The Potentials and Challenges of Trace Compilation:

Lessons learned from building a trace-JIT on top of J9 JVM

(Joint work with Hiroshige Hayashizaki and Hiroshi Inoue, IBM research)
Trace-based compilation

- **Traces** are created based on frequently executed paths recorded, and compiled and executed instead of methods.
- A trace captures hot paths only.
A Brief History of Trace Compilation

- Trace compilation was commonly in software binary translators (SDT)
  - e.g., DynamoRIO, Strata

- Dynamo was the first to demonstrate optimization potentials (PLDI’00)
  - average 10% speedup over binaries compiled at –O2
  - improvements come mostly from better code layout and simple folding

- HotpathVM (VEE’06) and YETI are the first two Java trace JITs
  - target resource constrained system where full blown JITting is not practical
  - claim trace compilation is cost effective and easy to develop
  - demonstrate several to 10X speedup over interpreter, but are considerably slower than Hotspot

- TraceMonkey (PLDI’08) is the first Javascript trace JIT
  - use trace compilation for aggressive type specialization
  - demonstrated 2x to 20x speedups on loop-intensive scripts
  - outperformed V8 (method-based) on some benchmarks

- PyPy is the first Python trace-JIT
  - tracing through Python runtime and interpreter, which are both implemented in RPython
  - use trace compilation for aggressive concretization
  - best performing Python implementation but with compatibility issues

- Other dynamic scripting language trace-JIT
  - HotPy (Python) and LuaJIT (Lua)
Open Questions of Trace Compilation

1. How does the best trace JIT today compare to the best method-JIT?

2. What is (the essence of) trace compilation?

3. What makes trace compilation be potentially better than method-JIT?

4. Can trace compilation be better than method compilation?
Outline

- Introduction
- Potentials of trace compilation
- Improve trace selection
- Evaluation of FIORANO Trace-JIT
- Research challenges
- Summary and Future work
Dynamism and Concretization

Concretization: The process of concretizing a general principle or idea by delineating, particularizing, or exemplifying it.

- Dynamism in dynamic typing
  - a variable can be of different types during program execution, so do operations
- Type concretization
  - convert generic types/operations to specific types/operations

A, B, C, D are dynamically typed, BINARY_ADD is type generic

Concretization happens naturally during execution

<table>
<thead>
<tr>
<th>run 1</th>
<th>run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD A (int)</td>
<td>LOAD A (float)</td>
</tr>
<tr>
<td>LOAD B (int)</td>
<td>LOAD B (float)</td>
</tr>
<tr>
<td>BINARY_ADD (int)</td>
<td>BINARY_ADD (float)</td>
</tr>
<tr>
<td>LOAD C (int)</td>
<td>LOAD C (float)</td>
</tr>
<tr>
<td>BINARY_ADD (int)</td>
<td>BINARY_ADD (float)</td>
</tr>
<tr>
<td>STORE D (int)</td>
<td>STORE D (float)</td>
</tr>
</tbody>
</table>
Dynamism and Concretization (Cont‘)

Dynamism in control-flow: a conditional or indirect branch may take different control-flow directions at different time

- Control-flow concretization
  - take a specific path among many possible control-flow paths
  - smallest concretization happens when selecting one of the outgoing edges of a dispatch node

- Call concretization
  - make callee visible to the caller
  - otherwise a call is opaque to the optimizer

if (x != 0)
while (!end)
call foo
method foo

Control-flow concretization
Call concretization
Potentials of Trace Compilation

The potential of trace compilation lies in its *unique* ability to concretize, e.g.,
- Type concretization
- Control-flow concretization
- Call concretization

However, concretization is hardly new and definite not unique to trace compilation
- Method-based inlining: call concretization
- Partial evaluation: control-flow and value concretization
- Versioning and specialization: general concretization
- Most of the above can be done in a method-JIT via analysis or profiling

What’s uniqueness is *how* concretization is done by trace compilation
- Concretization is done by a *seemingly dumb* but *yet very powerful* heuristics: by following a real execution
Concretization by Profiling vs. Tracing

Example of method inlining
1. Profile targets of (important) call-sites
2. Determine which call-sites should be inlined
3. Inlined the call target into the caller

Profile-directed inlining appears to be very intelligent:
- Profiling is customarily designed for method inlining
- Can decide whether or what to inline
- Can use many heuristics to improve inlining precision and cost effectiveness

Example of tracing through methods
1. Select the head of a new trace
2. Start recording the trace by adding instructions being executed to a buffer
3. Terminate recording if a trace termination condition is encountered

Tracing appears to be a very dumb profiling mechanism:
- e.g., trace recording may happen to follow the execution of a cold path
- “Inlining” benefit may be incidental as tracing is not designed for inlining
- Hard to control or fine-tune inlining effect

Following real execution conveys a crucial information, context sensitivity
- tracing can achieve complete context-sensitive with very little cost
- profiling is context insensitive or shallowly context sensitive (heavyweight profiling)
A trace can span multiple methods
- Free from method boundaries
- In Web Server workloads, there are deep (>100) layers of methods

Trace Compilation Potential: Concretize control-flow dynamism in large workloads.
The FIORANO Trace-JIT Project

- Goal: Improve the performance of large Java applications, e.g., WebSphere Application Server (WAS)

- The current IBM Java JIT compiler (and almost all Java JIT compilers) uses method-based compilation
  - most effective when optimizing a few program hot-spots

- Large Java server applications typically exhibit a flat execution profile
  - no really hot methods, deep call chains, small methods, not loop intensive

What’s unique in our approach?

- Explore trace-based compilation for Java on server workloads
  - Use trace formation to expand compilation scope beyond method boundaries

- Build a trace-JIT on top of a method-based JVM/JIT
  - We believe the essence of trace compilation is about better region formation
  - Opportunity to directly compare trace-JIT and method-JIT
Outline

- Introduction
- Potentials of trace compilation
- Improve trace selection
- Evaluation of FIORANO Trace-JIT
- Research challenges
- Summary and Future work
The Essence of Trace Compilation

While compilation theories (mostly method-based) have matured over many decades, there are two unique aspects of trace compilation.

- **Trace selection** as the algorithm to select codes for compilation
  - Method-JIT counterpart would be (hotness-based) method selection + inlining
  - Trace selection, as an algorithm, behaves very differently from method selection

- **Trace optimization** as how to effectively optimize/codegen traces
  - Traces as a compilation unit are distinctively different from methods
    - e.g., it may stop and end at arbitrary points of a method
  - Can traditional method-based optimizations be applied to traces?
    - POPL’11 proves the (un)soundness of some conventional optimizations when applied to traces
  - In our CGO’11 paper, we addressed such unsoundness by extending a method-JIT to support trace compilation, our main conclusions are:
    - *Scope mismatch* is a main cause of the unsoundness of conventional optimizations
    - A method-based compilation framework can be extended to compile traces by addressing a few key differences between traces and methods
    - Many trace-specific optimizations can be done by conventional redundancy elimination optimizations
Trace Selection

- **Trace selection** forms traces out of executed instructions at runtime
  - An active area of research as it is at the heart of any trace compilation system

- Dynamo pioneered a form of two-step trace selection, called *next-executing-tail* (NET)
  1. Trace head selection: identify starting point of a trace by frequency-profiling a pool of potential trace heads
     A. Targets of backward branches (i.e., loop headers), or
     B. Instructions immediately following the exit point of a trace (exit-heads)
  2. Trace recording: record a trace from the selected trace head until meeting one of the trace termination conditions, e.g.,
     A. when encountering the head of an already formed trace
     B. When encountering a backward taken branch
     C. when the trace recording buffer overflows
NET/LEI Selection

(a) Control-flow graph

(b) NET/LEI selection

- Traces are initially built from targets of backward branches
- They gradually grow out of side-branches (side-exits) of existing traces
- Trace size is determined by the termination conditions used in trace recording
Last-Executing-Iteration (LEI) Selection

- NET selection termination condition
  - A. when encountering the head of an already formed trace
  - B. when encountering a backward taken branch
  - C. when the trace recording buffer overflows

- LEI selection improves condition#2 in NET selection
  - A. when encountering the head of an already formed trace
  - B. when detected a repeating PC in recording buffer
  - C. when the trace recording buffer overflows

Why is LEI better than NET?
- repeating-PC detects cyclic paths
- stop-at-backward-branch prevents formation of cyclic traces starting from non loop-headers
Trace-Tree Selection

- NET and LEI are one-pass trace selection algorithms
  - traces are formed out of one recording and are typically linear
- Trace-tree selection is a multi-pass trace selection algorithm that
  - combines linear traces formed from multiple recording
  - forms traces with tree topology to represent loops
  - used in HotpathVM, TraceMonkey, and SPUR

(a) control-flow graph of a loop
(b) a trace-tree
Improving Trace Selection for our Trace-JIT

- Our baseline selection algorithm for the Trace-JIT is LEI
  - In order to compete with the method-JIT, the trace-JIT needs to match the coverage of the method-JIT (>99%)
  - One-pass selection algorithms can achieve target coverage and LEI is the state-of-the-art
  - Trace-Tree selection has a coverage issue for non-loopy codes

- Using LEI selection, our trace-JIT is about 45% slower than J9/JIT, even after
  - the trace-JIT achieved >99% coverage
  - aggressive runtime overhead reduction
  - enabling most of the warm-level optimizations in the method-JIT
Limitation of stop-at-existing-head termination conditions:
- Limit the ability to capture cyclic paths: one cyclic path per loop
- Limit the ability to “inline”: when a method entry becomes a trace head, no subsequence trace can “inline” this method
Limitation of Stop-at-existing-head: “Inlining” Effect

Stop-at-existing head limits the “inlining” effect on traces
– when a trace is formed at the entry point of a method, the method cannot be “inlined” to any subsequent traces
Limitation of Stop-at-Repeating-PC

– Repeating PC detects cycles in a flow-graph
– Cyclic paths do not always represent truly repeating execution patterns
– *False* loops are often formed on execution paths with multiple invocations to the same method

Details see our ASPLOS’11 paper
– Proposed *call-stack comparison* at repeating PCs to detect false loops
– Proposed other approximation algorithms to detect false-loops
– Crystallized the concept of repetition detection in trace selection

**Figure 1.** False loop and false loop filtering
Research Challenges of Trace Selection

- Trace selection algorithms today are mostly driven empirically
  - They are relatively easy to implement, but a lot harder to reason about

- Lack of deep understanding on trace selection from the algorithm’s point of view
  - What are the necessary components of a selection algorithm?
  - What are pathological behaviors of bad selection algorithm design?
  - How to evaluate a trace selection algorithm?

- Little understanding on inherent properties of the trace formation of a program
  - While there is one CFG (static representation) for a program, there can be many different trace formations
  - What defines a good trace and a good trace formation for a program?
  - While static program representation is based on graph theory, there is not much theoretical foundation of trace formation for a program

- The design space of trace selection is vast, and not thoroughly explored
  - How potential trace heads are selected
  - How traces are dispatched and counters are updated
  - How to terminate trace recording
  - How to group traces
  - Other unknown factors
Outline

- Introduction

- Potentials of trace compilation

- Improve trace selection

- Evaluation of FIORANO Trace-JIT

- Summary and Future work
Trace JIT Overview

Language VM (extended)
- CPython
  - J9 interpreter with hook
  - Trace dispatch
  - Trace selection engine

Trace runtime (new)
- Trace cache
- Counter cache
- Code cache
- Bytecode trace

Testarossa (extended)
- J9 frontend
  - Python frontend
  - IL generation
  - Code generation
  - Optimization
  - TRIL
Example of Trace Lifecycle

trace selection engine is driven by a sequence of events from interpreter.

trace selection engine forms a trace. Each BC is labeled by the method name and bcIndex to show its origin.
*(starting from JBiadd is not possible in Java method compilation!)*

IL gen translates the trace into TR-IR. Implicit loads are inserted before the first bytecode if the operand stack is not empty upon entry to the trace.

Code generator emits binary codes.
Basic Features of FIORANO Trace JIT

Where are we now?

- Built a robust trace JIT based on J9/Testarossa
  - Multithreading
  - Monitor entry/exit
  - Garbage Collection
  - Exception
  - Async compilation
  - JNI

- Enabled most warm-level optimizers except for some loop opts and escape analysis

- Thorough design space exploration on trace selection, new algorithms

- Significant efforts to reduce trace runtime overhead

Limitations of Trace-JIT

- No re-compilation support

- Interpreter profiling disabled

- Do not support most of the loop optimizers

- No escape analysis

- No hot-level optimizations (warm is the max)
Steady-State Performance of Fiorano Trace-JIT

Trace-JIT outperforms full-opt method-JIT
Trace Selection Improvements

![Graph showing steady-state performance improvements](image-url)
Inlining effect of trace JIT
(number of method boundaries included in one trace)

Trace-JIT provides larger compilation scope than method-JIT with (full) inlining
😊 Less method invocation overhead, more compiler optimization opportunities
😢 Potentially larger JITted code size due to duplicated code among traces
Trace Compilation vs Partial Inlining (I)

Partially inlined regions

- Inline a complete path of a callee method into the caller (space efficiency)
- Start and end at method boundaries with potential side-exits

Trace regions

- Formed out of runtime execution paths
- Partial inlining naturally occurs in traces
- Most trace regions start at non canonical program boundaries

Distribution of # Static Traces by Trace Starting Points

<30% traces start from loop header or method entry
Trace Selection vs. Method Inlining (II)

ASSUMPTION: when a call graph is too big to be fully inlined into the root node

Method (partial) inlining forms hierarchical regions

Trace selection forms contiguous regions
- blue, brown, green
Outline

- Introduction
- Potentials of trace compilation
- Improve trace selection
- Evaluation of FIORANO Trace-JIT
- Summary and Future work
Concluding Remarks

Trace compilation is an exciting new approach to dynamic compilation, but it is still at its early age of explorations

Findings that surprised us:

– Trace regions are truly different from method regions

– A method-based optimizer can be retrofitted to optimize (non-canonical) traces and be quite effective

– Extending trace scopes matters a lot in performance

– Increasing trace lengths are easy, controlling code size is tricky

– Linear traces are not necessarily inferior to structured traces

Open research questions:

– Can trace compilation compete with method compilation and why?
– What is the essence of trace compilation?

Our on-going explorations:

– Establish general theories on trace compilation
– Control code size without limiting trace scope
– Trace formation beyond linear traces
– Run real server workloads
References
