Reducing Dynamic Compilation Latency

Igor Böhm
Concurrent and Parallel Dynamic Compilation

Igor Böhm
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
Dynamic Compilation using one Concurrent JIT Compiler

Main Thread

Compiler Thread

Thread 1

Time
Dynamic Compilation using one Concurrent JIT Compiler

- Main Thread
  - Interp
  - Profile
  - Interp
  - Native
  - Interp
  - Profile
  - Interp
  - Profile
  - Native

- Compiler Thread
- Thread 1
  - Compile
  - Compile
  - Compile
Dynamic Compilation using one Concurrent JIT Compiler

Main Thread

Compiler Thread

Thread 1
Dynamic Compilation using one Concurrent JIT Compiler

Main Thread

Time

Critical path
Dynamic Compilation using one Concurrent JIT Compiler

Main Thread

1. Interpretation
2. Profile
3. Interpretation
4. Native
5. Interp
6. Profile
7. Interp
8. Profile
9. Native

Compiler Thread

Thread 1

Compile

Dynamic Compilation using Concurrent and Parallel JIT Compiler Task Farm

Main Thread

Profile

Native

Profile

Native

Profile

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

1. **Main Thread**
   - Interp
   - Profile
   - Interp
   - Native
   - Interp
   - Profile
   - Interp
   - Profile
   - Native

   **Compiler Thread**
   - Compile
   - Compile

   **Critical Path**

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

   **Main Thread**
   - Profile
   - Native
   - Profile
   - Native
   - Profile
   - Native

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

Main Thread

1. Interpretation
2. Profile
3. Interpretation
4. Native
5. Interpretation
6. Profile
7. Interpretation
8. Native

Critical Path

Dynamic Compilation using Concurrent and Parallel JIT Compiler Task Farm

Main Thread

1. Profile
2. Native
3. Profile
4. Native
5. Profile
6. Native

Critical Path

Thread 1

1. Compile
2. Compile

Thread 2

1. Compile
2. Compile

Thread 3

1. Compile
2. Compile
3. Compile

Thread 4

1. Compile
2. Compile
3. Compile
4. Compile
Dynamic Compilation using one Concurrent JIT Compiler

Main Thread

<table>
<thead>
<tr>
<th>Interp</th>
<th>Profile</th>
<th>Interp</th>
<th>Native</th>
<th>Interp</th>
<th>Profile</th>
<th>Interp</th>
<th>Profile</th>
<th>Native</th>
</tr>
</thead>
</table>

Compiler Thread

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Compile</th>
<th>Compile</th>
</tr>
</thead>
</table>

Dynamic Compilation using Concurrent and Parallel JIT Compiler Task Farm

Main Thread

<table>
<thead>
<tr>
<th>Profile</th>
<th>Native</th>
<th>Profile</th>
<th>Native</th>
<th>Profile</th>
<th>Native</th>
</tr>
</thead>
</table>

Compiler Thread

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Compile</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Thread 2</th>
<th>Compile</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Thread 3</th>
<th>Compile</th>
</tr>
</thead>
</table>
Dynamic Compilation using one Concurrent JIT Compiler

Time

Main Thread

Interp | Profile | Interp | Native | Interp | Profile | Interp | Profile | Native

Compiler Thread

Thread 1

Compile

Thread 2

Compile

Thread 3

Compile

Dynamic Compilation using Concurrent and Parallel JIT Compiler Task Farm

Time

Main Thread

Profile | Native | Profile | Native | Profile | Native

Compiler Thread

Thread 1

Compile

Thread 2

Compile

Thread 3

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
- CIL
- JavaScript

Dynamic
- ARCompact
- 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

- Basic Block
- CFG Edges
- Return to Interpreter
Incremental Code Discovery

Trace Interval

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>A</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>F</th>
<th>G</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
</table>

Time

Region after $t'$

A

B

C

D

Basic Block

CFG Edges

Return to Interpreter
Incremental Code Discovery
Incremental Code Discovery

Trace Interval

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>A</th>
<th>F</th>
<th>G</th>
<th>A</th>
<th>B</th>
<th>I</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
</table>

Time

- \( t' \)
- \( t'' \)
- \( t''' \)

Region after \( t' \)

- A
- B
- C
- D

Region after \( t'' \)

- A
- B
- C
- D
- E
- F
- G

Region after \( t''' \)

- A
- H
- B
- C
- D
- E
- I
- F
- G

Basic Block

CFG Edges

Return to Interpreter
Incremental Code Discovery

Region == Dynamic CFG

Trace Interval

Time

A B C D E F G A

A B

F G A B

I F G

Region after t'''

A

H B

C

D

E I

F

G

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)

Regions break tracing into intervals

page is a fixed size container for translation

Simulation

Time

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

Page is a fixed size container for translation

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

Region 1
Region 2
Region 3
Region 4

Regions

Async registration of compiled regions

Hide Dynamic Compilation Latency

Page is a fixed size container for translation

Page (size variable)
- 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

Time

Region 1
Region 2
Region 3
Region 4

Dynamic Compilation Worker Thread 1
Dynamic Compilation Worker Thread 2
Dynamic Compilation Worker Thread 3

Native Code Execution
Native Code Discovery
Trace
Native
Trace
Native
Trace
Native
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)

Interval 1
Interval 2
Interval 3
Interval 4
Interval 5
Temporal Region Partitioning
Spatial Region Partitioning

Regions

Dynamic Code Discovery
Native Code Execution

Simulation

Trace
Native
Trace
Native
Trace
Native

Dynamic Compilation Worker Thread 1
Region 1
Region 3
Dynamic Compilation Worker Thread 2
Region 2
Dynamic Compilation Worker Thread 3
Region 4

Exploit Task Parallelism
Region 5
Region 9
Region 11
Region 6
Region 8
Region 12
Region 7
Region 10

Page (size variable)
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

Execution Loop

- PC address
  - Block Translated
    - Yes → Native Code Execution
    - No → New Block
  - New Block
    - Yes → Record Block in Region
    - No → Interpretive Block Simulation
  - Interpretive Block Simulation
    - End of Trace Interval
      - Yes → Analyse Recorded Regions
      - No → Hot Regions Present
    - No → End of Trace Interval
  - Analyse Recorded Regions
    - Yes → Enqueue Hot Regions and Continue
    - No → Hot Regions Present
  - Hot Regions Present
    - Yes → Enqueue Hot Regions and Continue
    - No → End of Trace Interval
Concurrent and Parallel JIT Compiler Design

Execution Loop

PC address

Block Translated

Yes

Native Code Execution

No

New Block

Yes

Record Block in Region

No

Interpretive Block Simulation

End of Trace Interval

Yes

Recorded Regions

No

Analyse Recorded Regions

Hot Regions Present

Yes

Enqueue Hot Regions and Continue

No

Enqueue

Continue

Native Code Execution

Translation Priority Queue

Region 1
Region 2
Region 6
Region 7
Region N

Concurrent Shared Data-Structure
Concurrent and Parallel JIT Compiler Design

Execution Loop

PC address

Block Translated

Yes

Native Code Execution

No

New Block

Yes

Record Block in Region

No

Interpretive Block Simulation

End of Trace Interval

Yes

Analyse Recorded Regions

No

Hot Regions Present

Yes

Enqueue Hot Regions and Continue

No

Enqueue

Continue

Concurrent and Parallel Dynamic Compilation Task Farm

Translation Priority Queue

Region 1
Region 2
Region 6
Region 7
Region N

Concurrent Shared Data Structure

Dequeue and Farm Out

JIT 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

Interpretive Block Simulation

Record Block in Region
Concurrent and Parallel JIT Compiler Design

Dynamic work scheduling

Execution Loop

Concurrent and Parallel Dynamic Compilation Task Farm

JIT Compilation Task Farm

Concurrent and Parallel JIT Compiler Design
Concurrent and Parallel JIT Compiler Design

Execution Loop

- **PC address**
  - Block Translated
    - Yes: Native Code Execution
    - No: New Block
      - Yes: Record Block in Region
      - No: Interpretive Block Simulation
        - Yes: End of Trace Interval
        - No: Analyse Recorded Regions
          - Yes: Hot Regions Present
            - Yes: Enqueue Hot Regions and Continue
            - No: Continue
          - No: Enqueue Hot Regions and Continue

- **Dynamic work scheduling**
- **Adaptive hotspot selection**

Concurrent and Parallel Dynamic Compilation Task Farm

- Translation Priority Queue
  - Region 1
  - Region 2
  - Region 6
  - Region 7
  - Region N

- Concurrent Shared Data-Structure
- Dequeue and Farm Out

JIT 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
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

Baseline

Speedup

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

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

Data measured on standard x86 quad-core machine
Speedup BioPerf

Speedup

Baseline

clustalw 0.34
fasta-ssearch 0.11
promlk 0.06
grappa 0.47
hmmsearch 0.94
hmmpfam 0.12
tcoffee 0.81
blastp 0.68
glimmer 0.19
ce 0.08
average 0.44

Measured on standard x86 quad-core machine

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

Measured on standard x86 quad-core machine
### Speedup BioPerf

**Measured on standard x86 quad-core machine**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Baseline</th>
<th>Interpreted-only Execution</th>
<th>Execution using concurrent JIT Compiler</th>
<th>Execution using concurrent and parallel JIT Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>clustalw</td>
<td>0.34</td>
<td>1.07</td>
<td>2.01</td>
<td>1.38</td>
</tr>
<tr>
<td>fasta-ssearch</td>
<td>0.11</td>
<td>1.02</td>
<td>1.52</td>
<td>1.01</td>
</tr>
<tr>
<td>promlk</td>
<td>0.06</td>
<td>1.35</td>
<td>2.08</td>
<td>1.43</td>
</tr>
<tr>
<td>grappa</td>
<td>0.47</td>
<td>1.03</td>
<td>1.29</td>
<td>1.01</td>
</tr>
<tr>
<td>hmmsearch</td>
<td>0.94</td>
<td>0.68</td>
<td>1.35</td>
<td>0.81</td>
</tr>
<tr>
<td>hmmpfam</td>
<td>0.12</td>
<td>0.68</td>
<td>1.52</td>
<td>1.03</td>
</tr>
<tr>
<td>tcoffee</td>
<td>0.81</td>
<td>0.19</td>
<td>1.52</td>
<td>0.94</td>
</tr>
<tr>
<td>blastp</td>
<td>0.68</td>
<td>0.19</td>
<td>1.03</td>
<td>0.94</td>
</tr>
<tr>
<td>glimmer</td>
<td>0.19</td>
<td>0.08</td>
<td>1.03</td>
<td>0.94</td>
</tr>
<tr>
<td>ce</td>
<td>0.08</td>
<td>0.44</td>
<td>1.03</td>
<td>0.94</td>
</tr>
<tr>
<td>average</td>
<td>0.44</td>
<td>1.38</td>
<td>1.38</td>
<td>1.38</td>
</tr>
</tbody>
</table>
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]

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Speedup</th>
<th>Interpreted-only Execution</th>
<th>Execution using concurrent JIT Compiler</th>
<th>Execution using concurrent and parallel JIT Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench</td>
<td>0.88</td>
<td>0.10</td>
<td>0.09</td>
<td>0.09</td>
</tr>
<tr>
<td>bzip2</td>
<td>0.89</td>
<td>0.10</td>
<td>0.08</td>
<td>0.09</td>
</tr>
<tr>
<td>gcc</td>
<td>0.21</td>
<td>0.17</td>
<td>0.18</td>
<td>0.18</td>
</tr>
<tr>
<td>milc</td>
<td>0.08</td>
<td>0.11</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>gobmk</td>
<td>0.17</td>
<td>0.13</td>
<td>0.13</td>
<td>0.13</td>
</tr>
<tr>
<td>soplex</td>
<td>0.08</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>povray</td>
<td>0.11</td>
<td>0.15</td>
<td>0.15</td>
<td>0.15</td>
</tr>
<tr>
<td>hammer</td>
<td>0.02</td>
<td>0.09</td>
<td>0.09</td>
<td>0.09</td>
</tr>
<tr>
<td>sjeng</td>
<td>0.15</td>
<td>0.09</td>
<td>0.09</td>
<td>0.09</td>
</tr>
<tr>
<td>libquantum</td>
<td>0.15</td>
<td>0.26</td>
<td>0.26</td>
<td>0.26</td>
</tr>
<tr>
<td>h264ref</td>
<td>0.09</td>
<td>0.09</td>
<td>0.09</td>
<td>0.09</td>
</tr>
<tr>
<td>lbm</td>
<td>0.09</td>
<td>0.10</td>
<td>0.10</td>
<td>0.10</td>
</tr>
<tr>
<td>omnetpp</td>
<td>0.26</td>
<td>0.09</td>
<td>0.09</td>
<td>0.09</td>
</tr>
<tr>
<td>astar</td>
<td>0.09</td>
<td>0.11</td>
<td>0.11</td>
<td>0.11</td>
</tr>
<tr>
<td>sphinx4</td>
<td>0.09</td>
<td>0.17</td>
<td>0.17</td>
<td>0.17</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>1.22</td>
<td>0.28</td>
<td>0.28</td>
<td>0.28</td>
</tr>
<tr>
<td>average</td>
<td>0.85</td>
<td>0.85</td>
<td>0.85</td>
<td>0.85</td>
</tr>
</tbody>
</table>

Measured on standard x86 quad-core machine

**Note:**
- 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]

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Interpreted-only Execution</th>
<th>Execution using concurrent JIT Compiler</th>
<th>Execution using concurrent and parallel JIT Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench</td>
<td>0.88</td>
<td>1.07</td>
<td>1.15</td>
</tr>
<tr>
<td>bisect</td>
<td>0.89</td>
<td>1.07</td>
<td>1.15</td>
</tr>
<tr>
<td>gcc</td>
<td>0.27</td>
<td>1.08</td>
<td>1.15</td>
</tr>
<tr>
<td>mcf</td>
<td>0.86</td>
<td>1.01</td>
<td>1.15</td>
</tr>
<tr>
<td>milc</td>
<td>0.13</td>
<td>1.01</td>
<td>1.15</td>
</tr>
<tr>
<td>gobmk</td>
<td>0.08</td>
<td>1.01</td>
<td>1.15</td>
</tr>
<tr>
<td>soplex</td>
<td>0.11</td>
<td>1.01</td>
<td>1.15</td>
</tr>
<tr>
<td>povray</td>
<td>0.08</td>
<td>1.01</td>
<td>1.15</td>
</tr>
<tr>
<td>hammer</td>
<td>0.18</td>
<td>1.02</td>
<td>1.16</td>
</tr>
<tr>
<td>sjeng</td>
<td>0.07</td>
<td>1.03</td>
<td>1.16</td>
</tr>
<tr>
<td>liquarium</td>
<td>0.13</td>
<td>1.04</td>
<td>1.16</td>
</tr>
<tr>
<td>h264ref</td>
<td>0.26</td>
<td>1.06</td>
<td>1.16</td>
</tr>
<tr>
<td>lbm</td>
<td>0.09</td>
<td>1.01</td>
<td>1.16</td>
</tr>
<tr>
<td>omnetpp</td>
<td>0.09</td>
<td>1.07</td>
<td>1.15</td>
</tr>
<tr>
<td>astar</td>
<td>0.15</td>
<td>1.04</td>
<td>1.15</td>
</tr>
<tr>
<td>sphinx3</td>
<td>0.18</td>
<td>1.04</td>
<td>1.15</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>0.21</td>
<td>1.04</td>
<td>1.15</td>
</tr>
<tr>
<td>average</td>
<td>0.24</td>
<td>1.15</td>
<td>1.15</td>
</tr>
</tbody>
</table>

**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]
Speedup EEMBC CoreMark
very short running embedded benchmarks
[worst-case scenario]

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

403.gcc - Regions Compiled

- Regions compiled T1
- Regions compiled T3

Time Interval
Effect of Concurrent and Parallel JIT Compilation on Throughput

403GCC - Regions Compiled

- Regions compiled T1
- Regions compiled T3

Time Interval
Effect of Concurrent and Parallel JIT Compilation on Throughput

403.gcc - Regions Compiled

- Regions compiled T1
- Regions compiled T3

Time Interval
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

- Regions compiled T1
- Regions compiled T3

higher throughput

403.gcc - Average Queue Length

- Avg Queue Length T1
- Avg Queue Length T3
Effect of Concurrent and Parallel JIT Compilation on Throughput

403.gcc - Regions Compiled
- Regions compiled T1
- Regions compiled T3

higher throughput

403.gcc - Average Queue Length
- Avg Queue Length T1
- Avg Queue Length T3
Effect of Concurrent and Parallel JIT Compilation on Throughput

higher throughput
Effect of Concurrent and Parallel JIT Compilation on Throughput

- Higher throughput
- Smaller queue length

403.gcc - Regions Compiled
- Regions compiled T1
- Regions compiled T3

403.gcc - Average Queue Length
- Avg Queue Length T1
- Avg Queue Length T3
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)

- **Regions**
  - **Thread 1**
  - **Thread 2**

- **Interval**
  - Interval 1
  - Interval 2
  - Interval 3
  - Interval 4
  - Interval 5

- **Native Code Execution**
  - Native

- **Shared Regions**
  - A
  - B
  - C
  - D

- **Dynamic Compilation**
  - Worker Thread 1
  - Worker Thread 2
  - Worker Thread 3
Concurrent and Parallel JIT Compilation in Action
(trace sharing)

Regions
Thread 1

T1

Region 1
Region 2
Region 3
Region 4

Interval 1
Interval 2
Interval 3
Interval 4
Interval 5

Tracing
Native
Native
Tracing
Native

Threads

T1
T2

Regions
Thread 2

Region 1
Region 2
Region 3
Region 4

Tag Entry

Region Tagged for Multiple Threads
Register Translation for T1 and T2

Tag Existing Entry

Shared Regions

A
B
C
D

Dynamic Compilation
Worker Thread 1

Region 1
Region 2
Region 3
Region 4

Dynamic Compilation
Worker Thread 2

Region 1
Region 4

Dynamic Compilation
Worker Thread 3

Region 2
Region 3

Native Code Execution
Native

Regions
Thread 1

T1

Region 1
Region 2
Region 3
Region 4

Interval 1
Interval 2
Interval 3
Interval 4
Interval 5

Tracing
Native
Native
Tracing
Native

Threads

T1
T2

Regions
Thread 2

Region 1
Region 2
Region 3
Region 4

Tag Entry

Region Tagged for Multiple Threads
Register Translation for T1 and T2

Tag Existing Entry

Shared Regions

A
B
C
D

Dynamic Compilation
Worker Thread 1

Region 1
Region 2
Region 3
Region 4

Dynamic Compilation
Worker Thread 2

Region 1
Region 4

Dynamic Compilation
Worker Thread 3

Region 2
Region 3

Native Code Execution
Native

Regions
Thread 1

T1

Region 1
Region 2
Region 3
Region 4

Interval 1
Interval 2
Interval 3
Interval 4
Interval 5

Tracing
Native
Native
Tracing
Native

Threads

T1
T2

Regions
Thread 2

Region 1
Region 2
Region 3
Region 4

Tag Entry

Region Tagged for Multiple Threads
Register Translation for T1 and T2

Tag Existing Entry

Shared Regions

A
B
C
D

Dynamic Compilation
Worker Thread 1

Region 1
Region 2
Region 3
Region 4

Dynamic Compilation
Worker Thread 2

Region 1
Region 4

Dynamic Compilation
Worker Thread 3

Region 2
Region 3

Native Code Execution
Native

Regions
Thread 1

T1

Region 1
Region 2
Region 3
Region 4

Interval 1
Interval 2
Interval 3
Interval 4
Interval 5

Tracing
Native
Native
Tracing
Native

Threads

T1
T2

Regions
Thread 2

Region 1
Region 2
Region 3
Region 4

Tag Entry

Region Tagged for Multiple Threads
Register Translation for T1 and T2

Tag Existing Entry

Shared Regions

A
B
C
D

Dynamic Compilation
Worker Thread 1

Region 1
Region 2
Region 3
Region 4

Dynamic Compilation
Worker Thread 2

Region 1
Region 4

Dynamic Compilation
Worker Thread 3

Region 2
Region 3

Native Code Execution
Native
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

Video Decoding and Playback

Full System OS Simulation
Thank You