IA64 Architecture and Compilers

Dr. Allan Knies, Intel, IA64 Processor Division
Dr. Wei Li, Intel, Microcomputer Software Labs
Dr. Jesse Fang, Intel, Microprocessor Research Labs
IA-64 Application Architecture
Tutorial

Allan D. Knies
IA-64 Architecture and Performance Group
Intel Corporation
allan.knies@intel.com
Objectives for This Tutorial

Provide background for some of the architectural decisions

Provide a description of the major features of the IA-64 application architecture
  – Provide introduction and overview
  – Describe software and performance usage models
  – Mention relevant design issues

Show an example of IA-64 feature usage (C -> asm)
Agenda for This Tutorial

IA-64 history and strategy

IA-64 application architecture overview

C -> IA-64 example

Reference slides (included, but not covered)
IA-64 Definition History

Two concurrent 64-bit architecture developments:
- IAX at Intel from 1991
  - Conventional 64-bit RISC
- Wideword at HP Labs from 1987
  - Unconventional 64-bit VLIW derivative

IA-64 definition started in 1994
- Extensive participation of Intel and HP architects, compiler writers, micro-architects, logic/circuit designers
- Several customers also participated as definition partners

Currently there are 3 generations of microprocessors in different stages of design
IA-64 Strategies

Extracting parallelism is difficult
- Existing architectures contain limitations that prevent sufficient parallelism on in-order implementations

Strategy
- Allow the compiler to exploit parallelism by removing static scheduling barriers (control and data speculation)
- Enable wider machines through large register files, static dependence specification, static resource allocation

Hot Chips 1999
IA-64 Strategies

Branches interrupt control flow/scheduling
  – Mispredictions limit performance
  – Even with perfect branch prediction, small basic blocks of code cannot fully utilize wide machines

Strategies
  – Allow compiler to eliminate branches (and increase basic block size) with predication
  – Reduce the number and duration of branch mispredicts by using compiler generated branch hints
  – Allow compiler to schedule more than one branch per clock - multiway branch
IA-64 Strategies

Memory latency is difficult to hide
  – Increasing relative to processor speed (larger cache miss penalties)

Strategy
  – Allow the compile to schedule for longer latencies by using control and data speculation
  – Explicit compiler control of data movement through an architecturally visible memory hierarchy
IA-64 Strategies

Procedure calls interrupt scheduling/control flow
  – Software modularity is standard
  – Call overhead from saving/restoring registers

Strategy
  – Provide special support for software modularity
  – Reduce procedure call/return overhead
    • Register Stack
    • Register Stack Engine (RSE)
IA-64 Strategies Summary

Move complexity of resource allocation, scheduling, and parallel execution to compiler

Provide features that enable the compiler to reschedule programs using advanced features (predication, speculation)

Enable wide execution by providing processor implementations that the compiler can take advantage of
Agenda for This Tutorial

IA-64 history and strategy

IA-64 application architecture overview

C -> IA-64 example

Reference slides included (but not covered)
  – Loop Support
  – Register Stack
  – Memory Support
  – Floating Point, Multi-media, 3D Graphics
IA-64 Application Architecture Tutorial

Application State
Instruction Format
Integer Instructions
Execution Semantics
Control Speculation, Data Speculation
Predication
Parallel Compares
Branch Architecture

Hot Chips 1999
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)
- User Mask (UM)
- CPU Identifiers (CPUID)
- Performance Monitors (PMC,PMD)

Memory
## IA-64 Application Architecture Tutorial

- Application State
- Instruction Format
- Integer Instructions
- Execution Semantics
- Control Speculation, Data Speculation
- Predication
- Parallel Compares
- Branch Architecture
### Instruction Formats: Bundles

#### Instruction Types

- **M**: Memory
- **I**: Shifts, MM
- **A**: ALU
- **B**: Branch
- **F**: Floating point
- **L+X**: Long

#### Template types

- **Regular**: MII, MLX, MMI, MFI, MMF
- **Stop**: MI_I, M_MI
- **Branch**: MIB, MMB, MFB, MBB, BBB

All come in two versions:
- with stop at end
- without stop at end
Instruction Formats: Instructions

<table>
<thead>
<tr>
<th>major opc 4b</th>
<th>minor opcode or immediate 10 bits</th>
<th>register id 7 bits</th>
<th>register id 7 bits</th>
<th>register id 7 bits</th>
<th>qual. pred 6 bits</th>
</tr>
</thead>
</table>

Qualifying predicates (6 bits)

- A few instructions do not have a QP

Register operand identifiers (7 bits)
Register result identifier(s) (6 or 7 bits)
Immediate operands (8-22 bits)
Minor opcode
Major opcode (4 bits)
IA-64 Application Architecture Tutorial

Application State
Instruction Format
Integer Instructions
Execution Semantics
Control Speculation, Data Speculation
Predication
Parallel Compares
Branch Architecture
Integer Instructions

Memory - load, store, semaphore, . . .
Arithmetic - add, subtract, shladd, . . .
Compare - lt, gt, eq, ne, . . ., tbit, tnat
Logical - and, or
Bitfields - deposit, extract
Shift Pair
Character
Shifts - left, right
32-bit support - cmp4, shladdp4
Move - various register files moves
No-ops
IA-64 Application Architecture Tutorial

Application State
Instruction Format
Integer Instructions
Execution Semantics
Control Speculation, Data Speculation
Predication
Parallel Compares
Branch Architecture

Hot Chips 1999
Execution Semantics

Traditional architectures have sequential semantics

– The machine must always *behave* as if the instructions were executed in an unpipelined sequential fashion
– If a machine actually issues instructions in a different order or issues more than one instruction at a time, it must insure sequential execution semantics are obeyed

<table>
<thead>
<tr>
<th>Case 1 - Dependent</th>
<th>Case 2 - Independent</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1 = r2, r3</td>
<td>add r1 = r2, r3</td>
</tr>
<tr>
<td>sub r4 = r1, r2</td>
<td>sub r4 = r11, r21</td>
</tr>
<tr>
<td>shl r2 = r4, r8</td>
<td>shl r12 = r14, r8</td>
</tr>
</tbody>
</table>
Execution Semantics

IA-64 has parallel semantics

- The compiler uses templates with stops to indicate dependent operations
- Hardware does not have to check for dependent operations within instruction groups
  - WAR register dependences allowed
  - Memory operations still require sequential semantics
- Dependences 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>

Hot Chips 1999
IA-64 Application Architecture Tutorial

Application State
Instruction Format
Integer Instructions
Execution Semantics

Control Speculation, Data Speculation
Predication
Parallel Compares
Branch Architecture

Hot Chips 1999
Control and Data Speculation

Two kinds of instructions in IA-64 programs
- Non-Speculative Instructions -- known to be useful/needed
  - would have been executed in the original program
- Speculative instructions -- may or may not be used
  - Schedule operations before results are known to be needed
  - Usually boosts performance, but occasionally may degrade
  - Heuristics can guide compiler in aggressiveness
  - Need profile data for maximum benefit

Two kinds of speculation
- Control and Data

Moving loads up is a key to performance
- Hide increasing memory latency
- Computation chains frequently begin with loads
### Speculation

Separates loads into 2 parts: speculative loading of data and detection of conflicts/faults.

<table>
<thead>
<tr>
<th>Control Speculation</th>
<th>Data Speculation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Original:</strong></td>
<td></td>
</tr>
<tr>
<td>[(p1)] \texttt{br.cond} \texttt{ld8 r1 = [ r2 ]}</td>
<td>\texttt{st4 [ r3 ] = r7 } \texttt{ld8 r1 = [ r2 ]}</td>
</tr>
<tr>
<td><strong>Transformed:</strong></td>
<td></td>
</tr>
<tr>
<td>\texttt{ld8.s r1 = [ r2 ]} \ldots \texttt{chk.s r1, recovery}</td>
<td>\texttt{ld8.a r1 = [ r2 ]} \ldots \texttt{chk.a r1, recovery}</td>
</tr>
</tbody>
</table>

---

Hot Chips 1999
Control Speculation

Example:

– Suppose `br.cond` is a check for a null pointer
– Suppose the load of `r1` is dereferencing that pointer and then using it
– Normally, the compiler cannot reschedule the load before the branch because of potential fault

Control speculation is ...

– Moving loads (and possibly instructions that use the loaded values) above branches on which their execution is dependent
Control Speculation: Step 1

Separate load behavior from exception behavior
- ld.s which defers exceptions
- chk.s which checks for deferred exceptions

Exception token propagates from ld.s to chk.s
- NaT bits in General Registers, NaTVal (Special NaN value) in FP Registers
Control Speculation: Step 2

Reschedule ld8.s

– Now, ld8.s will defer a fault and set the NaT bit on r1
– chk.s checks r1’s NaT bit and branches/faults if necessary

Allows faults to propagate
– NaT bits in General Registers, NaTVal (Special NaN value) in FP Registers
Hoisting Uses

The uses of speculative data can also be executed speculatively.
Requires extra recovery code and chk.s.

```
ld.s
instr 1
instr 2
uses
br

chk.s
(Home Block)

Recovery code

ld
uses
br home
```
NaT Propagation

All computation instructions propagate NaTs to reduce number of checks required

```
ld8.s r3 = [r9]
ld8.s r4 = [r10] ;;
add r6 = r3, r4 ;;
ld8.s r5 = [r6]
p1,p2 = cmp....
(p1) br
```

Only one check needed

```
rec:
ld8 r3 = [r9]
ld8 r4 = [r10] ;;
add r6 = r3, r4 ;;
ld8 r5 = [r6]
br home
```

```
chk.s r5, rec
home:
sub r7 = r5,r2
```
Exception Deferral

Deferral allows the efficient delay of costly exceptions

OS-controlled deferral of data-related faults
  – Page faults
  – Protection violations
  – ...

NaTs/Chks enable deferral with recovery
Architectural Support for Control Speculation

65th bit (NaT bit) on each GR indicates if an exception has occurred

Special speculative loads that set the NaT bit if a deferable exception occurs

Special chk.s instruction that checks the NaT bit and branches to recovery, if set

Computational instructions propagate NaTs like IEEE NaN’s

Compare operations propagate “false” when writing predicates
Example:
- \texttt{st1} writes into memory
  - the
  - the store and the load addresses
  - load before the store
Such store to load dependences are

Data speculation is ...
- instructions that use the loaded stores
Data Speculation: Step 1

Separate load behavior from overlap detection
- \texttt{ld8.a} which performs normal loads and keeps bookkeeping (ALAT)
- \texttt{chk.a} which checks ALAT to see if conflicting store has occurred

Advanced load address table
- \texttt{ld8.a} puts information about advanced loads into table (address ranges accessed)
- stores and other memory writers ‘snoop’ ALAT and if overlapping loads are found, entries are deleted
- \texttt{chk.a} checks to see if a corresponding entry is in ALAT

\begin{verbatim}
instr1
instr2
... 
st1 [r3] = r4
ld8 r1 = [r2] ;;
add r3 = r1,r4
\end{verbatim}

\begin{verbatim}
instr1
instr2
...
st1 [r3] = r4
ld8.a r1 = r2 ;;
chk.a r1, recovery
add r3 = r1,r4
\end{verbatim}
Data Speculation: Step 2

Reschedule ld8.a

- Now, ld8.a will allocate an entry in the ALAT
- If the st1 instruction overlaps with the ld8.a address, then the ALAT entry will be removed
- chk.a checks for matching entry in ALAT -- if found, speculation was ok, if not found, need to re-execute
Data Speculation

Original  Transform  Reschedule

store  load  use

store  load  use

chk.s  use

chk.s

Hot Chips 1999
Hoisting Uses

Uses can be hoisted, but then chk.a needed for recovery

No hoisted uses

<table>
<thead>
<tr>
<th>ld8.a r1= instr 1 instr 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>st8</td>
</tr>
<tr>
<td>ld.c r1= uses =r1</td>
</tr>
</tbody>
</table>

With hoisted uses

<table>
<thead>
<tr>
<th>ld8.a r1= instr 1 uses =r1 instr 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>st8</td>
</tr>
<tr>
<td>chk.a r1,rec</td>
</tr>
</tbody>
</table>

Recovery code

| ld8 r1= uses =r1 br home           |
Architectural Support for Data Speculation

ALAT - HW structure containing information about outstanding advanced loads

Instructions
- ld.a - advanced loads
- ld.c - check loads
- chk.a - advance load checks

Speculative Advanced loads - ld.sa - is an control speculative advanced load with fault deferral (combines ld.a and ld.s)
IA-64 Application Architecture Tutorial

Application State
Instruction Format
Integer Instructions
Execution Semantics
Control Speculation, Data Speculation
Predication
Parallel Compares
Branch Architecture

Hot Chips 1999
Predication Concepts

Branching causes difficult to handle effects
  – Istream changes (reduces fetching efficiency)
  – Requires branch prediction hardware
  – Requires execution of branch instructions
  – Potential branch mispredictions

IA-64 provides predication
  – Allows some branches to be moved
  – Allows some types of safe code motion beyond branches
  – Basis for branch architecture and conditional execution
Predication

\[
\text{cmp.eq} \\
(p1) \text{ add } \ r7 = r2, \ r4 \\
\text{sa}
\]

If p1 is performed, else it acts as a nop
If p2 is performed, else it acts as a nop
Control Flow Simplification

Predication

- Change control flow dependences into data dependences
- Removes branches
  - reduce/eliminate mispredictions and branch bubbles
  - instruction fetch efficiency
  - exposes parallelism

Hot Chips 1999
Multiple Selects

Original

Transform/Reschedule

cmp p1,p2 = ;;
(p2) cmp p3,p4 =

(p1) r8 = 5
(p3) r8 = 7
(p4) r8 = 10
Downward Code Motion

Original

Transform

Reschedule

Hot Chips 1999
Architectural Support

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

Compare and test instructions write 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 is squashed
IA-64 Application Architecture Tutorial

Application State
Instruction Format
Integer Instructions
Execution Semantics
Control Speculation, Data Speculation
Predication
Parallel Compares
Branch Architecture
Parallel Compares

Parallel compares allow compound conditionals to be executed in a single instruction group.

Example:

```c
if ( a && b && c ) { . . . }
```

Assembly:

```assembly
cmp.eq pl = r0,r0 ;; // init pl=1
cmp.ne.and pl = rA,0
cmp.ne.and pl = rB,0
cmp.ne.and pl = rC,0
```
Height Reduction

Original

cmp pA = (pA) br. cond

cmp pB = (pB) br. cond

cmp pC = (pC) br. cond

Transform/Reschedule

cmp.and pABC = (pABC) br. cond

cmp.and pABC =

cmp.and pABC =

Hot Chips 1999
Architectural Support

Compare
- equality: eq, ne
- relational: only against zero
- tbit and tnat

Allows for both ‘and’ and ‘or’ compares
- one side: and
- one side: or
- both sides of conditional: or.andcm, and.orcm
IA-64 Application Architecture Tutorial

Application State
Instruction Format
Integer Instructions
Execution Semantics
Control Speculation, Data Speculation
Predication
Parallel Compares
Branch Architecture

Hot Chips 1999
Branch Architecture

IP-offset branches (21-bit disp.)

Branch registers
  – 8 registers for indirect jumps, call/ret link

Multi-way branches
  – Bundle 1-3 branches in a bundle
  – Allow multiple bundles to participate
Branch Execution

Unconditional branch

(cmp p1 = cond)
(p0) br target;

Conditional branches

(cmp p1 = cond)
(p1) br target;

Compare and branch can be in same instruction group

Compiler-directed static prediction w/dynamic prediction
  – Reduced false mispredicts due to aliasing
  – Frees space in H/W predictor
  – Can give hint for dynamic predictor
Multiway Branches

Allow multiple branch targets to be selected in one instruction group

Example:

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

Four possible instructions executed next:
- fall through, target_1, target_2, or address in b1

Hot Chips 1999
Control Height Reduction

**w/o Speculation**

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

P1

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

P3

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

P5

**Hoisting Loads**

- ld8.s r6 = (ra)
  
  ld8.s r7 = (rb)
  
  ld8.s r8 = (rc)

- chk r6, rec0
  
  (p1) br exit1

P2

- Chk r7, rec1
  
  (p3) br exit2

P4

- Chk r8, rec2
  
  (p5) br exit3

P6

**IA-64**

- ld8.s r6 = (ra)
  
  ld8.s r7 = (rb)
  
  ld8.s r8 = (rc)

- chk r6, rec0
  
  (p2) chk r7, rec1
  
  (p4) chk r8, rec2

- (p1) br exit1
  
  (p3) br exit2
  
  (p5) br exit3

3 branch cycles

1 branch cycle
Control Height Reduction

Original

Transform /Reschedule

Multiways Added

Hot Chips 1999
Notes:

- Multiple branches per clock is a natural side-effect of speculation
- Allows fast selection of multiple branch targets
- Branch prediction for both single and multiple branches is important for good performance
- Compiler profiling can help facilitate the use of hints
- Hints may reduce needed size/functionality of hardware predictors
- Works in conjunction with control speculation, data speculation, predication, and parallel compares
Agenda for This Tutorial

- IA-64 history and strategy
- IA-64 application architecture overview
- C -> IA-64 example
- Reference slides included (but not covered)
if (((Theory->Flags[ChunkNum] & 0x0008))
    && ((Theory->Flags[ChunkNum] & 0x0040))
    && (*(Theory->ChunkAddr[ChunkNum] - 28)) == SizeOfUnit) {
    StackPtr = (*(Theory->ChunkAddr[ChunkNum] - 20));
    if (Index >= StackPtr) {
        if (SetGetSwi)
            *Status = -10009;
        else {
            Mem_DumpChunkChunk (0, ChunkNum);
            *Status = 1005; }
    } else {
        if (*(Theory->ChunkAddr[ChunkNum] - 28)) != SizeOfUnit) {
            *Status = 1003;
        } else {
            *Status = 1004; }
        Mem_DumpChunkChunk (0, ChunkNum);
    }
}
return((Test= *Status==0 ? True: Ut_PrintErr (F,Z,*Status)));
Synthesis: ChkGetChunk()

Assumptions for code examples

- Abstract machine model
- Unlimited instruction issue (execution) resources
- Loads have 2 cycle latency to first level cache
- All other instructions 1 cycle latency
Synthesis

Hot Chips 1999
Synthesis: ChkGetChunk()

ld8  rT  = [ &rT  ]        0:B1
ld8  rCN = [ &rCN ] ;;   0:B1
add  rAdF  = rT , 8        2:B1
shladd rOff  = rCN , 4;;     2:B1
add  rAdFs = rOff,rAdF;;     3:B1
ld8  rFD = [ rAdFs ] ;;  4:B1
and  rMask8 = rFD ,0x8 ;; 6:B1
cmp.eq p1,p2 = rMask8,0    7:B1
(p1) br.cond GREEN
and  rMask4 = rFD,0x40;;  0:B2
cmp.eq p3,p4 = rMask4,0    1:B2
(p3) br.cond GREEN
add  rAdCA = 16, rT ;;   0:B3
add  rAdCAS= rAdCA,rOff;; 1:B3
ld8  rTmp  = [rAdCAS] ;;  2:B3
sub  rTmp2 = rTmp , 28 ;; 4:B3
ld8  rDR = [ rTmp2 ];; 6:B3
cmp.eq p5,p6 = rDR,VAL 7:B3
(p5) br.cond GREEN
// fallthru is TURQUOISE

ld8  rT  = [ &rT  ]        0:B1
ld8  rCN = [ &rCN ] ;;   0:B1
add  rAdCA = 16, rT 2:B3
add  rAdF  = rT , 8 2:B1
shladd rOff  = rCN , 4;;     2:B1
add  rAdFs = rOff,rAdF;;     3:B1
ld8  rFD = [ rAdFs ] 2:B1
ld8.s rTmp  = [rAdCAS] ;;  4:B1
and  rMask4 = rFD,0x40 6:B2
and  rMask8 = rFD,0x8 6:B1
sub  rTmp2 = rTmp , 28 ;; 6:B3
cmp.eq p3,p4 = rMask4,0 7:B2
cmp.eq p1,p2 = rMask8,0 7:B1
ld8.s rDR = [ rTmp2 ];; 7:B3
cmp.eq p5,p6 = rDR,VAL 9:B3
(p1) br.cond GREEN
(p3) br.cond GREEN
(p5) br.cond GREEN
// 19 instructions/10 cycles
// fallthru is TURQUOISE
Synthesis: ChkGetChunk()

```
ld8  rT  = [ &rT  ]        0:B1
ld8  rCN = [ &rCN ] ;;    0:B1
add  rAdCA  = 16,  rT      2:B3
add  rAdF  =  rT , 8        2:B1
shladd rOff  =  rCN , 4;;    2:B1
add  rAdCAs =  rAdCA , rOff    3:B3
add  rAdFs  =  rOff , rAdF;;  3:B1
ld8  rFD  = [ rAdFs ]     4:B1
ld8.s rTmp  = [rAdCAs] ;;   4:B3
and  rMask4 =  rFD ,0x40   6:B2
and  rMask8 =  rFD ,0x8    6:B1
sub  rTmp2 =  rTmp , 28 ;;  6:B3
cmp.eq p3,p4 = rMask4,0      7:B2
cmp.eq p1,p2 = rMask8,0      7:B1
ld8.s rDR  = [ rTmp2 ] ;;   7:B3
cmp.eq p5,p6 = rDR,VAL       9:B3
(p1)  br .cond  GREEN
(p3)  br .cond  GREEN
(p5)  br .cond  GREEN
// 19 instructions/10 cycles: < 2 IPC
// fallthru is TURQUOISE
```
Synthesis

Hot Chips 1999
Synthesis: ChkGetChunk()

Steps:
1) Speculate ld8 instructions into red block cycle 1
2) Copy and speculate mov instructions into red blocks cycle 0
3) Speculate cmp instruction into red block cycle 9
4) Predicate both sides of the conditional
Synthesis: ChkGetChunk()

Steps:
1) Replace rTest with r8
2) Predicate both sides of conditional
Synthesis: ChkGetChunk()

Steps:
1) speculate all the `ld8`'s in to red block cycle 1
2) speculate the `cmp.eq` in to red block cycle 3
3) copy and speculate the `mov rVal` in to red block cycle 0
4) predicate both sides of conditional
Synthesis

call+1

10

4

call + 1

call + 1

Hot Chips 1999
Steps:
1) speculate `sub`, `ld8`, and `cmp.ge` into the red block (cycles 6, 7, and 9)
2) predicate the `st8` with `(p2)`
3) concatenate with blue block
4) (cont. next page)
Synthesis: ChkGetChunk()

Steps:
1) qualify p13 and p12 (now in red block) so they can only be true when both p2 and p14 are true (use parallel and-compares) by either of the following:
   – adding 4 parallel compares to red block
   – lengthening red block by 1 cycle
2) now it is safe to remove the first PURPLE and BLUE branches
Synthesis
Synthesis: ChkGetChunk()

Steps:
1) Note that \( p_2 \) is the ‘blue/turquoise’ branch predicate and that \( p_{12} \) and \( p_{13} \) are qualified with \( p_2 \) already
2) Predicate \( \text{br . call } DC(CN) \) with \( p_1 \)
3) If we further qualify \( p_8 \) and \( p_9 \) with \( p_1 \) (the ‘green branch’) in the red block, then the green and blue instructions are guaranteed to be independent! Can be done by either:
   – adding 3 parallel compares to red block
   – lengthening red block by 1 cycle
Synthesis

Cycles = 12 + 2 calls
Agenda for This Tutorial

IA-64 history and strategy

IA-64 application architecture overview

C -> IA-64 example

Reference slides included (but not covered)
  - Loop Support
  - Register Stack
  - Memory Support
  - Floating Point, Multi-media, 3D Graphics

Hot Chips 1999
Key IA-64 Features

Loop Support*

Register Stack*

Memory Support

Floating Point, Multi-media, 3D Graphics

* Some slides provided by Dale Morris, HP Cupertino
Register Rotation

Motivation:
- pipeline-schedule loops onto HW
- remove extraneous work from loop
- minimize start-up overhead
- small code footprint
- maximum computational throughput with few instructions
GR Stack Frame w/ Rotation

Current Frame Marker (CFM)

Size of Rotating (sor)
GR Rotation

Size of rotating region multiple of 8
Rotating region overlays current frame

– Overlay allows rotation & stack renaming in a single level of

– Must copy input registers before loop
Rotating

Static

Upper 3/4 of register file rotates
Predicate Rotation

Rotating

Upper 3/4 of register file rotates

Static

Hot Chips 1999
Register Rotation & RRB

Separate Rotating Register Base for each: GRs, FRs, PRs
Loop branches decrement all register rotating bases (RRB)
Instructions contain a “virtual” register number

Palm Sunny Springs

RRB=0

Hot Chips 1999
Register Rotation & RRB

Separate Rotating Register Base for each: GRs, FRs, PRs
Loop branches decrement all register rotating bases (RRB)
Instructions contain a “virtual” register number

```
lsl  R34
st   R35
```

RRB=0
Register Rotation & RRB

Separate Rotating Register Base for each: GRs, FRs, PRs
Loop branches decrement all register rotating bases (RRB)
Instructions contain a “virtual” register number

\[
\text{Palm Springs} \quad \begin{array}{c}
\text{ld} \quad \text{st}_2 \quad \text{R35} \\
\text{Id} \quad \text{R34}
\end{array}
\]

RRB = -1

Hot Chips 1999
Separate Rotating Register Base for each: GRs, FRs, PRs
Loop branches decrement all register rotating bases (RRB)
Instructions contain a “virtual” register number
- RRB + virtual register number = physical register number.

```
        is
        st  R35
        ld4 R34
```

```
Palm | Springs | is | Sunny
34: Springs
33: is
32: Sunny
127:
126:
```
Separate Rotating Register Base for each: GRs, FRs, PRs
Loop branches decrement all register rotating bases (RRB)
Instructions contain a “virtual” register number
   RRB + virtual register number = physical register number.
Loop Branches

br.cloop uses LC for simple, non-pipelined loops
   – decrements LC and loops until LC is 0
br.ctop uses LC and EC for pipelined counted loops
br.wtop uses branch predicate and EC for pipelined “while” loops
br.cexit, br.wexit used for unrolled, pipelined loops
Function (simplified):

- if (LC>0) {
    LC--; pr[63]=1; rrb--; loop;
} else if (EC>0) {
    EC--; pr[63]=0; rrb--; loop;
} else
    fall_through;

LC counts main loop iterations
EC counts pipeline stages for drain
Software Pipelining

Overlapping execution of different loop iterations

More iterations in same amount of time

Hot Chips 1999
Software Pipelining

Synergistic use of IA-64 features:
- Full Predication
- Special branches
- Register rotation: removes loop copy overhead
- Predicate rotation: removes prologue & epilogue

Traditional architectures use loop unrolling
- High overhead: extra code for loop body, prologue, and epilogue

Especially Useful for Integer Code With Small Number of Loop Iterations
Pipelined Loop Example

DAXPY inner loop
- \( dy[i] = dy[i] + (da \times dx[i]) \)
- 2 loads, 1 fma, 1 store / iteration

Machine assumptions
- can do 2 loads, 1 store, 1 fma, 1 br / cycle
- load latency of 2 clocks
- fma latency of 1 clocks (not realistic, but good for example)
Example: Pipeline

Each column represents 1 source iteration

load dx, dy

tmp = dy + da * dx

store dy
Example Code

```
.rotf dx[3], dy[3], tmp[2]

    mov   ar.lc = 3         // #iterations-1
    mov   ar.ec = 4         // #stages
    mov   pr.rot = 0x10000
    ;;
    looptop:

    (p16) ldfd   dx[0] = [dxsp],8
    (p16) ldfd   dy[0] = [dysp],8
    (p18) fma.d  tmp[0] = da, dx[2], dy[2]
    (p19) stfd   [dydp] = tmp[1],8
       br.ctop looptop
    ;;
```

Hot Chips 1999
Loop Execution

Initialization

RRB=0
19: 0
18: 0
17: 0
16: 1
63: 0

(p19)
(p18)
(p18)
(p16)
(p63)

LC=3   EC=4

Execution Sequence
(p16) \text{ld}_x
(p16) \text{ld}_y
(p18) fma
(p19) st
Loop Execution

Execution Sequence

(p16) ld\_x \quad (p16) ld\_y \quad (p18) fma \quad (p19) st

RRB=-1 

LC=2 \quad EC=4

Branch 1
Loop Execution

Execution Sequence

(p16) ld_x
(p16) ld_y
(p18) fma
(p19) st
(p16) ld_x
(p16) ld_y
(p18) fma
(p19) st
(p16) ld_x
(p16) ld_y
(p18) fma
(p19) st

RRB=-2
LC=1
EC=4

Branch 2
Execution Sequence

(p16) ld
(p16) ld
(p16) ld
(p18) fd
(p18) fma
(p19) st
(p19) st
(p19) st

RRB=-3
LC=0

Branch 3
Loop Execution

Execution Sequence

(p16) ld      ld_{y}     fma     (p19)
(p16) ld      ld_{y}     fma     (p19)
(p16) ld      ld_{y}     (p18) fma     (p19) st
(p16) x       ld_{x}     x       (p16) ld
(p16) y       (p16) y     (p16) ld
(p16) x       fma     st

RRB=-4

LC=0   EC=3
Loop Execution

Execution Sequence

Branch 5

RRB=-5
LC=0 EC=2
Loop Execution

Execution Sequence

... (p16) \(ld_x\) (p16) \(ld_y\) (p18) fma (p19) st

RRB = -6
EC = 1

Branch 6
Loop Execution

**Execution Sequence**

- `ld (p18) st (p19)
- `ld x (p16) y fma st
- `ld (p18) st (p19)
- `ld fma st
- `ld st
- `ld st
- `ld st
- `ld st
- `ld st
- `ld st
- `ld st

**Loop Execution**

- `RRB=-7 LC=0 EC=0
- `Branch 7
Pipelining & Latency

Suppose we change the latencies

− load latency of 6 clocks
− fma latency of 4 clocks
Example: New Pipeline

Each column represents 1 source iteration

load \, dy

= dy \, da *

store dy
```
.rotf dx[7], dy[7], tmp[5]

mov   ar.lc = 3          // #iterations-1
mov   ar.ec = 11         // #stages
mov   pr.rot = 0x10000
;;
looptop:
   (p16) ldfd   dx[0] = [dxsp],8
   (p16) ldfd   dy[0] = [dysp],8
   (p22) fma.d  tmp[0] = da, dx[6], dy[6]
   (p26) stfd   [dydp] = tmp[4],8
   br.ctop  looptop
   ;;
```

Updated Loop
Rotation: Summary

Loop pipelining maximizes performance; minimizes overhead
- Avoids code expansion of unrolling and code explosion of prologue and epilogue
- Smaller code means fewer cache misses
- Greater performance improvements in higher latency conditions

Reduced overhead allows S/W pipelining of small loops with unknown trip counts
- Typical of integer scalar codes
Key IA-64 Features

Loop Support*

Register Stack*

Memory Support

Floating Point, Multi-media, 3D Graphics

* Some slides provided by Dale Morris, HP Cupertino

Hot Chips 1999
IA-64 Register Model

Stack & Rotation support
SW-visible renaming resources
HW simplicity and explicit control
Register Stack

Motivation:
- Automatic save/restore of GRs on procedure call/return
- Cache traffic reduction
- Latency hiding of register spill/fill
General Registers

- Stacked
- Static
GR Stack Frame

127
illegal

outputs

locals
(inputs)

32
31

Static

0

size of frame (sof)

size of locals (sol)

Current Frame Marker (CFM)

sol

sof

Hot Chips 1999
GR Stack Frame - Example

- Size of frame (sof): 52
- Size of locals (sol): 46
- CFM: 32

Diagram:
- Out: 14
- Loc: 21
GR Stack Frame - Call

![Diagram showing stack frame with locations and labels for out, loc, sol, sof in different sections.]

Hot Chips 1999
GR Stack Frame - Allocate

Hot Chips 1999
GR Stack Frame - Return

Hot Chips 1999
Instructions

br.call
  - Copies CFM to PFM
  - Creates new frame with only output regs
  - Saves local regs from previous frame

alloc
  - Resizes current frame
  - Saves PFM to a GR
Instructions (cont.)

mov to PFS
   – Restores PFM from a GR
br.ret
   – Restores CFM from PFM
   – Restores local regs for previous frame
Key IA-64 Features

Loop Support

Register Stack

Memory Support

Floating Point, Multi-media, 3D Graphics
Memory

Byte addressable
Accessed with 64-bit pointers
  – Upper 3-bits is segment id
  – Limited support for 32-bit pointers
Access granularity and alignment
  – 1, 2, 4, 8, 10, 16 bytes
  – Alignment on naturally aligned boundaries is recommended
    • Performance penalty may result if not
  – Instructions are always 16-byte aligned
Accessed big or little endian byte order
32-bit virtual addressing support
Memory Hierarchy Control

Explicit control of cache allocation and deallocation
  – Specify levels of the memory hierarchy affected by the access
  – Allocation and Flush resolution is at least 32-bytes

Allocation
  – Allocation hints indicate at which level allocation takes place
    • But always implies bringing the data close to the CPU
  – Used in load, store, and explicit prefetch instructions

Deallocation and Flush
  – Invalidates the addressed line in all levels of cache hierarchy
  – Write data back to memory if necessary
Key IA-64 Features

- Loop Support
- Register Stack
- Memory Support
- Floating Point, Multi-media, 3D Graphics
Floating-point Architecture

IEEE 754 compliant
Single, double, double extended (80-bit)
Canonical representation in 82-bit FP registers
Multiply-add instruction
128 floating-point registers
  – Rotating, not stacking
Load double/single pair
Multiple FP status registers for speculation
Multimedia Support

Audio and video functions typically perform the same operation on arrays of data values.

IA-64 defines a set of instructions to treat general register’s as 8x8, 4x16, or 2x32 bit elements.

- Three major types of instructions are defined:
  - Addition and subtraction (including special purpose forms)
  - Left shift, signed and unsigned right shift
  - Pack/Unpack; converts between different element sizes.

Semantically compatible with IA-32’s MMX Technology.
Parallel FP Support

Enable Cost-effective 3D Graphics platforms

Exploit data parallelism in applications using 32-bit floating-point data

- Most applications and geometry calculations (transforms and lighting) are done with 32-bit floating-point numbers
- Provides 2X increase in computation resources for 32-bit data parallel floating-point operations

Floating-point Registers treated as 2x32 bit single precision elements

- Full IEEE compliance
  - single, double, double-extended data types, packed-64
- similar instructions as for scalar floating-point
- availability of fast divide (non IEEE)