Static Scheduling

- basic pipeline: single, in-order issue
- first extension: multiple issue (superscalar)
- second extension: scheduling instructions for more ILP
  - option #1: dynamic scheduling (by the hardware)
  - option #2: static scheduling (by the compiler)

Readings

H+P
- chapter 4 (strongly recommended)

Recent Research Paper
- EPIC/IA-64

Then, H+P A.8 & 3

VLIW: Very Long Instruction Word

- problems with superscalar implementation
  - wide fetch+branch prediction (can partially fix w/ trace cache)
  - N^2 bypass (can partially fix with clustering)
  - N^2 dependence cross-check (stall+bypass logic)

one alternative: VLIW (very long instruction word)

- single-issue pipe, but unit is N-instruction group (VLIW)
  - instructions in VLIW are guaranteed (by compiler) to be independent
  + processor does not have to dependence-check within a VLIW
  - VLIW travels down pipe as a unit
  - typically “slotted” (i.e., 1st must be ALU, 2nd must be load, etc.)

VLIW History

- started with microcode (“horizontal microcode”)
- academic projects
  - ELI-512 [Fisher, ’85]
  - Illinois IMPACT [Hwu, ’91]
- commercial machines
  - MultiFlow [Colwell+Fisher, ’85] ⇒ failed
  - Cydrome [Rau, ’85] ⇒ failed
  - EPIC (IA-64, Itanium) [Colwell,Fisher+Rau, ’97] ⇒ ??
  - Transmeta [Ditzel, ’99]: translates x86 to VLIW ⇒ ??
  - many embedded controllers (TI, Motorola) are VLIW ⇒ success
Pure VLIW

- pure VLIW: no hardware dependence-checks at all
- not even between VLIW groups
- compiler responsible for scheduling entire pipeline
  - including stall cycles
  - possible if you know structure of pipeline and latencies exactly
- problem 1: pipe & latencies vary across implementations
  - recompile for new implementations (or risk missing a stall)?
  - TransMeta solves this problem by recompiling on-the-fly
- problem 2: latencies are NOT fixed within implementation
  - don’t use caches? (forget it)
  - schedule assuming cache miss? (no point to having caches)
  - not many VLIW purists left

A VLIW Compromise

compromise: EPIC (Explicitly Parallel Instruction Computing)

- less rigid than VLIW (not really VLIW at all)
- variable width instruction words
  - implemented as “bundles” with dependence bits
  - makes code compatible with different width machines
- assumes inter-bundle stall logic provided by hardware
  - makes code compatible with different pipeline depths, op latencies
  - enables stalls on cache misses (actually, out-of-order too)
  - exploits any information on parallelism compiler can give
  - compatible with multiple implementations of same arch
    - e.g., IA64, Itanium

ILP and Scheduling

no point to having an N-wide pipeline if, on average, many fewer than N independent instructions per cycle
- performance is important
- but utilization (actual/peak performance) is also

Code Example: SAXPY

- SAXPY (single-precision A*X+Y)
  - linear algebra routine (used in solving systems of equations)
  - part of famous “Livermore Loops” kernel (early benchmark)

for (I=0;I<N;I++)
  Z[I] = A*X[I] + Y[I]

  ldf f0, X(r1)   // loop:
  mulf f4,f0,f2   // assume A in f2
  ldf f6, Y(r1)   // X,Y,Z are constant addresses
  add f8,f6,f4
  stf f8, Z(r1)
  add r1,r1, #4   // assume I in r1
  ble r1,r2,loop  // assume N*4 in r2
Default SAXPY Performance

- scalar, pipelined processor (for illustration)
  - 5 cycle FP mult, 2 cycle FP add, both fully pipelined
  - full bypassing, branches predicted taken

\[
\begin{array}{cccccccccccccccc}
\text{ldf } f_0, A(r_1) & F & D & X & M & W \\
\text{mulf } f_4, f_0, f_2 & F & D & d^* & E^* & E^* & E^* & E^* & W \\
\text{ldf } f_6, B(r_1) & F & p^* & D & X & M & W \\
\text{addf } f_8, f_6, f_4 & F & D & d^* & d^* & d^* & E^* & w \\
\text{stf } f_8, C(r_1) & F & p^* & p^* & D & X & M & W \\
\text{add } r_1, r_1, #4 & F & D & X & M & W \\
\text{ble } r_1, r_2, \text{loop} & F & D & X & M & W \\
\end{array}
\]

- single iteration (7 instructions) latency: 15 cycles
- performance: 7 instructions / 15 cycles \(\Rightarrow\) IPC = 0.47
- utilization: 0.47 actual IPC / 1 peak IPC \(\Rightarrow\) 47%

Performance and Utilization

- superscalar pipeline
  - same configuration, just two at a time

\[
\begin{array}{cccccccccccccccc}
\text{ldf } f_0, A(r_1) & F & D & X & M & W \\
\text{mulf } f_4, f_0, f_2 & F & D & d^* & E^* & E^* & E^* & E^* & W \\
\text{ldf } f_6, B(r_1) & F & p^* & X & M & W \\
\text{addf } f_8, f_6, f_4 & F & p^* & p^* & D & d^* & d^* & d^* & E^* & + & + & W \\
\text{stf } f_8, C(r_1) & F & p^* & p^* & p^* & X & d^* & M & W \\
\text{add } r_1, r_1, #4 & F & p^* & D & p^* & p^* & D & p^* & X & M & W \\
\text{ble } r_1, r_2, \text{loop} & F & p^* & p^* & p^* & p^* & p^* & D & p^* & d^* & M & W \\
\end{array}
\]

- performance: still 15 cycles - not any better (why?)
- utilization: 0.47 actual IPC / 2 peak IPC \(\Rightarrow\) 24%
- notice: more hazards \(\Rightarrow\) stalls (why?)
- notice: each stall more expensive (why?)

Scheduling and Issue

Instruction Scheduling

- idea: independent instructions between slow ops and uses
  - otherwise pipeline sits idle waiting for RAW to resolve
  - we have already seen dynamic pipeline scheduling
  - to do this we need independent instructions

- scheduling scope: code region we are scheduling
  - the bigger the better (more independent instructions to choose from)
  - once scope is defined, schedule is pretty obvious
  - trick is making a large scope (schedule across branches???)

- compiler scheduling techniques (more about these later)
  - loop unrolling (for loops)
  - software pipelining (also for loops)
  - trace scheduling (for general control-flow)
Scheduling: Compiler or Hardware?

• compiler
  + large scheduling scope (full program), large "lookahead"
  + enables simple hardware with fast clock
    – low branch prediction accuracy (profiling? see next slide!)
    – no information on latencies like cache misses (profiling?)
    – pain to speculate and recover from mis-speculation (h/w support?)

• hardware
  + better branch prediction accuracy
  + dynamic information on latencies (cache misses) and dependences
    + easy to speculate & recover from mis-speculation
    – finite on-chip instruction buffering limits scheduling scope
    – more complicated hardware (more power? tougher to verify?)
    – slower clock

Aside: Profiling

profile: (statistical) information about program tendencies

• run program once with a test input and see how it behaves
• hope that other inputs lead to similar behaviors
• compiler can use this info for scheduling
• profiling can be a useful technique
  • must be used carefully - else, can harm performance
• popular research topic
  • gaining importance

Loop Unrolling SAXPY

we want to separate dependent operations from one another
  • but not enough flexibility within single iteration of loop
  • longest chain of operations is 9 cycles
    • load result (1 cycle)
    • forward to multiply (5 cycles)
    • forward to add (2 cycles)
    • forward to store (1 cycle)
    • can’t hide 9 cycles of latency using 7 instructions
    • how about 9 cycles of latency twice in 14 instructions?
  • loop unrolling: schedule 2 loop iterations together

Unrolling Part 1: Fuse Iterations

• combine two (in general, N) iterations of loop
  • fuse loop control (induction increment + backward branch)
  • adjust implicit uses of internal induction variables (r1 in example)
  ldf f0,X(r1)
  mulf f4,f0,f2
  ldf f6,Y(r1)
  addf f8,f6,f4
  stf f8,Z(r1)
  add r1,r1,#4
  ble r1,r2,loop

  ldf f0,X(r1)
  mulf f4,f0,f2
  ldf f6,Y(r1)
  addf f8,f6,f4
  stf f8,Z(r1)
  add r1,r1,#8
  ble r1,r2,loop

  ldf f0,X+4(r1)
  mulf f4,f0,f2
  ldf f6,Y+4(r1)
  addf f8,f6,f4
  stf f8,Z+4(r1)
  add r1,r1,#4
  ble r1,r2,loop
Unrolling Part 2: Pipeline Schedule

- pipeline schedule to reduce RAW stalls

```
ldf f0, X(r1)
mulf f4, f0, f2
ldf f6, Y(r1)
addf f8, f6, f4
stf f8, Z(r1)
ldf f0, X+4(r1)
mulf f4, f0, f2
ldf f6, Y+4(r1)
addf f8, f6, f4
stf f8, Z+4(r1)
add r1, r1, #8
ble r1, r2, loop
```

Unrolling Part 3: Rename Registers

- pipeline scheduling caused WAR hazards
  - so we rename registers to solve this problem

```
ldf f0, X(r1)
mulf f4, f0, f2
ldf f6, Y(r1)
mulf f4, f0, f2
ldf f6, Y(r1)
addf f8, f6, f4
stf f8, Z(r1)
add r1, r1, #8
ble r1, r2, loop
```

Unrolled SAXPY Performance

- 2 iterations (12 instructions) → 17 cycles (fewer stalls)
  - before unrolling, it took 15 cycles for 1 iteration!

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Shortcomings of Loop Unrolling

- code growth
- poor scheduling along “seams” of unrolled copies
- doesn’t handle inter-iteration dependences (recurrences)

```
for(I=0; I<N; I++)
    X[I] = A*X[I-1]; // each iteration depends on prior
```

```
ldf f2, X-4(r1)
mulf f4, f2, f0
stf f4, X(r1)
add r1, r1, #4
ble r1, r2, loop
```

```
ldf f12, X-4(r1)
mulf f14, f12, f0
stf f14, X(r1)
add r1, r1, #8
ble r1, r2, loop
```

1 dependence chain → can’t schedule
Software Pipelining

Software pipelining: deals with these problems
• also called symbolic loop unrolling
• reinvented a few times under different guises
  • microcode [Charlesworth '81]
  • polycyclic scheduling [Rau '85]
  • general loop unrolling [Lam '88]
• basic idea:
  • start with original/unmodified logical loop
  • convert to new loop w/insts from different iterations of original loop
  • requires some prologue and epilogue code to cleanup edges

The Pipeline Analogy

• hardware pipelining
  • any cycle contains:
    • stage 3 of inst i, stage 2 of inst i+1, stage 1 of inst i+2
• software pipelining
  • cycle \( \Rightarrow \) software pipelined (physical) loop iteration
  • instruction \( \Rightarrow \) logical (original/unmodified) loop iteration
  • stage \( \Rightarrow \) instruction
• a single physical iteration contains instructions from multiple original iterations:
  • inst 3 of iteration i, inst 2 of iteration i+1, inst 1 of iteration i+2

Software Pipelining Example

• physical iteration (box) contains:
  • \( \text{stf} \) from original iteration i
  • \( \text{ldf, mulf} \) from original iteration i+1

• prologue: get pipeline started (\( \text{ldf, mulf} \) from iteration 0)
• epilogue: finish up leftovers (\( \text{stf} \) from last iteration)

Software Pipelining Pipeline Diagrams

• same diagram, new terminology
  • cycles \( \Rightarrow \) physical iterations (across)
  • instructions \( \Rightarrow \) logical iterations (down)
  • stages \( \Rightarrow \) instructions (\( \text{LM} = \text{ldf}, \text{mulf}, \text{S} = \text{stf} \))

\( \text{p1} \)
\( \text{p2} \)
\( \text{p3} \)
\( \text{p4} \)
\( \text{p5} \)

\( \text{e.g., physical iteration 2 has stf from logical iteration 1 and has ldf/mulf from logical iteration 2} \)

• NOTICE, within physical iteration, instruction groups are in reverse order
• that's OK, if groups are unrelated/parallel \( \Rightarrow \) but are they here??
• perfect for VLIW if groups are parallel
Software Pipelining Example II

- vary software pipelining structure to tolerate more latency
  - e.g., ldf, mulf, stf from 3 different iterations (not just 2)

```
ldf f2,X(r1)
mulf f4,f2,f0
stf f4,X(r1)
add r1,r1,#4
ble r1,r2,loop
ldf f2,X(r1)
mulf f4,f2,f0
stf f4,X(r1)
add r1,r1,#4
ble r1,r2,loop
ldf f2,X(r1)
mulf f4,f2,f0
stf f4,X(r1)
add r1,r1,#4
ble r1,r2,loop
```

Software Pipelining

+ doesn't increase code size (much)
  - just some extra prologue/epilogue code
+ can vary degree of pipelining to tolerate longer latencies
  - “software superpipelining”
  - one physical iteration: instructions from logical iterations i,i+2,i+4
  - hard to do conditionals within loops
  - tricky register allocation sometimes

Trace Scheduling

problem: not everything is a loop
idea: for general non-loop situations
  - find common paths in program
  - realign basic blocks to form straight-line trace
    - basic block: single-entry, single-exit instruction sequence
    - trace (aka superblock, hyperblock): fused basic block sequence
  - schedule instructions within trace
  - create fixup code outside trace in case trace != actual path
    - this can be pretty nasty
  - trace scheduling
    - [Ellis,'85]

Trace Scheduling Example

```
A = Y[i];   # A in f2
if (A == 0)
  A = W[i];
else
  Y[i] = 0;
Z[i] = A*X[i];
```

```
0: ldf f2,Y(r1)
1: bfnez f2,instr#4
  A = W[i];
2: ldf f2,W(r1)
3: jump instr#5
4: stf f0,Y(r1)
5: ldf f4,X(r1)
6: mulf f6,f4,f2
7: stf f6,Z(r1)
```

- scheduling problem: separate #6 (3 cycles) from #7
  - but how to move mulf (and #5 ldf) above if-then-else?
Basic Blocks and Superblocks

- Choose most common path: A, C, D
- Assumes you know branch #1’s frequency (e.g., via profiling)
- Fuse into one large “superblock” & schedule
- Create repair code just in case real path was A, B, D ...

Superblock Scheduling 1

First scheduling move: move #5, #6 above #4
- Moved load (#5) above store (#4)
- We can tell this is OK, but can the compiler?
  - If yes, fine
  - Otherwise, compiler needs to do something

Repair Blocks

- Change sense of branch condition (bfnez to bfeqz)
- Repair block: may need to duplicate code (block D here)
- Haven’t scheduled superblock yet ...

ISA Support for Load-Store Speculation

- Change #5 to advanced load, ld.f.a
- “Advanced” means advanced past unknown store
- Processor tracks load address, matches with other stores
- Insert chk.a to check store collision. If collision? Repair
- Called “memory conflict buffer (MCB)”, adopted by IA64
Superblock Scheduling 2

Superblock

```
#0: ld f2, Y(r1)
#5: ld f4, X(r1)
#6: mul f6, f4, f2
#1: bfeqz f2, #2
#4: st f0, Y(r1)
#7: st f6, Z(r1)
#8: chk a f4, #9
```

second scheduling move: move #5, (load) #6 above #1 (branch)
- that's OK, since load did not depend on branch
- was going to execute anyway --> not speculative

scheduling non-move: don't move #4 (store) above #1 (branch)
- why? hard (but possible) to undo a store in repair block

```
#0: ld f2, Y(r1)
#5: ld f4, X(r1)
#6: mul f6, f4, f2
#1: bfeqz f2, #2
#4: st f0, Y(r1)
#7: st f6, Z(r1)
```

repair code

```
#2: ld f2, W(r1)
#6': mul f6, f4, f2
#7': st f6, Z(r1)
```

Superblock Scheduling 3

Superblock

```
#0: ld f2, Y(r1)
#2: ld f8, W(r1)
#5: ld f4, X(r1)
#6: mul f6, f4, f8
#1: bfeqz f2, #4
#7: st f6, Z(r1)
```

move #2 (load), #5, and #6 above #1 (branch)
- rename f2 to f8 to avoid name conflicts
- is this an OK thing to do?
  - from a store standpoint, yes
  - what about from a fault standpoint? what if #2 faults?

```
#0: ld f2, Y(r1)
#2: ld f8, W(r1)
#5: ld f4, X(r1)
#6: mul f6, f4, f8
#1: bfeqz f2, #4
#7: st f6, Z(r1)
```

repair code

```
#4: st f0, Y(r1)
#6': mul f6, f4, f2
#7': st f6, Z(r1)
```

Superblock Scheduling

Superblock

```
#0: ld f2, Y(r1)
#1: bfeq f2, #4
```

A

```
#0: ld f2, Y(r1)
```

B

```
#2: ld f2, W(r1)
```

C

```
#4: st f0, Y(r1)
```

D

```
#5: ld f4, X(r1)
```

NT = 90%

T = 10%

what if #1 (branch) was biased the other way?

ISA Support for Load-Branch Speculation

Superblock

```
#0: ld f2, Y(r1)
#2: ld f8, W(r1)
#5: ld f4, X(r1)
#6: mul f6, f4, f8
#1: bfeqz f2, #4
#7: st f6, Z(r1)
```

repair code

```
#4: st f0, Y(r1)
#6': mul f6, f4, f2
#7': st f6, Z(r1)
```

```
#0: ld f2, Y(r1)
#2: ld f8, W(r1)
#5: ld f4, X(r1)
#6: mul f6, f4, f8
#1: bfeqz f2, #4
#7: st f6, Z(r1)
```

repair code

```
#4: st f0, Y(r1)
#6': mul f6, f4, f2
#7': st f6, Z(r1)
```

• change #2 to speculative load, ld f.s
  - "speculative" means speculative above unknown branch
- similarly, change #6 to speculative multiply, mul f.s
- processor keeps interrupt bits with registers f8, f6
- interrupt handled when f6 is used by non-speculative #7
- called "poison bit" or "deferred interrupt", adopted by IA64
Hyperblock Scheduling

what if branch #1 is not biased?

- create a large block from both paths (all 4 basic blocks)
- called a hyperblock
- use predication to conditionally execute instructions

ISA Support for Predication

hyperblock

#0: ldf f2,Y(r1)
#1: sltip p1,f2,#0
#2: ldf.p f2,W(r1), p1
#4: stf.np f0,Y(r1), p1
#5: ldf f4,X(r1)
#6: mulf f6,f4,f2
#7: stf f6,Z(r1)

- change branch #1 to set-predicate instruction, sltip
- change instructions #2 and #4 to predicated instructions
  - ldf.p perform load instruction if predicate is true
  - stf.np perform store instruction if predicate is not-true

Predication

two levels of predication

- full predication: can tag every instruction with predicate
  - adopted by IA64
- conditional register moves: (CMOVE)
  - construct appearance of full predication from one basic primitive

```
cmoveqz r1,r2,r3       // if (r3 == 0) r1 = r2;
```

- may require a lot of code duplication
  - adopted by Alpha, IA32
- “if-conversion”: converts control-flow to data-flow
  + eliminates branches
  - why can it be bad?

Static Scheduling Summary

- loop unrolling
  + reduces branch frequency
  - expands code size, have to handle “extra” iterations
- software pipelining
  + no dependences in loop body
  - does not reduce branch frequency, need prologue/epilogue blocks
- trace scheduling
  + works for non-loops
  - more complex than unrolling and software pipelining
- ISA support
  - speculative loads, advanced loads
  - predication
Where We Stand Now

We have covered the following topics:

• performance and benchmarking
• instruction sets
• pipelining
• static scheduling

next up: dynamic scheduling