An Overview of IA-64 Architectural Features and Compiler Optimization

Yong-fong Lee
Intel Corporation
Outline

- IA-64 Architecture Overview
- Architectural Features and Compiler Optimization
  - Control speculation
  - Data speculation
  - Predication
  - Parallel compares
  - Multi-way branches
  - Memory hierarchy control
- IA-64 Architectural Support for Software Pipelining
- Summary
IA-64 Strategies

- Move the complexity of resource allocation and instruction scheduling to the compiler
  - no complex runtime dependence checking and resource management in hardware

- Provide architectural features that enable aggressive compile-time optimization to utilize ILP (instruction-level parallelism)
  - predication, speculation, register rotation, parallel compares, multi-way branches, architecturally visible memory hierarchy

- Provide wide-issue processor implementations that the compiler can take advantage of
  - large register files, multiple execution units
IA-64 Application State

- Directly accessible CPU state
  - 128 x 65-bit General registers (GR)
  - 128 x 82-bit Floating-point registers (FR)
  - 64 x 1-bit Predicate registers (PR)
  - 8 x 64-bit Branch registers (BR)

- Indirectly accessible CPU state
  - Current Frame Marker (CFM)
  - Instruction Pointer (IP)

- Control and Status registers
  - 19 Application registers (AR)
    - LC and EC
  - User Mask (UM)
  - CPU Identifiers (CPUID)
  - Performance Monitors (PMC, PMD)

- Memory
# Instruction Formats: Bundles

![Instruction Formats Diagram](image)

<table>
<thead>
<tr>
<th>Instruction 2 41 bits</th>
<th>Instruction 1 41 bits</th>
<th>Instruction 0 41 bits</th>
<th>template 5 bits</th>
</tr>
</thead>
</table>

- **Instruction Types**
  - M: Memory
  - I: Shifts, MM
  - A: ALU (M or I)
  - B: Branch
  - F: Floating point
  - L+X: Long immediate

- **Template types**
  - Regular: MII, MLX, MMI, MFI, MMF
  - Stop: MI_I, M_MI
    - One bundle expanded into two for execution
  - Branch: MIB, MMB, MFB, MBB, BBB
  - All come in two versions:
    - with *stop* at end
    - without *stop* at end
Instruction Groups

- A sequence of instructions with no register dependencies
  - **Exception:** WAR register dependencies allowed
  - Memory operations still require sequential semantics
- The compiler uses templates with stops to delimit instruction groups
- The hardware issues one or more bundles of instructions within an instruction group
  - Does not have to check for register dependencies
- Dependencies disabled by predication dynamically

<table>
<thead>
<tr>
<th>Case 1 - Dependent</th>
<th>Case 2 - Independent</th>
<th>Case 3 - Predication</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1 = r2, r3 ;;</td>
<td>add r1 = r2, r3</td>
<td>(p1) add r1 = r2, r3</td>
</tr>
<tr>
<td>sub r4 = r1, r2 ;;</td>
<td>sub r4 = r11, r21</td>
<td>(p2) sub r1 = r2, r3</td>
</tr>
<tr>
<td>shl r2 = r4, r8</td>
<td>shl r12 = r14, r8 ;;</td>
<td>shl r12 = r1, r8</td>
</tr>
</tbody>
</table>
Control Speculation

Original

Transform

Reschedule

- load
- use
- load deferring exception
- check exception
- use
- load deferring exception
- check exception
- use
Architectural Support for Control Speculation

- **Mechanism for deferred exceptions**
  - 65th bit (NaT bit) in each GR indicates if an exception has occurred.
  - Special value NaTVal (a special NaN) in each FR indicates if an exception has occurred.

- **Setting and propagation of deferred exceptions**
  - Speculative loads (ld.s) set the NaT bit or NaTVal if a deferrable exception occurs.
  - Speculative checks (chk.s) check the NaT bit or NaTVal and branches to recovery, if detected.
  - Computational instructions propagate NaT and NaTVal like IEEE NaN’s.
  - Compare instructions propagate “false” when writing predicates or leave them unchanged, depending on the compare type.
Compiler Directed Control Speculation

1. Separate load behavior from exception behavior
   - \texttt{ld.s} defers exceptions
   - \texttt{chk.s} checks for deferred exceptions

2. Reschedule \texttt{ld8.s}
   - \texttt{ld8.s} will defer a fault and set the NaT bit on \texttt{r1}
   - \texttt{chk.s} checks \texttt{r1}'s NaT bit and branches/faults if necessary
Cost-Benefit Consideration for Control Speculation

- Increased resource pressure and code size
  - chk.s takes an instruction slot

- Extended register live ranges

- More memory traffic
  - ld.s may be executed unnecessarily

- Profile sensitivity
  - Should not speculate a load above a branch from an infrequent successor block
Data Speculation

Original

Transform

Reschedule

store
load
use

store
load ignoring conflict
use

load ignoring conflict
use
store
check addressing conflict
Architectural Support for Data Speculation

- ALAT - Advanced load address table
  - HW structure containing information about outstanding advanced load addresses
  - “Snoop” on stores and other memory writes to delete overlapping advanced load addresses

- Instructions
  - ld.a - advanced load
  - ld.c - check load
  - chk.a - advance load checks
  - ld.sa - combined control and data speculation.
Compiler Directed Data Speculation

1. Separate load behavior from overlap detection
   - *ld8.a* can be scheduled passing aliased stores
   - *chk.a* detects conflict

2. Reschedule *ld8.a*
   - *ld8.a* allocates an entry in the ALAT when executed
   - If the *st1* overlaps with the *ld8.a*, the ALAT entry will be removed
   - *chk.a* checks for matching entry in ALAT -- if found, speculation was ok; if not found, need to perform recovery
Issues with Data Speculation

- Efficient recovery mechanism
  - Straightforward implementation may incur pipeline flush, branch miss, and Icache miss

- Cost-benefit consideration
  - Increased resource pressure and code size
    - ld -> ld.a/chk.a
  - Extended register live ranges
  - ALAT size and associativity
  - Conflict rates of memory operations
    - Average recovery overhead may cost more than the load latency
Predication

- Allow instructions to be dynamically turned on or off by using a predicate register value

- Example:
  
  \[
  \text{cmp.eq } p1, p2 = r1, r2 ;;
  \]
  
  \[
  (p1) \quad \text{add } r7 = r2, r4
  \]
  
  \[
  (p2) \quad \text{ld8 } r7 = [ r8 ]
  \]

  - If \( p1 \) is true, \text{add} is performed, else \text{add} acts as a nop
  - If \( p2 \) is true, \text{ld8} is performed, else \text{ld8} acts as a nop
Architectural Support for Predication

- 64 1-bit predicate registers (true/false)
  - p0 - p63

- Compare and test instructions write two predicates with results of comparison/test
  - most compare/test write result and complement
  - Ex: cmp.eq p1,p2 = r1,0

- Almost all instructions can have a qualifying predicate (qp)
  - Ex: (p1) add r1 = r2, r3
  - if qp is true, instruction executed normally
  - if qp is false, instruction nullified
Compiler IF-Conversion

- Convert control dependence into data dependence
- Remove branches to
  - Reduce/eliminate branch mispredictions and branch bubbles
  - Improve instruction fetch efficiency
  - Better utilize wide-issue machine resources
To move non-speculative instructions downward
– such as stores or chk’s
Issues with Predication

- Icache pollution from nullified code

- Region formation for IF-conversion
  - Identify hard-to-predict branches
    » Branch frequency?
    » Branch direction change?
    » Branch misprediction profile?
  - Balance cold and hot paths in a region
  - Performance potential for control-intensive scalar code?

- Data flow analysis of predicated code
  - More precise live range information for register allocation
  - Optimization of predicated code
Parallel Compares

- Parallel compares allow compound conditionals to be executed in a single instruction group.
  - Permit WAW in an instruction group

Example:

```c
if ( A && B && C ) { S }
```

Assembly:

```assembly
cmp.eq p1 = r0,r0 ;; // initialize p1=1
cmp.ne.and p1 = rA,0
cmp.ne.and p1 = rB,0
cmp.ne.and p1 = rC,0
(p1) S
```
Control Height Reduction by Parallel Compares

Original

Transform/Reschedule

cmp pA = (pA) br.cond

cmp pB = (pB) br.cond

cmp pC = (pC) br.cond

cmp.and pABC = cmp.and pABC = cmp.and pABC = (pABC) br.cond
Multi-way Branches

- Allow multiple branch targets to be specified in one instruction group

Example:

```plaintext
{ .bbb
  (p1)   br.cond target_1
  (p2)   br.cond target_2
  (p3)   br.call b1
}
```

- Control transfers to the first true target in the sequence:
  - target_1
  - target_2
  - content of branch register b1
  - fall-through
Control Height Reduction by Multi-way Branch

- Speculation, IF-conversion with side exits, multi-way branch

Original

- ld8 r6 = [ra]
  (p1) br exit1

- ld8 r7 = [rb]
  (p3) br exit2

- ld8 r8 = [rc]
  (p5) br exit3

Control Speculation

- ld8 r6 = [ra]
  ld8.s r7 = [rb]
  ld8.s r8 = [rc]

  (p1) br exit1

  (p3) br exit2

  (p5) br exit3

IF-conversion multi-way branch

- ld8 r6 = [ra]
  ld8.s r7 = [rb]
  ld8.s r8 = [rc]

  (p2) chk r7, rec1
  (p3) br exit2

  (p4) chk r8, rec2

  (p1) br exit1

  (p3) br exit2

  (p5) br exit3

1 branch cycle

3 branch cycles

very rare

rare

P1

P2

P3

P4

P5

P6
Memory Hierarchy Control

- Orchestrate data movement across the cache hierarchy
  - Locality hints
    » specified in load, store, Ifetch instructions
    » Default - temporal locality at level 1
    » NT1/NT2 - no temporal locality at level 1/2
    » NTA - no temporal locality at all levels
  - Explicit prefetch
    » Ifetch instruction
  - Implicit prefetch
    » post-increment with load, store, Ifetch instructions
    » bring the line containing the post-incremented address
Selective Prefetching

- Reuse analysis for the best level in the hierarchy
- Avoid redundant prefetches (by using predication)

```c
for i = 1, M
    for j = 1, N
        A[j, i] = B[0, j] + B[0, j+1]
    end_for
end_for
```

```c
for i = 1, M
    for j = 1, N
        A[j, i] = B[0, j] + B[0, j+1]
        if (mod(j, 8) == 0)
            lfetch.nt1(A[j+d, i])
        end_for
    end_for
end_for
```
Post-Increment

\[
\text{for } (i = 1; i < 10; i += 1) \\
a[i+N] = a[i-1] + a[i+1]
\]

\[
r = a \\
\text{LOOP:} \\
r1 = r-8 \\
tmp1 = \text{load } *(r1) \\
r2 = r+8 \\
tmp2 = \text{load } *(r2) \\
tmp3 = tmp1 + tmp2 \\
r3 = r+N*8 \\
\text{store } tmp3 \text{ (r3)} \\
r = r+8 \\
\text{if } r < a+80 \text{ goto LOOP}
\]

\[
r = a - 8 \\
r1 = a + 8 \\
r2 = a + N*8 +8 \\
\text{LOOP:} \\
tmp1 = \text{load } *(r), r+=8 \\
tmp2 = \text{load } *(r1), r1+=8 \\
tmp3 = tmp1+ tmp2 \\
\text{store } tmp3 \text{ (r2), } r2+=8 \\
\text{if } r < a+88 \text{ goto LOOP}
\]
Outline

- IA-64 Architecture Overview
- Architectural Features and Compiler Optimization
  - Control speculation
  - Data speculation
  - Predication
  - Parallel compares
  - Multi-way branches
  - Memory hierarchy control
- IA-64 Architectural Support for Software Pipelining
- Summary
Software Pipelining

- Hardware pipelining is a hardware implementation technique for overlapping multiple instructions in execution.
- Software pipelining is a software/compiler technique for overlapping instruction instances from multiple loop iterations in execution.
  - ld1, add1, st1 from iteration #1
  - ld2, add2, st2 from iteration #2
  - ld3, add3, st3 from iteration #3

L1: ld4 r4 = [r5], 4 ;; // 0
    add r7 = r4, r9 ;; // 1
    st4 [r6] = r7, 4  // 2
    br.cloop L1 ;;    // 2

Simplistic Pipeline

```
ld 1
ld 2 add 1
ld 3 add 2 st 1
add 3 st 2
st 3
```
Modulo scheduling is a simple, effective algorithm for software pipelining loops.

- **Initiation Interval (II)** - number of cycles between start of successive iterations.
  - minimal II = max( Resource II, Recurrence II )

- **Prolog** - fill pipeline
- **Kernel** - steady state, 1 iteration completes every II cycles
- **Epilog** - drain pipeline

- # of stages = ceil( length of loop schedule / II )
Modulo-Scheduled Loop Without Hardware Support

Kernel must be unrolled because of no rotating registers
1 cycle per iteration, 3x performance improvement
About 5x code-size expansion
IA-64 Support for SWP

- Full predication by IF-conversion
  - Remove control flow in loops to allow for modulo scheduling
- Rotating registers
- Loop Count (LC) and Epilog Count (EC) application registers
- Special loop-type branches
Objective of IA-64 SWP Support

- Reduce code size expansion
  - Limited unrolling of kernel <= rotating GRs, FRs, and predicates
  - No prolog, epilog code <= rotating predicates, EC
  - Reduced loop-maintenance overhead <= loop-type branches, LC

- Avoid loop-type branch mispredictions
  - Static hints avoid first-iteration misprediction
  - Loop-type branches together with LC and EC allow the hardware to predict loop exit early

```
L1: (p16) ld4 r32 = [r5], 4
    (p17) add r34 = r33, r9
    (p18) st4 [r6] = r35, 4
    br.ctop.dptk L1
```
Rotating Registers

- Rotation registers
  - GR rotation: Programmable sized region of the general register file rotates
- FP registers 32 through 127 rotate
- Predicates 16 to 63 rotate
  - FR rotation: 3/4 of FR can be used for rotation
  - PR rotation: 3/4 of PR can be used for rotation
- State
  - CFM.rrb.gr - register rename base for general registers
  - CFM.rrb.fr - rename base for floating-point registers
  - CFM.rrb.pr - rename base for predicate registers
  - CFM.sor - size of rotating region of general registers
- CFM.rrb’s decremented modulo the size of the rotating region
Stacking and Rotation

Every frame consists of locals, rotating, and outputs

Virtual

32+sof-1
32+sol

Outputs

Local

32+sort

size of rotating
- must be multiple of 8
- must be <= sof
Loop-Type Branch

- Loop control registers
  - LC: loop count register
  - EC: epilog count register
- br.ctop uses LC and EC for pipelining counted loops
Rotating Stage Predicates in Modulo-Scheduled Loop

Generate this loop:

- \( ar.lc = 4 \)
- \( ar.ec = 3 \)
- \( pr.rot = 0x10000 \)

Get this trace:
(5 iterations)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>ld</th>
<th>add</th>
<th>st</th>
<th>br.ctop</th>
<th>p16</th>
<th>p17</th>
<th>p18</th>
<th>LC</th>
<th>EC</th>
<th>br</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>F</td>
</tr>
</tbody>
</table>

- 1 cycle per iteration, 3x performance improvement
- Minimal code size expansion
Issues with Software Pipelining

- Loop with multiple exits
  - Cost of converting it into a single exit or generating explicit epilogs
- Control and data speculation in software pipelined loop
  - Decision is not as easy as in list scheduling
- Allocation of rotating register and non-rotating registers
  - separated or in a single pass?
- Loops in scalar code
  - Loops with control flow
    » Cost of full IF-conversion
  - Loops with short trip counts
    » Peeling, unrolling, or software pipelining?
  - Data dependence testing for pointer references most challenging
    » Loop carried dependence with distance conservatively assumed to be one
The compiler should take full advantage of IA-64 architectural features to generate optimized code.

<table>
<thead>
<tr>
<th>Architectural Features</th>
<th>Compiler Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control and data speculation, Predication, Multi-way branches</td>
<td>Global scheduling with control and data speculation, Predicate promotion</td>
</tr>
<tr>
<td>Multi-way branches, Parallel compares</td>
<td>Control height reduction</td>
</tr>
<tr>
<td>Predication, Parallel compares</td>
<td>IF-conversion, Opt of predicated code</td>
</tr>
<tr>
<td>Rotating registers, Predication, Loop-type branches</td>
<td>Modulo scheduling</td>
</tr>
<tr>
<td>Locality hints, Prefetch, Post-increment</td>
<td>Memory hierarchy control</td>
</tr>
</tbody>
</table>
Compiler Challenges

- Compiler architecture issues
  - Seamless integration of optimization components
    - IF-conversion generates predicated instructions
    - Scheduling and register allocation query relationships among predicates
  - A scheduler-centric compilation framework?
    - A global scheduler drives all other optimization and transformation components

- Understand interactions among optimization and transformation components/techniques
  - Tail duplication increases ILP
  - Tail duplication increases code size
  - ILP will help more? Or code size expansion will hurt more?
For More Information

- IA-64 Application Developer's Architecture Guide
  - Hard copy (free of charge)
    » Call 1-800-548-4725
    » Order number 245188-001
  - Download from IA-64 home page:
    http://developer.intel.com/design/ia64/index.htm

- Intel Itanium Processor Microarchitecture Overview
  - Download from
    http://developer.intel.com/design/ia64/microarch_ovw/index.htm
More IF-Conversion

- Side-exits in IF-conversion region

Original

Transform/
Reschedule

 cmp p1,p2 = ...
   (p1) br

 cmp p3,p4 = ...
   (p2) cmp . unc p3,p4 = ...

r8 = 5
   cmp p1,p2 = ...
   (p2) cmp . unc p3,p4 = ...
   (p1) r8 = 5
   (px) br

r8 = 7
   cmp p3,p4 = ...
   (p3) r8 = 7
   (p4) r8 = 10
   (py) br

r8 = 10
   cmp p1,p2 = ...
   (p4) r8 = 10
   (p4) cmp . py = ...
   (py) br

r8 = 10
   cmp p1,p2 = ...
   (p1) br

r8 = 7
   cmp p3,p4 = ...
   (p3) br

r8 = 7
   cmp px = ...
   (px) br

r8 = 5
   cmp px = ...
   (px) br

rare

rare
A Simple Counted Loop

- Assume an even number of iterations
- 5 cycles for 2 original iterations vs. 4 cycles for 1 iteration
- ~2x code size
- Induction variables r5 and r6 impose dep on ld->ld and st->st
Modulo-Scheduled Loop Branch Types and Stage Predicates

State

- PR16 is defined to be the first stage predicate for counted loops
  - simple fill and drain of pipeline
- Any predicate can be the first stage predicate for while loops
  - early (speculative) stages of the pipeline may not have a predicate
- LC: loop count application register
  - initialize to (trip count - 1)
- EC: epilog count application register
  - initialize to (# epilog stages + 1)

br.ctop, br.cexit always effectively write LC, EC, CFM.rrb’s
br.wtop, br.wexit always effectively write EC, CFM.rrb’s
br.ctop, br.cexit, br.wtop, br.wexit always write PR[63]
PR[63] becomes PR[16] after rotation