Dynamic Binary Translation and Optimization

Erik R. Altman
Kemal Ebcioğlu
IBM T.J. Watson Research Center

Micro-33
December 13, 2000
Timetable for Micro-33 Tutorial on Dynamic Binary Translation and Optimization

Wednesday, December 13, 2000

2:30 - 2:50  Kemal Ebcioglu: Future Challenges
2:50 - 2:55  Erik Altman: DAISY Demo
2:55 - 3:20  Erik Altman: Binary Translation Issues
3:20 - 3:35  Break
3:35 - 5:00  Erik Altman: DAISY, Crusoe, Dynamo
5:00 - 5:15  Break
5:15 - 5:45  Kemal Ebcioglu: LaTTe
DAISY Schematic

AIX Applications

AIX

DAISY Software

VLIW Machine

PowerPC
DAISY Memory Map

- Translator
- Translated Code
- Side Tables
- System Software

PowerPC Memory
DAISY Source Architecture

• Most DAISY work uses **PowerPC** as the *source architecture*.

• But, the DAISY approach is general:
  
  – ICS’2000 reported how to use DAISY with **S/390** as the *source architecture*.
  
Optimization Unit

• Page:
  – Used by first versions of DAISY.

• Arbitrary: with translated code forming a tree region:
  – Currently used by DAISY.

• Basic Block
• Single path/trace
• Loop
• Function
Problems of Page

- Cross page boundary, have indirect branch, or other serializer every 10-15 instructions.
- Generate code for all reachable paths on page, even those not executed.
- Unnecessary serializations to limit code explosion.

16 paths A-E. To avoid code bloat, page-based serializes after 2 paths with D ⇒ Bad performance on 14/16 paths.
Sample of Tree Region

Original Control Flow Graph

Control Flow Graph of Translated Code
Characteristics of Tree Region

- At most one reaching definition.
- Register allocation simplified.
- Optimization can specialize for each path
  - E.g. A register may contain a constant along one path.
- Exit point of group uniquely specifies path through group
  ⇒ Good feedback info
- Code explosion can be a problem.
DAISY Operation

Interpret Ins, Add to Grp X
DAISY Operation

Interpret Ins, Add to Grp X

Stopping Point
DAISY Operation

Interpret Ins, Add to Grp X → Stopping Point

No
DAISY Operation

Interpret Ins, Add to Grp X

Stopping Point

Y = X
X = New Grp

Yes

No
DAISY Operation

Interpret Ins, Add to Grp X

Stopping Point

Y=X
X=New Grp

Interp 30x
DAISY Operation

- **Interpret Ins, Add to Grp X**
- **Stopping Point**
  - **Yes**: Y=X, X=New Grp
  - **No**: At Prev Translated Entry Pt
- **Interp 30x**
DAISY Operation

Interpret Ins, Add to Grp X

At Prev Translated Entry Pt

Stopping Point

Yes

Y=X

X=New Grp

No

Interp 30x
DAISY Operation

Interpret Ins, Add to Grp X → Stopping Point

At Prev Translated Entry Pt

Y=X
X=New Grp

Interp 30x

Translate and Sched Group Y to VLIW Instruc
DAISY Operation

- Interpret Ins, Add to Grp X
- Stopping Point
- Y=X
  X=New Grp
- Translate and Sched Group Y to VLIW Instruc
- At Prev Translated Entry Pt
- Interp 30x
  Yes
  No
  No
  No
  Yes
DAISY Operation

- Interpret Ins, Add to Grp X
- At Prev Translated Entry Pt
  - No
  - Yes: Execute Group’s VLIW Translation
- Stopping Point
  - Yes: Y=X, X=New Grp
  - No
- Interp 30x
  - No
  - Yes: Translate and Sched Group Y to VLIW Instruc
DAISY Operation

1. Interpret Instruction, Add to Group X
2. At Previous Translated Entry Point
   - No: Interpret 30x
   - Yes: Execute Group’s VLIW Translation
3. Stopping Point
   - Yes: Y = X, X = New Group
   - No: Translate and Schedule Group Y to VLIW Instruction
DAISY Stopping Points

Frequently Executed Code?

- No
  - > 24 Ops
    - Yes
    - Good Stop Ins: BLR, BCTR, BL, Loopback
  - ILP > 3
    - Yes
    - Good Stop Ins: BLR, BCTR, BL, Loopback
- Yes
  - ILP > 10
    - Yes
    - At Good Stop Ins
      - Yes
      - Good Stop Point
Extending Groups

If \texttt{beq} taken later in execution, \texttt{TRANSLATOR} can extend existing VLIW group, or start another at \texttt{X+64}.
Extending Groups

**PowerPC**

- $X$
- $X+4$
- $X+8$
- $X+12$
- $X+16$
- $X+20$
- $X+64$
- $X+68$

**DAISY VLIW Group**

- op1
- op2
- op3
- beq
- op4
- op5
- op6
- op7

$b$ TRANSLATOR

$b$ X+24 Translation
Knowing When to Reoptimize

- Add profiling code at group exits
- Timer Interrupts
  - Coarse Granularity/Less Accurate
- Hardware Array of Cached Counters
  - DAISY Approach
  - Indexed by Exit Point of Group
  - Auto-incremented at Group Exit
  - 8K Entries, 8-way associative
DAISY Optimizations

- ILP Scheduling with data and control speculation
- Loop Unrolling
- Alias Analysis
- Load-Store Telescoping
- Copy propagation
- Combining
- Unification
- Limited dead code elimination
Load-Store Telescoping

- `def r1`
- `STORE r1, (X)`
- `...`
- `LOAD r2, (X)`
- `...`
- `STORE r2, (Y)`
- `...`
- `LOAD r3, (Y)`
- `use r3`
Load-Store Telescoping

- \texttt{def} \hspace{1em} r1
- \texttt{STORE} \hspace{1em} r1, (X)
- \texttt{LOAD} \hspace{1em} r2, (X)
- \texttt{STORE} \hspace{1em} r2, (Y)
- \texttt{LOAD} \hspace{1em} r3, (Y)
- \texttt{use} \hspace{1em} r3

- \texttt{def} \hspace{1em} r1
- \texttt{STORE} \hspace{1em} r1, (X)
- \texttt{LOAD} \hspace{1em} r2, (X)
- \texttt{STORE} \hspace{1em} r2, (Y)
- \texttt{LOAD} \hspace{1em} r3, (Y)
- \texttt{use} \hspace{1em} r3
DAISY Register Conventions

- DAISY registers r0-r31 always contain the values in PowerPC registers r0-r31.

- DAISY registers r36-r63 are used for renaming speculative results during scheduling, and as scratchpads.

- DAISY register r32 has PowerPC `counter` value.

- DAISY register r33 has PowerPC `linkreg` value.

- DAISY register r35 contains the constant 0.
  
  - On PowerPC memory accesses, using r0 as an address means literal 0. Keeping 0 in r35 simplifies renaming of r0 in some cases.

- DAISY register r34 contains the now defunct Power MQ register.
DAISY Scheduling Example
**Original PowerPC Code**

1) \text{add} & r1, r2, r3
2) \text{bc} & L1
3) \text{sli} & r12, r1, 3
4) \text{xor} & r4, r5, r6
5) \text{and} & r8, r4, r7
6) \text{bc} & L2
7) \text{b} & \text{OFFPAGE}
8) L1: \text{sub} & r9, r10, r11
9) \text{b} & \text{OFFPAGE}
10) L2: \text{cntlz} & r11, r4
11) \text{b} & \text{OFFPAGE}

**Translated VLIW Code**

\text{VLIW1:}

\begin{align*}
& \text{add} & r1, r2, r3 \\
& \text{add} & r1, r2, r3
\end{align*}
Original PowerPC Code

1) add r1, r2, r3
2) bc L1
3) sli r12, r1, 3
4) xor r4, r5, r6
5) and r8, r4, r7
6) bc L2
7) b OFFPAGE
8) L1: sub r9, r10, r11
9) b OFFPAGE
10) L2: cntlz r11, r4
11) b OFFPAGE

Translated VLIW Code

VLIW1:

add r1, r2, r3
bc L1

**Original PowerPC Code**

1) `add     r1, r2, r3`
2) `bc       L1`
3) `sli     r12, r1, 3`
4) `xor     r4, r5, r6`
5) `and     r8, r4, r7`
6) `bc       L2`
7) `b        OFFPAGE`
8) L1:  `sub   r9, r10, r11`
9) `b        OFFPAGE`
10) L2:  `cntlz r11, r4`
11) `b        OFFPAGE`

**Translated VLIW Code**

```
VLIW1:
    add r1, r2, r3
    bc   L1
    b     VLIW2

VLIW2:
    sli r12, r1, 3
```
Original PowerPC Code

1) add r1, r2, r3
2) bc L1
3) sli r12, r1, 3
4) xor r4, r5, r6
5) and r8, r4, r7
6) bc L2
7) b OFFPAGE
8) L1: sub r9, r10, r11
9) b OFFPAGE
10) L2: cntlz r11, r4
11) b OFFPAGE

Translated VLIW Code

VLIW1:
  add r1, r2, r3
  bc L1
  xor r63, r5, r6
  b VLIW2

VLIW2:
  sli r12, r1, 3
  r4 = r63
Original PowerPC Code

1) add r1, r2, r3
2) bc L1
3) sli r12, r1, 3
4) xor r4, r5, r6
5) and r8, r4, r7
6) bc L2
7) b OFFPAGE
8) L1: sub r9, r10, r11
9) b OFFPAGE
10) L2: cntlz r11, r4
11) b OFFPAGE

Translated VLIW Code

VLIW1:

add r1, r2, r3
bc L1
xor r63, r5, r6
b VLIW2

VLIW2:

sli r12, r1, 3
r4=r63
and r8, r63, r7
Original PowerPC Code

1) add    r1,r2,r3
2) bc     L1
3) sli    r12,r1,3
4) xor    r4,r5,r6
5) and    r8,r4,r7
6) bc     L2
7) b      OFFPAGE
8) L1: sub r9,r10,r11
9) b      OFFPAGE
10) L2: cntlz r11,r4
11) b     OFFPAGE

Translated VLIW Code

VLIW1:
- add r1,r2,r3
- bc   L1
- xor r63,r5,r6
- sub r9,r10,r11
- b    VLIW2
- b    OFFPAGE

VLIW2:
- sli r12,r1,3
- r4=r63
- and r8,r63,r7
- bc   L2
- cntlz r11,r63
- b    OFFPAGE
- b    OFFPAGE
DAISY Speculation

- DAISY aggressively speculates operations:
- **Control Speculation:** Operations Above Branches
- **Data Speculation:** Loads above possibly aliased stores.
DAISY Data Speculation

- Loads moved above possibly aliased stores have a load-verify instruction inserted at the in-order location of the original load.

- **Load-verify** reloads the value and checks if it is the same as the speculatively loaded value.

- **Yes** ⇒ Speculation ok – execution continues normally.

- **No** ⇒ Speculation bad – Trap to VMM and recover.
DAISY Data Speculation Recovery

- Speculative values in non-
  
  PowerPC 
  
  regs can be thrown away.

- Execution can resume with using the in-order value of the load.

- If a particular load-verify fails frequently, the VMM can retranslate the block of code, so as not to speculate this load.

- Result: Simple alias analysis by translator suffices.
Load-Verify Patent

• The **load-verify** concept comes from U.S. Patent **5758051**:  
  – “Method and apparatus for reordering memory operations in a processor.”
  – **Inventors:** Jaime Moreno, Mayan Moudgill.
DAISY Address Translation

On Hit:
- Real Address
  - WIMG.W
    (Write-through cache)
  - WIMG.I
    (Inhibit Caching)
  - WIMG.M
    (Memory Coherent)
  - WIMG.G
    (Guarded Access)

On Miss:
- Software Page Table Walk
- Raise Exception (Sometimes)
- Update R/C bits (Referenced, Changed)
DAISY: Dealing with I/O Accesses

- Check **WIMG** Bits associated with each **PowerPC** Page.
- (Translation off uses **WIMG=0011**).
- **I** = Caching Inhibited Page.
- **G** = Guarded Storage.
- **I=1** and **G=1** ⇒ I/O Access.
- Speculative I/O Access ⇒ **Quash**.
Cross-Group Branches (1)

Multiple ways to branch between translated code groups:

- **Branch directly:**
  - Fast.
  - VMM must track invalidations of translated groups and fix other groups that branch to them.
Cross-Group Branches (2)

Multiple ways to branch between translated code groups:

- Cross group branches are looked up in a special LVIA cache.

- **LVIA** = **Load VLIW Instruction Address**.

  \[
  \text{lvia } rX, (rY) \\
  b \quad rX
  \]

  \[
  rY = \text{PPC Code Addr} \quad \Rightarrow \quad \text{LVIA Cache} \quad \Rightarrow \quad \text{VLIW Addr} = rX
  \]

  - The **LVIA** cache is backed up by a larger memory list of translations akin to page tables.
  - If no translation exists, the **LVIA** op returns the address of the translator.
• Asynchronous exceptions such as timer interrupts can be delayed until a convenient point:

  – Convenient points are typically group transitions.
  – At group transitions, all state for emulated code is contained in source architecture registers and memory — nothing is in speculative DAISY only registers.

• Upon fielding an asynchronous exception:

  – DAISY sets registers such as SRR0 indicating the PowerPC address of the excepting instruction.
  – DAISY translates the PowerPC exception handler if this has not been done previously.
  – DAISY executes the translated exception handler.
Precise PowerPC Exceptions in DAISY (2)

- If there are multiple operations in a VLIW instruction and one causes a **synchronous** exception,
- The entire VLIW instruction is treated as unexecuted.
- The DAISY VMM then interprets the **PowerPC** operations corresponding to the non-speculative operations in the VLIW instruction to find the exact point of the exception.
- It then invokes a translation of the **PowerPC** exception handler.
DAISY: Getting PowerPC Exception Address

- VMM knows where to find **PowerPC** operations corresponding to a VLIW instruction as follows:
  
  - **VPA** Register with **PowerPC** Effective Address of Page (20 bits).
  
  - When translating from **PowerPC** to VLIW, VMM updates side table with 10 bits for offset in **PowerPC** page corresponding to start of each VLIW instruction.
  
  - Combining VPA and offset gives the full effective address of the **PowerPC** code corresponding to this VLIW.
  
  - VMM translates this effective address to a real address.
  
  - When translated VLIW code spans a **PowerPC** page boundary, it must contain an operation to update the VPA register and verify that the real address of the next page remains unchanged.
DAISY VPA

Updated when translated VLIW code passes page boundary in PowerPC code.

In PowerPC page corresponding to VLIW instruction.
## DAISY Example: PowerPC Code

<table>
<thead>
<tr>
<th>OP#</th>
<th>ADDR</th>
<th>PowerPC</th>
<th>OP</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>0x00</td>
<td>li</td>
<td>r5</td>
<td>=0</td>
</tr>
<tr>
<td>02</td>
<td>0x04</td>
<td>addi</td>
<td>r3</td>
<td>=r3,-1</td>
</tr>
<tr>
<td>03</td>
<td>0x08</td>
<td>lbzu</td>
<td>r3,</td>
<td>r4=1(r3)</td>
</tr>
<tr>
<td>04</td>
<td>0x0C</td>
<td>cmpi</td>
<td>cr0</td>
<td>=r4,0</td>
</tr>
<tr>
<td>05</td>
<td>0x10</td>
<td>addi</td>
<td>r5</td>
<td>=r5,1</td>
</tr>
<tr>
<td>06</td>
<td>0x14</td>
<td>bf</td>
<td>cr0 .eq,</td>
<td>0x08</td>
</tr>
<tr>
<td>07</td>
<td>0x08</td>
<td>lbzu</td>
<td>r3,</td>
<td>r4=1(r3)</td>
</tr>
<tr>
<td>08</td>
<td>0x0C</td>
<td>cmpi</td>
<td>cr0</td>
<td>=r4,0</td>
</tr>
<tr>
<td>09</td>
<td>0x10</td>
<td>addi</td>
<td>r5</td>
<td>=r5,1</td>
</tr>
<tr>
<td>10</td>
<td>0x14</td>
<td>bf</td>
<td>cr0 .eq,</td>
<td>0x08</td>
</tr>
<tr>
<td>67</td>
<td>0x08</td>
<td>lbzu</td>
<td>r3,</td>
<td>r4=1(r3)</td>
</tr>
<tr>
<td>68</td>
<td>0x0C</td>
<td>cmpi</td>
<td>cr0</td>
<td>=r4,0</td>
</tr>
<tr>
<td>69</td>
<td>0x10</td>
<td>addi</td>
<td>r5</td>
<td>=r5,1</td>
</tr>
<tr>
<td>70</td>
<td>0x14</td>
<td>bf</td>
<td>cr0 .eq,</td>
<td>0x08</td>
</tr>
<tr>
<td>71</td>
<td>0x18</td>
<td>mr</td>
<td>r3</td>
<td>=r5</td>
</tr>
<tr>
<td>72</td>
<td>0x1C</td>
<td>blr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DAISY Example: VLIW Code

IRUPT → Restart at 0x08

V1:
- lbz r4=1(r3)
- addi r3=r3,1
- addi r63=r5,1
- lbz r62=2(r3)
- addi r61=r3,2
- addi r60=r5,2
- cmplr cr8=r33,0x1234
- b V2

IRUPT → Restart at 0x0C

V2:
- b V3

IRUPT → Restart at 0x0C

V3:
- cmpi cr0=r4,0
- mr r5=r63
- cmpi cr15=r62,0
- b V4

IRUPT → Restart at 0x14

V4:
- cr0.eq
- cr15.eq
- mr r4=r62
- mr r3=r61
- mr cr0=cr15
- mr r5=r60
- goto TIP_3
- goto TIP_5

(V1)
- goto TIP_7

(Start interpreting at 0x18)

(Start interpreting at 0x1C)

(Execute translated code for 0x1234, the return address)
The PowerPC architecture requires that an Instruction Cache Block Invalidate (ICBI) op occur whenever code is modified.

DAISY uses ICBI as a signal to invalidate any translations included in the invalidated block.

Keeping track of translations on a 32-byte block basis consumes a great deal of memory:

- 512 Million 32-byte blocks in 4 GBytes of memory.
- Even with 1 bit per block, 64 MBytes needed.
- ⇒ DAISY keeps track on 4KByte page basis.
DAISY and Self-Modifying Code (2)

- Many architectures like x86 and S/390 do not have ICBI-like instructions.

- Such architectures need hardware bits on each chunk of memory (invisible to the source architecture).

- When a translation is made of a chunk, the bit is set.

- If anything writes to a chunk of memory with a bit set, an exception occurs, alerting the VMM to invalidate translations from that chunk.

- Transmeta uses a scheme like this where a chunk is a page.
DAISY and Realtime Code (1)

- Dynamic Translation of realtime code is problematic.
- Known realtime code such as exception handlers can be “pretranslated.”
- In general, such code cannot be known ahead of time.
- The first time realtime code is encountered, it executes slowly — with the overhead of interpretation or translation.

**Result:** Variability and delay that may be unacceptable for realtime.
In booting AIX under DAISY dynamic translation simulator, there were no realtime problems with disks, network controllers, keyboard or mouse input, or any other hardware.

However, the kerberos system requires the system time on a machine to match that of the server.

– (kerberos system authentication expires after a certain time, and with unsynchronized systems, this becomes problematic.)
Management of Translations

- Throw away all translations when cache full.
  - Dynamo.

- LRU or Generational Garbage Collection.
  - DAISY.
    - Must throw out translations when invalidated by program.
    - Must throw out translations when translation cache is full.
    - Must throw out translations when tables indexing translation are cache full.
DAISY Performance
DAISY VLIW Size

- 16 Issue
- 16 ALU’s
- 8 Load/Store Units
- 3 Branches per Cycle
- 64 Integer Registers
- Clustered
  - 4 Clusters of 4 ALU’s
  - 2 Load/Store Units per Cluster
  - Immediate Bypass within Cluster
  - Extra Cycle Outside Cluster
## DAISY Cache Parameters

<table>
<thead>
<tr>
<th>Cache</th>
<th>Size</th>
<th>Linesize</th>
<th>Assoc</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1-I</td>
<td>64K</td>
<td>1K</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>L2-I</td>
<td>1M</td>
<td>2K</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>L1-D</td>
<td>32K</td>
<td>256</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>L2-D</td>
<td>512K</td>
<td>256</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>L3</td>
<td>32M</td>
<td>256</td>
<td>8</td>
<td>42</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td></td>
<td></td>
<td>150</td>
</tr>
</tbody>
</table>
Finite Resource CPI

![Finite Resource CPI Chart]

Finite Resource

Infinite Resource
ICache CPI

finite Resource

Infinite Resource
DCache CPI

The graph illustrates the CPI (Cycles Per Instruction) for various workloads: LI, M88K, IJPE, VORTEX, PERL, COMpress, GO, GCC, and TPC-C. The x-axis represents different workloads, while the y-axis shows the CPI values ranging from -0.1 to 0.7. The bars are color-coded to represent different resources:

- **Green** for DCache
- **Blue** for ICache
- **Yellow** for Finite Resource
- **Red** for Infinite Resource

The data indicates that different workloads have varying CPI values, reflecting the impact of resource availability on performance.
DTLB CPI

The graph illustrates the CPI (Cycles Per Instruction) for various benchmarks and tasks under different conditions.

- **DTLB**: Direct Translation Lookaside Buffer
- **DCache**: Data Cache
- **ICache**: Instruction Cache
- **Finite Resource**: Limited resources
- **Infinite Resource**: Unlimited resources

The x-axis represents different benchmarks and tasks:
- LI
- M88K
- IJPEF
- VORTEX
- PERL
- COMPRESS
- GO
- GCC
- TPC-C

The y-axis represents the CPI values ranging from -0.1 to 0.7. The bars show the CPI under different resource conditions.
Translation and Overall C

[Chart showing the comparison of different benchmarks across various categories: Translation, DTLB, DCache, ICache, ICache, Finite Resource, and Infinite Resource.]
Instruction Reuse

Reuse Rate = \frac{\text{# of Dynamic Ins in Trace}}{\text{# of Unique Ins Addresses in}}

- LI
- M8K
- IJPE
- VORTEX
- PERL
- COMPRESS
- GO
- GCC
- TPC-C
Window Size and ILP

![Bar chart showing Window Size and 10X Infinite ILP for various applications.](chart.png)
Instructions Booting AIX

- ICBI's
- Non-NOP ICBI's
- # of PPC
- PPC Ins Blks To
- PPC Ins per Blk

Occurrences

- Firmware
- AIX
- X-Windows
New Code Seen During Boot

- PowerPC Ins Executed
- 100,000 * PowerPC Blocks Seen

Groups Executed (Millions)

Counts

- Firmware Ends
- Boot Ends
ICBI’s Seen During Boot

Graph showing the counts of ICBI’s and 1000 * Non-NOP ICBI’s executed over time, with two distinct lines indicating Firmware Ends and Boot Ends.
Exceptions During Boot

- # of ISI Exceptions
  - Page Faults
  - Memory Protection
- # of DSI Exceptions
  - Page Faults
  - Memory Protection
- # of Alignment Exceptions
- # of Program Exceptions
  - Traps
  - Illegal
  - Privileged
  - IEEE Float
- # of System Calls

### Occurrences

- Firmware
- AIX
- X-Windows
Transmeta Crusoe
Apparent Crusoe Goals

- Full x86 compatibility, including all system, I/O and other code.
- Performance near x86 implementations from other vendors.
- Low Power.
Crusoe Schematic

x86 Applications

Windows, Linux, ...
BIOS

Code Morphing Software

Crusoe Hardware

CMS = VMM
Similarities to DAISY

- Full System Translation.
- VLIW designed specially for emulation.
- Interpretation and Translation Used.
Crusoe Processor

Up to 4 atoms (ops) per molecule

128-bit Molecule

FADD | ADD | LD | BRCC

Floating Point Unit | Integer ALU #0 | Load Store Unit | Branch Unit

In order pipeline

64 Registers

Working

Shadow

For x86 Use  For Crusoe Use
Based on diagram at http://www.sandpile.org/impl/pics/transmeta/tm5400/5400_diag.jpg
Crusoe Observations (1)

- **TM5400** has 8 KBytes of onchip local memory for data and 8 KBytes of onchip local memory for instructions.
- These presumably allow the most critical parts of CMS to remain resident for quick access and without perturbing x86 code and data.
- The high 16-way associativity of the L1 DCache acts to minimize conflicts between data used by the x86 code and that used by CMS.
Crusoe Observations (2)

- Memory controller integrated because part of the PC architecture.
- In DAISY, we found the PowerPC firmware powered off DRAM banks for testing during system startup.
- Since the DAISY VMM was in these DRAM banks, this power-off killed DAISY.
- By having the memory controller under Crusoe control, Transmeta should be able to avoid problems like this DRAM power-off.
**Additional Crusoe Processor Details**

- Same condition code format as traditional x86 processors.
- Same 80-bit floating point format as traditional x86 processors.
- 7 stage integer pipeline.
  - Pentium IV has a 20 stage pipeline.
- 10 stage floating point pipeline.
- TM5400 has about 7 million transistors.
  - Athlon has 22 million.
Crusoe Startup

512 KByte FlashROM CMS
Crusoe Startup

512 KByte FlashROM CMS

Decompress

RAM

2 MBytes of Crusoe Code
Crusoe Startup

512 KByte FlashROM
CMS

Decompress

RAM

2 MBytes of Crusoe Code

14 MBytes of Crusoe Data
Crusoe Startup

- 512 KByte FlashROM CMS
- Decompress
- RAM
- 2 MBytes of Crusoe Code
- 14 MBytes of Crusoe Data
- 8 KB Crusoe Local Ins Mem
- 8 KB Crusoe Local Data Mem
Crusoe Exception Handling (1)

Copy x86 regs to shadow when finish translated code fragment.

64 Registers

For x86 Use  For Crusoe Use
Crusoe Exception Handling (2)

Copy x86 regs from shadow when exception occurs in translated code fragment.

*E.g. Page Fault*

64 Registers

- For x86 Use
- For Crusoe Use
Crusoe Exception Handling (3)

<table>
<thead>
<tr>
<th>Gated Store Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**On exception**, purge and reset store buffer of entries made by current code fragment.

**Note:** Size of Gated Store Buffer limits number of stores in translated group.
Benefits of Transmeta’s Commit and Rollback

- Commit and rollback allows a wider variety of optimizations than DAISY — which can rollback only to the start of an individual VLIW instruction.
  - Removal of loop invariants.
  - Strength reduction.
  - Better dead code elimination.
Crusoe and Speculative Loads

(Alias Buffers)

AB

ldp (X)  
(Speculative Load)

Addr  Size

(Hardware)
Crusoe and Speculative Loads

Idp (X) → AB (Alias Buffer)

Addr | Size

(Store Under Alias Mask) stam (Y)

Alias Buffer

Hardware

Aliasing

Yes

Raise Exception to Crusoe VMM

No

Continue Execution
**Crusoe Speculative Loads**

- **Good:** No extra reload needed as with `load-verify`.
  - Reduces # of Instructions Executed.
  - Reduces Load/Store Bandwidth.

- **Bad:** Number of speculated loads limited by size of AB.

- **Bad:** False aliasing may occur:
  - Speculative load from untaken path aliased with store on taken path.
  - A false aliasing problem in straight line code?

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Translated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>st (W)</code></td>
<td><code>ldp (X)</code></td>
</tr>
<tr>
<td><code>ld (X)</code></td>
<td><code>ldp (Z)</code></td>
</tr>
<tr>
<td><code>st (Y)</code></td>
<td><code>stam (W)</code></td>
</tr>
<tr>
<td><code>ld (Z)</code></td>
<td><code>stam (Y)</code></td>
</tr>
</tbody>
</table>

**Question:** AB structure cleared on group transition?
Crusoe Speculative Loads

- Some concepts of Transmeta’s speculative load scheme appear similar to those in other approaches:
  
  - **U.S. Patent 5542067**: “Method and apparatus for improving performance of out of sequence load operations in a computer system.”
  
  - **Inventors**: Kemal Ebcioğlu, Manoj Kumar, Eric Kronstadt.

  - **U.S. Patent 5625835**: “Method and apparatus for reordering memory operations in a superscalar or very long instruction word processor.”
  
  - **Inventors**: Kemal Ebcioğlu, Dave Luick, Jaime Moreno, Gabriel M. Silberman, P.B. Winterfield.
Frequently Executed Code and Optimization

- “The first few times a specific x86 code sequence is executed, Code Morphing interprets the code . . .”
- “If [CMS has] been through an area 50 times, we figure we can optimize that and make it run faster.”
- “… different levels of optimization inside the code-morphing software.”
- “Code morphing translates the x86 instructions into highly optimized and extremely fast VLIW native instructions,
  - Executes the code, and
  - Caches the native instruction translations for future use.”
CMS Collection of Profiling Data

- Collect Profile Information about:
  - Most frequently executed blocks of code.
  - Whether code is running out of ROM or RAM.
  - Whether code is doing any I/O cycles.
Why does Crusoe Have Low Power Consumption?

- \( \text{LongRun}^{TM} \) power management software slows chip frequency \( (F) \) and voltage \( (V) \) when processing does not require full CPU.

- \( \text{Power} = \frac{1}{2} CV^2 F \)

- Instruction Scheduling that must be done continuously in an out-of-order superscalar done once in Crusoe.

- Optimization removes many operations such as condition code setting.

- Simple, in-order hardware. Complexity is in software.
Additional Transmeta Challenges

- **x86** instructions can cross page boundaries.
  - A translated code fragment may have to be invalidated if either page is changed or paged out.

- Partial register results, e.g. a write to 8-bit `ah` modifies the top 8 bits of `ax`.

- **x86** 16 bit code and segment registers.
Dynamo
Dynamo Similarities to DAISY and Crusoe

- Extensive Optimization.
- Interpretation and Translation Used.
- **Source Architecture** code unmodified.
Dynamo Goal

“... to use a piece of software to improve the performance of a native, statically optimized program binary, while it is executing.”

⇒ Dynamo is a JIT compiler from PA-RISC to PA-RISC.
Dynamo Characteristics

- “Written entirely in user-level software.”
- “Does not depend on any special programming language, compiler, operating system, or hardware support.”
- “The program binary is . . . left untouched during Dynamo’s operation.”
DAISY Operation

1. **Interpret Ins, Add to Grp X**
   - If Yes, go to **Y=X, X=New Grp**.
   - If No, go to **At Prev Translated Entry Pt**.

2. **At Prev Translated Entry Pt**
   - If Yes, go to **Execute Group’s VLIW Translation**.
   - If No, go to **Interp 30x**.

3. **Interp 30x**
   - If Yes, go to **Translate and Sched Group Y to VLIW Instruc**.
   - If No, go to **Stopping Point**.

4. **Stopping Point**
   - If Yes, go to **Y=X, X=New Grp**.
   - If No, go to **Interp 30x**.

**Execute Group’s VLIW Translation**

**Translate and Sched Group Y to VLIW Instruc**
Dynamo Traces (1)

- **Traces** are the unit of **Dynamo** optimization.

- A **trace** is a single path through the code, similar to **superblocks**.

- A **trace** may go through:
  
  - Indirect branches.
  
  - Function Calls and Returns.
  
  - Virtual Function Calls.
Dynamo Traces (2)

- Dynamo efficiently finds **Traces**.

  1. Look for good *start-of-trace points*, like targets of loop back edges.
  2. Look for “hot” frequently executed *start-of-trace points*.
  3. The chosen **trace** is the set of basic blocks that follow a *start-of-trace point* at the point it becomes “hot.”
  4. **MRET** = **Most Recently Executed Tail**.
Dynamo Optimizations (1)

- Two pass optimizer: **Forward** and **Backward**.
- Eliminate unconditional branches since *(Traces straightened)*.
- Remove **Call-Return** pairs in a trace.
- Copy propagation, **constant propagation**, strength reduction, **loop invariant code motion**, loop unrolling.
- Fragment linking, i.e. direct branching between translated groups.
  - $40\times$ improvement compared to always branching via the **Dynamo** software.
Dynamo Optimizations (2)

- Redundant Assignment Removal.
- Redundant load removal.
  - Since loads may be volatile, they are removed only when symbol table info is available guaranteeing safeness.
Pre-emptive Flushing

• Complex TCache management schemes can be time consuming.

• Dynamo looks for sudden jumps in trace creation rate.
  – Then the entire TCache is flushed.
  – Such flushing allows Dynamo to readapt to new code patterns.
  – It also allows for a quick reset of many data structures and memory pools without costly garbage collection.
Dynamo and Signals (1)

- **Dynamo** handles all signals.
- Signal handlers in the emulated program are translated by **Dynamo** like other code.

- **Question:** What happens if a program running under **Dynamo** installs its own signal handler at some random point?
Like DAISY, Dynamo waits for a code fragment to end before notifying the emulated program and its signal handler of the exception.

For synchronous signals, Dynamo proposes a scheme to use logs, so as to be able to backtrack and report precise PA-RISC state to the emulated signal handler.

However, the current implementation uses an aggressive / conservative scheme, where aggressive optimizations are turned off if precise exceptions are needed.

- These optimizations include dead code elimination, and removal of loop invariants.
**Dynamo Bail Outs**

- Systems like **DAISY** and **Crusoe** that translate to a different target architecture than source architecture cannot *bail out*.

- **Dynamo** can *bail out*.

- When **Dynamo** detects that translated fragments are being created at too high a rate, it does *bail out*.

- *Bailing out* limits the amount of slowdown that **Dynamo** can cause for a program.

- **Caveat:** After *bailing out*, **Dynamo** cannot regain control of a program.
Dynamo Performance
Dynamo Overhead

![Bar chart showing Dynamo Overhead for different benchmarks: m88k, li, perl, deltablue, ijpeg, go, vortex, and the average. The y-axis represents Dynamo Overhead (Percent) ranging from 0 to 1.2, and the x-axis lists the benchmarks. The bars indicate varying degrees of overhead, with 'li' and 'deltablue' showing higher values than others. The average overhead is also highlighted.]
Fragments in 90% Cover Set

- m88k
- compress
- li
- perl
- deltablue
- ijpeg
- go
- vortex

- go: 1038
Observations and Summary
Requirements of Target Architecture

- None – all of these are Turing machines
Desirable Properties of Target Architecture

- Opcodes similar to source architecture
- Condition code settings similar to source architecture
- Address translation similar to source architecture
- Floating point format similar to source architecture
- More registers than source architecture
- I/O system, mem controller similar to source architecture
- Memory map consistent with source architecture
- Able to hide part of real memory from source architecture for use by VMM
- Timers consistent with source architecture
- Load-Verify Instruction or other way to speculate loads.
Open Problems (1)

- Can a binary translation machine have generally better performance than a well-designed superscalar?
- Can all realtime problems be avoided?
- What memory management schemes are best?
- Transarently increasing or decreasing amount of memory for VMM after system booted.
Open Problems (2)

- Should the **Target** architecture ever be exposed for users to access directly (and bypass the source architecture layer)?
  - **Pro:** Avoid translation overhead.
  - **Pro:** Full compiler optimizer can be used.
  - **Con:** Difficult to upgrade/change underlying machine.
  - **Con:** Lose “under the covers” optimization and profile directed feedback.
  - **Con:** More complicated system with code for both **Source** and **Target** architectures.
Summary

- There are many approaches to dynamic translation and optimization.
- The technology is maturing and proving itself.
- Faster, better optimization is still needed.
DAISY Open Source

- A user-mode version of DAISY is now available as open source.
Timetable for Micro-33 Tutorial on

Dynamic Binary Translation and Optimization

Wednesday, December 13, 2000

2:30 - 2:50  Kemal Ebcioglu: Future Challenges
2:50 - 2:55  Erik Altman: DAISY Demo
2:55 - 3:20  Erik Altman: Binary Translation Issues
3:20 - 3:35  Break
3:35 - 5:00  Erik Altman: DAISY, Crusoe, Dynamo
5:00 - 5:15  Break
5:15 - 5:45  Kemal Ebcioglu: LaTTe