CS184b:
Computer Architecture
[Single Threaded Architecture: abstractions, quantification, and optimizations]

Day15: February 27, 2000
Binary Translation

Today

• Problem
• Idea
• Complications
• Strategy
• ? Success?
Problem

• Lifetime of programs >> lifetime of piece of hardware (technology generation)
• Getting high performance out of old, binary code in hardware is expensive
  – superscalar overhead…
• Recompilation not viable
  – only ABI seems well enough defined; captures and encapsulates whole program
• There are newer/better architectures that can exploit hardware parallelism

Idea

• Treat ABI as a source language
  – the specification
• Cross compile (translate) old ISA to new architecture (ISA?)
• Do it below the model level
  – user doesn’t need to be cognizant of translation
• Run on simpler/cheaper/faster/newer hardware
Complications

- User visibility
- Preserving semantics
  - e.g. condition code generation
- Interfacing
  - preserve visible machine state
  - interrupt state
- Finding the code
  - self-modifying/runtime generated code
  - library code

Base

- Each operation has a meaning
  - behavior
  - affect on state of machine

- stws r29, 8(r8)
  - tmp=r8+8
  - store r29 into [tmp]
- add r1,r2,r3
  - r1=(r2+r3) mod 2^{31}
  - carry flag = (r2+r3>= 2^{31})
Capture Meaning

• Build flowgraph of instruction semantics
  – not unlike the IR (intermediate representation) for a compiler
    • what use to translate from a high-level language to ISA/machine code
  – e.g. IR saw for Bulldog (trace scheduling)

Optimize

• Use IR/flowgraph
  – eliminate dead code
    • esp. dead conditionals
    • e.g. carry set which is not used
  – figure out scheduling flexibility
    • find ILP
Trace Schedule

• Reorganize code
• Pick traces as linearize
• Cover with target machine operations
• Allocate registers
  – (rename registers)
  – may have to preserve register assignments at some boundaries
• Write out code

Details

• Seldom instruction→instruction transliteration
  – extra semantics (condition codes)
  – multi-instruction sequences
    • loading large constants
    • procedure call return
  – different power
    • offset addressing?,
    • compare and branch vs. branch on register
• Often want to recognize code sequence
Complications

• How do we find the code?
  – Known starting point
  – Entry points
  – walk the code
  – …but, ultimately, executing the code is the original semantic definition
    • may not exist until branch to...

Finding the Code

• **Problem:** can’t always identify statically
• **Solution:** wait until “execution” finds it
  – delayed binding
  – when branch to a segment of code,
    • certainly know where it is
    • and need to run it
  – translate code when branch to it
    • first time
    • nth-time?
Common Prospect

- Translating code is large fixed cost
  - but has low incremental cost on each use
  - hopefully comparable to or less than running original on old machine
- Interpreting/Emulating code may be faster than “compiling” it
  - if the code is run once
- Which should we do?

Optimization Prospects

- Translation vs. Emulation
  - $T_{\text{trun}} = T_{\text{trans}} + nT_{\text{op}}$
  - $T_{\text{trans}} > T_{\text{em,op}} > T_{\text{op}}$
- If compute long enough
  - $nT_{\text{op}} >> T_{\text{trans}}$
  - → amortize out load
Competitive Approach

• Run program emulated
• When a block is run “enough”, translate
• Consider
  – $N_{\text{thresh}} T_{\text{emop}} = T_{\text{translate}}$
• Always w/in factor of two of optimal
  – if $N<N_{\text{thresh}}$ optimal
  – if $N=N_{\text{thresh}}$ paid extra $T_{\text{translate}} = 2 \times \text{optimal}$
  – as $N>>N_{\text{thresh}}$ extra time amortized out with translation overhead
    • think $T_{\text{translate}} \sim 2T_{\text{translate}}$

On-the-fly Translation Flow

• Emulate operations
• Watch frequency of use on basic blocks
• When run enough,
  – translate code
  – save translation
• In future, run translated code for basic block
Translation “Cache”

• When branch
  – translate branch target to new address
  – if hit, there is a translation,
    • run translation
  – if miss, no translation
    • run in emulation (update run statistics)

Alternately/Additionally

• Rewrite branch targets so address translated code sequence
  – when emulator finds branch from translated sequence to translated sequence
  – update the target address of the branching instruction to point to the translated code
Self-Modifying Code

- Mark pages holding a translated branch as read only
- Take write fault when code tries to write to translated code
- In fault-handler, flush old page translation

Precise Exceptions

- Again, want exception visibility relative to simple, sequential model
  - …and now old instruction set model
- Imposing ordering/state preservation is expensive
Precise Exceptions

• Modern BT technique [hardware support]
  – “backup register” file
  – commit/rollback of register file
  – commit on memories
  – on rollback, recompute preserving precise state
    • drop back to emulation?
• …active work on software-only solutions
  – e.g. IBM/WBT’00

Remarkable Convergence?

• Aries: HP PA-RISC → IA-64
  – new architecture
• IBM: PowerPC → BOA
  – ultra-high clock rate architecture? (2GHz)
    • IBM claims 50% improvement over scaling?
    • 700ps = 1.4GHz in 0.18µm
• Transmeta: x86 → Crusoe
  – efficient architecture, avoid x86 baggage
Remarkable Convergence?

- All doing dynamic translation
  - frequency based
- To EPIC/VLIW architectures

Performance

[CAVEAT: trade magazine, numbers for system]
## Academic Static Binary Translation

<table>
<thead>
<tr>
<th>Program</th>
<th>Translated Code</th>
<th>Native Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gcc opt</td>
<td>cc opt</td>
</tr>
<tr>
<td>Fibo(40) sec bytes</td>
<td>27.7</td>
<td>28.5</td>
</tr>
<tr>
<td></td>
<td>16,512</td>
<td>7,292</td>
</tr>
<tr>
<td>Sieve(3000) sec bytes</td>
<td>17.8</td>
<td>17.4</td>
</tr>
<tr>
<td></td>
<td>16,244</td>
<td>6,548</td>
</tr>
<tr>
<td>Mbanner(500K) sec bytes</td>
<td>42.5</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>22,240</td>
<td></td>
</tr>
</tbody>
</table>

### Static SPARC to Pentium Translation

[Cifuentes et. al., Binary Translation Workshop 1999]

## Academic/Static BT

<table>
<thead>
<tr>
<th>Program</th>
<th>Translated Code</th>
<th>Native Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gcc opt</td>
<td>cc opt</td>
</tr>
<tr>
<td>Fibo(40) sec bytes</td>
<td>23.0</td>
<td>24.3</td>
</tr>
<tr>
<td></td>
<td>24,916</td>
<td>6,680</td>
</tr>
<tr>
<td>Sieve(3000) sec bytes</td>
<td>26.9</td>
<td>23.9</td>
</tr>
<tr>
<td></td>
<td>24,776</td>
<td>6,312</td>
</tr>
<tr>
<td>Mbanner(500K) sec bytes</td>
<td>53.3</td>
<td>36.9</td>
</tr>
<tr>
<td></td>
<td>34,198</td>
<td>21,448</td>
</tr>
</tbody>
</table>

### Static Pentium to SPARC Translation

[Cifuentes et. al., Binary Translation Workshop 1999]
Academic/Dynamic BT

<table>
<thead>
<tr>
<th>Tool</th>
<th>Startup Time</th>
<th>Translation Time</th>
<th>Execution Time without caching</th>
<th>Execution Time with caching</th>
<th>Optimization</th>
<th>Execution Time without caching</th>
<th>Execution Time with caching</th>
<th>Sector go-</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sleev3200</td>
<td>0.54</td>
<td>5.14</td>
<td>98.23</td>
<td>73.14</td>
<td>0.24</td>
<td>80.98</td>
<td>96.29</td>
<td>29.22</td>
</tr>
<tr>
<td>Floo3200</td>
<td>0.54</td>
<td>0.10</td>
<td>185.17</td>
<td>154.97</td>
<td>0.31</td>
<td>147.56</td>
<td>135.69</td>
<td>41.10</td>
</tr>
<tr>
<td>DECAF</td>
<td>0.62</td>
<td>0.24</td>
<td>219.01</td>
<td>116.28</td>
<td>0.37</td>
<td>146.22</td>
<td>124.28</td>
<td>22.85</td>
</tr>
</tbody>
</table>

Table 1: Pentium to SPARC translation (second)

[Ung+Cifuentes, Binary Translation Workshop 2000]

Upcoming

- **Next Class (Last): Thursday, March 8th**
  - no class this Thursday
  - no class Tuesday (week from today)

- **Want to see:**
  - assignments 6, 7, 8
  - finished by next class (3/8)
Big Ideas

• Well-defined model
  – High value for longevity
  – Preserve semantics of model
  – How implemented irrelevant

• Hoist work to earliest possible binding time
  – dependencies, parallelism, renaming
  – hoist ahead of execution
    • ahead of heavy use
  – reuse work across many uses

• Use feedback to discover common case