Re-constructing High-Level Information for Language-specific Binary Re-optimization

Toshihiko Koju
IBM Research - Tokyo

Reid Copeland
IBM Canada

Motohiro Kawahito
IBM Research - Tokyo

Moriyoshi Ohara
IBM Research - Tokyo
Language-specific binary optimizer can achieve competitive performance relative to the state-of-the-art source compiler.

Achieved +40.1% by our binary optimizer vs. +55.2% by the latest source compiler.
Outline

- Motivation & Goal.
- Our Approach.
  - Four Major Optimizations.
- Performance Evaluation.
- Conclusions.
Compilation Technologies are becoming Increasingly Important

- Recent processors are equipped with specialized HW units. – e.g. SIMD, FPGA, GPU, DFP (Decimal Floating-Point)

- Re-compilation with the latest compilation technologies is typically required to exploit them.
Compiler Migration is Often Challenging for Large Enterprise Customers

- Adapting to changes in the following can be a significant undertaking:
  - language, compiler option, build environment, runtime environment

Diagram:
- Downlevel processor
- Existing application
- Compiler migration
- Re-compilation
- Latest compiler
- Latest processor
- DFP
- SIMD
- FPGA
- GPU
- Re-compiled application
Ease Migration by Binary Optimization

- We have developed a production-level binary optimizer for mainframe COBOL applications.
  - IBM Automatic Binary Optimizer for z/OS.

- The binary optimizer utilizes the same optimization engine of the latest source code COBOL compiler.

**Source code re-compilation:**

- source code
- latest compiler
- re-compiled binary

**Binary re-optimization:**

- existing binary
- binary optimizer
- optimized binary

Provide strong compatibility with the current environment.
Traditional Approach does not Work

- Naive translation from machine instructions to IR (intermediate Representation) degraded the performance.
  - This is typically for binary translators based on general compiler engines.

![Relative Performance Chart]

- original binary
- naive translation w/o optimizations
- naive translation with optimizations
- source code re-compilation
Our Goal

- Achieve competitive performance relative to the state-of-the-art source code compiler.
Outline

- Motivation & Goal.
- Our Approach.
  - Four Major Optimizations.
- Performance Evaluation.
- Conclusions.
We Need High-level Information to Optimize Binaries

- Compiler optimizations require "HLI" (high-level information).
  - e.g. variables, data types, scopes, constants, library semantics

- Only limited HLI is provided by the naive translation.
  - Resulting in limited optimization opportunities.
Our Approach: Language-Specific Binary Optimization

- General binary optimizers typically do not assume source languages which results in better applicability.
  - e.g. Dynamo [Bala00], DynamoRIO [Bruening03], DBT86/RASP [Hertzberg11]

- Our binary optimizer assumes the source language to reconstruct HLI by using the "contextual information".
  - e.g. data structure, register usage, instruction pattern
Re-construction of the Key HLI

- **Machine instructions**
  - Abstract interpretation:
    - Data structure
    - Register usage

- **Use of EFA (Effective Address)**
  - Use-define analysis:
    - EFA
    - Data structure

- **Variable**
  - Liveness analysis:
    - Location
    - Data structure

- **Data structure**
  - Data type

- **Scope**
  - Generation of aliasing information:
    - Location
    - Scope

- **Aliasing**
  - Inspection of instruction pattern:
    - Variable
    - Instruction pattern

- **Constant value**
  - Determination of routine:
    - EFA
    - Data structure

- **Runtime routine**
  - Analyze parameter:
    - Semantics of routine
    - Parameter information

- **Original operation**
  - Inspection of literal pool:
    - EFA
    - Data structure
Abstract Interpretation: Analysis for EFA

- Take advantage of knowledge about data structures and register usages of COBOL.

### Machine instructions:
- (a) Load R2,(R9+0x200)
- (b) Load R3,(R2+0x100)
- (c) Load R4,(R2+0x200)
- (d) Load R5,(R2+R4)

### COBOL Data structures & register usage:
- **R9**
- **R13**
- **R12**

### Abstract interpretation:
- (a) EFA=control block+off_heap  
  R2=&heap
- (b) EFA=heap+0x100  
  R3=unknown
- (c) EFA=heap+0x200  
  R4=unknown
- (d) EFA=heap+unknown  
  R5=unknown

Without the contextual information, we cannot tell EFA which is a basis of many other HLI.
Four Major Optimizations

- There are four significant optimizations which are newly introduced to the latest source code compiler.
  1. Type Reduction of Decimal Type.
  2. Strength Reduction of Edited Moves.
Optimization 1: Type Reduction of Decimal Type (1/2)

- Convert decimal types to decimal floating-point (DFP) types.
  - Computation on decimal type: memory-to-memory
  - Computation on DFP type: register-to-register

**Representation of integer number 87:**

- Binary integer type: 87
- Zoned decimal type: 0xF8F7
- Packed decimal type: 0x087F

Business applications perform a lot of decimal computations.

**Simple Example:**

**COBOL:**
ZonedDecimal A, B;
A = A + B

**Original instruction:**
- ZonedToPacked (R13+272, 5 bytes), (R2+0, 8 bytes)
- ByteOR (R13+276), 0xf
- ZonedToPacked (R13+280, 5 bytes), (R2+8, 8 bytes)
- ByteOR (R13+284), 0xf
- PackedAdd (R13+272, 5 bytes), (R13+280, 5 bytes)
- PackedToZoned (R2+0, 8 bytes), (R13+272, 5 bytes)
Optimization 1: Type Reduction of Decimal Type (2/2)

Original instruction:

ZonedToPacked (R13+272, 5 bytes), (R2+0, 8 bytes)
ByteOR (R13+276), 0xf
ZonedToPacked (R13+280, 5 bytes), (R2+8, 8 bytes)
ByteOR (R13+284),0xf
PackedAdd (R13+272, 5bytes), (R13+280, 5 bytes)
PackedToZoned (R2+0, 8 bytes), (R13+272, 5 bytes)

Optimized instruction:

ZonedToDFP FPR1,*(A)
ZonedToDFP FPR2,*(B)
DFPAdd FPR1,FPR2
DFPToZoned FPR1,*FPR1

Use-Def Analysis: *D=DEF, U=Use

ZonedToPacked D U
ByteOR UD
ZonedToPacked D U
ByteOR UD
PackedAdd UD U
PackedToZoned U D

Variables: TMP1 TMP2 A B

Type Reduction

Re-constructed HLI about variables

ZonedToPacked TMP1, A
PackedSetSign TMP1, plus
ZonedToPacked TMP2, B
PackedSetSign TMP2, plus
PackedAdd TMP1, TMP2
PackedToZoned TMP1, TMP2

Variables:

© 2016 IBM Corporation
Optimization 2: Strength Reduction of Edited Moves

- Generate a specialized instruction sequence for a string format operation in COBOL.

**Simple Example:**

**COBOL:**
* FOO: string of the format 9999.
MOVE 123 TO FOO. ➔ FOO=0123

Original instruction:
EDIT control-bytes,numeric

EDIT is a millicode instruction which interprets each byte of the control-bytes.

Re-construct HLI about the original operation from control-bytes.

Optimized instruction:
ConvToZoned TMP, numeric
CopyZoned FOO, TMP
Optimization 3: Skipping Truncations of Binary Numbers

- Generate guard code to skip unnecessary truncation of binary integer numbers.

**Simple Example:**

**COBOL:**
* FOO: binary number with 4 digits.
  FOO=FOO+1000.

**Original code:**
  TMP=FOO+1000
  FOO=TMP%10000

Divide instruction is unconditionally executed to suppress overflows.

**Overflow is rare.**

**Optimized code:**
  TMP=FOO+1000
  if abs(TMP) >= 10000
     TMP=TMP%10000
  FOO=TMP

Re-construct HLI about the original operation:
- Quot is not in use.
- Dividend is a constant of Pow10.
Optimization 4: Inlining Efficient Version of Routines

- Replace a runtime routine with the equivalent code which exploits a more efficient algorithm and newer instructions.

**Simple Example:**

**COBOL:**
* FOO, BAR: large packed decimal numbers (e.g. 18 digits).
  FOO=FOO/BAR

**Original instruction:**
CALL LargeDivide

**Runtime path length is more than 100 instructions.**

**Optimized instruction:**
- PackedToDFP FPR1,FOO
- PackedToDFP FPR2,BAR
- DFPDivide FPR1,FPR2
- DFPRound FPR1
- DFPToPacked FPR1,FOO

**Exploit new DFP instructions.**

**Re-construct HLI about the original operation:**
- Semantics of the runtime routine.
- Parameter information.
Outline

- Motivation & Goal.
- Our Approach.
  - Four Major Optimizations.
- Performance Evaluation.
- Conclusions.
Experimental Environment

- **Machine:**
  - 5.5 GHz zEC12 running z/OS.

- **Benchmark:**
  - IBM Internal benchmark suite for the development of the COBOL compilers.

- **Binary optimizer:**

- **Original source code compiler:**

- **Latest source code compiler:**

<table>
<thead>
<tr>
<th>Decimal Benchmarks</th>
<th>Non-Decimal Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>cevalnd</td>
<td>telco</td>
</tr>
<tr>
<td>amort6</td>
<td>invupd</td>
</tr>
<tr>
<td>djpcom85</td>
<td>amort2</td>
</tr>
<tr>
<td>seqpr</td>
<td>atm06</td>
</tr>
<tr>
<td>ixseq</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>accounting computations.</td>
<td>telecom system.</td>
</tr>
<tr>
<td>mortgage payment comparisons and amortization computations (external FP).</td>
<td>updating sales master records.</td>
</tr>
<tr>
<td>record processing.</td>
<td>mortgage payment comparisons and amortization computations (long FP).</td>
</tr>
<tr>
<td>updating sequential file records.</td>
<td>automatic teller machine transactions.</td>
</tr>
</tbody>
</table>
Performance Impact of Each Optimization

- a. O0 (no optimization) → -37.7%
- b. O1 (without HLI) → -10.0%
- c. b. + Skipping truncation → -1.4%
- d. c. + Type reduction → +25.4%
- e. d. + Optimization of Edited move → +36.7%
- f. e. + inlining routines (full) → +40.1%

Improved performance from -10.0% to +40.1% by re-constructing HLI.

- The baseline (100%) is the performance of the original binaries.
  - Original binaries are compiled with the highest optimization-level of the old compiler.

Decimal benchmarks vs. non-decimal benchmarks

- Higher is better
- Relative Performance
- cevalnd, amort6, djpcom85, seqpr, ixseq, telco, invupd, amort2, atm06, geo. mean
Performance Comparisons with the Re-compilation

- The baseline (100%) is the performance of the original binaries.
  - Original binaries are compiled with the highest optimization-level of the old compiler.

The diagram shows relative performance comparisons with the re-compilation:

- f. e. + inlining routines (full) \( \rightarrow +40.1\% \)
- g. latest source code compiler \( \rightarrow +55.2\% \)

Good competitiveness of the binary optimizer relative to the state-of-the-art source code compiler.

Decimal benchmarks: cevalnd, amort6, djpcom85, seqpr, ixseq, telco, invupd, amort2, atm06, geo. mean

Non-decimal benchmarks: Performance comparisons with the re-compilation.
Performance Comparisons with the Re-compilation

- The baseline (100%) is the performance of the original binaries.
  - Original binaries are compiled with the highest optimization-level of the old compiler.

Can get even closer by:
- Enabling loop optimizations.
- Extend range of analysis of runtime routines.

Decimal benchmarks

Non-decimal benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Relative Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>cevalnd</td>
<td>closest to baseline</td>
</tr>
<tr>
<td>amort6</td>
<td>close to baseline</td>
</tr>
<tr>
<td>djpcom85</td>
<td>close to baseline</td>
</tr>
<tr>
<td>seqpr</td>
<td>close to baseline</td>
</tr>
<tr>
<td>ixseq</td>
<td>close to baseline</td>
</tr>
<tr>
<td>telco</td>
<td>close to baseline</td>
</tr>
<tr>
<td>invupd</td>
<td>close to baseline</td>
</tr>
<tr>
<td>amort2</td>
<td>close to baseline</td>
</tr>
<tr>
<td>atm06</td>
<td>close to baseline</td>
</tr>
<tr>
<td>geo. mean</td>
<td>close to baseline</td>
</tr>
</tbody>
</table>
Outline

- Motivation & Goal.
- Our Approach.
  - Four Major Optimizations.
- Performance Evaluation.
- Conclusions.
Conclusion

Language-specific binary optimizer can achieve competitive performance relative to the state-of-the-art source compiler.

Achieved +40.1% by our binary optimizer vs. +55.2% by the latest source compiler.
Questions?