Improving the Performance of Trace-based Systems by False Loop Filtering

Hiroshige Hayashizaki*, Peng Wu#, Hiroshi Inoue*, Mauricio J. Serrano#, and Toshio Nakatani*

* IBM Research – Tokyo
# IBM Research – Watson Research Center
Outline

- **Motivation**
  - To improve the performance of trace-based compilation systems

- **Trace-based compilation**
  - emerging compilation approach
  - easy to implement, do specialization, break method boundaries

- **Problem**
  - Traces are too short and fragmented

- **Contribution**
  - Identified a problem of *false loops* that has not been addressed before
  - Propose *false loop filtering* to create longer traces
Outline

- Motivation
  - To improve the performance of trace-based compilation systems

- Trace-based compilation
  - emerging compilation approach
  - easy to implement, do specialization, break method boundaries

- Problem
  - Traces are too short and fragmented

- Contribution
  - Identified a problem of false loops that has not been addressed before
  - Propose false loop filtering to create longer traces
Trace-based compilation: trace selection [Bala+00, ...]

- **Traces** are the basic units for compilation
  - Traces are created dynamically based on frequently executed paths
  - Flexible & larger compilation scope
    - Rare/irrelevant blocks are omitted
    - A trace can span multiple methods

```
method f

... → A: ...
A: ... → B: call g()
B: call g() → C: ...
C: ... → while(...) → ...

while(...) → ... → R: rare path
R: rare path → Q: return
Q: return
```

```
method g

P: if(x==0)
P: if(x==0) → R: rare path
R: rare path
```

```
... → B: call g()
B: call g() → C: ...
C: ... → while(...) → ...
while(...) → ... → R: rare path
R: rare path → Q: return
Q: return
```
Trace-based compilation: trace selection [Bala+00, ...]

- **Traces** are the basic units for compilation
  - Traces are created dynamically based on frequently executed paths
  - Flexible & larger compilation scope
    - Rare/irrelevant blocks are omitted
    - A trace can span multiple methods
Trace-based compilation: trace selection [Bala+00, ...]

- **Traces** are the basic units for compilation
  - Traces are created dynamically based on frequently executed paths
  - Flexible & larger compilation scope
    - Rare/irrelevant blocks are omitted
    - A trace can span multiple methods
Trace-based compilation: trace selection [Bala+00, ...]

- **Traces** are the basic units for compilation
  - Traces are created dynamically based on frequently executed paths
  - Flexible & larger compilation scope
    - Rare/irrelevant blocks are omitted
    - A trace can span multiple methods

![Trace diagram]

Trace 1

A: ...
B: call g()
C: ...
while(...)
Trace-based compilation: trace execution [Bala+00, ...]

- A compiled trace is
  1. **entered** at trace head
  2. **executed** while the control is in the trace
  3. and **exited** to interpreter or to another trace

```
Trace 1

A: ...
B: call g()
C: ...
while(...)
P: if(x==0)
Q: return
```
Longer traces are preferable for high performance

- Entering/Exiting a trace (trace transition) is costly operation
  - doing compensation, looking up next trace to execute, etc.
- Spending longer time in a trace leads to better performance
  - Less trace transitions → less runtime overhead
  - Larger compilation scope → better optimizations, better code quality
Outline

- **Motivation**
  - To improve the performance of trace-based compilation systems

- **Trace-based compilation**
  - emerging compilation approach
  - easy to implement, do specialization, break method boundaries

- **Problem**
  - Traces are too short and fragmented

- **Contribution**
  - Identified a problem of *false loops* that has not been addressed before
  - Propose *false loop filtering* to create longer traces
Our Contribution

- Identified a problem of **false loops** in a basic class of trace selection algorithms that has not been addressed before
- Propose **false loop filtering** and improved dynamic trace length
  - Dynamic trace length = number of basic blocks executed between entering a trace and exiting from the trace

### Average Dynamic Trace Length (in DaCapo)

- **Higher is Better**
- **traces became up to 6.5x longer**
- **traces became 2.2x longer on average**
Traces are terminated at loops [Hiniker+05, ...]

- When a loop is detected in the execution path, traces are terminated at that point
  - To capture loops in a trace
- This heuristic works well
  *if the detected loop is in fact a loop* (= is iterated many times)
Traces are terminated at loops [Hiniker+05, ...]

- When a loop is detected in the execution path, traces are terminated at that point
  - To capture loops in a trace
- This heuristic works well
  if the detected loop is in fact a loop (= is iterated many times)
Loop detection by repeating instruction address [Hiniker+05..]

- If the same instruction address appears in an execution path twice, then the path between two occurrences is detected as a loop.
- Detected loops are expected to be repeated many times.
False loops

- An execution path that starts and ends at the same instruction address but is NOT repeatedly executed
- Example: a method g is called twice from different call sites
  - There is a pathological false loop starting from the first call to g and ending at the second call to g!
False loops

- An execution path that starts and ends at the same instruction address but is NOT repeatedly executed
- Example: a method g is called twice from different call sites
  - There is a pathological *false loop*
    - starting from the first call to g and ending at the second call to g!

```plaintext
method f

for(){
  A: ...
  B: call g()
  C: ...
  D: call g()
  E: ...
}

method g

Executed Inst Addr
A: ...
B: call g()
method g()
C: ...
D: call g()
method g()
E: ...
A: ...
```

Same
False loops

- An execution path that starts and ends at the same instruction address but is NOT repeatedly executed
- Example: a method g is called twice from different call sites
  - There is a pathological *false loop* starting from the first call to g and ending at the second call to g!

```
method f

for(){
  A: ...
  B: call g()
  C: ...
  D: call g()
  E: ...
}

method g

B: call g()
method g()
C: ...
D: call g()
method g()
E: ...
A: ...
```

Executed Inst Addr

- A: ...
- B: call g()
- method g()
- C: ...
- D: call g()
- method g()
- E: ...
- A: ...

Same
Problems caused by false loops

- Traces become shorter and fragmented
- Lower compiled code quality and more runtime overhead
  - Traces do not capture repeated execution patterns
  - More trace transitions

Obtained traces

Preferred trace

```
for(){
  A: ...
  B: call g()
  C: ...
  D: call g()
  E: ...
}
```

```
for(){
  A: ...
  B: call g()
  C: ...
  D: call g()
  E: ...
}
```
Outline

- **Motivation**
  - To improve the performance of trace-based compilation systems
- **Trace-based compilation**
  - emerging compilation approach
  - easy to implement, do specialization, break method boundaries
- **Problem**
  - Traces are too short and fragmented
- **Contribution**
  - Identified a problem of *false loops* that has not been addressed before
  - Propose *false loop filtering* to create longer traces
Our proposal: False Loop Filtering

- Detect false loops and do not consider them as loops
  - do not terminate traces at false loops
- Calling contexts are different at the beginning and end of the false loop

**Preferred trace**

```
for(){
    A: ...
    B: call g()
    C: ...
    D: call g()
    E: ...
}
```
False Loop Filtering by Call Stack Comparison

- False loops can be detected by comparing the call stacks, in addition to instruction address.
  - If the instruction addresses are the same but the call stacks are different, then a false loop is detected.
Variations of False Loop Filtering Algorithms

- False loop filtering can be used in systems where full call stack information is not available (e.g. binary tracing systems)
  - False loop filtering can be done without full call stacks by using approximation heuristics with some inaccuracy
Evaluation - Trace-based Java JIT compiler

- We applied false loop filtering to our trace-based Java VM/JIT compiler
  - Developed based on IBM's J9 JVM
  - Creates linear traces
  - Our trace-based Java achieves nearly the same performance as the J9 JVM with the same optimization level
  - Details of our trace JIT will be also presented at CGO 2011
  - Executed on IBM POWER 6 (4.0GHz, 4-core 8-thread), AIX 6.1
Result - Dynamic Trace Length

- Dynamic trace length increased up to 6.5x
Result - Execution time

- **16% faster** on average (*up to 37%*) for DaCapo benchmark
  - Traces became longer
    - Less transition → lower overhead
    - Better optimizations
- Execution time of false loop filtering itself is negligible

**Relative Execution Time (steady-state)**

Lower is Better

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>without false loop filtering</th>
<th>with false loop filtering</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bloat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>chart</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eclipse</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hsqldb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jython</td>
<td></td>
<td></td>
</tr>
<tr>
<td>luindex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lusearch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>prmd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xalan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>geomean</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

- **Motivation & Problem**
  - To improve the performance of trace-based compilation systems
  - Longer traces are preferable: lower transition cost and better optimization
  - But traces were too short and fragmented

- **Contribution**
  - Identified a problem of *false loops* that has not been addressed before
    - False loop is an execution path that starts and ends at the same instruction address but is NOT repeatedly executed
    - Traces are terminated at false loops and become shorter and fragmented
  - Propose *false loop filtering* by comparing call stacks to create longer traces
    - Showed performance improvement up to 37%
  - Considering the problem of false loops is important when designing new trace selection algorithms