Predication and Speculation

Last time
- Instruction scheduling
- Profile-guided optimizations
  - How can we increase our scheduling window?
  - How can we move excepting instructions (loads) above splits?

Today
- Brief history of computer architecture
- Predication and speculation
- Compiling for IA-64

A Brief History of Computer Architecture

The Early Years: CISC
- Programmed by humans
- Feature bloat:
  - Provide many instructions
  - Provide many addressing modes
  - Variable length instructions
  - Complex instructions
    - VAX: REMQHI, EDITPC, POLYF

Problem
- Difficult to implement efficiently
- Difficult to pipeline
- Difficult to generate good code for
A Brief History of Computer Architecture (cont)

The Early 1980s: RISC
- Simplify the ISA to facilitate pipelining
  - Uniform instruction format simplifies decoding
  - Uniform instructions easier to pipeline
  - Pipelining improves clock speeds

Uniform ISA Simplifies Compilation
- Stanford: Produce an architecture that leverages their strong compiler group
- Berkeley: Produce an architecture that does not require heroic compilation

Problems
- Uncertain latency
- No binary compatibility

A Brief History of Computer Architecture (cont)

The 1990’s: Dynamic Superscalar
- Simplified pipelining and more transistors enable hardware scheduling
  - Re-order instructions
  - Hardware speculation (branch prediction)
  - Increased issue width

Note
- We’re talking about implementation trends here, not changes in the architecture

Problems
- The bureaucracy problem
  - More and more resources being devoted to control and management
  - Fewer and fewer resources being devoted to actual work
  - ILP limited (typically between 1 and 2)
A Brief History of Computer Architecture (cont)

The 1990’s: CISC implemented on RISC core
- Provide binary compatibility
- Dynamically translate CISC instructions to RISC instructions
- Best of both worlds?

Note
- This again is a microarchitectural change, not an architectural change

Problems
- Hardware complexity
  - Hardware still needs to discover parallelism
  - Still have the $n^2$ scheduling problem
  - Still difficult to compile for

Implicitly Sequential Instruction Stream

Problems
- Compilers can expose parallelism
- Compilers must eventually emit linear code
- Hardware must then re-analyze code to perform OoO execution
  - Hardware loses information available to the compiler
  - Compiler and hardware can only communicate through the sequential stream of instructions, so hardware does redundant work

How can we solve this problem?
Explicitly Parallel Instruction Stream

source code → compiler → parallel machine code → hardware

A solution
- Hardware does not need to re-analyze code to detect dependences
- Hardware does not perform OoO execution

VLIW: Very Long Instruction Word
- Each instruction controls multiple functional units
- Each instruction is explicitly parallel

VLIW

Basic idea
- Each instruction controls multiple functional units
- Rely on compilers to perform scheduling and to identify parallelism
- Simplified hardware implementations

Benefits
- Compiler can look at a larger window of instructions than hardware
- Can improve the scheduler even after a chip has been fabricated

Problems
- Slow compilation times
- No binary compatibility
- Difficult for compilers to deal with aliasing and long latencies
- Code is implementation-specific
VLIW and IA-64

VLIW
− Big in the embedded market
  − Binary compatibility is less of an issue
− An old idea
  − Horizontal microcode
  − Multiflow (1980’s)
  − Intel i860 (early 1990’s)

Terminology
− EPIC: Explicitly Parallel Instruction Computer
  − New twist on VLIW
  − Don’t make code implementation-specific
− IA-64 is Intel’s EPIC instruction set
− Itanium is the first IA64 implementation

Explicitly Parallel Instruction Sets: IA-64

IA-64 Design Philosophy
− Break the model of implicitly sequential execution
  − Use template bits to specify instructions that can execute in parallel
  − Issue these independent instructions to the FPU’s in any order
  − (Templates will cause some increase in code size)
− The hardware can then grab large chunks of instructions and simply feed them to the functional units
  − Hardware does not spend a lot of time figuring out order of execution; hence, simplified hardware control
  − Statically scheduled code
− Hardware can then provide a larger number of registers
  − 128 (about 4 times more than current microprocessors)
  − Number of registers fixed by the architecture, but number of functional units is not
IA-64

A return to hardware “simplicity”
- Revisit the ideas of VLIW
- Simplify the hardware to make it faster
- Spend larger percentage of cycles doing actual work
- Spend larger percentage of hardware on registers, caches, and FPU’s
- Use larger number of registers to support more parallelism

Engineering goal
- Produce an “inherently scalable architecture”
- Design an architecture—an ISA—for which there can be many implementations
- This flexibility allows the implementation to change for “years to come”

Two Key Performance Bottlenecks

Branches
- Modern microprocessors perform good branch prediction
- But when they mispredict, the penalty is high and getting higher
  - Penalties increase as we increase pipeline depths
  - Estimates: 20-30% of performance goes to branch mispredictions [Intel98]
- Branches also lead to small basic blocks, which restrict latency hiding opportunities

Memory latency
- CPU speed doubles every 18 months (60% annual increase)
- Memory speed increase about 5% per year
Control dependences inhibit parallelism

Don’t know whether to execute instr3 or instr5 until the cmp is completed

Branches Limit Performance

Predicated Execution

Add a predicate flag to each instruction

If predicate is true, the instruction is executed

If predicate is false, the instruction is not executed

Predicates are simply bits in a register

Converts control flow into data flow

Exposes parallelism

With predicate flags, instr3 – instr7 can all be fetched in parallel

Benefits?

Fewer branches (fewer mispredictions)

Larger basic blocks

More parallelism

Idea

This is called if-conversion
The Memory Latency Problem

Memory Latency

- Writes can be done out of order and can be buffered
- Loads are the problem: processor must wait for loads to complete before using the loaded value
- Standard latency-hiding trick: issue non-blocking load as early as possible to hide latency

The Problem

- Loads typically issued at beginning of basic block
- Can’t move the Load outside the basic block
  - If the Load were to cause an exception when the basic block is not executed, then the early Load causes an erroneous exception

(Redex) Speculative Loads

Split-phase operation

- Standard trick in parallel computing
- Issue the load (load.s) as early as you wish
- Detect any exception and record it somewhere with the target of the load
- Can later check to see whether the load completed successfully: chk.s

Benefits?

- More freedom to move code—can now move Loads above branches as long as the check is in the original basic block
- Complication: What happens if chk.s is issued without a corresponding load.s?
  - This is clearly an error, so we need to be careful about where we move the load.s
**Data) Speculative Loads**

**Issue**
- Want to speculate that a load is not data dependent on a preceding store

**Split-phase operation**
- Issue early advanced load (record entry in advanced load address table (ALAT))
- Clear corresponding ALAT entries at store
- Check instruction looks for ALAT entry (branch to patch code if not found)

**Note**
- Can speculate instructions that depend on load, too

---

**Predication is an Old Idea**

**High performance computing**
- SIMD machines (Single Instruction Multiple Data)
  - All processors operate in lock-step but operate on different data
  - What do you do with control flow?

```
if (A[i][j] < 0)
    A[i][j] = -A[i][j]
```

- Compute a mask of 0’s and 1’s
- Execute both halves of the control flow using the appropriate mask
- Can do this in either hardware or software

```
Mask[i][j] = (A[i][j] < 0)
```
Is Predication a Good Idea?

Where should we perform predication?

Runtime information helps
- Branch behavior
- Load latencies

Degree of predication depends on issue width
- The ISA can be implementation-independent
- But the compilers that emit code cannot be implementation-independent

Where should predication occur?

Hyperblocks
- not Extended Basic Blocks, because EBBs do not include any join nodes except possibly the head
- not Superblocks, because they involve only one path
- single entry, multiple exits, no nested inner loops

[Mahlke et. al. 92] paper
- Basic blocks put into hyperblock based on ...
  - instruction frequency (more frequent is better)
  - size of the basic block (smaller is better)
  - instruction characteristics (procedure calls and unresolvable memory accesses are bad)
Is Speculation a Good Idea?

What are the disadvantages of speculation?
- Wasted work

The real question: Who should perform speculation?
- The hardware can exploit runtime information
- The compiler can exploit a much larger scope

Speculation increases parallelism
- Increase performance by exploiting parallelism
- Other examples of this?
  - Asynchronous communication in parallel computing
  - Continuations in functional languages
  - Multi-tasking

Implications

IA64
- The ideas are not new
- The willingness to change the ISA is new and significant

Implications for compilers
- Increased role of the compiler
- More control over sequencing, prefetching, stores, branch prediction
  - Hardware doesn’t “undo” the compiler’s work

Future systems
- What is the right division of labor between the compiler and the hardware?
- How else can compilers be used to simplify the hardware and make the hardware more effective?
- Can we improve the communication between the compiler and hardware?
Epilogue

Intel announces 64-bit IA-32
– The end of IA-64

What went wrong with IA-64?

What does the future hold for Dynamic Superscalar? VLIW?

Concepts

Predication and speculation

Performance bottlenecks
– Branches, memory latency

IA-64 characteristics
– VLIW
– Support for data/control speculation (if-conversion), predication, and on and on...

Role of compiler
– Must work hard!
– Has less available information
– Has larger scope
Next Time

Lecture
– Dynamic compilation