An Infrastructure for Adaptive Dynamic Optimization

Derek Bruening, Timothy Garnett, and Saman Amarasinghe
Laboratory for Computer Science, Massachusetts Institute of Technology

Brian Connolly and John Gay
Dynamic Optimization

- Static optimization infrastructures are the most common
  - Limited by predicting program behavior at runtime
  - Optimized binaries are hard to debug
- Dynamic optimization is much less common
- Dynamic optimization involves optimizing during runtime
  - Does not rely on how code was compiled
  - Requires instant responses to changes in program behavior
DynamoRIO and API

- DynamoRIO is a dynamic code modification system
- API for building external modules (clients)
  - Abstracts lower-level DynamoRIO details away
  - Uses linear streams of code and adaptive representations of instructions
- Example optimizations implemented using API
  - Up to 40% speedup relative to base DynamoRIO
  - Average 12% speedup
- Dynamic optimization suffers from overhead of performing the optimization
Motivation/Problem

- Very few dynamic optimization frameworks at this time
  - Strata, DELI, Dynamo
- Static optimization is limited
  - No runtime adjustments for code block frequency
  - Optimized binaries are hard to debug
- Dynamic optimization offers many benefits
  - Can improve performance of binaries despite how they are compiled
  - Allows for adaptive, architecture-specific, and inter-module optimizations
  - Adaptive optimizations require instant responses
Related Work

● **Strata**
  ○ Client interface to build custom dynamic code modification tools
  ○ No support for re-optimizing fragments

● **DELI**
  ○ Hooks for transforming traces as they are built
  ○ No mechanism for re-optimizing traces, single level of detail

● **Other API-less frameworks**
  ○ Dynamo, Wiggins/Redstone, Mojo

● Essentially, other frameworks either lack an API or don’t support re-optimizing traces after being placed in cache
Key Contributions

- API for DynamoRIO to provide framework for implementing dynamic analyses and optimizations
- Interface for building external modules - called clients
  - 4 example clients built to demonstrate API
- Achieves efficiency through linear streams of code and adaptive levels of detail
DynamoRIO Key Features

- Basic block cache
  - Avoids overhead of context switch to DynamoRIO

- Trace cache
  - Trace - combines basic blocks that are frequently used together
  - Checks to ensure that block target is same as trace

- Indirect branch lookup
  - Hashtable of code cache addresses
  - Allows for quick lookup of targets in indirect branches
  - If target changes, full address must be looked up in hashtable
Control Flow Graph

Bruening, Garnett, Amarasinghe (March 2003). "An Infrastructure for Adaptive dynamic Optimization"
## Performance of Features

<table>
<thead>
<tr>
<th>System Type</th>
<th>Normalized Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>crafty</td>
</tr>
<tr>
<td>Emulation</td>
<td>~300.0</td>
</tr>
<tr>
<td>+ Basic block cache</td>
<td>26.1</td>
</tr>
<tr>
<td>+ Link direct branches</td>
<td>5.1</td>
</tr>
<tr>
<td>+ Link indirect branches</td>
<td>2.0</td>
</tr>
<tr>
<td>+ Traces</td>
<td>1.7</td>
</tr>
</tbody>
</table>

Bruening, Garnett, Amarasinghe (March 2003). "An Infrastructure for Adaptive dynamic Optimization"
### Instruction Representation 0-2

#### Level 0
- Raw Instruction Bytes

```
8d 34 01 8b 46 0c 2b 46 1c 0f b7 4e 08 cl e1 07 3b cl 0f 8d a2 0a 00 00
```

#### Level 1
- Instructions split for each machine instruction, but still stored in raw bits

<table>
<thead>
<tr>
<th>Raw Bits</th>
<th>Opcode</th>
<th>Eflags</th>
</tr>
</thead>
<tbody>
<tr>
<td>8d 34 01</td>
<td>lea</td>
<td>-</td>
</tr>
<tr>
<td>8b 46 0c</td>
<td>mov</td>
<td>-</td>
</tr>
<tr>
<td>2b 46 1c</td>
<td>sub</td>
<td>WCPASSO</td>
</tr>
<tr>
<td>0f b7 4e 08</td>
<td>movzx</td>
<td>-</td>
</tr>
<tr>
<td>cl e1 07</td>
<td>shl</td>
<td>WCPASSO</td>
</tr>
<tr>
<td>3b cl</td>
<td>cmp</td>
<td>WCPASSO</td>
</tr>
<tr>
<td>0f 8d a2 0a 00 00</td>
<td>jnl</td>
<td>RSO</td>
</tr>
</tbody>
</table>

#### Level 2
- Decoded to show opcode and eflags register

- Level 0: raw instruction bytes
- Level 1: Instructions split for each machine instruction, but still stored in raw bits
- Level 2: Decoded to show opcode and eflags register

Bruening, Garnett, Amarasignhe (March 2003). "An Infrastructure for Adaptive dynamic Optimization"
Instruction Representation 3-4

- **Level 3**: Fully-decoded, stores opcode, prefixes, and eflags, as well as operand arrays for sources and destinations
- **Level 4**: Fully-decoded instruction that has been modified or newly created, with no valid copy of raw bits
  - Must be re-encoded to obtain machine representation

Bruening, Garnett, Amarasignhe (March 2003). "An Infrastructure for Adaptive dynamic Optimization"
Adaptive Optimization

- Instructions typically stored in low-level for efficiency
  - Problematic because instructions vary greatly in complexity, require significant overhead to decode
- Clients can re-optimize code after it is placed in the cache
- This is done by re-creating an instruction list and replacing the old version in the cache with it
- Can perform this replacement while execution is still in the old fragment
  - Allows a trace to generate a new version of itself
Experiments and Test Cases

- **Redundant Load Removal**
  - Checks registers for variable, preventing redundant loads within basic blocks
  - gcc highest optimization level contains redundant loads

- **Strength Reduction (inc2add)**
  - Pentium 4, replace `inc` with `add 1` (`dec` and `sub 1` also)
  - Simply walk through basic blocks looking for instructions
  - Good example to show why architecture-specific optimizations should be done dynamically
Experiments and Test Cases

- **Indirect Branch Dispatch**
  - Hashtable lookup if branch isn’t inlined target of trace
  - Diverts control from lookup to hot_targets
  - Records target of each indirect branch

Experiments and Test Cases

- Custom Traces
  - DynamoRIO traces often end with return in another trace
  - More hash lookups
  - Essentially tries to compress functions and loop procedures into single code blocks
Experiment Results

- Floating Point benefitted more than Integer benchmarks
- FP -- 12% improvement over native
- FP + Int -- performance matches native, 12% increase over base DynamoRIO

Bruening, Garnett, Amarasinghe (March 2003). "An Infrastructure for Adaptive dynamic Optimization"
Experiment Results

- Why some major slowdowns?
  - DynamoRIO translates indirect branches to indirect jumps, costly indirect branch mispredictions
  - Pentium have return address predictors, but not indirect jump predictors
  - Also short code runs with little code reuse
Critique

- Today dynamic optimization frameworks are used for program analysis
  - In this case, DynamoRIO’s sole purpose was optimization
- Questionable results
  - Very few of the clients provide an improvement in execution time
  - Underlying features of the framework cause performance issues
- Further overhead reduction (Indirect branch dispatch table)
Follow-ons

- DynamoRIO released in June 2002, paper published April 2003
- Developers at MIT started Determina, offering Memory Firewall around the concept of program shepherding
  - Monitors instruction origin and control flow
  - Acquired by VMWare 2007
Follow-ons

- Dr. Memory is built upon DynamoRIO
  - Along with a number of other analysis tools
- Valgrind is another tool with some functionality similar to Dr. Memory
- Pin - a dynamic binary instrumentation framework
  - Enables the creation of dynamic program analysis tools
Thanks!