

Chapter 4

Instruction-Level Parallelism: Software Approaches Part 1

Slides: D. Patterson, W. Gross, V. Hayward, T. Arbel

Exposing ILP in the Compiler

- Last chapter: exploiting ILP in hardware.
 - Binary compatibility
- What if we can change the ISA?
- Static scheduling, branch prediction and issue
 - Requires advanced compiler techniques
- The ideas in this chapter are behind two of the newest ISAs: The Intel Itanium (workstation, server) and the Transmeta Crusoe (low-power embedded)
- Also used in DSP chips

Basic Compiler Scheduling

- The idea: keep the pipeline full
 - Avoid stalls due to hazards
- Scheduling
 - find a sequence of instructions that can be overlapped in the pipeline
- We will look at scheduling in the compiler. The hardware then executes the scheduled code in-order
- How do we achieve our goal of keeping the pipeline full ?

Basic Compiler Scheduling

- A dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of the source instruction
- For example, in a pipeline with forwarding
 - latency of the EX stage (ALU) is 0.
 - The data memory latency is 1
- A compiler's ability to perform this scheduling depends on:
 - The amount of ILP in the program
 - The latencies of the functional units

Basic Compiler Scheduling

- Assume the classic 5-stage integer pipeline
- Integer ALU latency is 0 CC
- Integer load latency is 1 CC
- Branch delay is 1 CC
- Fully pipelined FUs (assume no structural hazards)
- Assume the following FP latencies (averages):

Producer	Consumer	Latency (CCs)
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Loop Example

- Adding a scalar to a vector (loop is parallel since the body of each iteration is independent)

```
for (i = 1000; i > 0; i=i-1)
    x[i] = x[i] + s;
```

```
Loop:  L.D      F0,0(R1)      ;F0=array element
        ADD.D   F4,F0,F2     ;add scalar from F2
        S.D     F4,0(R1)     ;store result
        DADDUI  R1,R1,#-8    ;decrement pointer 8 bytes
        BNE    R1,R2,Loop    ;branch R1!=R2
```

Loop Example

- Unscheduled code: 10 clock cycles

```
1  Loop:      L.D      F0,0(R1)
2              stall
3              ADD.D   F4,F0,F2
4              stall
5              stall
6              S.D     F4,0(R1)
7              DADDUI  R1,R1,#-8
8              stall
9              BNE     R1,R2,Loop
10             stall
```

Loop Example

- Scheduled code: 6 cycles
- Not trivial: S.D. depends on DAADUI. Swap them but change address
- Problem: only doing work on the array element in 3/6 cycles. Other 3 are for loop overhead

```
1  Loop:      L.D      F0,0(R1)
2              DADDUI   R1,R1,#-8
3              ADD.D    F4,F0,F2
4              stall
5              BNE     R1,R2,Loop ; delayed branch
6              S.D     F4,8(R1)  ; altered
```


Loop Unrolling

- *Unroll* the loop
 - Replicate the body of the loop many times
 - Adjust the loop termination code
- Eliminating the branch allows instructions from different iterations to be scheduled together
 - In this case we can eliminate the data stall

Unroll Loop Four Times (straightforward way)

```
1 Loop:L.D    F0,0(R1)
2    ADD.D   F4,F0,F2
3    S.D    F4,0(R1)
4    L.D    F6,-8(R1)
5    ADD.D   F8,F6,F2
6    S.D    F8,-8(R1)
7    L.D    F10,-16(R1)
8    ADD.D   F12,F10,F2
9    S.D    F12,-16(R1)
10   L.D    F14,-24(R1)
11   ADD.D   F16,F14,F2
12   S.D    F16,-24(R1)
13   DADDUI R1,R1,#-32
14   BNE    R1,R2,LOOP
```

Annotations:

- 1 cycle stall (between lines 1 and 2)
- 2 cycles stall (between lines 2 and 3)
- 1 cycle stall (between lines 13 and 14)

Green annotations:

- `;drop DADDUI & BNE` (next to lines 3, 6, 9)
- `;alter to 4*8` (next to line 13)

Rewrite loop to
minimize stalls?

$14 + 4 \times (1+2) + 2 = 28$ clock cycles, or 7 per iteration
Assumes R1 is multiple of 32 (# loops a multiple of 4)

Unrolled Loop Detail

- Do not usually know upper bound of loop
- Suppose it is n , and we would like to unroll the loop to make k copies of the body
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
 - 1st executes $(n \bmod k)$ times and has a body that is the original loop
 - 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
 - For large values of n , most of the execution time will be spent in the unrolled loop

Unrolled Loop That Minimizes Stalls

```
1 Loop: L.D    F0,0(R1)
2       L.D    F6,-8(R1)
3       L.D    F10,-16(R1)
4       L.D    F14,-24(R1)
5       ADD.D  F4,F0,F2
6       ADD.D  F8,F6,F2
7       ADD.D  F12,F10,F2
8       ADD.D  F16,F14,F2
9       S.D    F4,0(R1)
10      S.D    F8,-8(R1)
11      DADDUI R1,R1,#-32
12      S.D    F12,-16(R1)
13      BNE   R1,R2,LOOP
14      S.D    F16,8(R1) ; 8-32 = -24
```

- **What assumptions made when moved code?**

- OK to move store past DADDUI even though changes register
- OK to move loads before stores: get right data?
- When is it safe for compiler to do such changes?

14 clock cycles, or 3.5 per iteration

Compiler Perspectives on Code Movement

- Compiler concerned about dependencies in **program**
- Whether or not a HW hazard depends on **pipeline**
- Try to schedule to avoid hazards that cause performance losses
- (True) **Data dependencies** (RAW if a hazard for HW)
 - Instruction i produces a result used by instruction j , or
 - Instruction j is data dependent on instruction k , and instruction k is data dependent on instruction i .
- If dependent, can't execute in parallel
- Easy to determine for registers (fixed names)
- Hard for memory ("**memory disambiguation**" problem):
 - Does $100(R4) = 20(R6)$?
 - From different loop iterations, does $20(R6) = 20(R6)$?

Where are the name dependencies?

```
1 Loop:L.D    F0,0(R1)
2      ADD.D  F4,F0,F2
3      S.D    F4,0(R1)      ;drop DADDUI & BNE
4      L.D    F0,-8(R1)
5      ADD.D  F4,F0,F2
6      S.D    F4,-8(R1)    ;drop DADDUI & BNE
7      L.D    F0,-16(R1)
8      ADD.D  F4,F0,F2
9      S.D    F4,-16(R1)  ;drop DADDUI & BNE
10     L.D    F0,-24(R1)
11     ADD.D  F4,F0,F2
12     S.D    F4,-24(R1)
13     DADDUI R1,R1,#-32   ;alter to 4*8
14     BNE    R1,R2,LOOP
15     NOP
```

How can remove them?

Where are the name dependencies?

```
1 Loop:L.D    F0,0(R1)
2      ADD.D  F4,F0,F2
3      S.D    F4,0(R1)      ;drop DADDUI & BNE
4      L.D    F6,-8(R1)
5      ADD.D  F8,F6,F2
6      S.D    F8,-8(R1)    ;drop DADDUI & BNE
7      L.D    F10,-16(R1)
8      ADD.D  F12,F10,F2
9      S.D    F12,-16(R1)  ;drop DADDUI & BNE
10     L.D    F14,-24(R1)
11     ADD.D  F16,F14,F2
12     S.D    F16,-24(R1)
13     DADDUI R1,R1,#-32    ;alter to 4*8
14     BNE    R1,R2,LOOP
15     NOP
```

The Original "register renaming"

Compiler Perspectives on Code Movement

- Name dependencies are hard to discover for memory Accesses
 - Does $100(R4) = 20(R6)$?
 - From different loop iterations, does $20(R6) = 20(R6)$?
- Our example required compiler to know that if R1 doesn't change then:

$$0(R1) \neq -8(R1) \neq -16(R1) \neq -24(R1)$$

There were no dependencies between some loads and stores so they could be moved by each other

Steps Compiler Performed to Unroll

- Check OK to move the S.D after DADDUI and BNEZ, and find amount to adjust S.D offset
- Determine unrolling the loop would be useful by finding that the loop iterations were independent
- Rename registers to avoid name dependencies
- Eliminate extra test and branch instructions and adjust the loop termination and iteration code
- Determine loads and stores in unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent
 - requires analyzing memory addresses and finding that they do not refer to the same address.
- Schedule the code, preserving any dependences needed to yield same result as the original code

Drawbacks

- Code length (an issue for embedded processors)
- Uses lots of registers
 - "Register pressure"
 - Could be a problem with aggressive unrolling and scheduling, especially on multiple issue machines

Multiple Issue

- Consider a simple statically scheduled 2-issue MIPS

	Integer instruction	FP instruction
Loop	L.D F0, 0 (R1)	
	L.D F6, -8 (R1)	
	L.D F10, -16 (R1)	ADD.D F4, F0, F2
	L.D F14, -24 (R1)	ADD.D F8, F6, F2
	L.D F18, -32 (R1)	ADD.D F12, F10, F2
	S.D F4, 0 (R1)	ADD.D F16, F14, F2
	S.D F8, -8 (R1)	ADD.D F20, F18, F2
	S.D F12, -16 (R1)	
	DADDUI R1, R1, #-40	
	S.D F16, 16 (R1)	
	BNE R1, R2, Loop	
	S.D F20, 8 (R1)	

2.4 cc per iteration

Static Branch Prediction

- We saw this idea earlier
 - Delayed branches

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

Static Branch Prediction Strategies

- **Predict-taken**
 - Midprediction rate = untaken branch frequency
 - SPEC: 34% misprediction (9% to 59%)
- **Predict based on branch direction**
 - E.g. predict forward-going branches as not taken and backwards-going branches as taken
- **Collect profile information by running the program a few times. Recompile with this profile information.**
 - Studies have showed that even when the data changes the profile is pretty accurate

Static Branch Prediction

- **Static branch prediction is useful when:**
 1. Branch delays are exposed by architecture
 2. Assisting dynamic predictors (IA-64)
 3. Determining which code paths are more frequent (for code scheduling)

Static Multiple Issue: VLIW

- **Recall superscalar multiple-issue processors:**
 - Decide how many instructions to issue on-the-fly
- **Statically scheduled superscalar:**
 - HW to check for dependencies between instructions in a packet and between instructions in a packet and ones already in the pipeline
- **What if we do the dependence checking in the compiler?**
 - Format an instruction packet with either no dependencies or at least indicate if they are present
 - Simpler hardware

VLIW

- Very long instruction word (VLIW)
- Idea has been around for a long time
- 64 to 128 bit packets
- Drawback: they can be inflexible.
 - Requires recompilation for different versions of the hardware
- Latest versions use software to assist hardware decisions (EPIC → IA-64)

The VLIW Idea

- Multiple, independent FUs
- Find independent operations and package them together into a very long instruction word
- Eliminates the expensive hardware that does this in a superscalar
- Superscalar processors are especially expensive for wide issue widths (e.g. > 4) so VLIW machines tend to focus on issue widths of > 4

VLIW

- **E.g. 5-issue VLIW**
 - 1 integer (incl. branch)
 - 2 FP
 - 2 memory ref.
- **Code must have enough parallelism to fill the operation slots and keep the FUs busy**
- **Find this parallelism by loop unrolling and scheduling**

VLIW Example

Mem Ref 1	Mem Ref 2	FP op1	FP op2	Int. op/Branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-24(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-32(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

- 9 cycles
- 23 operations
- 2.5 operations / cycle
- Efficiency (percent of available slots used) = 60%
- Large number of registers used !

VLIW Issues

- **Increased code size**
 - Need to aggressively unroll loops
 - Waste bits whenever instructions are not full
 - Use clever encoding or compression
- **Limitations of lock-step operation**
 - No hazard detection h/w
 - A stall in one FU must stall the whole processor (can't predict cache stalls)
 - Recent processors relax this and use h/w to allow unsynchronized execution
- **Binary code compatibility**
 - Different pipeline organizations require different code (i.e. more FUs)
 - One solution: object code translation (Crusoe: rapidly developing)
 - Another solution: relax this approach (IA-64)

Chapter 4

Instruction-Level Parallelism: Software Approaches Part 2

Slides: D. Patterson, W. Gross, V. Hayward, T. Arbel

Advanced Compiler Support

- We will study techniques used by modern compilers such as gcc
- Dependencies: **true** and **name**
- This concept also applies to high-level code
- Compilers can detect parallelism in high-level code that hardware would be blind to

```
for (i = 1000; i > 0; i=i-1)
    x[i] = x[i] + s
```

Loop-Carried Dependencies

```
for (i = 1000; i > 0; i=i-1)
    x[i] = x[i] + s
```

- If data accesses in an iteration depend on data values produced in earlier iterations we say there is a **loop-carried dependence**
- This is a **parallel** loop since there are no loop-carried dependencies.
 - Except for the "induction variable" i , but this can be recognized and eliminated (e.g. loop unrolling)

Detecting and Exposing Loop-Level Parallelism

- Inspect the code to detect name and data dependencies
- Name dependencies can be eliminated by using more storage (“software renaming”)
 - Left with a chain of data dependencies
- If the data dependency chain can be broken, then the loop has some parallelism
- If all data dependencies are within one iteration, the loop is parallel

Loop-Carried Dependencies

- Dependencies can exist between statements in a block or across blocks
- Example: recurrences
 - A variable is defined based on the value of that variable in an earlier iteration

e.g.

```
for (i=0;i<=100;++i)
    y[i] = y[i-5] + y[i]
```

Carries a dependency with a **dependence distance** of 5

Finding Dependencies in Loops

- *Need to analyze memory references to look for ones that refer to the same addresses*
- *Difficult in the general case*

e.g. X[Y[i]]

Finding Dependencies in Loops

- Consider finding dependencies in the case when the array indices are "affine"
- An affine index has the form $ai + b$ where i is the loop index and a and b are constants
- To detect a dependence, we need to determine if two affine array indices are equal. i.e

$$ai + b = ci + d$$

GCD Test

- A sufficient test to test for the *absence* of a dependency is the GCD test:
- for references $a_i + b$ and $c_i + d$, if a loop dependency exists, then $\text{GCD}(c, a)$ divides $(d-b)$
 - x divides y if y/x is an integer and there is no remainder
- Therefore, do the GCD test. If $\text{GCD}(c, a)$ does not divide $d-b$ then there is no dependency.
 - However, the case exists where $\text{GCD}(c, a)$ divides $d-b$ and there is still no dependency. (because the loop bounds are not considered)

Examples of GCD Test

```
for(i=1;i<=100;++1)
  x[2i+3] = x[2i] + 1.0
```

- **GCD(2,2) does not divide -3**
 - No dependency is possible

```
for(i=1;i<=100;++1)
  x[2i+3] = x[2i+1] + 1.0
```

- **2 divides -2**
 - dependency is possible
- **In general, deciding if a dependency definitely exists requires an algorithm with an exponential number of steps ("NP-complete") and is not practical**
 - A few important sub cases are implemented in modern compilers

Classifying Dependencies

- In addition to detecting the presence of dependencies, compilers want to classify the type of dependencies
- E.g. Find the dependencies in:

```
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 dependence

Antidependence

Output dependence

Example cont'd

```
for (i=1;i<100;i=i+1){  
    /* Y renamed to T to remove o.d. */  
    T[i] = X[i] / c;  
    /* X renamed to U to remove a.d. */  
    U[i] = X[i] + c;  
    /* Y renamed to T to remove a.d. */  
    Z[i] = T[i] + c;  
    Y[i] = c - T[i];  
}
```

- Second statement is now independent
- Third and fourth only dependent on first

Compiler Loop-Level Transformations

- Transform this loop to make it parallel

```
for (i=1; i < 100; i++) {  
    a[i] = b[i] + c[i];    /* S1 */  
    b[i] = a[i] + d[i];    /* S2 */  
    a[i+1] = a[i] + e[i];  /* S3 */  
}
```


Dependence Analysis

```
for (i=1; i < 100; i++) {  
    a[i] = b[i] + c[i];      /* S1 */  
    b[i] = a[i] + d[i];     /* S2 */  
    a[i+1] = a[i] + e[i];   /* S3 */  
}
```

true data dependency
(not loop-carried)

Output dependency
(loop-carried)

true data dependency
(loop-carried)

Antidependency
(not loop-carried)

Dependence Analysis

```
a[1] = b[1] + c[1];    /* S1 */
b[1] = a[1] + d[1];    /* S2 */
a[2] = a[1] + e[1];    /* S3 */
a[2] = b[2] + c[2];    /* S1 */
b[2] = a[2] + d[2];    /* S2 */
a[3] = a[2] + e[2];    /* S3 */
a[3] = b[3] + c[3];    /* S1 */
b[3] = a[3] + d[3];    /* S2 */
a[4] = a[3] + e[3];    /* S3 */
...
```

- **S3 does no useful work as its result is overwritten by S1 (except on last iteration)**


Remove S3

```
for (i=1; i < 100; i++) {  
    a[i] = b[i] + c[i];    /* S1 */  
    U[i] = a[i] + d[i];    /* S2 */  
}  
a[100] = a[99] + e[99];
```

- Remove antidependence by software renaming
- No loop carried dependencies (parallel loop)

Another Example of LLP

```
for (i=1; i < 100; i++) {  
    a[i] = a[i] + b[i];    /* S1 */  
    b[i+1] = c[i] + d[i]; /* S2 */  
} True Dep (loop carried)
```



- No dependence from S1 to S2
- Can this loop be made parallel?
- No **cycles** in the dependencies, so yes!

Transformed Parallel Loop

```
a[1] = a[1] + b[1]
for (i=1; i <= 99; i++) {
    b[i+1] = c[i] + d[i];
    a[i+1] = a[i+1] + b[i+1];
}
b[101] = c[100] + d[100]
```

Algebraic Optimization of Recurrences

- E.g. `sum = sum + x;`
- Unroll a loop with this recurrence 5 times
`sum = sum + x1 + x2 + x3 + x4 + x5;`
- 5 dependent operations
- Algebraic optimization

$$\text{sum} = ((\text{sum} + x1) + (x2 + x3)) + (x4 + x5)$$

- 3 dependent operations

Arithmetic Techniques

- Transformations based on associative and commutative properties of arithmetic
 - not true for limited range and precision, so be careful...
 - Compilers usually will not do these unless explicitly enabled

Back Substitution

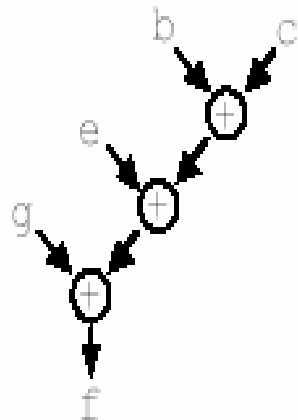
- E.g. replace

```
DADDUI R1,R2,#4    /* a = b + 4 */  
DADDUI R1,R1,#4    /* a = a + 4 */
```

with

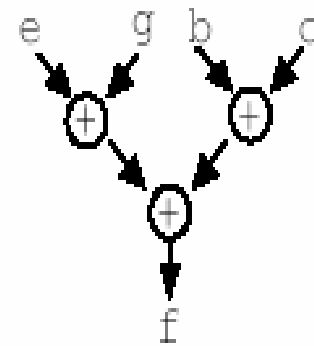
```
DADDUI R1,R2,#8    /* a = b + 8 */
```


Tree Height Reduction



```
ADDR1,R2,R3 // a=b+c  
ADDR4,R1,R6 // d=a+e  
ADDR8,R4,R7 // f=d+g
```

Three clock cycles



```
ADDR1,R2,R3 // a=b+c  
ADDR4,R6,R7 // d=e+g  
ADDR8,R1,R4 // f=a+d
```

Two clock cycles

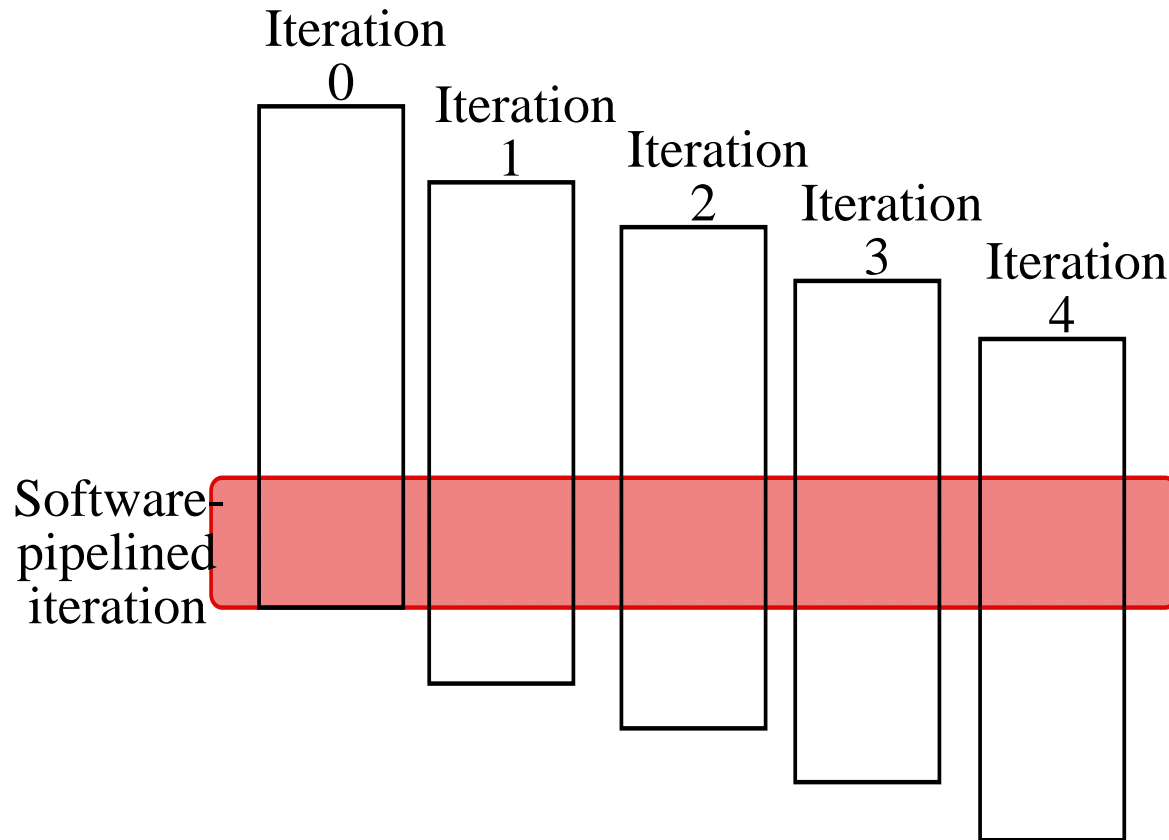
This applies to cases such as

$sum = sum + x1 + x2 + x4 + x5 = ((sum + x1) + (x2 + x3)) + (x4 + x5)$,
etc. The goal of all these transformations is to reduce unnecessary dependencies.

Software Pipelining

- The general idea of these optimizations is to uncover long sequences of statements without control statements
- Reorganize loops to interleave instructions from different iterations
 - This is the software counterpart to what Tomasulo's algorithm does in hardware
- Dependent instructions within a single loop iteration are then separated from one another by an entire loop body
 - Increases possibilities of scheduling without stalls

Software Pipelining



Software Pipelining Example

```
Loop:      L.D.      F0,0(R1)
           ADD.D     F4,F0,F2
           S.D       F4,0(R1)
           DADDUI    R1,R1,#-8
           BNE      R1,R2,LOOP
```

- 10 cycles

Step 1: Symbolic Loop Unrolling

ITER i	L.D.	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)
ITER $i+1$	L.D.	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)
ITER $i+2$	L.D.	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)

Step 2: Select Instructions from Different Iterations

ITER i	L.D.	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)
ITER $i+1$	L.D.	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)
ITER $i+2$	L.D.	F0,0(R1)
	ADD.D	F4,F0,F2
	S.D	F4,0(R1)

Step 3. Combine into loop and add init and cleanup code

INIT CODE

```
Loop:  S.D.          F4,16(R1) ;stores into M[i]
        ADD.D       F4,F0,F2  ;adds to M[i-1]
        L.D         F0,0(R1)  ;loads M[i-2]
        DADDUI      R1,R1,#-8
        BNE         R1,R2,LOOP
```

CLEAN UP CODE

- 5 clock cycles (assuming DAADUI scheduled before the ADD.D and the L.D is scheduled in the branch delay slot)

Software Pipelining

- **Advantage:** yields shorter code than loop unrolling and uses fewer registers
- **Software pipelining is crucial for VLIW processors**
 - The above example could be compiled into one instruction
- **Often, both software pipelining and loop unrolling are used**

Global Code Scheduling

- Things get complex if there is control flow inside the loop this requires moving instructions across branches
 - global code scheduling
- Find the the longest sequence of dependent calculations (critical path)
- Compress the critical path to the shortest sequence of instructions that preserves control and data dependencies
- We will not cover this topic

Chapter 4

Instruction-Level Parallelism: Software Approaches Part 3

Slides: D. Patterson, W. Gross, V. Hayward, T. Arbel

Hardware Support For Exposing Parallelism at Compile Time

- Recent hardware adds support to facilitate the job of compilers
 - Conditional instructions
 - Hardware support for compiler speculation

Conditional Instructions

- **Conditional (or “predicated”) instructions**
 - Refer to a condition
 - If the condition is true, then execute the instruction normally
 - If the condition is false, cancel the instruction (the execution continues as if the instruction was a nop)
- **Can be used to eliminate branches**
 - Control dependence is converted to a data dependence
 - Moves the condition evaluation from near the front of the pipeline to the end of the pipeline (register write)
 - Could improve performance

Conditional Moves

```
if (a == 0)
    s = t;
```

```
        BNEZ          R1, skip
        ADDU          R2, R3, R0
```

Skip:

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

- Control dependence has been converted to a data dependence

Example

- Absolute value

`A = abs(b)`

`if (b < 0) {a = -b} else {a = b}`

- Implement as a pair of conditional moves or as one unconditional move (`a = b`) and one conditional (`a = -b`)

Conditional Instructions

- MIPS, Alpha, PowerPC, SPARC and x86 all support conditional moves
- IA-64 supports full predication !
 - Every instruction is conditional
 - Can convert blocks of code that are branch dependent
 - More complex (can lower clock rate)
- Example (dual issue)

First slot	Second slot
LW R1,40(R2)	ADD R3,R4,R5
	ADD R6,R3,R7
BEQZ R10,L	
LW R8,0(R10)	
LW R9,0(R8)	

First slot	Second slot
LW R1,40(R2)	ADD R3,R4,R5
LWC R8,0(R10),R10	ADD R6,R3,R7
BEQZ R10,L	
LW R9,0(R8)	

speculation

Compiler Speculation with Hardware Support

- To implement speculation a compiler has to move control dependent instructions before a branch
 - Predicated (conditional) instructions provide one way to speculate
- In many cases we would like to move instructions even before the condition evaluation (predicated instructions will not work here)
 1. Need to find instructions that can be speculatively moved (with possible register renaming) and not affect data flow
 2. Need to ignore exceptions in speculated instructions until we know that they should really occur
 3. Need to be able to speculatively exchange loads and stores or stores and stores which might have address conflicts
- Need h/w support for 2 and 3

Compiler Speculation Example

`if (a == 0) then a = b else a = a + 4`

```
LD      R1,0(R3)      ; load A
BNEZ    R1,L1         ; test A
LD      R1,0(R2)      ; then clause
J       L2            ; skip else
L1:     DADDI R1,R1,#4 ; else clause
L2:     SD      R1,0(R3) ; store A
```

Compiler Speculation Example

- Assume the “then” clause is almost always executed
- With compiler speculation:

```
LD      R1,0(R3)      ; load A
LD      R14,0(R2)     ; spec load B
BNEZ    R1,L3         ; other branch of if
DADDI   R14,R1,#4     ; else clause
L3:     SD      R14,0(R3) ; non spec store
```

- The “then” clause is executed speculatively (hoping it is frequent case)
- If prediction is incorrect, DADDI will overwrite the incorrect value in R14

Exceptions

- Exception behaviour has been changed by the transformation
 - The speculative load could generate a page fault that would not have happened if the load was left in the control dependent code
- To deal with this, processors introduce hardware mechanisms to preserve exception behaviour of speculative code
 - Speculation check instructions (pseudo nonexcepting loads)
 - Poison bits to turn off register writes
- Details are skipped

Transmeta Crusoe

- Code compatible with x86
- Low power
- VLIW
- 6-stage in-order integer pipeline
- 10-stage floating point pipeline
- x86 instructions translated on-the-fly into Crusoe instructions in firmware
 - "Code Morphing™"

Crusoe Instruction Format

Memory | Generic FP/Int op | Integer op | Immediate

Memory | Generic FP/Int op | Integer op | Branch

Code Morphing

- **x86 code interpreted instruction-by-instruction**
- **Basic blocks cached and reused**
- **Speculative reordering**
 - **Shadowed register file**
 - **Speculative loads and conditional moves**
- **Low-power features**
 - **Dynamic voltage and frequency scaling !**

Summary

- Complex compiler optimizations needed to extract significant amounts of ILP
- As issue rate increases, the gap between peak and sustained performance grows quickly
- No “silver bullet” approach to building multiple-issue processors
- Over time, techniques from h/w and s/w techniques are sneaking into the other

Guess the Processors

Issue Rate	Clock Rate (2001)	Transistors w/without caches (M)	Power (W)	SPECbase CPU2000 int/fp
4	1 GHz	15 / 6 Alpha 21264	107	561/643
3	2 GHz	42 / 23 Pentium 4	67	636/648
3	1 GHz	28 / 9.5 Pentium 3	36	454/329
6	0.8 GHz	25 / 17 Itanium	130	379/714