Experiences with Dynamic Binary Translation

David R. Ditzel
VP and Chief Architect for Hybrid Parallel Computing
Intel Corporation

June 21, 2008
ISCA AMAS-BT Workshop Keynote
Any opinions expressed in the following talk are purely my own, and do not necessarily reflect any official position of any of my current or previous employers. Technical data presented in this presentation is believed to have been previously disclosed in prior public conferences, presentations or patent publications.
Agenda

• Examples
• History
• Transmeta: Proof of concept
• Hardware support for binary translation
• Myth busters
• Futures
• Summary
Binary Translation Success Stories

- Pentium Pro and its successors
  - Translates x86 binary code into internal UOPS via hw
- Intel IA32EL
  - Translates user level x86 programs on IPF via sw
  - Performance about 60% of native machine
- JAVA JITs / Microsoft MSIL / VMware
- Transitive Technologies
  - Rosetta runs Apple PowerPC user programs on x86
- Transmeta Crusoe and Efficeon mobile processors
  - Translates x86 binary code into internal UOPS via sw
  - Transparent full system level translation as real products
  - Performance comparable to Intel’s mobile CPUs
Good work worth mentioning:

- **XDOS**: Early static DOS to Unix
- **DEC FX!32**: x86 user apps to Alpha
- **IBM DAISY, BOA**: PowerPC on VLIW
- **Elbrus nArch, E2K**: SPARC, x86 to VLIW
- **Sun SoftWindows**: x86 to SPARC/Solaris
- **Intel PIN**: x86 binary instrumentation
- **Intel StarDBT**: Intel research translator
- **and many others ...**
- **see Virtual Machines**: Book by Jim Smith & Ravi Nair
Hybrid Processors

• “Binary translation” can mean too many things
  – User level – Only the application is translated
  – System level – Both application and OS are translated
  – Static Translation – Done once to make a new binary
  – Dynamic Translation – Done while program is running, and usually leaves no permanent copy (Like P4 trace cache or JAVA JIT)

• A Hybrid processor – in this presentation
  – Specifically means a microprocessor implemented as a hardware/software co-design using software binary translation and optimization with special purpose hardware so that it can function as a fully compatible microprocessor with performance comparable to pure hardware implementations.
Early hardware translation 1962-1975

• High Level Language Computer announced in 1971
• Supported by Gordon Moore and Robert Noyce at Fairchild until they left to form Intel
• Goal was to reduce cost of software by using hardware
• Implemented Compiler, Text Editor, OS in logic gates
  – 2 gates or 1 FF per 14-pin DIP package, no ROM
  – 20 thousand chips – took several years to debug at ISU
  – five years of my life thinking about CISC vs RISC

Lessons:
• Don’t build your compiler or OS in logic gates
• Use right combination of Hardware, Software or μCode

1 Source: F. Soltis, Fortress Rochester: The Inside Story of IBM iSeries, page 364
Debugging SYMBOL, the inspiration for RISC

A Symbol “terminal” could consist of up to 99 physical I/O devices. The terminal shown at left used a modified IBM Selectric typewriter, editing keyboard, status display, card reader, and line printer. The book next to the typewriter contains operating system and utility source listings. The photo above, a side view of the Symbol mainframe, shows the maintenance processor (far left), power supplies for +4.5 volts at 1000 amps (below the mainframe), and disk memory, core memory, and paging drum (background). The front view of the mainframe (photo at right) shows a printed circuit card on an extender for testing. Each card held 200 ICs. Wing panels indicate the value of bus signals—100 on the left, 100 on the right, and 50 on top. Additional wire on the PCB was used for “bug” fixes. Each processor could be monitored from a “processor active” lamp on the system's control panel.

AT&T CRISP

- C-Language Reduced Instruction Set Processor
- Announced 1987, 1st CMOS superscalar chip
- HW translated into 180-bit wide decoded UOP cache
- Branch folding, used “Stack Cache” as registers
- Low power, used in EO Tablet computer
- SW binary translator proposed in 1985 to move software from CISC WE32100

Lessons
- External to internal translation worked well
- Decouple external and internal ISA

Source: D. Ditzel, Hardware Architecture of the CRISP Microprocessor, ISCA, June 1987
BT for Binary Instrumentation

- MIPS pixie/pixstats ~1987
  - Statically modified binaries to count instructions

- Sun Spix/Spixstats/Span/Shadow/Shade ~1988-95
  - Success led to more functionality
  - Ability to patch in analysis routines while running
  - Shade ~1990 enhanced dynamic techniques
    - able to run MIPS and x86 code on SPARC, about 1/3 native speed
    - Amazed at the performance of non-native code

Lessons – Critical realizations in 1995
- Co-design the underlying hardware to reduce inefficiencies
- Could move from less than native to net performance gain
- Meant Hybrid processors could compete with mainstream
Transmeta

- Transmeta was founded in 1995, with the idea to co-design hardware and software together in a machine whose goal was provide an efficient base for binary translation of x86 programs
- Spent $600M in R&D over 12 years to learn the tricks
- CMS was invisible to users, no compatibility bugs
- Five generations of processors designed
  - First two never shipped, insufficient performance
  - Crusoe product—Pentium class for mobile (250nm-180nm)
  - Efficeon product—Pentium-M class for mobile (130nm-90nm)
  - Next generation design that never shipped

Lesson: Worked well and was on a rapid learning curve
  - IPC improving at roughly 2x / generation
To produce high performance while remaining perfectly faithful to an existing architecture, translator must optimize aggressively:

- Speculation: Translator makes aggressive assumptions about code to achieve higher performance

- Recovery: Check assumptions and rollback to commit points if they prove to be false, for precise interpretation

- Adaptive retranslation: If recovery is required too often, retranslate with less aggressive assumptions

<< Slides from Jim Dehnert’s 2003 CGO presentation >>
**Shadow registers:** Working and shadow copies of x86 registers

**Commit atom:** Copies working registers to shadow registers, commits memory writes

**Rollback atom:** Copies shadow registers to working registers, discards memory writes
Example: Precise Exceptions

**Problem**: CMS rearranges operations in a translation, but x86 has precise exception semantics

**Speculation**: CMS translations scheduled assuming no exceptions will occur

**Recovery**: If an exception occurs, rollback to preceding consistent commit point, and interpret sequentially

**Adaptive retranslation**: An instruction causing exceptions too often is isolated, and the rest of the original translated code is retranslated so it won’t need rollback
Example: Memory-mapped I/O

**Problem:** Arbitrary x86 memory operations may be memory-mapped I/O, which must not be reordered and may not be rolled back.

**Speculation:** CMS translates memory operations as *normal*, reorders them at will (unless marked by interpreter as *abnormal*).

**Recovery:** A normal reference to abnormal memory causes a hardware fault, rollback, and appropriate interpretation.

**Adaptive retranslation:** A memory op that is abnormal too often is retranslated with commits before and after it, and with other constraints necessary for memory-mapped I/O.
Figure 2: Degradation Caused by Suppressing Memory Reordering

| Application                  | DOS Boot | Linux Boot | OS/2 Boot | Windows 95 Boot | Windows 98 Boot | Windows ME Boot | Windows NT Boot | Windows XP Boot | Mean (all boots) | Quake Demo2 (DOS) | 023.eqntott (Linux) | 026.compress (Linux) | 072.sc (Linux) | 085.gcc (Linux) | 047.tomcatv (Linux) | 048.ora (Linux) | 052.alvinn (Linux) | 077.mdljsp2 (Linux) | Multimedia (Win98) | CPUmark99 (Win98) | Quattro Pro (WinNT) | Wordperfect (WinNT) | Mean (all apps) |
|------------------------------|----------|------------|-----------|-----------------|----------------|----------------|-----------------|-----------------|------------------|------------------|---------------------|--------------------|-------------------|----------------|----------------|--------------------|----------------|-------------------|-------------------|--------------------|----------------|-----------------|------------------|----------------------|
| Internal boot load           | 1.48%    | 7.58%      | 1.95%     | 1.95%           | 17.19%         | 26.08%         | 4.19%           | 19.64%          | 10.09%           | 29.71%           | 44.64%              | 97.29%             | 66.22%            | 78.38%          | 35.44%          | 37.54%             | 22.76%          | 24.24%            | 17.97%            | 33.14%             | 24.24%            | 17.97%          |
Example: Data Speculation (Aliases)

**Problem:** To achieve effect of out-of-order processors on memory operations, CMS needs to reorder memory operations

**Speculation:** CMS assumes memory operations don’t alias unless it can prove otherwise, reorders accordingly

**Recovery:** Memops scheduled earlier set an *alias register*; later memops that might alias check the register and trap on overlap

**Adaptive retranslation:** Translation that takes alias faults too often is translated without reordering
Example: Data Speculation (Aliases)

Simulations obtained without using alias HW

No adaptive retranslation cuts PCmark2002 3d vector performance by a factor of 47.5 on hardware
Self-Modifying Code (SMC)

**Original problem**: If the x86 code modifies itself, the CMS translations must be invalidated or otherwise adapt.

**Speculation**: Normal translations assume no SMC

**Simple recovery**: Write-protect x86 code pages, find and invalidate corresponding translations if a fault occurs

**Secondary problems**:
- Inefficient for self-modifying code: granularity too large
- Can’t distinguish data in same page as code

**Costs incurred by CMS**:
- Handling fault, invalidating translations, special processing
- Generating new translations for new code
SMC: Fine-Grain Protection

**First refinement:** Hardware support for sub-page granularity

Only needed for a few pages at a time, allowing tiny hardware cache

Without fine-grain protection, the worst cases:

<table>
<thead>
<tr>
<th></th>
<th>Faults</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Win95 Boot</td>
<td>52.8x</td>
<td>2.2x</td>
</tr>
<tr>
<td>Win98 Boot</td>
<td>59.4x</td>
<td>3.8x</td>
</tr>
<tr>
<td>MultimediaMark</td>
<td>46.8x</td>
<td>1.6x</td>
</tr>
<tr>
<td>WinStone Corel</td>
<td>54.2x</td>
<td>2.1x</td>
</tr>
<tr>
<td>Quake Demo2</td>
<td>7.7x</td>
<td>1.02x</td>
</tr>
</tbody>
</table>
Top 10 hardware support needs for efficient Hybrid Processors

1. Software controlled state. Commit/Rollback/Abort
2. More registers than architected state
3. Alias detection under software control
4. Self modifying code detection with fine grain support
5. Auto-typing of pure memory vs I/O
6. Fast traps supported by underlying runtime system
7. Instruction primitives for fast interpretation
8. Private memory. ~5% of DRAM for translated code
9. Private non-volatile storage (FLASH ROM) with every chip
10. Competitive CPU hardware, ie uOp ISA, IPC and clock rate

Lesson: With proper hardware support, software can deliver outstanding net performance and performance per watt.
Speculative Execution and x86 State Control

Hardware for software controlled Commit/Rollback enables wide optimization regions

All registered state has two copies
  • Working state
  • Committed State

Registered State includes:
  • Integer and Floating Point register files
  • Condition code flags
  • All other state registers

Support for speculative stores between commit points in L1 data cache
  • New cache state
  • 1 new tag bit “speculative”
  • Flash (1-cycle) flush (on rollback) or convert to dirty (commit)

New UOP level Instructions
  • Commit: Copies working registers to committed state (1-cycle)
  • Rollback: Copies committed registers to working registers (1-cycle)
  • Abort: To runtime trap handler, which can do fix up, rollback, and branch
PS – The Efficeon processor efficiently implemented transactional memory using this technique.
Each clock, Efficeon can issue from one to eight 32-bit instruction “atoms” to 11 functional units.

Instruction

atom1 atom2 atom3 atom4 atom5 atom6 atom7 atom8

Functional Units

FP / MMX SSE / SSE2

MMX SSE / SSE2

Integer ALU-1

Integer ALU-2

Alias

Control

Load or Store or 32-bit add

Load or Store or 32-bit add

Branch

Exec-1

Exec-2

Source: 2003 Microprocessor Forum Presentation
Code Morphing Software

**4 Gear System Significantly Improved Responsiveness and Overall Performance**

1st Gear
Executes 1 instruction at a time

- Profiles code at runtime
- Gathers data for flow analysis
- Gathers branch frequencies and directions
- Detects load/store typing (IO vs memory)

Filters out infrequently executed code

**No startup cost**
**Lowest speed**

Source: 2003 Microprocessor Forum Presentation
4 Gear System Significantly Improved Responsiveness and Overall Performance

1st Gear

2nd Gear

Uses profile data to create initial translations after code reaches 1st threshold.

- Translates a “Region” of up to 100 x86 instructions.
- Adds flow graph “Shape” information
- Light Optimization
- “Greedy” scheduling

Low translation overhead
Fast execution

Source: 2003 Microprocessor Forum Presentation
Code Morphing Software

4 Gear System Significantly Improved Responsiveness and Overall Performance

1st Gear

2nd Gear

3rd Gear

Further optimizes the 2nd gear regions

- Common sub-expression elimination
- Memory re-ordering
- Significant code optimization
- Critical path scheduling

Medium translation overhead
Faster execution

Source: 2003 Microprocessor Forum Presentation
**4 Gear System Significantly Improved Responsiveness and Overall Performance**

1st Gear

2nd Gear

3rd Gear

4th Gear

Most advanced optimizations for “hottest” code regions.

- Splices together multiple regions
- Optimizes across region boundaries
- Used advanced behavioral data
- Critical path scheduling

**Highest translation overhead**

**Fastest execution**

Source: 2003 Microprocessor Forum Presentation
Some of Efficeon’s Code Morphing Software optimizations:
- Aggressive scheduling of instruction level parallelism
- Out-of-Order instruction execution obtained on In-Order hardware
- Critical path height reduction
- Common sub-expression elimination
- Uses “Address Alias Checking Hardware”
  - Re-ordering of loads and stores even with potential aliases
  - Elimination of loads and stores even with potential aliases
- Software register renaming
- Fusing operations
- Dead code elimination
- Removal of conditional branches
- Adaptive re-translation during program execution
- Loop unrolling and optimization
  - Remove Exit Branches
  - Code motion across back-edge
  - Loop invariant code motion
  - Strength reduction

Important “hot” codes get the most optimization
**Performance Comparison**
From Sept 2004 Microprocessor Forum

### Relative Performance

#### CPUMark
- **Transmeta Efficeon**
  - Score: 170
- **Intel Pentium-M**
  - Score: 134
- **Intel Celeron-M**
  - Score: 109

#### PCMark2004
- **Transmeta Efficeon**
  - Score: 2452
- **Intel Pentium-M**
  - Score: 1984
- **Intel Celeron-M**
  - Score: 1706

---

Benchmarks run at Transmeta on similarly configured systems, using highest MHz shipping in comparable notebook systems.

**Efficeon-2 data measured at 1.6 GHz with 512MB DDR333, 5400rpm 2.5” HDD, NV17 graphics**

**Intel Pentium-M data measured at 1.1 GHz with 512MB DDR333, 5400rpm 2.5” HDD, i855GMCH**

**Intel Celeron-M data estimated by measuring Celeron-M at 800 MHz (scores: 97/1517) and estimating 933 MHz score**

---

Source: 2004 Microprocessor Forum Presentation
Transmeta proved Dynamic Binary Translation works in real products, not just in theory.

A few examples of the hundreds of different systems all using Crusoe/Efficeon with Dynamic Binary Translation.

- SONY
- Hitachi
- Fujitsu
- Sharp
- Blade Servers

- Gateway/AOL “Connected Touch Pad”
- FIC “Aqua” Internet Appliance
- Microsoft Tablet PC
- OQO Handheld WinXP
Common Untrue Myths

Myth: Software overhead is very high with dynamic translation

Myth: Binary translation software can’t be reliable, and will be full of bugs

Myth: Static compilation is faster than dynamic binary translation

Myth: Can’t build a competitive processor using binary translation

Myth: Binary translation won’t be important in the future
Why we need Binary Translation

Good reasons for Binary Translation

Innovation
  • To allow processor innovation not tied to particular instruction sets
  • Using BT to provide backwards compatibility

Performance
  • To enable new means to improve processor performance
  • Using BT to provide backwards compatibility

Power
  • To enable simpler and lower power processors
  • Using BT to provide backwards compatibility
Conclusions

Binary Translation based processors can work well, no longer a theory.

Special purpose hardware support is needed, co-designed with software.

Software translation can be done poorly or well, special care is needed to keep translation overhead low.

Many opportunities for clever hardware/software co-design tradeoffs

This is a technological approach still in its infancy

Prediction: Dynamic Binary Translation will become a basic technique used in future processor design, as integral as logic gates and microcode are today.