

# HY425 Lecture 09: Software to exploit ILP

Dimitrios S. Nikolopoulos

University of Crete and FORTH-ICS

November 4, 2010

# ILP techniques

## Hardware

- ▶ Dynamic scheduling with scoreboard
- ▶ Renaming (Tomasulo, renaming registers)
- ▶ Branch prediction
- ▶ Multiple issue
- ▶ Speculation

## Software

- ▶ Instruction scheduling
- ▶ Code transformations (topic of next lecture)

# What limits ILP

## Software and hardware issues

- ▶ Limits of parallelism in programs
  - ▶ Data flow – true data dependencies
  - ▶ Control flow – control dependencies
  - ▶ Code generation, scheduling by compiler
- ▶ Hardware complexity
  - ▶ Large storage structures – branch prediction, ROB, window
  - ▶ Complex logic – dependence control, associative searches
  - ▶ Higher bandwidth – multiple issue, multiple outstanding instructions
  - ▶ Long latencies – memory system (caches, DRAM)

## Recap: add scalar to vector

### Unrolling and renaming (6 cycles per iteration)

```

Loop:   LD      F0, 0(R1)
        ADDD   F4, F0, F2
        SD     F4, 0(R1)
        LD     F6, -8(R1)
        ADDD   F8, F6, F2
        SD     F8, -8(R1)
        LD     F10, -16(R1)
        ADDD   F12, F10, F2
        SD     F12, -16(R1)
        LD     F14, -24(R1)
        ADDD   F16, F14, F2
        SD     F16, -24(R1)
        ADDI   R1, R1, -32
        BNE   R1, R2, Loop
  
```

- ▶ Pros Unrolling lowers loop overhead (ADDI, BNE)
- ▶ Cons: Unrolling grows code size
- ▶ Cons: Register pressure

## Recap: add scalar to vector

### Unrolling and renaming with improved instruction scheduling (3.5 cycles per iteration)

```
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      F4, 0(R1)
        SD      F8, -8(R1)
        SD      F12, -16(R1)
        ADDI   R1, R1, -32
        BNE   R1, R2, Loop
        SD      F16, 8(R1)
```

## Recap: add scalar to vector

### Unrolling and renaming with dual-issue

Integer instruction	FP instruction	Clock cycle
Loop: LD F0,0(R1)		1
LD F6,-8(R1)		2
LD F10,-16(R1)	ADDD F4,F0,F2	3
LD F14,-24(R1)	ADDD F8,F6,F2	4
LD F18,-32(R1)	ADDD F12,F10,F2	5
SD F4,0(R1)	ADDD F16,F14,F2	6
SD F8,-8(R1)	ADDD F20,F18,F2	7
SD F12,-16(R1)		8
SD F16,-24(R1)		9
ADDI R1,R1,-40		10
BNE R1,R2,Loop		11
SD F20,-32(R1)		12

2.4 cycles per iteration

## Predict branches as always taken

### Example

DSUBU and BEQZ need to stall

```
L:      LD      R1, 0(R2)
        DSUBU  R1, R1, R3
        BEQZ  R1, L
        OR    R4, R5, R6
        DADDU R10, R4, R3
L:      DADDU  R7, R8, R9
```

### Example

```
LD      R1, 0(R2)
DADDU   R7, R8, R9      #speculative
DSUBU   R1, R1, R3
BEQZ    R1, L
OR      R4, R5, R6
DADDU   R10, R4, R3
L:
```

- ▶ Second control-dependent DADDU speculatively moved before branch to eliminate stall
- ▶ Note that moved DADDU is **not data-dependent** on OR, or first DADDU

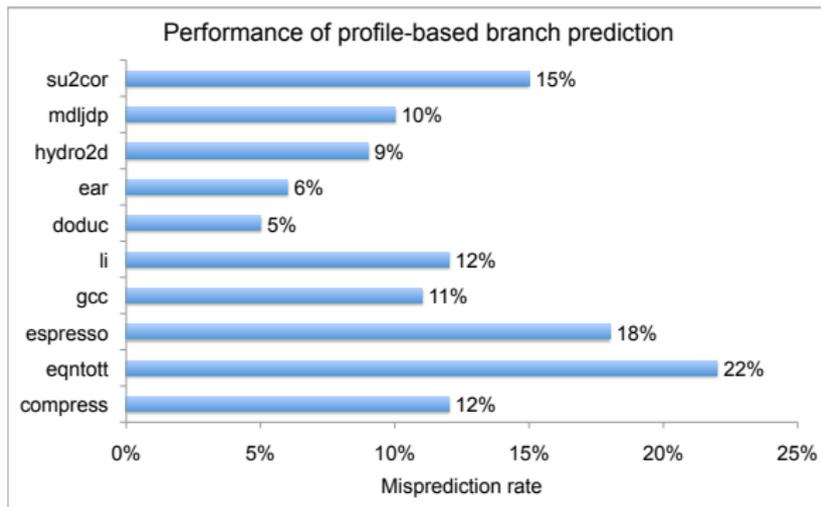
## Static branch prediction alternatives

### Simple offline branch prediction schemes

- ▶ Predict always taken
  - ▶ 34% misprediction rate, high variance
- ▶ Predict based on direction of branch
  - ▶ Forward not taken, backward taken
  - ▶ Misprediction rates 30%–40%
- ▶ Predict based on execution profile
  - ▶ Branch bias (mostly taken or not taken)
  - ▶ Accuracy sensitive to input

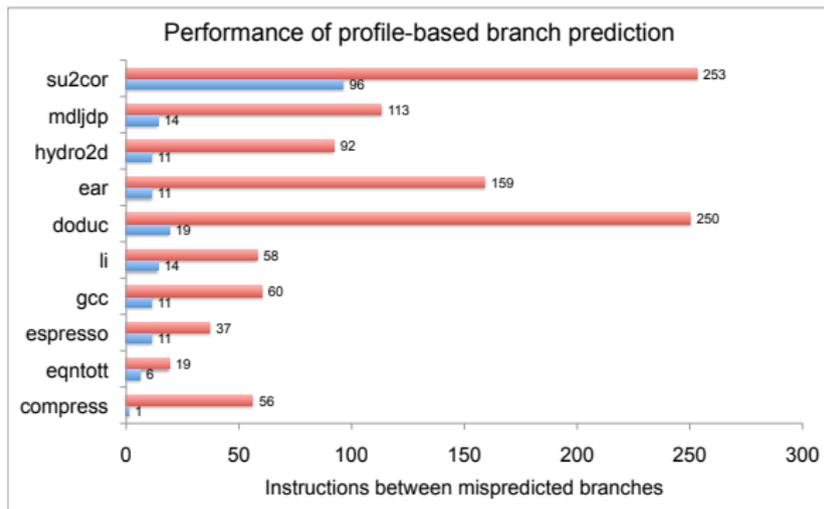
## Performance of profile-based branch prediction

### SPECCPU92 results



## Profile-based vs. static prediction

### SPECCPU92 results



## VLIW processors

### Statically scheduled multiple-issue processors

- ▶ Reduce hardware cost compared to dynamically scheduled
- ▶ Advanced compiler support for exploiting ILP
  - ▶ Instructions scheduled in packets
  - ▶ No dependences among instructions in packet
- ▶ Long instruction word (64+ bits)
- ▶ Explicit parallelism among instructions
  - ▶ Compiler guarantees that instructions are independent
- ▶ Multiple functional units
- ▶ Parallelism exploited via loop unrolling and instruction scheduling

## Add scalar to vector example

### Unrolling and code scheduling in VLIW

- ▶ 2 load/store, 2 INT, 1 FP unit
- ▶ 1.29 cycles per iteration (vs. 2.4 in two-issue superscalar)

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer branch
LD F0, 0(R1)	LD F6, -8(R1)			
LD F10, -16(R1)	LD F14, -24(R1)			
LD F18, -32(R1)	LD F22, -40(R1)	ADDD F4, F0, F2	ADDD F8, F6, F2	
LD F26, -48(R1)		ADDD F12, F10, F2	ADDD F16, F14, F2	
		ADDD F20, F18, F2	ADDD F24, F22, F2	
SD F4, 0(R1)	SD F8, -8(R1)	ADDD F28, F26, F2		
SD F12, -16(R1)	SD F16, -24(R1)			
SD F20, -24(R1)	SD F24, -32(R1)			ADDI R1, R1, -56
SD F28, 8(R1)				BNE R1, R2, Loop

## VLIW processors

### Statically scheduled multiple-issue processors

- ▶ Parallelism sought within and across basic blocks
  - ▶ Static branch prediction
  - ▶ Hardware support for predicated instruction execution

### Design implications

- ▶ Increased code size
- ▶ Lock-step execution of instruction bundles
  - ▶ Stall in a FU causes entire processor pipeline stall
  - ▶ Hard to schedule instructions upon cache misses
  - ▶ **Solution: check hazards and dependences at issue time, use hardware to enable unsynchronized execution**

## VLIW processors (cont.)

### Code compatibility

- ▶ Old binary can not run if number of FUs changes
  - ▶ Format of instruction bundles is changed
  - ▶ Binary translation **across VLIW HW generations**
  - ▶ Binary translation **from superscalar to VLIW**
- ▶ Superscalar runs unmodified binaries from previous HW generations
  - ▶ only code scheduling may require changes for performance

# Compiler dependence analysis of source code

## Loop-level parallelization

- ▶ Dependence analysis for detecting **loop-carried dependences**
  - ▶ Dependences between instructions in two lexicographically ordered iterations of a loop
  - ▶ Lexicographical ordering produces the equivalent of a sequential in-order execution of all instructions in a loop
- ▶ Independent iterations can be unrolled at will
- ▶ Independent iterations can execute in parallel
  - ▶ Key to exploit multiple processors

## Example

```
for (i=1; i<=100; i=i+1) {  
  A[i+1] = A[i] + C[i]; /* S1 */  
  B[i+1] = B[i] + A[i+1]; /* S2*/  
}
```

- ▶ Loop-carried dependence on S1 (A[i+1] depends on A[i], C[i])
- ▶ Loop-carried dependence on S2 (B[i+1] depends on B[i])
- ▶ Same-iteration dependence on S2 (B[i+1] depends on A[i+1])
- ▶ Loop-carried dependences may or may not prevent parallelization

## Example

### Can this loop be parallelized?

```
for (i=1; i<=100; i=i+1) {  
  A[i] = B[i] + C[i]; /* S1 */  
  B[i+1] = C[i] + D[i]; /* S2*/  
}
```

- ▶ Peel one iteration from each end of the loop
- ▶ Notice that no iteration produces result for future iteration

```
A[1] = B[1] + C[1];  
for (i=1; i<=99; i=i+1) {  
  A[i+1] = B[i+1] + C[i+1]; /* S1 */  
  B[i+1] = C[i] + D[i]; /* S2*/  
}  
B[101] = C[100] + D[100];
```

- ▶ Loop-carried dependence eliminated

## Dependence analysis

### Limitations of analyzing memory references

- ▶ Static analysis indicates that there **may** be a dependence between two instructions due to naming of memory locations
- ▶ Dependence resolution requires disambiguation of memory references
  - ▶ Easy for scalar variables, **harder for arrays, hard for pointers**
- ▶ Dependences do not always prevent parallelization

### Uncovering parallelism in loops with dependences

```
for (i=6; i<=100; i=i+1) {  
    A[i] = A[i-5] + A[i];    /* S1 */  
}
```

- ▶ No loop-carried dependences in 5 iterations

## Dependence analysis 101

- ▶ Assume affine array indices:  $index = a \times i + b$
- ▶ Index in multi-dimensional array affine, if index in each dimension affine
- ▶ Assume two references  $a \times j + b$ ,  $c \times k + d$ , check if:
  - ▶ Array elements are within loop bounds:  
 $m \leq j \leq n, m \leq k \leq n$
  - ▶  $j$  precedes  $k$  (lexicographical ordering)
  - ▶  $a \times j + b = c \times k + d$
  - ▶ GCD test: test if  $GCD(a, c)$  divides  $(d - b)$
  - ▶ **Necessary but not sufficient condition**
- ▶  $a, b, c, d$  and bounds need to be known at compile-time

## Eliminating dependences through renaming

### Finding dependences in source code

```
for (i=1; i<=100; i=i+1) {  
    Y[i] = X[i] / c;    /* S1 */  
    X[i] = X[i] + c;    /* S2 */  
    Z[i] = Y[i] + c;    /* S3 */  
    Y[i] = c - Y[i];    /* S4 */  
}
```

- ▶ True dependences  $S1 \rightarrow S3$ ,  $S1 \rightarrow S4$
- ▶ Antidependence  $S1 \rightarrow S2$ ,  $S3 \rightarrow S4$
- ▶ Output dependence  $S1 \rightarrow S4$

## Eliminating dependences through renaming

### Finding dependences in source code

```
for (i=1; i<=100; i=i+1) {  
  Y[i] = X[i] / c;    /* S1 */  
  X[i] = X[i] + c;    /* S2 */  
  Z[i] = Y[i] + c;    /* S3 */  
  Y[i] = c - Y[i];    /* S4 */  
}
```

### Renaming resolves output and anti dependences

```
for (i=1; i<=100; i=i+1) {  
  T1[i] = X[i] / c;    /* S1 */  
  T2[i] = X[i] + c;    /* S2 */  
  Z[i] = T1[i] + c;    /* S3 */  
  Y[i] = c - T1[i];    /* S4 */  
}
```

## Limits of dependence analysis

### Examples of hard to analyze cases

- ▶ Hard to analyze pointer references
  - ▶ Determine if two pointers reference same memory location
  - ▶ Undecidable for dynamically allocated data structures
  - ▶ Hard if code uses with pointer arithmetic
- ▶ Array-indexed arrays, sparse arrays, indirect references
- ▶ Input-dependent dependences
- ▶ Inter-procedural dependences, analysis beyond basic blocks
- ▶ Conservatism of analysis
  - ▶ **Correctness precedes performance in compilers**

## Other compiler optimizations

### Copy propagation

```
ADDI R1, R2, 4  
ADDI R1, R1, 4
```

```
ADDI R1, R2, 8
```

### Tree height reduction

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

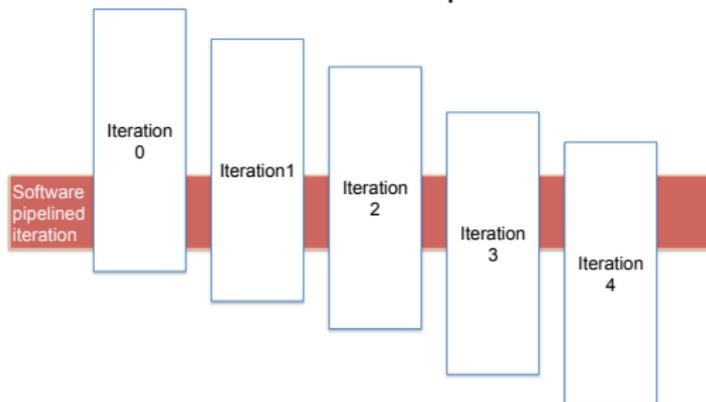
```
ADD R1, R2, R3  
ADD R4, R7, R6  
ADD R8, R1, R4
```

- ▶ Assumes addition is associative (not true in FP arithmetic)

# Software pipelining

## Symbolic loop unrolling

- ▶ Benefits of loop unrolling with reduced code size
- ▶ Instructions in loop body selected from **different loop iterations**
  - ▶ Increase distance between dependent instructions



## Software pipelining

### Loop unrolled 3 times

```
Iteration i:  LD F0,0(R1)
              ADDD F4,F0,F2
              SD F4,0(R1)
Iteration i+1: LD F0,0(R1)
              ADDD F4,F0,F2
              SD 0(R1),F4
Iteration i+2: LD F0,0(R1)
              ADDD F4,F0,F2
              SD F4,0(R1)
```

### Software pipelined loop

```
Loop:  SD      F4,16(R1)  #store to v[i]
        ADDD   F4,F0,F2  #add to v[i-1]
        LD     F0,0(R1)  #load v[i-2]
        ADDI   R1,R1,-8
        BNE   R1,R2,Loop
```

- ▶ 5 cycles/iteration (with dynamic scheduling and renaming)
- ▶ Need startup/cleanup code

## Software pipelining (cont.)

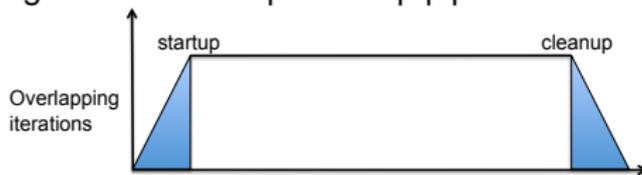
### SW pipelined loop with startup and cleanup code

```
#startup, assume i runs from 0 to n
ADDI  R1,R1,-16 #point to v[n-2]
LD    F0,16(R1) #load v[n]
ADDD  F4,F0,F2  #add v[n]
LD    F0,8(R1)  #load v[n-1]
#body for (i=2;i<=n-2;i++)
Loop: SD    F4,16(R1) #store to v[i]
      ADDD  F4,F0,F2  #add to v[i-1]
      LD    F0,0(R1)  #load v[i-2]
      ADDI  R1,R1,-8
      BNE  R1,R2,Loop
#cleanup
SD    F4,8(R1)  #store v[1]
ADDD  F4,F0,F2  #add v[0]
SD    F4,0(R1)  #store v[0]
```

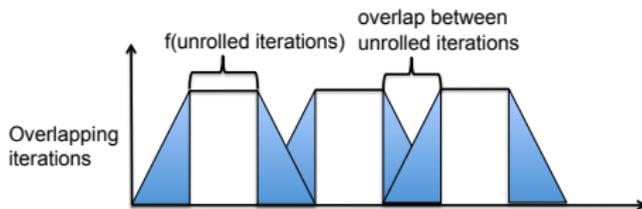
# Software pipelining versus unrolling

## Performance effects of SW pipelining vs. unrolling

- ▶ Unrolling reduces loop overhead per iteration
- ▶ SW pipelining reduces startup-cleanup pipeline overhead



(a) software pipelining



(b) loop unrolling

## Software pipelining (cont.)

### Advantages

- ▶ Less code space than conventional unrolling
- ▶ Loop runs at peak speed during steady state
  - ▶ Overhead only at loop initiation and termination
  - ▶ Complements unrolling

### Disadvantages

- ▶ Hard to overlap long latencies
  - ▶ Unrolling combined with SW pipelining
- ▶ Requires advanced compiler transformations

# Predication

## Conditional move

- ▶ A predicated instruction packs a conditional and an instruction
  - ▶ Instruction control-dependent on conditional
  - ▶ If conditional is false instruction is converted to no-op, otherwise executed
- ▶ Convert control dependence to data dependence

```
#if (A==0) {S=T;}  
# simple translation  
    BNEZ R1, L    #if (A==0)  
    ADDI R2, R3, 0    #S=T;  
L:  
# predicated instruction  
    CMOVZ R2, R3, R1    #move T to S if R1=0
```

# Predication

## Generalized predication

- ▶ Predicates applied to all instructions
- ▶ Enables predicated execution of large code blocks
- ▶ Speculatively puts time-critical instructions under predicates

## No predication

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

## Predication

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

## Predication implementation issues

### Preserve control and data flow, precise interrupts

- ▶ Speculative predicated instructions may not throw illegal exceptions
  - ▶ LWC may not throw exception if  $R10 == 0$
  - ▶ LWC may throw recoverable page fault if  $R10 \neq 0$
- ▶ Instruction conversion to nop
  - ▶ Early condition detection may not be possible due to data dependence
  - ▶ Late condition detection incurs stalls and consumes pipeline resources needlessly
- ▶ Instructions may be dependent on multiple branches
- ▶ Compiler able to find instruction slots and reorder

# Hardware support for speculation

## Alternatives for handling speculative exceptions

- ▶ Hardware and OS ignore exceptions from speculative instructions
- ▶ Mark speculative instructions and check for exceptions
  - ▶ Additional instructions to check for exceptions and recover
- ▶ Registers marked with poison bits to catch exceptions upon read
- ▶ Hardware buffers instruction results until instruction is no longer speculative

## Exception classes

- ▶ **Recoverable**: exception from speculative instruction may harm performance, but not preciseness
- ▶ **Unrecoverable**: exception from speculative instruction compromises preciseness

## Solution I: Ignore exceptions

### HW/SW solution

- ▶ Instruction causing exception returns undefined value
- ▶ Value not used if instruction is speculative
- ▶ Incorrect result if instruction is non-speculative
  - ▶ Compiler generates code to throw regular exception
- ▶ **Rename** registers receiving speculative results

### Non-speculative

```
# if (A==0) A=B; else A=A+4;
    LD   R1,0(R3)  ;load A
    BNEZ R1,L1    ;test A
    LD   R1,0(R2)  ;then load B
    J    L2
L1:   ADDI R1,R1,4 ;else
L2:   SD   R1,0(R3) ;store A
```

### Speculative

```
# if (A==0) A=B; else A=A+4;
    LD   R1,0(R3)  ;load A
    LD   R4,0(R2)  ;speculative load B
    BEQZ R1,L3    ;test A
    ADDI R4,R1,4   ;else
L3:   SD   R4,0(R3) ;non-speculative store
```

## Solution II: mark speculative instructions

```
# if (A==0) A=B; else A=A+4;
    LD   R1,0(R3)    ;load A
    SLD  R4,0(R2)    ;speculative load B
    BNEZ R1,L1      ;test A
    CHK  R4,recover ;speculation check
    J    L2          ;skip else
L1:   ADDI R4,R1,4   ;else
L2:   SD   R4,0(R3) ;store A
recover:...
```

- ▶ Instruction checking speculation status
- ▶ Jump to recovery code if exception
- ▶ Itanium CHK instruction

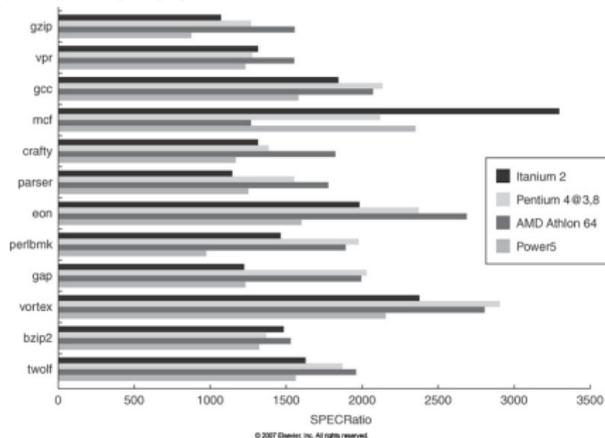
## Solution III: poison bits

```
# if (A==0) A=B; else A=A+4;
    LD   R1,0(R3)    ;load A
    SLD  R4,0(R2)    ;speculative load B
    BEQZ R1,L3      ;test A
    ADDI R4,R1,4    ;else
L3:   SD   R4,0(R3)  ;store A
```

- ▶ R4 marked with poison bit
- ▶ Use of R4 in SD raises exception if SLD raises exception
- ▶ Generate exception when result of offending instruction is used for the first time
- ▶ OS code needs to save poison bits during context switching

## Performance of VLIW processors

### Itanium vs. Alpha vs. Pentium 4



- ▶ Low INT performance
- ▶ Better FP performance, highly application-dependent
- ▶ Poor power-efficiency (performance/watt)

## What next?

### Alternatives to exploit parallelism

- ▶ **Vector processors and SIMD** – next lecture
- ▶ **Simultaneous multithreading** – lecture after next