Computer Systems Architecture CMSC 411 Unit 4b – Software instruction-level parallelism

Alan Sussman

April 25, 2006

Administrivia

- Project due Thursday – questions?
- Quiz 2 postponed to May 9
- Homework 5 due next Tuesday, May 2
- Homework 6 posted
- Read Chapter 4, except 4.8 and global code scheduling in 4.4
- Online course evaluation available at https://www.courses.umd.edu/online_evaluation

Loop unrolling

- To improve performance of pipelines and simple multiple issue processors
  - for static issue, and for dynamic issue with static scheduling
- To fill pipeline stalls
  - can be done dynamically in hardware, or statically in compiler
- Look again at an example we used before

Example - loop unrolling

original loop:

```
for i=1000, 999, ..., 1
   x[i] = x[i] + s;
```

unrolled to a depth of 4:

```
for i=1000, 996, 992,..., 4
   x[i]   = x[i]   + s;
   x[i-1] = x[i-1] + s;
   x[i-2] = x[i-2] + s;
   x[i-3] = x[i-3] + s;
end for
```

```
Loop:
   L.D     F0,0(R1)       x[i] = x[i] + s
   ADD.D   F4,F0,F2       uses F0 and F4
   S.D     F4,0(R1)
   L.D     F6,-8(R1)      x[i-1] = x[i-1] + s
   ADD.D   F8,F6,F2       uses F6 and F8
   S.D     F8,-8(R1)
   L.D     F10,-16(R1)    x[i-2] = x[i-2] + s
   ADD.D   F12,F10,F2     uses F10 and F12
   S.D     F12,-16(R1)
   L.D     F14,-24(R1)    x[i-3] = x[i-3] + s
   ADD.D   F16,F14,F2     uses F10 and F12
   S.D     F16,-24(R1)
   DSUBI   R1,R1,#32      point to next element
   BNE     R1,R2,Loop
```


And reschedule the loop

```
Loop:
   L.D     F0,0(R1)
   L.D     F6,-8(R1)
   L.D     F10,-16(R1)
   L.D     F14,-24(R1)
   ADD.D   F4, F0,F2
   ADD.D   F8,F6,F2
   ADD.D   F12,F10,F2
   ADD.D   F16,F14,F2
   S.D     F4,0(R1)
   S.D     F8,-8(R1)
   DSUBI   R1,R1,#32
   S.D     F12,16(R1)
   BNE     R1,R2,Loop
   S.D     F16, 8(R1)
```


Example (cont.)

- Note: if 1000 were not divisible by 4, we would have a loop like this plus added code to take care of the last few elements
- How well does the unrolled code pipeline?
  - uses (14 cycles)/(4 elements), instead of original code that used 6 cycles per element
    - assuming issue 1 instruction per cycle, and standard MIPS pipeline organization (load delays, functional unit latencies)

Loop unrolling (cont.)

- Limited only by:
  - number of available registers
  - size of instruction cache - want the unrolled loop to fit
- What is gained
  - fewer pipeline stalls/bubbles
  - less loop overhead - fewer **DSUBI**s and **BNE**s
- What is lost
  - longer code
  - many possibilities to introduce errors
  - slower compilation
  - more work for either the programmer or for the compiler writer

What did the compiler have to do?

- Determine that it was legal to move S.D after DSUBI and BNE, and adjust S.D offsets
- Determine that loop unrolling would be useful – improve performance
- Use different registers to avoid name dependences
- Eliminate extra test and branch instructions, and adjust loop termination and counter code
- Determine that loads and stores could be interchanged – the ones from different iterations are independent – requires memory address analysis
- Schedule the code, preserving true dependences

Dependences limit loop unrolling

```
Loop:
  L.D F0,0(R1)
  ADD.D F4,F0,F2
  S.D F4,0(R1)
  DSUBI  R1,R1,#8
  BNEZ R1,R2,Loop
```

L.D and BNEZ depend on result of DSUBI

- In unrolling, removed intermediate DSUBI instructions to reduce the *data dependence* for the L.D and the *control dependence* for the BNEZ
- There are also *antidependences,* so also made sure that later copies of the unrolled code used registers other than F0 & F4 - eliminated *name dependences*

True data dependences also limit unrolling

```
for i=1,...,1000
   x[i+1] = x[i] + c[i]       ;Uses value from previous iteration
   b[i+1] = d[i] + x[i+1]   ;Uses the value just computed
end for
```

- First assignment statement is an example of a *loop-carried dependence*
- Second assignment statement doesn't limit unrolling, but makes scheduling trickier

One problem for the programmer

- *Precise exception handling* becomes impossible if the compiler unrolls loops
- The order of operations that the user assumes is completely violated, so exceptions occur at unrecognizable locations
- Rather than precise exception handling, settle for the property that the unrolled code produces no new exceptions over the original code
- Also possible to provide software to simulate the code's behavior around the exception to help user diagnose the problem

Compilers need to detect data dependences

```
for i=1,2,...,100
  a(i) = b(i) + c(i);
  d(i) = a(i) * e(i);
end for
```

- Can unroll this loop, but must be careful not to interchange the order of the two statements
- Note that *a(i)* never needs to be loaded

Second example

```
for i=k,k+1,...,100
   a(i) = a(i-k) + a(i);
end for
```

- Quiz:
  - Unroll the loop to a depth of 4 assuming that *k*=1
  - Unroll the loop to a depth of 4 assuming that *k*=5
- Note how much more parallelism there is in the 2nd case, because the dependence is less of a problem

Third example

```
for i=1,2,...,100
   x(2*i+3) = x(2*i) * 5.0;
end for
```

- Is there any dependence in this loop?
  - Does it ever store into a location and then fetch the same value later?
- No! Always stores with odd index, and fetches with even

Fourth example

```
for i=1,2,...,100
   x(3*i) = x(2*i+3) * 5.0;
```

- Is there any dependence in this loop?
  - Does it ever store into a location and then fetch the same value later?
- Yes! Produce an example
  - i=5, store 15
  - i=6, fetch 15
- How to detect this in general?

Mathematics to the rescue

- If store to location $a*i+b$ and fetch from location $c*i+d$, then they can be equal if there are values i and I so that

$$a*i + b = c*I + d$$

- or, equivalently, if

$$a*i - c*I = d - b$$

- Suppose q is a divisor of a and c. Then q would also be a divisor of d-b.
- Therefore, if no divisor of a and c is also a divisor of d-b, then there can be no dependence.
- And only need to check the greatest common divisor (**gcd**) of a and c.

Back to the example

```
for i=1,2,...,100
   x(3*i) = x(2*i+3) * 5.0;
end for
```

- $a=3$, $b=0$, $c=2$, and $d=3$. The greatest common divisor of $a=3$ and $c=2$ is 1, and 1 divides d-$b=3$, so it is possible that loop dependences occur.

Fifth example

```
for i=1,2,...,100
   y(i) = x(i) / c;    S1
   x(i) = x(i) + c;    S2
   z(i) = y(i) + c;    S3
   y(i) = c - y(i);    S4
end for
```

- There are 5 dependences:
  - S3 depends on the result of S1
  - S4 depends on the result of S1
  - There is an antidependence between S1 and S2
  - There is an antidependence between S3 and S4
  - There is an output dependence between S1 and S4

Things that may fool a compiler regarding dependences

- Fortran *equivalence* statements

```
Real A(20,20)
Real B(400)
Equivalence (A,B)
```

- Then *B(21)* has the same address as *A(1,2)*
- pointer references
  - for C/C++ and Java
  - what does *p point at?
- array indexing through another array
  - Example: *x[index[i]]*
- dependence exists only for some input values, but inputs may never take on those values

Hardware support for the compiler

- Conditional/predicated instructions
  - to eliminate branches
- Methods to help compiler move code past branches
  - mainly to deal with exceptions properly
- Checks for address conflicts
  - to help with reordering loads and stores

Conditional instructions

- Condition is evaluated as part of the instruction execution
  - if condition true, normal execution
  - if condition false, instruction turned into a no-op
- Example: conditional move
  - move a value from one register to another if condition is true
  - can eliminate a branch in simple code sequences

Example: conditional move

- For code: **if (A==0) { S=T; }**
  - Assume R1, R2, R3 hold values of A, S, T

With branch:

```
    BNEZ R1, L
    ADDU R2, R3, R0
L:
```

With conditional move (if 3rd operand equals zero):

```
    CMOVZ R2, R3, R1
```

- Converts the control dependence into a data dependence
  - for a pipeline, moves the dependence from near beginning of pipeline (branch resolution) to end (register write)

Superscalar execution

- Predication helps with scheduling
- Example: superscalar that can issue 1 memory reference and 1 ALU op per cycle, or just 1 branch

1st instruction	2nd instruction
LW R1,40(R2)	ADD R3,R4,R5
	ADD R6,R3,R7
BEQZ R10,L	
LW **R8**,0(R10)	
LW R9,0(**R8**)	

LWC loads if 3rd operand not 0

1st instruction	2nd instruction
LW R1,40(R2)	ADD R3,R4,R5
LWC **R8**,0(R10),R10	ADD R6,R3,R7
BEQZ R10,L	
LW R9,0(**R8**)	

Limitations of cond. instructions

- Predicated instructions that are squashed still use processor resources
  - doesn't matter if resources would have been idle anyway
- Most useful when predicate can be evaluated early
  - want to avoid data hazards replacing control hazards
- Hard to do for complex control flow
  - for example, moving across multiple branches
- Conditional instructions may have higher cycle count or slower clock rate than unconditional ones

Compiler speculation with hardware support

- To move speculated instructions not just before branch, but before condition evaluation
- Compiler can help find instructions that can be speculatively moved and not affect program data flow
- Hard part is preserving exception behavior
  - a speculated instruction that is mispredicted should not cause an exception
  - 4 methods described in Section 4.5, so it can be done

Memory reference speculation with hardware support

- To move loads across stores, when compiler can't be sure it is legal
- Use a *speculative load* instruction
  - hardware saves address of memory location
  - if a subsequent store changes that location before the check (to end the speculation), then the speculation failed, otherwise it succeeded
  - on failure, need to redo load and re-execute all speculated instructions after the speculative load

Intel IA-64 Architecture and an Implementation

Instruction set architecture

- RISC-style, register-register, plus features to support compiler-based ILP
  - most instructions predicated using predicate registers
    - predicate registers set using compare or test instructions
  - integer registers use a register stack (like SPARC)
  - speculation for both control (deferring exceptions) and for memory references (load speculation)
  - overall, essentially a more flexible VLIW
    - compiler detects ILP and schedules operations, but not a single fixed format for VLIW instructions
  - it's a mess, and violates several of the guidelines we've talked about
    - especially in making tradeoffs obvious, and regularity (look at the limitations on instruction issue in H&P)

Itanium IA-64 implementation

- Up to 6 issues per clock, including 3 branches and 2 memory references
- 3 level cache – 2 on chip, one off
- 9 functional units, all pipelined
- 10 stage pipeline, in 4 major parts
  - *front-end (3)* – IF, branch prediction
  - *instruction delivery (2)* – send instructions to FUs, rename registers
  - *operand delivery (2)* – read registers, check predicates
  - *execution (3)* – also retires instructions, does writeback

Itanium instruction latencies

Instruction	**Latency**
Integer load[1]	1
Floating-point load[2]	9
Correctly predicted taken branch	0-3
Mispredicted branch	9
Integer ALU ops	0
FP arithmetic	4

Figure 4.15

1. Primary cache hit
2. Secondary cache hit

Fallacies

- A simple approach for multiple issue can be found that gets both high performance and doesn't use too many transistors or have high design complexity
  - no *silver bullets*
  - for simple approaches, as issue rate increases the gap between peak and sustained performance grows quickly
  - so more sophisticated techniques really are necessary – e.g., dynamic scheduling, hardware/software speculation, branch prediction, …
  - compilers getting very complex too – need sophisticated transformations, and lots of tuning