An Infrastructure for Adaptive Dynamic Optimization

Derek Bruening, Timothy Garnett, And Saman Amarasinghe
Laboratory For Computer Science
Massachusetts Institute of Technology

Presented by Glenn Smith, Max Thomas
DynamoRIO, Summarized

- Presents a framework for performing dynamic optimizations
- Changes (compiled) program behaviour at runtime
- Minimal overhead, but slower than native
  - The optimizations presented barely offset the framework overhead
Motivation

● Static optimization is not effective on “modern” programs
● Need new approaches
● Dynamic optimization seems promising, but…
  ○ It’s difficult, and few frameworks exist
Why Should We Care?

- We can’t check many things at compile time
- Sometimes the compiler misses things
- We’d like to be able to make changes at runtime…
  - But this requires a dynamic optimization engine
- Lets us do dynamic instrumentation
### Related Work (a.k.a. Other Frameworks From 2003)

<table>
<thead>
<tr>
<th>Framework</th>
<th>Target Platform(s)</th>
<th>Has API?</th>
<th>Re-optimization?</th>
<th>User-defined traces?</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELI</td>
<td>x86</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Strata</td>
<td>x86</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamo</td>
<td>PA-RISC</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Wiggins/Redstone</td>
<td>Alpha</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Mojo</td>
<td>x86 (Windows)</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Kistler</td>
<td>?</td>
<td>No</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>DynamoRIO</td>
<td>x86</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
What’s a DynamoRIO Anyways?

- Dynamic instrumentation/modification framework
- Built on top of Dynamo by the Runtime Introspection and Optimization team @ MIT
- Designed to run on IA-32 (a.k.a. x86)
  - Which makes things… *complicated.*
Intro to DynamoRIO

- DynamoRIO runs in the same memory space as host process
- Finds all basic blocks, then moves them into a code cache
- Direct jumps are overwritten to point to the cache
  - While indirect jumps redirect to a hash table lookup
- Blocks that are often called in sequence are linked together into a trace
  - Traces are treated (mostly) like basic blocks
- Performs optimizations on cache before execution
  - And also maybe at runtime, more on this later
Intro to the DynamoRIO Client API

- Clients implement functions that are called by DynamoRIO to insert instrumentation and make optimizations
  - Extensions allow adaptive optimization & custom traces

<table>
<thead>
<tr>
<th>Client Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void dynamorio_init()</td>
<td>Client initialization</td>
</tr>
<tr>
<td>void dynamorio_exit()</td>
<td>Client finalization</td>
</tr>
<tr>
<td>void dynamorio_thread_init(void *context)</td>
<td>Client per-thread initialization</td>
</tr>
<tr>
<td>void dynamorio_thread_exit(void *context)</td>
<td>Client per-thread finalization</td>
</tr>
<tr>
<td>void dynamorio_basic_block(void *context, app_pc tag, InstrList *bb)</td>
<td>Client processing of basic block</td>
</tr>
<tr>
<td>void dynamorio_trace(void *context, app_pc tag, InstrList *trace)</td>
<td>Client processing of trace</td>
</tr>
<tr>
<td>void dynamorio_fragment_deleted(void *context, app_pc tag)</td>
<td>Notifies client when a fragment is deleted from the code cache</td>
</tr>
<tr>
<td>int dynamorio_end_trace(void *context, app_pc trace_tag, app_pc next_tag)</td>
<td>Asks client whether to end the current trace</td>
</tr>
</tbody>
</table>
Intro to the DynamoRIO Client API

- Instructions in x86 are messy
  - Can be variable length, and are always non-trivial to decode
- Decoding every instruction in a block is really slow
- Encoding instructions is worse
- DynamoRIO gets around this by decoding (blocks of) instructions in levels
x86 Decoding: Level 0

8d 34 01 8b 46 0c 2b 46 1c 0f b7 4e 08 c1 e1 07 3b c1 0f 8d a2 0a 00 00
x86 Decoding: Level 1

8d 34 01
8b 46 0c
2b 46 1c
0f b7 4e 08
c1 e1 07
3b c1
0f 8d a2 0a 00 00
x86 Decoding: Level 2

8d 34 01      lea
8b 46 0c      mov
2b 46 1c      sub       eflags |= WCPAZSO
0f b7 4e 08   movzx
C1 E1 07      shl       eflags |= WCPAZSO
3b C1         cmp       eflags |= WCPAZSO
0f 8d a2 0a 00 00 jnl       eflags |= RSO
x86 Decoding: Level 3

8d 34 01  lea  (%ecx,%eax,1) -> %esi
8b 46 0c  mov  0xc(%esi) -> %eax
2b 46 1c  sub  0x1c(%esi) %eax -> %eax   (eflags)
0f b7 4e 08 movzx 0x8(%esi) -> %ecx
C1 e1 07  shl  $0x07 %ecx -> %ecx       (eflags)
3b c1    cmp  %eax %ecx                  (eflags)
0f 8d a2 0a 00 00 jnl  $0x77f52269      (eflags)
x86 Decoding: Level 4

???? lea (%ecx,%eax,1) -> %edi
???? mov 0xc(%edi) -> %eax
???? sub 0x1c(%edi) %eax -> %eax (eflags)
???? movzx 0x8(%edi) -> %ecx
3b c1 shl $0x07 %ecx -> %ecx (eflags)
0f 8d a2 0a 00 00 jnl $0x77f52269 (eflags)
Dynamic Optimization?

- Maybe your compiler can’t be trusted to optimize
- Maybe your CPU is weird and different
- Maybe you dynamically loaded some library
- (actually, all of these are true)
DynamoRIO Example Optimizations

- The Pentium 4 is bad at incrementing
  - Somehow `inc %eax` is slower than `add $1 %eax -> %eax`
- Compilers don’t know what CPU you have at compile-time
- Solution: Use DynamoRIO to replace those instructions
- Result: Small performance increase
DynamoRIO Example Optimizations

- When we branch, we jump to profiler first
- But indirect branches means we don’t know where
- Looking up branches at runtime is slow!
- Insert list of “hot” branch targets into assembly to mitigate worst offenders
- Result: Better than base profiling...

```assembly
call prof_routine
jmp hashtable_lookup

cmp real_target, hot_target_1
je hot_target_1
cmp real_target, hot_target_2
je hot_target_2
call prof_routine
jmp hashtable_lookup
```

Figure 4. Code transformation by our indirect branch dispatch optimization. A profiling routine rewrites its own trace to insert dispatches for the hottest targets among its samples, avoiding a hashtable lookup.

Diagram: Derek Bruening et. al. (March 2003). "An Infrastructure for Adaptive Dynamic Optimization".
DynamoRIO Example Optimizations

- Redundant Load Removal
- Value already in register, why access the stack?
- Works across basic blocks in the same trace
- GCC was pretty bad at this with floating-point instructions
- Result: Significant increase in some benchmarks
Optimization Successes

- Redundant Load Removal had some success
- Some floating-point benchmarks improved significantly

Diagram: Derek Bruening et. al. (March 2003). “An Infrastructure for Adaptive Dynamic Optimization”.
Optimization Failures

- Short runs of code with “little code-reuse”
- Sometimes even slower than base DynamoRIO
- Dynamic instrumentation is slow!

Diagram: Derek Bruening et. al. (March 2003). "An Infrastructure for Adaptive Dynamic Optimization".
Optimization Breakevens

- Those last few graphs were cherry-picked
- Most of the time they barely recouped init cost

Diagram: Derek Bruening et. al. (March 2003). "An Infrastructure for Adaptive Dynamic Optimization".
Why is Dynamic Optimization So Hard?

- Time is lost setting up framework
- Branch mispredictions are much more expensive
- Pentium 4 had prediction for return-calls, but not jumps!
  - DynamoRIO replaces returns with jumps
- Compilers are already pretty good at their job
Critique

● Paper misses many interesting applications of DynamoRIO
● Can be used for instrumentation, dynamic analysis
  ○ Security research, analysis, profiling
  ○ But paper glosses this over for dynamic optimization
Critique

- Closed source!
- Stymies additional research
- Has since been released under the BSD license
Critique

- Optimizations presented barely break even
  - Instrumentation itself is slow, uphill battle
  - Works (somewhat) for floating-point code
  - Less improvement for integer programs

Diagram: Derek Bruening et. al. (March 2003). “An Infrastructure for Adaptive Dynamic Optimization”. 
Later Advancements

● DrMemory!
  ○ And friends (drcache, drcpusim, etc.)
● Valgrind
  ○ Most commonly known for memcheck - DrMemory competitor
● Pin
  ○ Intel’s “Architecture Agnostic” framework
● Most former optimization frameworks are now instrumentation / analysis frameworks
  ○ Optimization is hard
Side Note: We’ve used it!

- “MemViz: A Memory and Cache Performance Visualizer”
- Our Final Project in Interactive Visualization ;)
- Explored using instruction tracing and recording memory uses at runtime
- Not an optimization (the opposite)
Questions?