Generating Low-Overhead Dynamic Binary Translators

Mathias Payer and Thomas R. Gross
Department of Computer Science
ETH Zürich
Motivation

- Binary Translation (BT) well known technique for “late” transformations
  - Extend or add features on the fly

- Flexibility of dynamic software BT incurs runtime overhead

- Complexity of transformations can be a challenge
  - Offer a high-level interface at compile time, compile into effective translation tables
Outline

- Introduction
- Design and Implementation
  - Table generation
  - Translator
- Optimization
- Conclusion
Binary Translation in a Nutshell

What about:
- Self modifying code?
- Shared libraries?
- Obfuscated Code?
Binary Translation in a Nutshell

Features:
- Translates all executed code
- Captures all indirect control flow transfers
- Just in time translation
Binary Translation in a Nutshell

Original program

Translator

Code cache

Table generator
supplies generated
opcode tables
at compile time

0
1
2
3
4

1' 2' 3'
Binary Translation in a Nutshell

Original program

Translator

Code cache

Translate

Mapping

0

1

2

3

4

3    3'

1    1'

2    2'

Trampoline to translate 4
fastBT

- Prototype for a dynamic BT system
- Machine-independent, OS-independent
  - Focus of this talk: IA32, Linux
Table Generation

- Translation tables describe individual instructions and are used to select the correct adapter functions

- Manual table construction is hard & cumbersome
  - Many instructions, write machine-code tables by hand

- Use automation and high level description!
  - Information about opcodes, possible encodings, and properties
  - Specify default translation actions

![Diagram]

- Intel IA32 opcode tables
- Table generator
  - High level interface
  - Adapter functions
- Optimized translator table
Table Generation

- Use table generator to offer high-level interface
  - Transforming opcode tables into runtime translation tables
  - Add analysis functions to control the table generation
    - Memory access?
    - What are src, dst, aux parameters?
    - FPU usage?
    - What kind of opcode?
    - What opcode class (load, store, arithmetic, control flow, ...)?
    - Immediate value as pointer?
    - etc.
Translator implementation

- Translator uses an iterator based approach and per-instruction actions

- Fundamentals to master low overhead:
  - Code cache
  - Inlining
  - Master (indirect) control transfers
Optimization

- Indirect control flow transfers are expensive
  - Runtime lookup and patching required
  - Indirect control transfer replaced by software trap

- Optimizations in fastBT:
  - Local branch prediction
  - Inlining a fast lookup into the code cache
  - Building on-the-fly shadow jump tables
Optimization: Branch prediction

- Cache the last one or two targets
- If there is a cache hit
  - No lookup is needed
  - Results in 3 to 5 instructions
- If there is a cache miss
  - Lookup the target and cache it for future use
  - Updating the cache costs additional instructions
Optimization: Fast lookup

- Emit an inlined fast lookup into the code cache
  - Uses the mapping table to translate the target
  - Optimized for direct hit in the mapping table
  - Results in 13 or 14 instructions
**Optimization: Shadow jump table**

- Build a shadow jump table, iff the original indirect control transfer uses a jump table
  - Initialize all entries with catch-all function
  - Lazy lookup and write-back in catch-all
  - Results in 5 instructions if the target is translated
Optimization: Problem

- Each optimization is only effective for some program locations and a specific program behavior
  - Low number of targets, few changes
    - Use a cache
  - High number of targets, many changes
    - Use fast lookup
  - Location has many different targets, all close to each other
    - Use a shadow jump-table

- An adaptive runtime optimization can select the best optimization for each indirect control transfer
Adaptive Optimization

- fastBT offers an adaptive optimization for indirect control transfers
  - Start with a prediction for 1 or 2 locations, count misses
  - Recover to a fast lookup, if count exceeds threshold
  - Construct a shadow jump-table, if the control transfer uses a jump table

- Adaptive optimizations bring competitive performance!
Benchmarks: Setup

- Used null-transformation to show translation overhead

- Used SPEC CPU2006 benchmarks to evaluate performance
  - We use the Test dataset for short running programs and the Ref dataset for long running programs

- Machine: E6850 Intel Core2Duo @ 3.00GHz
Related work

- **HDTrans**
  - S. Sridhar et al. HDTrans: a low-overhead dynamic translator. SIGARCH’07
  - Table based dynamic BT, no high level interface

- **DynamoRIO**
  - IR based optimizing BT, does not export a translation interface

- **PIN**
  - C.-K. Luk et al. Pin: building customized program analysis tools with dynamic instrumentation. In PLDI’05
  - High overhead, offers high level interface
Benchmarks: Ref dataset

- **400.perlbench**: 126%
- **445.gobmk**: 90%
- **483.xalancbmk**: 70%
- **447.dealII**: 50%
- **Average**: 40%

Categories:
- fastBT
- HDTrans
- PIN
- dynamoRIO
## Benchmarks: Ref dataset

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Function calls</th>
<th>inlined</th>
<th>Indirect jumps</th>
<th>jmptbl</th>
<th>pred</th>
<th>Indirect calls</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>25'814</td>
<td>8.1%</td>
<td>21'930</td>
<td>93.7%</td>
<td>6.3%</td>
<td>3'903</td>
<td>7.4%</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>18'001</td>
<td>1.3%</td>
<td>93</td>
<td>1.0%</td>
<td>99.0%</td>
<td>185</td>
<td>4.1%</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>28'888</td>
<td>10.6%</td>
<td>2'627</td>
<td>27.0%</td>
<td>63.6%</td>
<td>9'161</td>
<td>96.1%</td>
</tr>
<tr>
<td>447.dealII</td>
<td>52'756</td>
<td>54.5%</td>
<td>21'147</td>
<td>1.7%</td>
<td>98.3%</td>
<td>540</td>
<td>98.4%</td>
</tr>
</tbody>
</table>

1) All numbers are $\times 10^6$
## Benchmarks: Ref vs. Test Dataset

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Ref dataset</th>
<th>Test dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no BT [s]</td>
<td>fastBT</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>486</td>
<td>56%</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>611</td>
<td>18%</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>371</td>
<td>24%</td>
</tr>
<tr>
<td>447.dealII</td>
<td>552</td>
<td>44%</td>
</tr>
<tr>
<td>Average</td>
<td>839</td>
<td>6%</td>
</tr>
</tbody>
</table>
Benchmarks: Summary

- High overhead:
  - Many indirect control transfers
    - Function calls incur high overhead, even with optimizations
    - Indirect control transfers without caches or jump tables add overhead
  - High collision rate in mapping table
    - Expensive recoveries, try different rescheduling strategies

- Low overhead:
  - Few indirect control transfers
  - Cost of indirect control transfers is reduced through optimizations
Conclusion

- fastBT shows that it is possible to combine ease of use with efficient binary translation

- Adaptive optimizations select best optimization for individual locations

- Adaptive optimizations are necessary for low overhead in table based binary translators
Thanks for your attention!

- *fastBT* project page: [http://nebelwelt.net/fastBT](http://nebelwelt.net/fastBT)
- Contact: mathias.payer@inf.ethz.ch
- Kudos to:
  - Marcel Wirth, Peter Suter, Stephan Classen, and Antonio Barresi for code contributions
  - My colleagues for endless comments and reviews
Table Generation: Analysis Function

```cpp
bool isMemOp (const unsigned char* opcode,
const instr& disInf, std::string& action)
{
    bool res;

    /* check for memory access in instr. */
    res = mayOpAccessMem(disInf.dstFlags);
    res |= mayOpAccessMem(disInf.srcFlags);
    res |= mayOpAccessMem(disInf.auxFlags);

    /* change the default action */
    if (res) { action = "handleMemOp"; }

    return res;
}

// in main function:
addAnalysFunction(isMemOp);
```
Optimization: Efficient Code

- Static ind. call: `call *(fixed_location)`

1. Push original src IP

2. Compare actual target w/ cached target & branch if prediction ok

3. Recover if there is a misprediction
Optimization: Efficient Code

- Dynamic ind. call: \texttt{call *(reg)}

\begin{verbatim}
pushl src_addr
jmp *(reg)
\end{verbatim}

\begin{verbatim}
pushl src_addr, *(reg), %ebx, %ecx
movl 12(%esp), %ebx          # load target
movl %ebx, %ecx              # duplicate ip
andl HASH_PATTERN, %ebx     # hash fct
cmpl hashtlb(0, %ebx, 8), %ecx  # check
jne nohit
movl hashtlb+4(0, %ebx, 8), %ebx  # load trgt
movl %ebx, (tld->ind_jmp_targt)
popl %ecx, %ebx              # epilogue
leal 4(%esp), %esp           # readjust stack
jmp *(tld->ind_jmp_targt)    # jmp to trans.trgt

nohit: use ind_jump to recover
\end{verbatim}