CS/ECE 752
*Static ILP, Vectors, Multimedia*
Instructor: Prof. Wood

Computer Sciences Department
University of Wisconsin

Slides developed by Profs. Falsafi, Hill, Smith, Sohi, Vijaykumar and Wood of Carnegie Mellon University, Purdue University, and University of Wisconsin

Outline
• Software transformations to increase ILP
• VLIW and EPIC architectures
• Vectors
• Multimedia instruction sets

Software transformations
• Loop unrolling
• Software pipelining
• Trace scheduling
  □ needs non-excepting instructions

Loop Unrolling
Loop: LD F0, 0(R1)
  stall
ADD F4, F2, F0
SUB R1, R1, #8
SD 8(R1), F4
BNEZ R1, Loop

Load-Use Delay
### Loop Unrolling (4 times)

Loop: LD F0, 0(R1)
      ADDD F4, F0, F2
      SD 0(R1), F4
      LD F6, -8(R1)
      ADDD F8, F6, F2
      SD -8(R1), F8
      LD F10, -16(R1)
      ADDD F12, F10, F2
      SD -16(R1), F12
      LD F14, -24(R1)
      ADDD F16, F14, F2
      S D     0(R1), F4
      SD     -8(R1), F8
      SD     -16(R1), F12
      SD     -24(R1), F16
      SUB R1, R1, #32
      BE NE Z R1, Loop

• Reduces loop overhead
• Assumes iteration count is a multiple of four
• Adjust displacements of loads/stores
• Still have stalls, but provides opportunity to schedule instructions

### Loop Unrolling (reschedule)

Loop: LD F0, 0(R1)
      LD F6, -8(R1)
      LD F10, -16(R1)
      LD F14, -24(R1)
      ADDD F4, F0, F2
      ADDD F8, F6, F2
      ADDD F12, F10, F2
      ADDD F16, F14, F2
      SD 0(R1), F4
      SD -8(R1), F8
      SD -16(R1), F12
      SD -24(R1), F16
      SUB R1, R1, #32
      BE NE Z R1, Loop

3 instructions between load and use.

### Tree Height Reduction

- Reduce the height of the dependence computation tree
  - use arithmetic property of associativity

ADD R1, R2, R3  ADD R1, R2, R3
ADD R4, R1, R6  ADD R4, R6, R7
ADD R8, R4, R7  ADD R8, R1, R4

Takes 3 cycles  Takes 2 cycles

### Software Pipelining

Loop Unrolling:
- Overlaps loops if no inter-iteration dependence

Will not work for “recurrences”
e.g.
for (i = 1; i <= N; i++)
    sum = sum + a[i] * b[i]

Software pipelining will do better

Originally done by microcode people
Rediscovered by Davidson and later Monica Lam
Software Pipelining: What is it?

```
Sum = 0.0
for (I = 1; I <= N; I++)
    sum = sum + a[I] * b[I]

every iteration =>
    load a[I]
    load b[I]
    mult ab[I]
    add sum[I]
```

Software Pipelining: Timing

```
  a1   a2   a3
  b1   b2   b3
      *1

  a4    ....    aN
  b4    ....    bN
      *2    ....    *N-1
  +1    ....    +N-2
      +N-1   +N

Start-up block  pipeline  Finish-up block
```

Software Pipelining Cont.

```
Start-up block
load a[1]
load b[1]
load a[2]
load b[2]
mult ab[1]

Pipeline
for (I=3;I<=N);I++)
load a[I]
load b[I]
mult ab[I-1]
add sum[I-2]

Finish-up block
mult ab[N]
add sum[N-1]
add sum[N]
```

Software Pipelining: When?

- Will work for recurrences in loops
- Will not work for non-loop code
- Will not work well for loops with branches
- Pipeline depth depends on recurrence
Trace Scheduling

Statically schedule all code not just loops

What is the key impediment?

- Branches

Can we predict which way a branch goes?

- Common case (for error code)

Trace scheduling:

- Take the common case path (called a trace)
- Remove the branch
- Schedule code
- Check later for branch
- If the guess was wrong, execute "undo" code

Trace Scheduling: Example

<table>
<thead>
<tr>
<th>Original</th>
<th>Trace</th>
<th>Repair code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b[i] = &quot;old&quot;</code></td>
<td><code>b[i] = &quot;old&quot;</code></td>
<td><code>fixit:</code></td>
</tr>
<tr>
<td><code>a[i] =</code></td>
<td><code>a[i] =</code></td>
<td><code>restore old b[i]</code></td>
</tr>
<tr>
<td><code>if (a[i] &gt; 0) then</code></td>
<td><code>b[i] = &quot;new&quot;</code></td>
<td><code>X</code></td>
</tr>
<tr>
<td><code>b[i] = &quot;new&quot;</code></td>
<td><code>c[i] =</code></td>
<td><code>recalculate c[i]?</code></td>
</tr>
<tr>
<td><code>else</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>endif</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>c[i] =</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Use profiling or program analysis to statistically predict `a[i] > 0` - static prediction ~85% accurate

Code Transformations: Summary

Loop unrolling:

- large block to schedule
- reduces branch frequency
- expands code size
- have to handle "extra" iterations

Software pipelining:

- no dependencies in loop body
- does not reduce branch frequency
- need start-up and finish-up blocks

Trace scheduling:

- works for non-loops
- more complex than unrolling and software pipelining
- use predication to handle hard to predict branches
- does not seem to handle more general cases
- what about exceptions?
Ambiguous Dependences

Software limitation
What compiler can’t analyze => can’t schedule
Ambiguous memory dependences
e.g.,
*p1r =
... = *ptr2 + 4
e.g.,
a[b[i]] =
..... = a[c[i]] - 5
Registers don’t cause this. Why?

Software vs. Hardware

Software:
✓ can look at large amounts of code
✓ no hardware cost
✓ works for regular code
✓ engineering applications, numeric code, weather prediction
✗ low branch prediction accuracy
✗ does not have dynamic information on latencies

Equivalent techniques
indiffer in applicability

Hardware:
✓ high branch prediction accuracy
✓ has dynamic information on latencies (e.g., cache misses)
✓ transparent!
✓ works for generic code (any code)
✓ databases, desktop applications, numeric code, compilers
✗ limited reorder buffer size
✗ high cost/complexity

Ambitious Static Speculation

• Ability to move instructions without affecting program data flow
  ○ possibly use register renaming
• Ability to ignore exceptions in speculated instructions until point exceptions should occur
  ○ mechanisms to recognize exceptions at “original” code position
• Ability to interchange loads and stores with maybe aliases
Preserving Exception Behavior

- Hardware and OS combine to ignore exceptions for speculated instructions
  - OK for correct programs but not for "incorrect" ones
- Use non-excepting speculative instructions and checks
- Poison bits (in registers) when exceptions occurs; exception when register with poison bit is used
- Buffer result of speculative instruction in hardware buffer

Memory Reference Speculation

- Used to schedule loads aggressively ahead of ambiguous stores
- Special hardware to record addresses dynamically: Memory Conflict Buffer (MCB) or ALAT
  - records load addresses of speculated instructions
  - stores check buffer for address match (mis-speculation)
  - check load verifies speculation; jumps to fix-up code if needed

VLIW: Software ILP

Very long instruction word
Provide a long instruction word with one operation per FU
Instruction latencies are fixed
Compiler packs independent instructions into VLIW
  - compiler schedules all hardware resources
Entire long word issues as a "unit"
Result:
  - ILP with simple hardware, simple control, fast clock

VLIW: Scheduling

Code scheduling in software only
Loop unrolling, software pipelining, trace scheduling
Architectural support
  - deferred interrupts
    - enhances scheduling opportunity
  - predicated execution (conditional moves)
    - less need for hardware prediction
  - speculative loads + check instructions
  - more registers
    - renaming not necessary?
Predicated or Conditional Execution

Instructions accompanied by condition registers!
- Condition true => instruction is executed
- Condition false => instruction annulled
- E.g., bnez R1, Skip
  mov R2, R3
Change to:
  cmov R2, R3, R1
Better flow of fetch and decode
May lead to lots of nops!

VLIW: History

Floating point systems, array processors (early DSPs)
- Glen Culler (now retired)
- very successful in 70's
- all latencies fixed; fast, single-level memory
Multiflow
- Josh Fisher (now at HP)
- 1980's minicomputer
Cydrome
- Bob Rau (was at HP, recently deceased)
- 1980's minicomputer
Tera/Cray
- Burton Smith, 1980's-1990's supercomputer, keeps failing?

VLIW Problems

Same as those with static scheduling
Latencies are not fixed!
Caches are here to stay
Need mechanisms to deal with latency dynamically
**Case Study: IA-64 Registers**

- 128 64-bit wide GPRs
  - actually 65 bits wide to support poison bits
- 128 82-bit wide FPRs
- 64 1-bit predicate registers
- 8 64-bit branch registers
  - used for indirect branches

---

**Case Study: IA-64 (Instruction Format)**

- 41-bit instructions
- 3 instructions plus 5-bit template in 128-bit word (bundle)
- 5-bit template gives combination of resources used by 3 instructions in bundle
  - e.g., (M, I, B) or (M, I, I)
  - used to indicate instructions are independent

---

**Case Study: Itanium**

6-way issue (2 VLIW "bundles" of 3 instructions)
Issue in-order, 10-stage pipeline
Predicated execution using condition registers
- Compiler guarantees independence
- Load speculation for memory operations
Hierarchy of branch predictors including two-level
- Compiler branch hints
Register stack mapping
16K L1 I and D, 96K L2, 4M L3 caches
Other ILP Approaches: Vectors

A vector is a one-dimensional array of numbers
Many scientific and commercial programs use vectors
Numeric computations using floating-point
for (i = 1; i<=64; i++)
  a[i] = b[i] + c[i]
Integer computation on image pixels
Vector instructions perform an operation on each element
  addv a, b, c

Why Vectors?

Vector instructions allow deeper pipelines
  ▫ no intra-vector interlocks
  ▫ no intra-vector hazards
  ▫ inner loop control hazards eliminated
  ▫ need not issue multiple instructions
  ▫ vectors can present memory access pattern to hardware
Want deeper pipelines but
  ▫ interlock logic is hard to divide into more stages
  ▫ bubbles due to data hazard increase
  ▫ hard to issue multiple instructions per cycle
  ▫ fetch&issue bottleneck (Flynn bottleneck)

Vector Architectures (Appendix F)

Vector-Register Machines
  ▫ load/store architectures
  ▫ vector operations use vector register except ld/st
  ▫ register ports cheaper than memory ports
  ▫ optimized for small vectors

Memory-Memory vector machines
  ▫ all vectors reside in memory
  ▫ long startup latency
  ▫ memory ports expensive
  ▫ optimized for long vectors
Fact: most vectors are short
  ▫ early machines were memory-memory
  ▫ TI ASC, CDC STAR-100
  ▫ modern vector machines use vector-registers
**DLXV Architecture**

- Strongly based on CRAY-1
- Extend DLX with vector instructions
  - original DLX is a scalar unit
  - 32 integer and 32 FP registers
- Eight vector registers (V0-V7)
  - 64 double-precision FP each (4K bytes total)
- Five vector functional units
  - FP+, FP*, FP:, integer and logical
  - fully-pipelined with 2-20 stages
- Vector load/store units
  - fully-pipelined with 10-50 stages

**DLXV Instructions**

- Vector-vector instructions
  - operate on two vectors
  - produce a third vector
  - addv v1, v2, v3
- Vector-scalar instructions
  - operate on one vector and one scalar
  - addv v1, f0, v3
- Vector ld/st instructions
  - ld/st a vector from memory into a vector register
  - operates on contiguous addresses
  - lv [r1], v1; v[i] = M[r1+i]
  - sv v1, [r1]; M[r1+i] = v[i]

**DLXV Load/Store**

- ld/st vector with stride
  - vectors are not always contiguous in memory
  - add non-unit stride on each access
  - lw [r1, r2], v1; v[i] = M[r1+r2]
  - sw v1, [r1, r2]; M[r1+r2] = v[i]
- ld/st indexed
  - indirect accesses through an index vector
  - lw [r1, v2], v1; v[i] = M[r1+v2[i]]
  - sw v1, [r1, v2]; M[r1+v2[i]] = v[i]

**Vector Code Example**

DAXPY: double-precision a * x + y
for (I=1; k<=64; I++)
  y[i] = a*x[i]+y[i]
VLR 64
  lv [a], f0
  lv [x], v1
  mulv v1, f0, v2
addv v2, v3, v4
sv v4, [r]
Strip Mining

Not all vectors are 64 elements long

Vector length register (VLR)
- controls length of vector operations
- $0 < \text{VLR} < \text{MVL} = 64$

Strip mining
- a loop of vector code
- each iteration implements the code for 64 elements

Other Vector Operations

Use masked vector register for vectorizing if statements
- a mask specifies which vector elements are operated on
- can set the mask using logical compares ($\text{sltsv v1, f0}$)

Vector chaining
- bypassing vector values between functional units
- $\text{multv} --, --, v1$ and $\text{addv} --, v1$, -- executing simultaneously

Scatter/gather
- used for sparse arrays/matrices
- using an index vector, gather elements into a vector register
- operate on the vector
- put the vector back

Vector Memory

What kind of memory hierarchy would you use for vectors?

Compiler techniques

Final word: make the scalar unit fast!

Remember Amdahl's Law

CRAY-1 was the fastest scalar computer

MMX

Intro
Basics
Instruction Set
Examples
Integration into Pentium
Relationship to Vectors
**MMX**


Goal: 2x performance in multimedia (audio, video, etc.)

Key technique: Single Instruction Multiple Data (SIMD)

- used in the past to build large-scale supercomputers
- One instruction computes multiple data simultaneously

**MMX: Intro**

Most multimedia apps work on short integers

Pack data 8-bit or 16-bit data into 64-bit words

Operate on packed data like short vectors

- SIMD (around since Livermore S-1, 20 years ago!)
- Integrate into x86 FP registers
- Can improve performance by 8x (in theory)

- in reality performance is typically better than 2
- gives people another reason to think their PC is obsolete!

**MMX: Datatypes**

Native datatypes smaller than usual

- e.g., 8-bit pixels, 16-bit audio samples

Datatypes packed or compressed into 64 bits

- 1x64-bit quad-words
- 2x32-bit double-words
- 4x16-bit words
- 8x8-bit bytes

**MMX: Enhanced Instructions**

E.g., addb (for byte)

```
17 87 100 .... (5 more)
+17 13 200 ....
```

Saturating arithmetic \((100 + 200) \mod 256 = 44\)

Needed by multimedia algorithms
**MMX: Enhanced Instructions**

Also MOV’s
- move MMX datatypes to and from memory
- loads followed by stores

Pack/Unpack
- go back and forth between MMX and normal datatypes

Needed by multimedia computations

**MMX: Enhanced Instructions**

Mask

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>100</th>
<th>120</th>
<th>101</th>
<th>76</th>
<th>15</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

---

Then selectively operate on 15’s using the mask

Use to select which operands to operate on
- avoid branches
- enables pipelining
- CRAY did this 20 years ago

**MMX: Performance Example**

16 element dot product

\([a_1 \cdot a_2 \cdot a_3 \ldots \cdot a_16][x_1 b_1 b_2 b_3 \ldots \cdot b_{16}] = a_1 x_1 + \ldots + a_{16} x_{16}\]

Intel IA + few optimizations (compiler + hardware)
- 32 loads
- 16 multiplies
- 15 adds
- 12 loop control
- ------------------------
- 72 instructions

MMX using 16-bit data: 16 instructions

**MMX Constraints: Adding to Pentium**

Share registers FP
- ISA extensions but perfect backward compatibility
- 100% OS compatible (no extra regs, flags, exceptions)

Bit in CPUID instruction
- so applications test for MMX and include code

Use 64-bit datapaths

Pipeline capable of 2 MMX IPC

Cascade memory and execution stages to avoid stalls
MMX and Vectors

No Vector Length
Memory load/store: stride of one
Arithmetic integer only
Conditionals: builds byte masks like vector masks
Data movement: pack/unpack like vector scatter/gather