A Cross-Architectural Interface for Code Cache Manipulation

Kim Hazelwood and Robert Cohn
Software-Managed Code Caches

- Software-managed code caches store transformed code at run time to amortize overhead of dynamic optimizers.
- Contain a (potentially altered) copy of application code.
Code Cache Contents

Every application instruction executed is stored in the code cache (at least)

Code Regions

- Altered copies of application code
- Basic blocks and/or traces

Exit stubs

- Swap application ↔ VM state
- Return control to the dynamic optimizer
**Code Regions**

**Basic Blocks**

BBL A: Inst1 Inst2 Inst3 Branch B

**Traces**

CFG  In Memory  Trace
Exit Stubs

One exit stub exists for every exit from every trace or basic block

Functionality

Prepare for context switch
Return control to VM dispatch

Details

Each exit stub ≈ 3 instructions
A Goal of the Code Cache: Transparency

Pretend as though the original program is executing

Original Code:
0x1000 call 0x4000

Translated Code:
0x7000 push 0x1006
0x7006 jmp 0x8000

Push 0x1006 on stack, then jump to 0x4000

Code cache address mapping:
0x1000 → 0x7000 "caller"
0x4000 → 0x8000 "callee"
A Challenge and an Opportunity

Challenges

• Code caches hold the key to overall performance
  ➢ Self-modifying code
  ➢ Unloaded libraries
  ➢ Bounded sizes

Opportunities

• Ephemeral instrumentation
• Adaptive optimizations
• Security
Interesting Research Problems, but...

- Most systems hide all evidence of code caches
- Investigations have required source code access
- Code cache implementations are often tightly coupled to the rest of the system in subtle ways

Direct code cache access can be a powerful opportunity!
The Code Cache API

• We provide a clean, robust interface for accessing and altering code cache behavior and contents
  ➢ ATOM-style interface
  ➢ Rapid prototyping

• Users can
  ➢ Investigate code cache design decisions
  ➢ Investigate applications of binary modifiers

• Built upon the Pin dynamic instrumentation system
Building upon Pin

- A dynamic instrumentation system from Intel
- Multiple platform support
  - Four ISAs – IA32, EM64T, IPF, ARM
  - Four OSes – Linux, Windows, FreeBSD, MacOS
- Robust and stable (Pin can run itself!)
  - 12+ active developers
  - Nightly testing of 12 configurations on 25000 binaries
  - Automatic generation of user manuals
  - Large user base in academia and industry
  - Pinheads mailing list
- Seamless interaction with instrumentation interface
Pin’s Code Cache

Cache Block 1
Cache Block 2
Cache Block J

Next trace
Next stub
Cache Linking

- Trace #1
- Trace #2
- Trace #3
- Exit #1a
- Exit #1b
- Dispatch
Cache Client Interface

- Pin
- Address Space
- Cache Client APIs
- Virtual Machine (VM)
  - JIT Compiler
  - Emulation Unit
- Dispatcher
- Code Cache
- Operating System
- Hardware
Code Cache API

**Callbacks** – Events that trigger calls to user functions
- CacheIsFull, TraceInserted, EnteringCache, TraceLinked, BlockCreated, ...

**Actions** – Events a user can invoke via instrumentation
- ChangeCacheSize(Sz), FlushCache(), FlushBlock(Id), InvalidateTrace(SPC)...

**Lookups** – Returns IDs, mappings, handles for traces, exit stubs, cache blocks, ...

**Statistics** – Live summary of cache contents
- MemoryUsed, FlushCount, TracesInCache, CacheBlockCount, ...
Our Design Goals

• Ease of use
• Comparable performance to a direct implementation

Major instrumentation overhead source

• “State switch” between executing application and instrumentation code

Fundamental difference

• Nearly all callbacks occur from the VM (no state switch)
• All others would incur the same state switch overhead in a direct implementation...
Overhead of Empty Routines

![Bar chart showing relative performance of different routines with various overheads.

- PinNative
- All Callbacks
- Flush Only
- Cache Enter
- Trace Link
- Trace Insert

The chart compares the relative performance of various routines under different overhead conditions. The x-axis represents the routines (bzip2, crafty, eon, gap, gcc, gzip, mcf, parser, perl, twolf, vortex, vpr), and the y-axis shows the relative performance percentage varying from 0% to 500%. The chart highlights the impact of overhead on performance across different scenarios.]
**Code Cache API Utility**

**Code Cache Design**
- Cache replacement investigations
- Graphical visualization
- Architectural comparisons
  - IA-32, EM64T, Itanium, XScale

**Code Cache Applications**
- Optimization algorithms
- Security algorithms
Cache Replacement

```c
void main(int argc, char **argv) {
    PIN_Init(argc, argv);
    CODECACHE_CacheIsFull(FlushOnFull);
    PIN_StartProgram(); // Never returns
}
void FlushOnFull() {
    CODECACHE_FlushCache();
    cout << "SWOOSH!" << endl;
}
```

Eviction Granularities
- Entire Cache
- One Cache Block
- One Trace
- Address Range

```bash
% pin -cache_size 40960 -t flusher -- /bin/ls
SWOOSH!
SWOOSH!
SWOOSH!
<output of /bin/ls>
```
A Graphical Front-End

Hazelwood and Cohn, CGO 2006
### Design Challenge: ISA Idiosyncrasies

<table>
<thead>
<tr>
<th></th>
<th>ARM</th>
<th>IA-32/EM64T</th>
<th>IPF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
<td>RISC</td>
<td>CISC</td>
<td>VLIW</td>
</tr>
<tr>
<td><strong>Instruction</strong></td>
<td>Fixed length</td>
<td>Variable length, prefixes</td>
<td>Bundled</td>
</tr>
<tr>
<td><strong>Memory Instruction</strong></td>
<td>LD/ST</td>
<td>Any, implicit</td>
<td>LD/ST</td>
</tr>
<tr>
<td><strong>Memory op size</strong></td>
<td>Fixed</td>
<td>Variable length</td>
<td>Fixed</td>
</tr>
<tr>
<td><strong>Addressing modes</strong></td>
<td>Pre/post/iprel increment</td>
<td>Index/offset/scale/iprel</td>
<td>Post</td>
</tr>
<tr>
<td><strong>Predication</strong></td>
<td>Cond. codes</td>
<td>None</td>
<td>Predicate regs</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>Registers</td>
<td>Stack/registers</td>
<td>Stacked registers</td>
</tr>
</tbody>
</table>
Architectural Comparisons

<table>
<thead>
<tr>
<th></th>
<th>IA32</th>
<th>EM64T</th>
<th>IPF</th>
<th>Xscale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Size (Bytes)</td>
<td>1.0</td>
<td>2.6</td>
<td>3.8</td>
<td></td>
</tr>
<tr>
<td>Traces (Count)</td>
<td>1.0</td>
<td>1.9</td>
<td>1.0</td>
<td>0.7</td>
</tr>
<tr>
<td>Exit Stubs (Count)</td>
<td>1.0</td>
<td>2.2</td>
<td>2.2</td>
<td>0.5</td>
</tr>
<tr>
<td>Links (Count)</td>
<td>1.0</td>
<td>1.8</td>
<td>0.9</td>
<td>0.6</td>
</tr>
</tbody>
</table>
Architectural Comparisons (2)

<table>
<thead>
<tr>
<th></th>
<th>IA32</th>
<th>EM64T</th>
<th>IPF</th>
<th>Xscale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>17.2</td>
<td>19.5</td>
<td>58.1</td>
<td></td>
</tr>
<tr>
<td>Basic Blocks</td>
<td>3.9</td>
<td>4.5</td>
<td>6.4</td>
<td></td>
</tr>
<tr>
<td>Links</td>
<td>2.3</td>
<td>2.1</td>
<td>2.5</td>
<td>1.8</td>
</tr>
</tbody>
</table>
Design Challenge: Self-Modifying Code

The problem

Code cache must detect SMC and invalidate corresponding cached traces

Solutions

Many proposed ... but source code is usually necessary to investigate solutions
Self-Modifying Code Handler

void main (int argc, char **argv) {
    PIN_Init(argc, argv);
    TRACE_AddInstrumentFunction(InsertSmcCheck,0);
    PIN_StartProgram(); // Never returns
}

void InsertSmcCheck () {
    ... (variable declarations) ...
    memcpy(traceCopyAddr, traceAddr, traceSize);
    TRACE_InsertCall(trace, IPOINT_BEFORE, (AFUNPTR)DoSmcCheck,
                     IARG_PTR, traceAddr, IARG_PTR, traceCopyAddr,
                     IARG_UINT32, traceSize, IARG_CONTEXT, IARG_END);
}

void DoSmcCheck (VOID* traceAddr, VOID *traceCopyAddr,
                 USIZE traceSize, CONTEXT* ctxP) {
    if (memcmp(traceAddr, traceCopyAddr, traceSize) != 0) {
        CODECACHE_InvalidateTrace((ADDRINT)traceAddr);
        PIN_ExecuteAt(ctxP);
    }
}

(Written by Alex Skaletsky)
Adaptive Code Optimizations

Many reasons to **selectively invalidate** cached traces

- Ephemeral profiling
- Phase-based optimizations
- Adaptive algorithms
Two-Phase Instrumentation

- Memory reference instrumentation can be costly
- Can invalidate instrumented code after $N$ executions
Plug-In Tools Shipped with our API

**CacheSimulator** – Exercises most of the API

**CacheFlusher** – Performs a full cache flush

**CacheDoubler** – Doubles cache size when full

**LinkUnlink** – Watches all link activity

**BBTest** – User-defined trace sizing

**TraceInvalidator** – Invalidates hot traces

**SMCHandler** – Stores a copy of program instructions and invalidates stale code

... and several more ...
Summary

- Low overhead but highly functional interface to Pin’s code cache
- Enables introspection as well as adjustment of cache policies and contents
- One API → four ISAs → four OSes
- Works seamlessly with Pin’s instrumentation API

- Download it today!

http://rogue.colorado.edu/pin