Dynamic Compilation at the System Level

Erik Altman
Michael Gschwind

IBM T.J. Watson Research Center
Context

Motivation and examples will generally be in a DAISY or BOA context.
– i.e. Co-Designed Virtual Machines

Many ideas are applicable to other forms of dynamic optimization.
The DAISY and BOA* Teams

- Erik Altman
- Arthur Bright
- Al Chang
- Kemal Ebcioglu
- Michael Gschwind
- Marty Hopkins
- Krishnan Kailas
- Steve Kosonocky
- Sumedh Sathaye
- Craig Agricola
- Dave Appenzelller
- Zak Filan
- Jay LeBlanc
- Paul Ledak
- Jason Fritts (co-op)

* Binary translation
  Optimized Architecture
Outline of Tutorial

- **8:00 – 8:15** Motivation
- **8:15 – 8:30** Background
- **8:30 – 8:45** Translation Groups
- **8:45 – 9:30** Dynamic Optimizations (DO)
- **9:30 – 10:00** Virtual Machines (VM)
- **10:00 – 10:30** Coffee Break
- **10:30 – 11:00** BOA Support for VM / DO
- **11:00 – 11:30** BOA Arch and µArch
- **11:30 – 11:50** BOA Performance
- **11:50 – 11:55** BOA - DAISY Comparison
- **11:55 – 12:00** Summary and Conclusion
Motivation
Why System Level Dynamic Compilation?

- Seamless Dynamic Feedback
- Cross boundary optimization scope:
  - Shared Libraries
  - Operating System
- 100% compatibility with existing ISAs

Can handle:
- Arbitrary entry points
- Self-modifying code
- Multi-threading
- etc.
Why System Level Dynamic Compilation?

- Hardware cracking consumes time and/or transistors.
- Microcode emulation limits exploitation of ILP.
- Software translation avoids these problems but requires high instruction reuse.
  - Most apps have high reuse.
Why BOA / DAISY?

- Out of order superscalar processors achieve high performance
- ... But at the cost of high hardware complexity
  - Predictors
  - Complex decode
  - Complex issue queues with wakeup and issue logic
  - Register mapping tables
  - ...

Why BOA / DAISY?

- Out of order superscalar processors achieve high performance

... But at the cost of high power

- Many out of order components operate every cycle.
- Many components query a large set of data to operate on a single element.
- Same set of operations performed to get the same results.
Why BOA / DAISY?

- **Out of order superscalars achieve high performance**
- ... But at the cost of **deep pipelines**
  - Complex logic has long latency.
  - To achieve high frequency with long latency, super pipelining is required.
  - Deep pipelines require excellent branch predictors.
  - Excellent branch predictors are complex.
  - Complex logic has long latency ...
Why BOA / DAISY?

- Out of order superscalar processors achieve high performance

- **But at the cost of** high verification and debug complexity
  - schedule slips \(\Rightarrow\) performance slips

Moore's Law in action

<table>
<thead>
<tr>
<th>Schedule Slip</th>
<th>Relative Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 month</td>
<td>4%</td>
</tr>
<tr>
<td>3 month</td>
<td>12%</td>
</tr>
<tr>
<td>6 month</td>
<td>26%</td>
</tr>
<tr>
<td>9 month</td>
<td>41%</td>
</tr>
<tr>
<td>12 month</td>
<td>59%</td>
</tr>
<tr>
<td>18 month</td>
<td>100%</td>
</tr>
</tbody>
</table>
What does BOA / DAISY offer?

**Software Dynamic Optimization**

- Adapt code to dynamic runtime behavior
  - Scheduling
  - Optimization
  - Speculation

- Focus hardware design on fast execution
  - Reduce hardware complexity
  - Simpler logic ⇒ Faster logic
  - Less logic ⇒ Less power
Simple(r) Architecture

Good Architecture

Reduce hardware complexity

- But no high performance general purpose processor will ever be “simple”.
- Dynamic optimization allows some reduction in complexity.

BOA is simpler than DAISY in many ways:

⇒ Focus mostly on BOA
What BOA Offers Compilers

*Note:* Compilers ↔ Dynamic Optimizers

- Simple, orthogonal architecture
- Large register set
- Seamless Dynamic Feedback
- Cross boundary optimization scope:
  - Shared Libraries
  - Operating System
What BOA Offers Architects

- Shorter pipelines for same frequency.
- Fewer hardware predictors.
- Simpler issue logic.
- Less power by eliminating repetitive steps
  - E.g., Crack and Schedule
- Less debug and verification.
  - At least for the hardware component...
- Smaller chips and higher yield.
Background
BOA / DAISY System

L3 Cache

DAISY
VLIW

6xx Bus

Memory Controller

PCI Bus

PowerPC

DAISY

DAISY Flash ROM

PowerPC Flash ROM

Disk

Video

Network

Keyboard
Booting a BOA system

1. Reset starts executing in BOA Boot Flash
2. Initialize BOA environment
   - Stack, heap, translation cache, internal data structures…
3. Start compiling, then executing PowerPC boot ROM code at PowerPC reset address (0xFFF00100)
4. …eventually transfers to boot loader and causes it to be translated,
5. …loads OS, transfers control to OS, and causes it to be translated
6. …loads applications, transfers to apps, and causes apps to be translated
Boot a BOA system

- Simple in concept, harder in practice.
- **E.g. debugging:**
  - First part of firmware decompresses later part of firmware.
  - Later part of firmware is actually a FORTH interpreter.

- Debugging BOA is 3 levels removed from semantic actions being taken:
  - Discovering devices
  - Checking system integrity
Boot Time Oddities

- Self-check of memory turns off memory banks (via writes to I/O ports).
- Memory bank with BOA system code disabled.
  - BOA dies.
  - Moral: Must virtualize the memory controller, not just the processor to properly emulate system behavior.
Boot Time Oddities

- Frame buffer expects 4-byte stores.
- Code accessing frame buffer uses PowerPC `stswi` (store-string) instruction.
  - Cannot emulate `stswi` as sequence of store-byte instructions.
    - Machine dies with a bus error.
    - Such errors can be hard to isolate. (We know!)
Boot Time Oddities

- Redundancy can make debugging more difficult.
- **Example**: AIX boot sequence uses 3 techniques to establish IP address of machine.
- If any technique succeeds, machine gets an IP address.
- Bugs in one or two of the techniques can *(and did)* go undetected.
  - And were much harder to find later.
ICBI Instructions

In PowerPC, when instructions are modified, must execute:
- ICBI (Instruction Cache Block Invalidate) instruction

BOA uses ICBI as signal to invalidate translations:
- Must be able to efficiently invalidate
  - By age (if translation cache full)
  - By address (for ICBI instructions)

Little self-modifying code in PowerPC, but ICBI also used:
- During program load.
- For JIT-compiled code.

To be safe, AIX executes ICBI for all blocks on a page whenever a new page is loaded.
- Many NOP ICBI instructions executed.
- Reduce BOA VM overhead by fast check for NOP ICBI instructions.
ICBI Occurrences
Interrupt/Exception Frequencies
BOA / Dynamic Compilation
System Architecture

- **interpet insn X (PowerPC)**
- **update statistics**
- **goto next insn X**

- **seen X 15 times**
  - **form group @ X and translate PowerPC to BOA**
  - **X prev translated entry Pt**

- **execute group @ X BOA translation**

**No**

- **Yes**
Basic Dynamic Compilation Loop

Start

- Interpretation (optional)
- Compile Code Segment (Function, BB, etc)
- Translation Cache
- Transfer to Untranslated Code
- Execute group
- Store Translation Group in Translation Cache
DAISY / BOA Additions

Start

- LookupInsn Address
  - hit
    - Transfer to Untranslated Code
    - Branch to Translated Group
  - miss
    - Interception (optional)
    - Exception Handler

- Compile Code Segment (Function, BB, etc)
  - yes
    - Store Translation Group in Translation Cache
  - no
    - Unstructured binaries

- Translation Cache
  - Execute group
Unstructured binary code

- HLL dynamic compilation usually has defined code body and “natural” compilation units
  - Function, method, class,…

- Dynamic compilation of binary code performs dynamic code discovery:
  - Identify meaningful translation units
  - Prevent “rediscovery” and duplication of already translated code
Exception handling

Many instructions can raise exceptions
- Usually invisible to user-code
  - Small number can be reflected using UNIX signal() facility

Exceptions provide “invisible” control flow arcs from code
- Restricts optimizations across these arcs
- Infrequent
  - but must be handled correctly for system function
- Must be able to reconstitute state for exception handler
  - And provide exception code, registers, …

Exceptions vs. Interrupts
- Synchronous vs. Asynchronous
**Full system aspects**

- Large number of synchronous exceptions poses a significant burden on full system compilation
  - Many instructions can raise synchronous exceptions:
    - **Memory ops** *(page faults, protection)*
    - **Floating point** *(IEEE compliance)*
    - **Divide by zero**
    - **Control flow across pages** *(page faults, protection)*
    - ... 

- Number of events dynamically low
  - And not degrading common case is necessity for good performance
  - But when events occur they need to be handled in compliance with architecture.
System safety and correctness

Instruction execution always under VMM control.  
- Locus of control can be in:  
  - VMM Interpreter  
  - VMM Translator  
  - VMM Exception manager  
  - VMM Memory manager  
- Locus of control can be in VMM-generated traces.  
  - Traces transfer control only to each other,  
  - Or to VMM.  
- No way to inject native (i.e., uncontrolled) code into the system from the PowerPC layer

Protection is a combination of hardware primitives, and VMM-generated code
Translation Groups
Translation group structure

Multiple choices for structure of a translation group:

- Multiple-entry, multiple-exit block
  - Mimic [May ’87]

- Tree: Single entry, multiple exit
  - DAISY [ISCA ’97], [Europar ’99], [MICRO ’99]

- Trace: Single entry, multiple exit
  - BOA [WBT-99] (similar to superbblocks, or hardware trace cache)

- Atomic blocks
  - BOA study [WCED ’02]
Translated Code Size over Time: DAISY Tree Regions
BOA Group Formation

- Include ops from only a single path.
- Always follow most likely direction of conditional branch.
- If necessary, invert branch conditions so all branches expected to fall-thru.
- Optionally go thru register branches:
  
  ```
  cmpi cr15, PPC_LR, 0x1234
  bne EXIT_GROUP
  # Translated Code from 0x1234
  ...
  EXIT_GROUP: b BOA Syscode
  ```
Stopping Points

- Identify possible cut points in translation

Reduce code duplication
- By quantizing translation stop/start to a set of well-defined entry/exit points to translations
- Good choice of stopping point increases effectiveness by exploiting program structures
  - Branch, function call, function exit, …
  - Likely program join points map to group start
    - Translation groups can only be entered at top
    - Translated code only has join points at translation group boundary

“Hard stops”
- Stopping point & stopping condition
- Resource limits
Avoid enumerating all possible substrings of the dynamic execution

Stopping Points
Stopping Points

Stopping points allow exploitation of program structure
– Statistically, without detailed (i.e., expensive) analysis

mtctr 5
L..9:
  add 3,3,0
  subfc 0,0,3
  bdnz L..9
blr

or:
mtctr 5
add 3,3,0
subfc 0,0,3
bdnz L..9

mtctr 5
add 3,3,0
subfc 0,0,3
bdnz L..9

add 3,3,0
subfc 0,0,3
bdnz L..9

add 3,3,0
subfc 0,0,3
bdnz L..9

add 3,3,0
bdz L..9
add 3,3,0
subfc 0,0,3
bdnz L..9
Group Stopping Conditions

- Fickle branch
  - E.g., go one direction less than 60% of time
  - Termed Bias-9 ⇒ One way 9/15 times
    - Bias is a major knob to control group size

- # of ops in group > Threshold (60 ops)
- # of stores in group > Store Buffer Size
  - 32 entries in store buffer

Register Branches (optionally)
“Poor man’s path profiling”

- Limit data per profile:
  - No program structure information, as in advanced profile collection methods.

- For each branch, collect profile information:
  - Profile \((\text{prev\_branch\_address}, \text{this\_branch\_address})\) \(\leftrightarrow\) branch outcome

- Synthesize path information from 1 block deep path information
Group Quality

Group quality impacts performance

- Many ways to quantify "quality"
  - Fraction of early exits from a group?
  - Average \textit{static} length of group?
  - Average \textit{dynamic} length of group?
    - Combines fraction and location of early exits, and average static length of group
We find dynamic group length to be a good measure of quality.
Group Length vs CPI

Dynamic Group Length

Infinite Resource PowerPC CPI
Different conclusions published by others:

- Results depend on dynamic optimizations exploited:
  - Code Packing, Hot/Cold optimizations effective across group boundaries
    - Dynamo derives significant benefit from these

- ILP optimizations depend on speculating from correct path
  - BOA, DAISY exploit ILP optimizations
  - ILP exploitation limited on HP-PA
Group Length vs CPI: Theory

- ILP is monotonically non-decreasing with group length:
  - Consider $N$ consecutive instructions on some path.
  - Split these $N$ instructions into two groups, $G_1$ and $G_2$, each with $N/2$ instructions:
    - Any pair of schedules derived by scheduling $G_1$ and $G_2$ separately can also be derived by scheduling all $N$ instructions together.
      - Additional schedules are also possible when scheduling all $N$ instructions together.
      - Thus, ILP with a larger window is always at least as high as with two smaller windows.
A larger group size loses only if other penalties come into play:

- ICache pollution from duplicated code
- ITLB pollution from duplicated code
- Frequent translation cache overflows / flushes
- Roll back penalties
  - Discard useful work done
  - Potentially re-execute more slowly.
Percentage of Groups Completing All Instructions

- li
- perl
- m88k
- go
- jpeg
- vortex
- gcc
- compress
- tpcc
No / negative correlation between early group exits and performance
Group Execution Alternatives

- **Atomic**: All operations in a group execute or no operations in a group execute
  - From architecture point of view.

- **Incremental**: Architected State of Emulated ISA state updated incrementally in original program order as group executes.
  - DAISY Approach

- **Combination**: All operations in group up to point of exit update ISA state simultaneously:
  - May exit group “early”
    - Due to group not containing branch path taken in execution
    - Due to exception, e.g., page fault.
  - BOA Approach
Atomic Blocks

Published at [WCED ’02]

Atomic blocks put premium on group completion.

Allows far more aggressive optimization.

But the penalties of early exit are substantial:
- A rollback occurs, and all work must be discarded
- Slow mode is entered, with less aggressive groups (essentially basic blocks)
Benefits of *Atomic* and *Combination Group Execution*

- **Aggressive memory re-ordering**
  - Ability to resolve wrong assumptions about dependences
  - Ability to maintain correct MP behavior in view of changed memory access ordering

- **Resolves precise exception challenges**
  - All-or-nothing
  - No hidden exception arches…
  - Re-ordering of possible exceptions

- **Requires hardware support**
  ⇒ See later
Block-Structured CPI
PowerPC code for a program is typically broken into several traces, some of which may overlap, as with traces 1 and 3.

L1: add r1, r2, r3  
    subf r4, r5, r6  
    beq cr0, L1     # 10% Taken  

L4: xor r7, r8, r9  
    lw r10, 0(r11)  
    bne cr1, L2     # 90% Taken  
    ..               # Ops not in Trace 1, 3  
L2: and r12, r13, r14  
    b L3

L3: stw r15, 0(r16)  
    ...

BOA Trace 1
L1 - b L3

BOA Trace 3

BOA Trace 2
Code duplication

Code duplication due to group formation

- Same static instruction may be in multiple groups:
  - Group formation effectively performs tail duplication, inlining, unrolling, …

- Incremental code discovery and translation limits control of this duplication.

- Generated code can grow significantly compared to original code size.
Code Packing

- **Code packing:**
  - Within a group
  - Across groups

- **Within group:** Software Based Trace Caching
  - Application-directed code compaction
  - Similar concept to hardware trace cache
  - Much simpler to implement

- **Across group:**
  - Statistical Pettis-Hanson, Hot/Cold
  - Increase effectiveness of ICache and ITLB
  - Very helpful in HP Dynamo performance
Effects of Code Packing on Working Set: HP Dynamo Performance

Code packing implicitly performed as part of DO exploits instruction cache and TLB more efficiently.
Dynamic Optimization and Reoptimization
Code Re-optimization

- Reduce compile time with tiered compilation
- Translated code can be re-optimized
  - If "very hot" regions are identified, to optimize more aggressively
    - DAISY does this [ISCA ’97, Micro ’99]
  - To avoid frequent upsets and recovery cost
    - Misspeculation, code modification events,…
    - Transmeta Crusoe implements this [CGO ’03]

- Trigger re-translation using
  - Profiling support in DO target
  - In recovery code
Interpretation and Profiling

Experiments with interpretation and profiling to
- increase “translation quality”
- reduce translation cost
- code expansion

Interpret $i$ times and profile
- form group if interpreted $i$ times
- extend group beyond branch if branch shows bias of $\frac{n}{i}$
- explore different values for $n, i$
- problematic to form paths based on branch profiles
- path profiling potentially expensive
  - “poor man’s path profiling”
Statistics

- For each *conditional branch*, keep *taken* and *fall-thru* counts.

- For each *register branch*, keep list of register target values.

- Could keep load values for *value prediction*.
Profile-Based Improvements

Idea: Limit code bloat & compile time by confining aggressive optimizations to frequently executed regions of the code.

Idea explored in DAISY context [ICS’00].

Details in next slide …
Optimizing Frequently Executed Code

- Initial group formation
  - ILP goal is modest, and window size is conservative
  - e.g., 3 IPC infinite, 24 operations window

- Group modification due to re-execution
  - Profiling to determine reuse
    - Hardware/software counter-based scheme *(Details later).*
    - Timer-based profiling scheme
  - Two growth mechanisms
    - Multipath branch append
      - Frequently taken branch compiled into initial group as multipath
    - Group reoptimization
      - Recompile block exit point with more aggressive goals
      - e.g., 10 IPC infinite, 250 operations window size
Group Extension by Multipath Append

Add path to an existing group
Group Extension by Re-Optimization

Extend frequently executed existing group to increase ILP
Quasi-Static BOA Optimizations

- Instruction Scheduling
- Combining
- Copy Propagation
- Dead Code Elimination
- Code packing
- Load-Store Telescoping
- Register Port arbitration
- Replace “hard” ISA ins with ops that are easier to schedule / execute
- Improve predictability of execution path by code layout
Loop Index ➔ Sequentialize Iterations

```c
for (i = 0; i < N; i++) {
    a[i] = b[i] + c[i];
}
```

```
L1:  lbz          r4,b_offset(r3)   # Assembly code
     lbz          r5,c_offset(r3)
     add          r6,r4,r5
     stb          r6,a_offset(r3)
     addi         r3,r3,1           # r3 dependence
     cmpwi        cr0,r3,N          # can serialize code
     bne          cr0,L1
```
Combining to the Rescue

- Combine increment of loop index variable with dependent ops in subsequent iterations, e.g. 2\textsuperscript{nd} iteration:

\begin{verbatim}
.lbz r63,b_offset+1(r3)
.lbz r62,c_offset+1(r3)
.add r61,r63,r62
.stb r61,a_offset+1(r3)  # Wait till non-specul
.addi r63,r3,1           # to store
.cmpwi cr31,r3,N-1
.beq cr31,Loop_Exit
\end{verbatim}

- Now first and second (and later) iterations can execute in parallel – subject to resource constraints.
Load-Store Telescoping

- **xor**  
  - r31, r9, r4

- **stw**  
  - r31, 8(r1)

- ...  

- **lwz**  
  - r7, 8(r1)

- ...  

- **stw**  
  - r7, 64(r22)

- ...  

- **lwz**  
  - r14, 64(r22)

- **addi**  
  - r20, r14, 6

Telescope loads and stores together

- Can execute **addi** one cycle after **xor**

Such telescoping patterns common in function prologs and epilogs.

- Save/Restore callee/r saved registers.
Dynamic Optimizations

- Memory op speculation based on observed runtime dependence behavior
- Smarter group formation based on discovery of actual paths
  - Tree, superbloc, other...
- Value prediction based on observed data behavior
- Lightweight optimizations [Micro '99]
Dynamic Optimization Example

First Path Seen

\[ x = a \]
\[ *x = b \]
\[ c = *x \]

Copy propagation yields two independent operations which can be scheduled in the same VLIW.

Trace formation eliminates join point

Second Path Seen

\[ x = a \]
\[ c = *a \]

Load/store telescoping eliminates dependencies through memory. Copy propagation of \( c = b \) can open even more scheduling opportunities.
System Level Optimization Challenges

- Dynamic compilation at system level needs to be transparent
  - Compatibility guarantee
  - Same results as unoptimized original binary

- Microprocessor mechanisms defy many traditional optimizations
  - Analysis scope limited
    - Runtime limited
    - Dynamic code discovery

- e.g., liveness analysis
Precise Exceptions and Dynamic Optimization

Microprocessors usually offer precise exceptions to handle special conditions
- Occurs in all forms of binary compilation
  - Process-level ➔ `signal()` interface
  - System-level ➔ full range of architected exceptions
    - Demand paging
    - Divide by zero
    - Floating point

Must preserve semantics in presence of exceptions
- Machine state observability in unexpected locations
- Disabling optimizations will degrade performance
Detailed Example: Dead Code Elimination
Dead Code Example

Example Code Sequence

- (1) add r4, r3, r4 # DEAD!
- (2) lwz r3, 0(r9)
- (3) add r4, r3, r3

But a page fault at (2) lwz makes the dead value of r4 visible to the exception handler.

If the handler bases any actions on the value of r4, the program may fail.
Approaches to dynamic compilation in the presence of exceptions

- Severely restrict dead code elimination.
- Include a **safe mode** which disables “unsafe” optimizations.
- Rollback to a good state and interpret original code until exception is found.
- Improved code generation with dynamic state recovery \([WBT2000, CC2002]\)
Limiting dead code elimination

- Compute all dead results
- Commit results in-order

Used in DAISY [ISCA1997]
- high-ILP architecture
- excess operations have less performance impact
- dead results eliminated in scope of single atomic VLIW
- on exception, rollback to beginning of VLIW
Safe mode

- Safe mode uses only conservative optimizations
- Use safe mode to translate critical programs or program regions
- Critical code
  - detected by heuristics
  - specified by human intervention
- Heuristics and humans can be wrong
  - Incorrect execution if too aggressive
  - Performance degradation if too conservative
- Used in DYNAMO [HP1999]
Rollback to checkpoint

- Take checkpoints on group transitions
- Aggressively optimize within translation groups
- On exception,
  - rollback to checkpoint
  - then interpret original binary conservatively
- Rollback requires backing out of processor state and memory state changes
  - special, complex hardware required
  - memory rollback complex in MP
- Used in Transmeta, BOA [Computer2000]
Deferred State Materialization for Dynamic Optimization

- Optimize for common performance case
  - aggressive dead code elimination
    - keep enough state to materialize full state when exceptions occur

- State recovery to provide correct in-order state for exception processing
  - dead values materialized only when exception occurs
    - exceptions occur infrequently
    - modest cost for materializing full state

- Maximum performance during program execution
State Repair Concept

Original CFG

add r4, r3, r4
lwz r3, 0(r9)
add r4, r3, r3
add r4, r3, r4
exception handler

Improved CFG

***
lwz r3, 0(r9)
add r4, r3, r3
add r4, r3, r4
exception handler
Precise exception framework

- DAISY-like dynamic compilation environment

- Unit of operation is tree region
  - corresponds well to the mechanics of dynamic compilation
  - keeps algorithms simple $O(n)$ since no $\phi$ nodes

- FG in single static assignment form
  - simplifies overall algorithm
  - in particular, simplifies handling live ranges
Algorithm steps

- Tag instructions computing dead results (excl. exceptions)

- Tagged instructions will not be emitted into generated code
  - keep around as meta data ("repair notes")
  - could recompute meta data on demand
    - algorithm is deterministic

- Ensure that all state can be recomputed
  - by keeping information about elided instructions
  - by keeping inputs to elided instructions alive
    - until point where elided instructions are killed
    - this can increase or decrease register pressure
Live Range Analysis

A register is dead if
- (1) it is no longer referenced by actual instructions
- (2) elided instructions that reference it are dead (w.r.t. exceptions)

Liveness of one symbolic register $o$ can influence liveness of other registers $i$
- if register $o$ is not materialized immediately
- if registers $i$ are needed to materialize it

Represented by liveness equivalence
\[ s_i \equiv <s_j, s_k> \]
- if $s_i$ is live, then $s_j$, $s_k$ are live
- Algorithm significantly simplified by SSA
Algorithm steps

1. **foreach** operation *op*
2. **if** dead (target (*op*))
3. convert2repairnote (*op*)
4. **foreach** instruction killing target (*op*)
5. insert_use (target (*op*))
6. insert_equivalence (target (*op*)) ≡ sources (*op*)

Liveness analysis performed *before* algorithm
Register allocation performed *after* algorithm
Example: PowerPC Code

and. r4,r3,r4
lwz r3,0(r9)
add r4,r3,r3
addi r5,r3,80
lwz r3,0(r10)
addi. r5,r3,1
### Example: Intermediate Representation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>and. r4,r3,r4</code></td>
<td>Logical AND</td>
<td>$1. s4' = s3 &amp; s4$</td>
</tr>
<tr>
<td><code>lwz r3,0(r9)</code></td>
<td>Load Word Zero</td>
<td>$2. sc0' = (s3 &amp; s4) \text{ cmp } 0$</td>
</tr>
<tr>
<td><code>add r4,r3,r3</code></td>
<td>Add</td>
<td>$3. s3' = [s9]$</td>
</tr>
<tr>
<td><code>addi r5,r3,80</code></td>
<td>Add Immediate</td>
<td>$4. s4'' = s3' + s3'$</td>
</tr>
<tr>
<td><code>lwz r3,0(r10)</code></td>
<td>Load Word Zero</td>
<td>$5. s5' = s3' + 80$</td>
</tr>
<tr>
<td><code>addi. r5,r3,1</code></td>
<td>Add Immediate</td>
<td>$6. s3'' = [s10]$</td>
</tr>
<tr>
<td>$s3'' = s3'' + 1$</td>
<td></td>
<td>$7. s5'' = s3'' + 1$</td>
</tr>
<tr>
<td>$s3'' = s3'' + 1$</td>
<td></td>
<td>$8. sc0'' = (s3'' + 1) \text{ cmp } 0$</td>
</tr>
</tbody>
</table>
Intermediate Representation after Basic Algorithm

{  s4'  =  s3 & s4  }
{  sc0'  = (s3 & s4) cmp 0  }
  s3'  = [s9]
  s4'' = s3' + s3'
use s4' ;  s4' ≡ <s3, s4>
{  s5'  = s3' + 80  }
  s3'' = [s10]
  s5'' = s3'' + 1
use s5' ;  s5' ≡ <s3'>
  sc0'' = (s3'' + 1) cmp 0
use sc0' ;  sc0' ≡ <s3, s4>
Some observations

- Overly conservative
  - only need to materialize state if a synchronous exception can actually happen
  - only need to be able to materialize until the last synchronous exception which can observe state on any given path

- Reduce number of repair notes

- Reduce register pressure
  - by killing otherwise dead input registers to repair notes
Improvement potential

{ s4' = s3 & s4 }
{ sc0' = (s3 & s4) cmp 0 }
s3' = [s9]
s4'' = s3' + s3'
use s4' ; s4' \equiv <s3, s4>
{ s5' = s3' + 80 }
s3'' = [s10]
s5'' = s3'' + 1
use s5' ; s5' \equiv <s3'>

{ s5' = s3' + 80 }
s3'' = [s10]
s5'' = s3'' + 1
use sc0' ; sc0' \equiv <s3, s4>

sc0'' = (s3'' + 1) cmp 0
use sc0' ; sc0' \equiv <s3, s4>

{ s4' = s3 & s4 }
{ sc0' = (s3 & s4) cmp 0 }
s3' = [s9]
s4'' = s3' + s3'
use s4' ; s4' \equiv <s3, s4>
{ s5' = s3' + 80 }

s5'' = s3'' + 1
sc0'' = (s3'' + 1) cmp 0
use sc0' ; sc0' \equiv <s3, s4>
Synergistic with Other Optimizations

- Code Sinking
- Unspeculation
- Constant Propagation
- Constant Folding
- Commononing
Example: Application to other optimizations

Code sinking

- $s_5 = s_2 + 2$
- $s_4 = s_3 / s_2$
- $s_5' = s_2 + 2$

$\{ s_5 = s_2 + 2 \}$

$s_4 = s_3 / s_2$

$s_5' = s_2 + 2$

use $s_5$; $s_5 \equiv <s_2>$

Constant propag.

- $s_8 = 16$
- $s_9 = s_7 / s_8$
- $s_8 = ...$

$\{ s_8 = 16 \}$

$s_9 = s_7 / 16$

$s_8 = ...$

use $s_8$; (extraneous)

$s_8 = ...$
Example:
Application to other optimizations

Commoning

\[
\begin{align*}
s_5 &= s_2 + s_4 \\
s_7 &= [s_5+10] \\
s_9 &= s_2 + s_4 \\
s_8 &= [s_9+20] \\
s_9 &= \ldots
\end{align*}
\]

\[
\begin{align*}
s_5 &= s_2 + s_4 \\
s_7 &= [s_5+10] \\
s_9 &= s_2 + s_4 \text{ (dead)} \\
s_8 &= [s_9+20] \\
s_9 &= \ldots
\end{align*}
\]

\[
\begin{align*}
s_5 &= s_2 + s_4 \\
s_7 &= [s_5+10] \\
\{ \quad s_9 &= s_2 + s_4 \quad \} \\
s_8 &= [s_5+20] \\
\text{use } s_9 \quad ; \quad s_9 &\equiv <s_2,s_4> \\
s_9 &= \ldots
\end{align*}
\]
Emitted Code and Recovery Information

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>lwz R32,0(R9)</td>
<td>[r3 := R32]</td>
</tr>
<tr>
<td>0x04</td>
<td>add R3,R32,R32</td>
<td>[r4 := R3]</td>
</tr>
<tr>
<td>0x08</td>
<td>lwz R33,0(R10)</td>
<td>[r3 := R33]</td>
</tr>
<tr>
<td>0x0C</td>
<td>addi R5,R33,1</td>
<td>[-unchanged-]</td>
</tr>
<tr>
<td>0x10</td>
<td>cmpi CR0,R5,0</td>
<td>[-unchanged-]</td>
</tr>
</tbody>
</table>

S0 = R3 & R4
SC0 = (R3 & R4) cmp 0

[r4 := S0; cr0 := SC0]
S0 = R3 + 80

[r5 := S0; cr0 := SC0]
Evaluation

Detailed results and analysis in [CC2002]

Reduction in number of operations

- Aggressive group formation increases optimization potential
- Approach only deletes dead code within group
  - Avoid $\phi$ nodes and control flow merge
Power PC

% ops eliminated

Key: (a) aggressive, (c) conservative group formation
zSeries

% ops eliminated

Key: (a) aggressive, (c) conservative group formation
Complications in full system translation
- All observable state must match original architecture \textit{at any time}
- Full data flow analysis not possible
  - Exceptions represent potential control flow transfers
  - Synchronous exceptions are problematic (page faults, IEEE FP, ...)

Use variant of deferred materialization
- Traditional deferred materialization is no win because too expensive
  - Executes instructions at runtime to record information about elided ops
- Deferred materialization at the translation group (superblock) level
  - \textit{Statically} record information about elided operations
  - Extend live ranges of input operands to include live range of elided result
  - If exception is raised, recreate result
    - Cost of recovery only burdens infrequent exception case
    - Mainline code executes at full speed
Virtualization
Definition of Virtualization

Virtualization is the efficient emulation of an architecture
- A machine may virtualize itself or another.
- Virtualization normally includes all system state, not just user state.

Certain ISA features simplify virtualizing a machine on itself:
- E.g., Not being able to view system state while in user state.
History of Virtualization

Virtualization has a long history, e.g.:

- IBM 360 VM
- Goldberg, CACM 1972 (Formal Definition)
- DAISY (*Architecture as a layer of software*)
- Transmeta
  - Jim Smith and Ravi Nair call DAISY and Transmeta “Co-Designed Virtual Machines”
  - This tutorial terms them “DAISY Hosts”
- VMware virtualization of x86
- Synthetic instruction set
  - C Virtual Machine CVM (IBM) [Proc. IEEE ’01]
  - Virtual Instruction Set Comp. VISC (Adve et al.)
VM Environments

- Abstract execution environment
  - Portable runtime environment
    - Smalltalk, Java VM, …

- Process abstraction
  - Idealized virtual memory
  - Fewer difficult system/kernel code issues

- System-level
  - No modifications to operating system
  - More transparent
    - Less danger of compatibility issues
Why Virtualization is Used in Architecture

- Better Performance
- Portability:
  - VM as virtual execution environment
- Migration
- Architecture Simulation Environment
<table>
<thead>
<tr>
<th>Technique</th>
<th>OOO</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISA</td>
<td>Base ISA</td>
</tr>
<tr>
<td>General Optimizations</td>
<td>Too complex</td>
</tr>
<tr>
<td>Path-Predictive Fetching</td>
<td>IFetch Prediction</td>
</tr>
<tr>
<td>Code Compaction</td>
<td>Trace Cache</td>
</tr>
<tr>
<td>Select Insns to Issue</td>
<td>Wakeup/Select Logic</td>
</tr>
<tr>
<td>Precise Exceptions</td>
<td>Register Renaming</td>
</tr>
<tr>
<td>Complex Insns</td>
<td>Decoder Cracks</td>
</tr>
<tr>
<td>Form Issue Groups</td>
<td>Select Logic</td>
</tr>
</tbody>
</table>
## DO / Virtualization & Architecture Styles

<table>
<thead>
<tr>
<th>Technique</th>
<th>OOO</th>
<th>DO+OOO</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISA</td>
<td>Base ISA</td>
<td>Base ISA</td>
</tr>
<tr>
<td>General Optimizations</td>
<td>Too complex</td>
<td>DO Optimizes</td>
</tr>
<tr>
<td>Path-Predictive Fetching</td>
<td>IFetch Prediction</td>
<td>DO Improves Prediction</td>
</tr>
<tr>
<td>Code Compaction</td>
<td>Trace Cache</td>
<td>DO Performs Layout</td>
</tr>
<tr>
<td>Select Insns to Issue</td>
<td>Wakeup/Select Logic</td>
<td>Wakeup/Select Logic</td>
</tr>
<tr>
<td>Precise Exceptions</td>
<td>Register Renaming</td>
<td>Register Renaming</td>
</tr>
<tr>
<td>Complex Insns</td>
<td>Decoder Cracks</td>
<td>Decoder Cracks</td>
</tr>
<tr>
<td>Form Issue Groups</td>
<td>Select Logic</td>
<td>Select Logic</td>
</tr>
</tbody>
</table>
# DO / Virtualization & Architecture Styles

<table>
<thead>
<tr>
<th>Technique</th>
<th>OOO</th>
<th>DO+OOO</th>
<th>DO+IO</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ISA</strong></td>
<td>Base ISA</td>
<td>Base ISA</td>
<td>Base ISA</td>
</tr>
<tr>
<td><strong>General Optimizations</strong></td>
<td>Too complex</td>
<td>DO Optimizes</td>
<td>DO Optimizes</td>
</tr>
<tr>
<td><strong>Path-Predictive Fetching</strong></td>
<td>IFetch Prediction</td>
<td>DO Improves Prediction</td>
<td>DO Improves Prediction</td>
</tr>
<tr>
<td><strong>Code Compaction</strong></td>
<td>Trace Cache</td>
<td>DO Performs Layout</td>
<td>DO Performs Layout</td>
</tr>
<tr>
<td><strong>Select Insns to Issue</strong></td>
<td>Wakeup/Select Logic</td>
<td>Wakeup/Select Logic</td>
<td>DO adapts at Exec Time</td>
</tr>
<tr>
<td><strong>Precise Exceptions</strong></td>
<td>Register Renaming</td>
<td>Register Renaming</td>
<td>SW Recovery Code</td>
</tr>
<tr>
<td><strong>Complex Insns</strong></td>
<td>Decoder Cracks</td>
<td>Decoder Cracks</td>
<td>DO or HW</td>
</tr>
<tr>
<td><strong>Form Issue Groups</strong></td>
<td>Select Logic</td>
<td>Select Logic</td>
<td>Issue Logic</td>
</tr>
<tr>
<td>Technique</td>
<td>OOO</td>
<td>DO+OOO</td>
<td>DO+IO</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>--------</td>
<td>----------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>ISA</td>
<td>Base ISA</td>
<td>Base ISA</td>
<td>Base ISA</td>
</tr>
<tr>
<td>General Optimizations</td>
<td>Too complex</td>
<td>DO Optimizes</td>
<td>DO Optimizes</td>
</tr>
<tr>
<td>Path-Predictive Fetching</td>
<td>IFetch Prediction</td>
<td>DO Improves Prediction</td>
<td>DO Improves Prediction</td>
</tr>
<tr>
<td>Code Compaction</td>
<td>Trace Cache</td>
<td>DO Performs Layout</td>
<td>DO Performs Layout</td>
</tr>
<tr>
<td>Select Insns to Issue</td>
<td>Wakeup/Select Logic</td>
<td>Wakeup/Select Logic</td>
<td>DO adapts at Exec Time</td>
</tr>
<tr>
<td>Precise Exceptions</td>
<td>Register Renaming</td>
<td>Register Renaming</td>
<td>SW Recovery Code</td>
</tr>
<tr>
<td>Complex Insns</td>
<td>Decoder Cracks</td>
<td>Decoder Cracks</td>
<td>DO or HW</td>
</tr>
<tr>
<td>Form Issue Groups</td>
<td>Select Logic</td>
<td>Select Logic</td>
<td>Issue Logic</td>
</tr>
</tbody>
</table>
VM Targets for Dynamic Compilers

- Same architecture.
  - No architectural changes involved.
- Specific instance of compatible architecture.
  - Re-optimize for particular implementation, but considering specific parameters.
- DAISY Host: Design for efficient hosting.
  - Allow architectural simplification.
  - Design host to ensure efficient mapping.
- Migration and compatibility.
  - Make old code run on new host.
  - Efficient mapping not primary design constraint.
  - But may be criterion: Itanium
Virtualization of Same ISA

- Allows simpler implementation of the same architecture, e.g. Dynamo on PA-RISC.
- Provides ability to bail out and revert to native execution:
  - If overhead too high
  - For hard-to-emulate sequences
  - When no benefit of DO can be measured
    - Or DO actually degrades performance
Virtualization of Different ISA

Different architecture, e.g., RISC $\Rightarrow$ VLIW

Advantages:
- Simplify architecture
- Reduce decoding overhead
- Add more registers, add new concepts
- Code packing / straightening.

Disadvantage:
- All code must be emulated. Can cause severe degradation if low reuse, e.g. WinStone.
In designing a host as target for dynamic optimizer, consider:

- Cost of emulating source architecture’s semantics
- Efficient mapping of architected state
- Providing resources for compiler for use in optimization
- Support dynamic optimization architecture
Matching Source Architecture Semantics

- Data formats
  - Representation of integers, floats, SIMD vectors, condition codes.

- Definition and detection of boundary conditions
  - Overflow, exceptions, carry in/out...

- Not everything must have a 1:1 mapping
  - Frequent/important idioms must have a high-performance solution
    - Can be Hardware / Software hybrid

- Bias towards maintaining compatible definition of many operations
Efficient Mapping of State

- Each aspect of state must be stored somewhere and must be locatable.
- Frequently accessed state should be easy to retrieve. Suggests a significant fraction of source architecture registers should be stored in registers.
- At control flow joins, must have same register mappings. Bias assignment towards “home locations” or pre-existing assignments.
Register Mapping at Joins

if (a > b) {
    x = y + 1;
} else {
    x = y - 1;
}

z = x + w;

\[ \begin{align*}
\text{x} & \text{ in “then” clause in r55} \\
\text{x} & \text{ in “else” clause in r55.} \\
\text{Ensures that z at control flow join gets correct value.}
\end{align*} \]
DAISY Host System Considerations

- Address translation similar to source architecture
- System architecture consistent with emulated systems
  - 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:
    - VMM = Virtual Machine Monitor – the “OS” of the translator and optimizer.
  - Timers consistent with source architecture
BREAK
BOA Support for Virtualization and Dynamic Optimization
Use dynamic binary translation to map PowerPC code to code for high performance underlying machine.

BOA Goals

- Execute existing PowerPC code 100% compatibly
  - User and supervisor state.

- Execute at high performance
  - Good CPI and high frequency.

⇒ Use dynamic binary translation to map PowerPC code to code for high performance underlying machine.
Benefits of BOA Optimization Layer

- Eliminate performance-degrading ops
- Avoid use of complex hardware idioms:
  - E.g., condition register broadside read/write
  - Replace with ops that are easier to schedule/execute
- Exploit novel architecture concepts
BOA Support for DO / VM: Registers

- **64 integer registers vs 32 for PowerPC**
  - *r0 to r31* same as PowerPC.
  - *r36 to r63*: Used for renaming and scratch results
  - *r33*: Hold PowerPC **Ctr** register
  - *r34*: Hold PowerPC **Link** register
  - *r35*: Hold constant 0
    - Useful for PowerPC form `lwz r3,8(r0) → lwz r3,<AbsAddr 8>`
    - Allows hardware to treat all registers uniformly:
      - No special case for *r0*. Instead `lwz r3,8(r35)`

- **64 floating point registers vs 32 for PowerPC.**
- **32 condition register fields vs 8 for PowerPC.**
Example of Load Speculation

Original *PowerPC* Code:

- addi r4, r4, 1
- xor r5, r4, r9
- beq cr0, L1
- lwz r3, 0(r6)

New code:

- addi r4, r4, 1    lwz r63, 0(r6)
- xor r5, r4, r9    beq cr0, L1
- copy r3, r63
Additional BOA Support for DO / VM

- Extra bits with registers to help renaming
  - e.g., CA, OV bits
- $LRA = Load\ Real\ Address$ insn for crossing groups and pages
- Ability to quash speculative I/O ops
- Hardware counters for profiling
- $Store\ Order\ Buffer$ so can rollback to beginning of translation
- Details to follow …
Implicit Architectural State

Support ability to rename implicit architectural state, i.e. state not named in instruction opcode.

**Example of Implicit State:** PowerPC status bits:

- **CA** = Carry
- **OV** = Overflow
- **SO** = Summary Overflow (Any op ever overflow?)
- **FPSCR** = Floating Point Status and Control Register
  - Summary Denorm
  - Summary NaN
  - (Many more)
Implicit Architectural State

Example of Limitations from Implicit State

- addc <CA>, r3, r4, r5  \text{Op1}
- adde <CA>, r6, r3, r7, <CA>  \text{Op2}
- addc <CA>, r8, r9, r10  \text{Op3}

If CA cannot be renamed, Op3 cannot be scheduled prior to Op1 or Op2.

BOA supports such re-ordering of operations updating and using implicit state.

- Extend each integer register with CA and OV bits
**Implicit Architectural State**

Extend each integer register with **CA** and **OV** bits:

- `addc <r3.CA>, r3, r4, r5`  **Op1**
- `adde <r6.CA>, r3, r6, r7, <r3.CA>`  **Op2**
- `addc <r8.CA>, r8, r9, r10`  **Op3**

**Op3** can now be scheduled independently of **Op1** and **Op2**.
Branching Between Groups and Across Pages

**Problem 1:** When exiting one group of translated instructions, must know:
- If a successor group exists corresponding to the next PowerPC instruction.
- Location of that successor group.
- If that successor group is still valid.

**Problem 2:** When execution of a group crosses what was a page boundary in the original PowerPC code, must check if there is a PowerPC instruction page fault.
One solution to **Problem 1**: 
**LVIA** = **Load VLIW Instruction Address**

**LVIA** Semantics:
- If a valid, translated group exists starting at an address corresponding to PowerPC address \( RY \), load its address and branch to it.
- If no valid, translated group exists, the **LVIA** op returns the address of the translator.

The **LVIA** cache is backed up by a larger memory list of translations akin to page tables.
Direct Branching Between Groups

**Drawback** to LVIA approach:
- Must execute extra LVIA operations -- one per exit point of group.

**Alternative:** Branch directly between groups:
- **Advantages:**
  - Fast – Use single LRA op at group start to verify correctness.
  (See next foil.)
  - **LRA** op needed anyway. (See next foil.)
- **Disadvantage:** VMM must track invalidations of translated groups and fix other groups that branch to them.
LRA:
Cross-Group / Cross-Page Branches

- **LRA** = Load Real Address
- **LRA** Semantics:
  - Get *real* PowerPC *address* from PowerPC Virtual Addr
  - **Compare:**
    - Real address for Virtual Address *right now*
    - Real address for Virtual Address *when group was formed*
  - Trap if mismatch or translation fault
- Put **LRA** insn:
  - At every page crossing within a group.
  - At start of each group.
  - **Exception:** If group is reached only from other groups on same page, no **LRA** is needed at group start.

- Can generally schedule other operations in same cycle as **LRA**.
I/O

Must detect references to memory-mapped I/O:

- I/O references must be performed in-order.
- Cannot be executed speculatively:
  - Unknown/undefined side effects
  - Reads can effect behavior
- PowerPC ISA has WIMG bits for each page
  - WIMG bits flag I/O references
    - Among other things
I/O

BOA hardware detects speculative I/O references:

- Prevents execution, generates trap.
- Recover in software:
  - Last defense against incorrect I/O operation
- Performance Heuristics:
  - Detect likely I/O references in initial profiling:
    - Compile these references without speculation and with non-trapping instructions
  - Also recompile without speculation later, when detect a reference often refers to I/O space.
PowerPC Load/Store Difficulties

In addition to I/O accesses, LOADS and STORES cause two other major problems:

1. Referenced (R) and Changed (C) bits in the (PowerPC) page table must be updated.

2. Memory reserved for BOA must be inaccessible to PowerPC programs, even when PowerPC address translation is disabled and the PowerPC is executing in system/privileged mode.
PowerPC Load/Store Difficulties

To deal with these problems, BOA uses a special co-designed TLB:
Solving PowerPC Load/Store Difficulties with Co-Designed TLB

- Update of PowerPC R and C bits.
  - When a page is brought into the BOA TLB, the BOA VMM sets the R bit in the PowerPC page table.
  - IF a page is brought into the BOA TLB by a STORE:
    - The BOA VMM also sets the PowerPC C bit.
  - ELSE if the page is brought in by a LOAD:
    - The page is marked READ-ONLY in the BOA TLB.
    - If there is a later STORE to the page:
      - A BOA TLB Miss occurs
      - The BOA VMM sets the PowerPC C bit.
BOA Memory must be inaccessible to PowerPC programs:

- BOA devotes a READ-ONLY page, $B$, of its memory for “bad” PowerPC memory references.
- All locations in $B$ contain the value 0xFFFFFFFF
- Any PowerPC LOAD/STORE that attempts to access BOA memory is remapped to page $B$ by the BOA TLB.
  - LOADS return the value 0xFFFFFFFF
  - STORES act as a NOP.
DAISY uses hardware profiling:

- At each exit from a DAISY group put an instruction:
  
  ```
  count exitID, Cycles_On_Path
  ```

- `exitID` is unique among all exits from DAISY groups.

- `Cycles_On_Path` is the estimated number of cycles from the start of this group to this exit.

  DAISY dynamic optimizer computes this value.

- `exitID` is used to index a counter cache:

  - If counter cache has no entry for `exitID`:
    
    - Counter cache entry is set to `Cycles_On_Path`
    
  - ELSE counter cache entry is incremented by `Cycles_On_Path`. 
Hardware Exit Counters for Profiling

Asynchronously to the DAISY processor, the counter cache compares each of its entries to a threshold cycle count, $C$.

If the counter for an entry exceeds $C$, the counter cache signals an asynchronous exception to the DAISY VMM, along with the exitID for the entry.

The DAISY dynamic optimizer can then re-optimize or restructure the group along the path ending at exitID.
Hardware Exit Counters for Profiling

**Note:** The counter cache increments the threshold count $C$ each cycle:

$\Rightarrow$ An exception is signaled only if the time spent on a particular path exceeds a certain *percentage* of execution time, not an *absolute amount* of execution time.

We have found that an 8-way associative counter cache with 8K entries is almost as accurate as software profiling, but with far less software overhead / program slowdown.
At BOA group entry, save PowerPC register state to shadow registers. Save done in one cycle by hardware when branching to new group. Copy values to PowerPC registers only at group exits.

On exception:
- Rollback to start of group
- Restore PowerPC shadow registers
- Interpret to find exception
PowerPC State and Precise Exceptions

Scratch Regs  PowerPC Regs  Shadow Regs

Group End  Group Start  Exception
Support for STORES

Problem: STORE executes but later instruction in group page faults.

Want to rollback to group start, but must rescind all executed STORES.

Solution: Store Order Buffer (SOB)
- Stored values go to SOB, not memory
- At group exit, all pending STORES in SOB are marked eligible for commit to memory.
- SOB writes eligible values to memory in order.
Store Order Buffer (SOB)

SOB

GroupStart Ptr

GroupEnd Ptr
Store Order Buffer (SOB)

BOA Group

STORE

SOB

GroupStart Ptr

GroupEnd Ptr
Store Order Buffer (SOB)

BOA Group

STORE

... STORE

SOB

GroupStart Ptr

GroupEnd Ptr
Store Order Buffer (SOB)

- **BOA Group**
- **STORE**
- ... 
- **STORE**
- ... 
- **Exception**

⇒ **Rollback**
Store Order Buffer (SOB)

BOA Group
STORE
...
STORE
...
STORE

SOB

GroupStart Ptr
GroupEnd Ptr
Store Order Buffer (SOB)

**BOA Group**

**STORE**

...**STORE**

...**STORE**

**BSHAD**

**SOB**

Memory

GroupStart Ptr

GroupEnd Ptr
**MP implications of SOBs**

- Cannot release data to remote node
  - Could be rolled back later
  - Must tell requesting node to wait

---

**Processor 1**

- **Group**: A
- **Action**: Store A, store in S-CAM
- **Action**: Read B, cross-interrogate
  - "wait for commit"

**Processor 2**

- **Group**: B
- **Action**: Store B, store in S-CAM
- **Action**: Read A, cross-interrogate
  - "wait for commit"

**DEADLOCK**
SOB-Aware Protocol

Must break deadlocks
  – Detect deadlocks
    May be complex, involving multiple nodes
  – Avoid conditions which cause deadlock

Deadlock avoidance
  – Break possible cycles
  – No “wait for commit” is sufficient
    But not necessary
Avoiding “Wait for Commit”

- Cannot deliver data due to possibility of roll back
- Must respond
  - not responding is “wait for commit”

Solutions:
- Roll back immediately, then return old value
- Tell remote host to rollback and re-execute
  - Will prevent this host’s waiting on remote data
  - May be preferable for lock-guarded structures
    - Snooping a lock is not useful
    - Writer of lock should exit section as quickly as possible
Livelocks and Starvation

There is a danger of livelock.

- **Solution:** Allow one processor to make progress:
  - By picking a node to prioritize
    - Use token to distribute equitably and prevent starvation
  - Exponential back-off

- Use performance monitor infrastructure to identify groups suffering excessive interference
  - Recompile, e.g., using smaller groups to reduce probability of interference
Speculative Load Support

- Use counter to assign *sequence number* to each LOAD and STORE in a group.
- *Sequence number* part of opcode

On STORE, hardware checks:
- STORE address overlaps prev LOAD address?
- Prev LOAD addr *sequence number* > STORE *sequence number*?

If aliasing between a load and store:
- Rollback group to start and start interpretation
- Possibly retranslate to unspeculate LOAD
Speculative Load Support (1)

Use ctr to assign sequence number to each LOAD and STORE in a group.

Sequence number part of opcode:

<table>
<thead>
<tr>
<th>PowerPC Code</th>
<th>BOA Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD X</td>
<td>1 LOAD X</td>
</tr>
<tr>
<td>...</td>
<td>2 STORE Y</td>
</tr>
<tr>
<td>STORE Y</td>
<td>3 LOAD Z</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>LOAD Z</td>
<td>...</td>
</tr>
</tbody>
</table>
Speculative Load Support (2)

- STORE addr overlaps a prev LOAD addr
- Prev LOAD addr sequence number >
- STORE sequence number ?

BOA Group
1. LOAD X
3. LOAD Z
2. STORE Y
...

Z aliases with Y
Seq #3 > Seq #2
Speculative Load Support (3)

- If aliasing:
  - Rollback group to start and re-execute
  - Possibly retranslate to unspeculate LOAD
BOA Architecture and Microarchitecture
Target BOA system

4 way chip multiprocessor (CMP)
- Building block for large SMPs
- We will only present uniprocessor performance
- System performance largely dependent on memory nest

Shared on-chip unified L2 cache
## BOA Caches

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Line Size</th>
<th>Assoc</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 - Insn</td>
<td>256K</td>
<td>256</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>L1 - Data</td>
<td>64K</td>
<td>128</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>L2 - Joint</td>
<td>4M</td>
<td>128</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td></td>
<td></td>
<td>90</td>
</tr>
</tbody>
</table>
BOA CMP floorplan
BOA ISA (1)

- BOA is variable length VLIW machine.
- BOA instructions (bundles) are 128 bits.
  - Bundles have 3 primitive ops.
  - Primitive ops have 39 bits plus stop bit.
  - 8 bits of bundle reserved for future uses such as predication.
- Instruction Issue operates on instruction packets
  - Up to 6 primitive ops are issued together.
  - Only last op issued may have stop bit set.
BOA Instruction Packet (dynamic abstraction)
BOA Instruction Bundle
(static abstraction)

Bundle with 3 sequential ops/packets

Op
Packet
STOP

Op
Packet
STOP

Op
Packet
STOP
BOA ISA (2)

- **64** Integer Registers
- **64** Float Registers
- **16** 4-bit Condition Registers

Branches take 1 cycle:
- Branch mispredicts cost 7 cycles
- Static branch prediction
  - using interpreter stats
- At most one branch per cycle
- Branch and checkpoint
  - For compiled group transitions
**BOA Architecture**

- *PowerPC* ops from single path in an atomic group.
- **6 Issue**
- Ops assigned to FUs in pipeline
- **Stall-on-use**
- Memop sequence #'s, Address Comparators

- Predicated bundles of 3 ops
- **1 branch per cycle**
- Branch prediction
BOA Resources

- 6 Issue Slots
  - Positional encoding simplifies issuing
- 2 LOAD / STORE units
  - Each with own copy of register file
- 4 Integer units
  - Each with own copy of register file
- 2 Float units
- 1 Branch unit
- 32-entry Load and Store Buffers
- Register scoreboarding of LOAD values
  - Stall when try to use loaded value
BOA Microarchitecture

- Decoupled fetch-execute
- Front-end autonomously fetches bundles
  - Formats bundle-based encoding stream to packets
  - Prepare-to-branch option to redirect instruction fetch
- Disperses packet to per-unit issue queue
  - Can issue up to 6 instructions to 9 units
- Stall-free backend
  - Traditional “stall” conditions handled using “recirculation”
    - Quash & re-issue violating instruction and successors
      - No need for “instantaneous” communication
  - Branch misprediction uses similar scheme
    - Quash and re-issue from correct path
  - Exceptions are handled similarly
    - Quash and re-issue from exception vector address
BOA Pipelines

Do not write results if packet nullified due to stall condition.
BOA Latencies

- **Integer ops take 1 cycle**
  - No bypass $\Rightarrow$ *Dependent ops must be 2 cycles apart*

- **LOADs take 3 cycles**
  - No bypass $\Rightarrow$ *Dependent ops must be 4 cycles later*
Recirculation

- Fetch 1
- Fetch 2
- Decode
- Issue
- GPR Rd
- Execute
- Broadcast
- Writeback

Recirculation Buffer

High Frequency ⇔
Not send global stall signals.
Recirculate Insn instead.
Recirculation

Fetch 1
Fetch 2
Decode
Issue

GPR Rd
Execute
Broadcast
Writeback

Recirculation Buffer
Input Regs Ready
Recirculation

- Fetch 1
- Fetch 2
- Decode
- Issue
- GPR Rd
- Execute
- Broadcast
- Writeback

Pause

Recirculation Buffer

Input Regs Ready

Recirculate

Quash
Recirculation

Fetch 1
Fetch 2
Decode
Issue
GPR Rd
Execute
Broadcast
Writeback

Fetch 1
Fetch 2
Decode
Issue

Recirculation Buffer

Input Regs Ready
Recirculation
Recirculation
Recirculation

Fetch 1
Fetch 2
Decode
Issue
GPR Rd
Execute
Broadcast
Writeback

Fetch 1
Fetch 2
Decode
Issue
GPR Rd
Execute
Broadcast
Writeback

Recirculation Buffer

Input Regs Ready
Recirculation

- Fetch 1
- Fetch 2
- Decode
- Issue
- GPR Rd
- Execute
- Broadcast
- Writeback

- Fetch 1
- Fetch 2
- Decode
- Issue
- GPR Rd
- Execute
- Broadcast
- Writeback

- Recirculation Buffer
- Input Regs Ready
Recirculation

Fetch 1
Fetch 2
Decode
Issue

GPR Rd
Execute
Broadcast
Writeback

Fetch 1
Fetch 2
Decode
Issue

GPR Rd
Execute
Broadcast
Writeback

Recirculation Buffer
Recirculation

Fetch 1
Fetch 2
Decode
Issue

Fetch 1
Fetch 2
Decode
Issue

GPR Rd
Execute
Broadcast
Writeback

GPR Rd
Execute
Broadcast
Writeback

Recirculation Buffer
Input Regs Ready
Recirculation

- Fetch 1
  - Fetch 2
  - Decode
  - Issue

- GPR Rd
  - Execute
  - Broadcast
  - Writeback

- Execute
  - Broadcast
  - Writeback

- Recirculation Buffer

- Input Regs Ready
Recirculation
Recirculation

Fetch 1  Fetch 1
Fetch 2  Fetch 2
Decode  Decode
Issue   Issue

Recirculation Buffer

GPR Rd  GPR Rd
Execute  Execute
Broadcast Broadcast
Writeback Writeback

Input Regs Ready
Recirculation

- Fetch 1
- Fetch 2
- Decode
- Issue
- GPR Rd
- Execute
- Broadcast
- Writeback
- Recirculation Buffer
- Input Regs Ready
Recirculation

- Fetch 1
- Fetch 2
- Decode
- Issue
- GPR Rd
- Execute
- Broadcast
- Writeback

Recirculation Buffer

Input Regs Ready
Scoreboarding & Signal Distribution

- Distributed issue queue
  - Lockstep operation (issue entire packet only)

- Long running instructions scoreboard result
  - Late in pipeline after other conditions have been resolved
    - No need to “retract” a scoreboard dirty condition
  - Reduces pressure on scoreboard signal distribution
  - Operations: LOAD, SPR access…
    - Optionally long running FP to eliminate vertical NOPs

- Short latency operations do not interlock via scoreboard
  - BOA dynamic compiler must schedule at proper distance
  - All simple operations: ALU ops, shifter, store,…

- All units issue aggressively
  - Scoreboard accessed with GPR read
  - If any operand not ready, cancel instruction during EX stage
    - “QUASH” signal broadcast to all pipelines
  - Re-issued from recirculation buffer
Issue Logic

- Easy to schedule for known latencies in an in-order machine
- Scheduling ops of unknown latency is problematic
  - Stall-on-cache-miss / Stall-on-long-latency-op easy but penalizes performance
  - **Scoreboard variable latency ops**
  - **Optionally scoreboard long latency ops to reduce vertical NOPs (FDIV and similar ops)**
    - Keys to performance
- All interesting variable latency ops are long latency
- Have enough window to take time for updates
BOA Performance
Benchmarks

- **SPECint95**
  - Uniformly Sampled PowerPC Traces
  - 2 million instructions per sample
  - 50 samples per benchmark

- **TPC-C**
  - Special-purpose hardware
  - 170 million instruction trace
Factors in BOA Performance

- Instruction Reuse Rate
- # of times each instruction is translated
- Translator CPI
- Interpreter CPI
- Statistics CPI
- Synchronous Exception Rate
- ICache flushing from translator
- Average Group Length
- # of times interpret before translating
Execution Time of Translated Code

- Ignoring Cache Effects, \textbf{Cycles for Each Path Through Group} = \textbf{Number of VLIW Instructions}

- Total Cycles Spent in Group:

\[ \sum ( \text{# of VLIW Ins in } P ) \times ( \text{# Times } P \text{ Executes} ) \]

\textit{All Group Paths } P
Instruction Cache Cycles

- Layout VLIW code for all groups
- Index VLIW code by group exit points
- Go thru exit points in execution order
  - Iterate through all VLIW Instruction Addresses corresponding to each exit
  - Feed Addresses to Multilevel ICache Simul
  - Simulator includes history-based prefetch
Data Cache Cycles

- Modeling *Speculative Loads* Difficult in Trace-Based Environment
- Addresses for Speculative Ops not on actual Execution Path are Unknown
  - Use LD/ST addresses from PowerPC trace as input to DCache/DTLB simulation
- Multiply DCache/DTLB Stall Cycles by 1.7
- 1.7 = Increase in Execution-based DAISY
- Simulated DCache *not* lockup-free
  - BOA’s cache is lockup free
Translation Cycles

- Measure of CPI adder due to time spent in translation
- Average number of clocks required to translate an instruction
  - CPI of translator
    - translation - 2500 cycles
  - more sophisticated optimizations increase this penalty
  - delicate balance between translated code performance and translation overhead
- Number of times an instruction gets retranslated
- Reuse Rate
  - Time spent in translator per instruction is amortized by the repetition rate of that instruction

Cycles spent translating 1 instruction
- \((1/ \text{Reuse Rate}) \times (\text{Translation CPI} \times \text{Translations})\)
Overall CPI

Total Cycles for VLIW Execution:
- Infinite Cache Cycles +
- ICache Cycles +
- DCache Cycles +
- DTLB Cycles +
- Branch misprediction +
- Interpretation & Translation Overhead

CPI = Total VLIW Cycles / Orig PPC Ins

Translation Overhead Negligible
Effect of Bias on CPI

![Graph showing the effect of bias on CPI for different benchmarks and bias levels. The x-axis represents benchmarks like li, perl, m88k, go, ijpeg, vortex, gcc, compress, and tpc. The y-axis represents the CPI with values from 0 to 2.5. The graph compares the performance of CPU with different bias levels (Bias-8, Bias-12, Bias-15).]
Static and Dynamic Group Length, as Function of Bias

- Static group length (8/15)
- Static group length (12/15)
- Static group length (15/15)
- Dynamic group length (8/15)
- Dynamic group length (12/15)
- Dynamic group length (15/15)
Oracle Static Branch Prediction

Baseline
Oracle
BOA and DAISY CPI
Comparison of BOA to DAISY
## BOA and DAISY Differences (1)

<table>
<thead>
<tr>
<th>BOA</th>
<th>DAISY</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>PowerPC</em> ops from single path</td>
<td><em>PowerPC</em> ops from multiple paths</td>
</tr>
<tr>
<td>Stopping conditions relate to code expansion, resource limits</td>
<td>Stopping conditions are <em>ILP</em>-based. Re-optimization to increase ILP</td>
</tr>
<tr>
<td>Hardware-based resource constrained</td>
<td>Software-based issue-BW constrained</td>
</tr>
<tr>
<td>▪ Order buffers</td>
<td>▪ LV</td>
</tr>
<tr>
<td>▪ Rollback,…</td>
<td>▪ In-order commit,…</td>
</tr>
</tbody>
</table>
BOA and DAISY Differences (2)

**BOA**
- 6 Issue
- Ops assigned to FUs in pipeline
- **Stall-on-use**
- Memop sequence #'s, Address Comparators

**DAISY**
- 8-16 Issue
- Mini-Icache maps fixed cache locations to FUs
- **Stall-on-miss**
- Load-Verify Instructions
BOA and DAISY Differences (3)

**BOA**
- Predicated bundles of 3 ops
- *1 branch per cycle*
- Branch prediction

**DAISY**
- Tree instructions
- *Up to 3 branches per cycle*
- Encode successor cache line in instruction $\Rightarrow$ Fetch known insn each cycle
BOA and DAISY Differences (4)

BOA
- Exclusively targeted at PowerPC

DAISY
- Research to target multiple architectures
- Architecture commonality
- Architecture virtualization
Summary
and
Observations
Proof of Concept

System-level dynamic compilation demonstrated by:

- DAISY, BOA
- Transmeta
- FX!32
- IA32-EL
Optimization Opportunities and Challenges

New Optimization Opportunities:
- Load-Store Telescoping
- Non-conservative approaches to aliasing
- System-level optimization techniques, e.g., large pages “under the covers”

New Optimization Challenges:
- Dead Code Elimination
- Management of Translated Code
  - Hazelwood and Smith [CGO 2004]
New Paradigm

System-level dynamic compilation offers opportunities for paradigm shift

– Merged ISAs in one implementation
  – PowerPC / z-Series
  – x86 / IA-64

⇒ Lower development costs
⇒ Dynamically allocate fixed hardware to different ISAs in server farm