A Dynamic Tool for Finding Redundant Computations in Native Code

Software and Solutions Group

Kyungwoo Lee, Zino Benaissa, Juan Rodriguez

Intel Corporation

July 21, 2008
Motivation

- Compiler optimizations do not always deliver their full performance potential
  - Complicated interactions between optimizations
  - Unforeseen interactions with the target architecture

- Compiler engineers spend significant portion of their time tuning optimizations
  - Focus on hot-code of specific application
  - Use HW profile to assist identifying inefficiencies in the generated code

Could we build a new tool to help assess the quality of compiler optimizations?
## Why use Dynamic Analysis?

<table>
<thead>
<tr>
<th>Dynamic</th>
<th>Static</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pros</strong></td>
<td><strong>Cons</strong></td>
</tr>
<tr>
<td>- handles indirect jump, and dynamically loaded libraries</td>
<td>- Indirect jump and DLLs are a problem</td>
</tr>
<tr>
<td>- Observes the same address and value as they run</td>
<td>- Values cannot be observed. Symbolic analysis is conservative and limited.</td>
</tr>
<tr>
<td>- Simple verification technique</td>
<td>- Require multiple analysis to be effective – e.g. alias analysis, symbolic, range and control-flow, call graph.</td>
</tr>
<tr>
<td><strong>Cons</strong></td>
<td><strong>Pros</strong></td>
</tr>
<tr>
<td>- Unexecuted path/code are not analyzed</td>
<td>- All execution paths are covered</td>
</tr>
<tr>
<td>- High runtime overhead but can be made reasonable</td>
<td>- No runtime overhead</td>
</tr>
</tbody>
</table>
Performance Opportunity Finder (POF)

- POF identifies potentially redundant computations
  - that require contextual analysis relevant to (global/interprocedural) compiler optimization
    - Redundant sign/zero extensions
    - Redundant constant spills
    - Missing copy/constant propagations

- POF is implemented using Intel’s PIN* dynamic instrumentation framework
  - POF uses program execution contexts (register/memory value) and program execution paths
  - POF determines invariants associated with redundant computations

*http://rogue.colorado.edu/pin
How POF Works?

1. Instrumentation: Performs (dynamic) instrumentation on candidate instructions associated with redundant computation patterns.

2. Dynamic Analysis: Checks if invariants on the patterns are satisfied during the program execution.

3. (Re-instrumentation): Unnecessary instrumentation is removed.

4. At program exit, reports the locations (virtual addresses) of redundant computations and how many times they execute.
(Potentially) Redundant Computation Patterns

- **Redundant Sign/Zero Extension**
  - Instruments every: `movsx/movzx dst, src`
  - Check whether the value of **src register** is already sign/zero extended in its super register

- **Redundant Constant Spill**
  - Instruments every: `mov [esp + offset], src`
  - Check whether the value of **src** is the same over the execution
Missing Copy/Constant Propagation

**def:** \( \text{mov dst, src} / \text{mov dst, const} \)

**use:** \( \text{inst ... dst ...} \)

- Check if neither \textit{dst} nor \textit{src} is defined along any possible path between the \textit{def} and \textit{use} sites
  - Potentially, all instructions having register operands need to be instrumented.

- Basic-block level instrumentation for efficiency
  - Initially summary information (Uses/Defs) is built for each basic block using a bit vector representation.
  - POF propagates summary information at each basic block entrance during execution.
Missing Copy/Constant Propagation (Cont.)

- Data flows along the actual execution path
  - No control flow/call graphs are required.
- POF Handles control-flow merge

\[
eip_1: \text{mov eax, ebx} \quad eip_2: \text{mov eax, edx}
\]

\[
eip_3: \text{add ecx, eax}
\]

- A copy pair \((\text{eax, ebx})\) is created.
- A missing copy propagation from \(eip_1\) to \(eip_3\) is identified.
- Another copy pair \((\text{eax, edx})\) is created.
- The missing copy propagation is invalidated.
Reducing POF’s Runtime Overhead

- **Static instrumentation removal**
  - If a benchmark has several input scenarios, POF skips instrumentation on instructions that were proven to be non-redundant in the previous execution.

- **Dynamic instrumentation removal**
  - POF investigates how instructions become non-redundant, and decides when to re-instrument the binary
    - If the number of instructions that dynamically become non-redundant exceeds a certain number, POF flushes out all previous instrumentations and re-instruments only remaining candidates.
Experiments

● Hardware
  – Intel® Core™ 2 Duo 2.4GHz 4MB L2, FSB 1066MHz, 2GB memory
  – SUSE™ Linux* 10.0 with kernel 2.6.5

● Software Configuration
  – Benchmarks: SPEC® CPU2006 (CINT)
  – Compilers:
    • GCC version 3.4 and version 4.0 for IA-32(x86) and Intel® 64(x86-64) respectively (/O2)

● Evaluate the code generated by compilers
  1. Static counts: Number of redundant computations found in the binary
  2. Hot-score [%] = D / D_hottest
     where D is the number of times the redundant instructions execute, and D_hottest is the number of times the hottest instruction executes.

*Other names and brands may be claimed as the property of others.
In 403.gcc and 445.gobmk, there are two consecutive zero-extended moves (movzx EAX, byte ptr [EBX] ... movzx EAX, AL)

Overall, x86-64 compilers have more redundant sign/zero extensions than x86 compilers.
Redundant Sign/Zero Extension

- No more than 2% of redundant sign/zero extensions have only 5% Hot-score – most redundant computations are not in the hot path.
- Relatively, x86-64 compilers have higher Hot-scored redundant sign/zero extensions than x86.
Overall, x86 compilers have more redundant constant spills than x86-64 compilers.

This happens because register pressure is higher in x86 compilers.
Redundant Constant Spill

• Hot-scores for all redundant constant spills do not exceed 55[\%].
• Relatively, x86-64 compilers have higher Hot-scored redundant constant spills than x86 compilers.
• Counts def sites (instructions associated with copy pairs).
• Overall, x86 have more missing copy/constant propagations than x86-64 compilers.
Relatively, x86-64 compilers have higher Hot-scored redundant copy/constant pairs than x86 compilers.

There is little difference between version 3.4 and 4.0 for x86 compilers.
POF Execution Time Relative To the Original Run

- Average of three execution time of all runs of the reference input data sets with all patterns enabled
- Overhead is affected by the number of redundant computations found.
- POF has an average of 19X slowdown relative to the original run.
Conclusion & Next Steps

• POF is an original tool to help assess the quality of compiler optimizations
  – Our first implementation supports three patterns
  – Using various GCC compilers, we have performed a comparative study on the redundant computations.

• Future work
  – Add more redundant computation patterns for both architecture-independent/dependent optimizations.
  – Generalize a pattern description rule
  – Share POF with compiler developers and see how effectively they fix deficiencies in compiler optimizations
Acknowledgement

- Mark Charney and Hongjiu Lu for help with Pin and compiler framework
- Our peers for their valuable feedback, help, and encouragement during this project