A Trace-based Java JIT Compiler
Retrofitted from a Method-based Compiler

Hiroshi Inoue†, Hiroshige Hayashizaki†,
Peng Wu‡ and Toshio Nakatani†
† IBM Research – Tokyo
‡ IBM Research – T.J. Watson Research Center
Goal and Motivation

- **Goals**
  1. develop an efficient *trace-based Java JIT compiler (trace-JIT)* based on existing mature *method-based JIT compiler (method-JIT)*
  2. understand the benefits and drawbacks of the trace-JIT against the method-JIT

- **Why not method-JIT?**
  - Limited optimization opportunities in larger application with a flat execution profile (no hot spots)

- Can trace-JIT provide more optimization opportunities than method-JIT for such applications?
Background: Trace-based Compilation

- Using a **Trace**, a hot path identified at runtime, as a basic unit of compilation
Motivating Example

- A trace can span multiple methods
  - Free from method boundaries
  - In large server workloads, there are deep (>100) layers of methods
Outline

- Motivation
- Background
- Trace-JIT Architecture
- Performance Evaluation
- Future work and Summary
Baseline Method-JIT Components

Java VM

- interpreter
- compiled method dispatcher
- garbage collector
- class libraries

Method-JIT

- compilation request for frequently executed methods
- code generator
- IR generator
- optimizers

compilation request for frequently executed methods
Our Trace-JIT Architecture

Java VM

- interpreter
- trace dispatcher
- garbage collector
- class libraries

Tracing runtime

- trace selection engine
- hash map
- code cache

Trace-JIT

- trace side exit elimination
- IR generator
- code generator
- optimizers

(execution events)

(trace (Java bytecode))

(e.g. compiled code address)

(compiled code)

(new component)

(modified component)

(unmodified component)
Our Trace-JIT Architecture

- **Java VM**
  - **Interpreter**
  - **Execution events**

- **Tracing runtime**
  - **Trace selection engine**
  - **Trace (Java bytecode)**
  - (e.g. compiled code address)

- **Trace-JIT**
  - **Trace side exit elimination**
  - **IR generator**
  - **Code generator**
  - **Optimizers**

- **Modified to call a hook at control-flow events**:
  - branch
  - method invoke
  - method return
  - exception

- **Components**
  - **New component**
  - **Modified component**
  - **Unmodified component**

- **Code cache**

- **Garbage collector**

- **Class libraries**

A Trace-based Java JIT Compiler Retrofitted from a Method-based Compiler

© 2011 IBM Corporation
Our Trace-JIT Architecture

Java VM
- interpreter
- execution events

Tracing runtime
- trace selection engine
- trace (Java bytecode)

Trace-JIT
- trace side exit elimination
- IR generator
- optimizers

identify two types of hot paths
- linear trace
- cyclic trace

A

B

exit

stub

exit

A

B

exit

stub

modified component

unmodified component

identify two types of hot paths

A Trace-based Java JIT Compiler Retrofitted from a Method-based Compiler
Trace Selection

1. Identify a hot trace head
   - a taken target of a backward branch
   - a bytecode that follows a exit point of an existing trace
2. Record next execution path starting from the trace head
3. Stop recording when the trace being recorded:
   - forms a cycle (with our *false loop filtering*)
   - executes a backward branch
   - calls or returns to a JNI (native) method
   - throws an exception
   - reaches pre-defined maximum length (128 basic blocks)
Our Trace-JIT Architecture

Java VM

- interpreter
  - execution events

Tracing runtime

- trace selection engine
  - trace (Java bytecode)
- hash map
  - (e.g. compiled code address)

Trace-JIT

- trace side exit elimination
- IR generator
- optimizers
- code generator

compiled code
- code cache
- garbage collector
- class libraries
- new component
- modified component
- unmodified component
Our Trace-JIT Architecture

Java VM
- interpreter
- trace dispatcher
  - execution events

Tracing runtime
- trace selection engine
- hash map
  - trace (Java bytecode)
  - (e.g. compiled code address)

Trace-JIT
- code generator
- IR generator
- optimizers
- trace side exit elimination

Java stack design compatible with interpreter
to reduce overhead in JIT ↔ interpreter transitions

new component
modified component
unmodified component
Our Trace-JIT Architecture

Java VM
- interpreter
- trace dispatcher
- garbage collector
- class libraries

Tracing runtime
- trace selection engine
- hash map
- code cache
- compiled code

Trace-JIT
- trace side exit elimination
- IR generator
- code generator
- optimizers
- modified component
- unmodified component

execute events
trace (Java bytecode)
(e.g. compiled code address)
Technical challenge in reusing a method-based compiler for trace-JIT

**Scope mismatch problem**

- In method-JIT,
  - local variables **must be dead** at the start and the end of compilation scope

- In trace-JIT
  - local variables **may live** at the start and the end of compilation scope

→ Live range of local variables does not match with compilation scope in trace-JIT
Solving the scope mismatch problem

- **dead store elimination (DSE) as an example**

```java
void prepend(e) {
    p = head;
    do {
        tail = p;
        p = p->next;
    } while (p != NULL);
    tail->next = e;
    e->next = NULL;
}
```

Is this dead store? (no use in the trace)

No!

we analyze **outside** the compilation scope to identify liveness at the end of compilation scope
Analyze outside the compilation scope

- We identify all live variables at each compilation scope boundary point
  - trace head, trace exit points
- For each boundary point, we analyze the method that includes the point
  - mostly in *live range analyzer* and *use-def analyzer* in the framework, not in each optimizer
Our Trace-JIT Architecture

Java VM
- interpreter
- trace dispatcher
- garbage collector
- class libraries

Tracing runtime
- trace selection engine
- trace (Java bytecode)
- trace side exit elimination

Trace-JIT
- IR generator
- optimizers
- modified component
- unmodified component

Apply simple one-path value propagation to exploit simple topologies of traces

It removes potential side exits to reduce IR tree size and compilation time
Our techniques to reduce overhead

- **Hash Lookup Reduction Using a Shadow Array**
  - Allocate a shadow array for each method to store information corresponding to each bytecode (e.g. start address of compiled trace starting from that bytecode)
  - Lookup the shadow array instead of slow global hash map

- **JNI inclusion**
  - Include certain JNI methods into traces and call JNIs from traces directly
  - Reduced trace enter/exit overhead
  - Some recognized JNI methods are further optimized (inlined)
Our Trace-JIT Architecture

Java VM
- interpreter
- trace dispatcher
- execution events
- garbage collector
- class libraries

Tracing runtime
- trace selection engine
- trace (Java bytecode)
- hash map
- (e.g. compiled code address)

Trace-JIT
- trace side exit elimination
- IR generator
- code generator
- optimizers
- modified component
- unmodified component

key: bytecode address
value:
- start address of the compiled code
- counter to identify hot trace head
Shadow Array

- Shadow array is allocated for each method lazily (we allocate it on demand).
- The shadow array entry corresponding to a bytecode index can be easily found without hash calculation.

Method structure:

<table>
<thead>
<tr>
<th>name/signature</th>
<th>length</th>
<th>bytecodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>shadowArray</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bytecodes:
- 1 byte / entry

Shadow array:
- 4 bytes / entry

Start address of compiled code:
- NULL
- NULL
- NULL
- NULL
Outline

- Motivation
- Background
- Trace-JIT Architecture
- Performance Evaluation
- Future work and Summary
Performance Evaluation

- Hardware: IBM BladeCenter JS22
  - 4 cores (8 SMT threads) of POWER6 4.0GHz
  - 16 GB system memory
- Software:
  - AIX 6.1
  - Method-JIT: IBM JDK for Java 6 (32 bit)
  - Trace-JIT: Our Trace-JIT based on the same IBM JDK
    - used only standard optimization level (-Xjit:optlevel=warm)
    - 512 MB Java heap with large page enabled
    - generational garbage collector (gencon)
- Benchmark:
  - DaCapo benchmark suite 9.12
Steady state performance

- Trace-JIT was 22% slower to 26% faster than method-JIT
Execution time breakdown

😊 Trace-JIT often (not always) shows better JITted code performance (blue parts)
😊 Trace-JIT incurs larger runtime overhead (orange parts)
Execution time breakdown by trace length

These long traces typically crossed 80+ method boundaries

more inlining effect than method-JIT with inlining

larger compilation scope yields

😊 less runtime overhead due to transition

😊 more compiler optimization opportunities

shorter trace <- trace length in number of Java bytecodes -> longer trace
Inlining effect of trace JIT (number of method boundaries included in one trace)

Trace-JIT provides larger compilation scope than method-JIT with inlining
😊 Less method invocation overhead, more compiler optimization opportunities
😢 Potentially larger JITted code size due to duplicated code among traces
Compiled code size

- The larger code size was mainly caused by the duplicated codes among traces.
- On the positive side, traces include only frequently executed code sequence.

The graph shows a comparison of compiled code size for different applications, with the y-axis representing the relative compiled code size over method-JIT. The bar chart indicates that our trace-JIT generated 10% more code on average, with a geomean of 4.9.
Effect of hash lookup reduction using a shadow array

- improved performance by 27.4% on average
- using additional memory space: 1.3 MB on average and up to 6.8 MB (tomcat)
Effect of JNI inclusion

- improved performance by up to 2.7x (for sunflow) and about 15% on average
Outline

- Motivation
- Background
- Trace-JIT Architecture
- Performance Evaluation
- Future work and Summary
Optimization opportunities and challenges

- **Opportunities**
  - Potentially larger compilation scope than method-JIT
  - Simple control flow
    - main path of a trace is a very large extended basic block
  - Explicit control flow
    - like method inlining
  - More specialization
    - type specialization, value specialization etc

- **Challenges**
  - Interaction between trace selection and optimizations
    - e.g. Loop optimizations
Future work: Effective Loop Optimization in trace-JIT

- More loop optimizations in trace-JIT
  - backward-branch-based cyclic trace identification is not suitable for loop optimizations
  
  need to enhance trace selection algorithm to maximize the optimization opportunities

Java code:

```java
for (int i=0; i<4; i++) {
    j ++;
}
```
Summary

- We implemented trace-based Java JIT compiler based on the existing method-based JIT compiler
  - handling scope mismatch problem
  - reducing runtime overhead

- Our trace-JIT achieved almost comparable performance to mature method-JIT with almost same set opt optimizations
  - better JITted code performance in trade for larger runtime overhead
  - generating longer trace is a key to superior performance

Refer to the paper for
✓ our new runtime overhead reduction techniques
✓ more detailed comparisons including code size, compilation time and so on