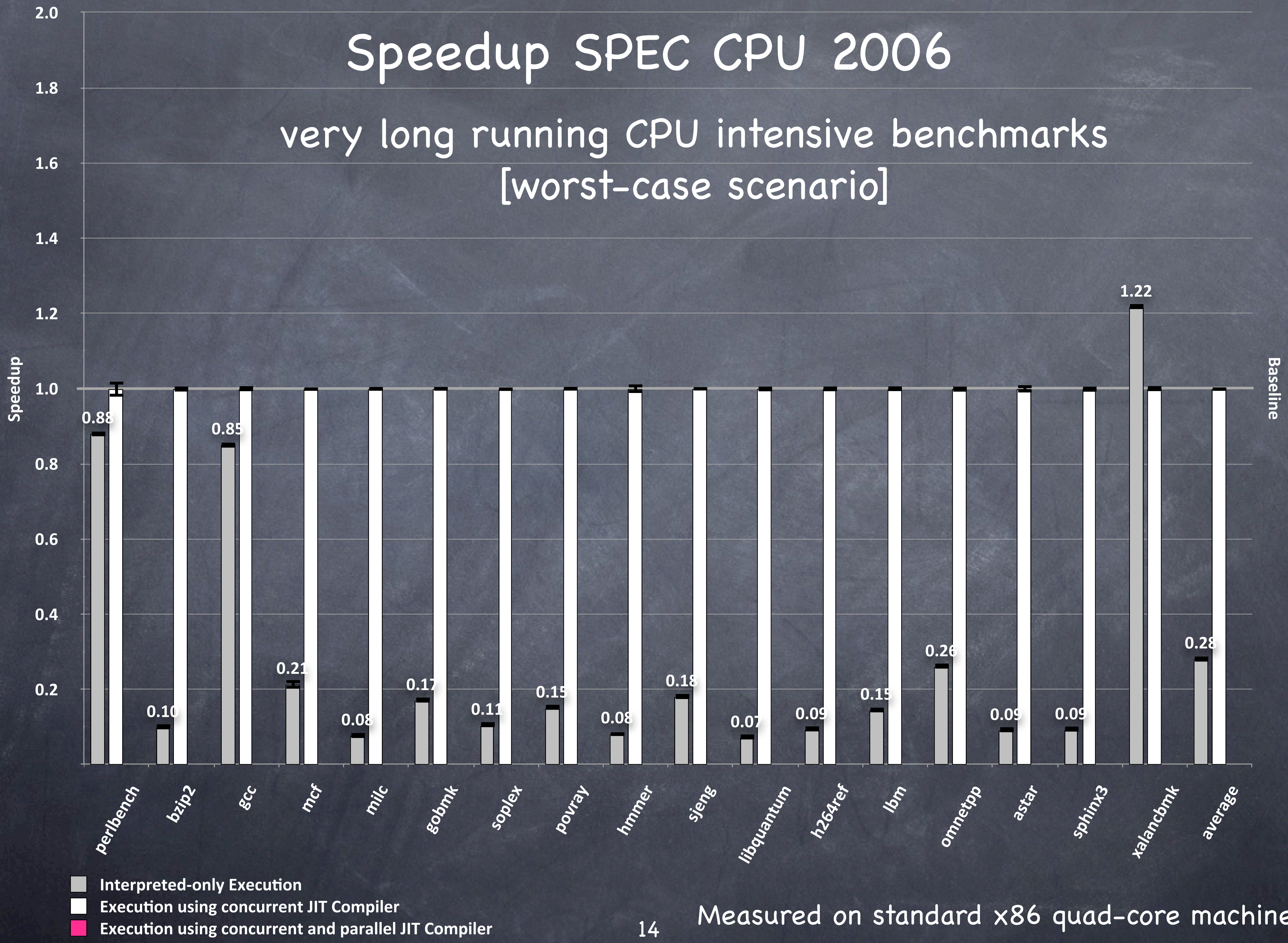


▨ Interpreted-only Execution
□ Execution using concurrent JIT Compiler
🟥 Execution using concurrent and parallel JIT Compiler

Measured on standard x86 quad-core machine

# Speedup SPEC CPU 2006

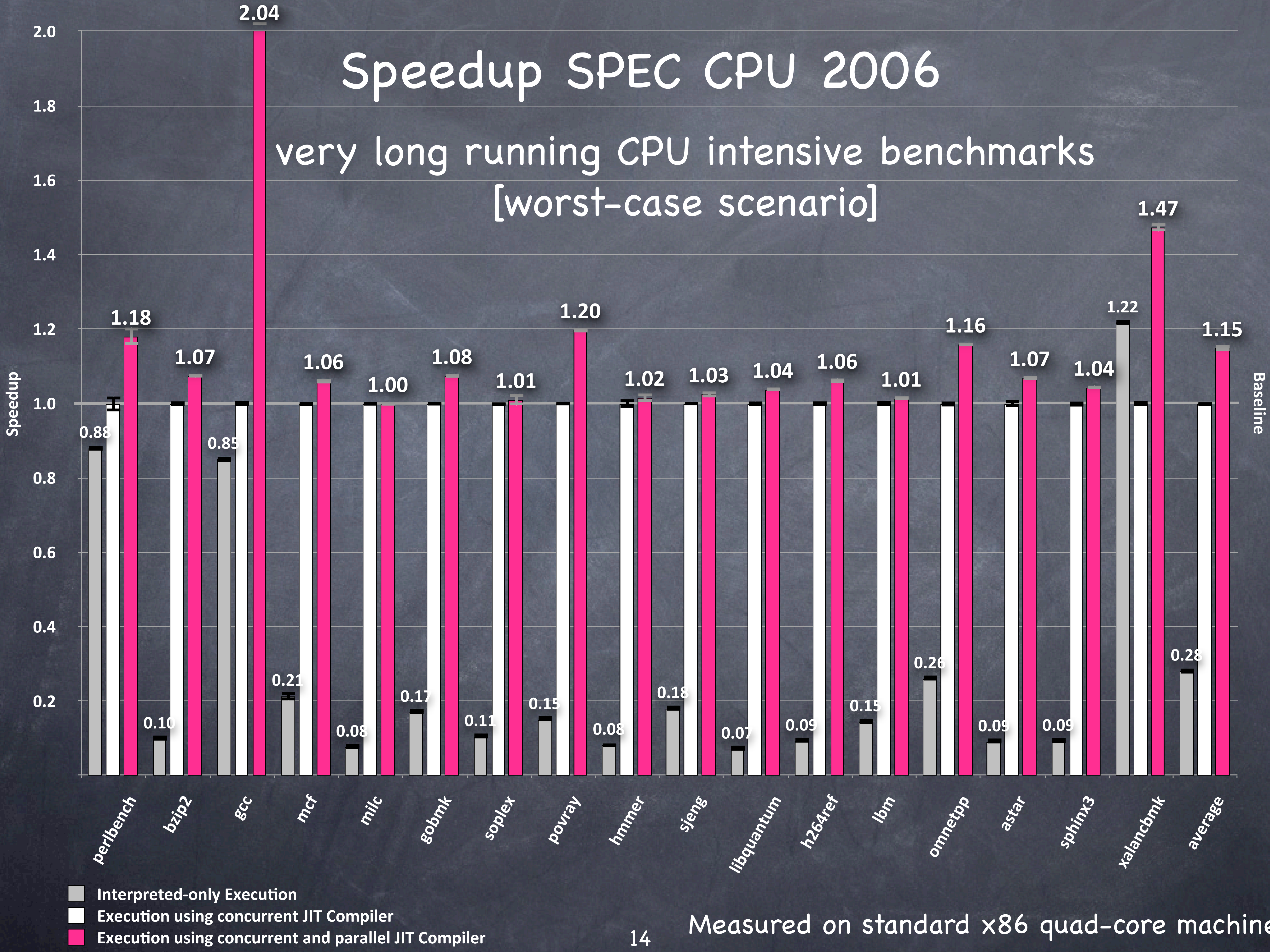
very long running CPU intensive benchmarks
[worst-case scenario]



Measured on standard x86 quad-core machine

# Speedup SPEC CPU 2006

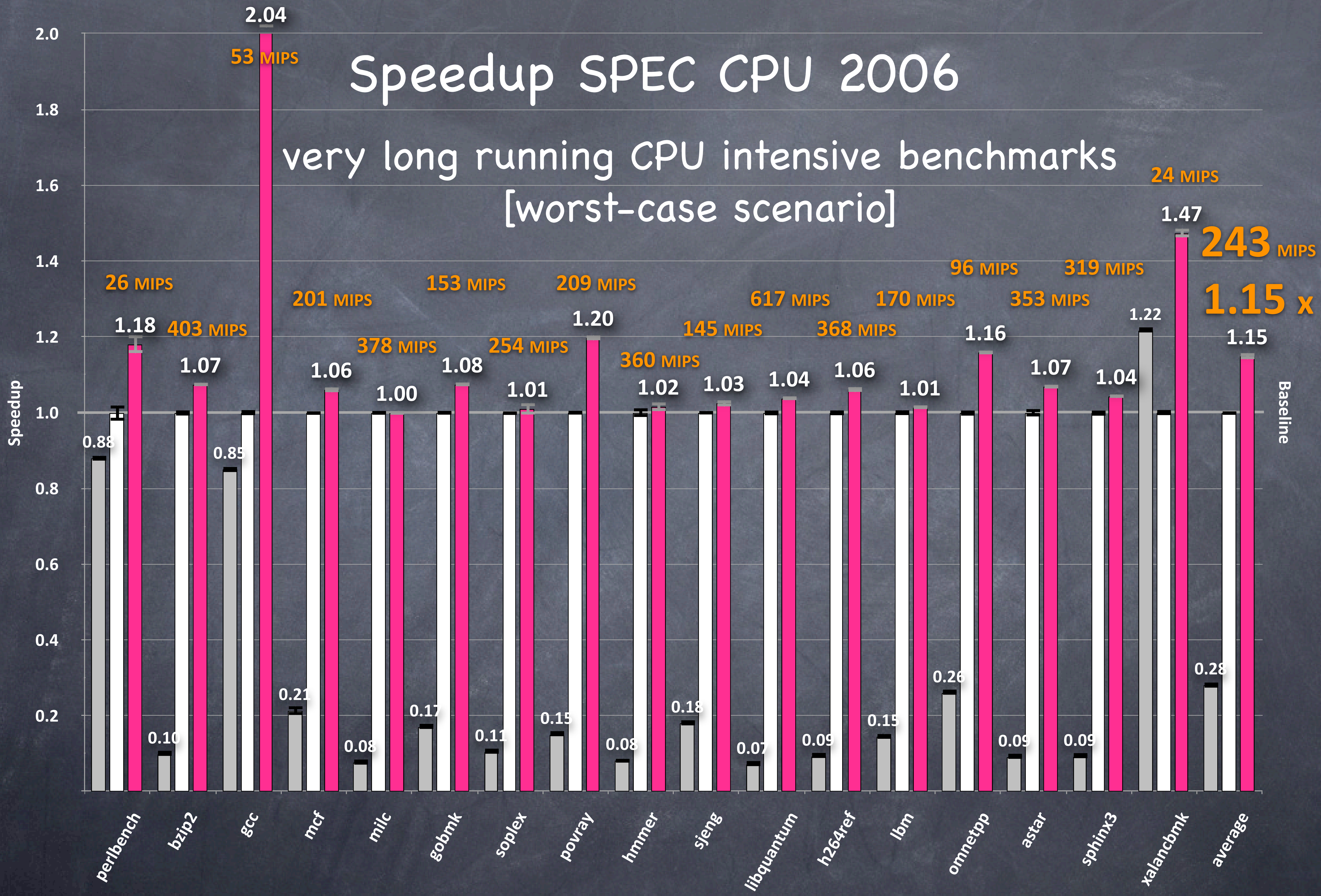
very long running CPU intensive benchmarks
[worst-case scenario]



Measured on standard x86 quad-core machine

# Speedup SPEC CPU 2006

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

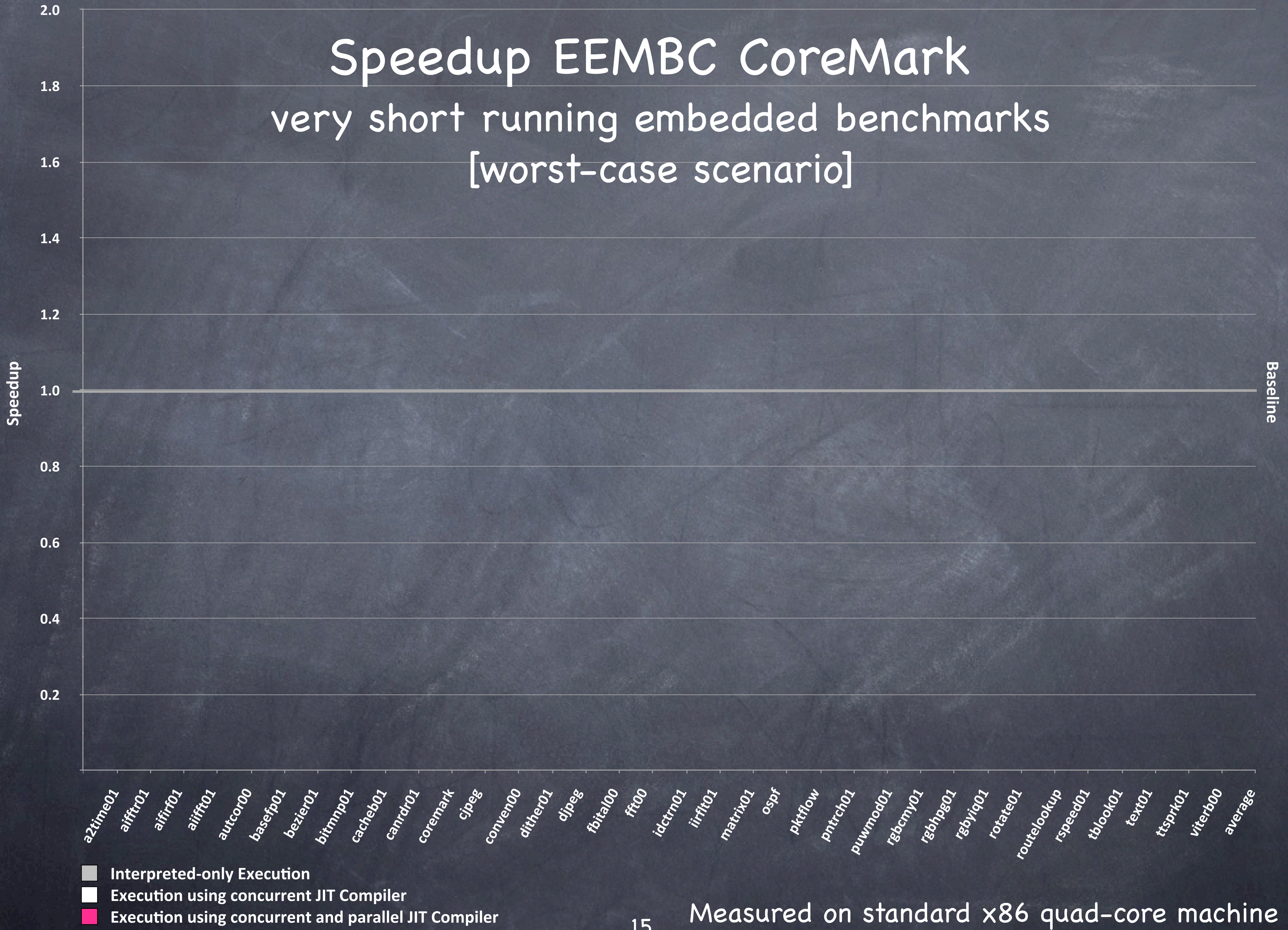


- ▨ Interpreted-only Execution
- □ Execution using concurrent JIT Compiler
- ■ Execution using concurrent and parallel JIT Compiler

Measured on standard x86 quad-core machine

# Speedup EEMBC CoreMark

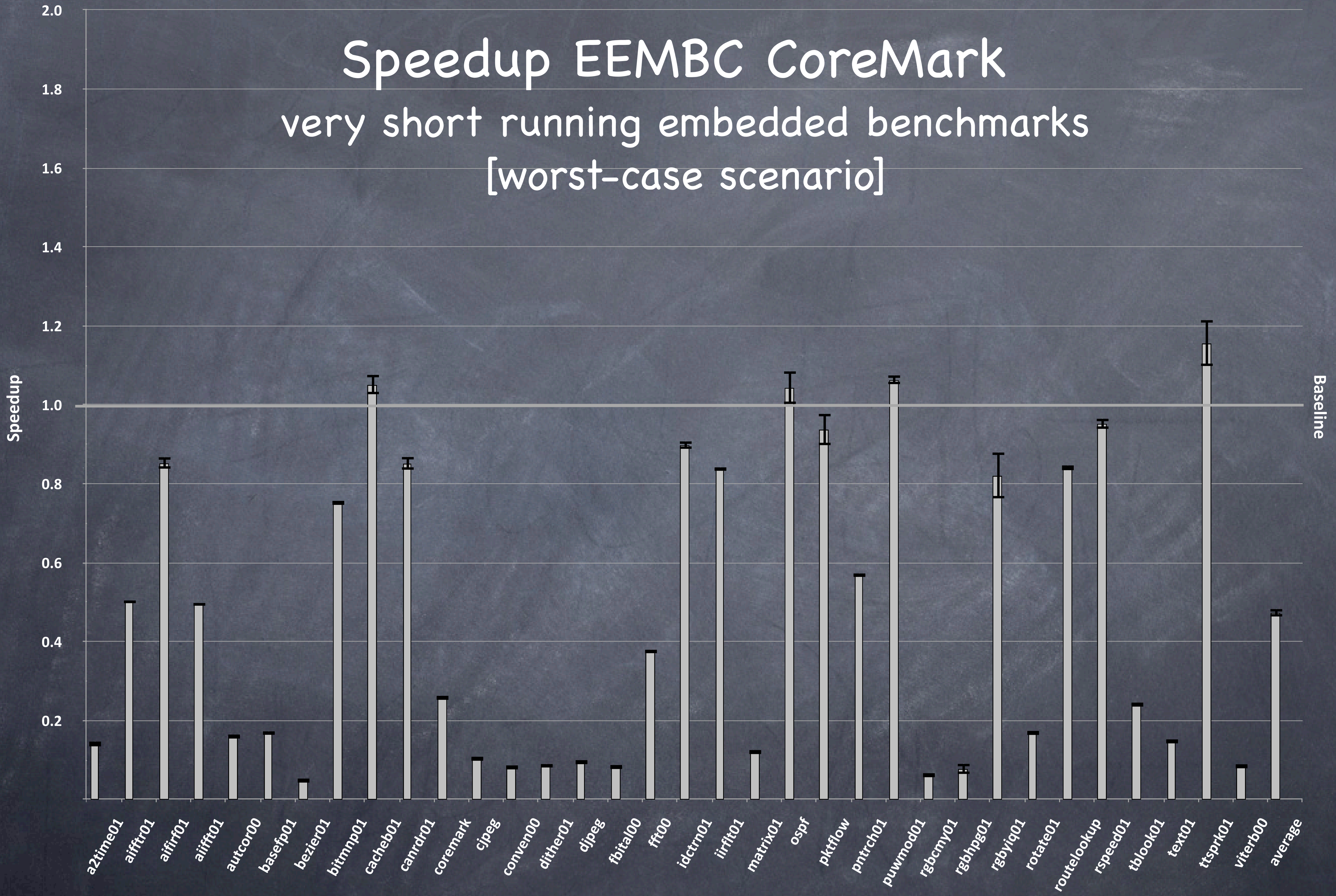
very short running embedded benchmarks
[worst-case scenario]



Measured on standard x86 quad-core machine

# Speedup EEMBC CoreMark

## very short running embedded benchmarks [worst-case scenario]

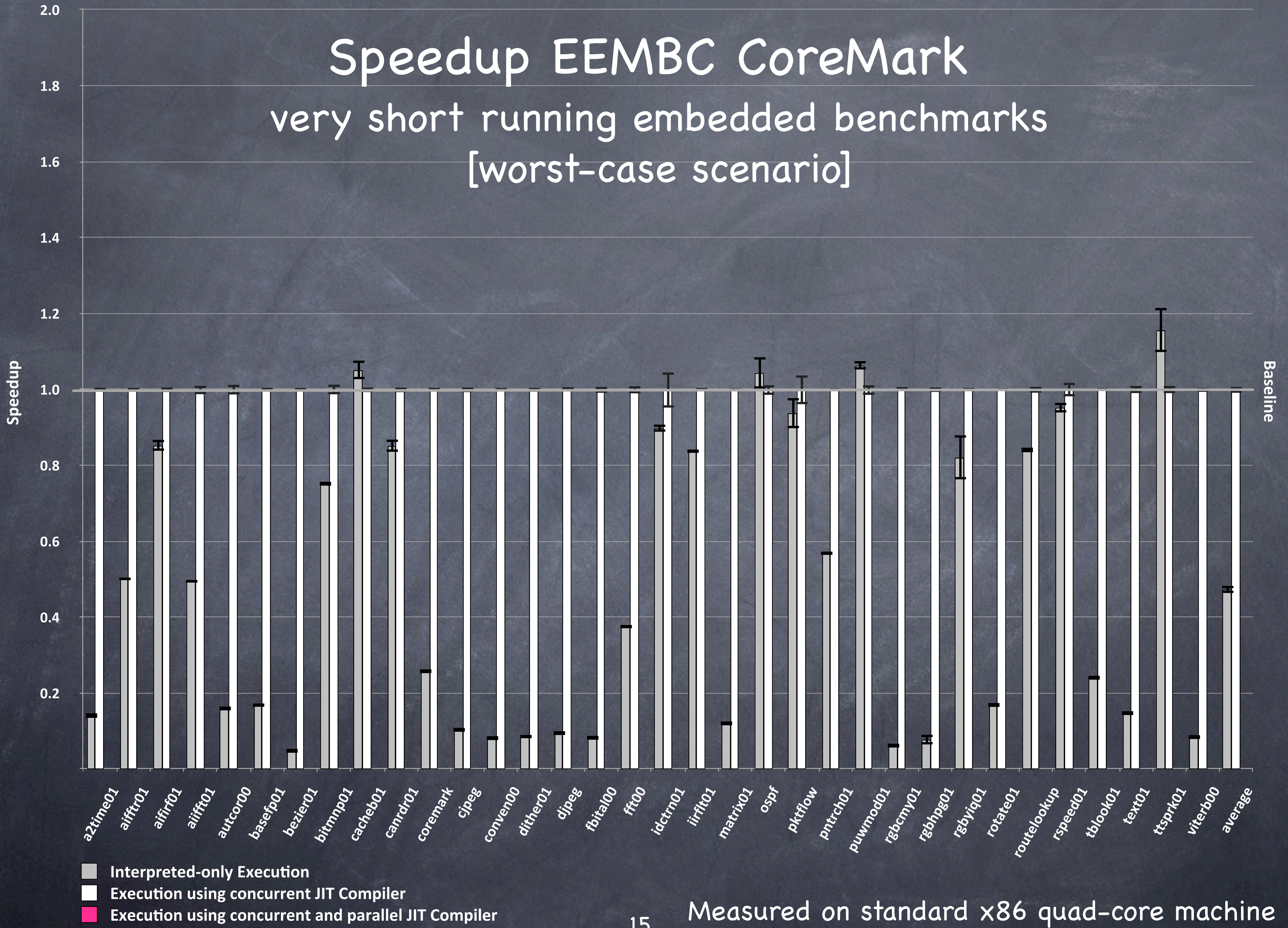


▨ Interpreted-only Execution
□ Execution using concurrent JIT Compiler
■ Execution using concurrent and parallel JIT Compiler

Measured on standard x86 quad-core machine

# Speedup EEMBC CoreMark

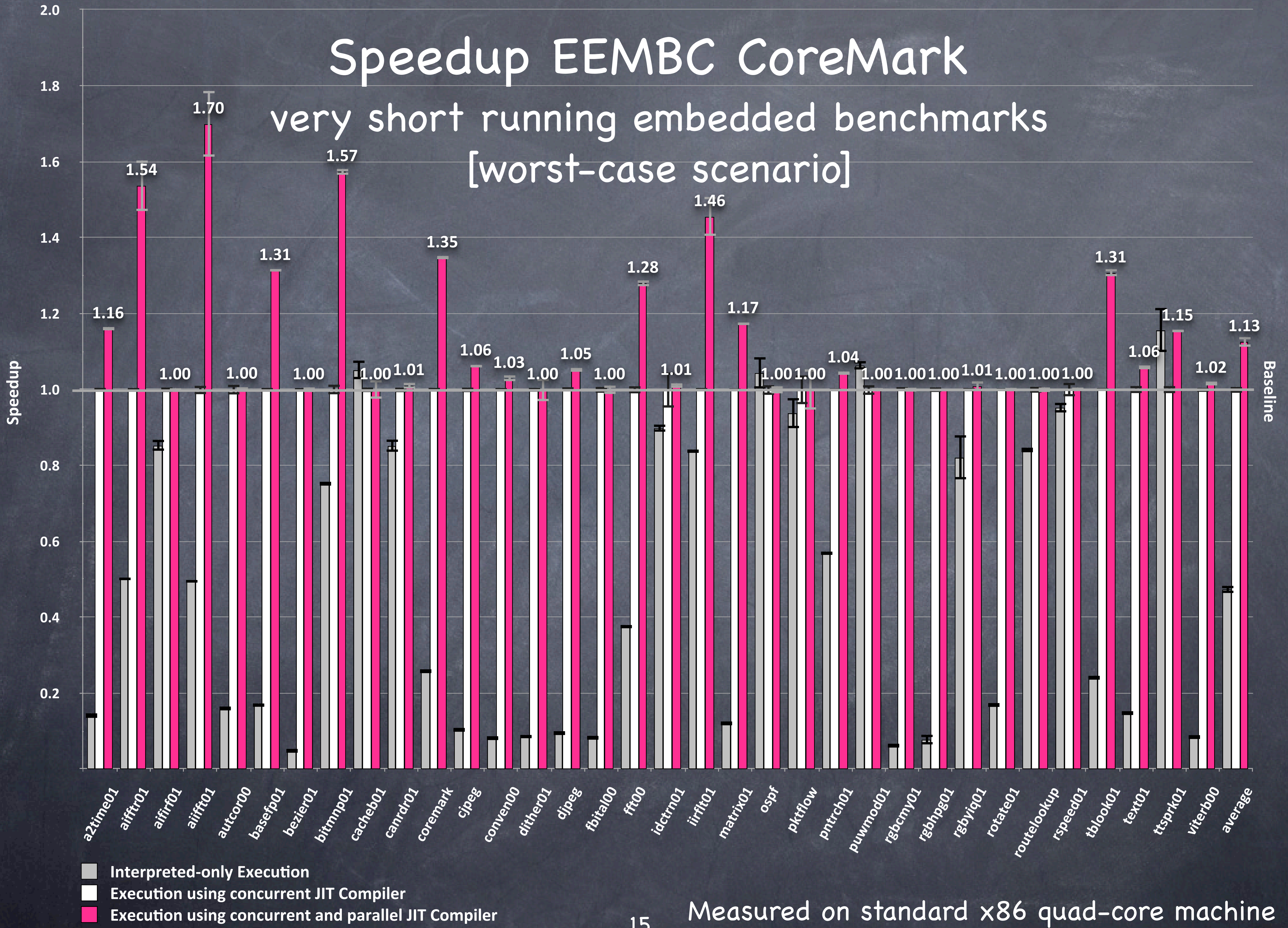
## very short running embedded benchmarks [worst-case scenario]



Measured on standard x86 quad-core machine

# Speedup EEMBC CoreMark

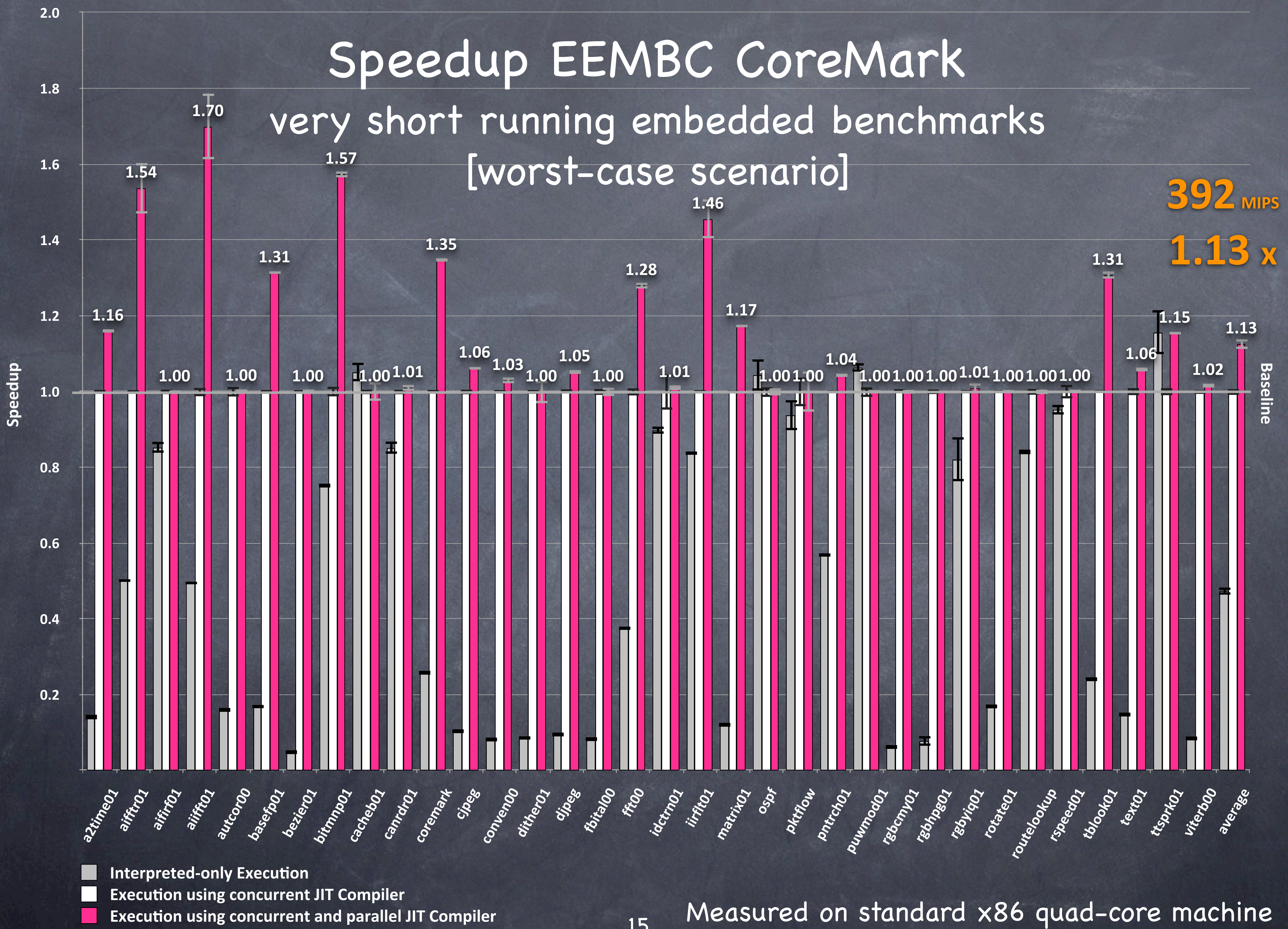
## very short running embedded benchmarks [worst-case scenario]



Measured on standard x86 quad-core machine

# Speedup EEMBC CoreMark

## very short running embedded benchmarks [worst-case scenario]



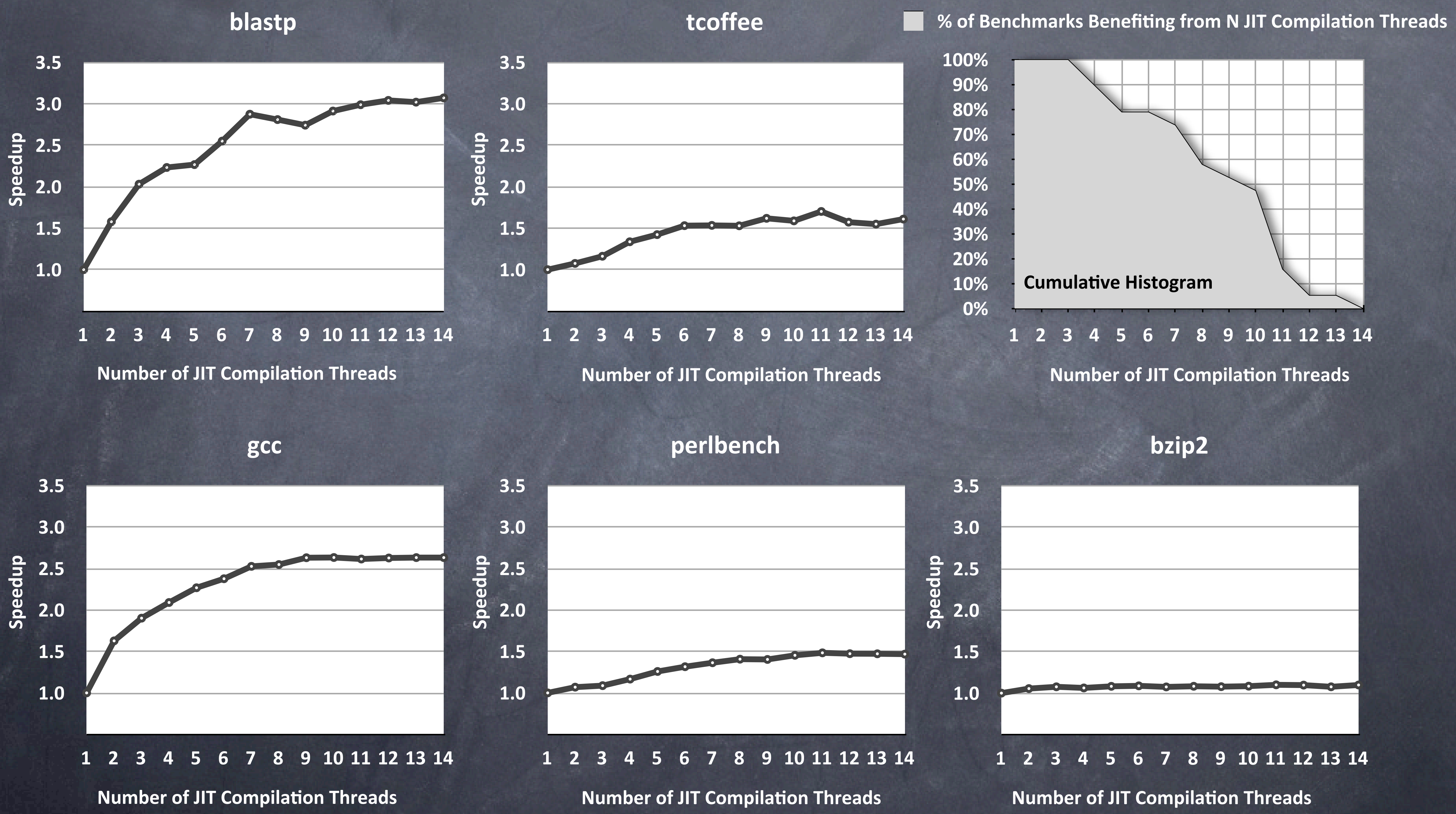
392 MIPS

1.13 x

Measured on standard x86 quad-core machine

# How far does it scale?

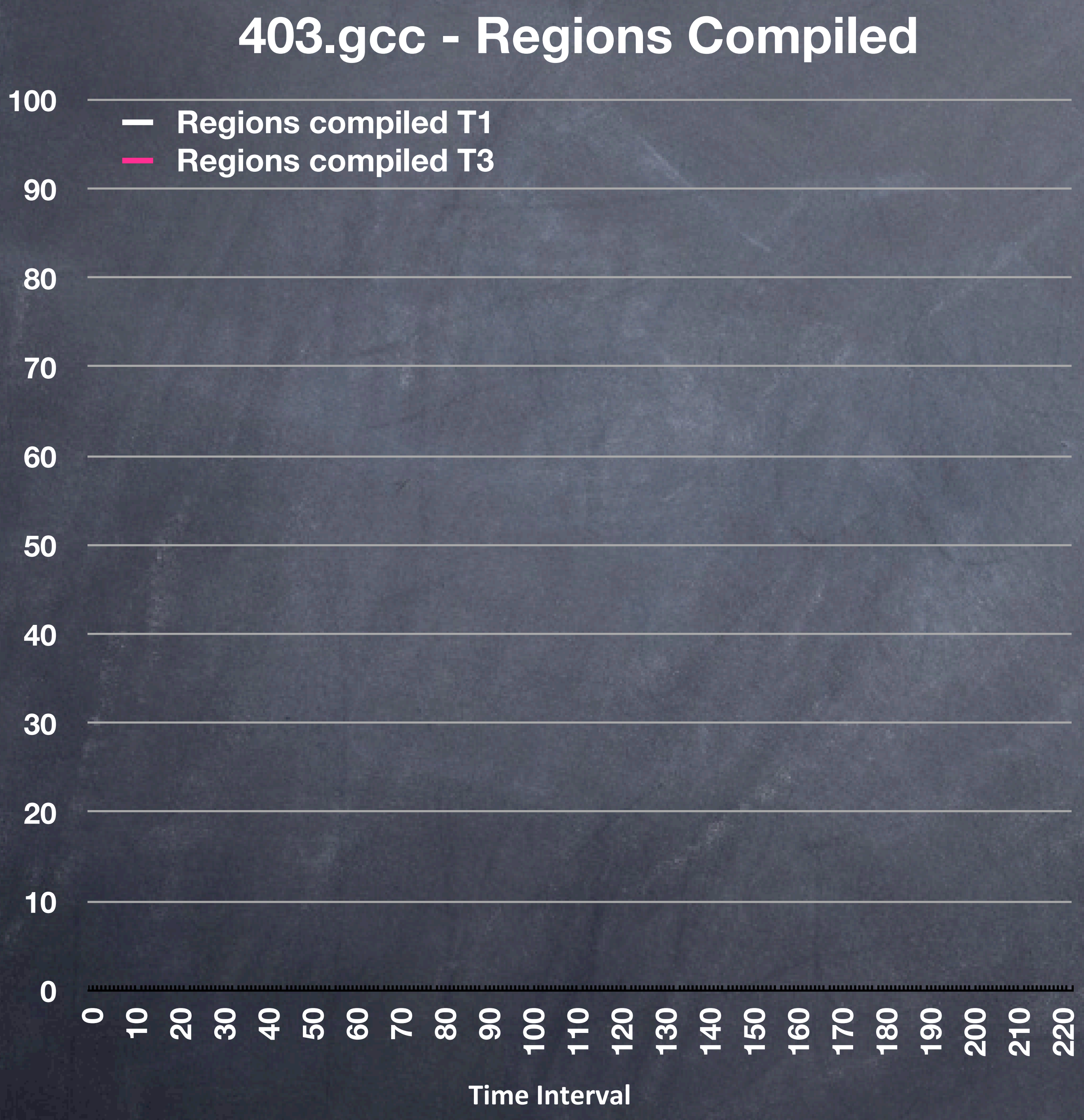
## What is a sensible number of JIT compilation threads?



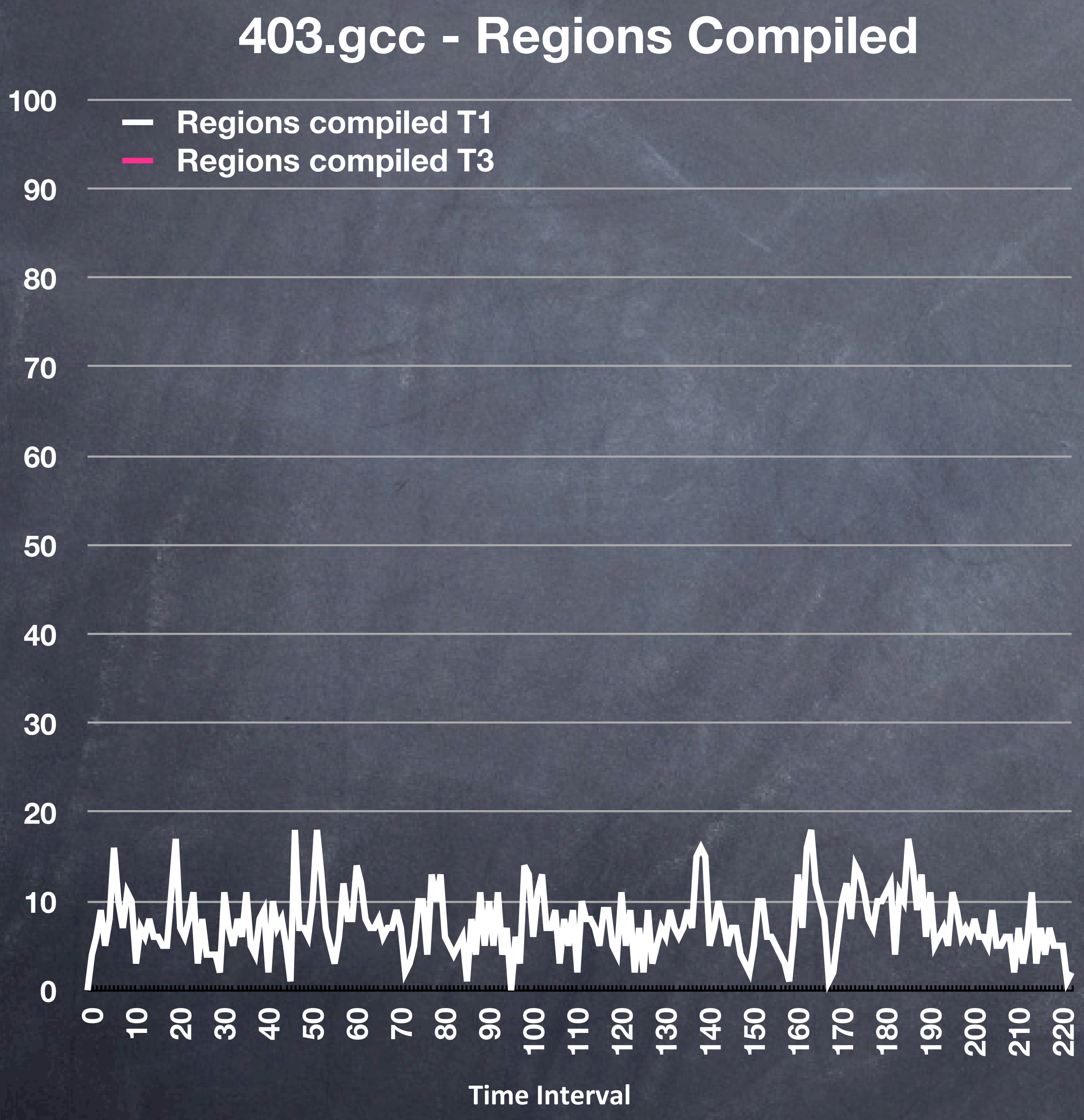
Measured on a 16-core machine

# Effect of Concurrent and Parallel JIT Compilation on Throughput

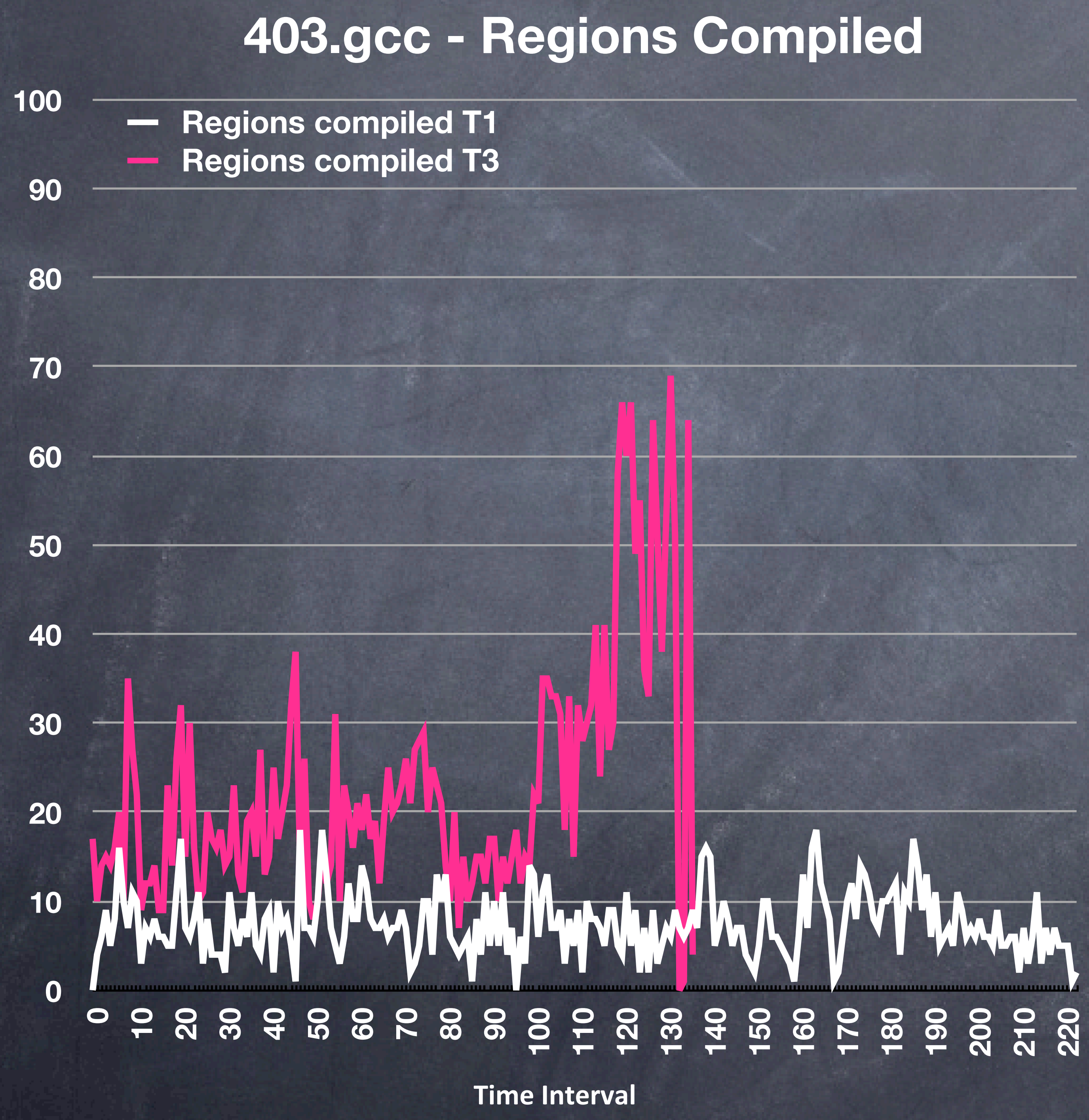
# Effect of Concurrent and Parallel JIT Compilation on Throughput



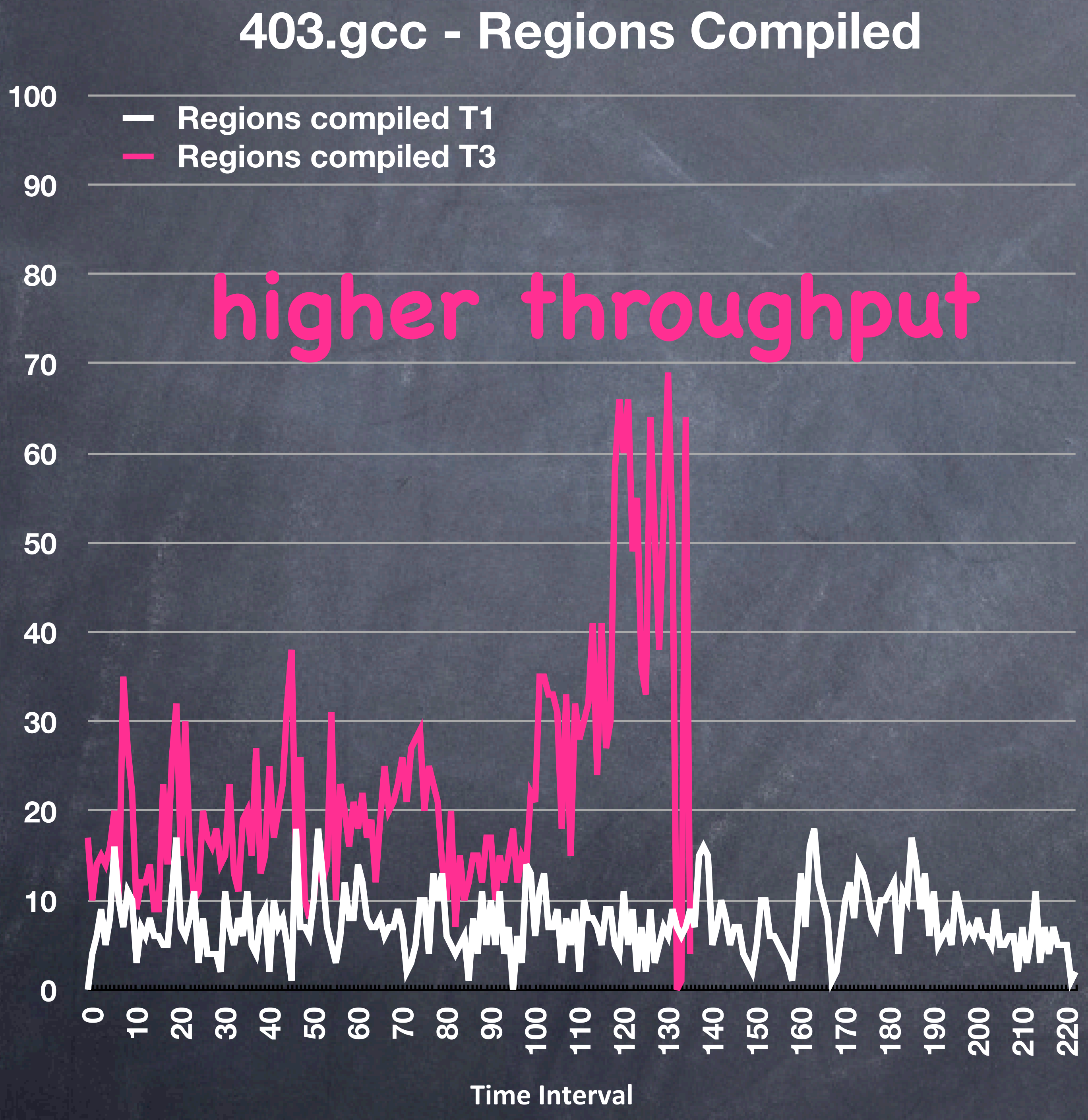
# Effect of Concurrent and Parallel JIT Compilation on Throughput



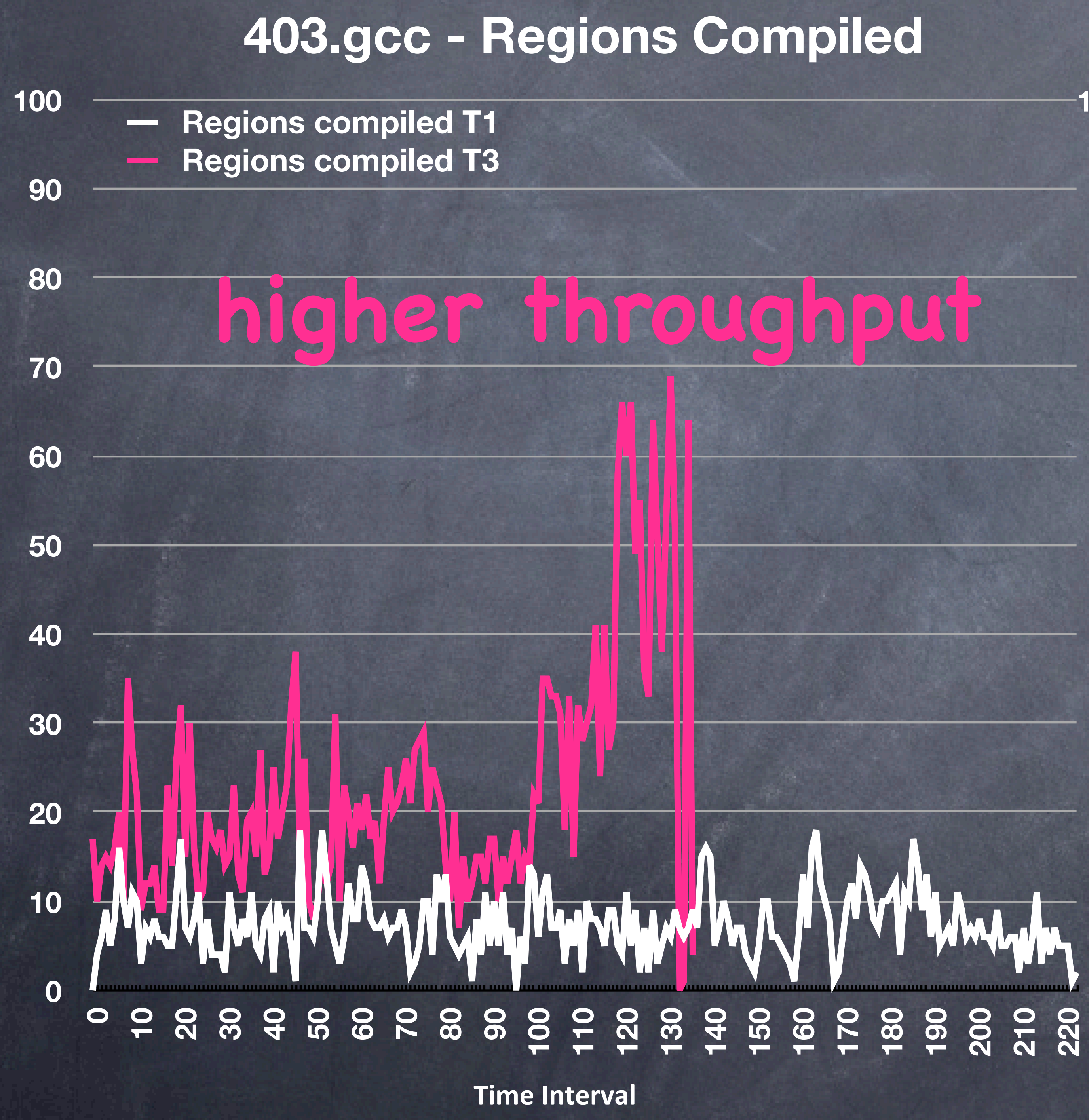
# Effect of Concurrent and Parallel JIT Compilation on Throughput

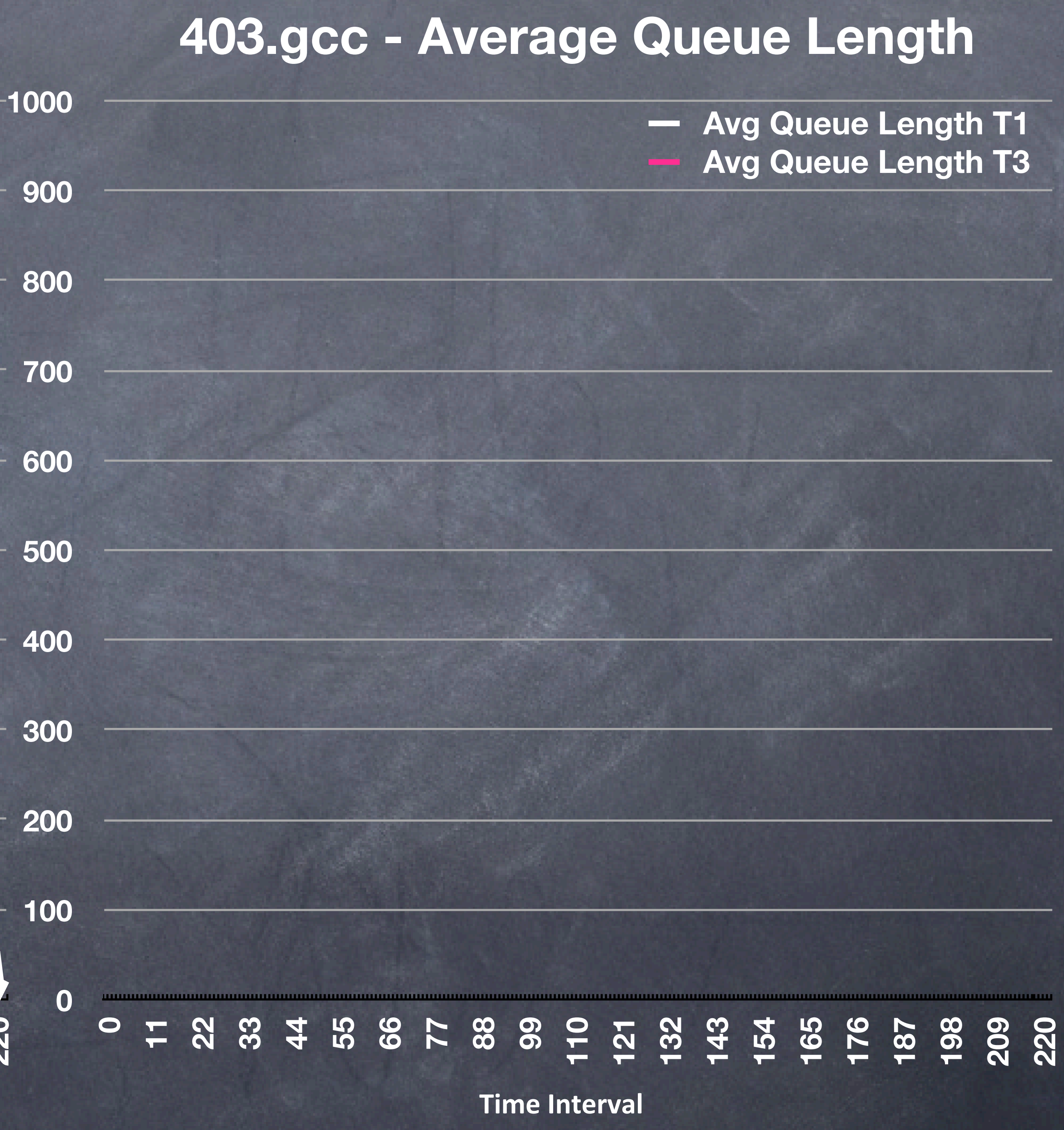


# Effect of Concurrent and Parallel JIT Compilation on Throughput

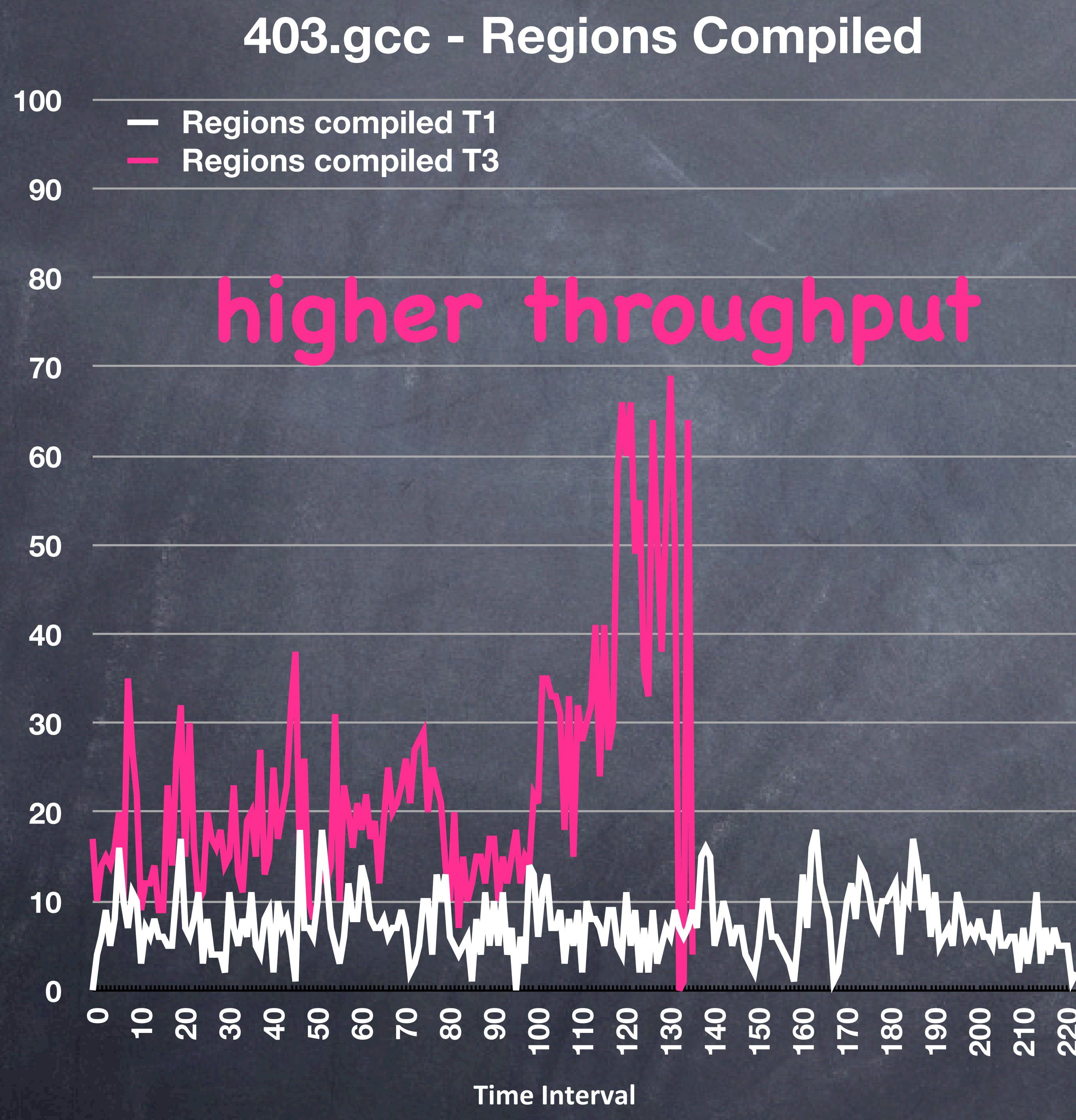


# Effect of Concurrent and Parallel JIT Compilation on Throughput

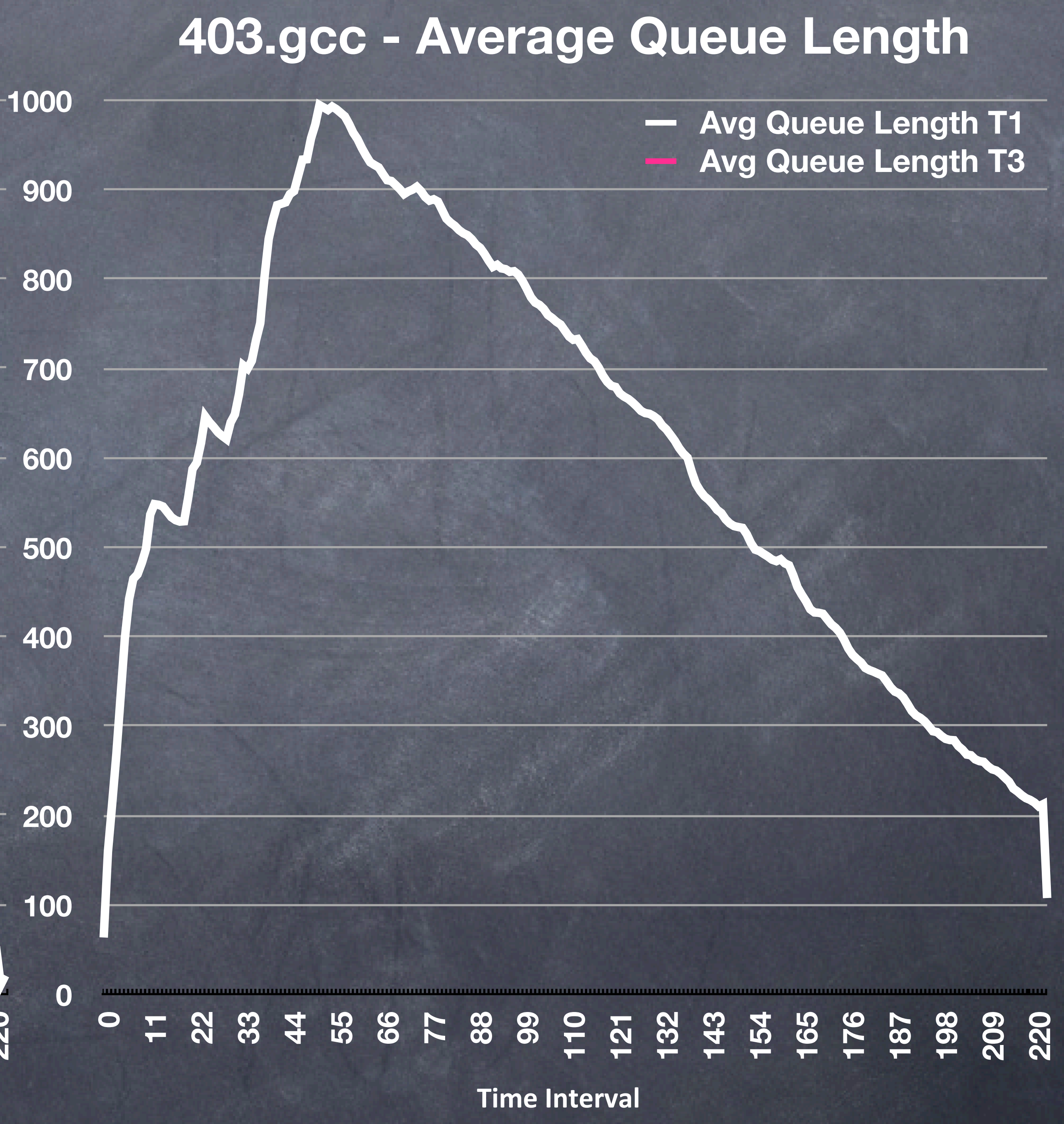




# Effect of Concurrent and Parallel JIT Compilation on Throughput

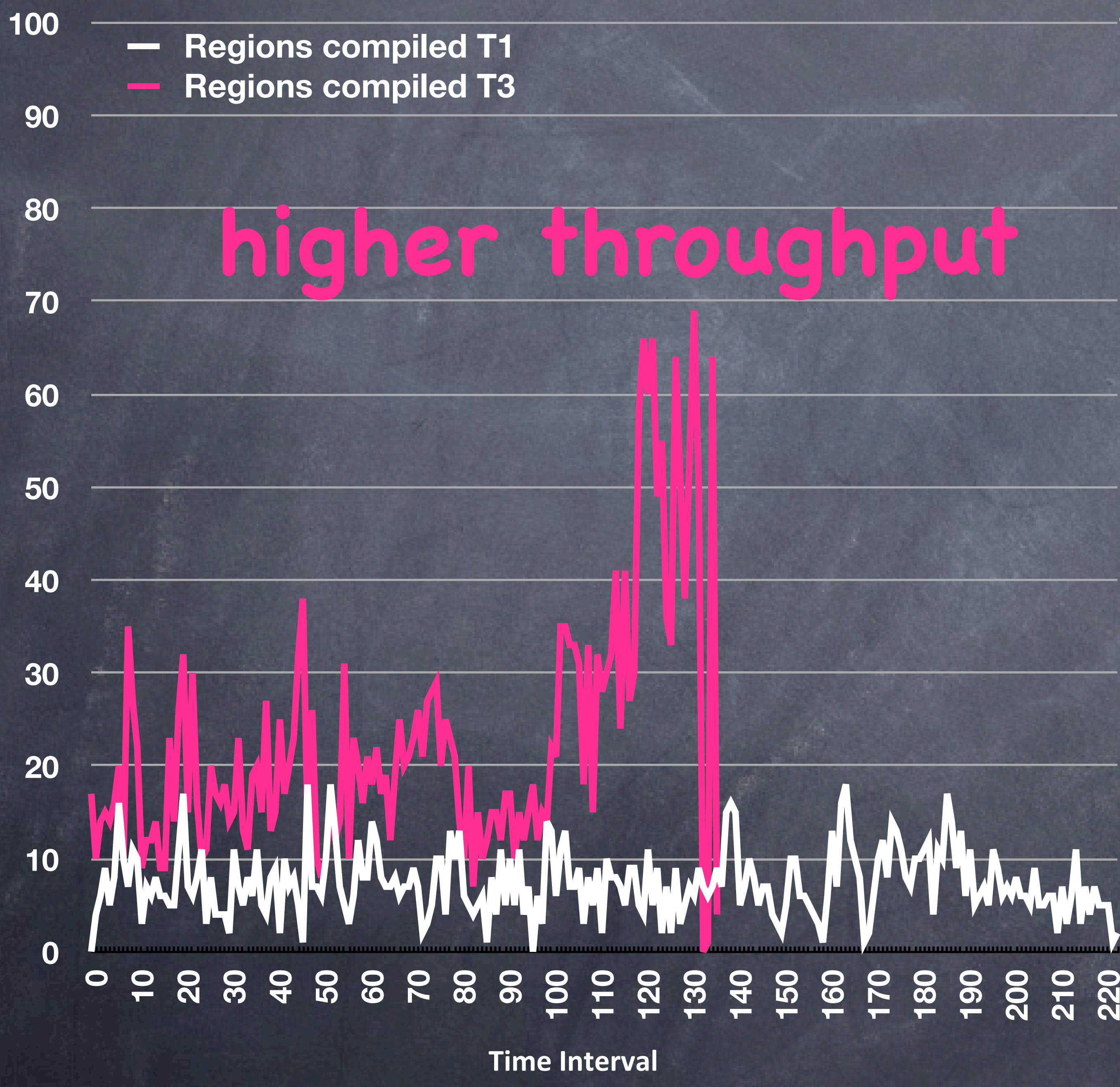

403.gcc - Regions Compiled

higher throughput


403.gcc - Average Queue Length

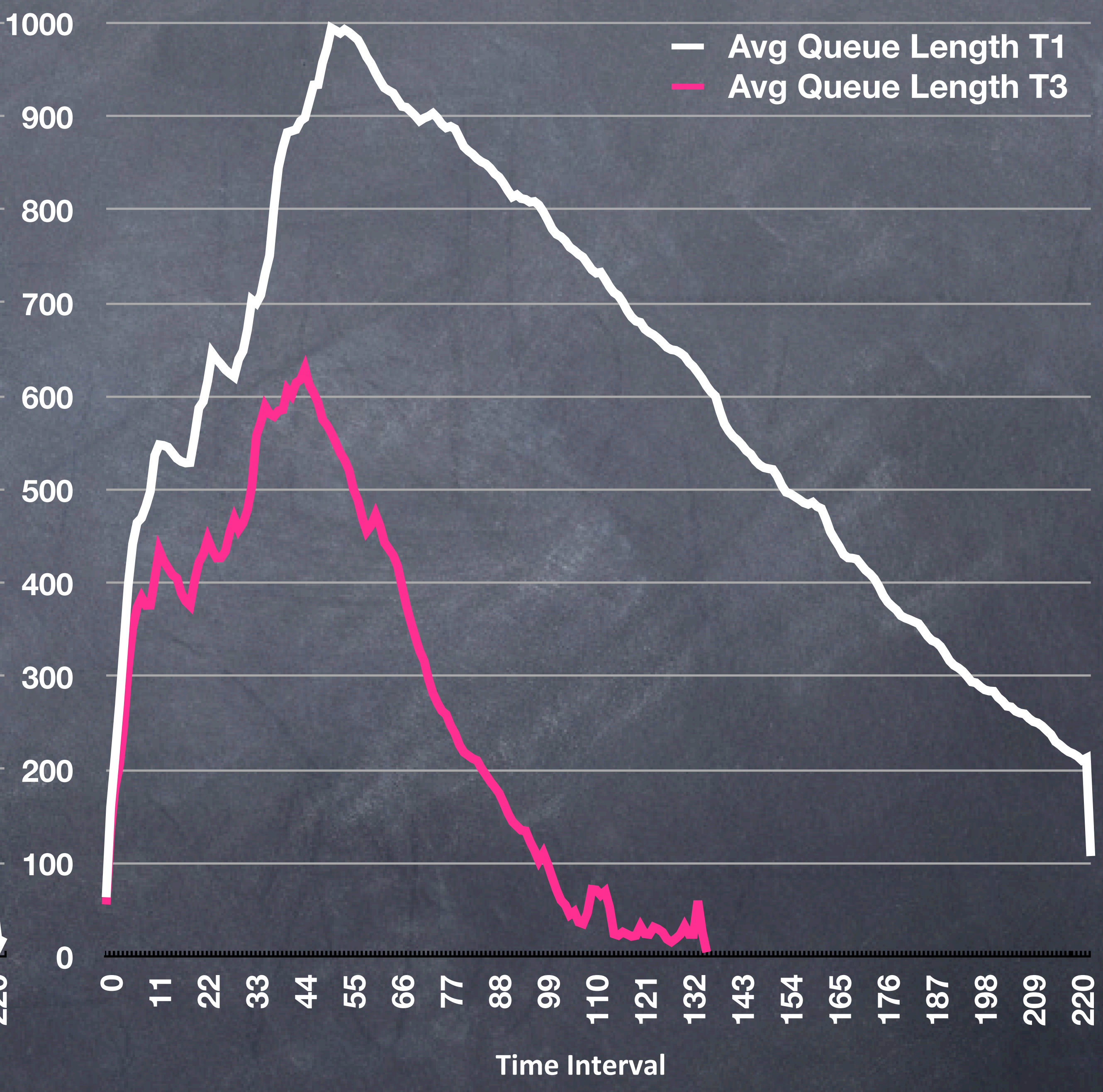
# Effect of Concurrent and Parallel JIT Compilation on Throughput

403.gcc - Regions Compiled

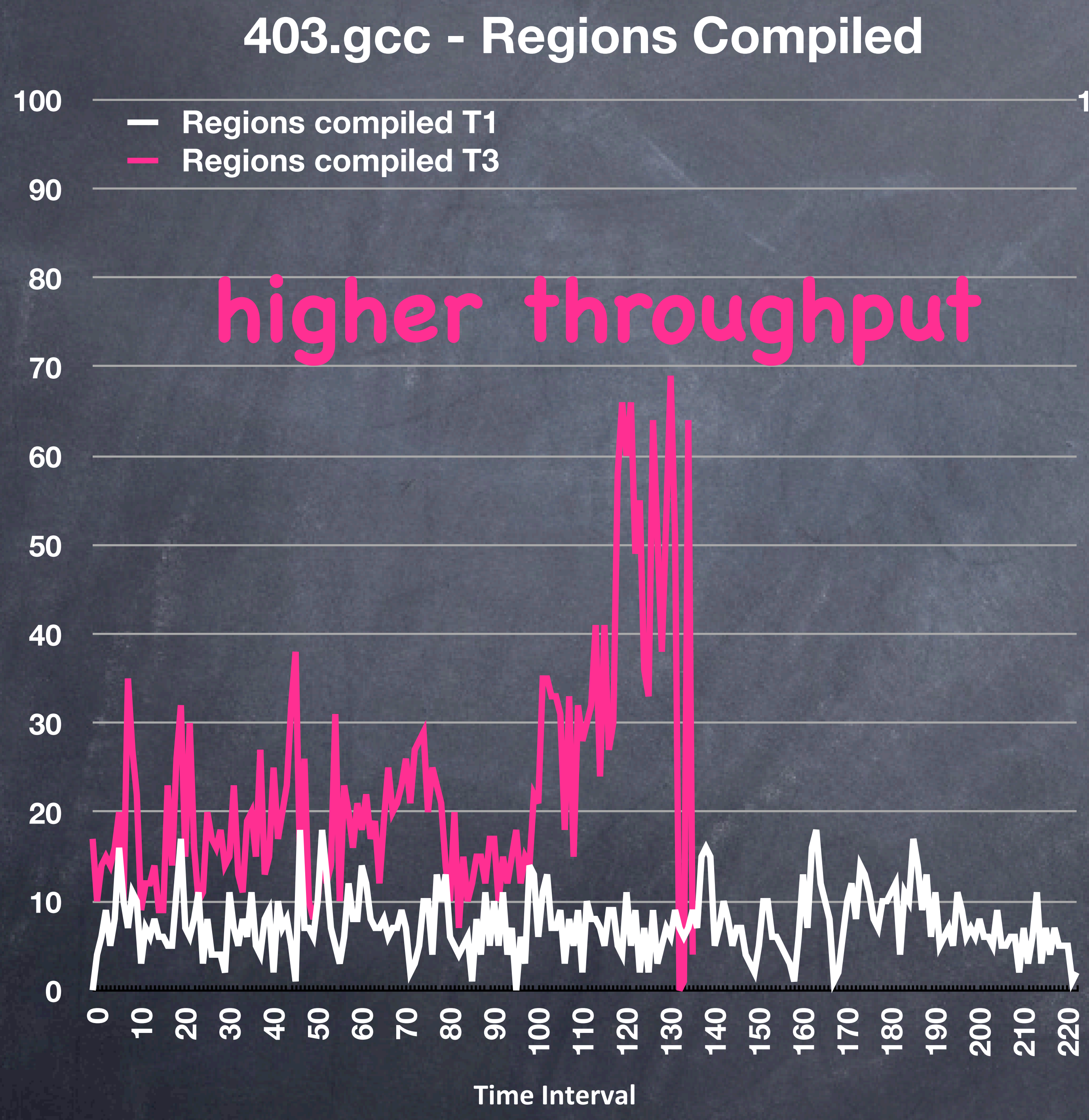


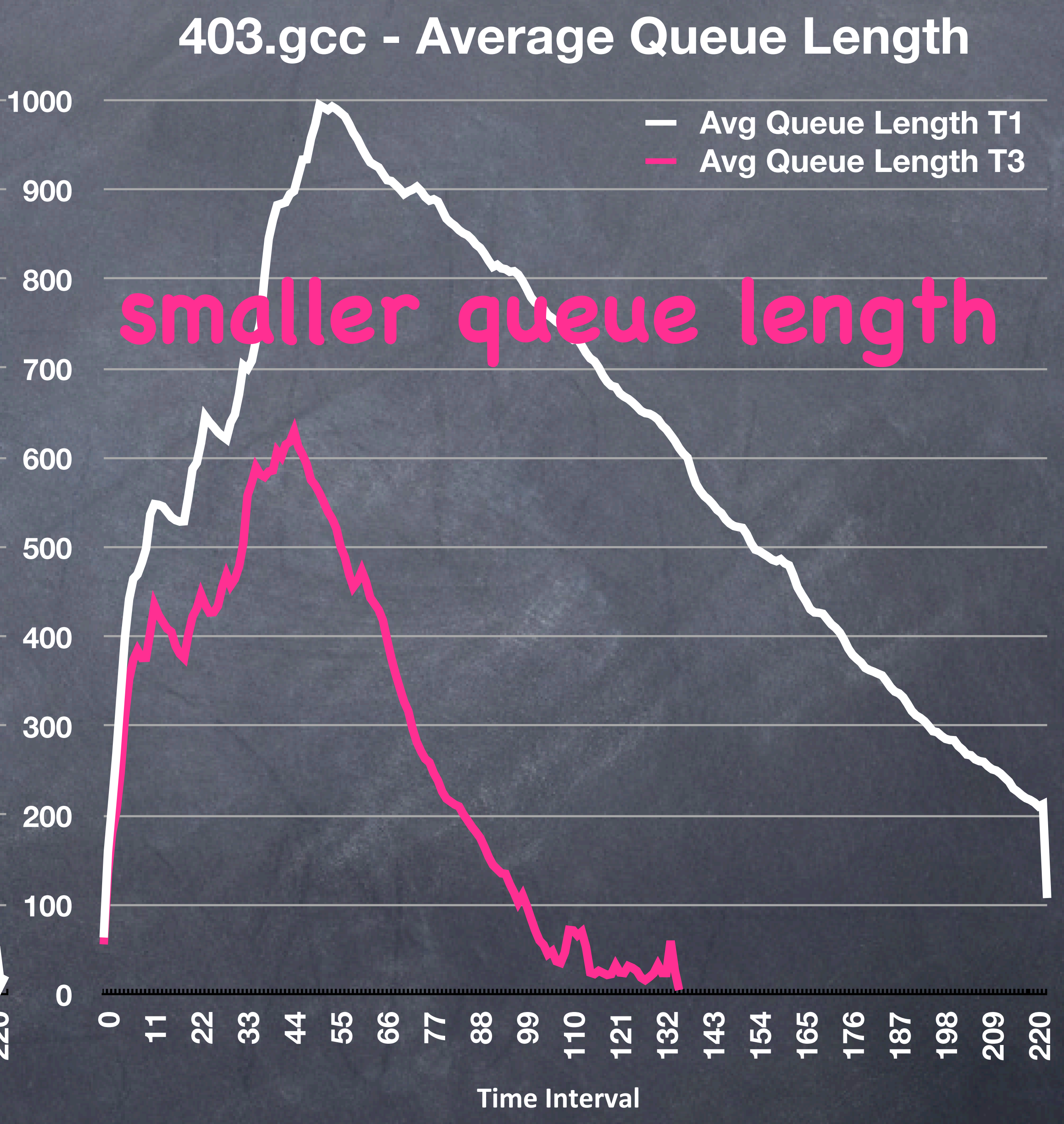
higher throughput

403.gcc - Average Queue Length



# Effect of Concurrent and Parallel JIT Compilation on Throughput

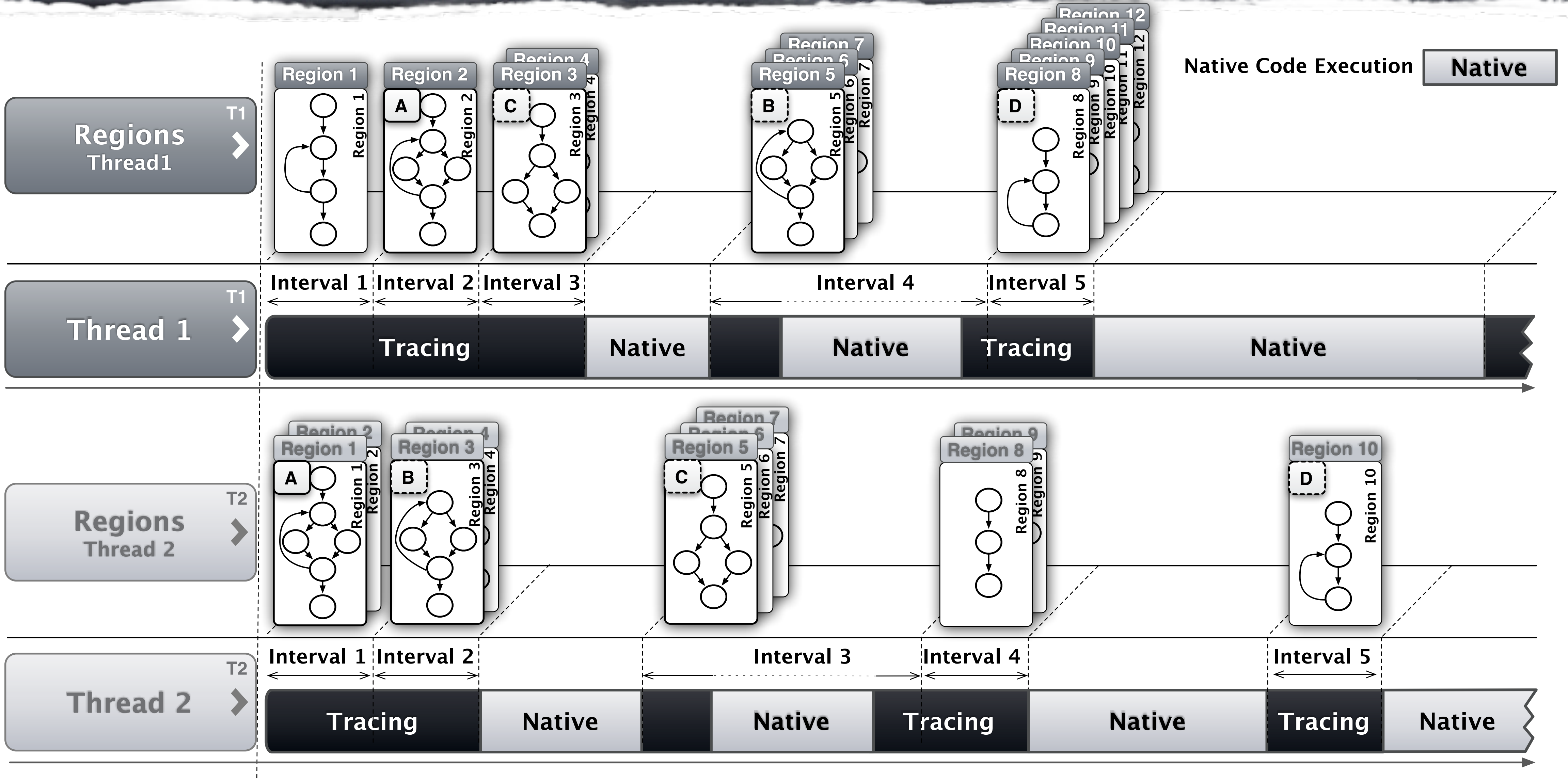




# Does this scale for multi-threaded/core applications?

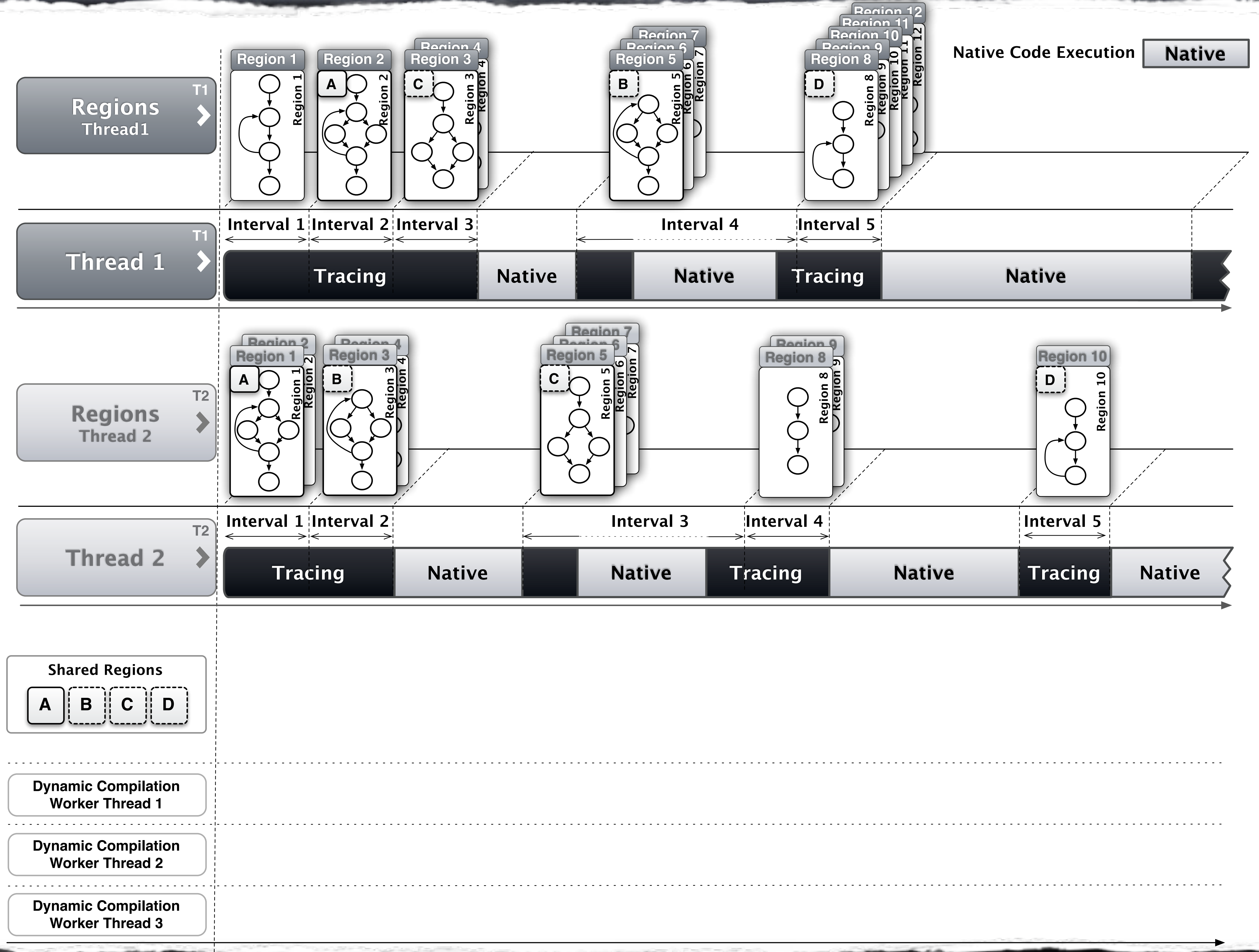
# Concurrent and Parallel JIT Compilation in Action

(trace sharing)



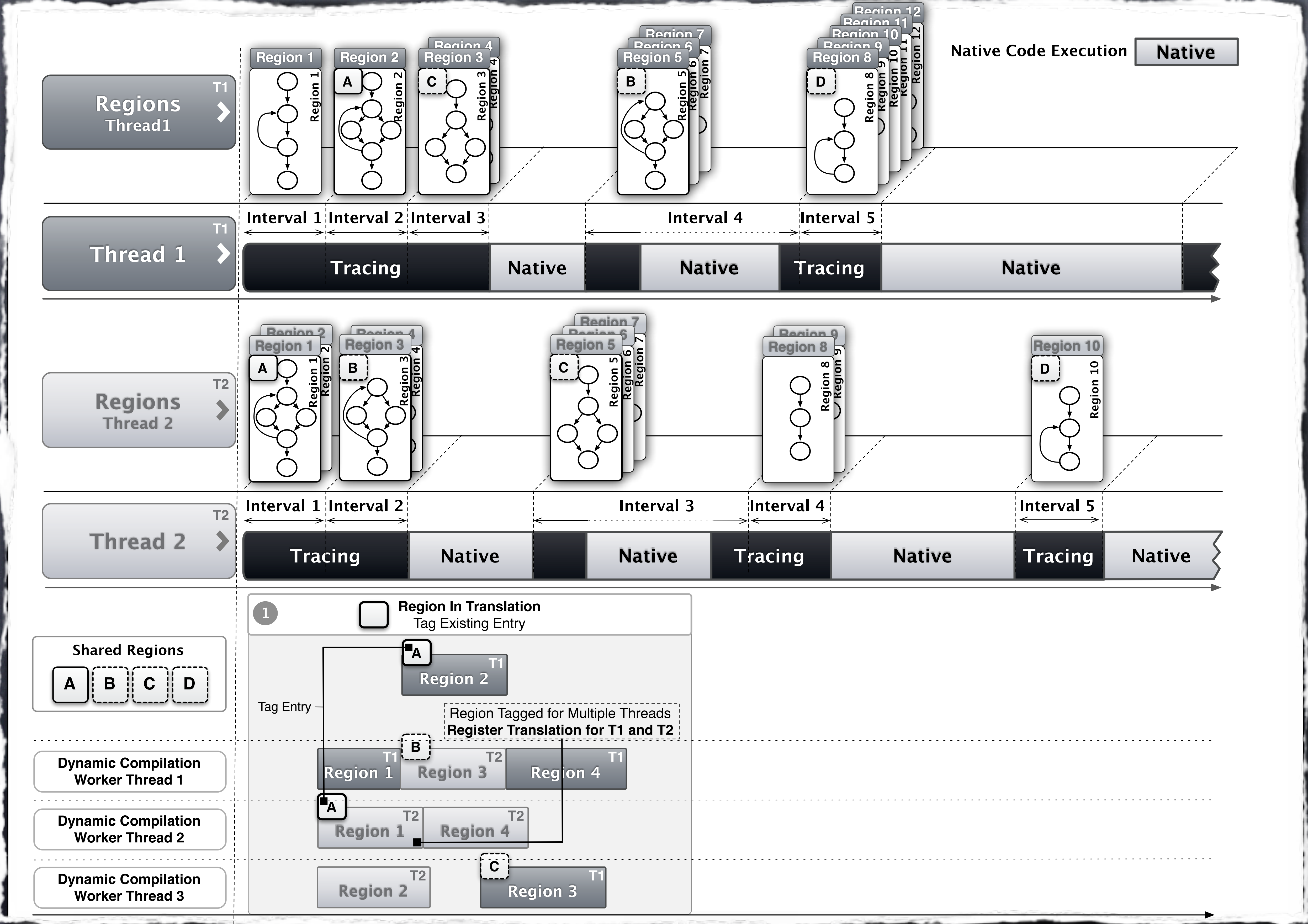
# Concurrent and Parallel JIT Compilation in Action

(trace sharing)



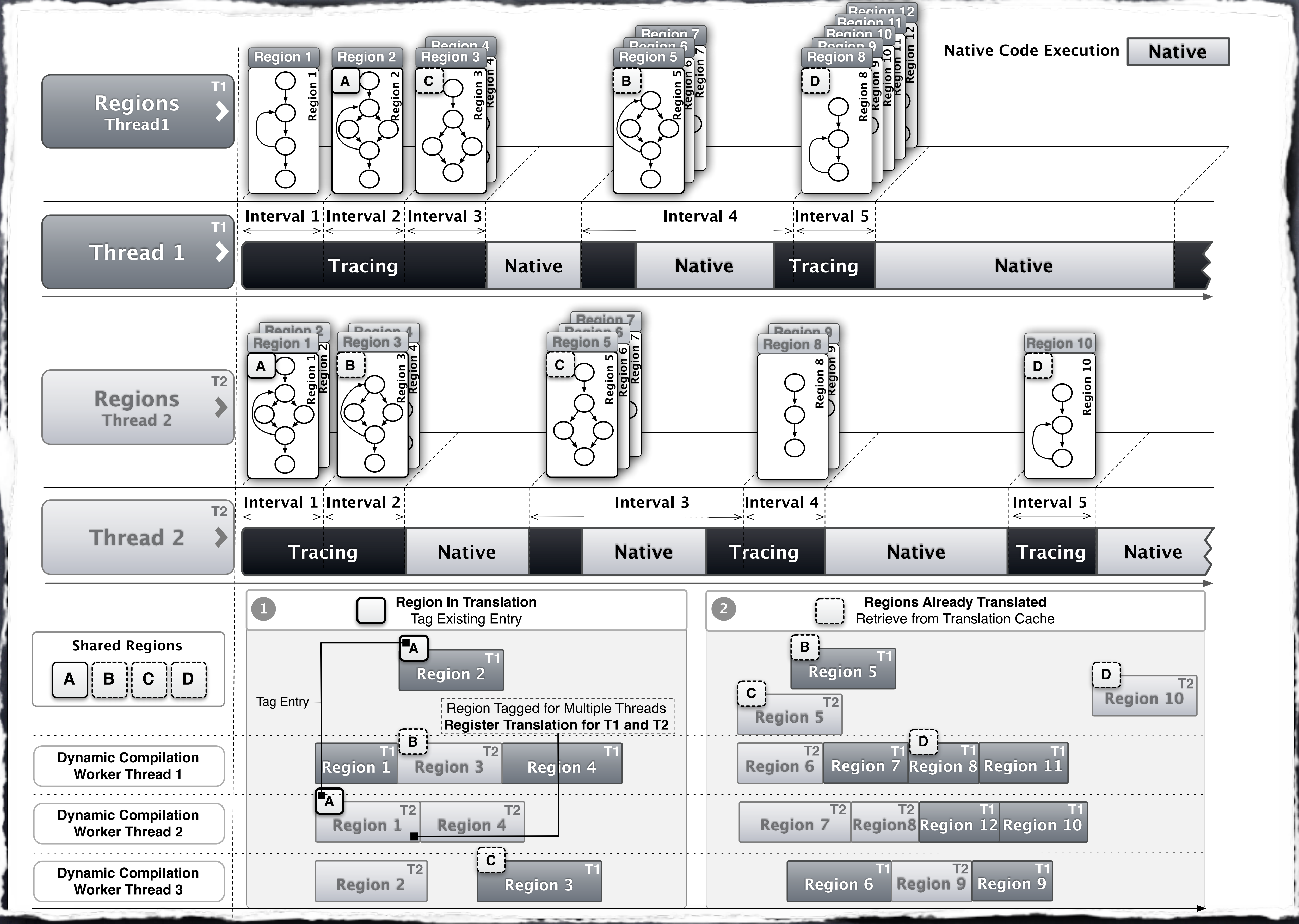
# Concurrent and Parallel JIT Compilation in Action

(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

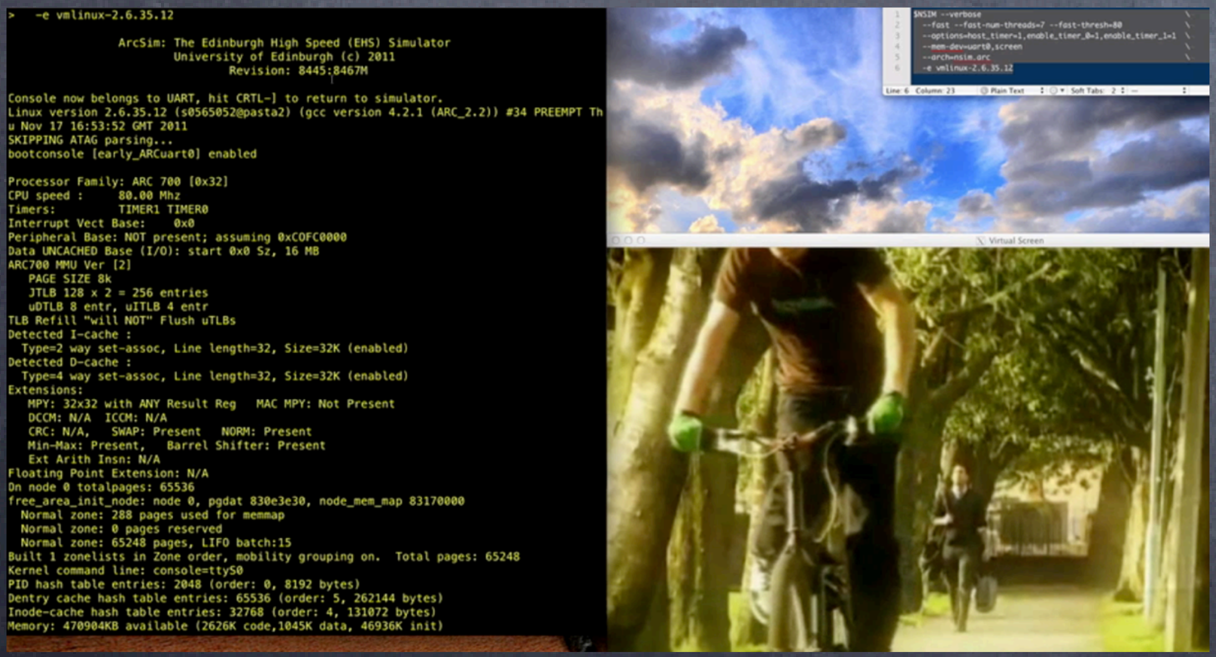
# Demos

Video Decoding
and Playback

# Demos

webm

Video Decoding and Playback



```
>   -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 CRTL-] to return to simulator.
Linux version 2.6.35.12 (s0565052@pasta2) (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 :
  Type=2 way set-assoc, Line length=32, Size=32K (enabled)
Detected D-cache :
  Type=4 way set-assoc, Line length=32, Size=32K (enabled)
Extensions:
   MPY: 32x32 with ANY Result Reg   MAC MPY: Not Present
   DCCM: N/A  ICCM: 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
Dn 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)
```

```
$NSIM --verbose                                              \
  --fast --fast-num-threads=7 --fast-thresh=80               \
  --options=host_timer=1,enable_timer_0=1,enable_timer_1=1   \
  --mem-dev=uart0,screen                                     \
  --arch=nsim.arc                                            \
  -e vmlinux-2.6.35.12
```

Full System OS Simulation

# Thank You