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

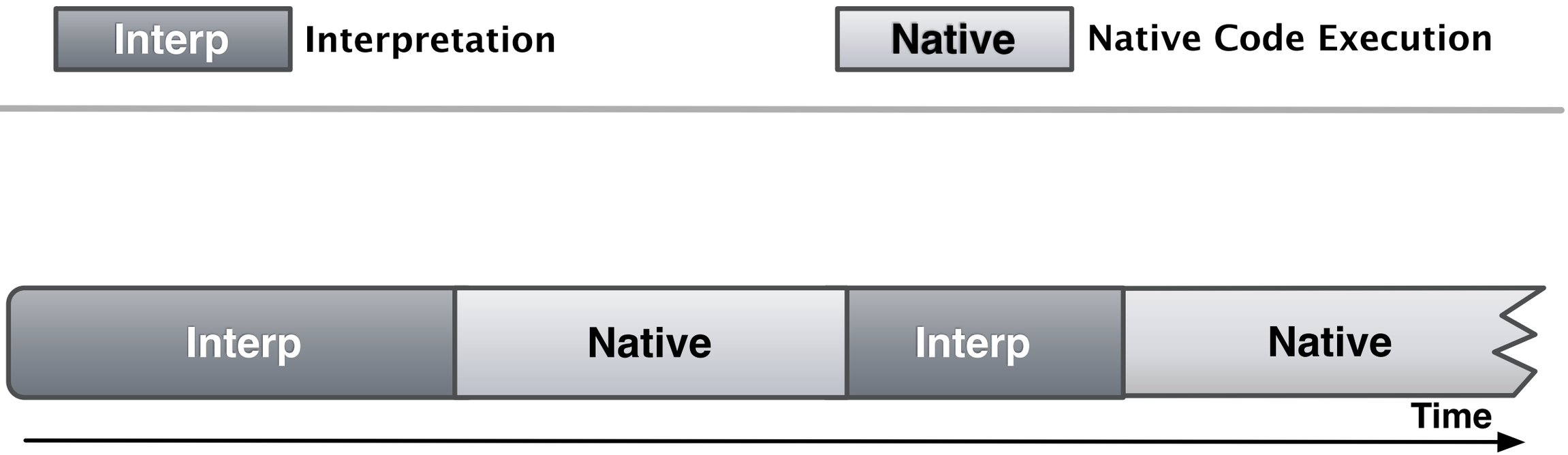
Dynamic Compilation

What do we want to improve?

Interp **Interpretation**

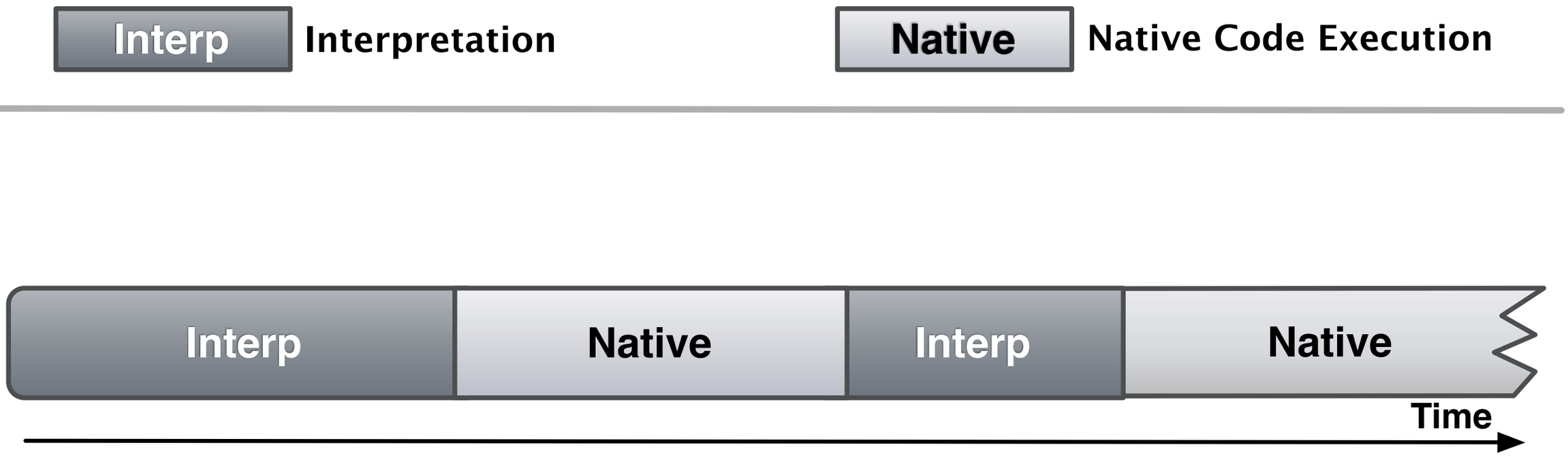
Native **Native Code Execution**

Dynamic Compilation
What do we want to improve?



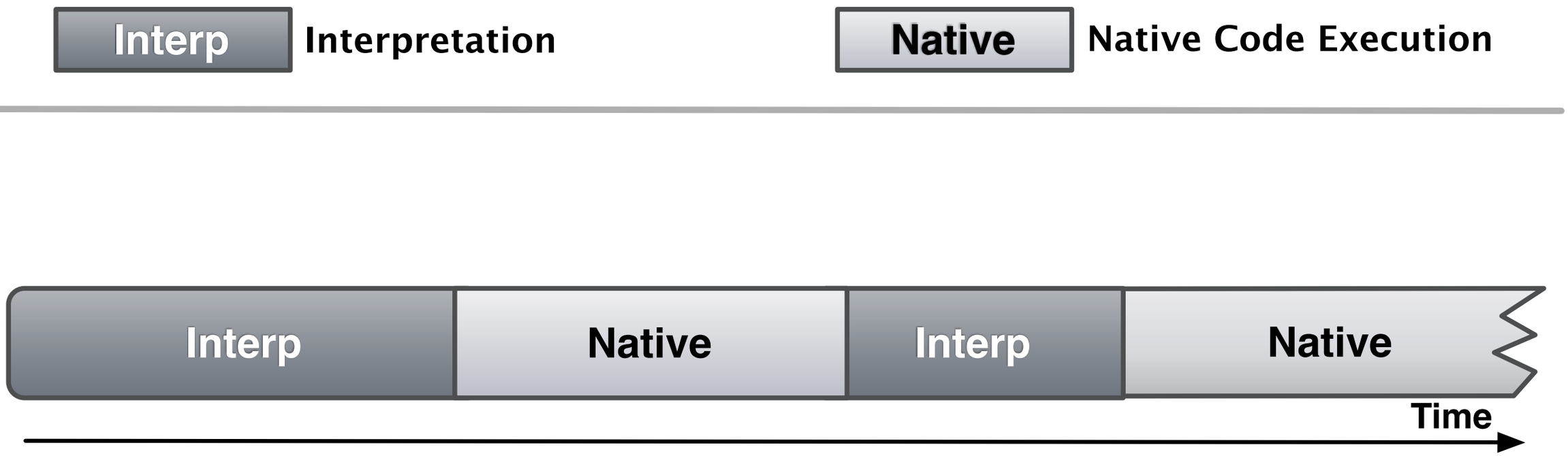
- initially code is interpreted

Dynamic Compilation
What do we want to improve?



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

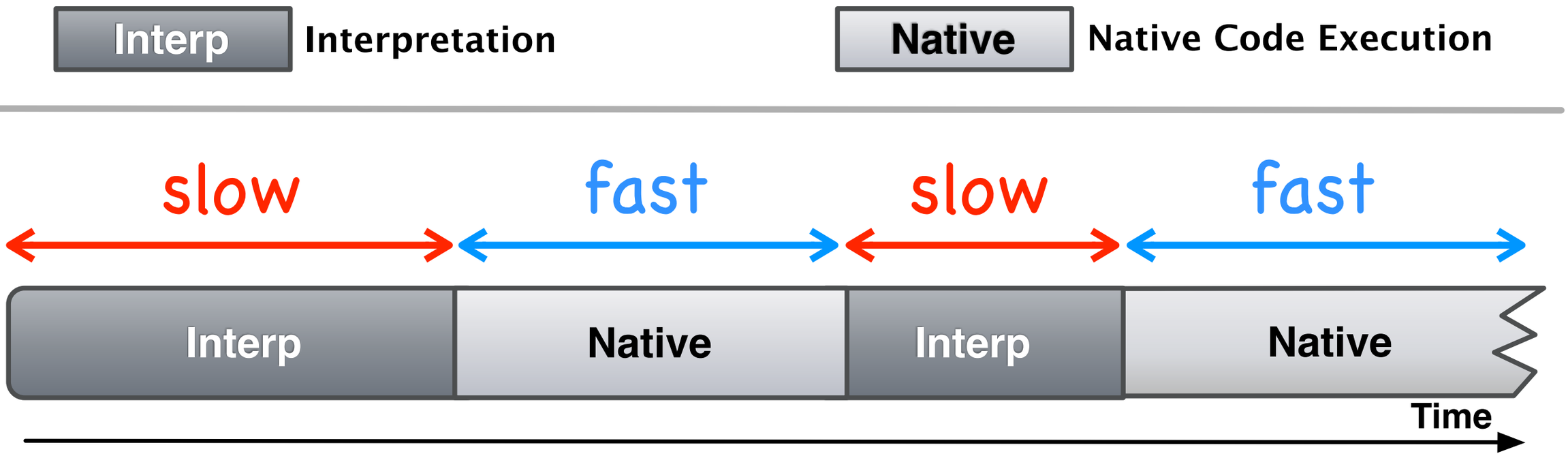
Dynamic Compilation
What do we want to improve?



- 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

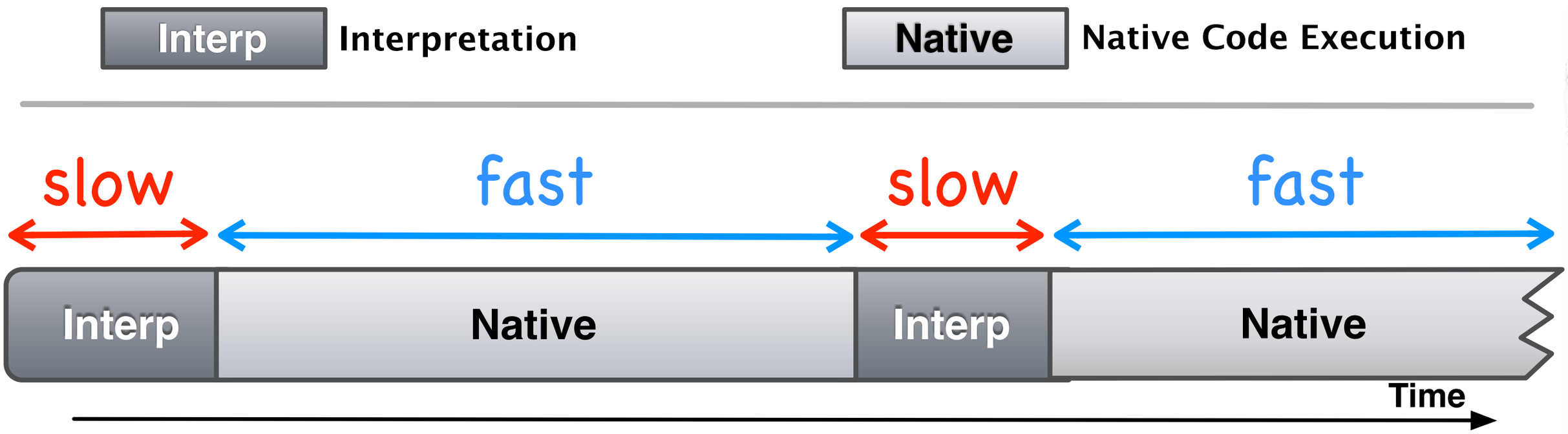
What do we want to improve?



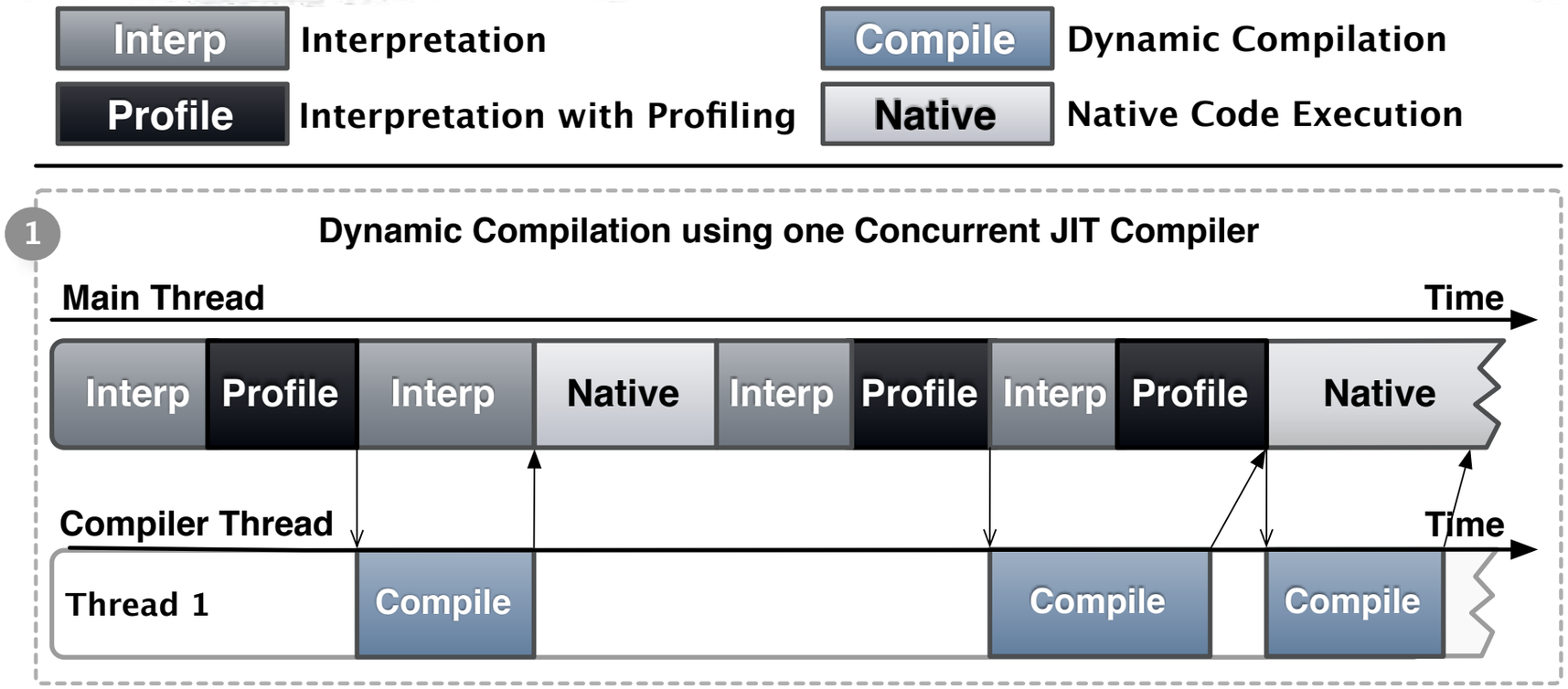
- initially code is interpreted
- frequently executed code is compiled on-the-fly
- switch from interpretive to native code execution as soon as dynamically compiled code is available

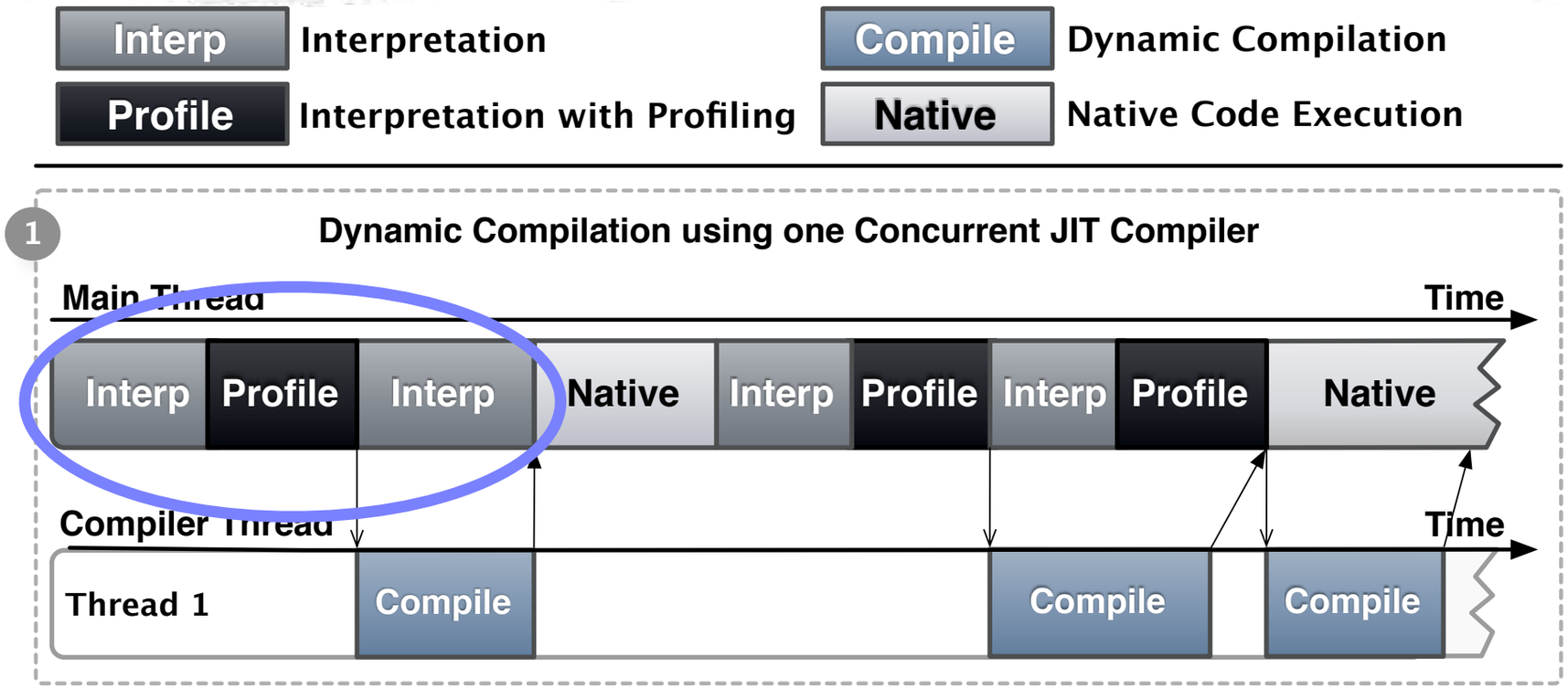
Dynamic Compilation

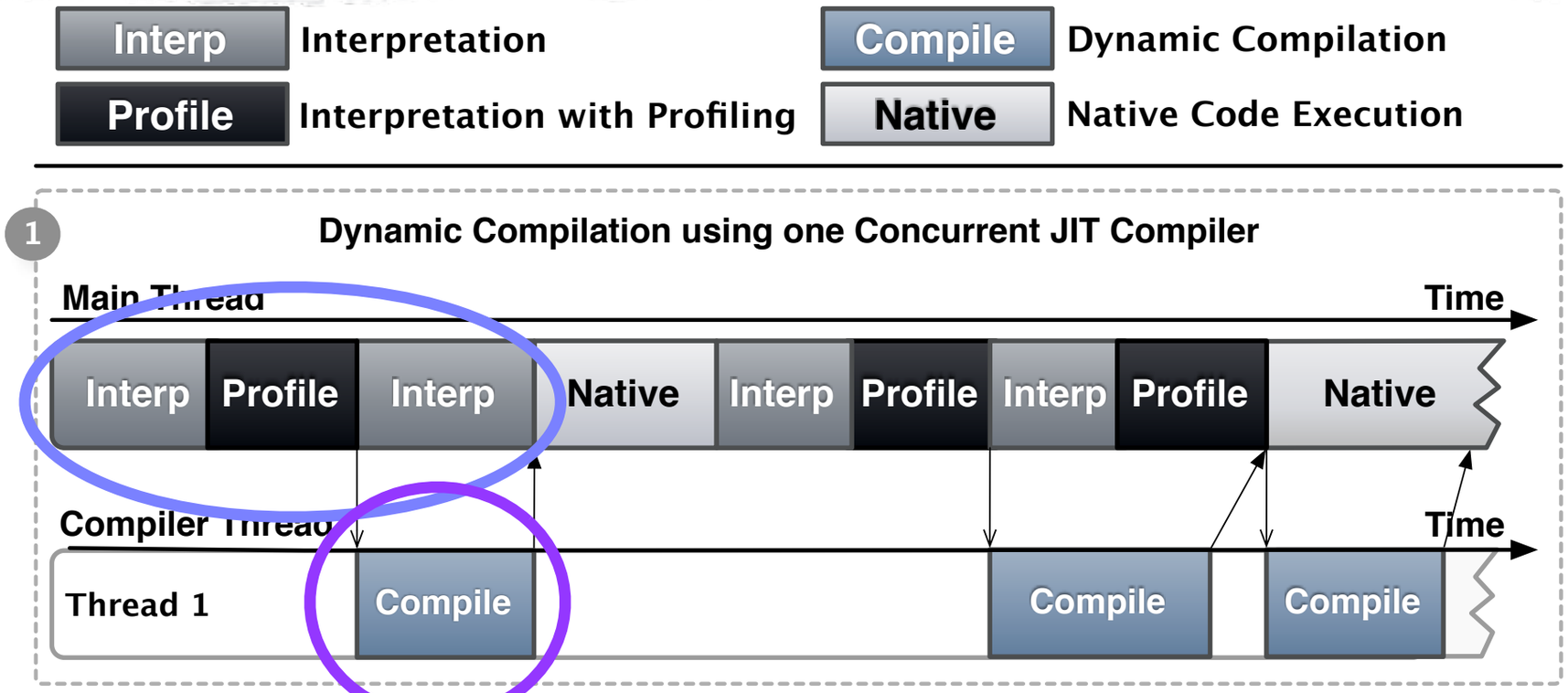
What do we want to improve?



Earlier transition from interpretive to native execution


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


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


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

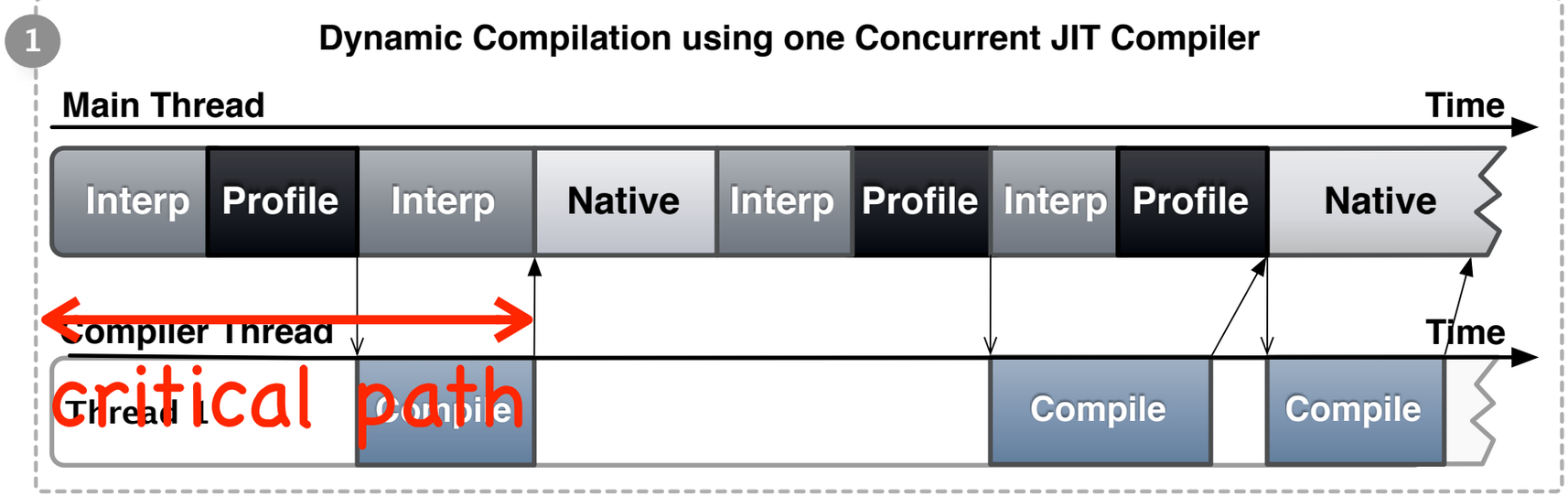
Interp Interpretation

Profile Interpretation with Profiling

Compile Dynamic Compilation

Native Native Code Execution

1



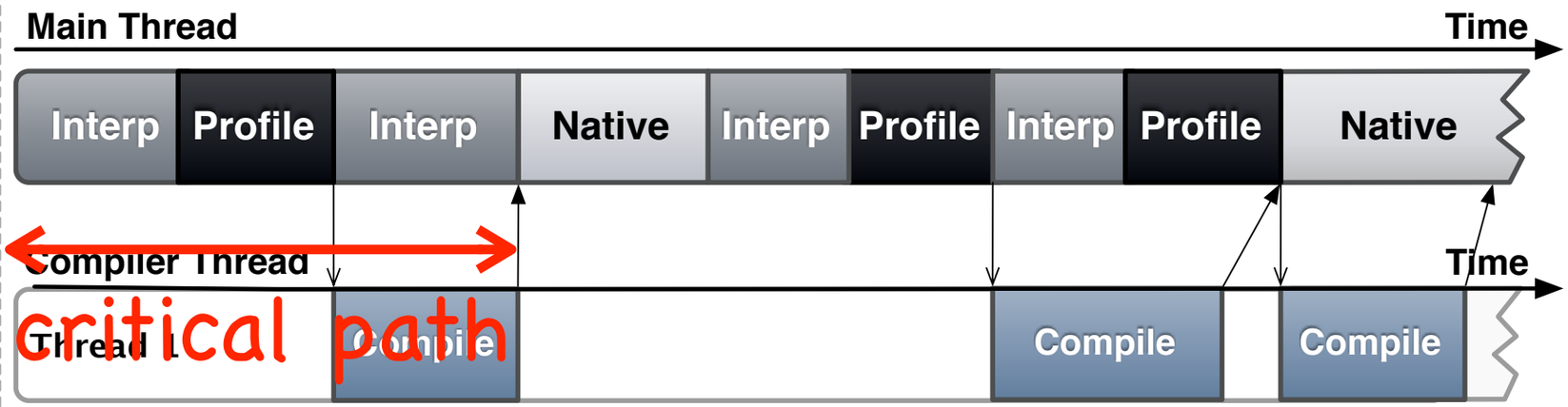
Interp Interpretation

Profile Interpretation with Profiling

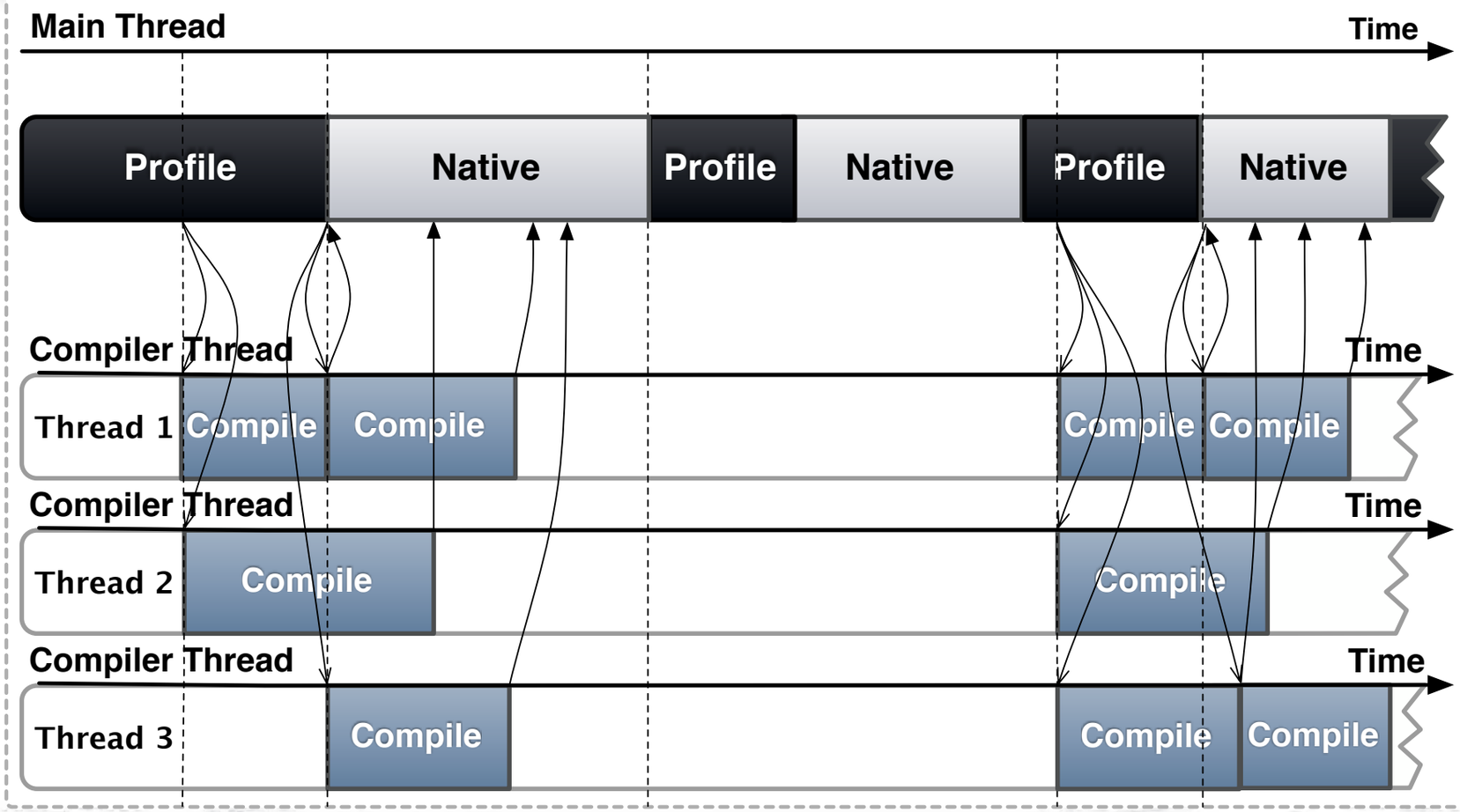
Compile Dynamic Compilation

Native Native Code Execution

1 Dynamic Compilation using one Concurrent JIT Compiler



2 Dynamic Compilation using Concurrent and Parallel JIT Compiler Task Farm



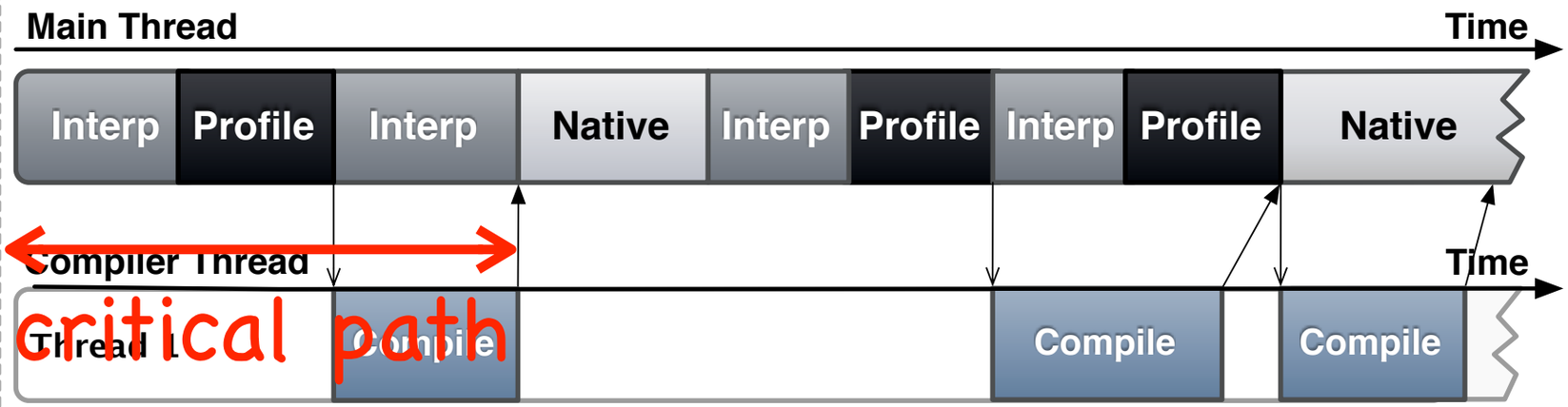
Interp Interpretation

Profile Interpretation with Profiling

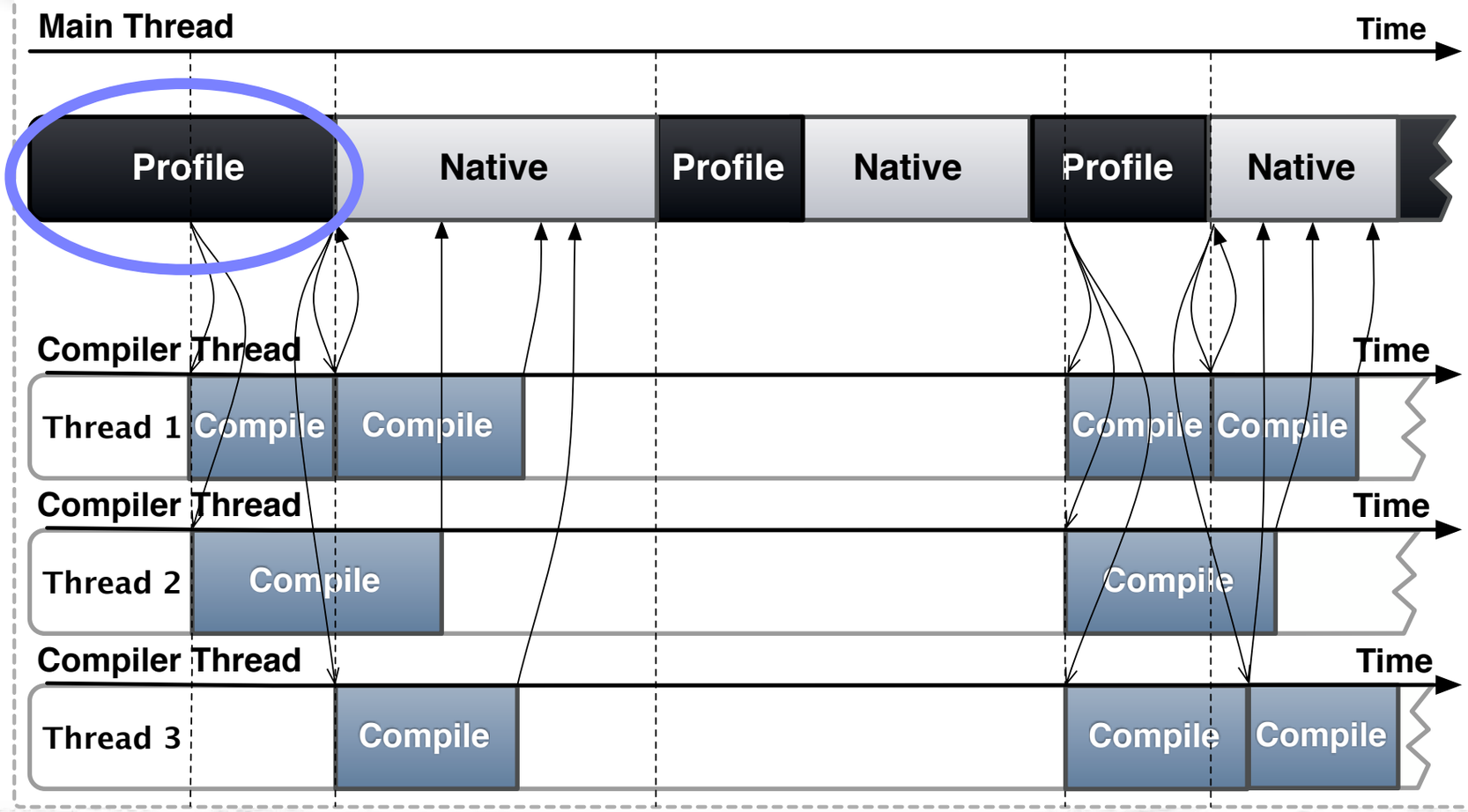
Compile Dynamic Compilation

Native Native Code Execution

1 Dynamic Compilation using one Concurrent JIT Compiler



2 Dynamic Compilation using Concurrent and Parallel JIT Compiler Task Farm



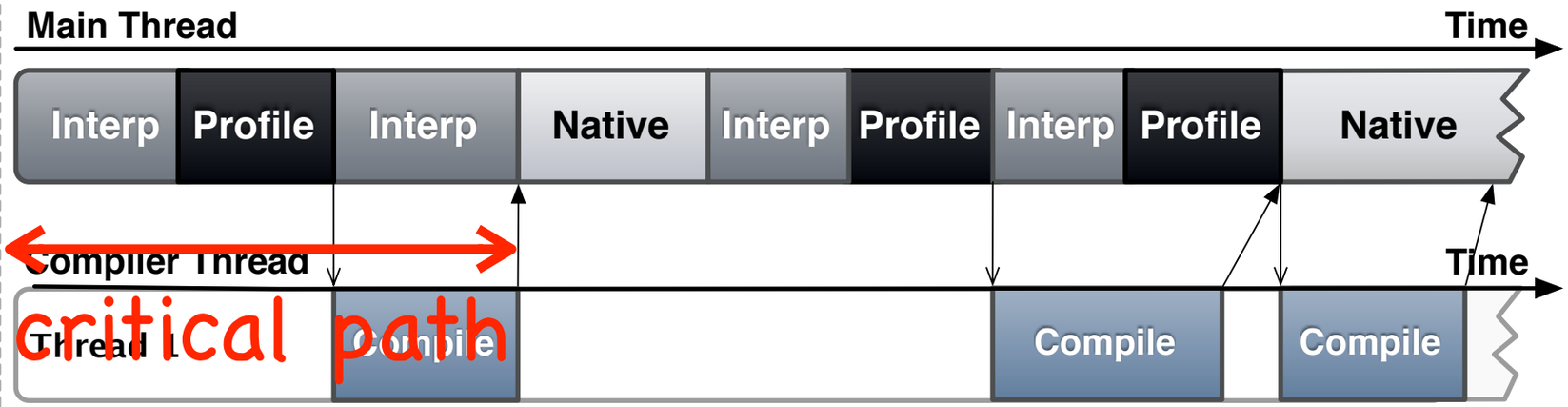
Interp Interpretation

Profile Interpretation with Profiling

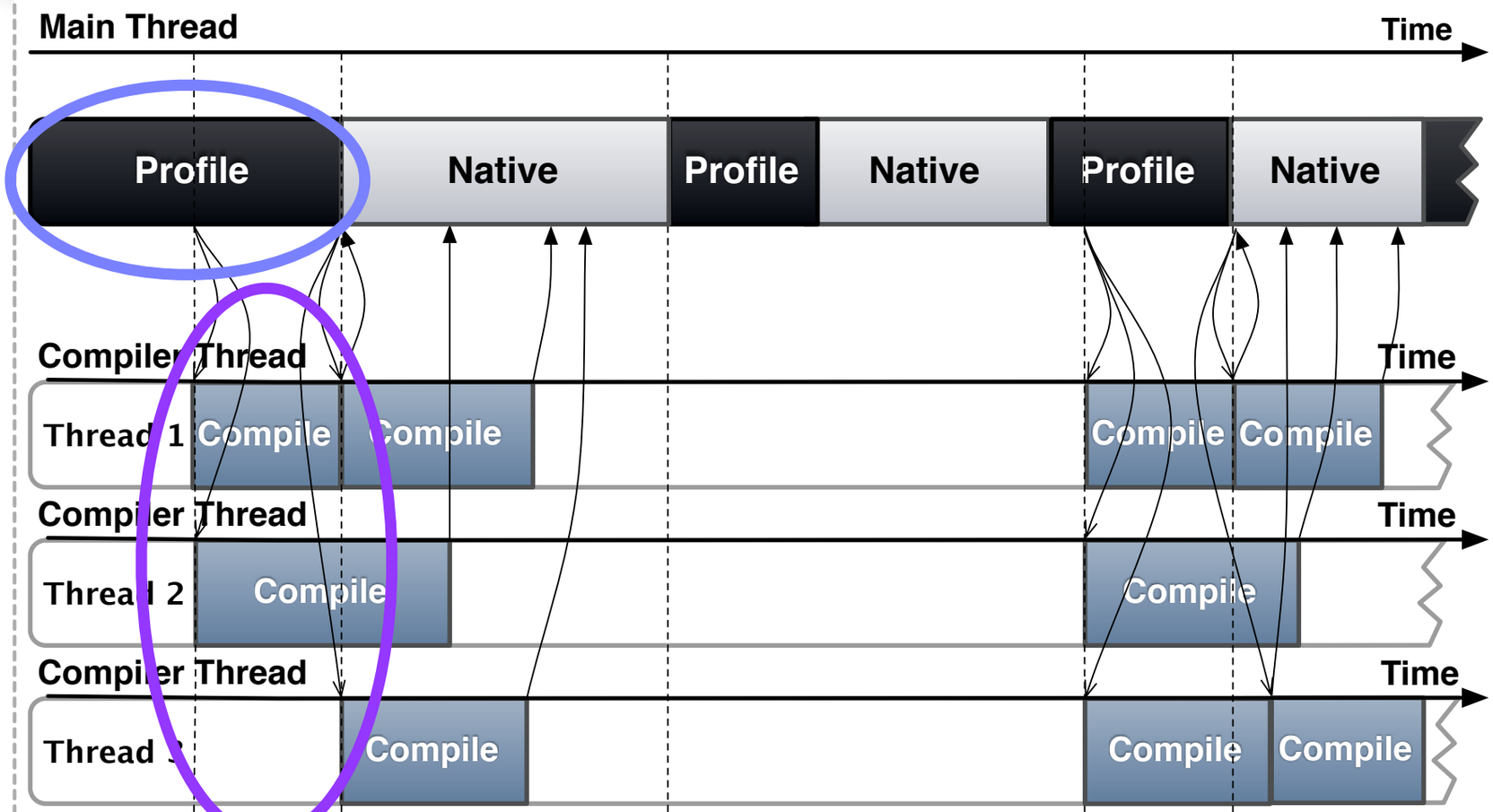
Compile Dynamic Compilation

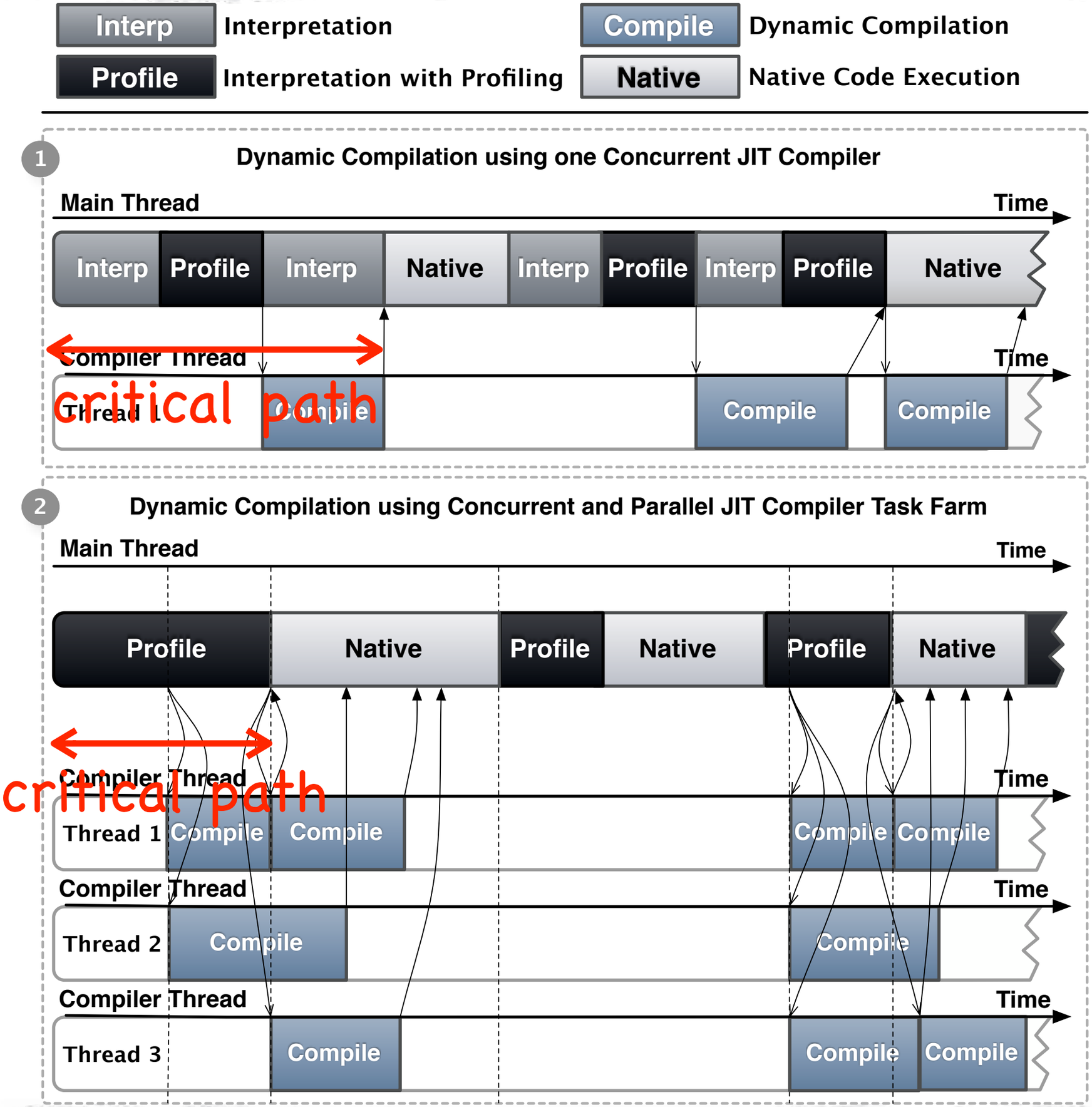
Native Native Code Execution

1 Dynamic Compilation using one Concurrent JIT Compiler

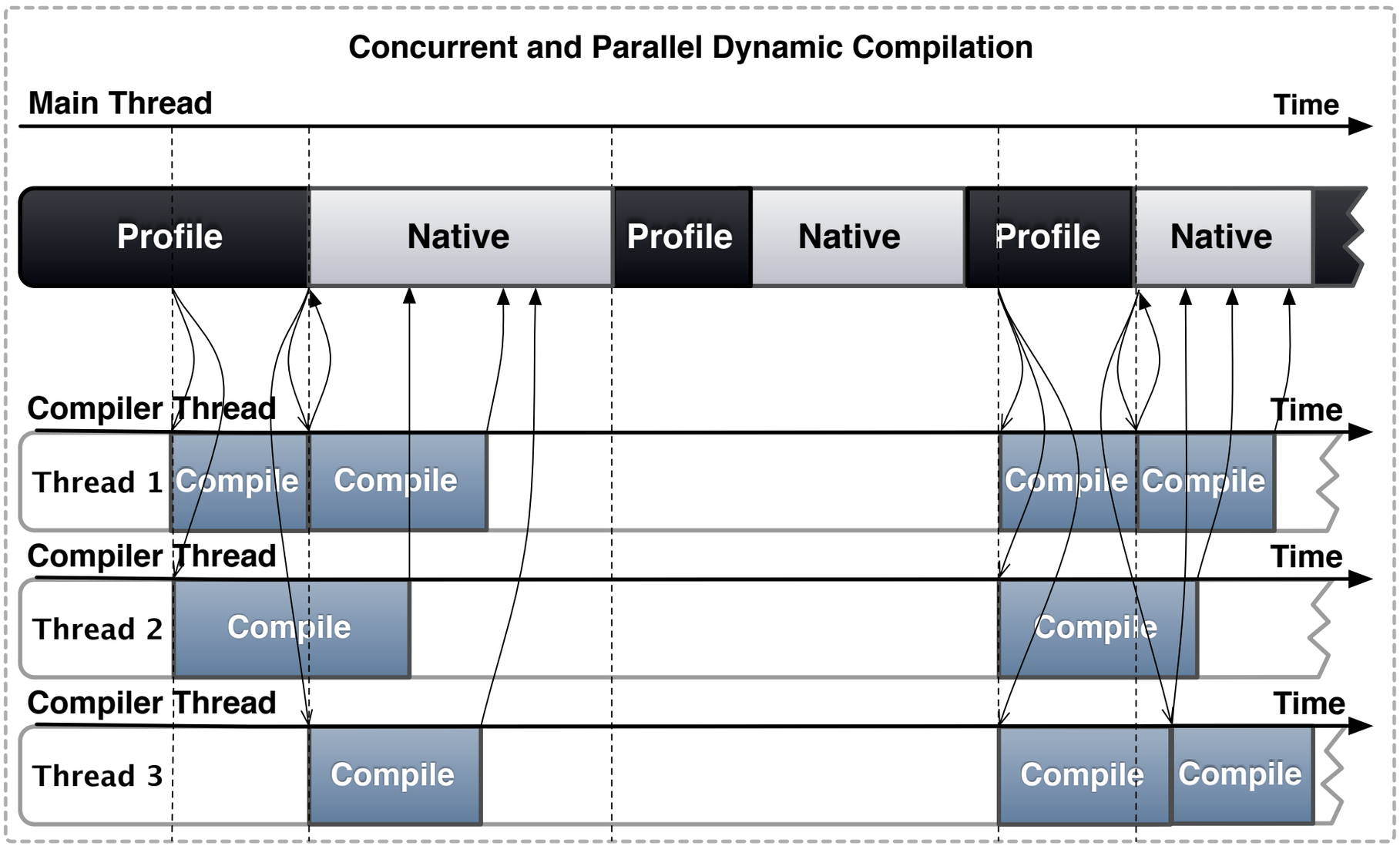


2 Dynamic Compilation using Concurrent and Parallel JIT Compiler Task Farm



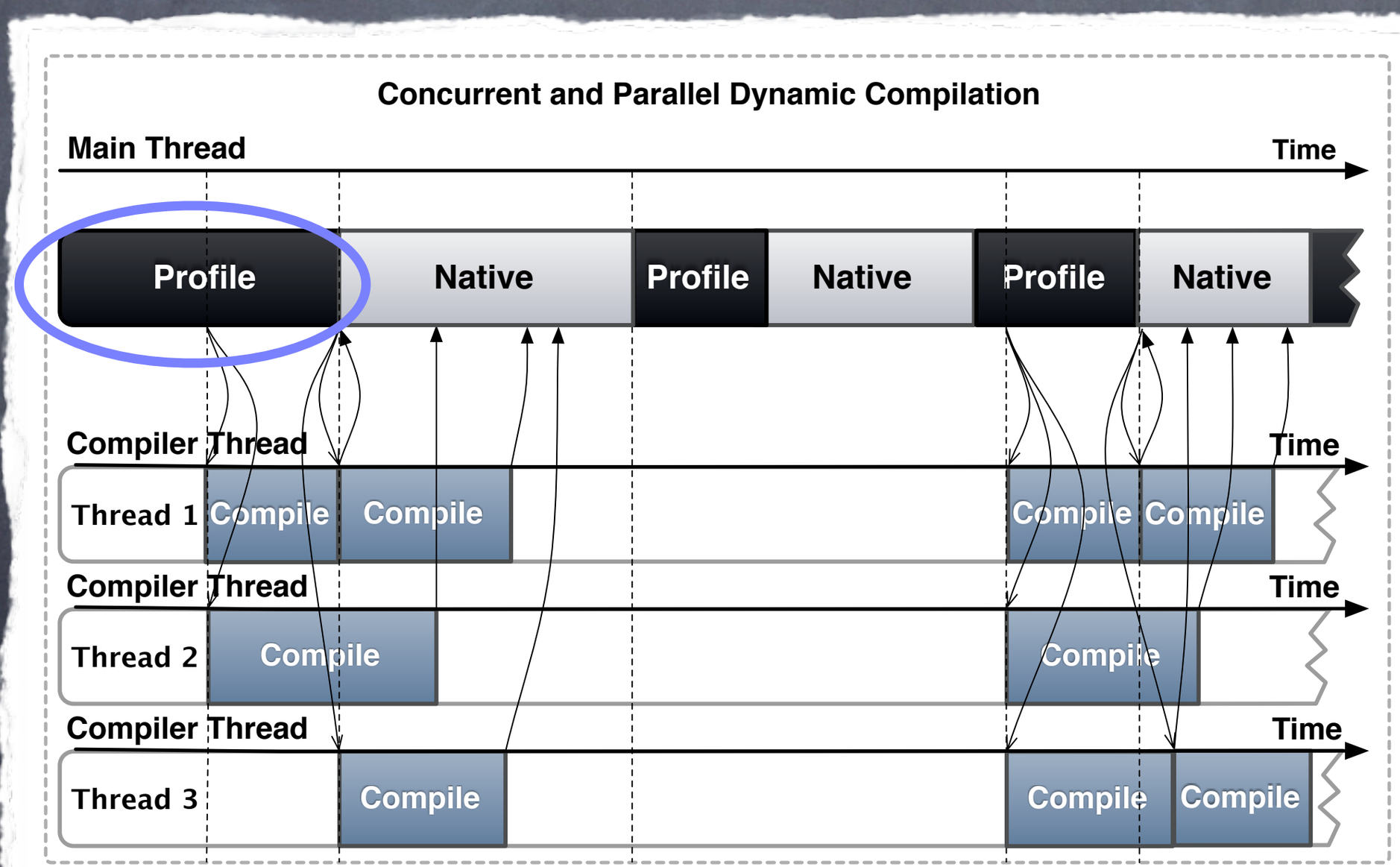

Interp
Interpretation
Compile
Dynamic Compilation
Profile
Interpretation with Profiling
Native
Native Code Execution
1
Dynamic Compilation using one Concurrent JIT Compiler
Main Thread
Time
Interp
Profile
Interp
Native
Interp
Profile
Interp
Profile
Native
critical path
Compiler Thread
Time
Thread 1
Compile
Compile
Compile
2
Dynamic Compilation using Concurrent and Parallel JIT Compiler Task Farm
Main Thread
Time
Profile
Native
Profile
Native
Profile
Native
critical path
Compiler Thread
Time
Thread 1
Compile
Compile
Compile
Compile
Compiler Thread
Time
Thread 2
Compile
Compile
Compiler Thread
Time
Thread 3
Compile
Compile
Compile

Solution To Dynamic Compilation Latency Problem



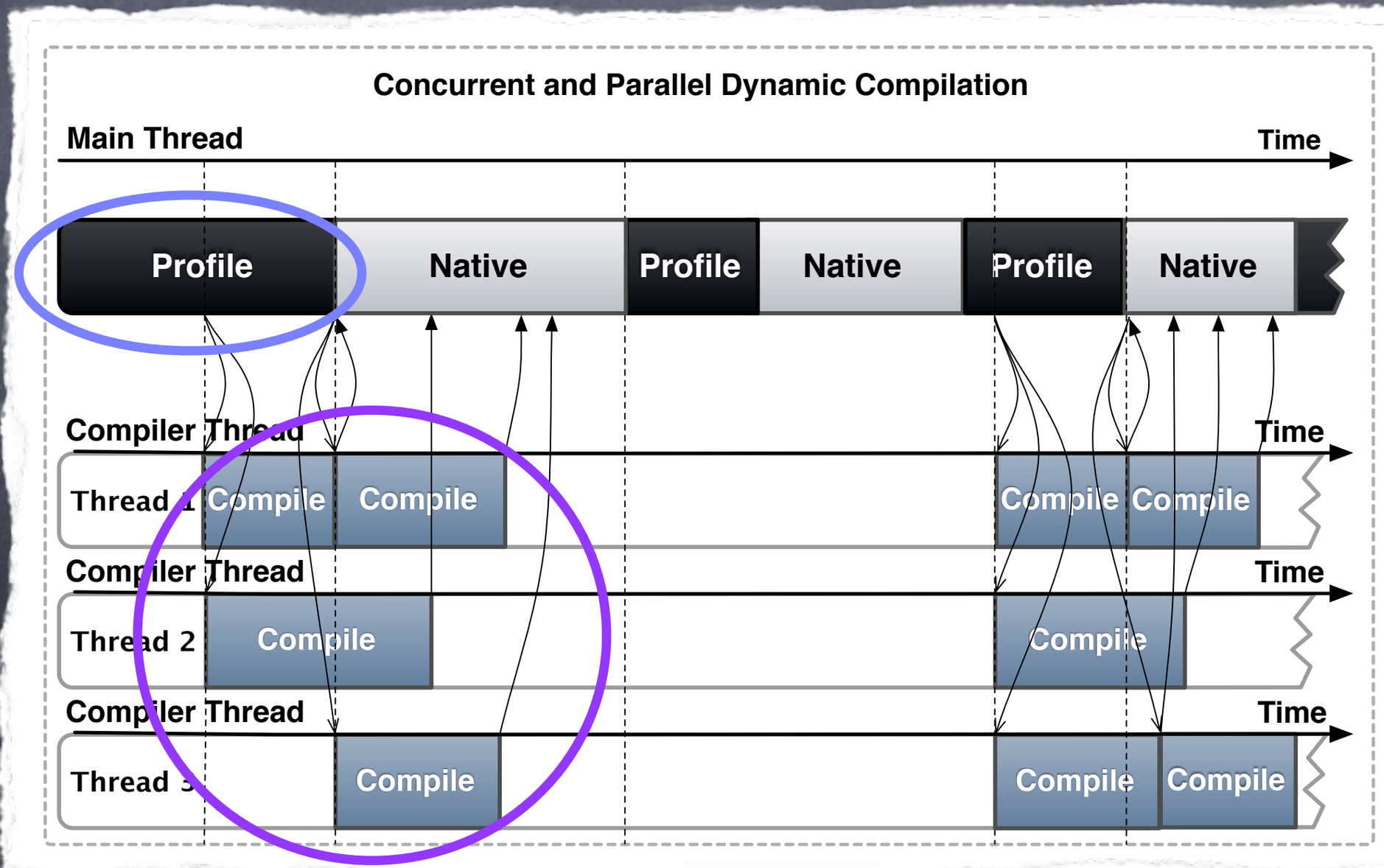
Solution To Dynamic Compilation Latency Problem

- 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

Dynamic

Java
byte-code

JavaScript

CIL

ARCompact
binary

x86
binary

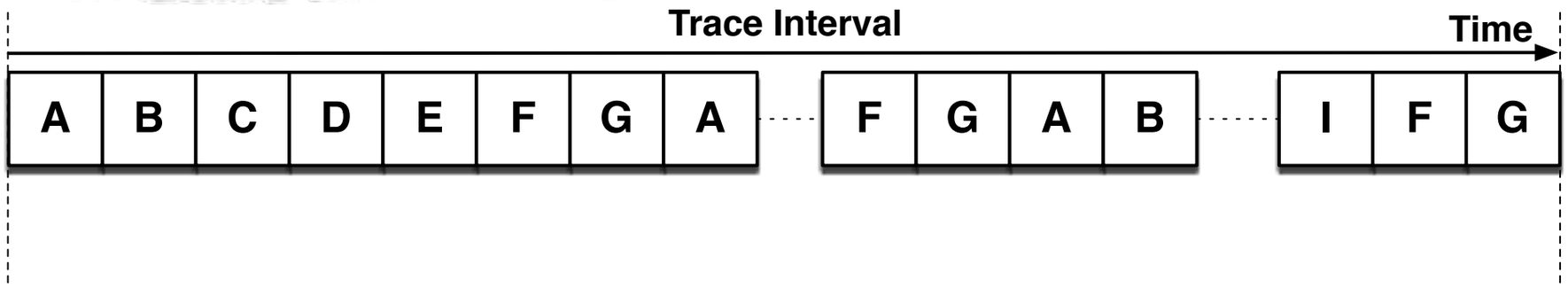
ARM
binary

How hard can Code Discovery be?

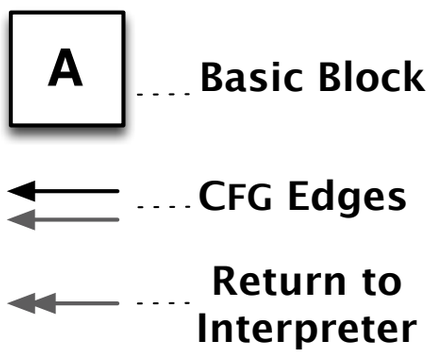
"A crucial problem in the decompilation or disassembly of computer programs is the identification of executable code, i.e. the separation of instructions from data. This problem, for most computer architectures, is equivalent to the Halting Problem and is therefore unsolvable in general."

[Horspool and Marovac - 1980]

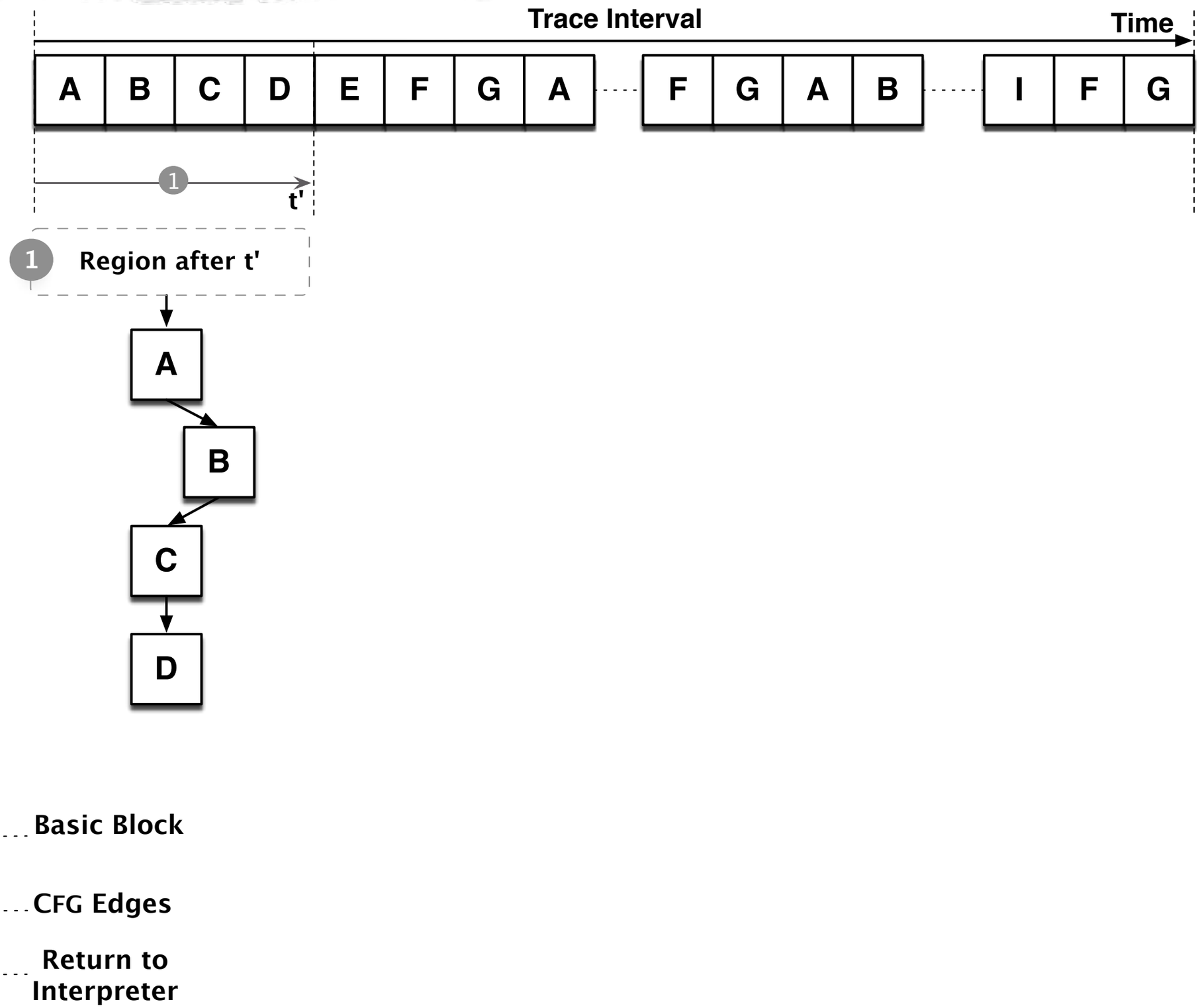
Incremental Code Discovery



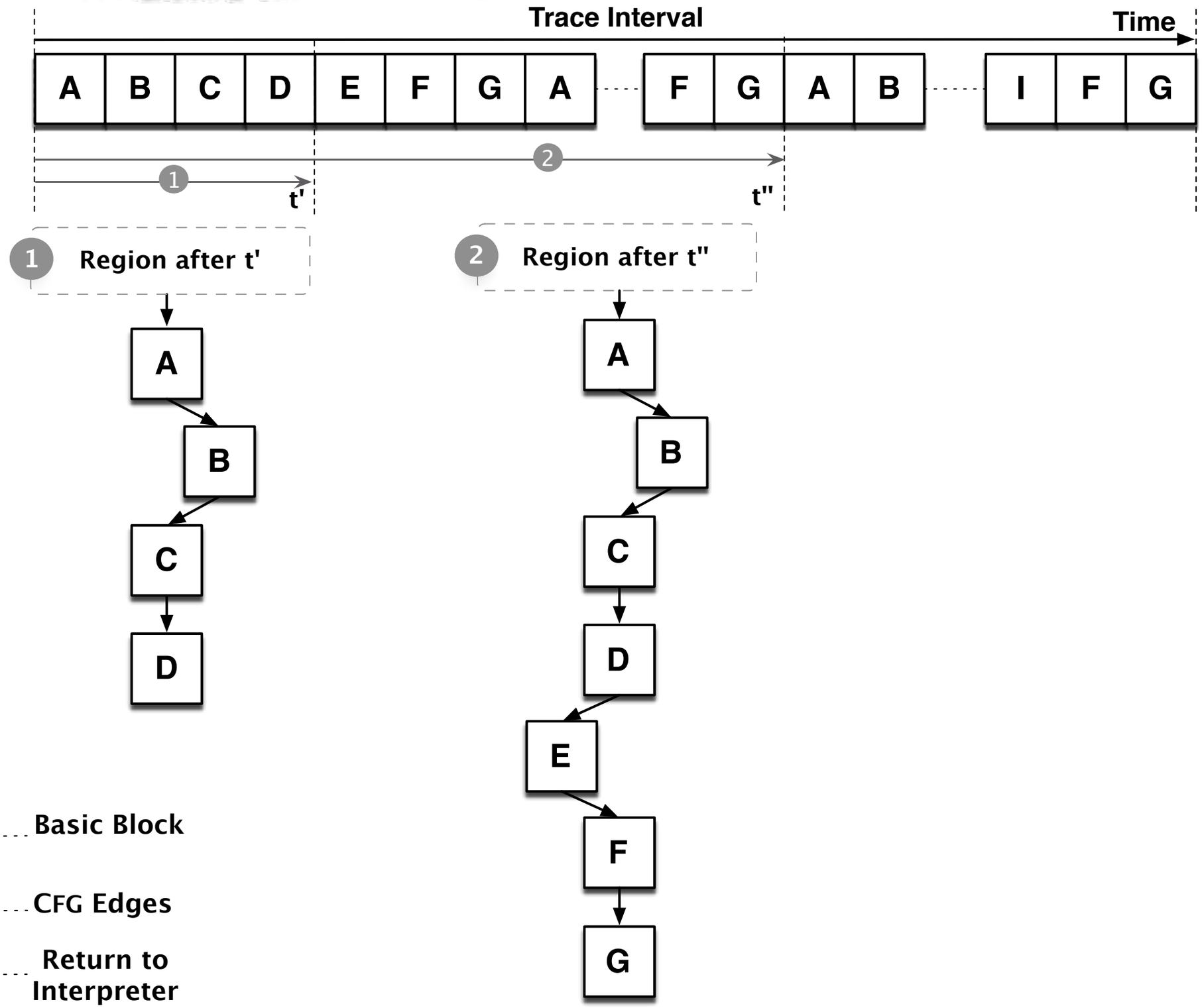
Sequence of interpreted basic blocks



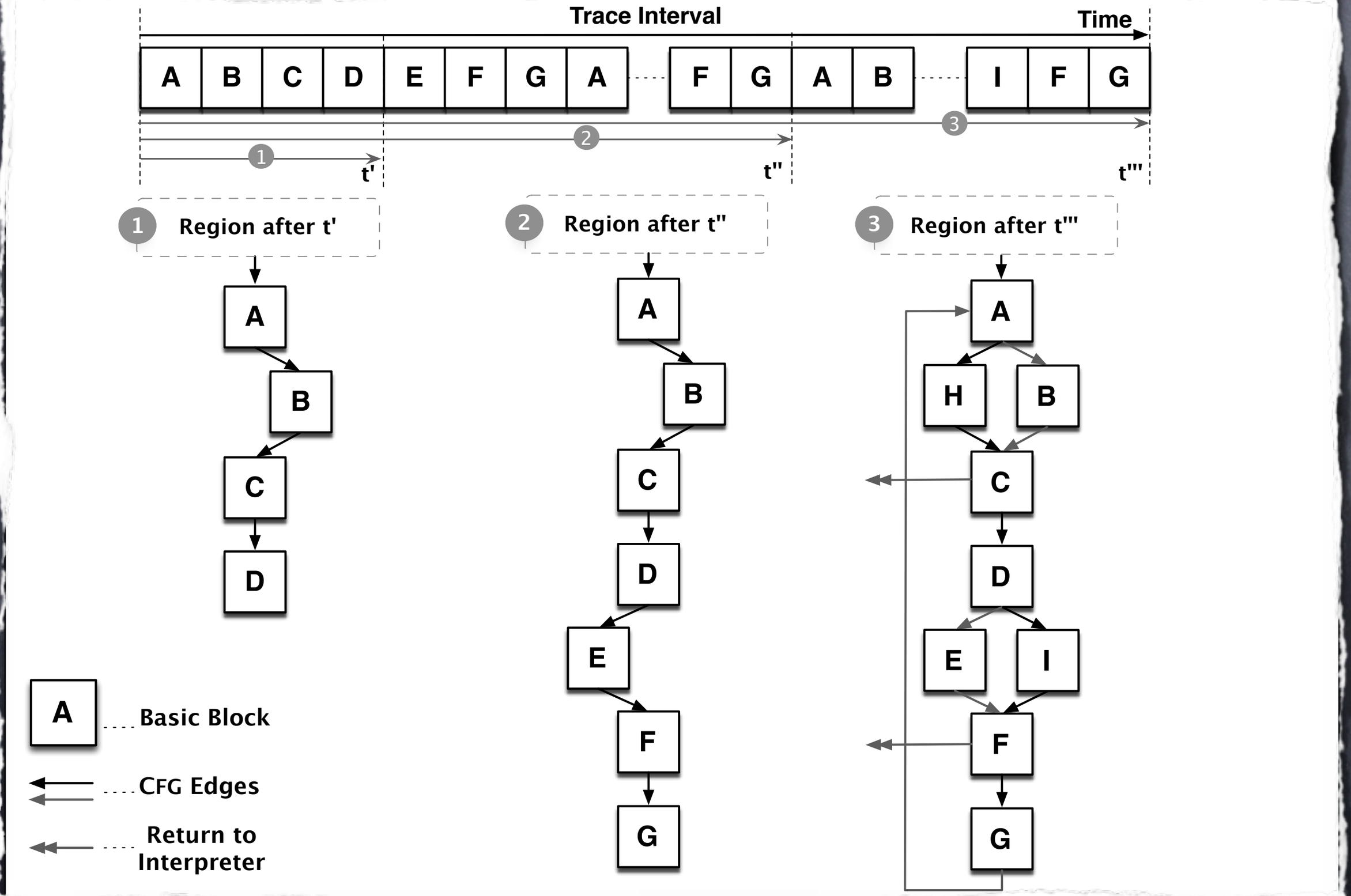
Incremental Code Discovery



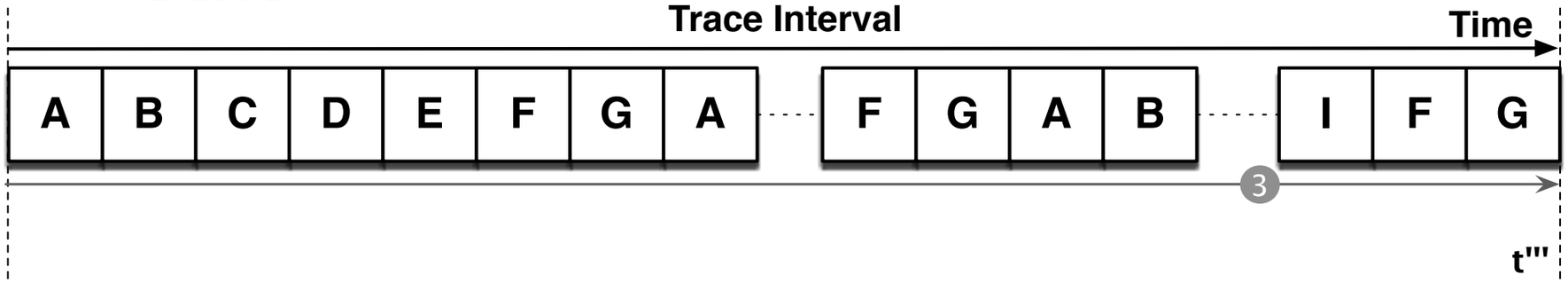
Incremental Code Discovery

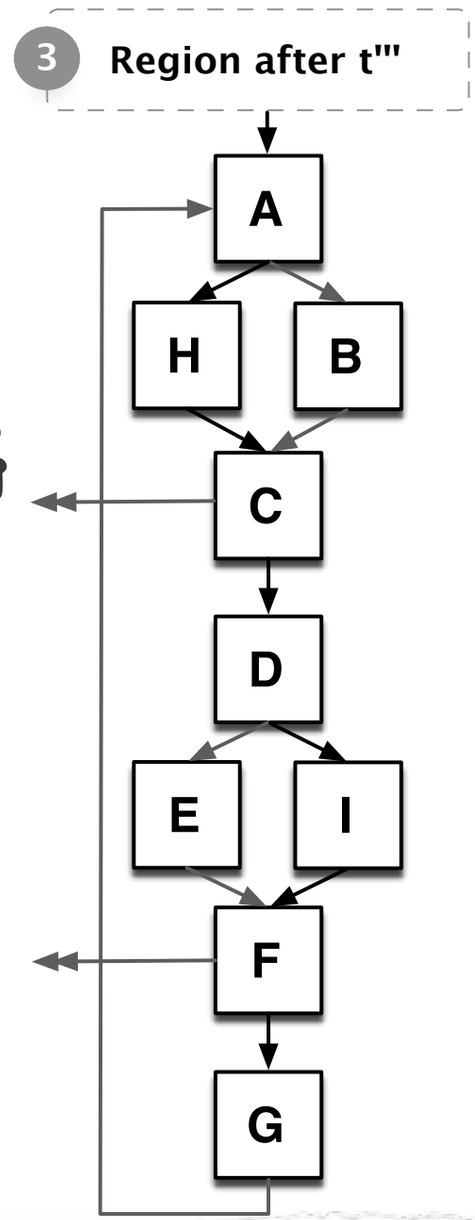


Incremental Code Discovery

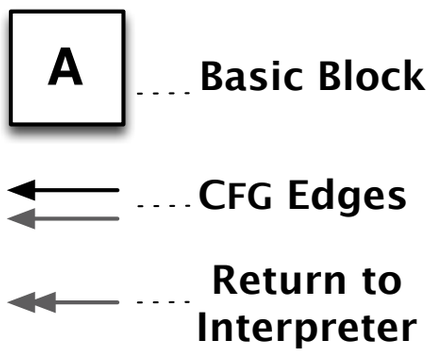


Incremental Code Discovery



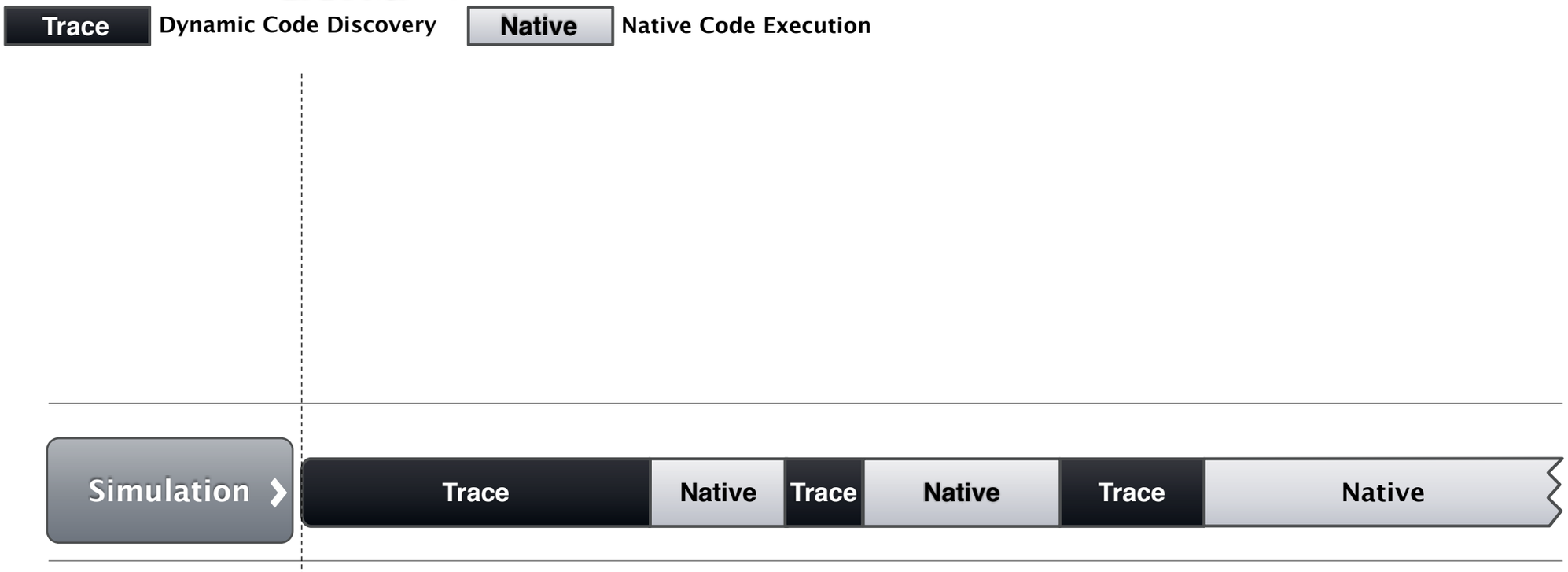


Region == Dynamic CFG



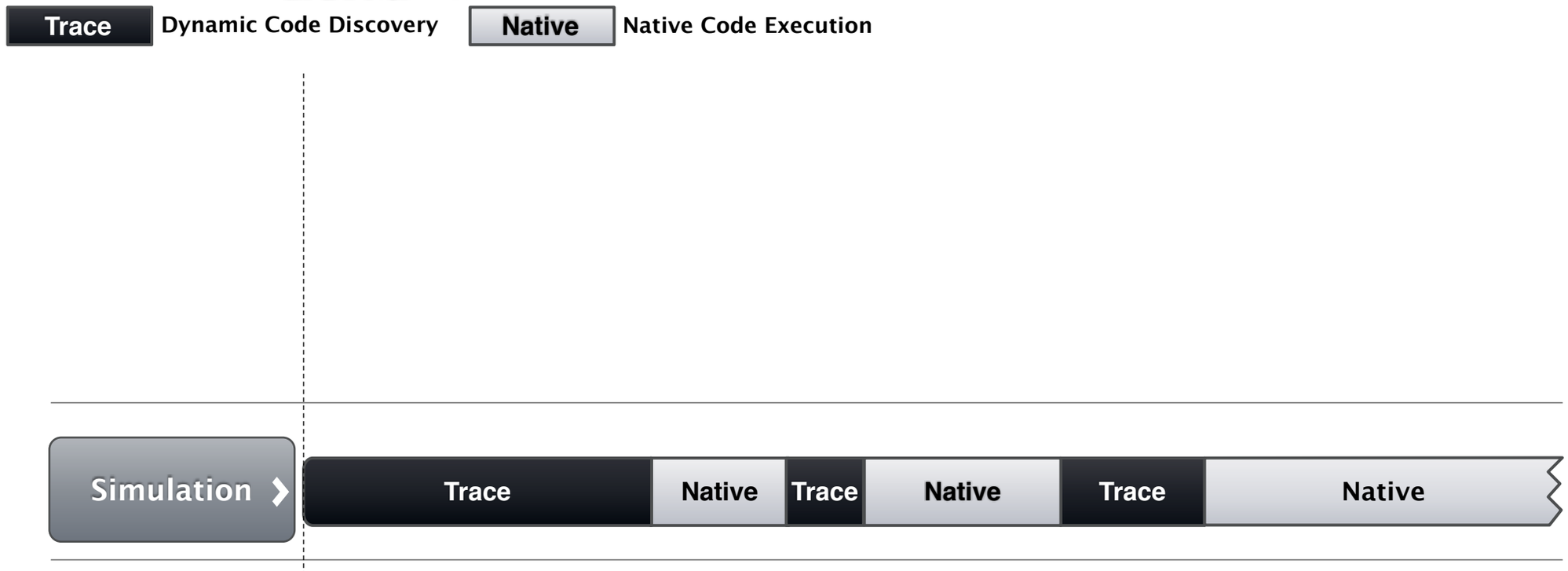
Concurrent and Parallel JIT Compilation in Action

(reducing the critical path)



Concurrent and Parallel JIT Compilation in Action

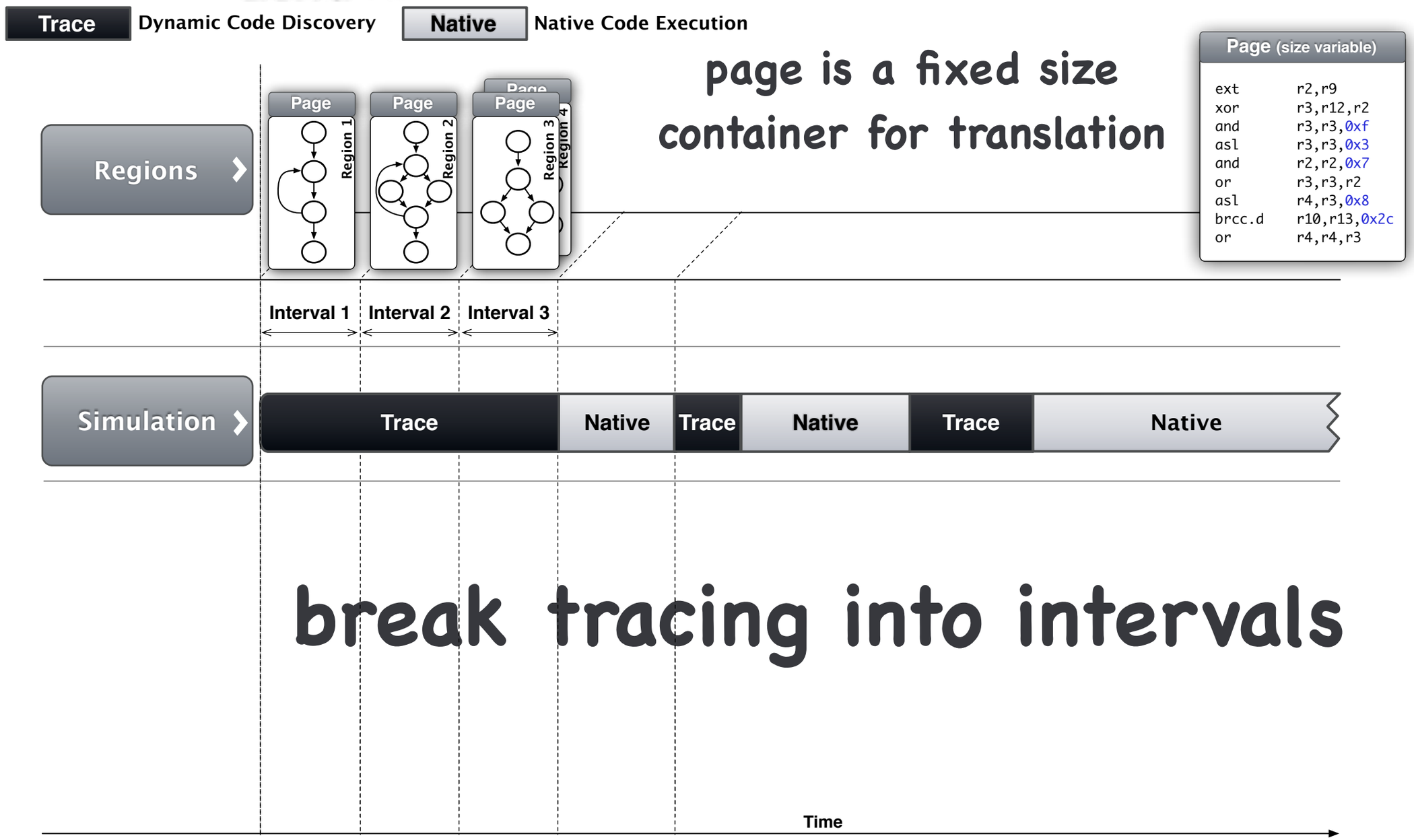
(reducing the critical path)



trace right from the start

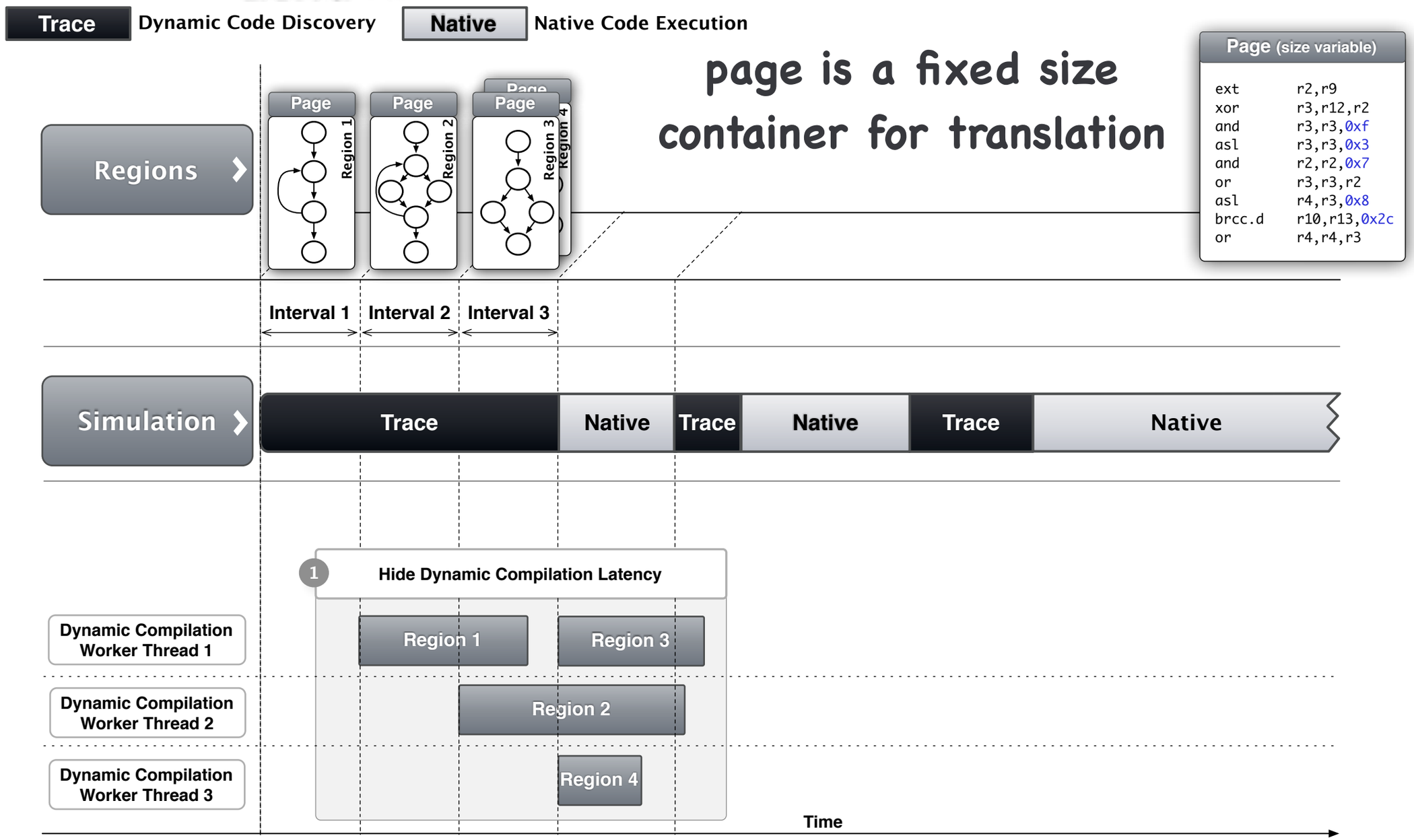
Concurrent and Parallel JIT Compilation in Action

(reducing the critical path)



Concurrent and Parallel JIT Compilation in Action

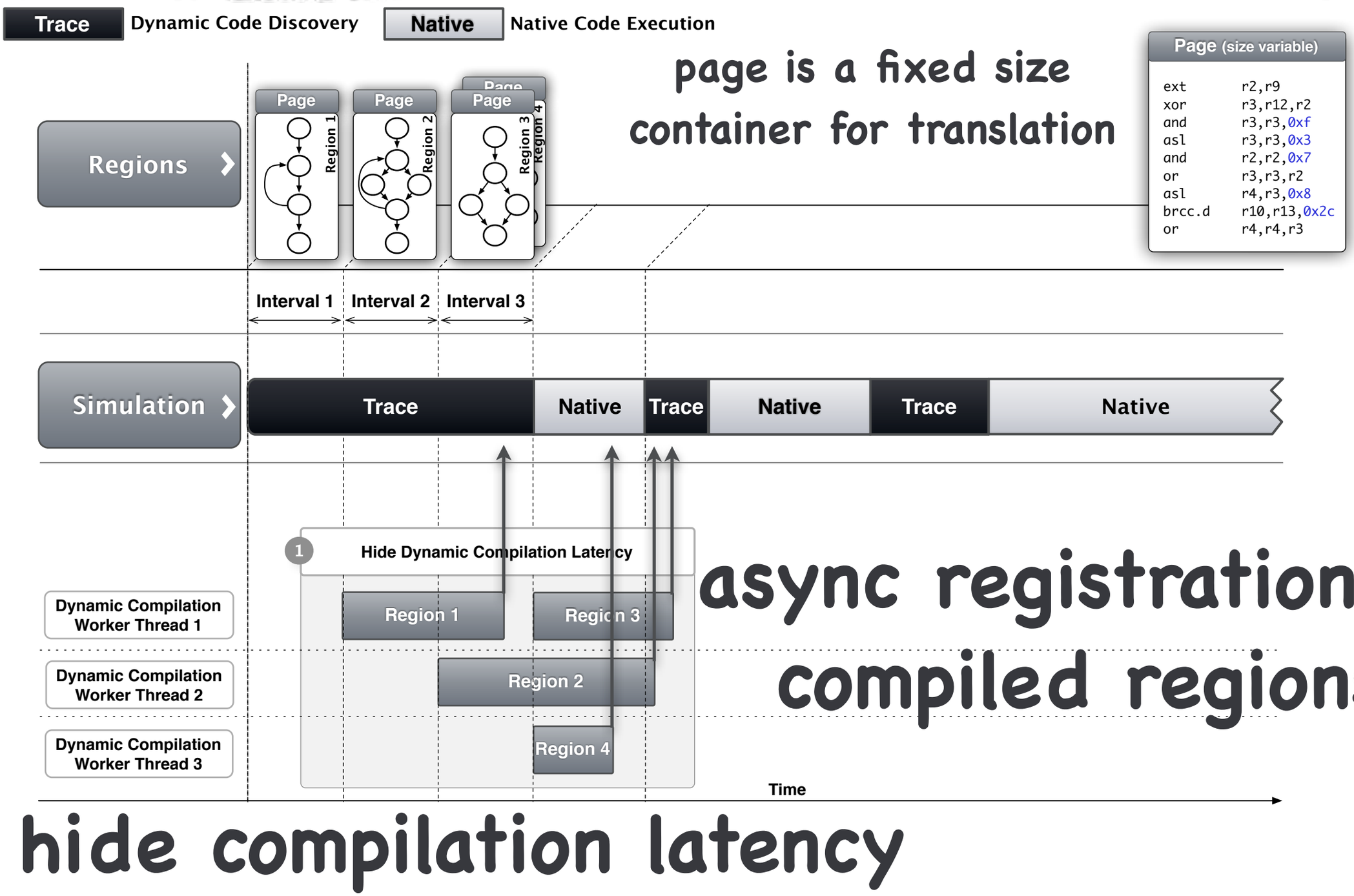
(reducing the critical path)



hide compilation latency

Concurrent and Parallel JIT Compilation in Action

(reducing the critical path)

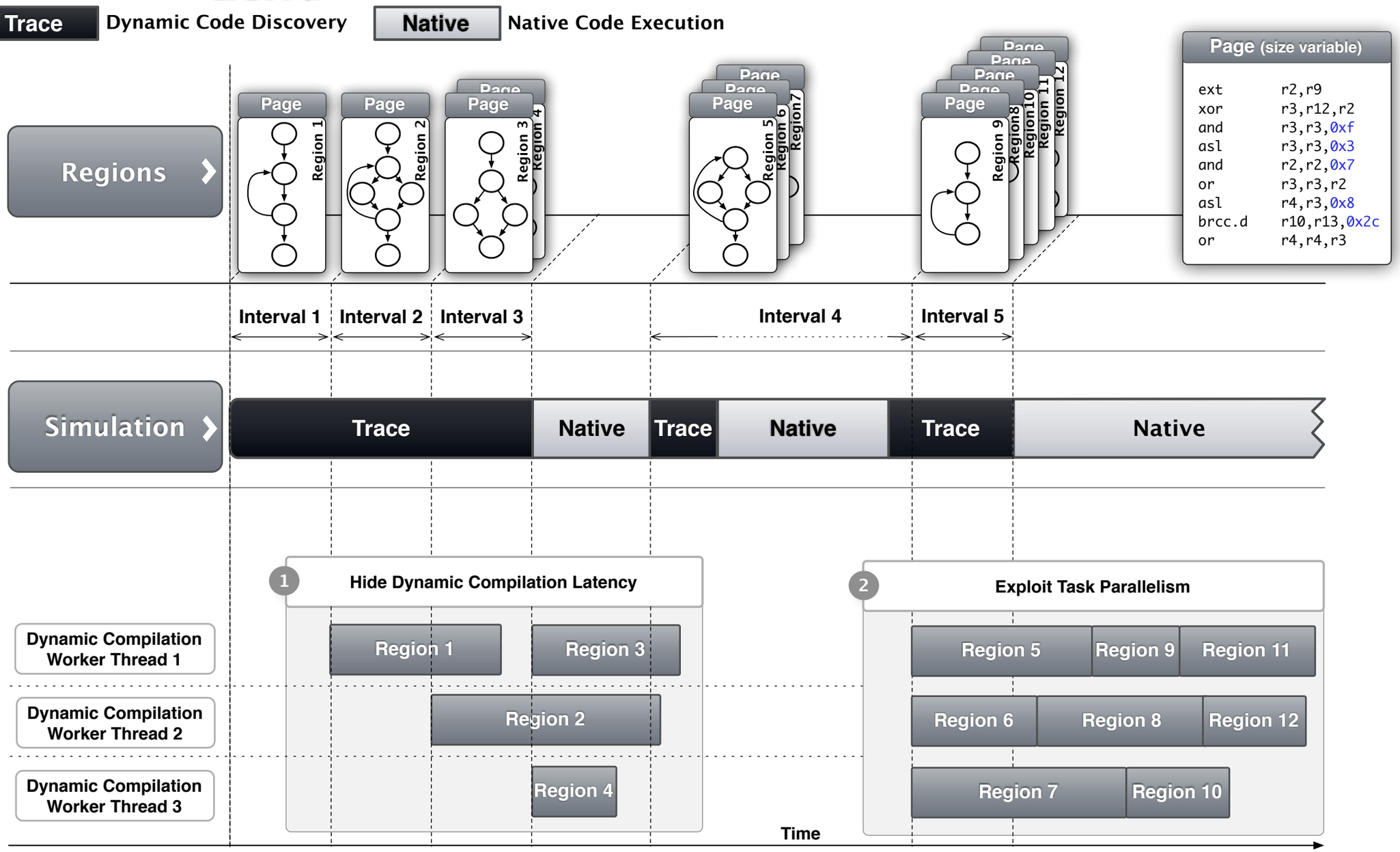


async registration of compiled regions

hide compilation latency

Concurrent and Parallel JIT Compilation in Action

(reducing the critical path)

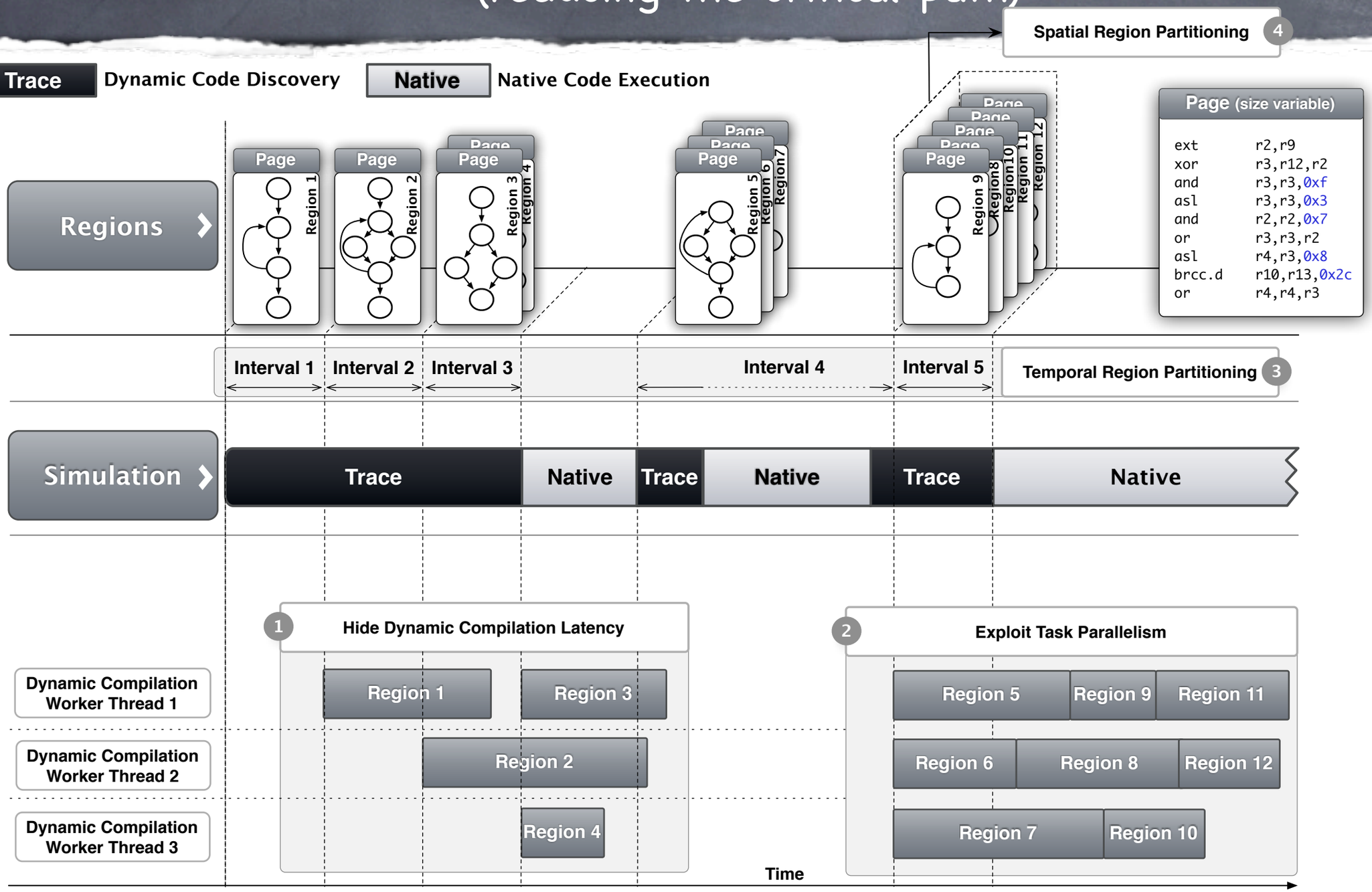


hide compilation latency

exploit task parallelism

Concurrent and Parallel JIT Compilation in Action

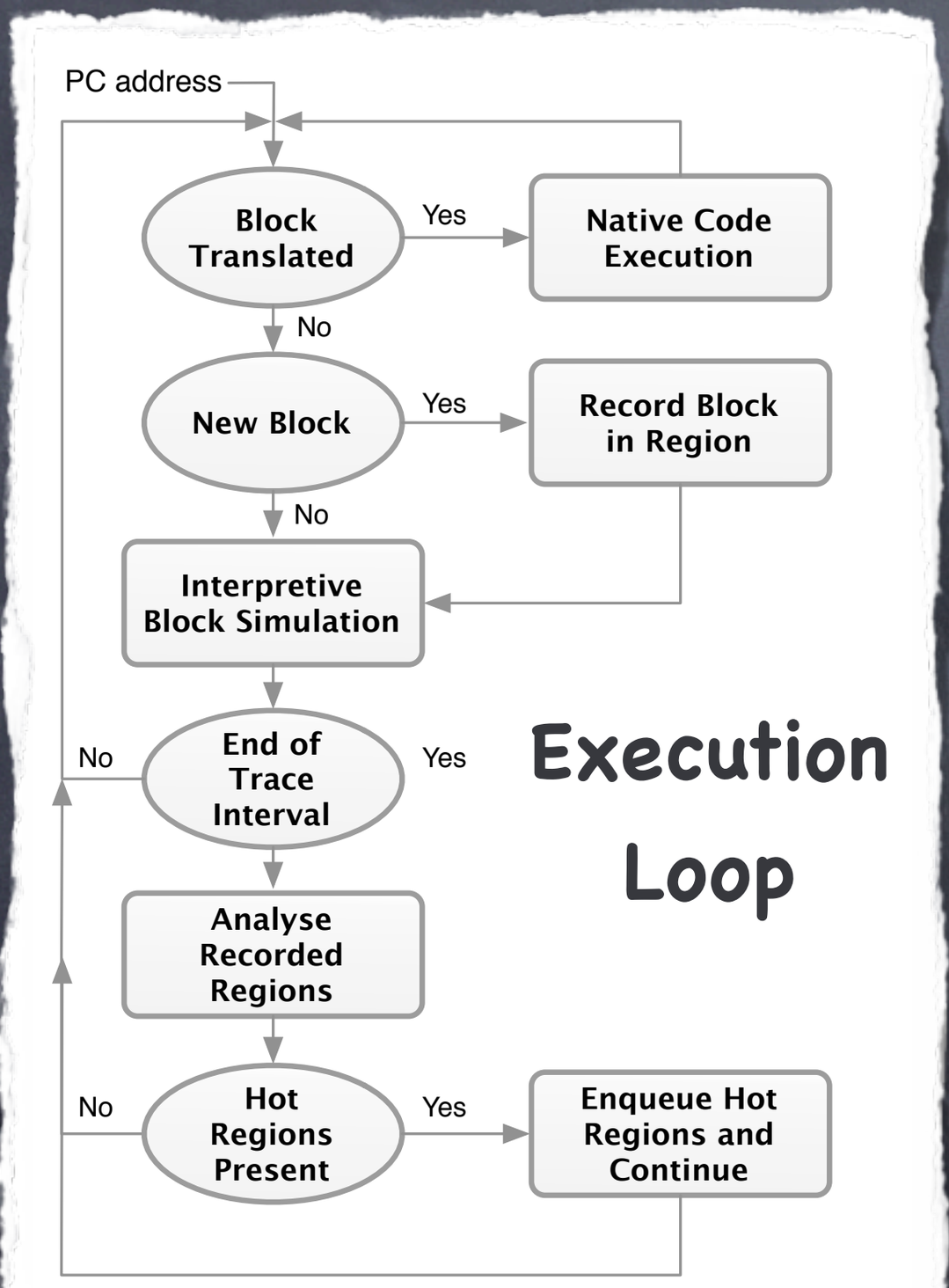
(reducing the critical path)



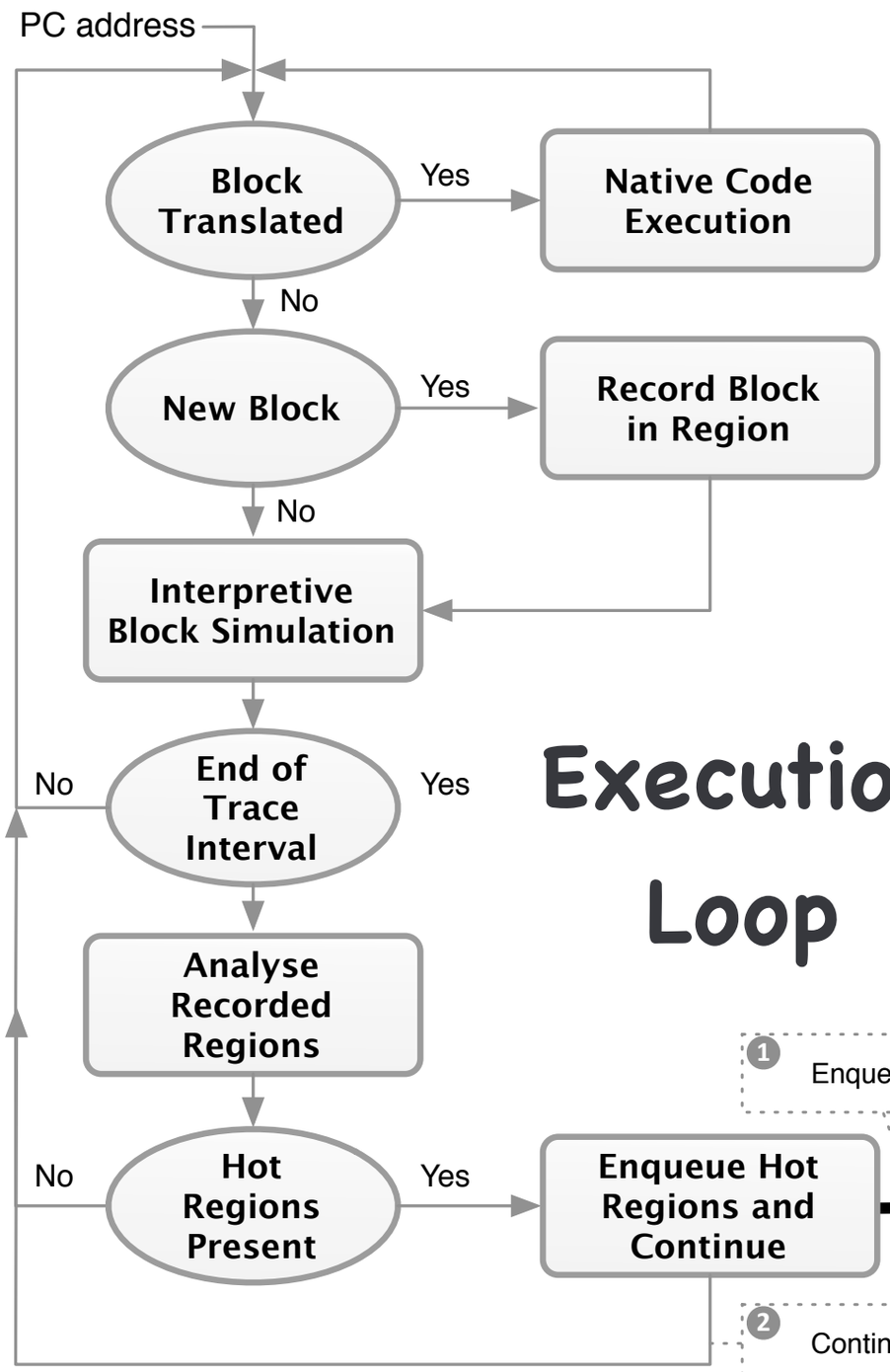
hide compilation latency

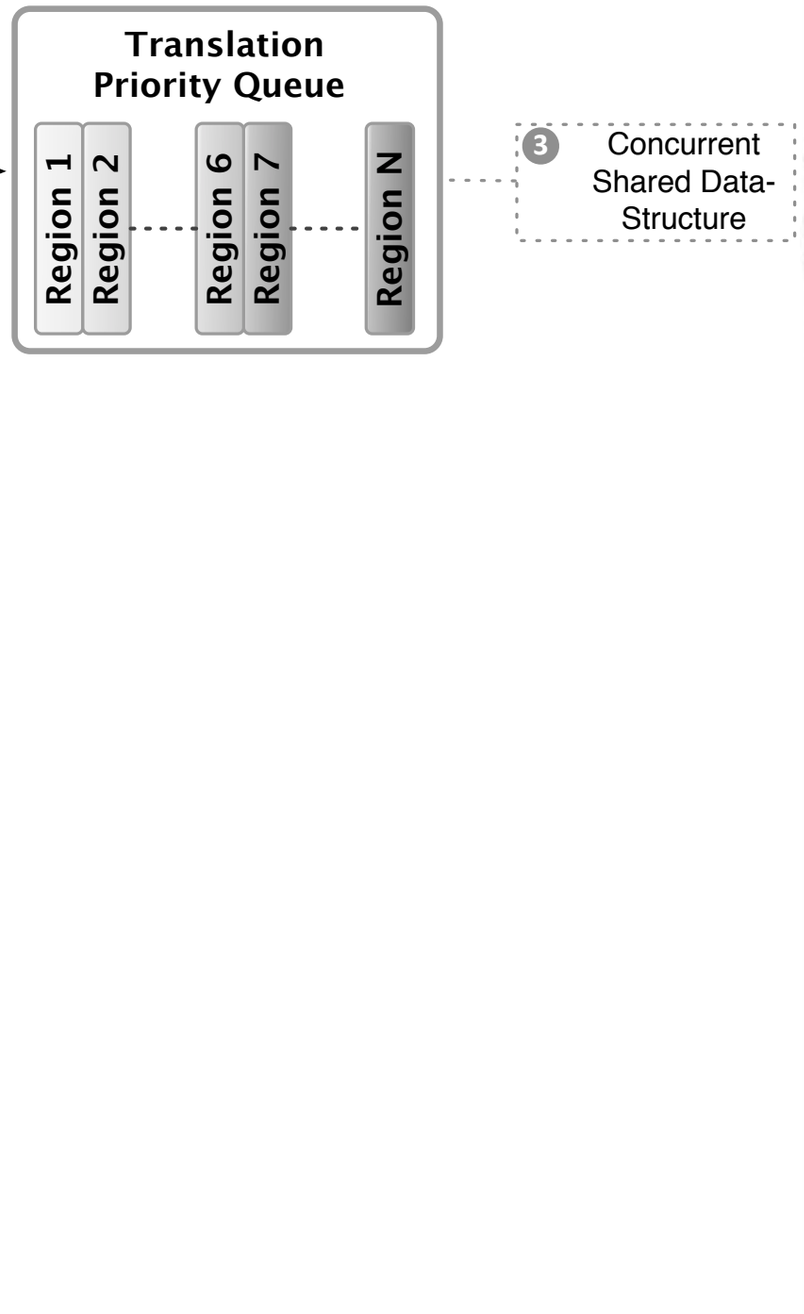
exploit task parallelism

Concurrent and Parallel JIT Compiler Design

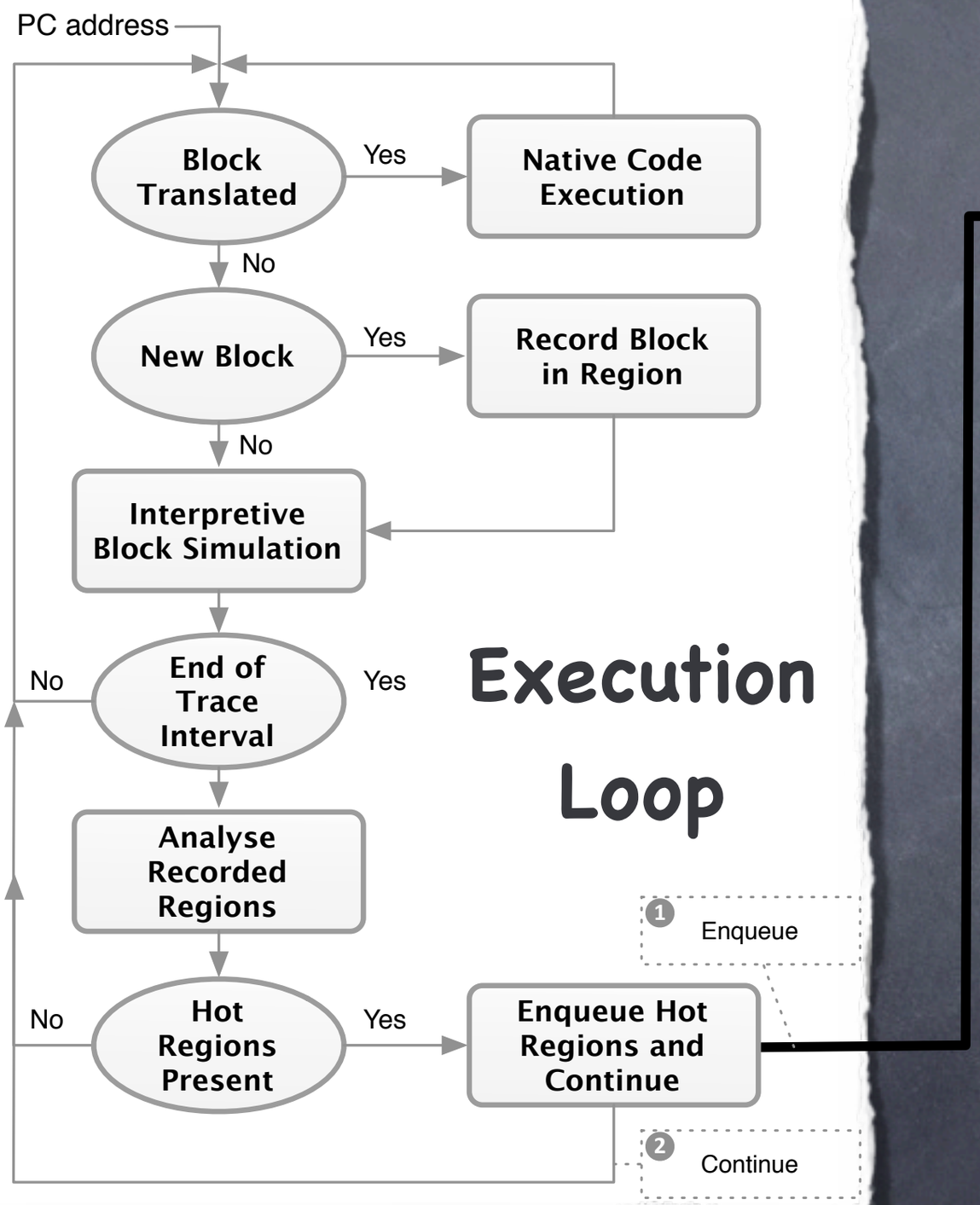


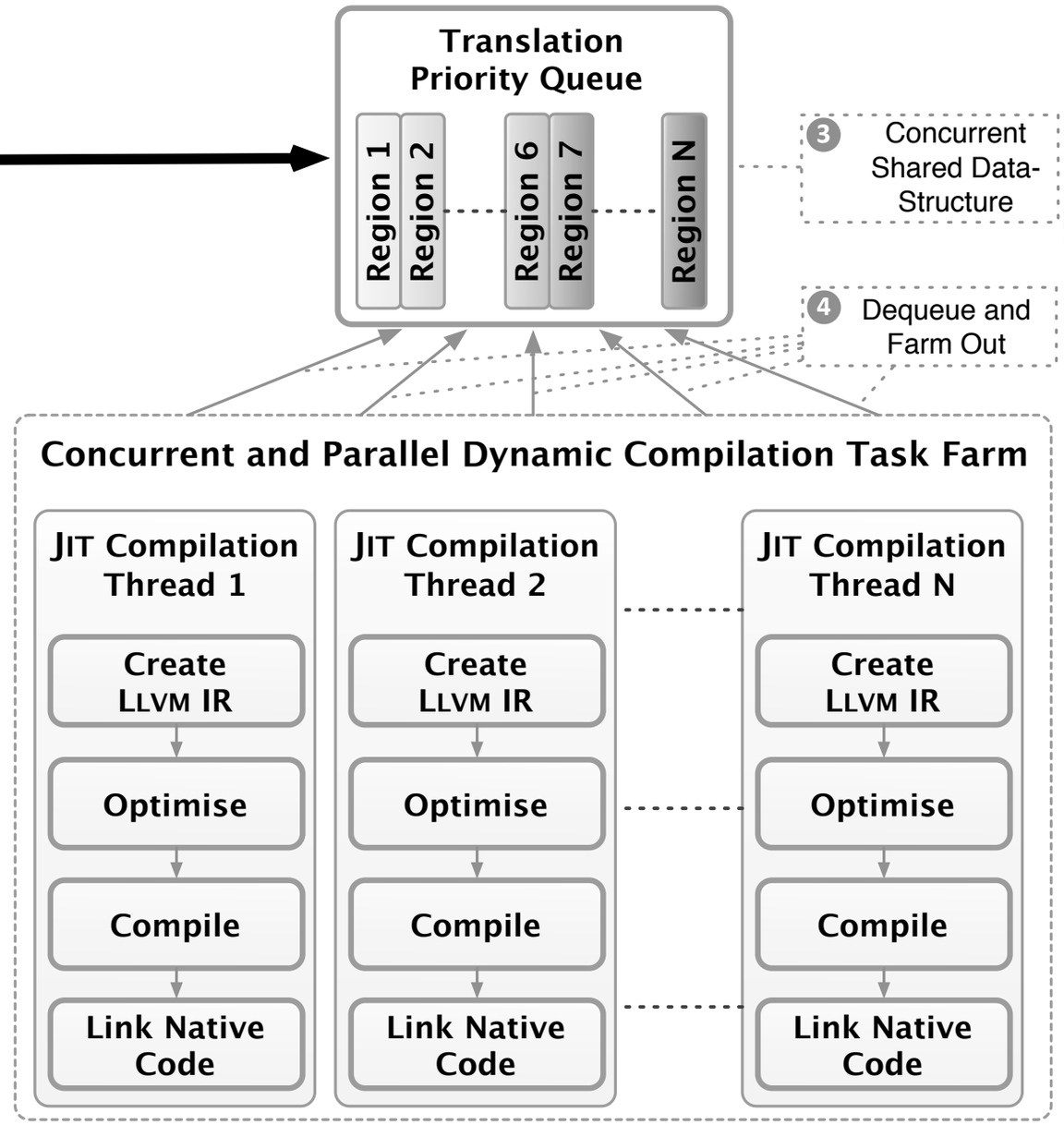
Concurrent and Parallel JIT Compiler Design





Concurrent and Parallel JIT Compiler Design

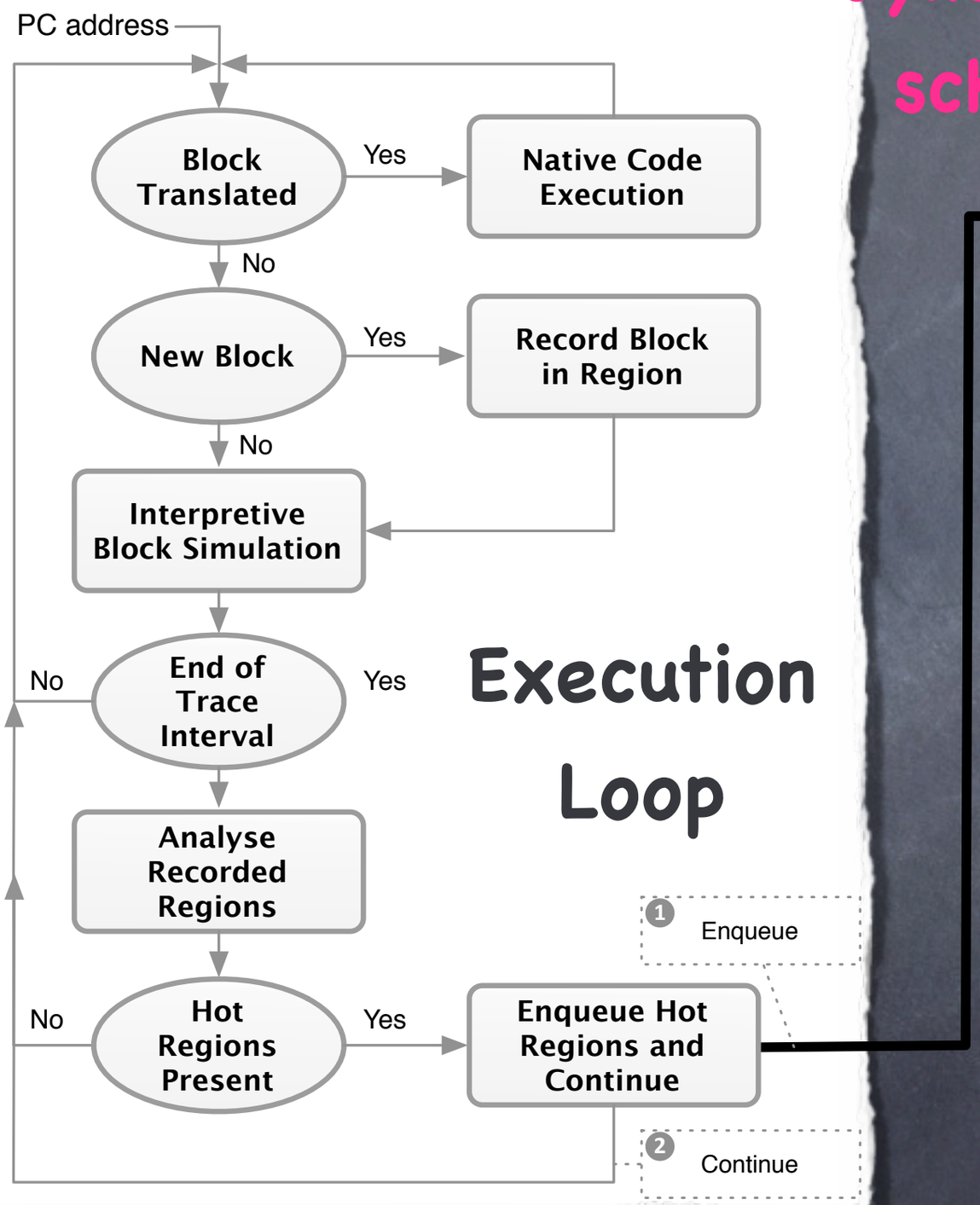




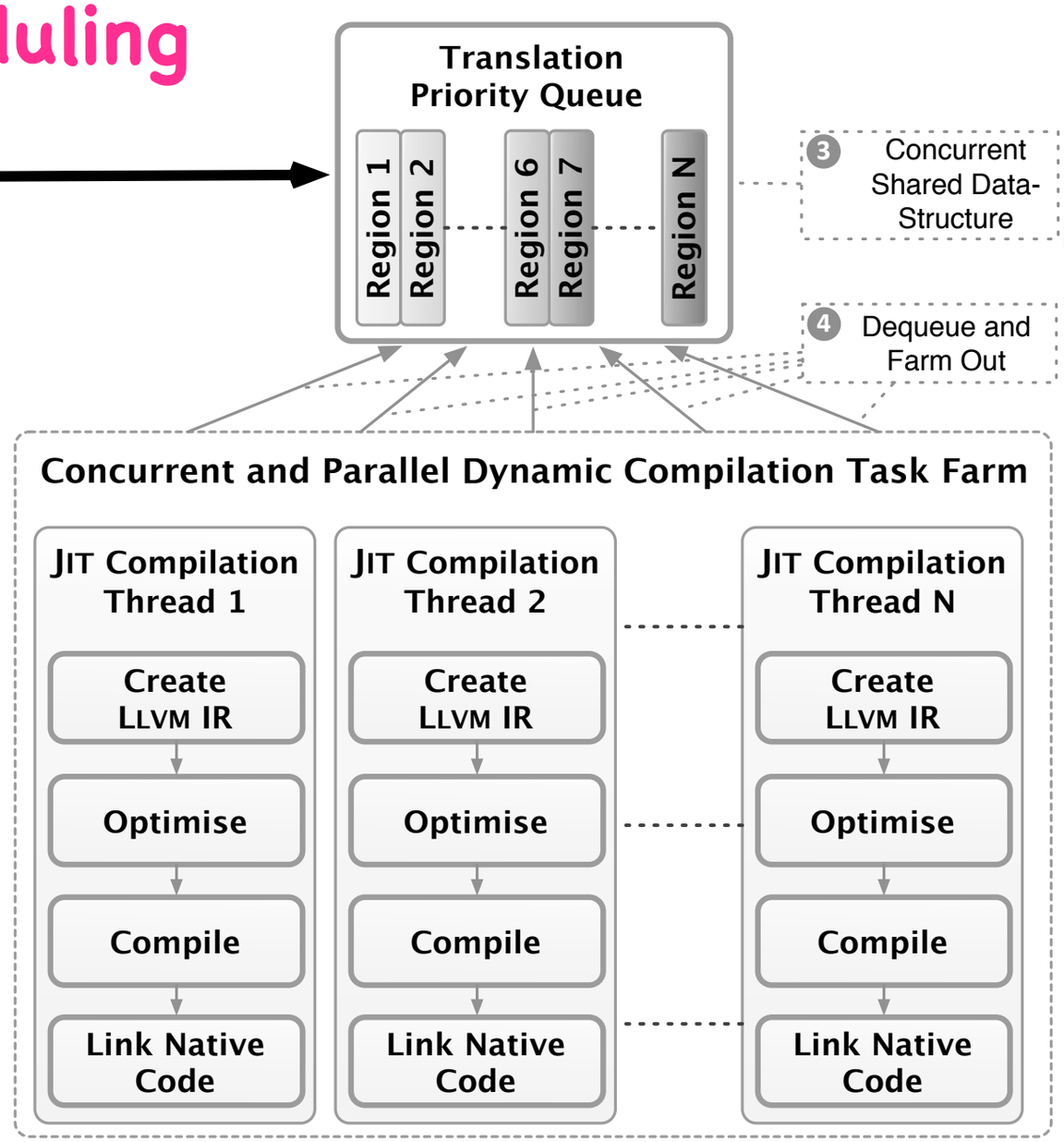
JIT Compilation Task Farm

Concurrent and Parallel JIT Compiler Design

dynamic work scheduling

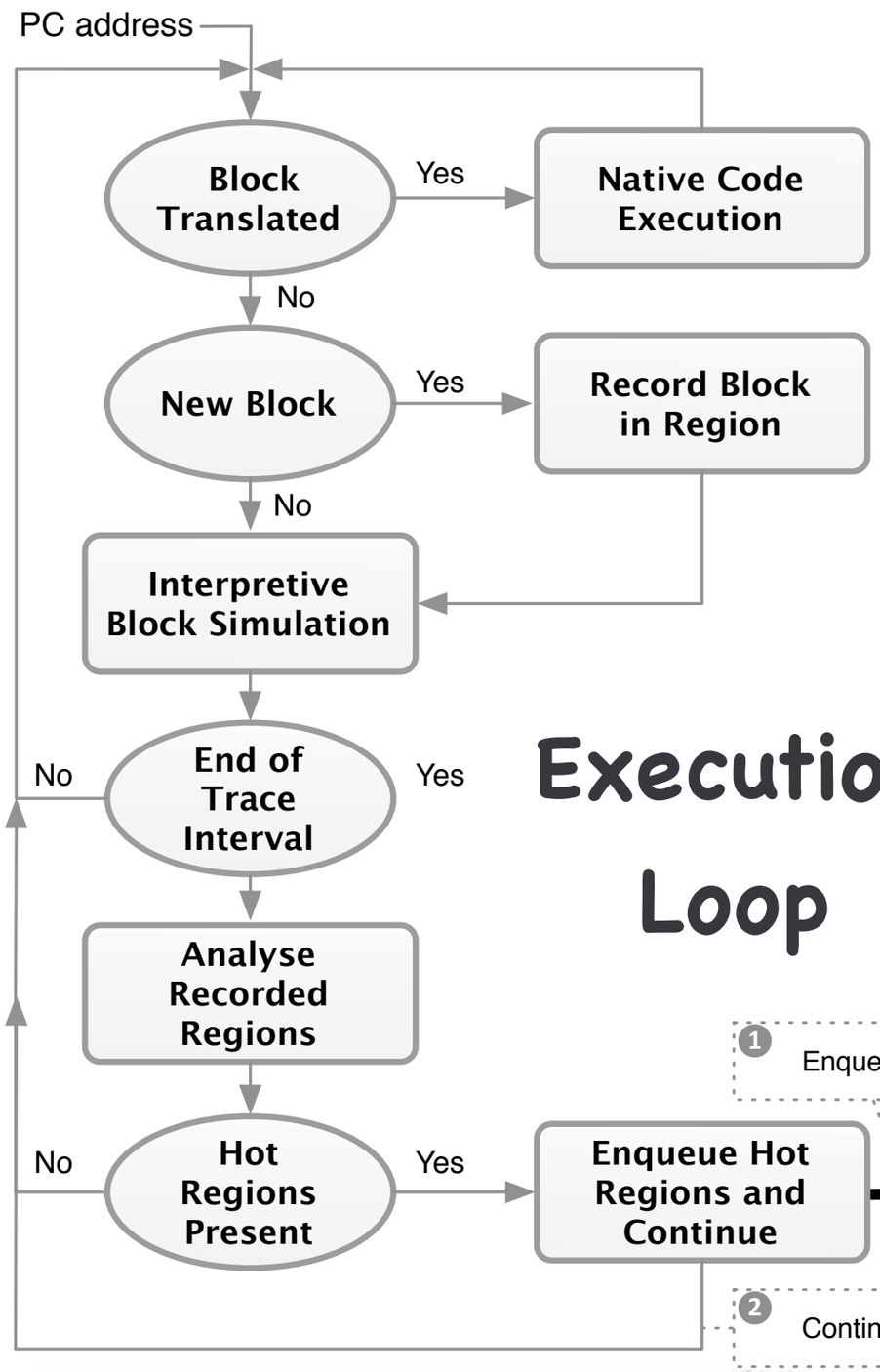


Execution Loop



JIT Compilation Task Farm

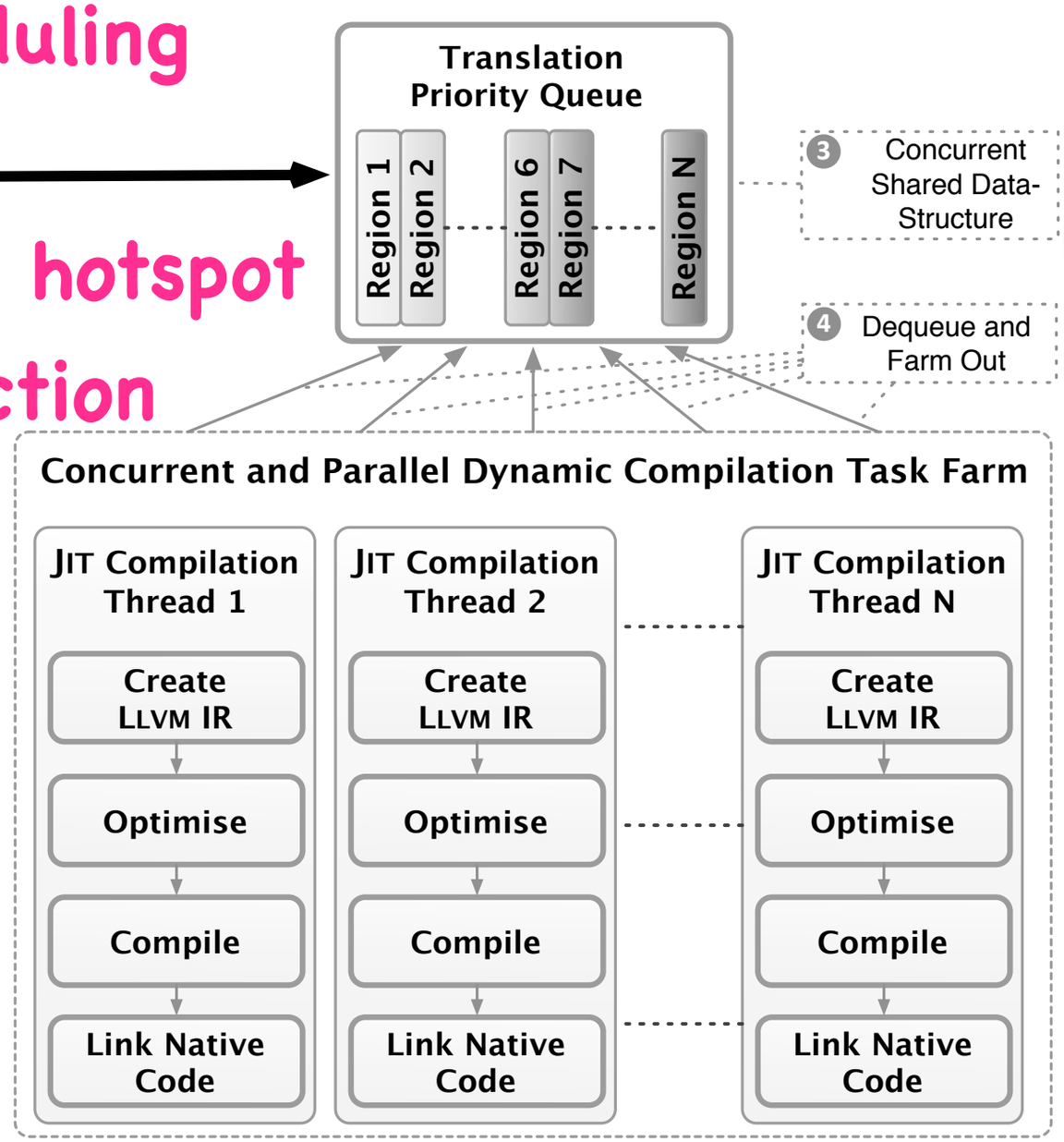
Concurrent and Parallel JIT Compiler Design



Execution Loop

dynamic work scheduling

adaptive hotspot selection



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();
      ...
  }

}
```

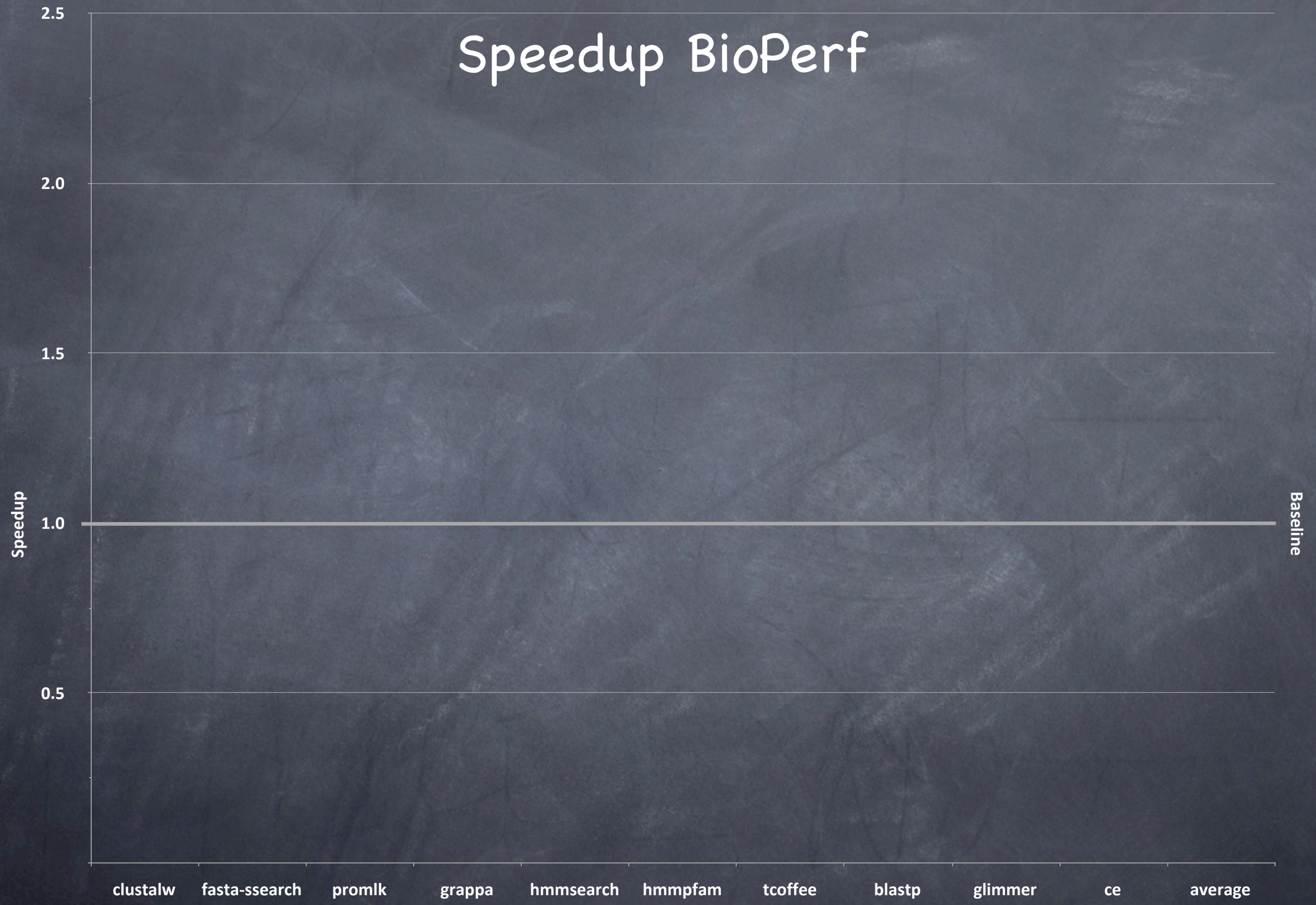
Concurrent and Parallel JIT Compiler Design Based on LLVM

```
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

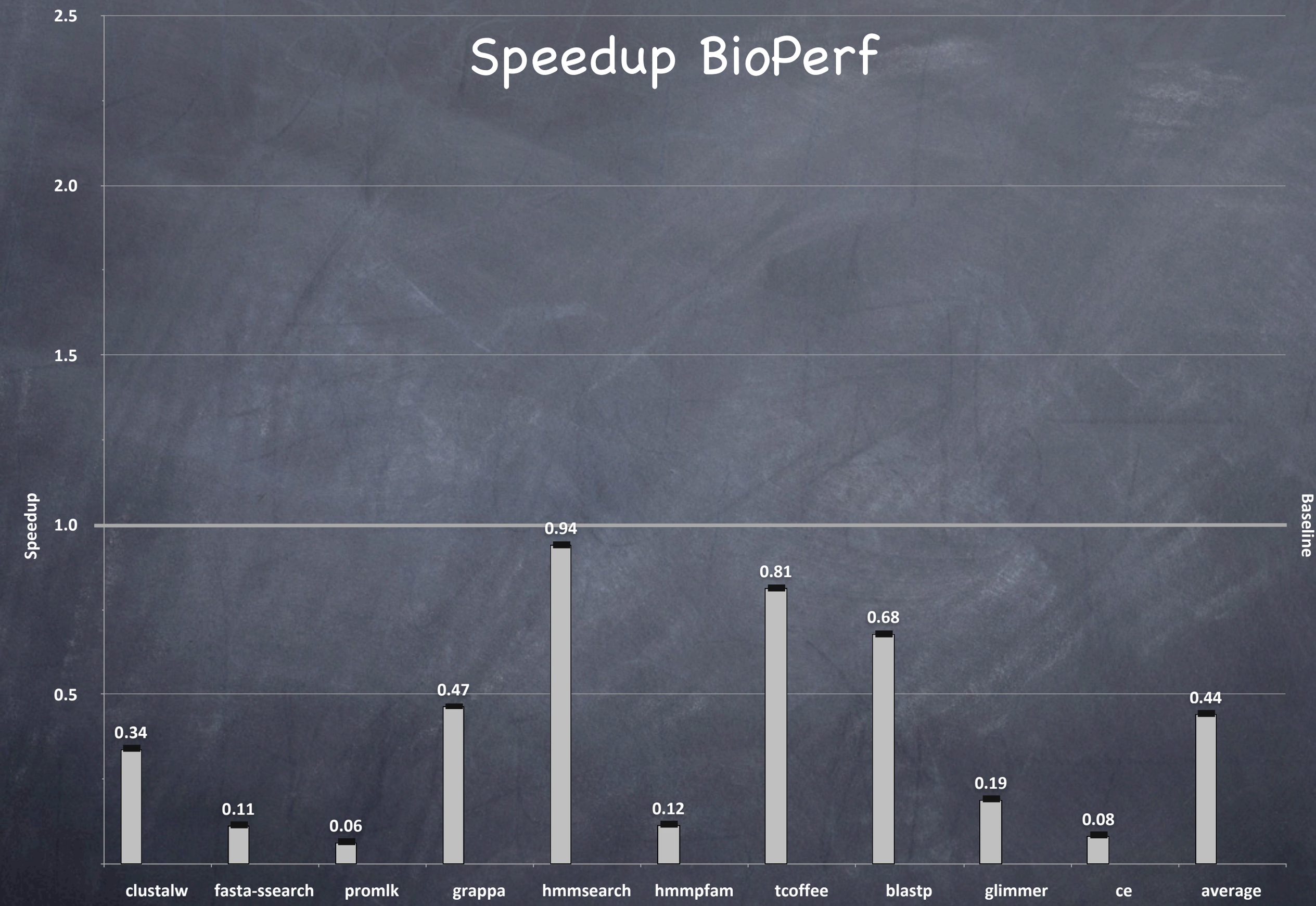
- Extensive evaluation using over 60 industry standard benchmarks built for ARCompact RISC platform:
  - BioPERF
  - SPEC CPU 2006
  - EEMBC and CoreMark
- Target Platform:
  - ARCompact RISC ISA targeting ARC 700 processor
- Simulation Platform:
  - standard x86 Dell Intel Xeon quad-core machine

Speedup BioPerf



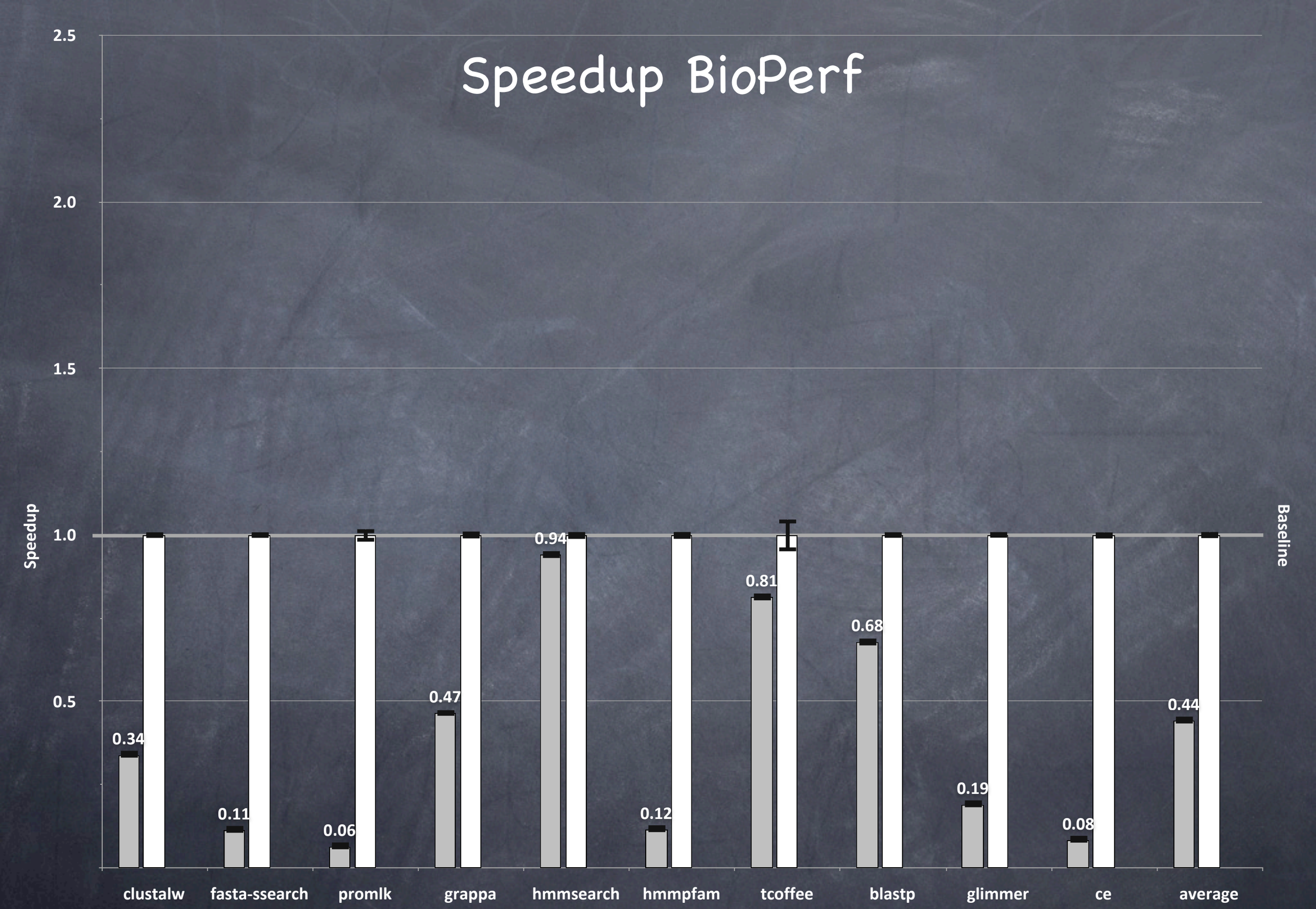
Measured on standard x86 quad-core machine

Speedup BioPerf



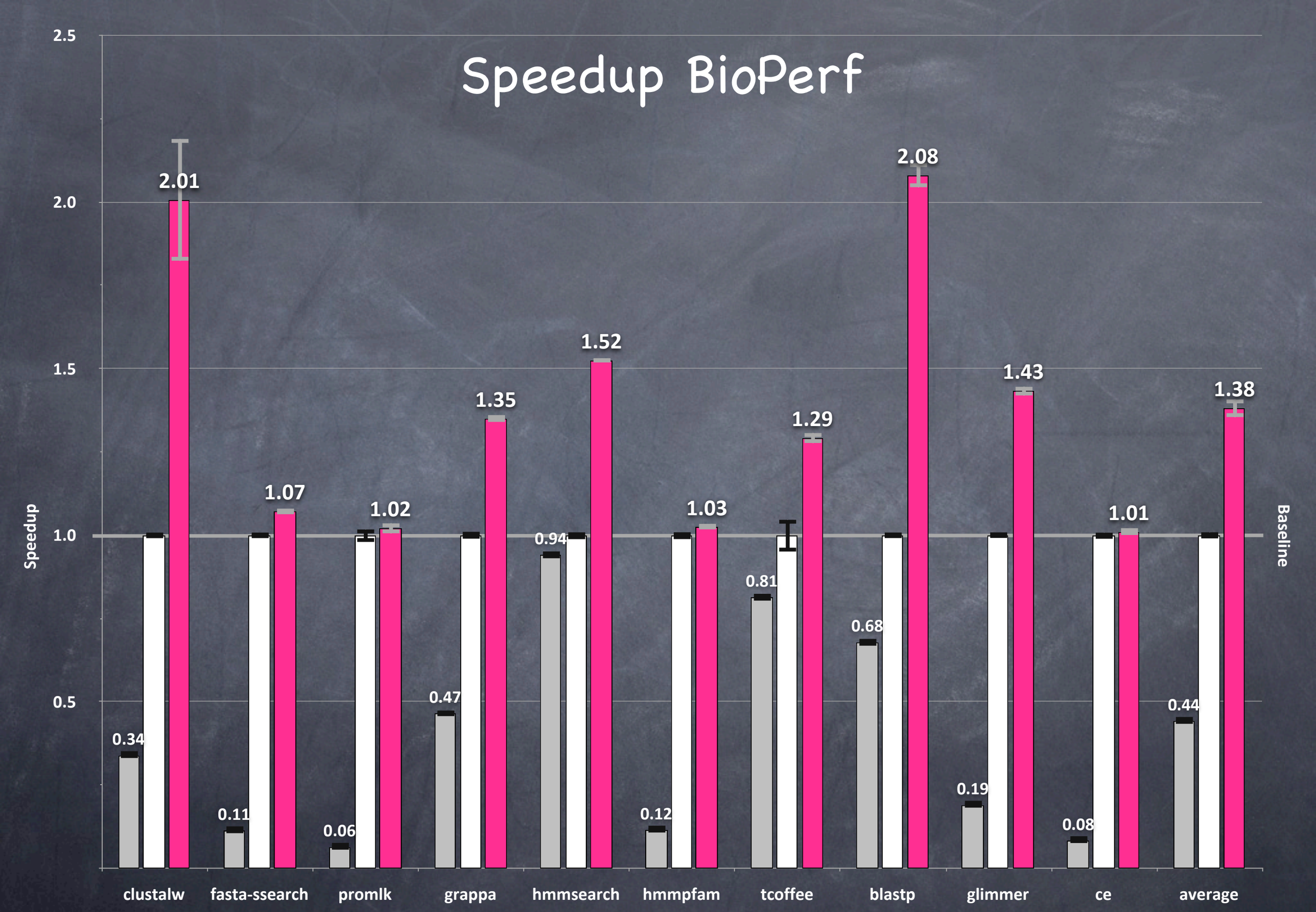
Measured on standard x86 quad-core machine

Speedup BioPerf



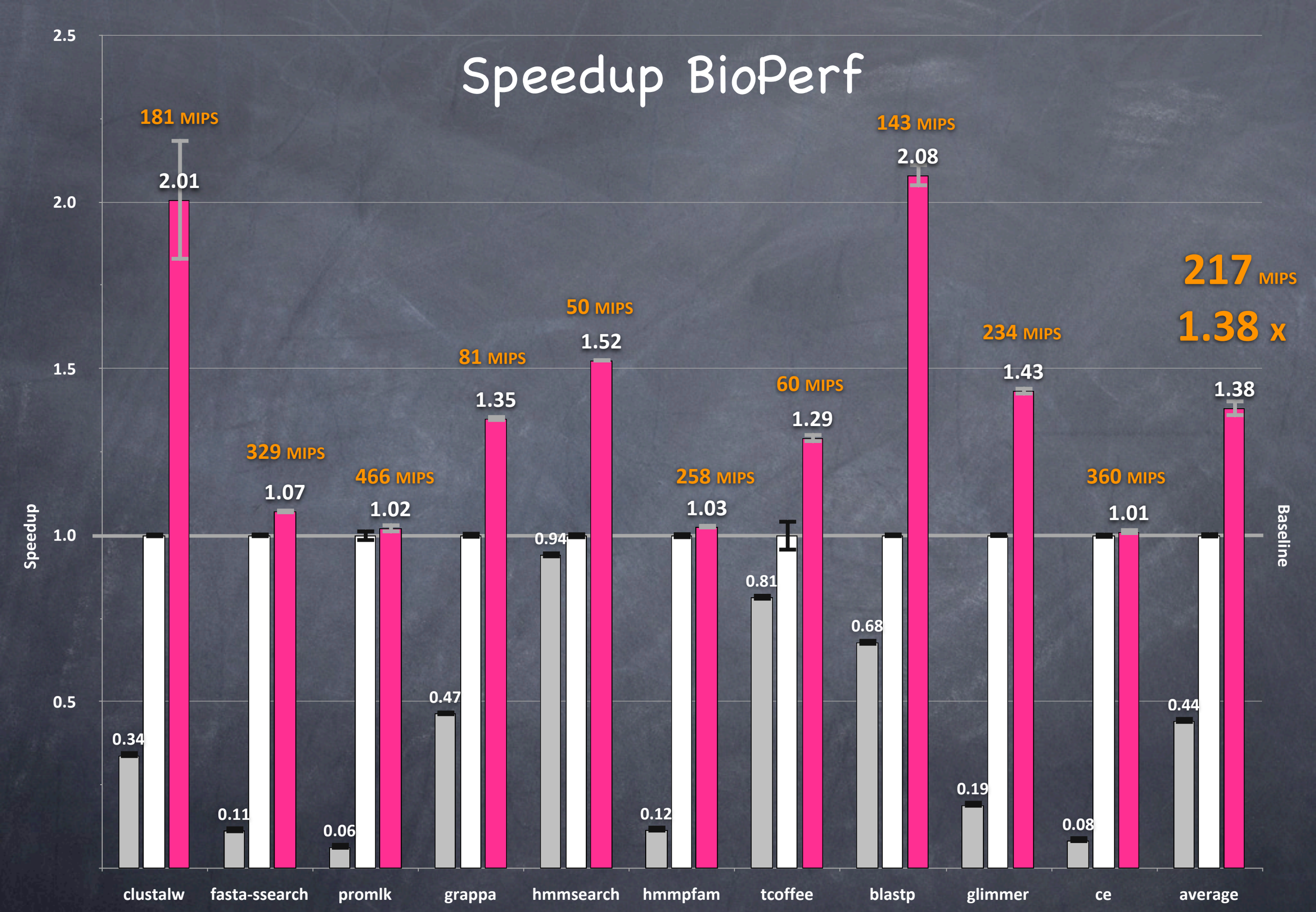
Measured on standard x86 quad-core machine

Speedup BioPerf



Measured on standard x86 quad-core machine

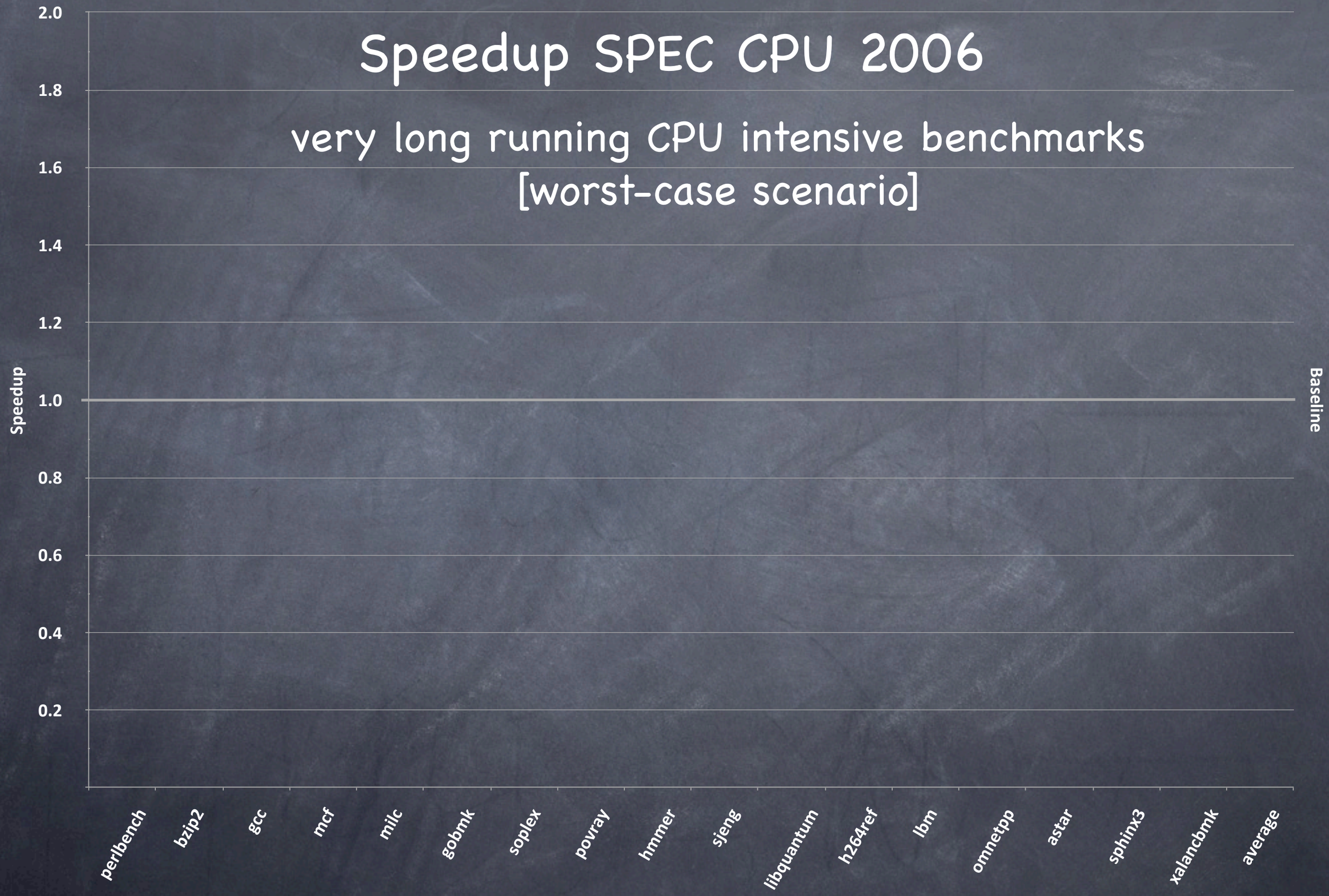
Speedup BioPerf



Measured on standard x86 quad-core machine

Speedup SPEC CPU 2006

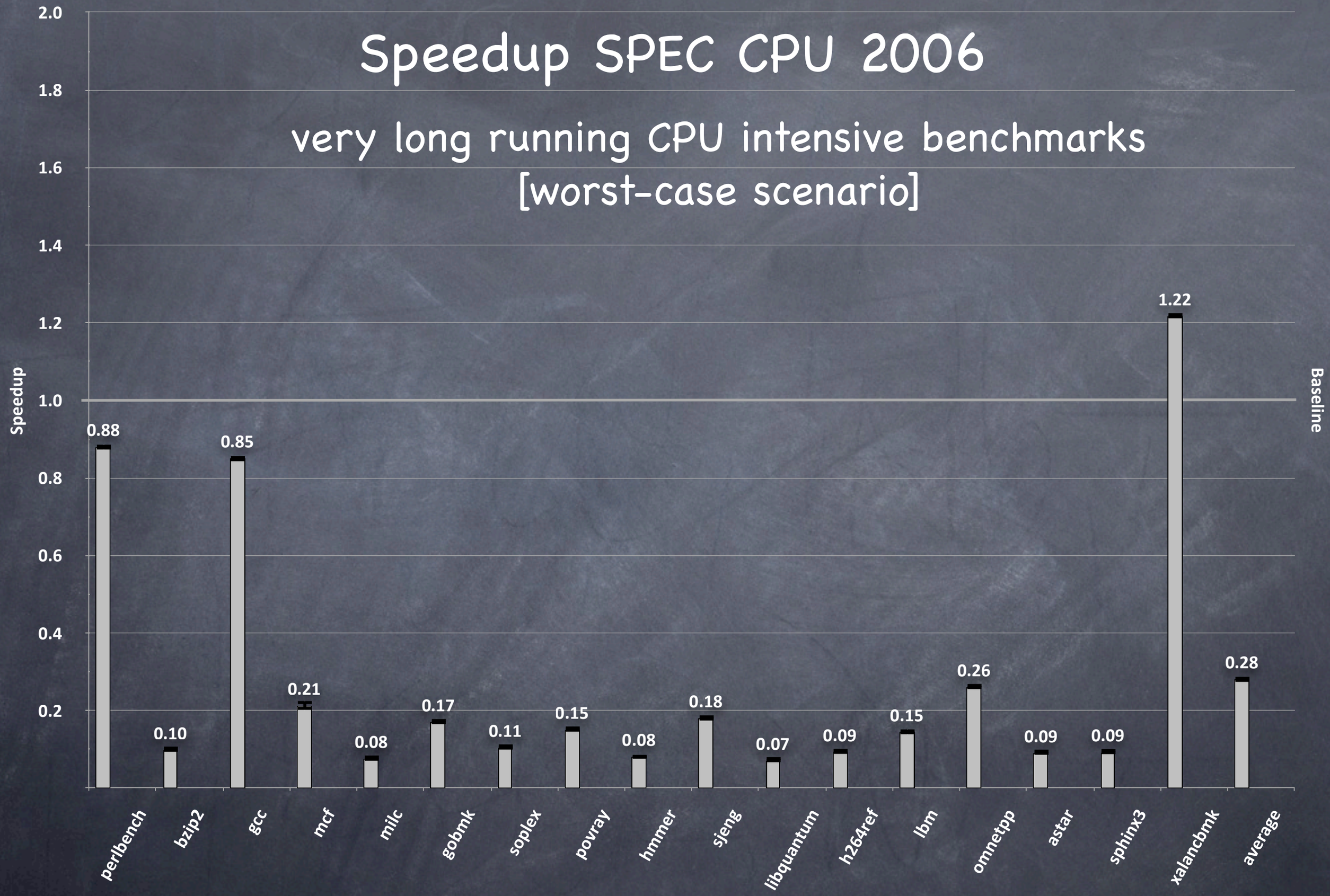
very long running CPU intensive benchmarks
[worst-case scenario]



Measured on standard x86 quad-core machine

Speedup SPEC CPU 2006

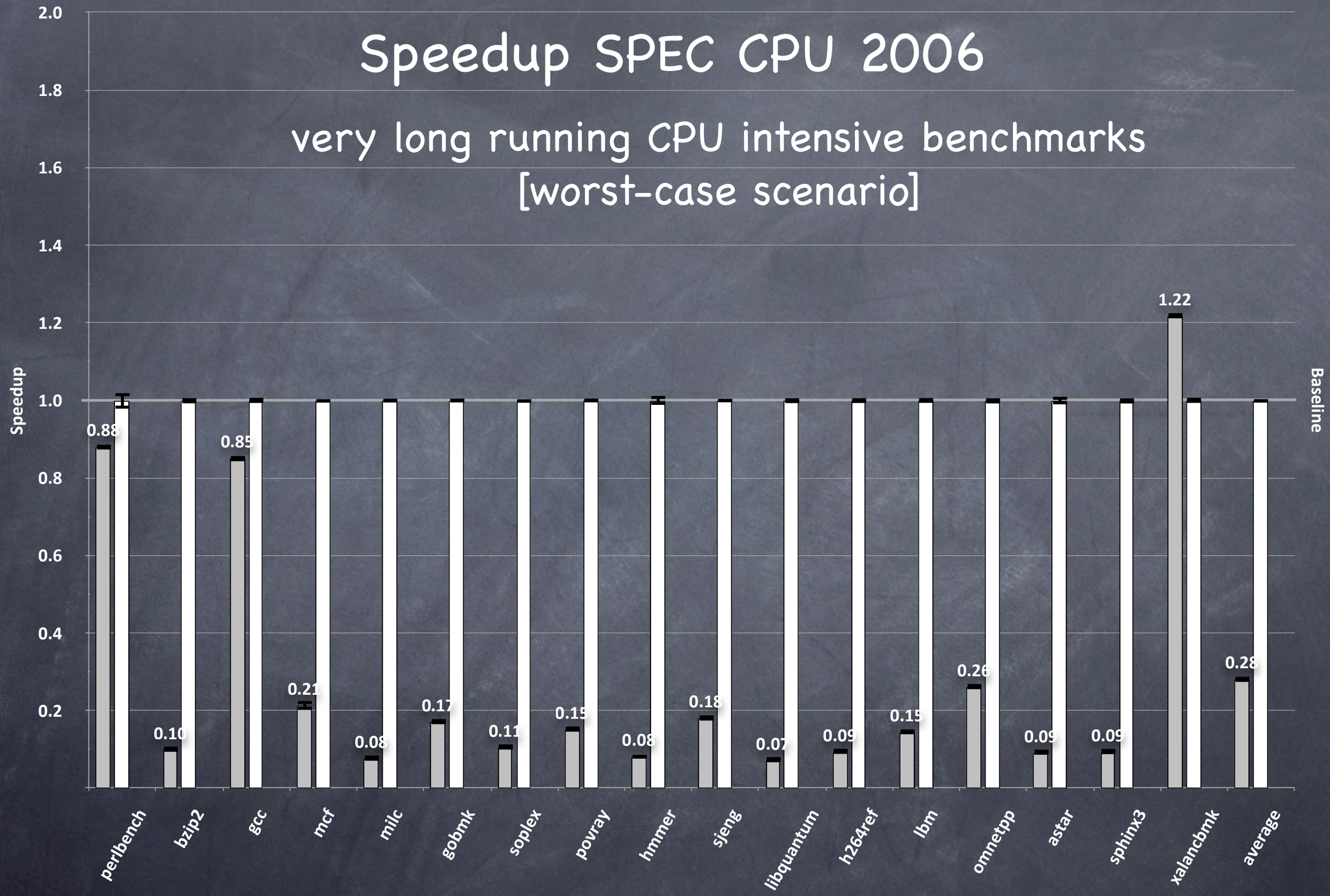
very long running CPU intensive benchmarks
[worst-case scenario]



Measured on standard x86 quad-core machine

Speedup SPEC CPU 2006

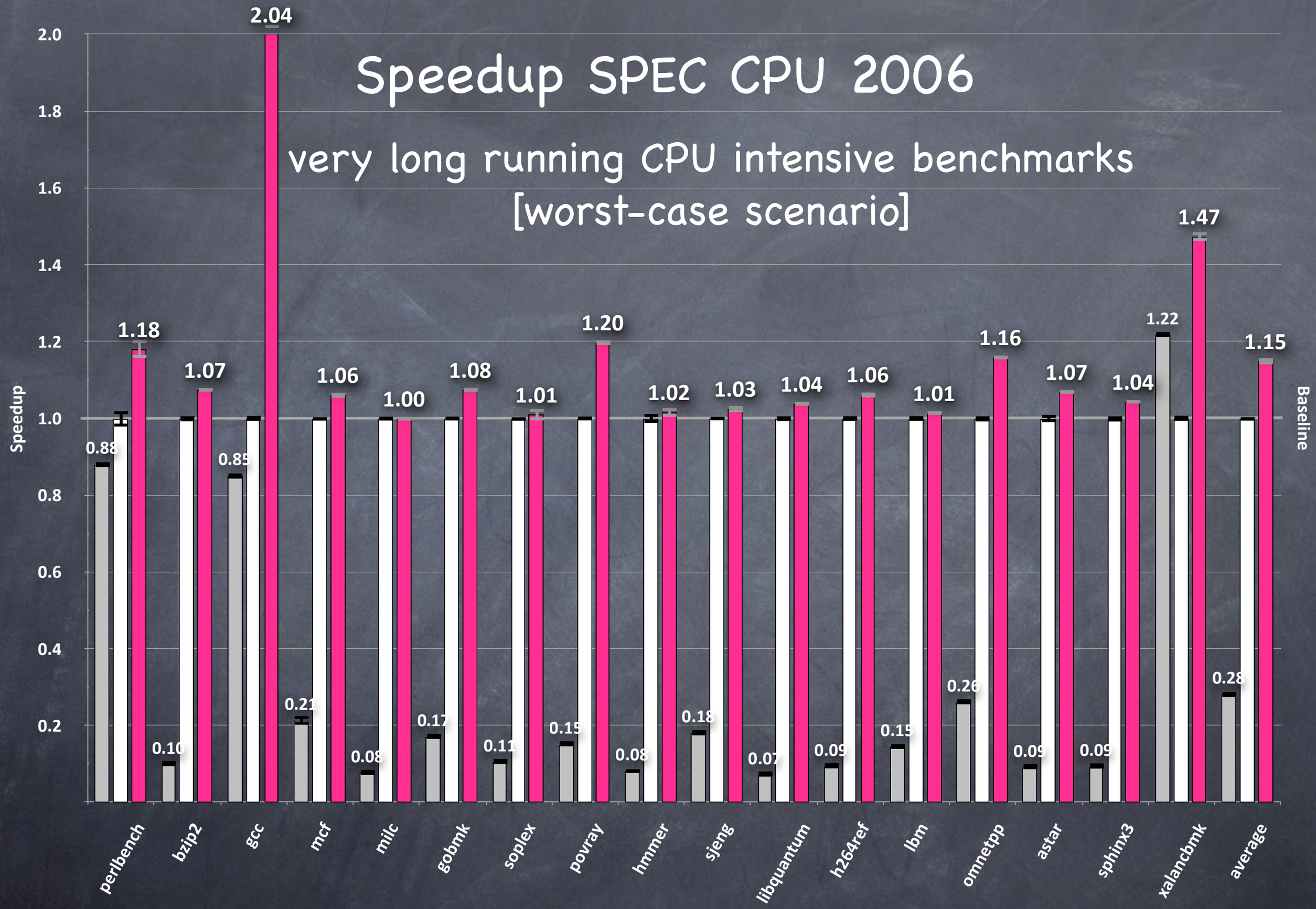
very long running CPU intensive benchmarks
[worst-case scenario]



Measured on standard x86 quad-core machine

Speedup SPEC CPU 2006

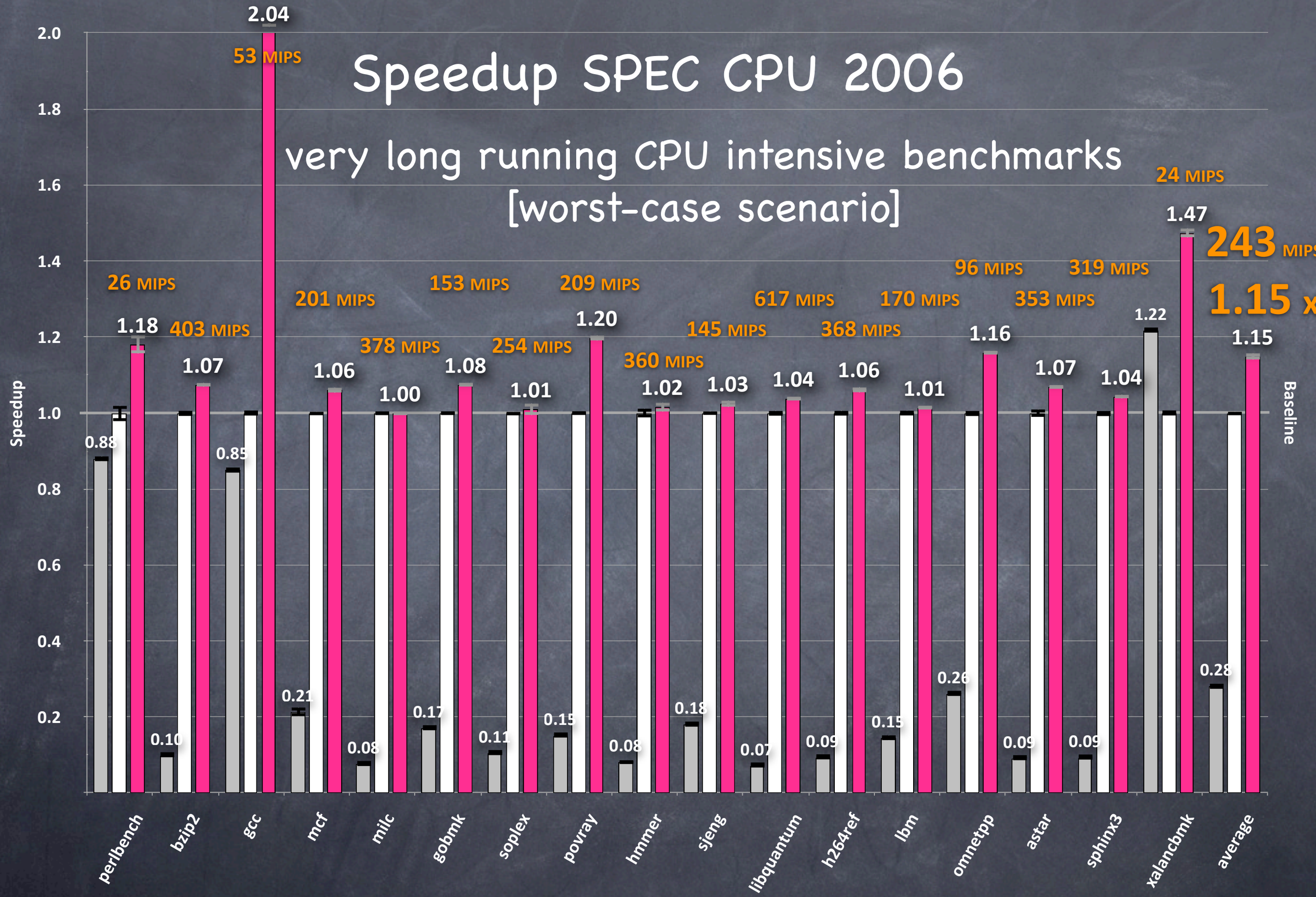
very long running CPU intensive benchmarks
[worst-case scenario]



Measured on standard x86 quad-core machine

Speedup SPEC CPU 2006

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

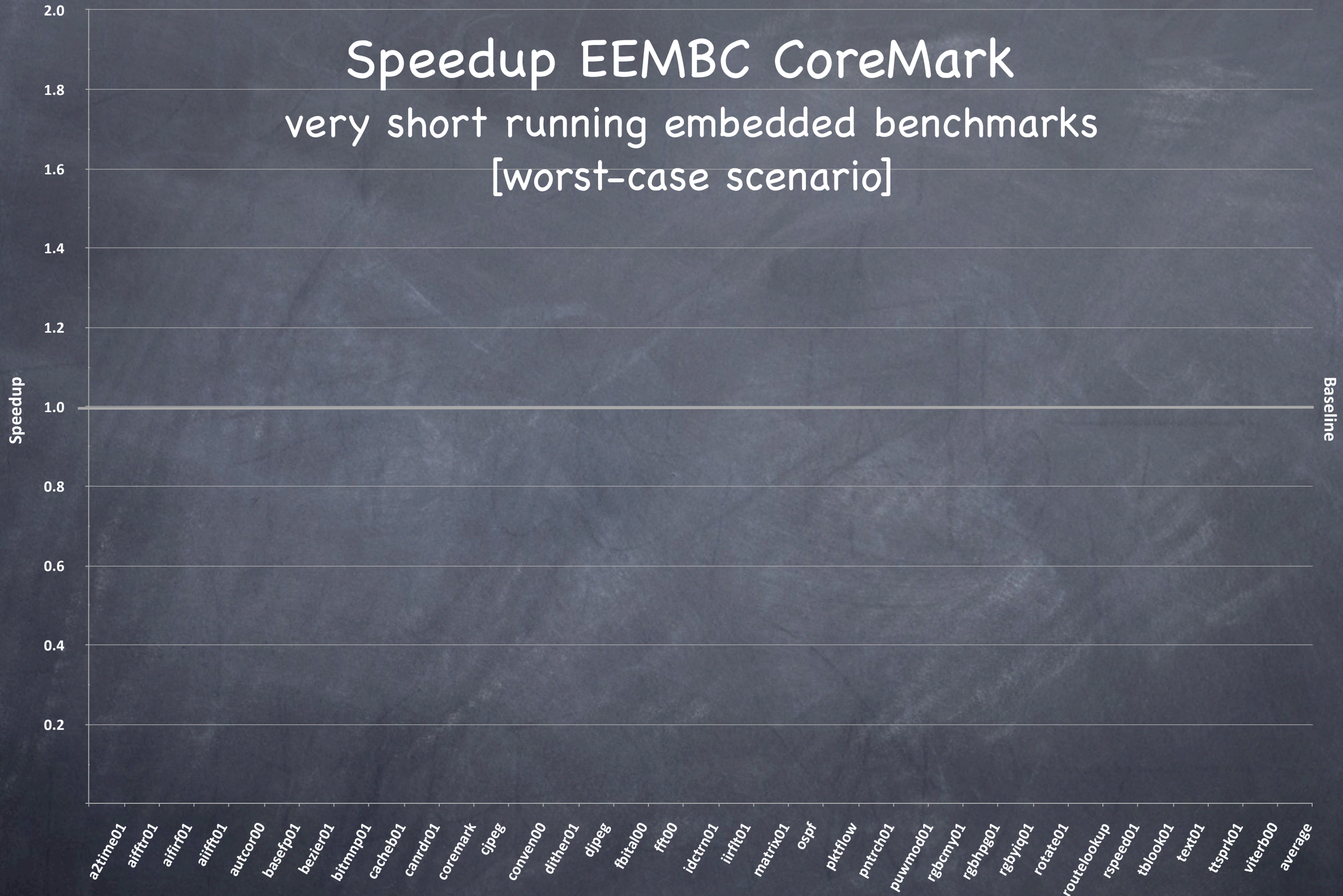


Measured on standard x86 quad-core machine

Speedup EEMBC CoreMark

very short running embedded benchmarks

[worst-case scenario]

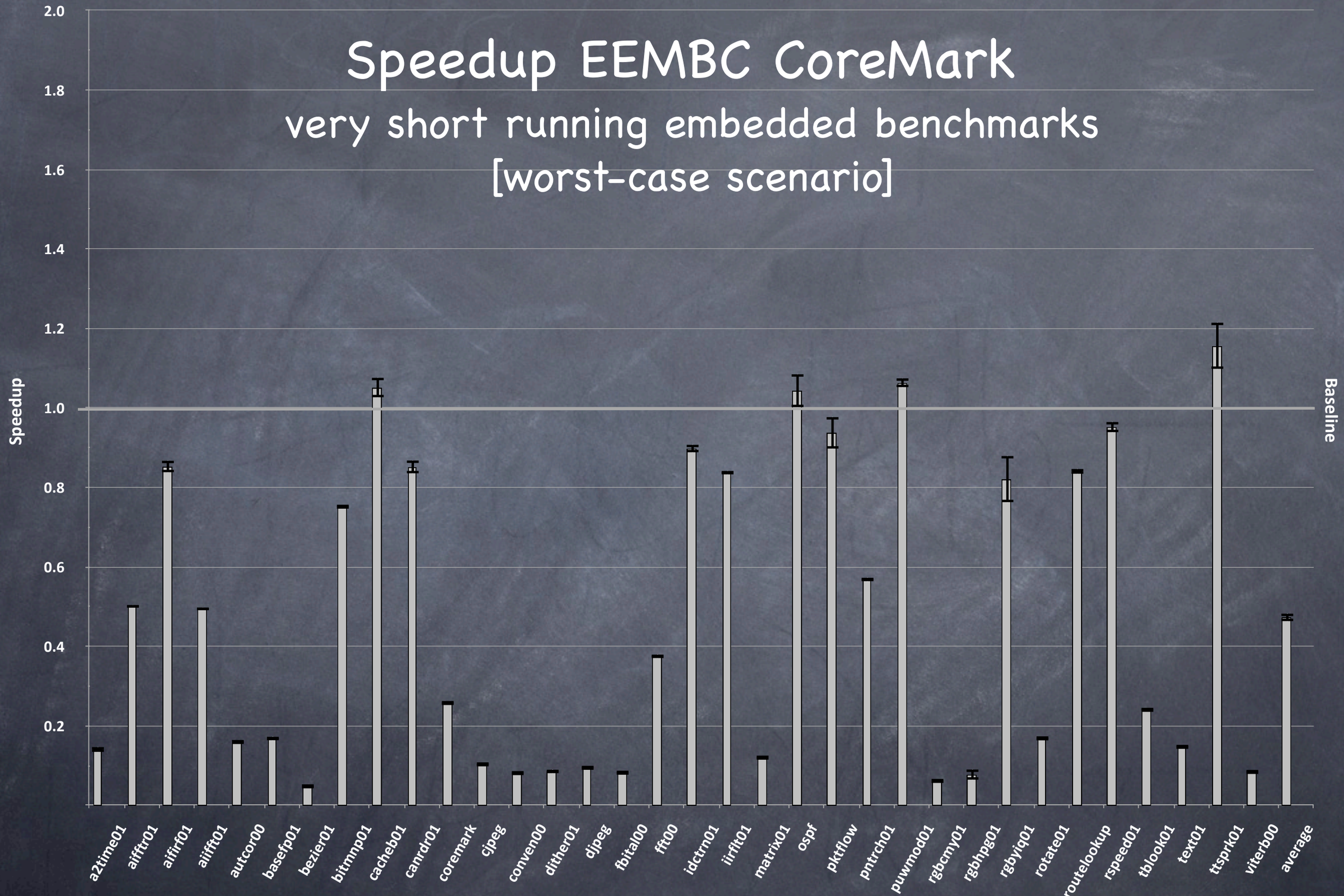


Measured on standard x86 quad-core machine

Speedup EEMBC CoreMark

very short running embedded benchmarks

[worst-case scenario]

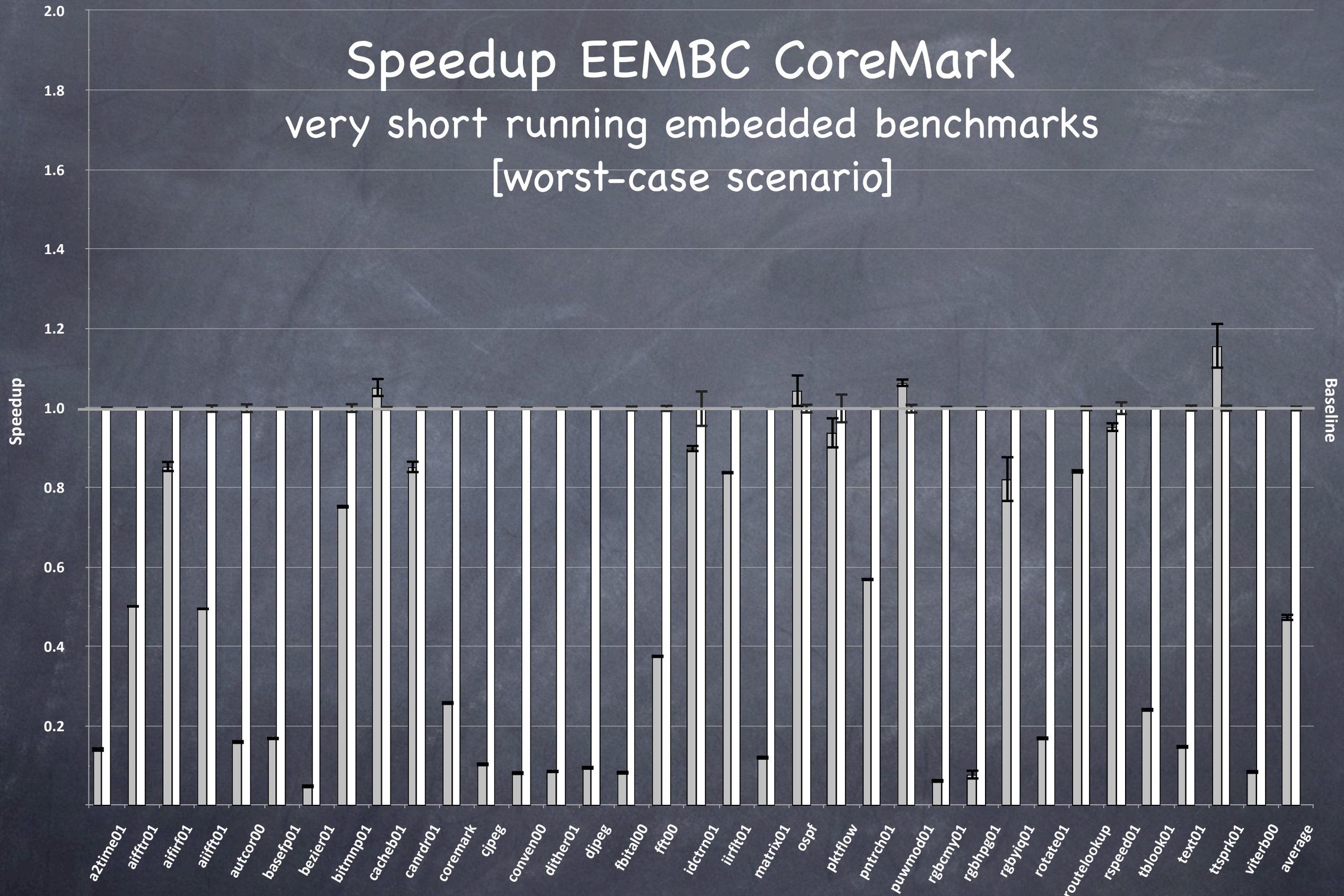


Measured on standard x86 quad-core machine

Speedup EEMBC CoreMark

very short running embedded benchmarks

[worst-case scenario]

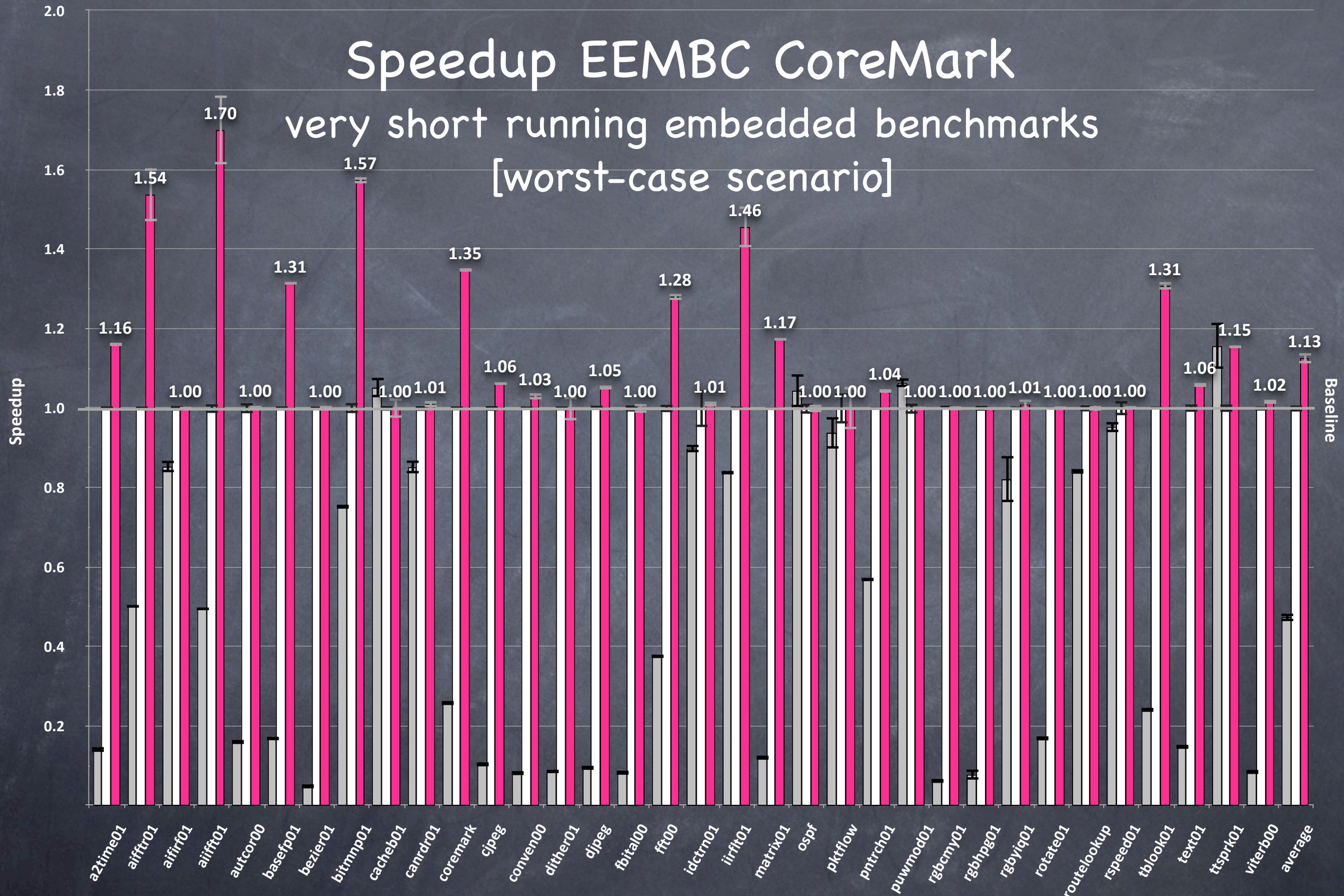


Measured on standard x86 quad-core machine

Speedup EEMBC CoreMark

very short running embedded benchmarks

[worst-case scenario]



Measured on standard x86 quad-core machine

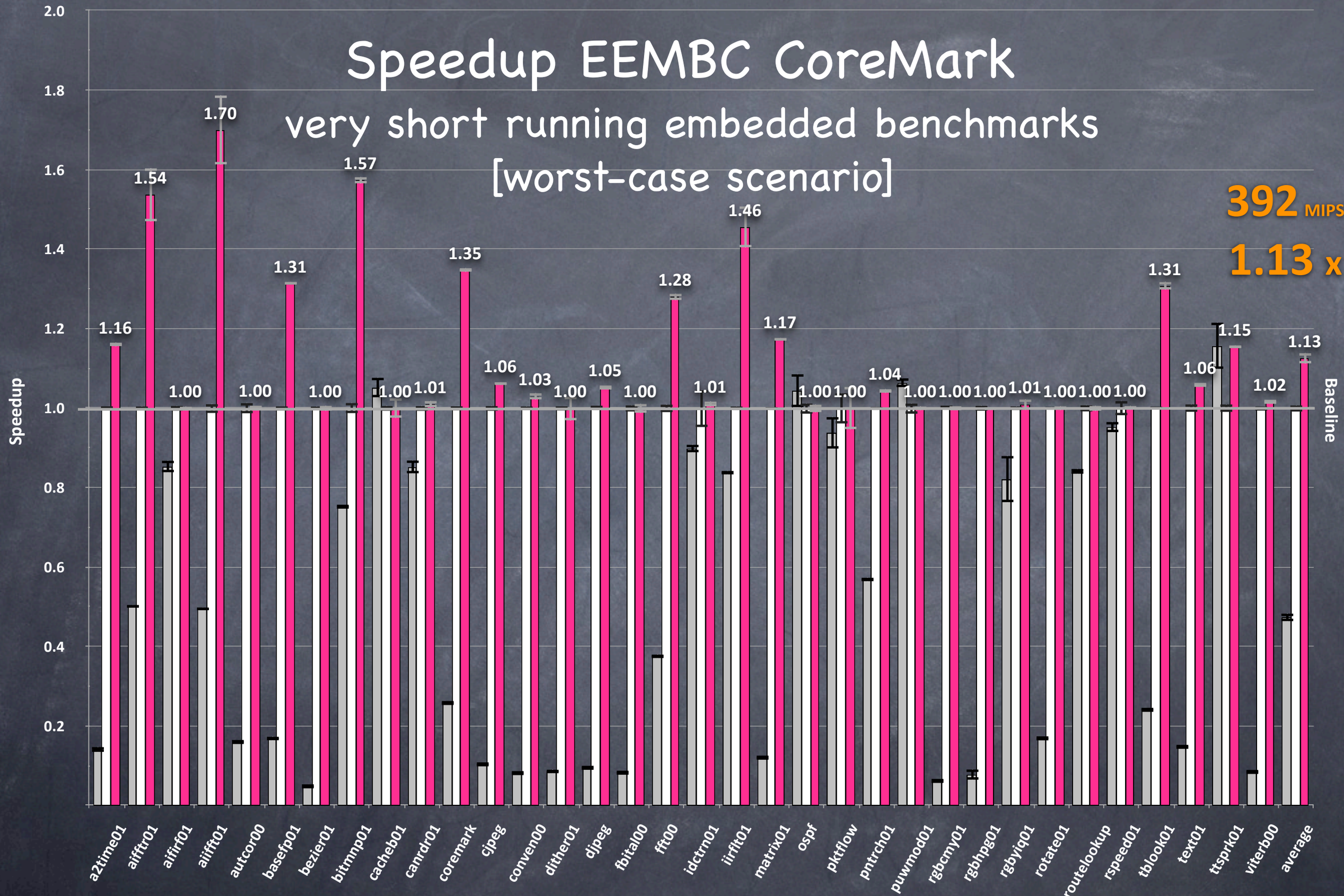
Speedup EEMBC CoreMark

very short running embedded benchmarks

[worst-case scenario]

392 MIPS

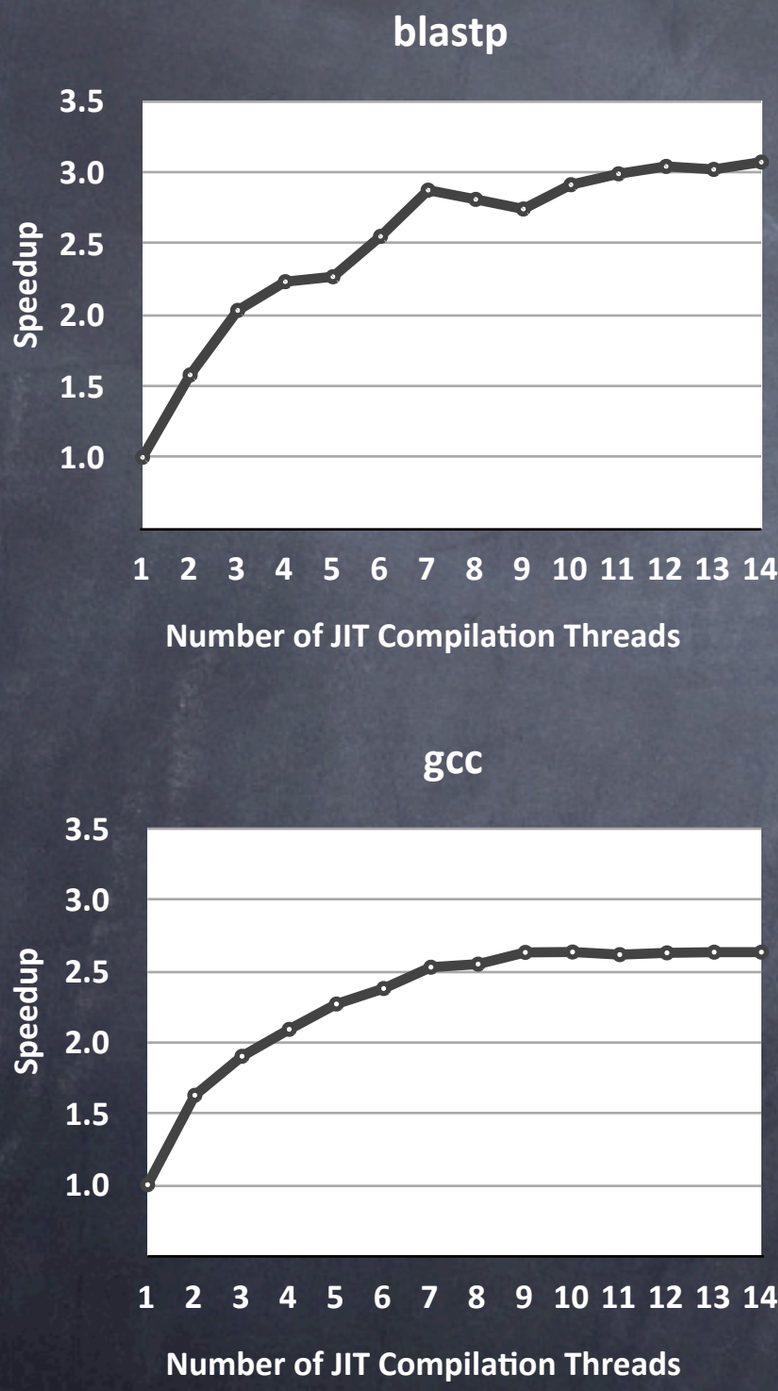
1.13 x

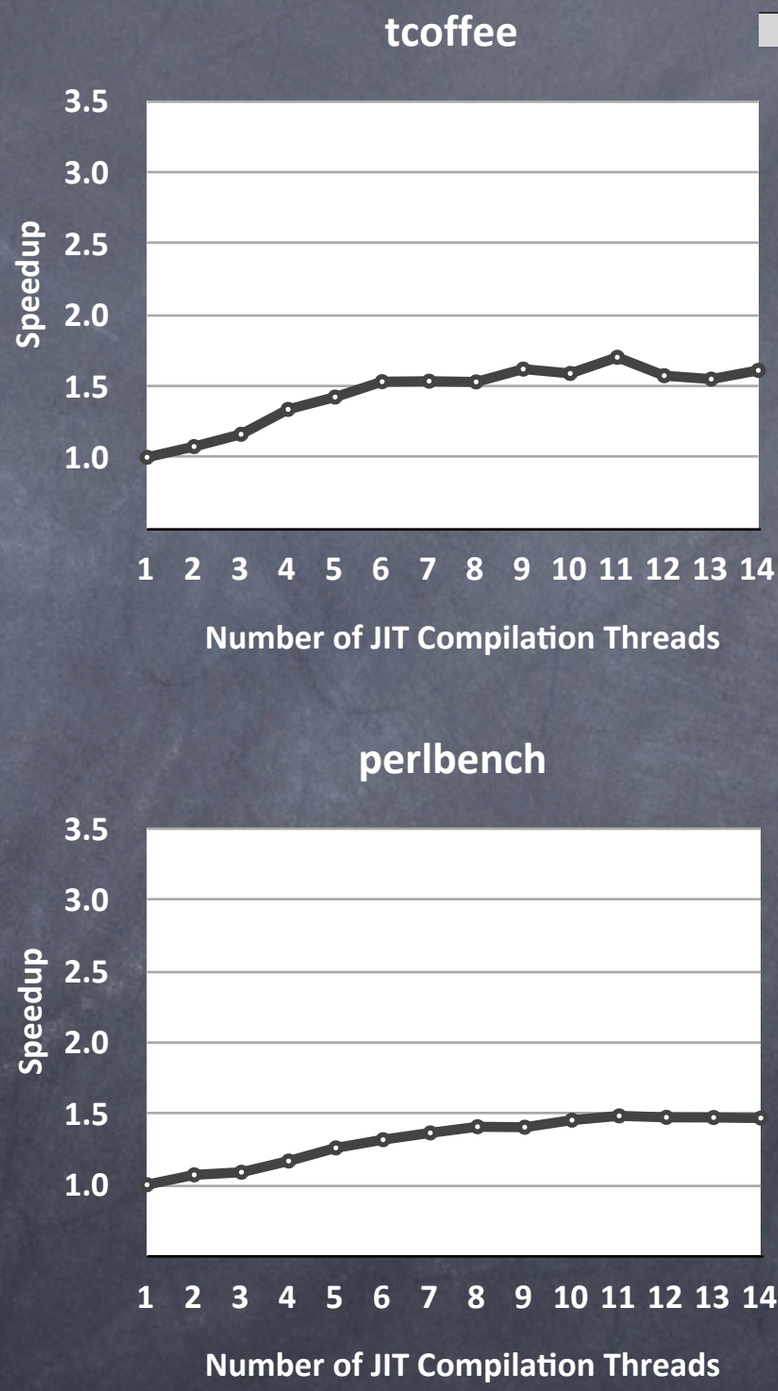


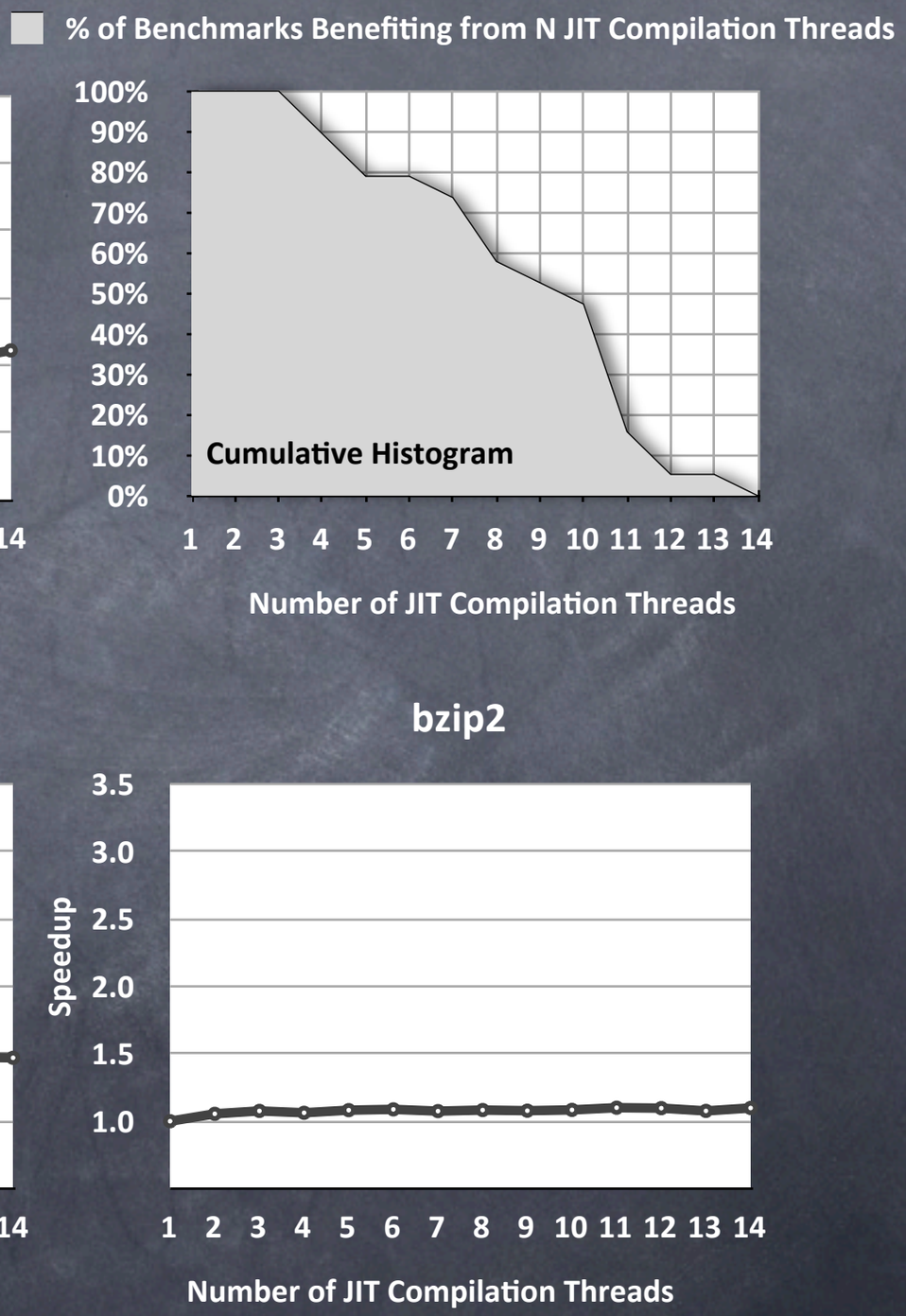
Measured on standard x86 quad-core machine

How far does it scale?

What is a sensible number of JIT compilation threads?



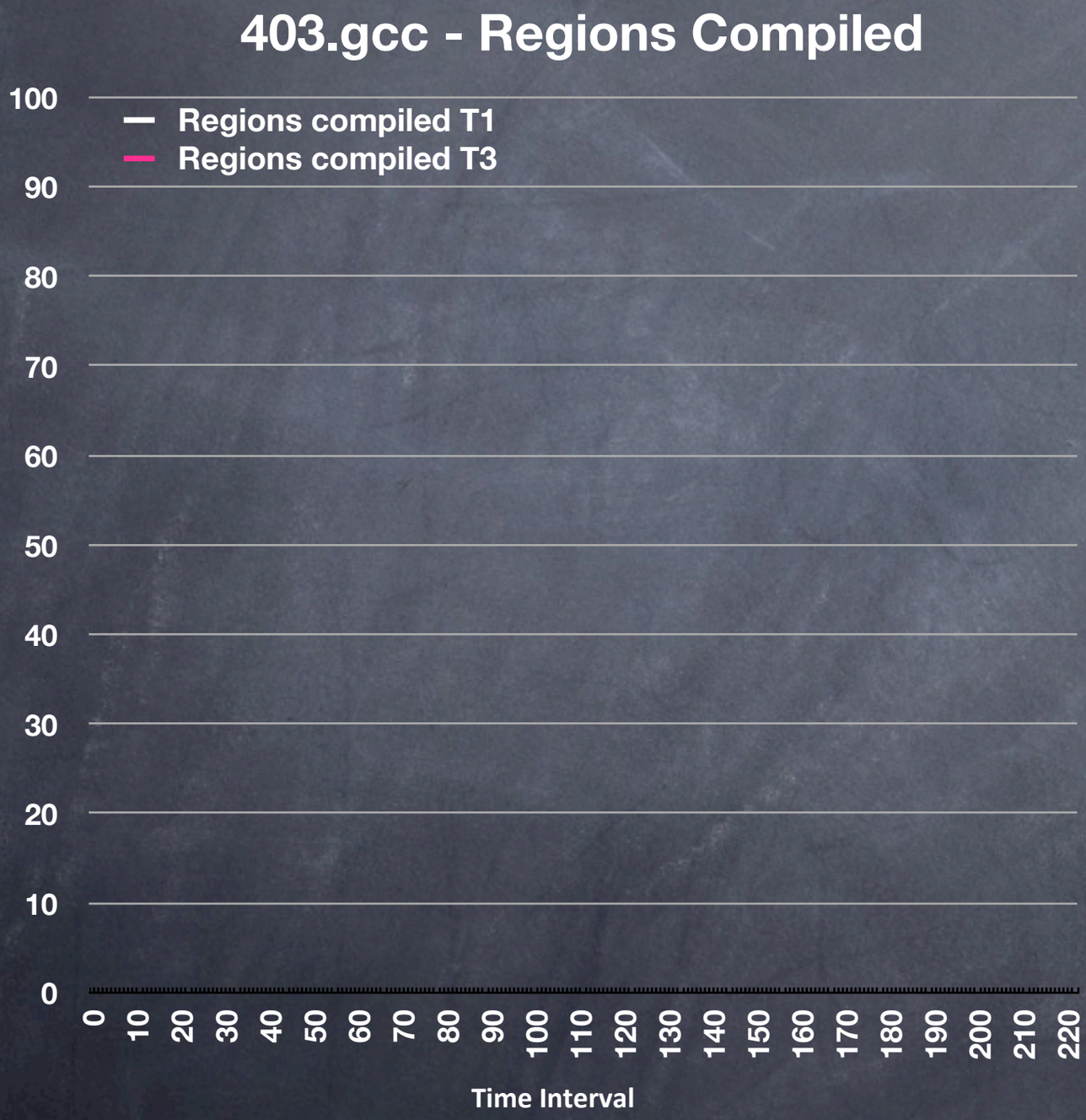




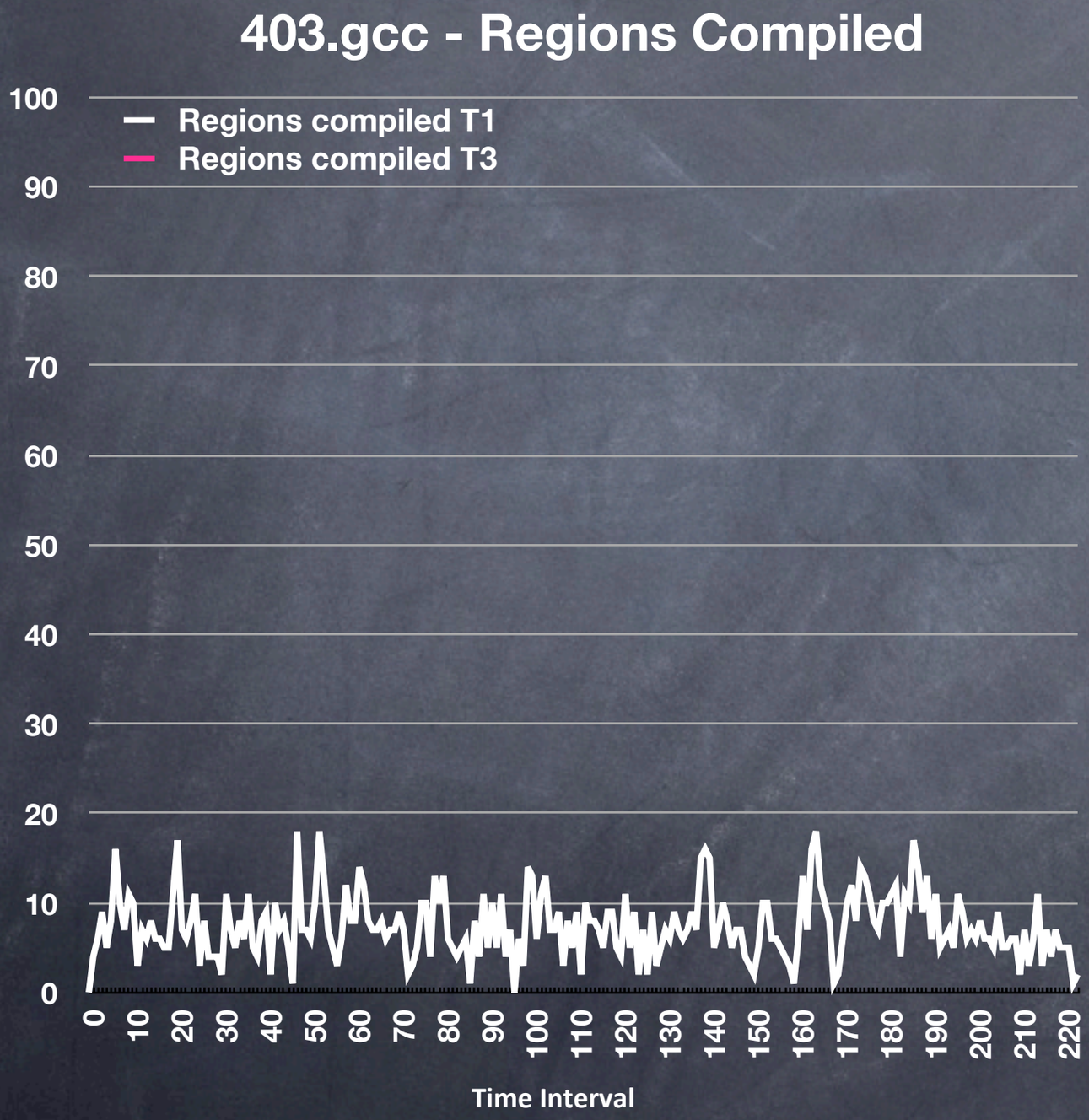
Measured on a 16-core machine

Effect of Concurrent and Parallel JIT Compilation on Throughput

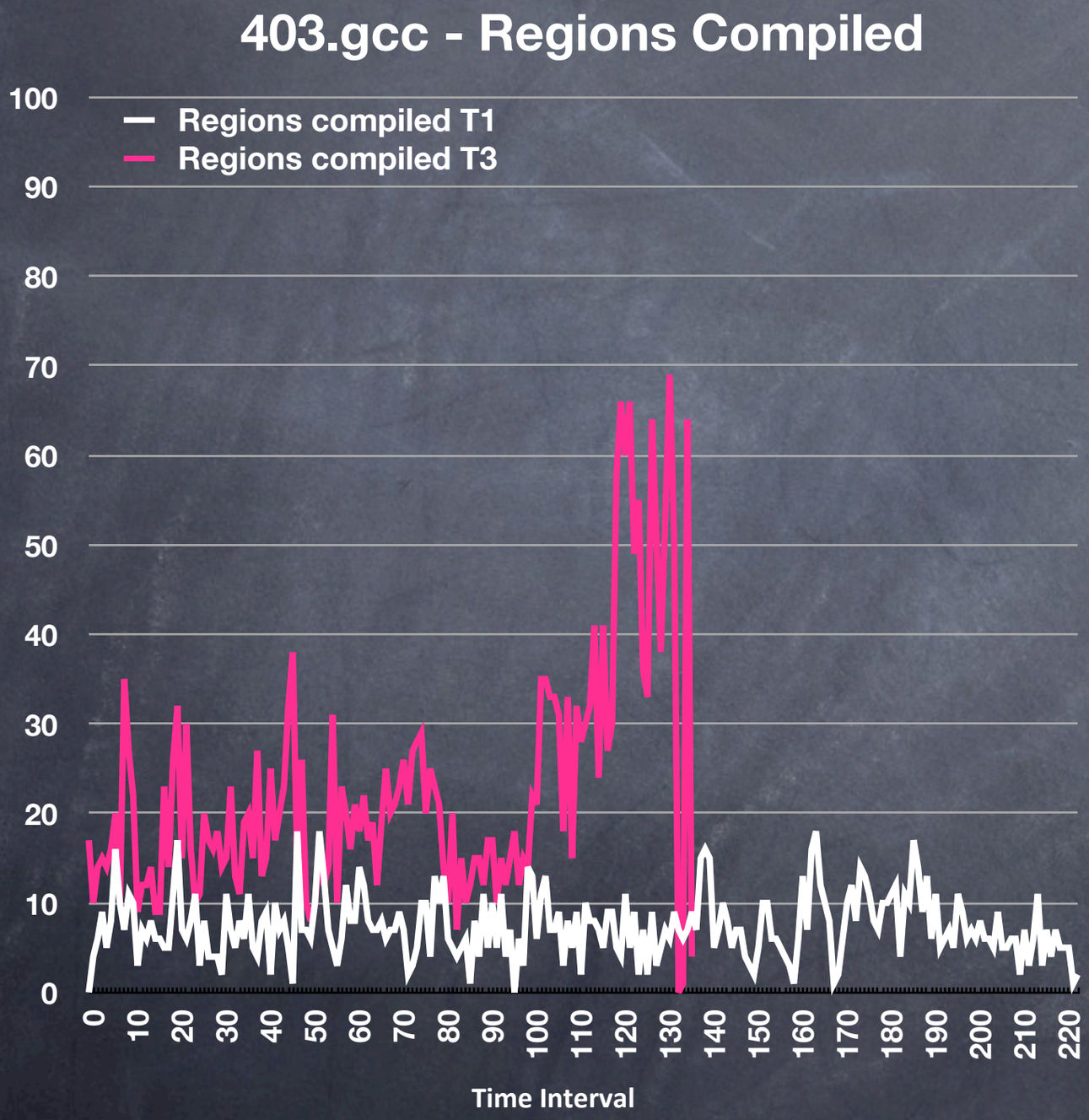
Effect of Concurrent and Parallel JIT Compilation on Throughput



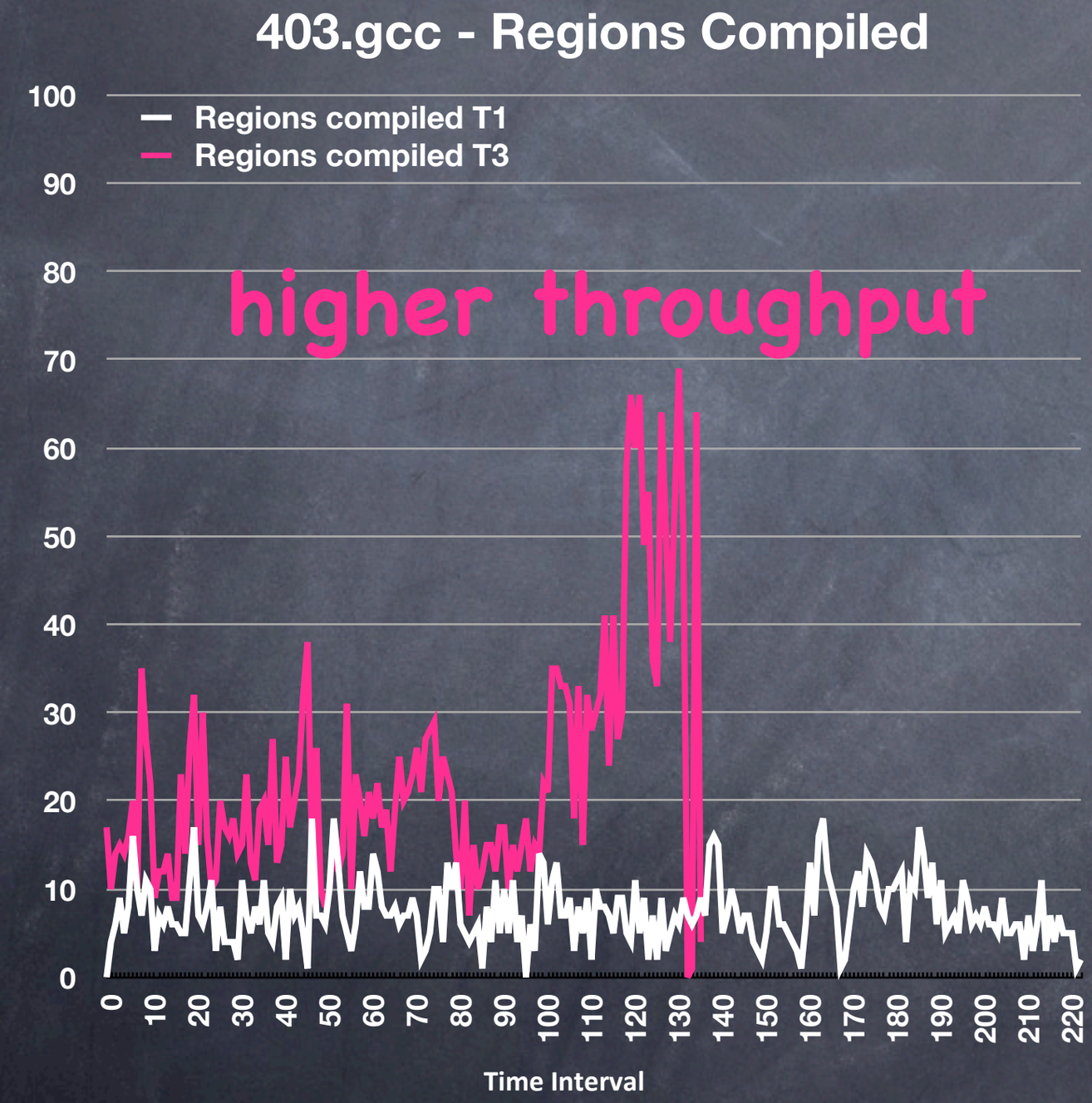
Effect of Concurrent and Parallel JIT Compilation on Throughput



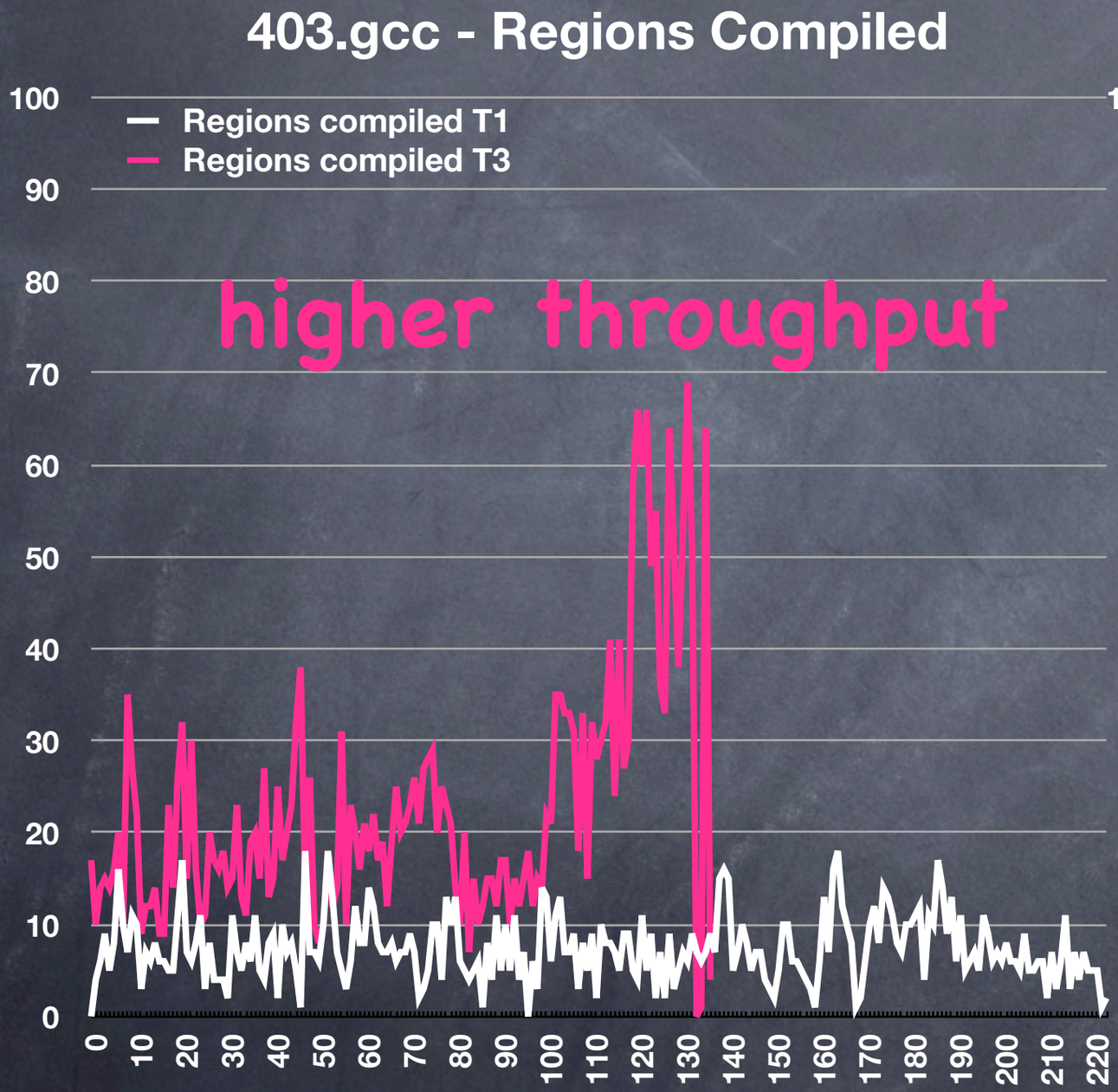
Effect of Concurrent and Parallel JIT Compilation on Throughput

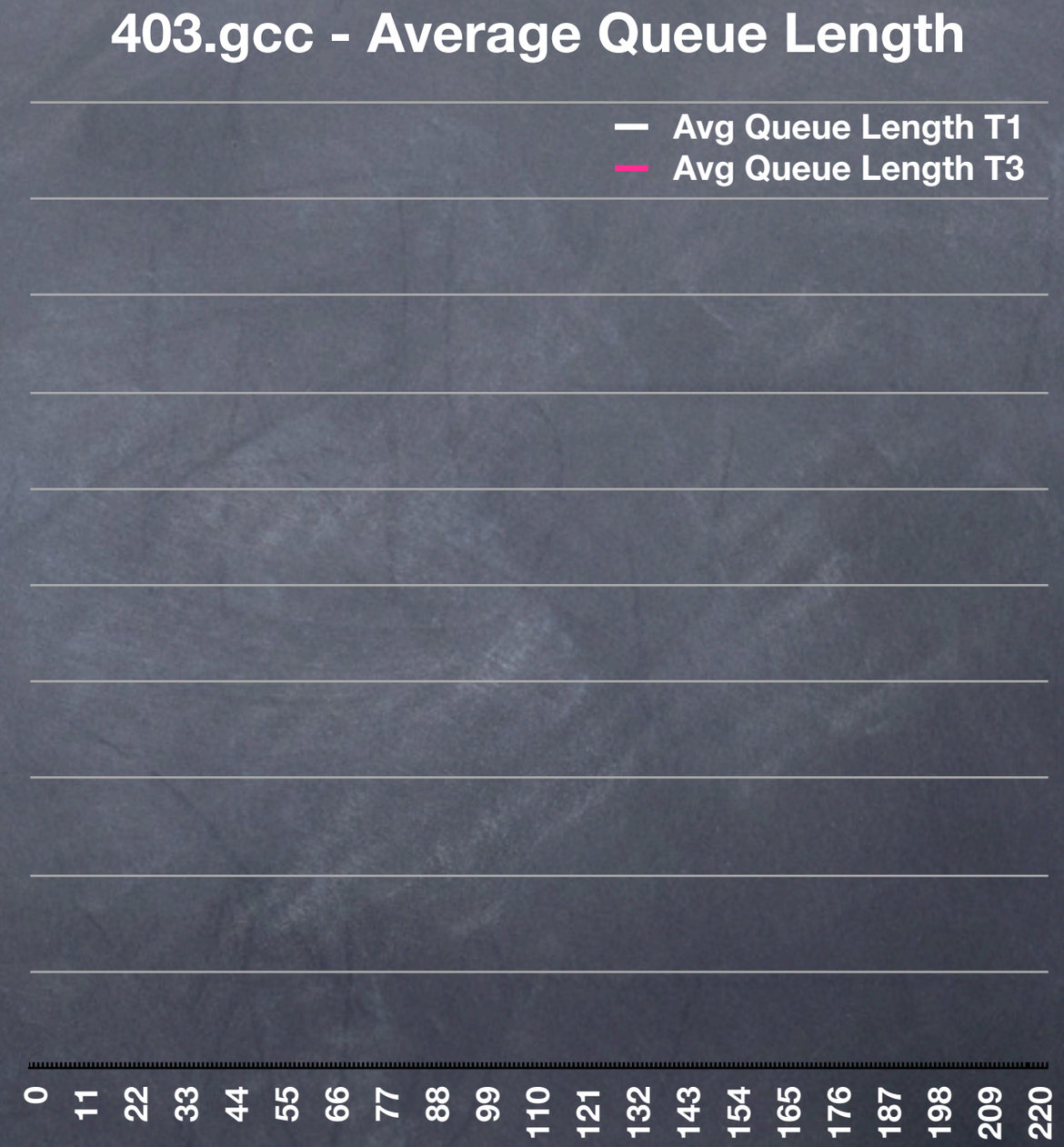


Effect of Concurrent and Parallel JIT Compilation on Throughput

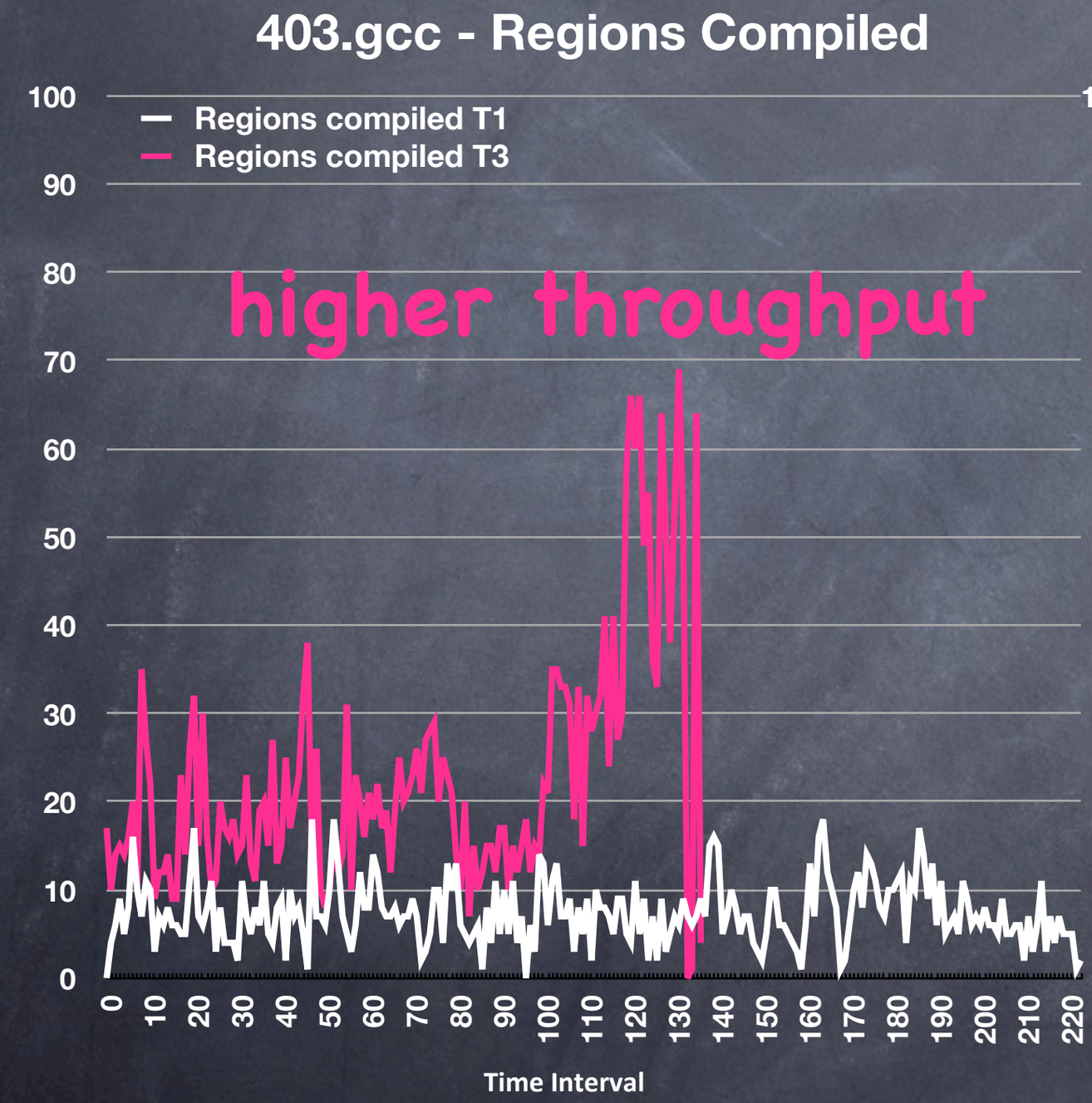


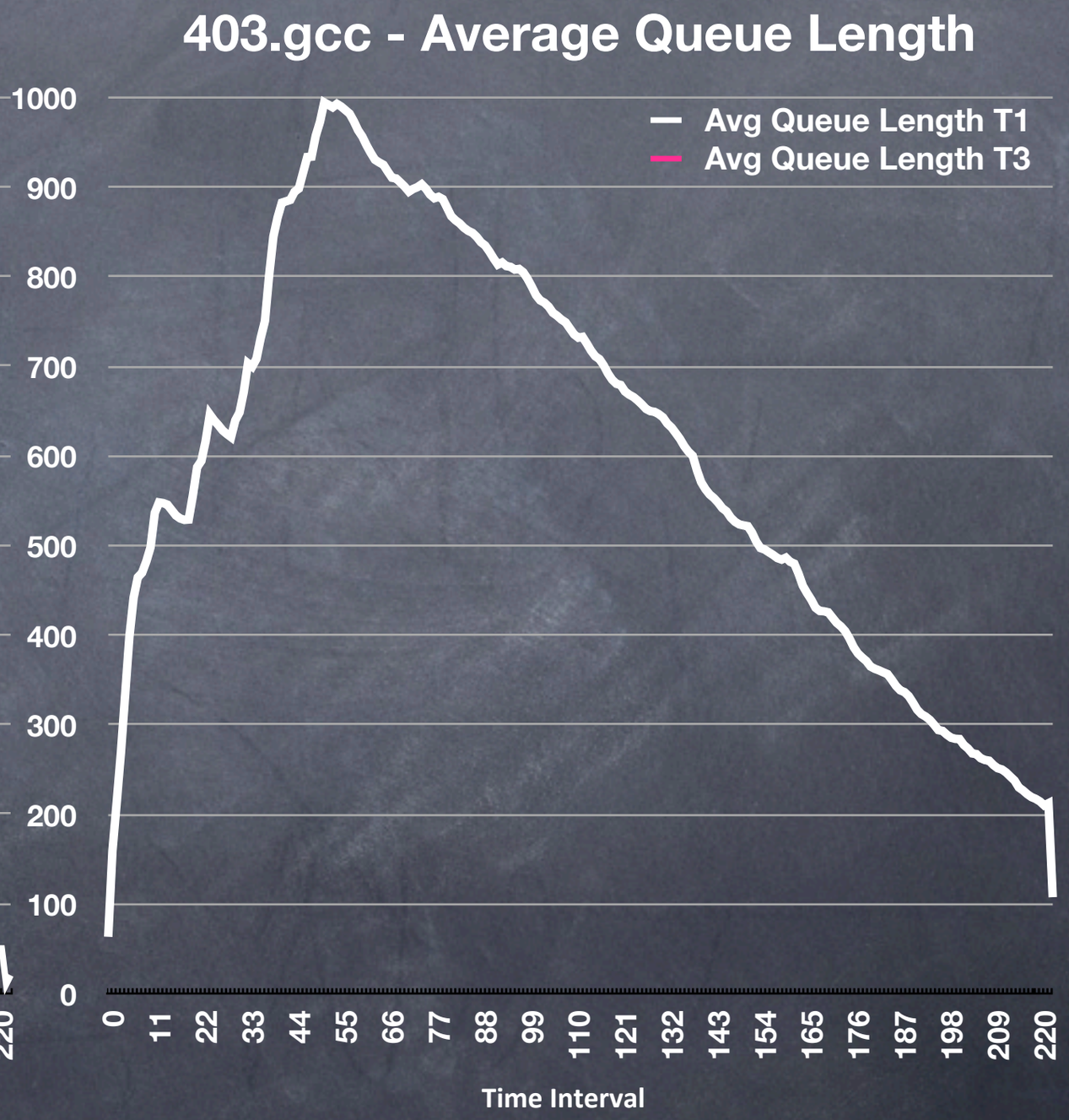
Effect of Concurrent and Parallel JIT Compilation on Throughput



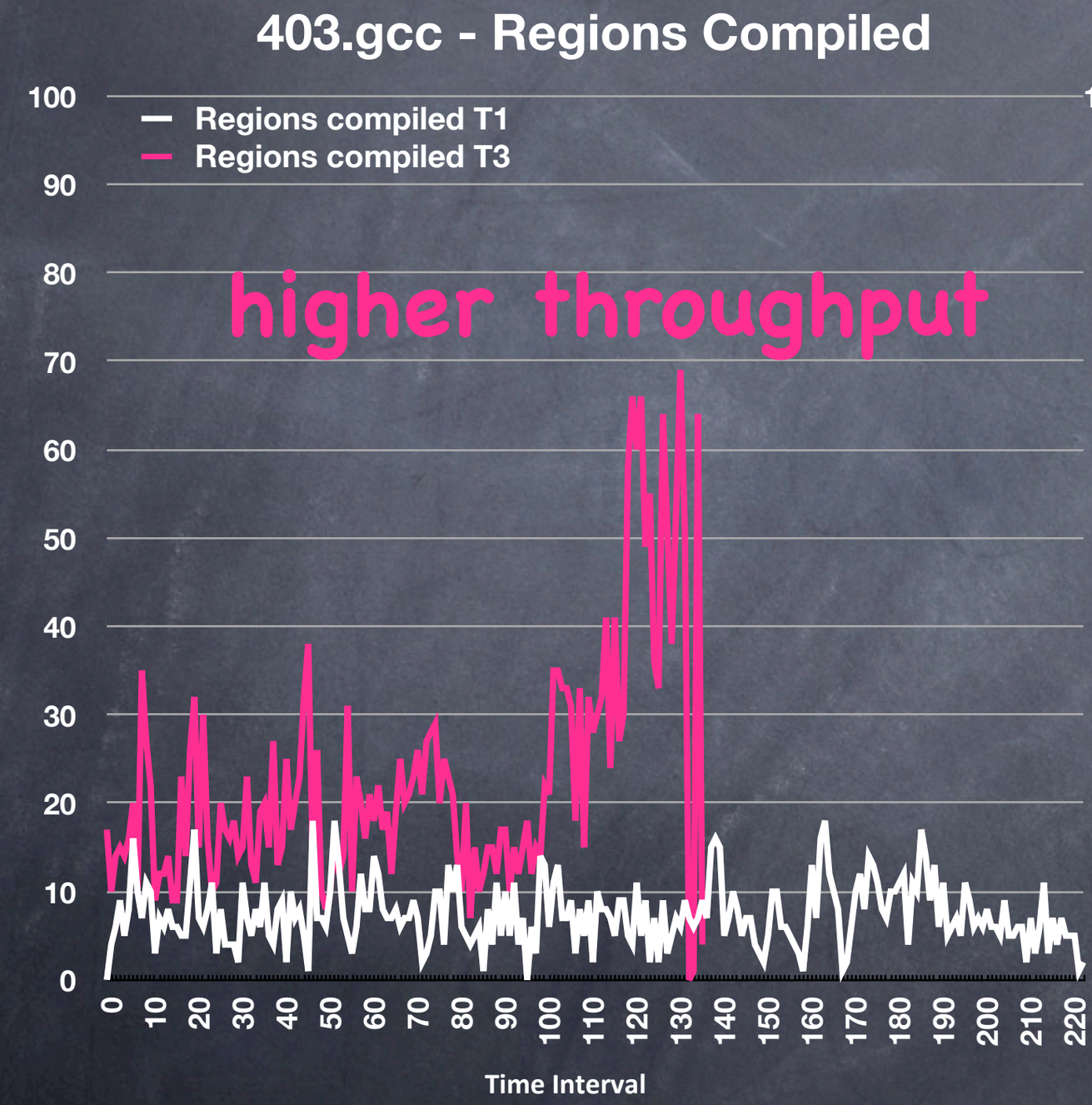


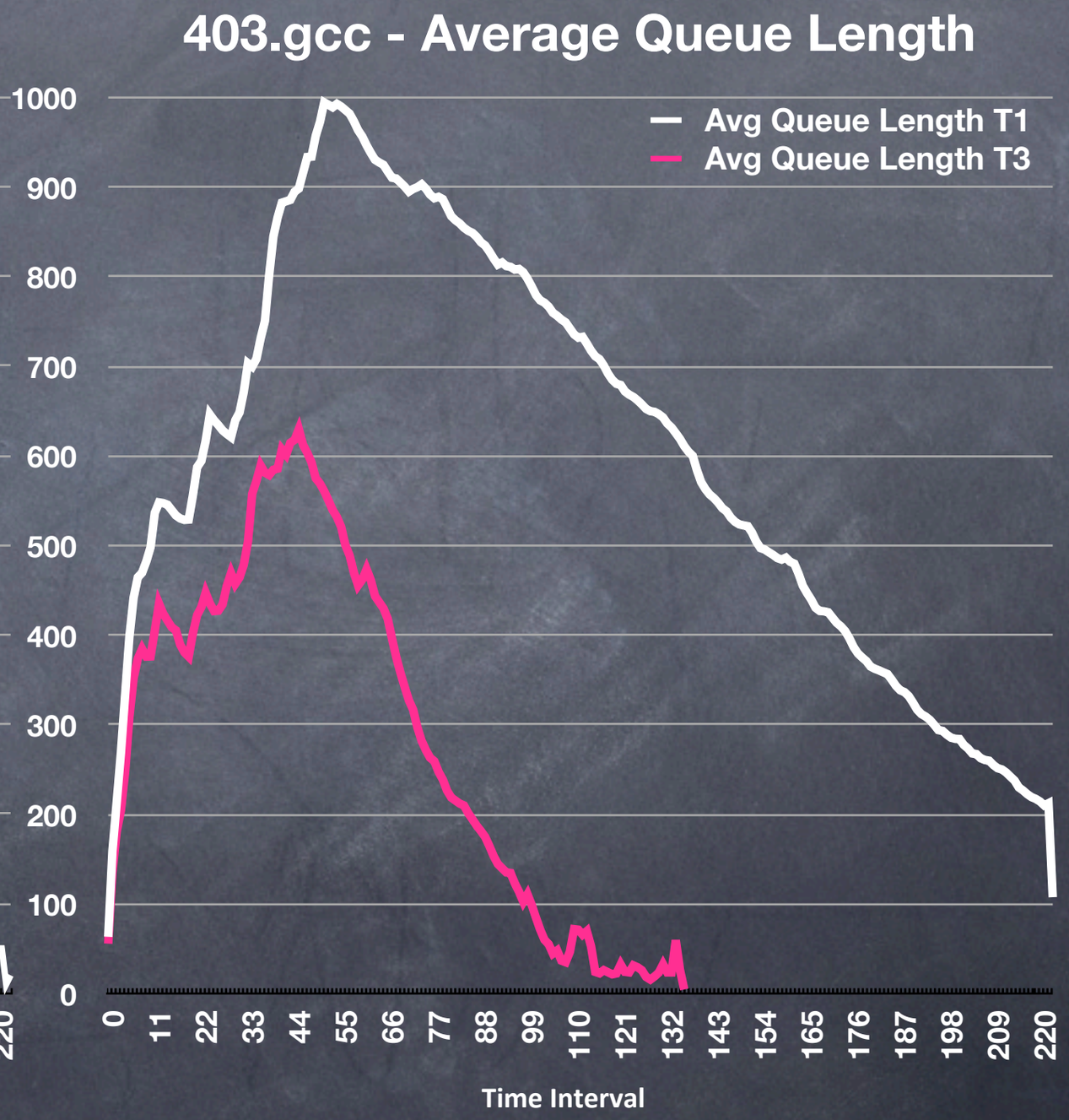
Effect of Concurrent and Parallel JIT Compilation on Throughput



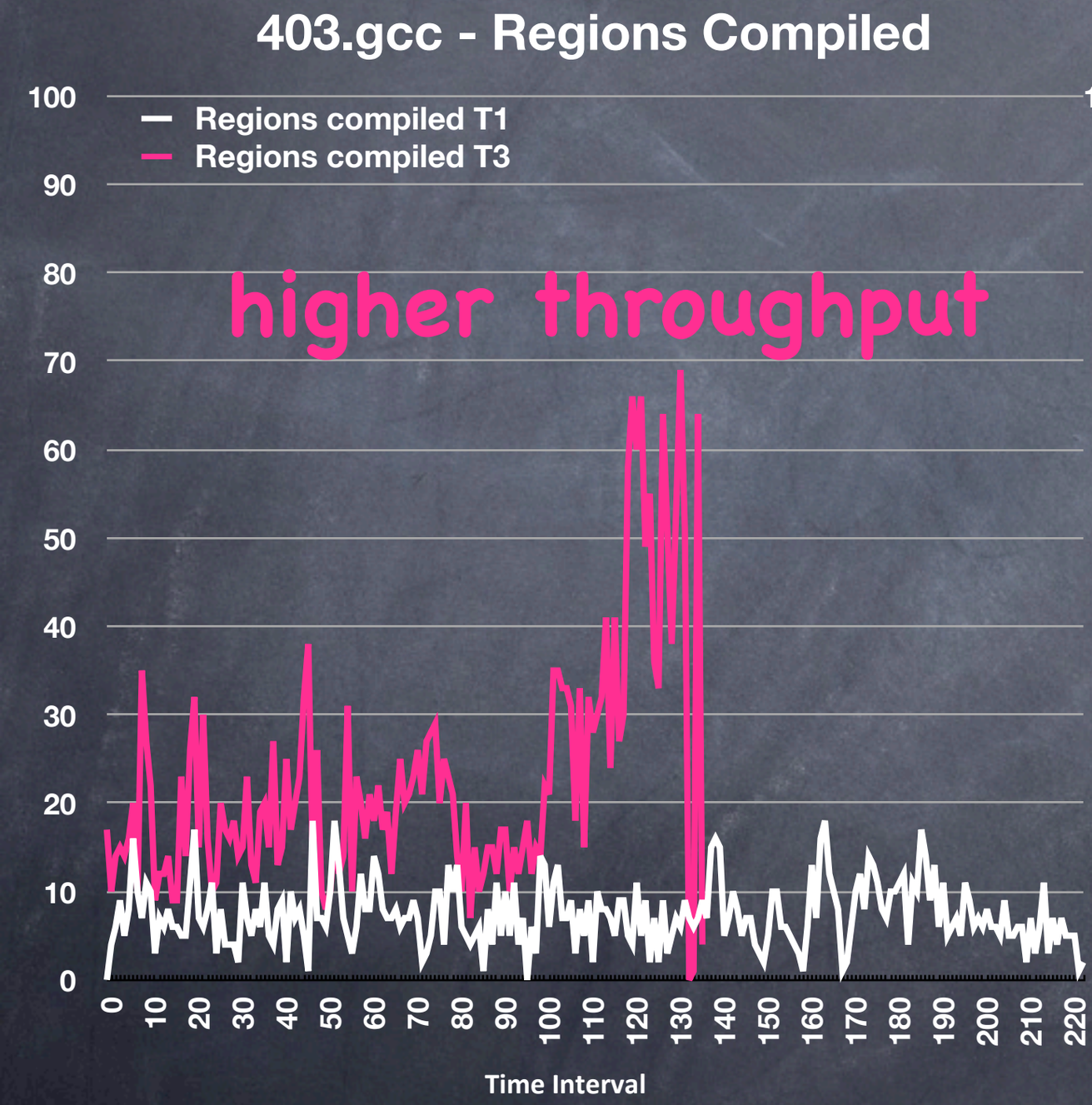


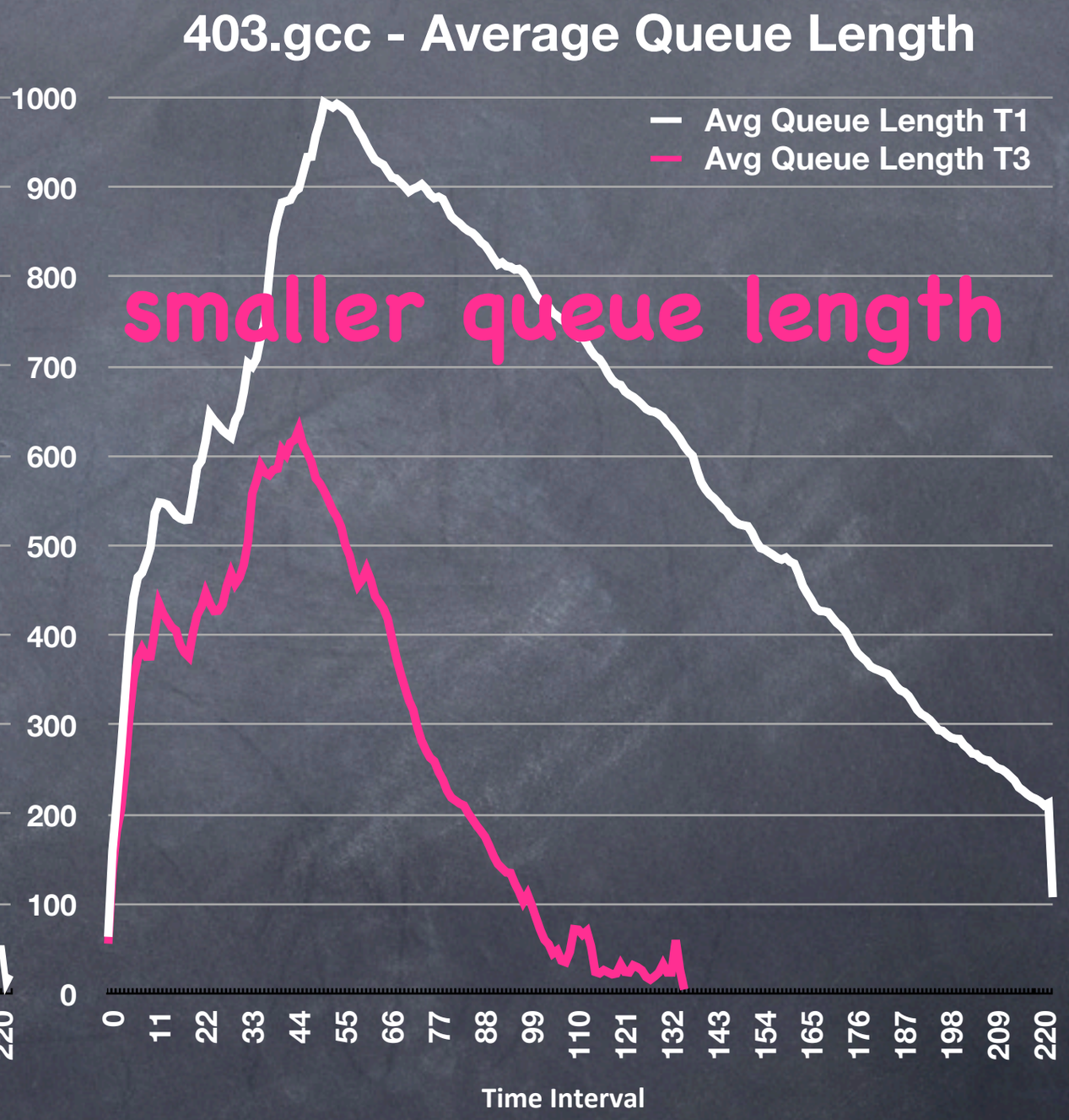
Effect of Concurrent and Parallel JIT Compilation on Throughput





Effect of Concurrent and Parallel JIT Compilation on Throughput

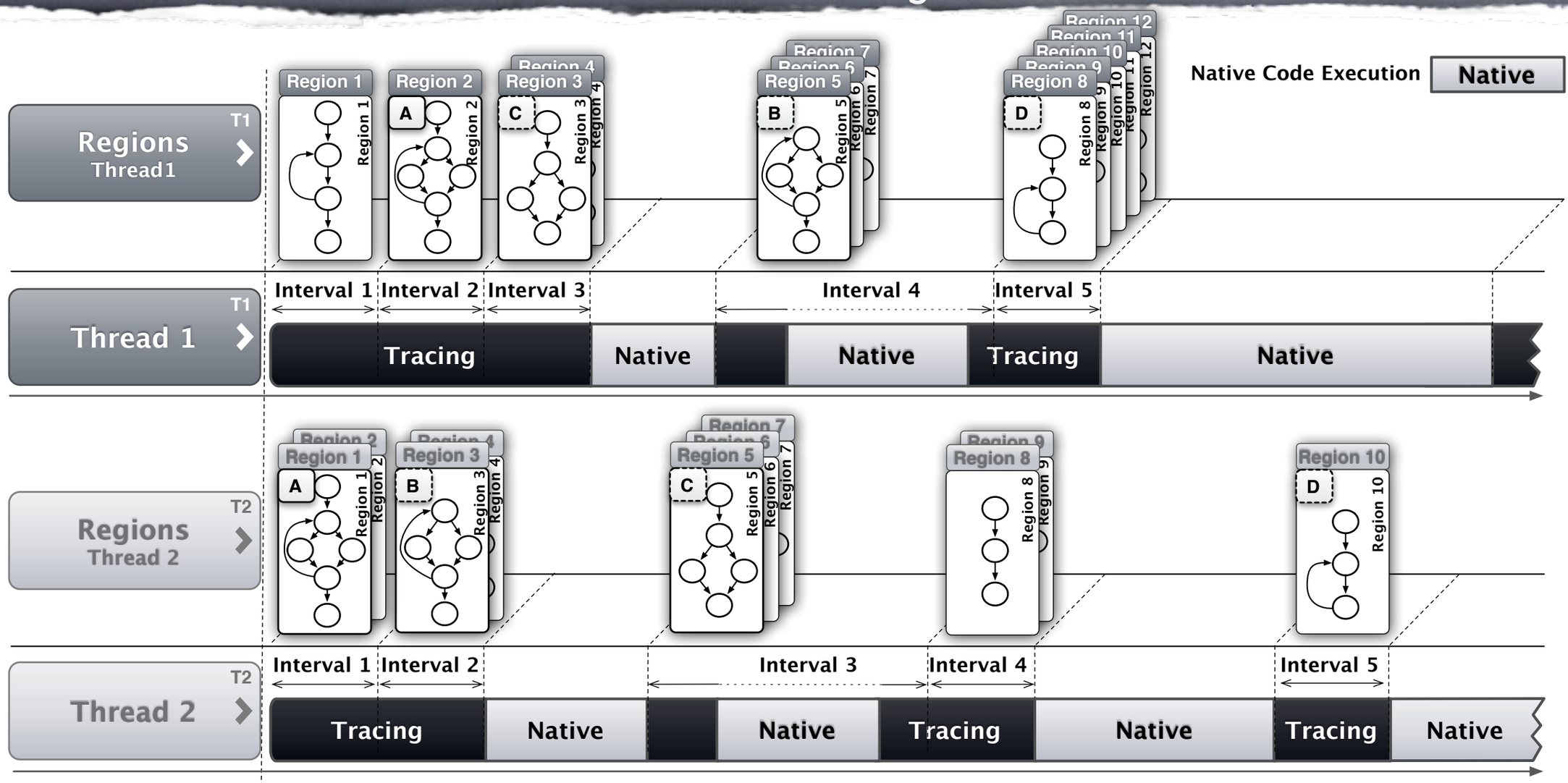




Does this scale for multi-threaded/core applications?

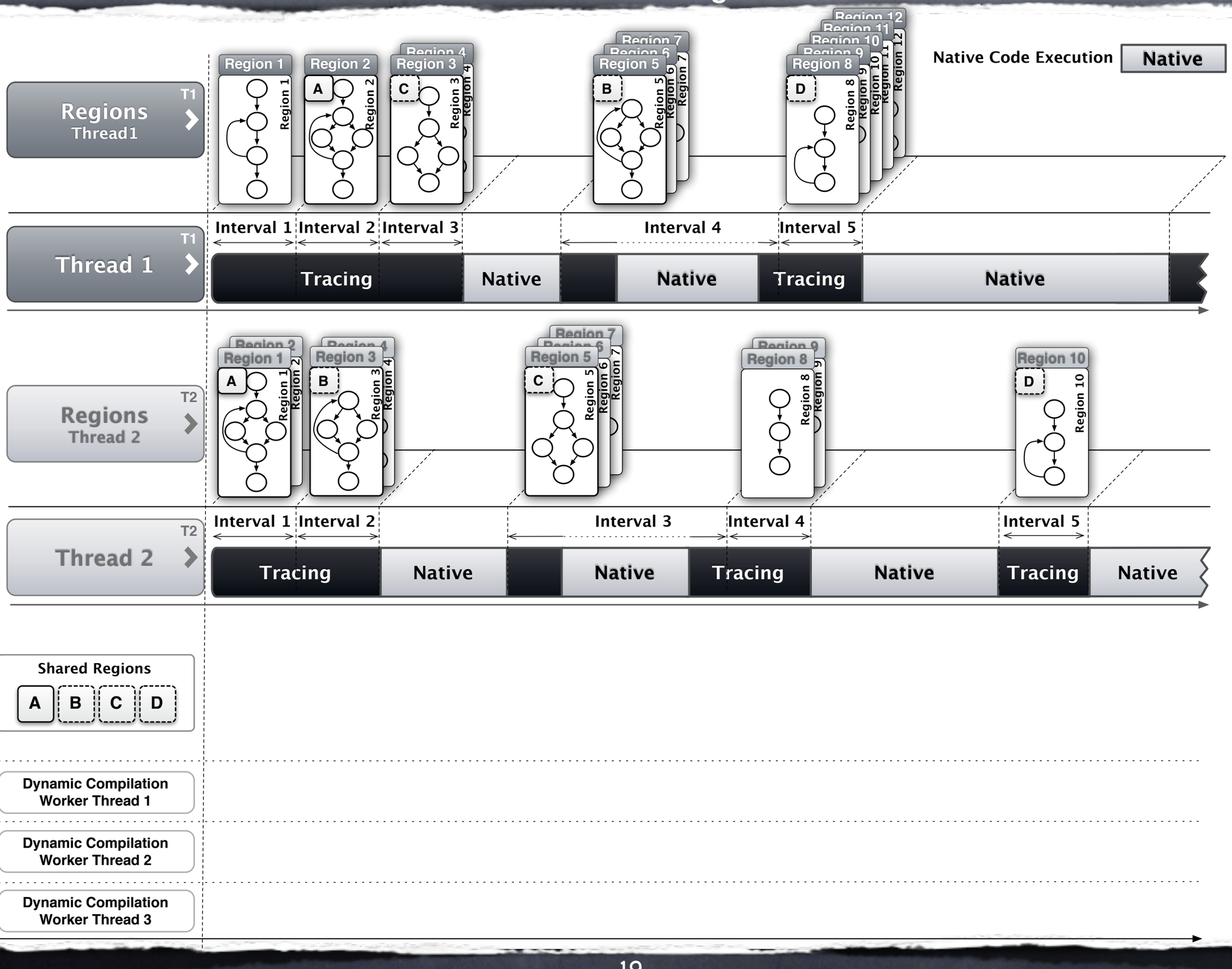
Concurrent and Parallel JIT Compilation in Action

(trace sharing)



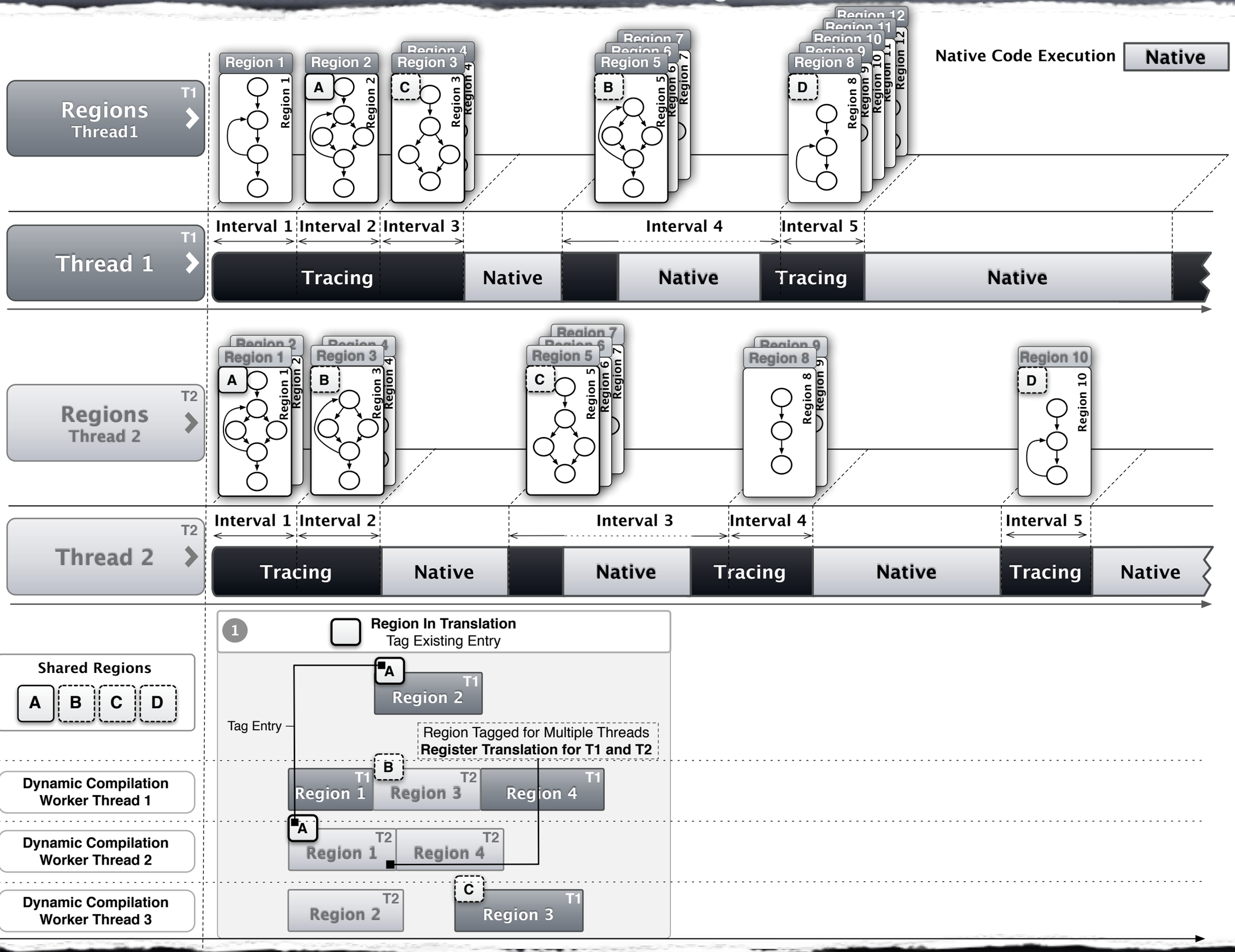
Concurrent and Parallel JIT Compilation in Action

(trace sharing)



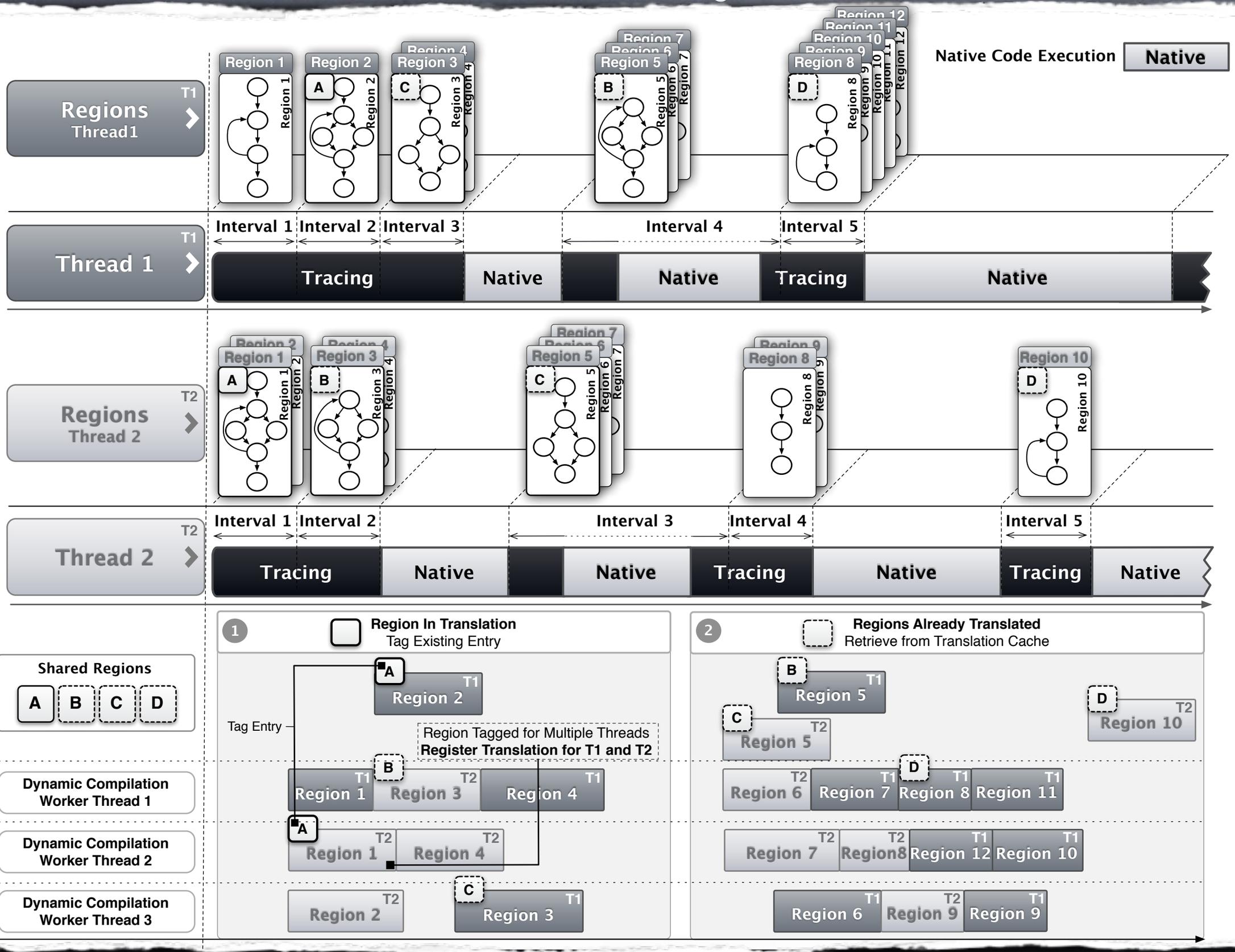
Concurrent and Parallel JIT Compilation in Action

(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

Demos

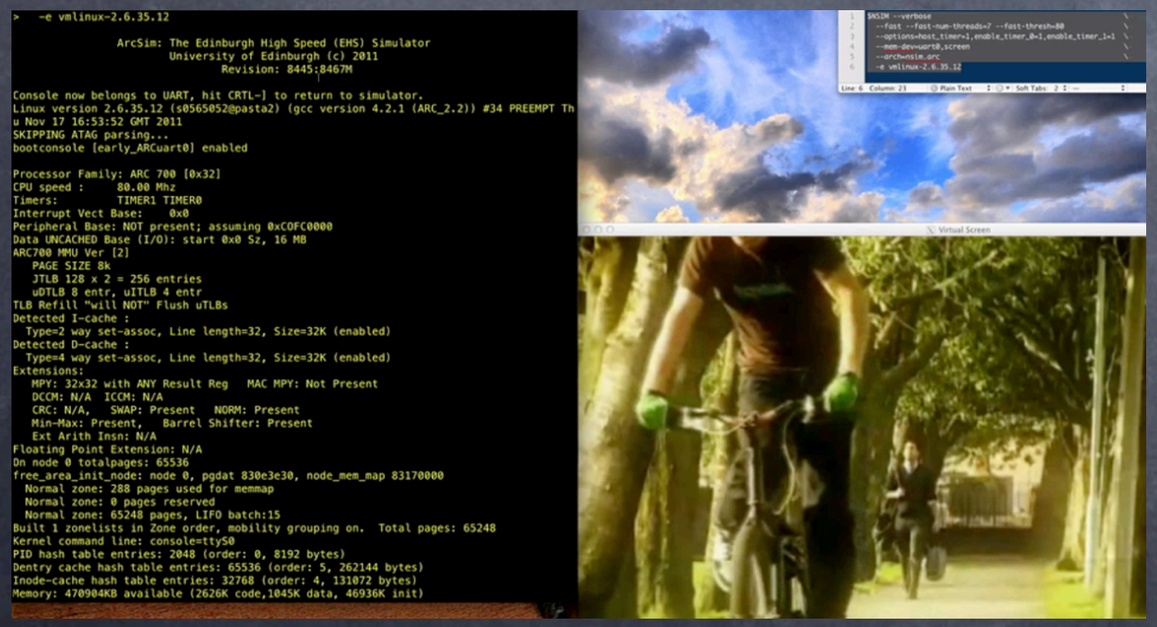
web▸m

Video Decoding
and Playback

Demos

web▸m

Video Decoding
and Playback



Full System OS Simulation

Thank You

Backup Slide

State-of-the-Art Dynamic Compilation Strategies

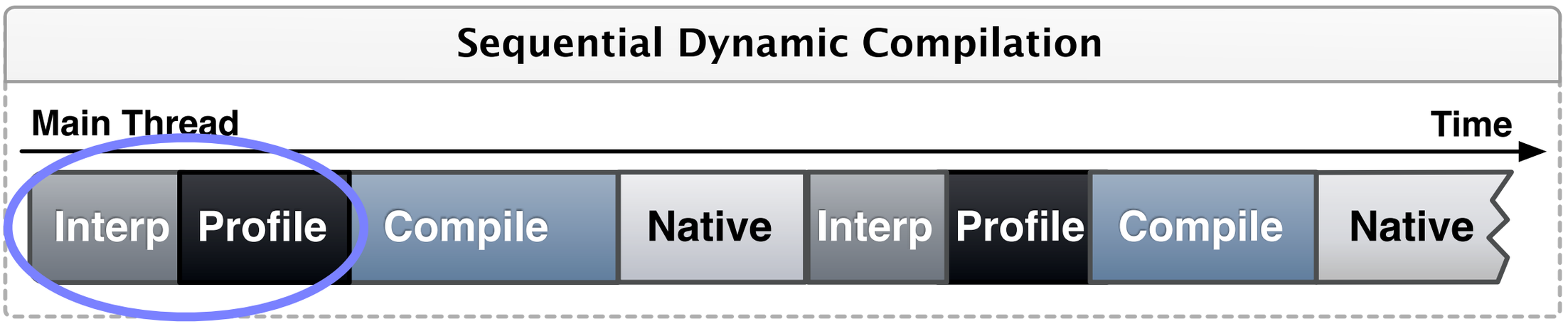
Interp	Interpretation	Compile	Dynamic Compilation
Profile	Interpretation with Profiling	Native	Native Code Execution

Sequential Dynamic Compilation

Backup Slide

State-of-the-Art Dynamic Compilation Strategies

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



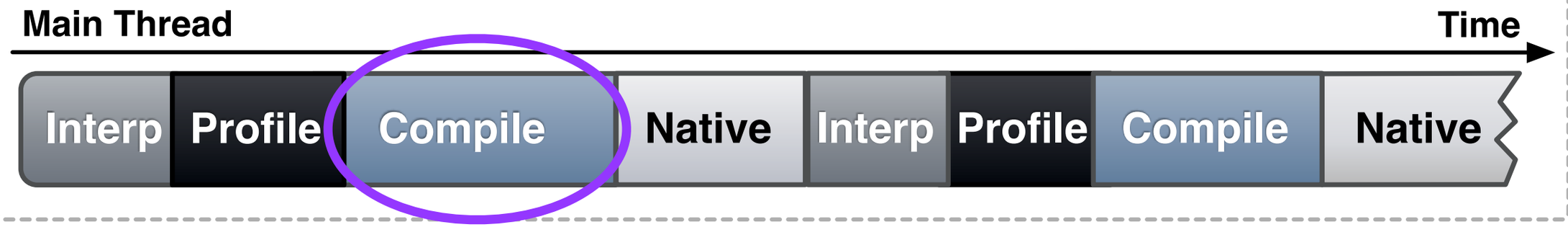
Code Discovery

Backup Slide

State-of-the-Art Dynamic Compilation Strategies

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

Sequential Dynamic Compilation



Dynamic Compilation

Backup Slide

State-of-the-Art Dynamic Compilation Strategies

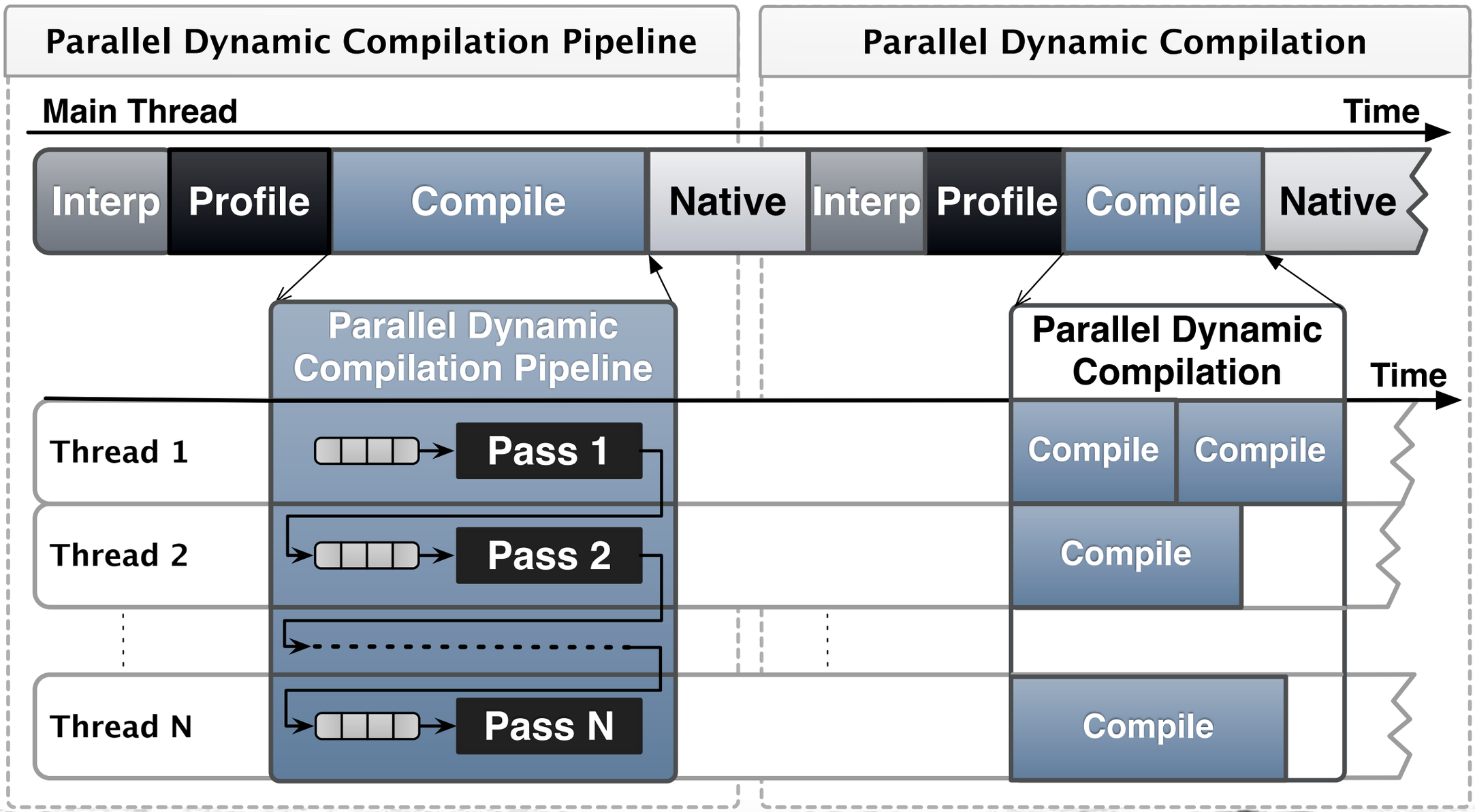
Interp	Interpretation	Compile	Dynamic Compilation
Profile	Interpretation with Profiling	Native	Native Code Execution

Sequential Dynamic Compilation

critical path

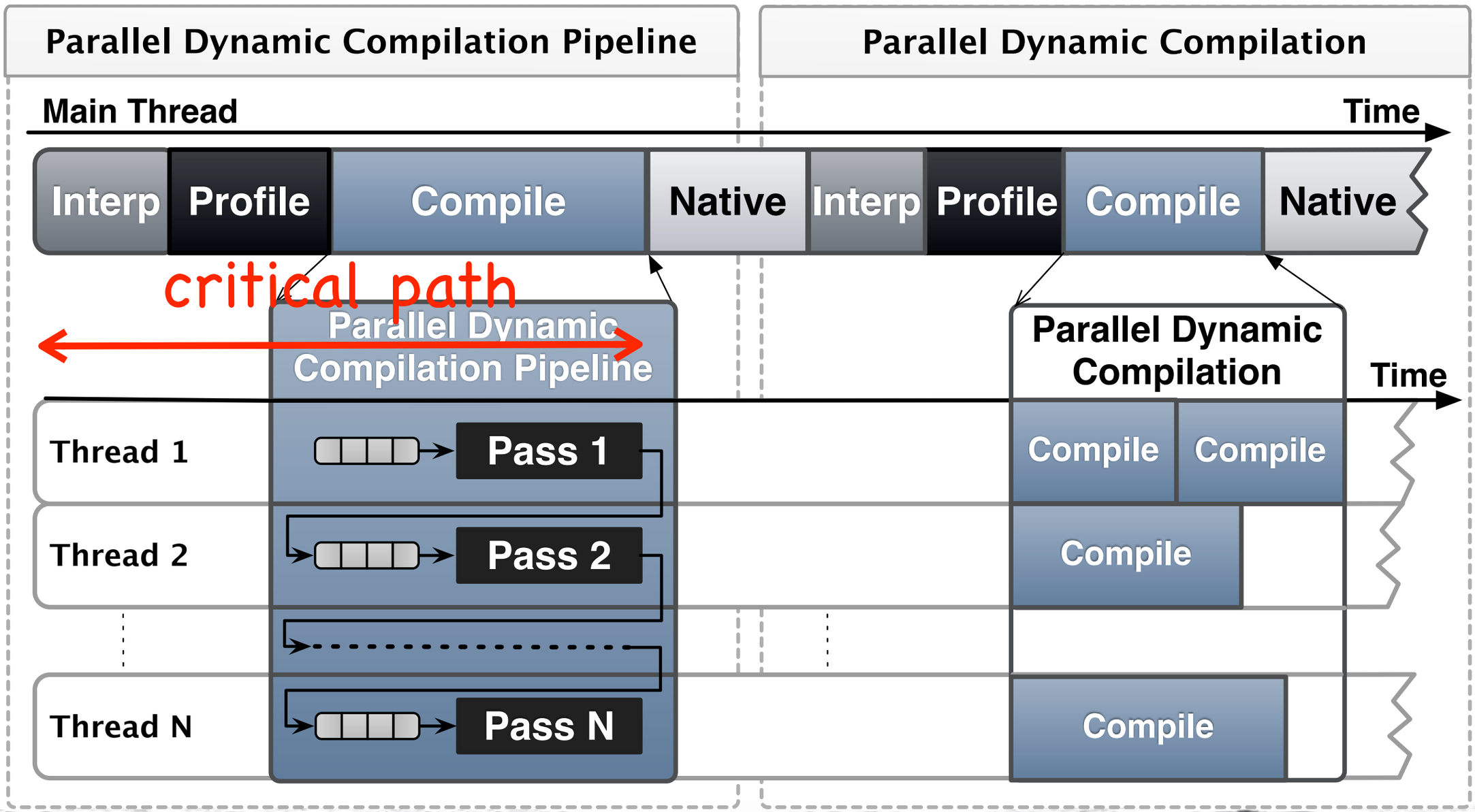
Backup Slide

State-of-the-Art Dynamic Compilation Strategies



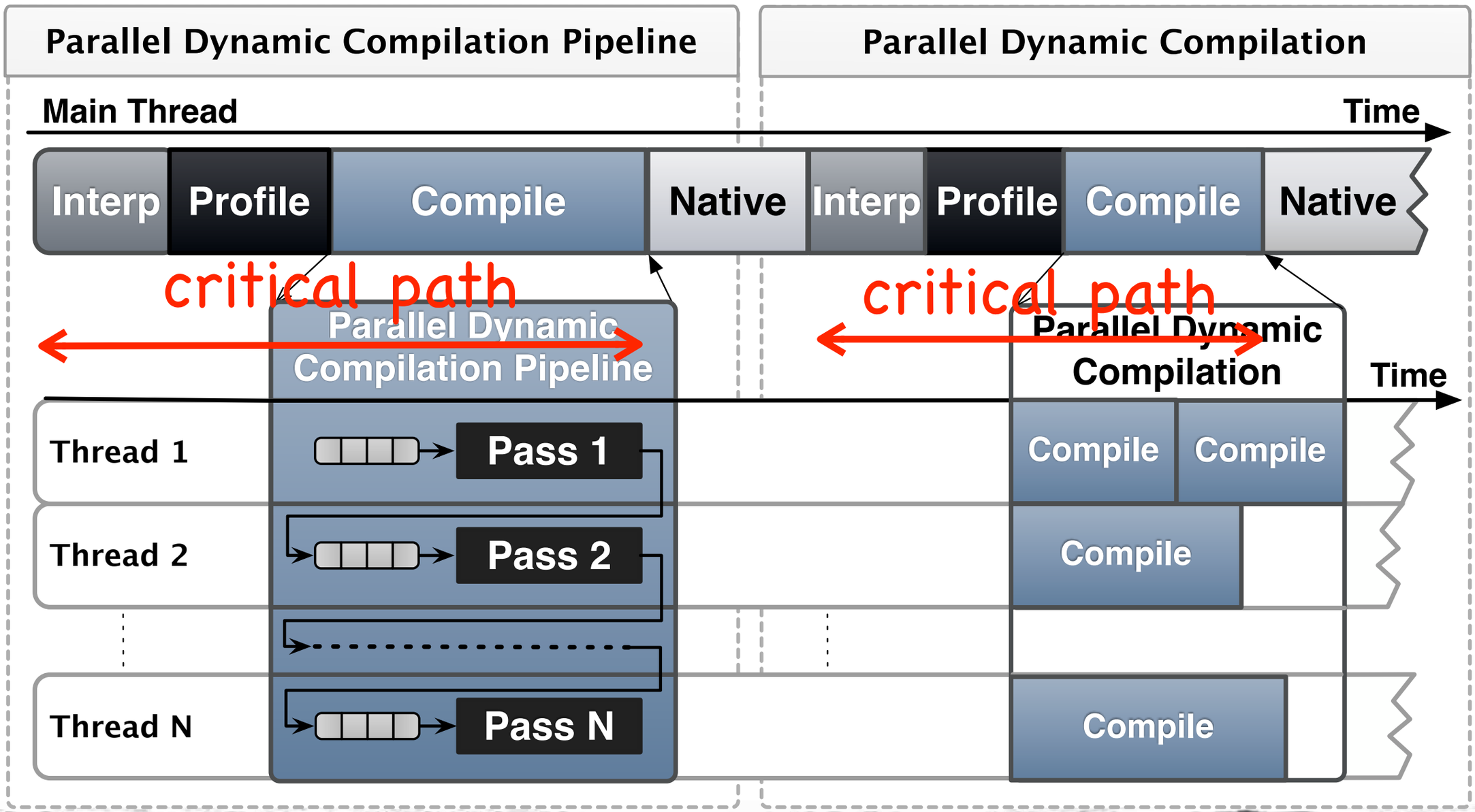
Backup Slide

State-of-the-Art Dynamic Compilation Strategies



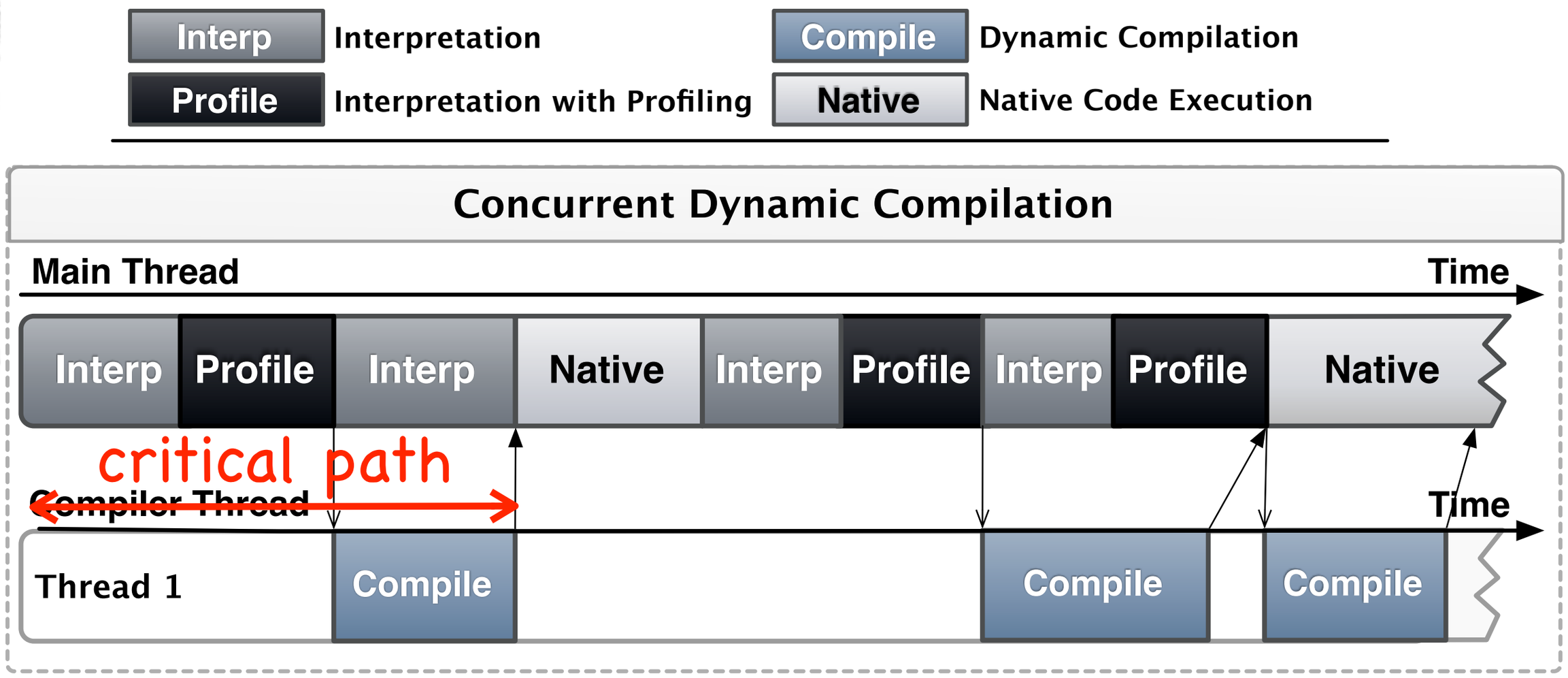
Backup Slide

State-of-the-Art Dynamic Compilation Strategies



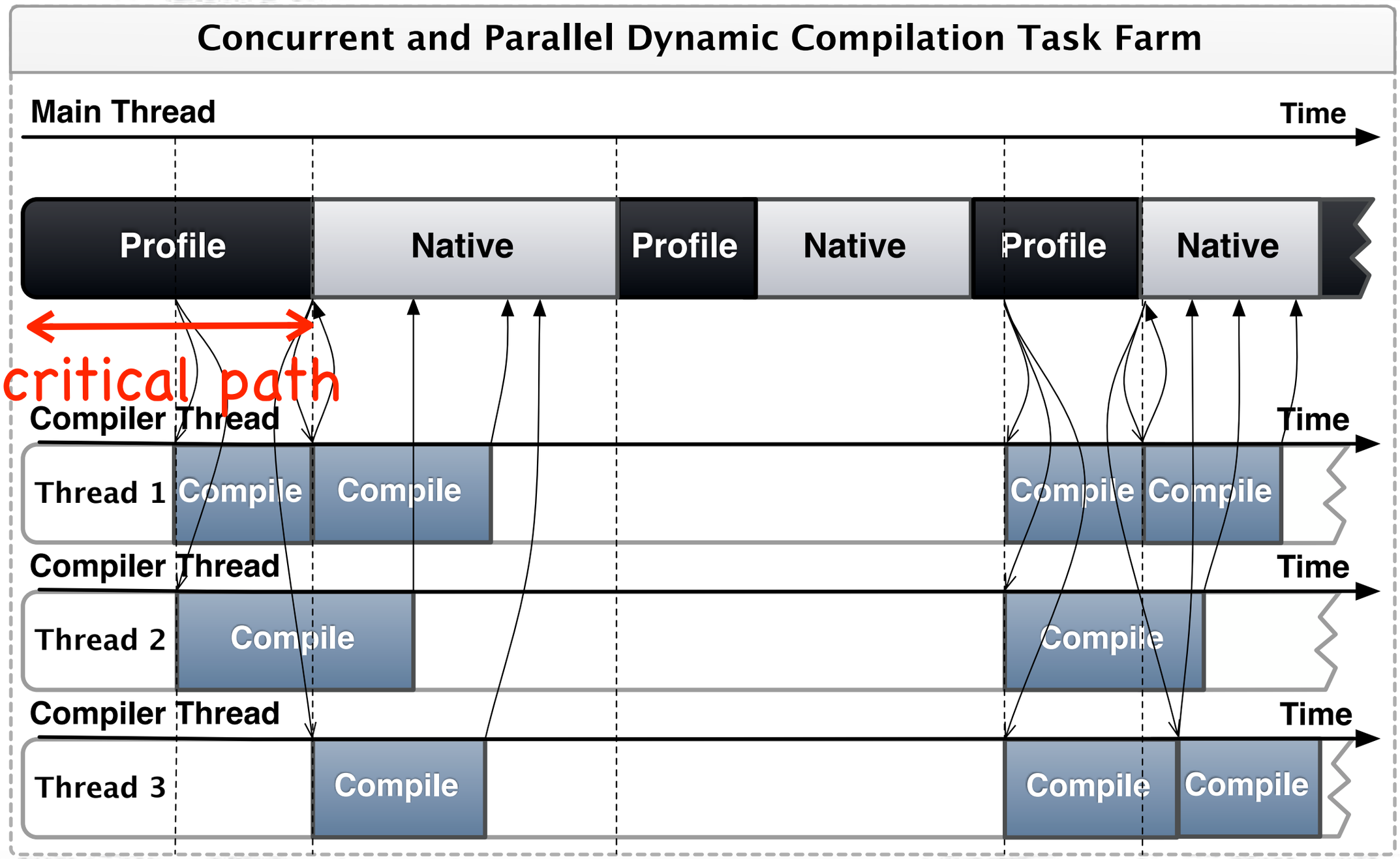
Backup Slide

State-of-the-Art Dynamic Compilation Strategies



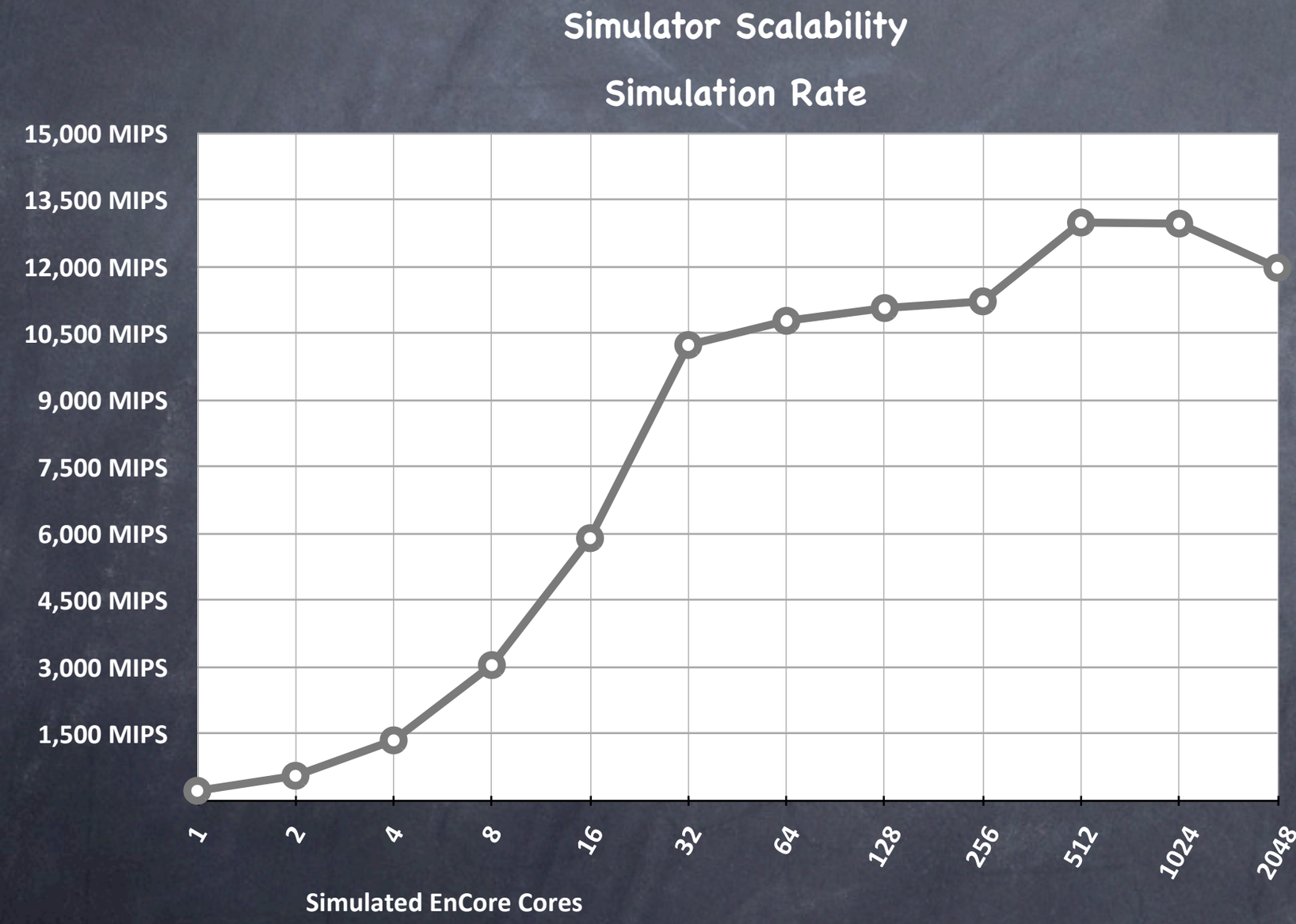
Backup Slide

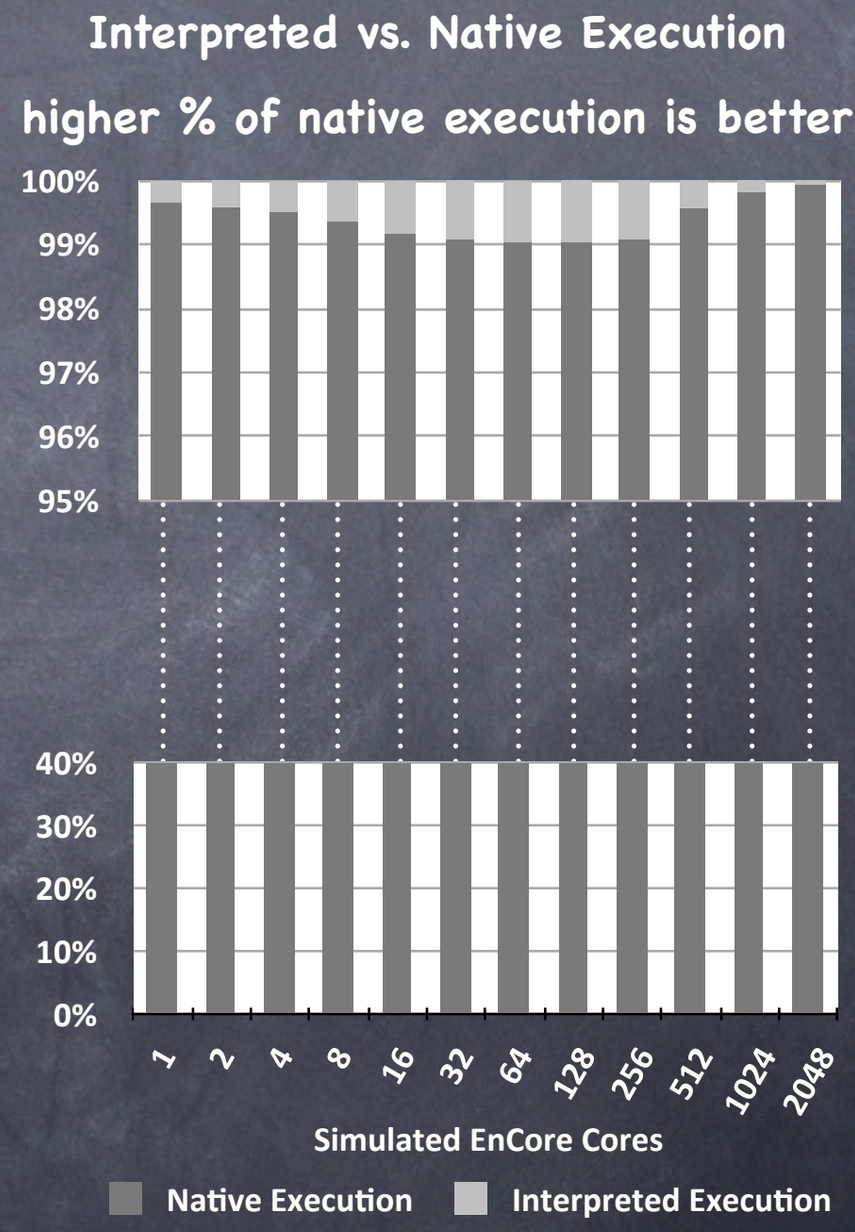
Concurrent and Parallel Dynamic Compilation



Backup Slide

How does this perform for multi-threaded applications?





LU - Splash 2 Benchmark

Measured on a 32-core machine

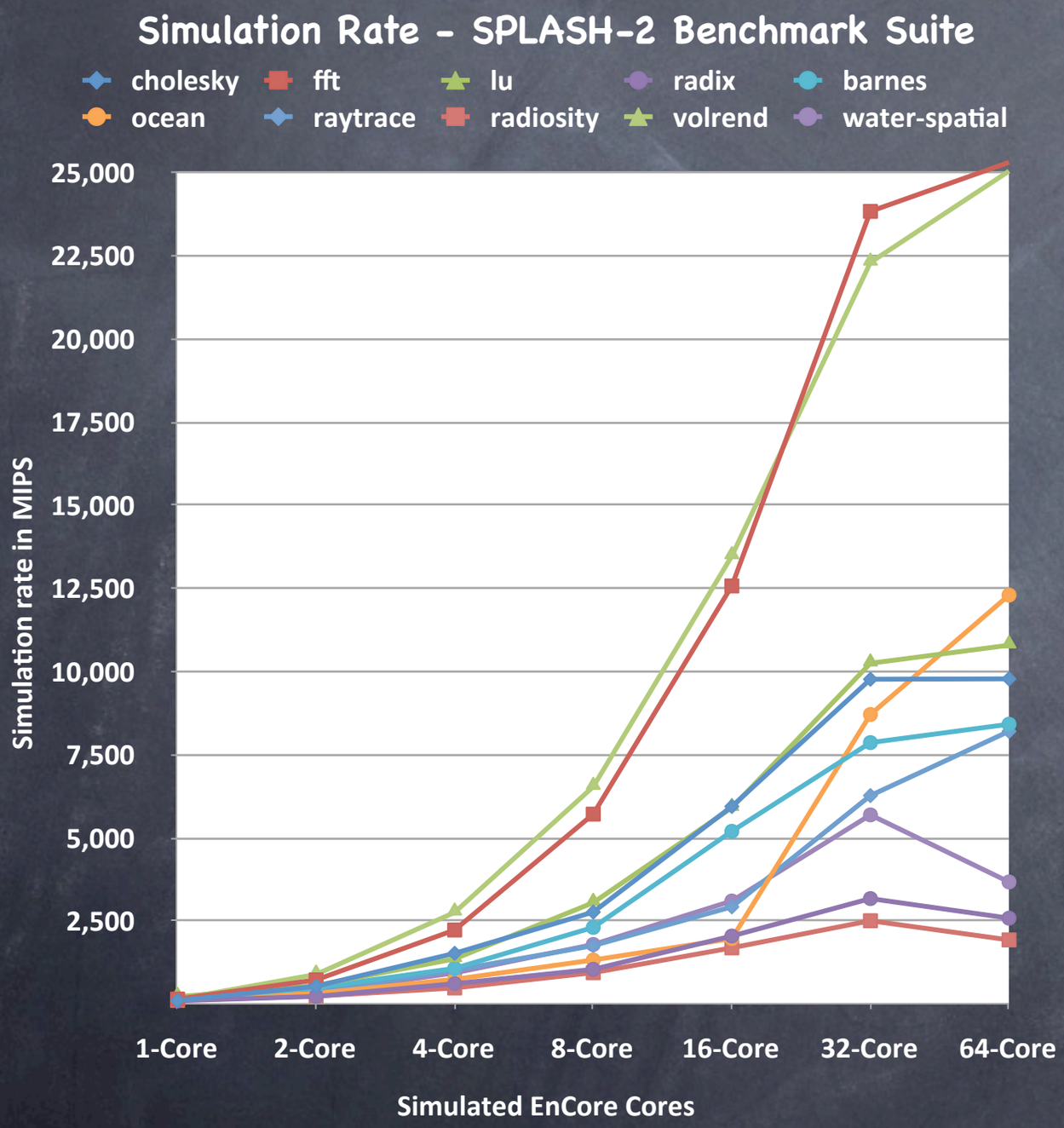
Backup Slide

How does this perform for multi-threaded applications?

Measured on a 32-core machine

Backup Slide

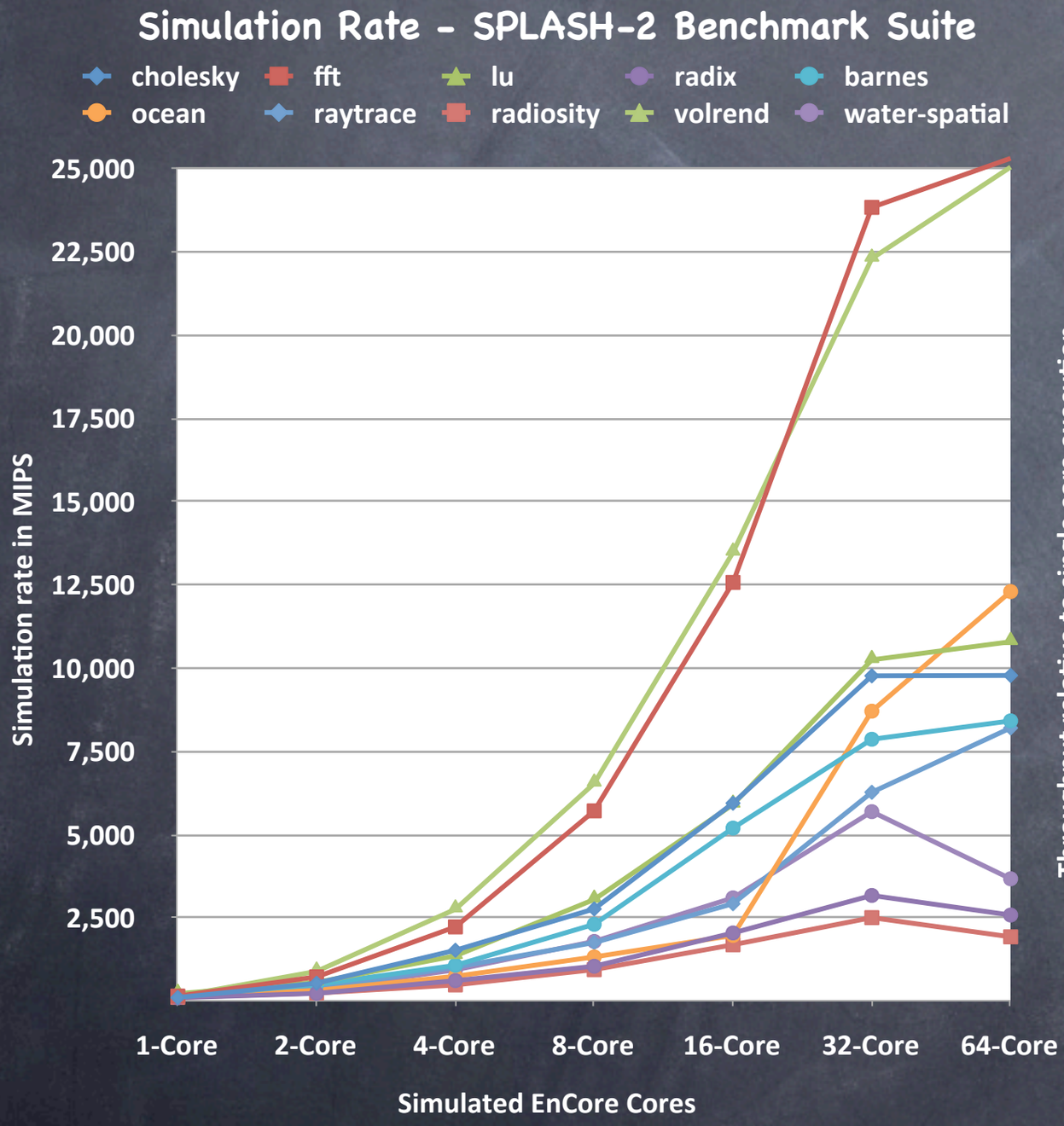
How does this perform for multi-threaded applications?

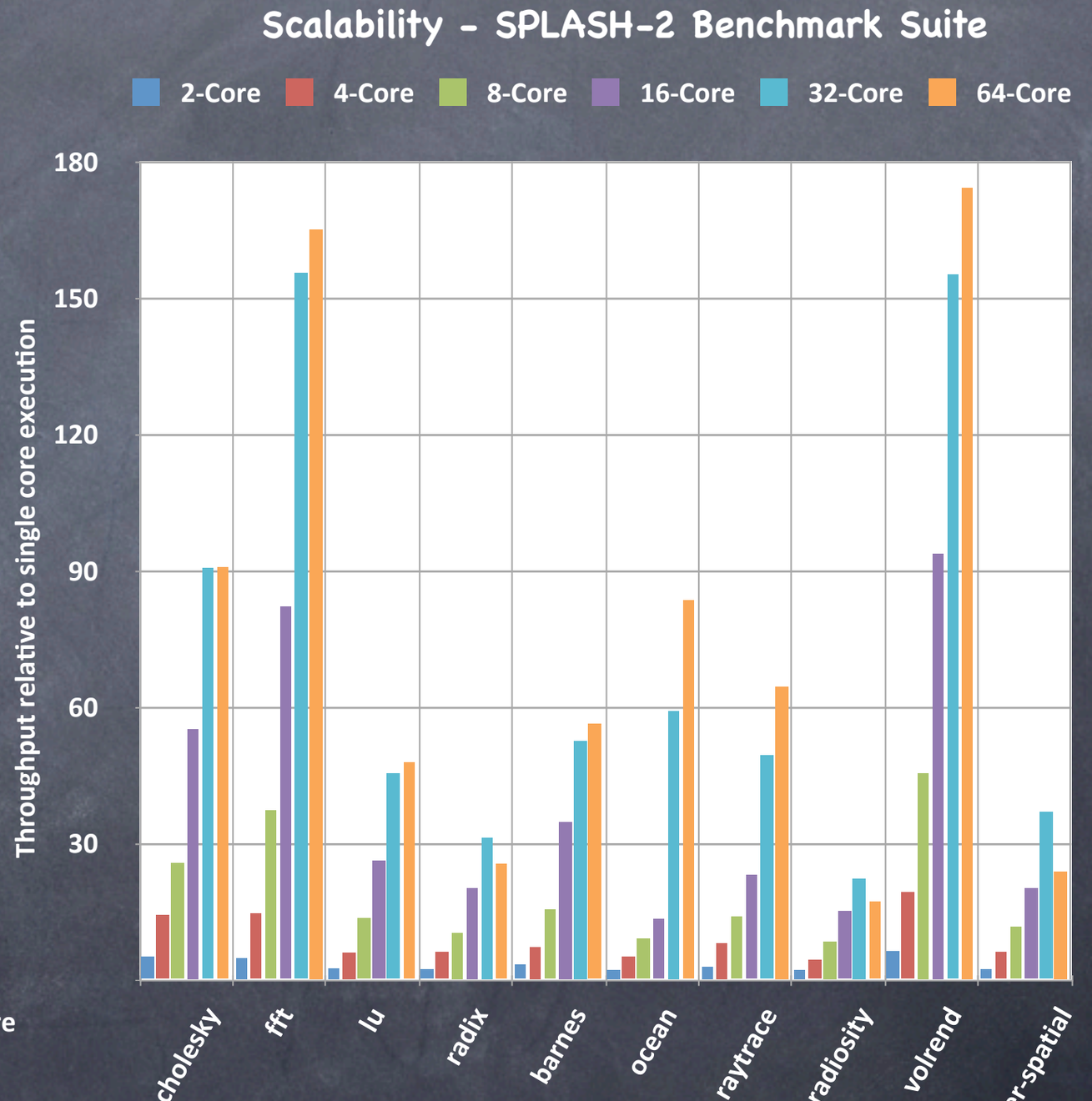


Measured on a 32-core machine

Backup Slide

How does this perform for multi-threaded applications?





Measured on a 32-core machine