IA-64 Architecture and Compiler Technology

Roy D.C. Ju
Compilers and Java Lab
Microprocessor Research Labs
Intel Corporation
roy.ju@intel.com

IA-64 Definition History

• Two concurrent 64-bit architecture developments:
  • IAX at Intel from 1991
  • 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

IA-64 Strategies

• 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

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/PMID)

- Memory
  http://developer.intel.com/design/ia64/download/adag.htm
  http://developer.intel.com/design/IA-64/microarch_ovw/index.htm

Instruction Formats: Bundles

• Instruction types
  M: Memory
  I: Shifts, MM
  A: ALU
  B: Branch
  F: Floating point
  L+X: Long

• Instruction 2
  41 bits

• Instruction 1
  41 bits

• Instruction 0
  41 bits

• Template
  5 bits

• 128 bits

• Instruction types
  Template types
  Regular: MI, MLX, MMI, MFI, MMF
  Stop: MI, M, MI
  Branch: MIB, MMB, MFB, MBB
  BBB

• All come in two versions:
  • with stop bit at end
  • without stop bit at end

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

Case 1 - Dependent
add r1 = r2, r3
sub r4 = r5, r1
shl r2 = r4, r6

Case 2 - Independent
add r1 = r2, r3
sub r4 = r5, r1
shl r2 = r4, r6
shl r12 = r14, r8

Case 3 - Predication
(p1) add r1 = r2, r3
(p2) sub r4 = r5, r1
(p3) shl r2 = r4, r6
(p4) shl r12 = r14, r8
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
- 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

Predication Concepts

- Branching causes difficulty to handle effects
  - I-stream changes (reduces fetching efficiency)
  - Requires branch prediction hardware
  - Requires execution of branch instructions
  - Potential branch mis-predictions
- IA-64 provides predication
  - Allows some branches to be removed
  - Allows some types of safe code motion beyond branches
  - Basis for branch architecture and conditional execution

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 deferrable 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

Architectural Support for Data Speculation

- ALAT - HW structure containing information about outstanding loads advanced across stores
- Instructions
  - ld.a - advanced loads
  - ld.c - check loads
  - chk.a - advance load checks
  - aliasing st invalidating entries in ALAT
- Speculative advanced loads - ld.sa - is a control speculative advanced load with fault deferral (combines ld.a and ld.s)

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

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 to resolve among multiple branch targets
  - Allow multiple bundles to participate
Other IA-64 Features

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

Overview of IA-64 Compiler Technology

- Predication
  - If-conversion
  - Uses regular, unconditional and parallel compares
- Software pipelining
  - Modulo scheduling and rotating register allocation
  - Uses rotating registers, stage predicates, loop branches
- Global instruction scheduling
  - Instruction scheduling across basic block boundaries
  - Uses control and data speculation, predication, post-increments, multi-way branches
- Global register allocation
  - Allocate registers for predicated code
  - Uses register stack

(1) Predication

- When branch mis-prediction rate is high, it is better to predicate
- Predication explores and creates more ILP
- Predication has the potential cost of increasing the critical path length
- Techniques
  - If-conversion
  - Parallel compare to reduce control height

If-conversion for Predication

- Identifying region of basic blocks based on resource requirement and profitability (branch mis-prediction rate, mis-prediction cost, and parallelism)
- Result: a predicated block

Reducing Control Height with parallel compares

- Convert nested if's into a single predicate
- Result: shorter control path by reducing the number of branches

Multiway Branch Example

- Use Multiway branches
  - Speculate compare (i.e. move above branch)
  - Do not reduce number of branches
(2) Software Pipelining

- Exploit parallelism across iterations without code bloat
- Architectural features used:
  - Rotating registers
  - Rotating predicates (stage predicates)
  - Predication to collapse control flow
  - Counted loop
  - While loop
- Modulo scheduling
- Rotating register allocation

(3) Global Code Scheduling

- Objectives
  - Increase parallelism
  - Fully use machine width
- Needs
  - Accurate machine model
- Uses architectural features
  - Large number of registers
  - Control and data speculation, checks and recovery code
  - Predicates
  - Multi-way branches

Overlapping Loop Iterations

- Exploit parallelism across loop iterations
- Result: kernel-only code without code expansion

Region Formation

- Scheduling regions are acyclic

Speculative Upward Code Movement

- Speculate both the load and the use
- Result: efficient use of machine resources
Predicated Upward Code Movement

- Guard with fall-through predicate
- Motion bounded by compare
- Result: predication can avoid speculative side effects

Example of Instruction Scheduling

- Control Flow Graph

Example of Instruction Scheduling

- with predication and possible speculation

(4) Global Register Allocation

- Objective
  Eliminate memory references. Minimize register spill and copying after load-store elimination
  Considering architectural features
  Large number of registers
  Predication

Register Allocation Example

- Modeled as a graph-coloring problem
  Nodes in the graph represent live ranges of variables
  Edges represent temporal overlap of the live ranges
  Nodes sharing an edge must be assigned different colors (registers)
Register Allocation Example

With Control Flow

\[
x = ... \\
y = ... \\
z = ... x
\]

Requires Two Colors

Register Allocation Example

With Predicate Analysis

\[
x = ... \\
y = ... \\
z = ...
\]

Now Back to Two Colors

Register Allocation Example

With Predication

\[
x = ... \\
y = ... \\
z = ...
\]

Now Requires Three Colors

Register Allocation Example

With Predicate Analysis

\[
x = ... \\
y = ... \\
z = ...
\]

Now Back to Two Colors

Summary

• Compiler is important for IA-64 performance
  backend should take full advantages of IA-64 architectural features to generate optimal code

• If-conversion
  predication with various compares

• Software pipelining
  rotating registers, stage predicates, loop branches

• Global scheduling across basic block boundaries
  control and data speculation, post-increments, multi-way branches,

• Global register allocation
  register stack, predication

Predicate Analysis for the Example

\[
x = ... \\
y = ... \\
z = ...
\]

p1 and p2 are disjoint
If p1 is TRUE, p2 is false and vice versa