Chainsaw: Using Binary Matching for Relative Instruction Mix Comparison

Tipp Moseley
Dirk Grunwald
U. Of Colorado

Ramesh Peri
Intel
Primitive Development Cycle for Compilers

1. Come up with an idea
2. Test it
3. If nothing improved, go to 1
4. Celebrate
Understanding Transformations Is Hard

- Code size vs. control flow
  - Inlining, loop transformations, superblocks, if-conversion
- Architectural Features
  - Alignment, speculation, instruction selection, multilevel memory hierarchy, prefetching
Unpredictable Results

445.gobmk Jump Instruction Comparison

- countlib
- remove_liberty
- verify_stored_board
- assimilate_string
- update_liberties
- propose_edge_moves
- chainlinks2
- hashtable_clear
- incremental_order_moves
- fastlib

Normalized Count

intel 9.1
gcc 4.1.0
Motivating Example

```c
assert(x == 1 || x == 2);
return array[x];
```
…this case is insignificant in the compiler vendor’s regression suite?
…it is in the kernel of an important customer’s application?

We need better tools!
Chainsaw Approach

- Chainsaw offers high resolution relative performance comparison for arbitrarily compiled binaries (including IPO)
- Match a log of execution events from two differently compiled programs
  - Use BASE as reference point
- Compare profiles of matched intervals to identify outliers
Matching Resolution

- Program- and function-level
  - Too big; miss opportunities
- Instructions and basic blocks
  - Too small; automatic comparisons are too noisy
- Loops
  - Just right; corresponds well with source
  - Be conservative; match innermost loop that makes sense
• Compiler can perform very aggressive optimization
  • Unrolling, peeling, splitting, merging, multiple entrypoints, versioning, …
• Larger functions can have many loops that look identical
Log-based Event Matching

- **Event**: call, return, or backedge
- **Anchor**: a pair of events that
  - Occurs the same number of times in both logs
  - Is semantically similar in both logs (e.g., same symbol names)
  - Has the same respective ordering to all other anchors in both logs
- **Interval**: execution history between two anchored events

- Similar to sequence alignment, but much simpler
  - Don’t deal with insertions/deletions; only align tokens that occur same number of times
Log-based Event Matching

Compilers can sometimes order function calls arbitrarily, so semantics and count alone are insufficient

- \( x = f() + g(); \)
- LOG 0: CALL f, RET, CALL g, RET
- LOG 1: CALL g, RET, CALL f, RET
Two anchors have the same ordering signature (OrdSig) \textit{iff} they occur the same number of time and in the same order in both logs.

For each log, an ordering table (OrdTable) is filled with the OrdSig for every pair of events in the log.
Computing Ordering Signatures

Traverse the log and maintain a stack-like structure ordering events

1. If the event, \( i \), is not on the stack, run \( \text{Update}(i, j) \) for every \( j \) residing on the stack. Place \( i \) on the stack.

2. If the event, \( i \), is on the stack, run \( \text{Update}(i, j) \) for every event \( j \) that has been seen since the last time \( i \) was seen. Move \( i \) to the back of the stack.

3. When the end of an interval is reached, run \( \text{Update}(i, j) \) for every event pair where \( i \) has been seen more recently than \( j \) on the stack.
Computing Ordering Signatures

Update function:
Modify ordtable[i, j] and ordtable[j, i] by xor with event count, circular shift, and multiply with large prime seed.
Ordering Signatures Example

Log: ABCBBA

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>0x21</td>
<td>0x67</td>
</tr>
<tr>
<td>B</td>
<td>0x11</td>
<td>3</td>
<td>0x03</td>
</tr>
<tr>
<td>C</td>
<td>0xeb</td>
<td>0x75</td>
<td>1</td>
</tr>
</tbody>
</table>

Log: AXYBCXXXBYBYAX

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>0x21</td>
<td>0x67</td>
<td>0x9a</td>
<td>0x1b</td>
</tr>
<tr>
<td>B</td>
<td>0x11</td>
<td>3</td>
<td>0x03</td>
<td>0xd</td>
<td>0x9f</td>
</tr>
<tr>
<td>C</td>
<td>0xeb</td>
<td>0x75</td>
<td>1</td>
<td>0x81</td>
<td>0x8c</td>
</tr>
<tr>
<td>X</td>
<td>0xb2</td>
<td>0x86</td>
<td>0x63</td>
<td>4</td>
<td>0x47</td>
</tr>
<tr>
<td>Y</td>
<td>0x19</td>
<td>0xfe</td>
<td>0x88</td>
<td>0x5e</td>
<td>3</td>
</tr>
</tbody>
</table>
Ordering Signatures Example

Initial state:
Sequence: ABCBBA
stack = []

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Ordering Signatures Example

Step 1:
Sequence: AABCDE
Actions: OrdTable[A, A]++
stack = [A]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Ordering Signatures Example

Step 2:
Sequence: A\textcolor{red}{B}C\textcolor{red}{B}B
Actions: OrdTable[B, B]++, Update(B, A)
stack = [A, B]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0xa1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0xb7</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Ordering Signatures Example

Step 3:
Sequence: AB_CBBA
Actions: OrdTable[C, C]++, Update(C, B),
         Update(C, A)
stack = [A, B, C]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0xa1</td>
<td>0xa1</td>
</tr>
<tr>
<td>B</td>
<td>0xb7</td>
<td>1</td>
<td>0xa1</td>
</tr>
<tr>
<td>C</td>
<td>0xb7</td>
<td>0xb7</td>
<td>1</td>
</tr>
</tbody>
</table>
Step 4:
Sequence: ABCBA
Actions: OrdTable[B, B]++, Update(B, C)
stack = [A, C, B]
Ordering Signatures Example

Step 5:
Sequence: ABCBBA
Actions: OrdTable[B, B]++
stack = [A, C, B]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0xa1</td>
<td>0xa1</td>
</tr>
<tr>
<td>B</td>
<td>0xb7</td>
<td>3</td>
<td>0x5a</td>
</tr>
<tr>
<td>C</td>
<td>0xb7</td>
<td>0x13</td>
<td>1</td>
</tr>
</tbody>
</table>
Ordering Signatures Example

Step 6:
Sequence: ABCBBAA
Actions: OrdTable[A, A]++, Update(A, B)
Update(A, C)
stack = [C, B, A]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>0x12</td>
<td>0x4e</td>
</tr>
<tr>
<td>B</td>
<td>0x61</td>
<td>3</td>
<td>0x5a</td>
</tr>
<tr>
<td>C</td>
<td>0x9f</td>
<td>0x13</td>
<td>1</td>
</tr>
</tbody>
</table>
Ordering Signatures Example

End of interval:
Sequence: <>
Actions: Update(A, B), Update(A, C),
          Update(B, C)
stack = [C, B, A]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>0x21</td>
<td>0x67</td>
</tr>
<tr>
<td>B</td>
<td>0x11</td>
<td>3</td>
<td>0x03</td>
</tr>
<tr>
<td>C</td>
<td>0xeb</td>
<td>0x75</td>
<td>1</td>
</tr>
</tbody>
</table>
Selecting Matches

• Score potential anchors
  – Sort by number of common OrdSigs in each log
  – Event counts must be equal
  – Events must be semantically similar

• Select first match with highest score

• Subsequent matches must be legally ordered with respect to previous matches
Real-world Challenges

- Event logs are enormous and slow to collect
  - Two-pass algorithm
    - Lightweight log (LWT)
      - Only CALL, RET, and non-inner loop BACKEDGE
    - Heavyweight log (HWT) – inner loops too
- Comparing optimized (OPT) binaries leads to ambiguity
  - Compare each log to unoptimized (BASE) and infer mapping
Chainsaw Overview
Static Instructions per Interval

GCC 4.2.3 -O2 Matched With GCC 4.2.3 -O3

- 95% of intervals have less than 19 static instructions
Dynamic Instructions per Interval

GCC 4.2.3 -O2 Matched With GCC 4.2.3 -O3

- 95% of intervals have less than 64 dynamic instructions
Analysis Overhead

- Total overhead around 1600X
• SPEC2006 test suite takes about 1.2GB
<table>
<thead>
<tr>
<th>Function</th>
<th>Max</th>
<th>linux,em64t,gnu,4.2.3,shared,-O2-noinline</th>
<th>linux,em64t,intel,10.0.023,shared,-O2-noinline</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu2006/445.gobmk/train/7/hashtable_clear==40cb58</td>
<td>22M</td>
<td>0.00</td>
<td>100.00</td>
</tr>
<tr>
<td>cpu2006/464.h264ref/train/0/IntraChromaPrediction==...</td>
<td>1M</td>
<td>0.00</td>
<td>100.00</td>
</tr>
<tr>
<td>cpu2006/416.gamess/train/0/dmtx==9d7e14</td>
<td>51M</td>
<td>0.00</td>
<td>100.00</td>
</tr>
<tr>
<td>cpu2006/454.calculix/train/0/e_c3d==431cecb</td>
<td>95k</td>
<td>0.00</td>
<td>100.00</td>
</tr>
<tr>
<td>cpu2006/437.leslie3d/train/0/MAIN==4144c8</td>
<td>50</td>
<td>0.00</td>
<td>100.00</td>
</tr>
<tr>
<td>cpu2006/435.gromacs/train/0/update==4b06bb</td>
<td>69M</td>
<td>0.00</td>
<td>100.00</td>
</tr>
<tr>
<td>cpu2006/401.bzip2/train/2/mainSort==4024ac</td>
<td>45M</td>
<td>0.00</td>
<td>100.00</td>
</tr>
<tr>
<td>cpu2006/482.sphinx3/train/0/hmm_clear==40c5e4</td>
<td>2M</td>
<td>0.00</td>
<td>100.00</td>
</tr>
<tr>
<td>cpu2006/464.h264ref/train/0/IntraChromaPrediction==...</td>
<td>1M</td>
<td>0.00</td>
<td>100.00</td>
</tr>
<tr>
<td>cpu2006/445.gobmk/train/5/hashtable_clear==40cb7f</td>
<td>20M</td>
<td>0.00</td>
<td>100.00</td>
</tr>
<tr>
<td></td>
<td>linux em64t gnu 3.4.6 shared -O2-noinline</td>
<td>linux em64t gnu 4.0.4 shared -O2-noinline</td>
<td>linux em64t gnu 4.1.2 shared -O2-noinline</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------------------------</td>
<td>------------------------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>cpu2006/400.perlbench /train/0</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>cpu2006/400.perlbench /train/1</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>cpu2006/400.perlbench /train/2</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>cpu2006/400.perlbench /train/3</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>cpu2006/400.perlbench /train/4</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>cpu2006/401.bzip2 /train/0</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>cpu2006/401.bzip2 /train/1</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>cpu2006/401.bzip2 /train/2</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>
Current Query: 'dyn.agg.total'

Matched Functions

<table>
<thead>
<tr>
<th>MAX</th>
<th>MAX</th>
<th>MAX</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>12k</td>
<td>100.0</td>
<td>72.8</td>
</tr>
<tr>
<td>49</td>
<td>12k</td>
<td>100.0</td>
<td>75.1</td>
</tr>
<tr>
<td>300</td>
<td>10k</td>
<td>98.4</td>
<td>100.0</td>
</tr>
<tr>
<td>298</td>
<td>3k</td>
<td>100.0</td>
<td>95.3</td>
</tr>
<tr>
<td>299</td>
<td>3k</td>
<td>100.0</td>
<td>95.2</td>
</tr>
<tr>
<td>47</td>
<td>3k</td>
<td>100.0</td>
<td>46.9</td>
</tr>
<tr>
<td>500</td>
<td>3k</td>
<td>100.0</td>
<td>46.9</td>
</tr>
<tr>
<td>dyn_agg.total</td>
<td>2M</td>
<td>84.62</td>
<td>100.00</td>
</tr>
<tr>
<td>dyn_agg_stack_writes</td>
<td>344k</td>
<td>75.00</td>
<td>100.00</td>
</tr>
<tr>
<td>dyn_agg_mem_writes</td>
<td>344k</td>
<td>75.00</td>
<td>100.00</td>
</tr>
<tr>
<td>dyn_agg_mem_writes8</td>
<td>344k</td>
<td>50.00</td>
<td>100.00</td>
</tr>
<tr>
<td>dyn_agg_stack_reads</td>
<td>172k</td>
<td>50.00</td>
<td>100.00</td>
</tr>
<tr>
<td>dyn_agg_mem_reads</td>
<td>172k</td>
<td>50.00</td>
<td>100.00</td>
</tr>
<tr>
<td>dyn_agg_mem_reads8</td>
<td>172k</td>
<td>50.00</td>
<td>100.00</td>
</tr>
<tr>
<td>dyn_agg_jmps</td>
<td>172k</td>
<td>50.00</td>
<td>100.00</td>
</tr>
<tr>
<td>dyn_agg_mem_writes4</td>
<td>86k</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>dyn_agg_indir_jmps</td>
<td>86k</td>
<td>100.00</td>
<td>100.00</td>
</tr>
</tbody>
</table>
Case Studies

• ISAs
  – IA-32, EM64T, IA-64

• Compilers
  – GCC 3.2.3, 3.4.6, 4.0.4, 4.1.2, 4.2.4, 4.3.0
  – ICC 9.1, 10.0, 11.0
  – LLVM 2.3 (IA-32 only)

• Machines
  – 3.0 GHz Core 2 Xeon
  – 900 MHz Itanium 2 McKinley
// Source: $O(N)$
int gx;
void Tailcall(int c) {
    if (c > 0) {
        gx += 100;
        Tailcall(c - 1);
    }
}

// ICC: $O(N)$
while (c > 0) {
    gx += 100;
    c--;
}

// GCC: $O(1)$
if (c > 0) {
    gx += c * 100;
}
Cross-ISA Comparison

\[
\text{do } i=1,132 \\
\text{text}(i:i)=' ' \\
\]

(a) IA-32 GCC 4.2.3 -O2

```
mov byte ptr ds[edi], 0x20
add edi, 0x1
cmp edi, edx
jnz 0x804e737
```

(b) EM64T GCC 4.3.0 -O2

```
lea eax, ptr [rbp-0x1]
mov edx, ebp
add ebp, 0x1
cmp edx, 0x84
mov dword ptr [rsp+0x71c], ebp
cdqe
mov byte ptr [rsp+rax*1+0x4e0], 0x20
jnz 0x40791f
```
## Data Alignment

<table>
<thead>
<tr>
<th>Metric</th>
<th>Max Count</th>
<th>ICS 10.0 –O2</th>
<th>ICS 10.0 –O3 –mtune</th>
</tr>
</thead>
<tbody>
<tr>
<td>dyn.agg.total</td>
<td>5B</td>
<td>100%</td>
<td>81.75%</td>
</tr>
<tr>
<td>dyn.agg.mem_reads</td>
<td>1B</td>
<td>100</td>
<td>84.09</td>
</tr>
<tr>
<td>dyn.agg.mem_writes</td>
<td>738M</td>
<td>100</td>
<td>94.74</td>
</tr>
<tr>
<td>dyn.cat.X87_ALU</td>
<td>4B</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>dyn.cat.DATAXFER</td>
<td>2B</td>
<td>20.54</td>
<td>100</td>
</tr>
<tr>
<td>dyn.cat.SSE</td>
<td>1B</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>dyn.cat.COND_BR</td>
<td>19M</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>dyn.pmu.MISALIGN_MEM_REF</td>
<td>347k</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

- With –O3 enabled, ICS doesn’t align array variable on the stack for loop from 444.namd – 87% slower
Performance Regression

```c
// 464.h264ref loop in image.c:1441
jj = max(0, min(s->size_y - 1, j))
```

- GCC 4.2.3 uses CMOV instruction
- GCC 4.3.0 uses branches
  - 90% slower,
  - 25X more cond. branches
for(XalanDOMString::size_type i = start; i < n; ++i) {
    accumCharUTF(chars[i]);
}

- ICPC inlines accumCharUTF, does not unroll
- LLVM unrolls loop 8 times
  - 8 calls to accumCharUTF
  - 5X more space, 2X more dyn. insts.,
    3X more mem reads, 7X more mem writes
Observations

• Finding performance bugs is easy!
  – “Big” anomalies take 1-2 hours of analysis
  – Smaller differences more abundant

• Compilers still do silly things
  – Direct jump to next instruction
  – Jump to return instruction
  – Spill all registers, then immediately refill
Summary

• Chainsaw: Infrastructure for relative profile comparison tools
  – Binary matching
  – Web-based profile viewer
Questions