EEF011 Computer Architecture
計算機結構

Chapter 4
Exploiting Instruction-Level Parallelism with Software Approaches

吳俊興
高雄大學資訊工程學系

November 2004

Chapter Overview

4.1 Basic Compiler Techniques for Exposing ILP

4.2 Static Branch Prediction

4.3 Static Multiple Issue: The VLIW Approach

4.4 Advanced Compiler Support for ILP

4.7 Intel IA-64 Architecture and Itanium Processor

4.1 Basic Compiler Techniques for Exposing ILP

- To avoid a pipeline stall, a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction
- How can compilers recognize and take advantage of ILP? It depends on
  - the amount of ILP available in the program
  - the latencies of the functional units in the pipeline (assume no structural hazards)
- Basic compiler technique – Loop Unrolling

Instruction producing result	Instruction using result	Latency in clock cycles
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

Figure 4.1 Latencies of FP operations used in this chapters. Assume an integer load latency of 1. Dranches have a delay of one clock cycle.

Basic Pipeline Scheduling

Example: adding a scalar to a vector

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

Straightforward MIPS code

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

Without any scheduling (10 cycles per loop)

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

Basic pipeline scheduling (6 cycles per loop; 3 for loop overhead: DADDUI and BNE)

```
Loop:  L.D     F0,0(R1)
       DADDUI  R1,R1,#-8
       ADD.D   F4,F0,F2
       stall
       BNE     R1,R2,Loop ;delayed branch
       S.D     F4,8(R1)   ;altered & interchanged with DADDUI
```

Loop Unrolling

Four copies of the loop body
- Eliminate 3 branch 3 decrements of R1 => improve performance
- Require symbolic substitution and simplification
- Increase code size substantially => expose more computation that can be scheduled => Gain from scheduling on unrolled loop is larger than original

Unrolling without Scheduling(28 cycles/14 instr.)

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

Unrolling with Scheduling(14 cycles)

```
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)
        DADDUI  R1,R1,#-32
        S.D     F12,16(R1)
        BNE     R1,R2,Loop
        S.D     F16,8(R1);8-32 = -24
```

Each L.D has 1 stall, each ADD.D 2, the DADDUI 1, the branch 1, plus 14 instruction issue cycles

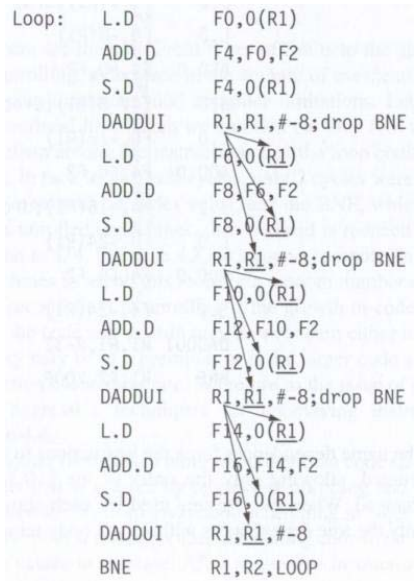
Summary of Loop Unrolling Example

Decisions and transformations:
1. Determine that it was legal to move the SD after the DADDUI and BNE, and find the amount to adjust the SD offset.
2. Determine that unrolling the loop would be useful by finding that the loop iterations were independent, except for the loop maintenance code.
3. Use different registers to avoid unnecessary constraints that would be forced by using the same registers for different computations.
4. Eliminate the extra tests and branches and adjust the loop termination and iteration code.
5. Determine that the loads and stores in the unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent. This requires analyzing the memory addresses and finding that they do not refer to the same address.
6. Schedule the code, preserving any dependences needed to yield the same result as the original code.

Key requirements: understanding of
- how an instruction depends on another
- how the instructions can be changed or reordered given the dependences

Symbolic Substitution to Remove Data Dependence

Unrolled but unoptimized with extra DADDUI



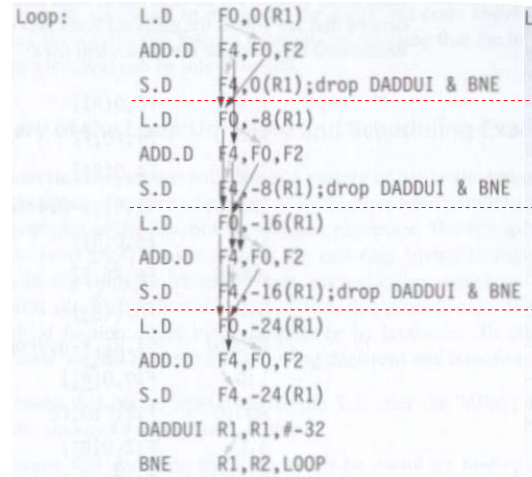
- symbolically computing the intermediate values and folding the computation into L.D and S.D instruction
- changing final DADDUI into a decrement by 32

Three DADDUI removed

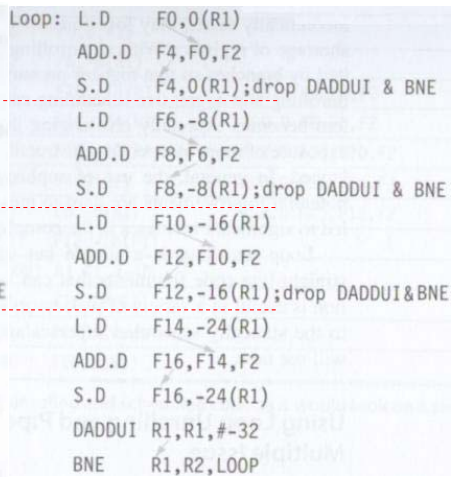
```
Loop:   L.D     F0,0(R1)
        ADD.D   F4,F0,F2
        S.D     F4,0(R1);drop DADDUI & BNE
        L.D     F6,-8(R1)
        ADD.D   F8,F6,F2
        S.D     F8,-8(R1);drop DADDUI & BNE
        L.D     F10,-16(R1)
        ADD.D   F12,F10,F2
        S.D     F12,-16(R1);drop DADDUI&BNE
        L.D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


Register Renaming to Remove Name Dependence

gray arrow: data dependence
black arrow: name dependence



Each loop becomes independent
Only true dependences remain (gray arrow)



(Software renaming by compilers)

Loop Unrolling with Pipeline Scheduling

Original loop

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

Loop unrolling only

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

Loop unrolling + pipeline scheduling

```
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)
      DADDUI R1,R1,#-32
      S.D    F12,16(R1)
      BNE    R1,R2,Loop
      S.D    F16,8(R1);8-32 = -24
```

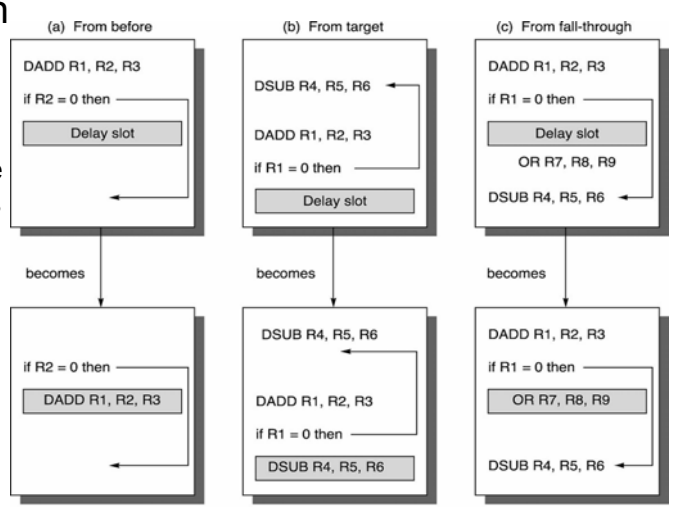
Three limits to the gains achievable by loop unrolling

1. loop overhead
   - decrease in the amount of overhead amortized with each unroll
2. code size limitations
   - larger loops => larger code size growth => larger instruction cache miss rate
3. compiler limitations
   - potential shortfall in registers

4.2 Static Branch Prediction

- used where branch behavior is highly predictable at compile time
- architectural feature to support static branch prediction
  – delayed branch

The instruction in the ***branch delay slot*** is executed whether or not the branch is taken (for zero cycle penalty)



Loop Unrolling and Pipeline Scheduling with Static Multiple Issue

No multiple issue (14 cycles)

```
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)
      DADDUI R1,R1,#-32
      S.D    F12,16(R1)
      BNE    R1,R2,Loop
      S.D    F16,8(R1)
```

Two issues:1 FP+1 INT/Mem/Branch (12 cycles/5 loops)

	Integer instruction		FP instruction	Clock cycle
Loop:	L.D	F0,0(R1)		1
	L.D	F6,-8(R1)		2
	L.D	F10,-16(R1)	ADD.D F4,F0,F2	3
	L.D	F14,-24(R1)	ADD.D F8,F6,F2	4
	L.D	F18,-32(R1)	ADD.D F12,F10,F2	5
	S.D	F4,0(R1)	ADD.D F16,F14,F2	6
	S.D	F8,-8(R1)	ADD.D F20,F18,F2	7
	S.D	F12,-16(R1)		8
	DADDUI	R1,R1,#-40		9
	S.D	F16,16(R1)		10
	BNE	R1,R2,Loop		11
	S.D	F20,8(R1)		12

Summary

Method	Original	Schedule	Unroll 4	Unroll 4 + Schedule	Unroll 5 + Schedule + Multiple Issue
# of Instr.	5	5	14	14	17
Cycles/loop	10	6	28/4=7	14/4=3.5	12/5=2.4

Static Branch Prediction for Load Stall

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

almost always taken

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

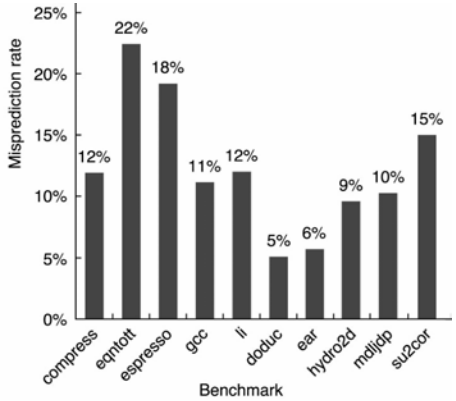
rarely taken

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

assume it's safe if mis-predicted

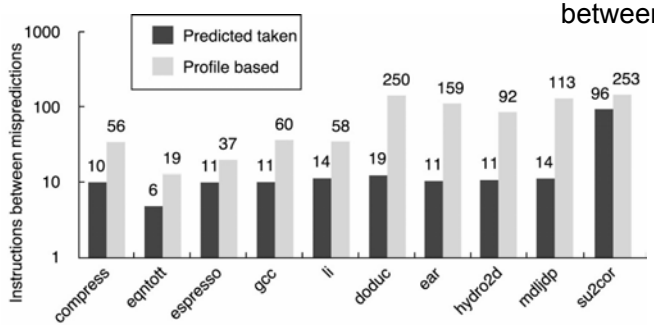
Static Branch Prediction Schemes

- Simplest scheme - predict branch as taken
  - **34% misprediction rate for SPEC programs (59% to 9%)**
- Direction-based scheme - predict backward-going branch as taken and forward-going branch as not taken
  - **Not good for SPEC programs**
  - **Overall misprediction rate is not less than 30% to 40%**
- Profile-based scheme – predict on the basis of profile information collected from earlier runs
  - **An individual branch is often highly biased toward taken or untaken**
  - **Changing the input so that the profile is for a different run leads to only a small change in the accuracy**



Profile-based Static Branch Prediction

Misprediction rate on SPEC92
•varying widely: 3% to 24%
•in average, 9% for FP programs and 15% for integer programs



Number of instructions executed between mispredicted branches

avg.	Taken	Profile
FP	30	173
INT	10	46
All	20	110
std dev	27	85

varying widely: depending on branch frequency and prediction precision

4.3 Static Multiple Issue: The VLIW Approach

Multiple Issue is the ability of the processor to start more than one instruction in a given cycle

Flavor I: Superscalar processors

- in-order issue varying number of instructions per clock (1-8)
- either statically scheduled (by the compiler) or dynamically scheduled (by the hardware, Tomasulo)

Flavor II: Very Long Instruction Word (VLIW)

- parallel issue a fixed number of instructions (4-16)
  - compilers choose instructions to be issued simultaneously
- formatted either as one very large instruction or as a fixed issue packet of smaller instructions
  - rigid in early VLIWs
- dependency checked and instruction scheduled by the compiler
  - simplifying hardware (maybe no or simpler dependency check)
  - complex compiler to uncovered enough parallelism
    - •loop unrolling
    - •local scheduling: operate on a single basic block
    - •global scheduling: may move code across branches
- Style: "Explicitly Parallel Instruction Computer (EPIC)"
  - Intel Architecture-64 (IA-64) 64-bit address
  - Joint HP/Intel agreement in 1999/2000

Unrolled version (x[i]=x[i]+s)

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

Basic VLIW Approach

23 operations in 9 clock cycles (2.5 operations per cycle)
- 9 cycle for 1 iteration for the base

Unroll 7 iterations in 9 cycles (1.29 cycles per loop)
- 2.4 for unrolled 5 and scheduled version

Require at least 8 FP registers for VLIW
- 2 FP registers for the base
- 5 FP registers for unrolled 5 and scheduled version

22/(5*9)=48.9% of functional units are empty

VLIW: 2 mem's, 2 FPs, 1 Int/Brx (ignore branch delayed slot)

	Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
1	L.D F0,0(R1)	L.D F6,-8(R1)			
2	L.D F10,-16(R1)	L.D F14,-24(R1)			
3	L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
4	L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
5			ADD.D F20,F18,F2	ADD.D F24,F22,F2	
6	S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
7	S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
8	S.D F20,24(R1)	S.D F24,16(R1)			
9	S.D F28,8(R1)				BNE R1,R2,Loop

Problems with Basic VLIW Approach

Technical problems

– increase in code size
  - unrolling
  - wasted bits for unused functional units in instruction encoding

  solutions: clever encoding, compress

– limitations of lockstep operation – synchronization restriction
  - To keep all functional units (FU) synchronized, a stall in any FU pipeline must cause the entire processor to stall, i.e. cache stall, exception

  recent solutions: FU operate more independently, hardware checks allow for unsynchronized execution once instructions are issued

Logistical problem – migration problem

– binary code compatibility – recompilation required for different numbers of FUs and unit latencies

  solution: object-code translation or emulation

4.4 Advanced Compiler Support For ILP

How can compilers be smart?
1. Produce good scheduling of code
2. Determine which loops might contain parallelism
3. Eliminate name dependencies

Compilers must be REALLY smart to figure out aliases -- pointers in C are a real problem

Two important ideas:

Software Pipelining - Symbolic Loop Unrolling

Trace Scheduling - Critical Path Scheduling

Loop-Level Dependence and Parallelism

Loop-carried dependence: data accesses in later iterations are dependent on data values produced in earlier iterations

structures analyzed at source level by compilers, such as
– loops
– array references
– induction variable computations

Loop-level parallelism

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

– dependence between two uses of x[i]
– dependent within a single iteration, not loop carried
– A loop is parallel if it can written without a cycle in the dependences, since the absence of a cycle means that the dependences give a partial ordering on the statements

P.320 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 */
}
```

Assume A, B, and C are distinct, non-overlapping arrays

Two different dependences

1.Dependence of S1 is on an earlier iteration of S1

S1: A[i+1] depends on A[i], S2: B[i+1] depends on B[i]

forces successive iterations to execute in series

2.B[i+1] in S2 uses A[i+1] computed by S1 in the same iteration

Multiple iterations of the loop could execute in parallel

P.321 Example

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

Loop-carried dependence between S2 (B[i+1]) and S1 (B[i])

- not circular: neither statement depends on itself
- S1 depends on S2, but S2 does not depend on S1

This loop can be made parallel

Interchanging

- no longer loop carried

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


Array Renaming

```
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 */
}
```

```
for (i=1; i<=100; i=i+1 {
    /* Y renamed to T to remove output dependence */
    T[i] = X[i] / c;
    /* X renamed to X1 to remove antidependence */
    X1[i] = X[i] + c;
    /* Y renamed to T to remove antidependence */
    Z[i] = T[i] + c;
    Y[i] = c - T[i];
}
```

1. True dependences from S1 to S3 and from S1 to S4, based on Y[i]
   - not loop carried
2. antidependence from S1 to S2, based on X[i]
3. antidependence from S3 to S4 for Y[i]
4. output dependence from S1 to S4, based on Y[i]

Register Renaming and Recurrence

Renaming: 2nd reference to A can be a reference to the register

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

Recurrence

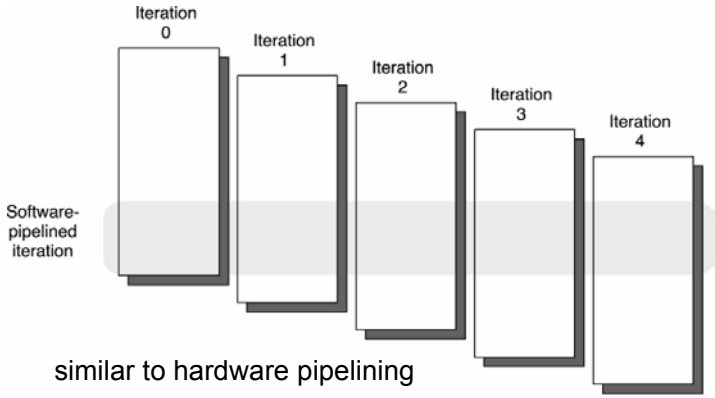
```
for (i=2;i<=100;i=i+1) {
      Y[i] = Y[i-1] + Y[i];
}
```

dependence distance of 5

```
for (i=6;i<=100;i=i+1) {
      Y[i] = Y[i-5] + Y[i];
}
```


Software Pipelining

- Observation: if iterations from loops are independent, then can get ILP by taking instructions from different iterations
- Software pipelining: interleave instructions from different loop iterations
  - reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (Tomasulo in SW)



similar to hardware pipelining

Software Pipelining Example

```
LOOP: 1 L.D     F0,0(R1)
      2 ADD.D   F4,F0,F2
      3 S.D     F4,0(R1)
      4 DADDUI  R1,R1,#8
      5 BNE     R1,R2,LOOP
```

(5/5)

2 RAWs!

```
1 L.D    F0,0(R1)
2 ADD.D  F4,F0,F2   4 L.D    F0,-8(R1)
3 S.D    F4,0(R1)   5 ADD.D  F4,F0,F2   7 L.D    F0,-16(R1)
                    6 S.D    F4,-8(R1)  8 ADD.D  F4,F0,F2
                                        9 S.D    F4,-16(R1)
```

Before: Unrolled 3 times (11/11)

```
1  L.D     F0,0(R1)
2  ADD.D   F4,F0,F2
3  S.D     F4,0(R1)
4  L.D     F0,-8(R1)
5  ADD.D   F4,F0,F2
6  S.D     F4,-8(R1)
7  L.D     F0,-16(R1)
8  ADD.D   F4,F0,F2
9  S.D     F4,-16(R1)
10 DADDUI  R1,R1,#24
11 BNE     R1,R2,LOOP
```

After: Software Pipelined (5/11)

```
1  L.D     F0,0(R1)
2  ADD.D   F4,F0,F2
4  L.D     F0,-8(R1)
3  S.D     F4,0(R1);   Stores M[i]
5  ADD.D   F4,F0,F2;   Adds to M[i-1]
7  L.D     F0,-16(R1); loads M[i-2]
10 DADDUI  R1,R1,#8
11 BNE     R1,R2,LOOP
6  S.D     F4,-8(R1)
8  ADD.D   F4,F0,F2
9  S.D     F4,-16(R1)
```

No RAW! (2 WARs)

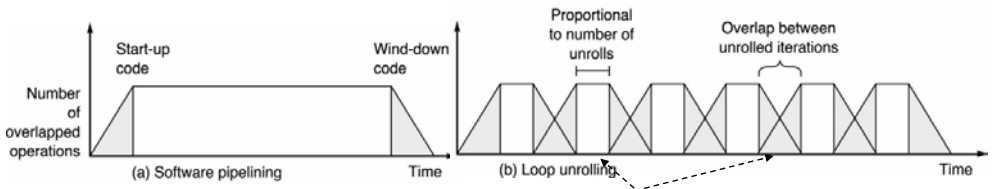
Software Pipelining and Loop Unrolling

- Software pipelining
  - Can be thought of as symbolic loop unrolling
  - May use loop-unrolling to figure out how to pipeline the loop
- Loop unrolling reduces the overhead of the loop
  - the branch and counter update code
  - but every time the inner unrolled loop still need be initiated

Software pipelining reduces the time when the loop is not running at peak speed to once per loop

- major advantage: consumes less code space
- In practice, compilation using software pipelining is quite difficult

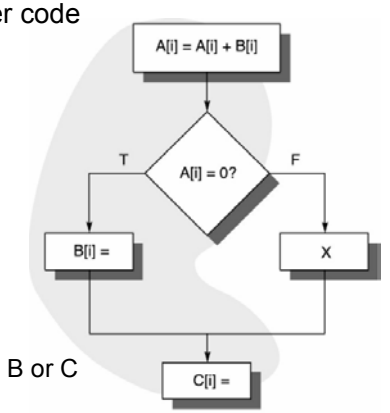
the best performance can come from doing both

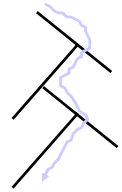


25 loops with 4 unrolled iterations each = 100 iterations

Global Code Scheduling

- Loop unrolling and s/w pipelining mainly work for basic blocks
- Global code scheduling: moving instructions across branches
  - Aims to compact a code fragment with internal control structure into the ***shortest possible sequence*** that preserves the data and control dependences
  - Requires estimates of the relative frequency of different paths
    - shortest possible sequence = shortest sequence for the ***critical path***
  - Can not guarantee faster code





Two tasks
1. Find the common path
2. Move the assignments to B or C

Trace Scheduling – Critical Path Scheduling

Two steps:

- ***Trace Selection***
  - Trace profiling first
  - Find likely sequence of basic blocks (***trace***) of (loop unrolled, statically predicted or profile predicted) long sequence of straight-line code
  - Unwinding frequent path
- ***Trace Compaction***
  - Squeeze trace into few VLIW instructions
  - Need *bookkeeping code* in case prediction is wrong
  - If an instruction can be moved and thereby make the trace execute faster, it is moved

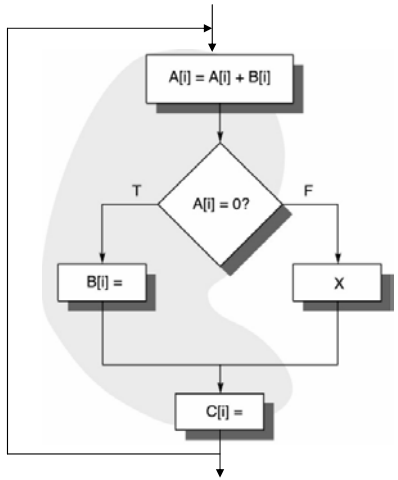
- Compiler undoes bad guess (discards values in registers)
- Subtle compiler bugs mean wrong answer
  vs. poorer performance; no hardware interlocks

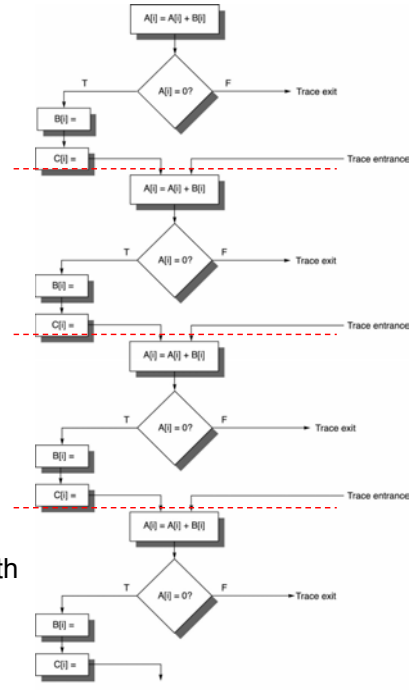
```
A[i] = A[i]+B[i]
if(A[i]==0)
      B[i]=
else
      X
C[i]=
```

⇨

```
A[i] = A[i]+B[i]
B[i]=
C[i]=
if(A[i]!=0) {
      undo
      X
}
```

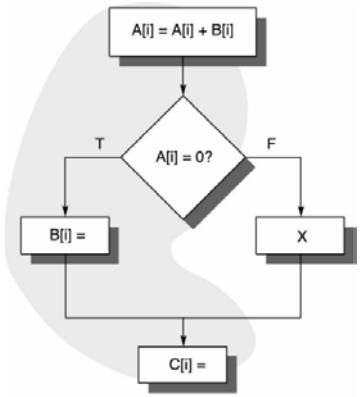
Unwinding Frequent Path



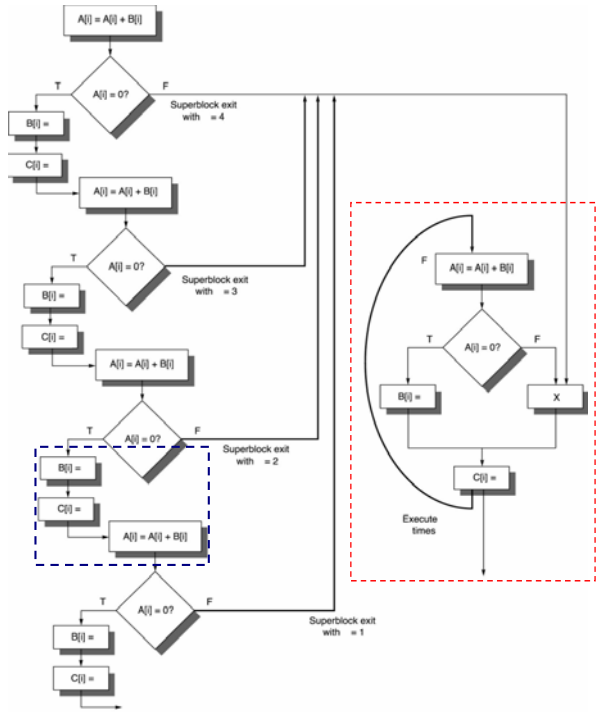


- Assume it's the inner loop and the likely path
- Unwind it four times
- Require the compiler to generate and trace the compensation code

Tail Duplication



Create a separate block that corresponds to the portion of the trace after the entry



Superblocks

- superblock: single entry point but allow multiple exits
  - Compacting a superblock is much easier than compacting a trace
- Tail duplication
  - Create a separate block that corresponds to the portion of the trace after the entry
- Superblock approach vs. trace-based approach
  - Reduce the complexity of bookkeeping and scheduling, but may enlarge code size
  - Both may be most appropriate when other techniques fail

Summary - Software Approaches to Exploiting ILP

- Basic block
  - loop unrolling and basic pipeline scheduling
  - software pipelining
- Trace
  - trace scheduling and unwinding path
- Super block
  - tail duplication

4.7 Intel IA-64 and Itanium Processor

Designed to benefit VLIW approach

IA-64 Register Model

- 128 64-bit GPR (65 bits actually)
- 128 82-bit floating-point registers
  - two extra exponent bits over the standard 80-bit IEEE format
- 64 1-bit predicate register
- 8 64-bit branch registers, used for indirect branches
- a variety of registers used for system control, etc.
- other supports:
  - register stack frame: like register window in SPARC
    - current frame pointer (CFM)
    - register stack engine

Instruction Groups and Bundle

Two concepts to achieve the benefits of implicit parallelism and ease of instruction decode

- **instruction group**: a sequence of consecutive instructions without register data dependences
  - instructions in the group can be executed in parallel
  - arbitrarily long, but the compiler explicitly indicates the boundary by placing a ***stop***
- **bundle**: fixed formatting of multiple instructions (3)
  - IA-64 instructions are encoded in bundles
  - 128 bits wide: 5-bit ***template field*** and three 41-bit instructions
    - the template field describes the presence of stops and specifies types of execution units for each instruction

Five Execution Unit Slots in IA-64

Execution unit slot	Instruction type	Instruction description	Example instructions
I-unit	A	Integer ALU	add, subtract, and, or, compare
	I	Non-ALU integer	integer and multimedia shifts, bit tests, moves
M-unit	A	Integer ALU	add, subtract, and, or, compare
	M	Memory access	Loads and stores for integer/FP registers
F-unit	F	Floating point	Floating-point instructions
B-unit	B	Branches	Conditional branches, calls, loop branches
L + X	L + X	Extended	Extended immediates, stops and no-ops

Figure 4.11 The five execution unit slots in the IA-64 architecture and what instructions types they may hold are shown. A-type instructions, which correspond to integer ALU instructions, may be placed in either an I-unit or M-unit slot. L + X slots are special, as they occupy two instruction slots; L + X instructions are used to encode 64-bit immediates and a few special instructions. L + X instructions are executed either by the I-unit or the B-unit.

24 Possible Template Values and Formats

8 possible values are reserved
Stops are indicated by heavy lines

Execution unit slot	Instruction type	Instruction description	Example instructions
I-unit	A	Integer ALU	add, subtract, and, or, compare
	I	Non-ALU integer	integer and multimedia shifts, bit tests, moves
M-unit	A	Integer ALU	add, subtract, and, or, compare
	M	Memory access	Loads and stores for integer/FP registers
F-unit	F	Floating point	Floating-point instructions
B-unit	B	Branches	Conditional branches, calls, loop branches
L + X	L + X	Extended	Extended immediates, stops and no-ops

Template	Slot 0	Slot 1	Slot 2
0	M	I	I
1	M	I	I
2	M	I	I
3	M	I	I
4	M	L	X
5	M	L	X
8	M	M	I
9	M	M	I
10	M	M	I
11	M	M	I
12	M	F	I
13	M	F	I
14	M	M	F
15	M	M	F
16	M	I	B
17	M	I	B
18	M	B	B
19	M	B	B
22	B	B	B
23	B	B	B
24	M	M	B
25	M	M	B
28	M	F	B
29	M	F	B

Example: Unroll x[i]=x[i]+s Seven Times

Bundle template	Slot 0	Slot 1	Slot 2	Execute cycle (1 bundle/cycle)
9: M M I	L.D F0,0(R1)	L.D F6,-8(R1)		1
14: M M F	L.D F10,-16(R1)	L.D F14,-24(R1)	ADD.D F4,F0,F2	3
15: M M F	L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F8,F6,F2	4
15: M M F	L.D F26,-48(R1)	S.D F4,0(R1)	ADD.D F12,F10,F2	6
15: M M F	S.D F8,-8(R1)	S.D F12,-16(R1)	ADD.D F16,F14,F2	9
15: M M F	S.D F16,-24(R1)		ADD.D F20,F18,F2	12
15: M M F	S.D F20,-32(R1)		ADD.D F24,F22,F2	15
15: M M F	S.D F24,-40(R1)		ADD.D F28,F26,F2	18
12: M M F	S.D F28,-48(R1)	DADDUI R1,R1,#-56	BNE R1,R2,Loop	21

(a) The code scheduled to minimize the number of bundles

9 bundles
21 cycles
85% of slots filled
(23/27)

Bundle template	Slot 0	Slot 1	Slot 2	Execute cycle (1 bundle/cycle)
8: M M I	L.D F0,0(R1)	L.D F6,-8(R1)		1
9: M M I	L.D F10,-16(R1)	L.D F14,-24(R1)		2
14: M M F	L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	3
14: M M F	L.D F26,-48(R1)		ADD.D F8,F6,F2	4
15: M M F			ADD.D F12,F10,F2	5
14: M M F		S.D F4,0(R1)	ADD.D F16,F14,F2	6
14: M M F		S.D F8,-8(R1)	ADD.D F20,F18,F2	7
15: M M F		S.D F12,-16(R1)	ADD.D F24,F22,F2	8
14: M M F		S.D F16,-24(R1)	ADD.D F28,F26,F2	9
9: M M I	S.D F20,-32(R1)	S.D F24,-40(R1)		11
8: M M I	S.D F28,-48(R1)	DADDUI R1,R1,#-56	BNE R1,R2,Loop	12

(b) The code scheduled to minimize the number of cycles assuming one bundle executed per cycle

11 bundles
12 cycles
70% of slots filled
(23/33)

Predication and Speculation Support

Traditional arch. with branch barrier

```
instr 1
instr 2
  :
br
ld r1=…
use …=r1
```

Itanium Support for Explicit Parallelism

```
ld.s r1=…
instr 1
instr 2
  :
br
chk.s r1
use …=r1
```

LOAD moved above branch by compiler

```
ld.s r1=…
instr 1
use …=r1  instr 2
  :
br
chk.s use
```

Uses moved above branch by compiler

Recovery code

```
ld r1=…
use …=r1
br
```


Some Instruction Formats of IA-64

See Textbook or Intel's manuals for more information

Instruction type	Number of formats	Representative instructions	Extra opcode bits	GPRs/ FPRs	Immediate bits	Other/comment
A	8	add, subtract, and, or	9	3	0	
		shift left and add	7	3	0	2-bit shift count
		ALU immediates	9	2	8	
		add immediate	3	2	14	
		add immediate	0	2	22	
		compare	4	2	0	2 predicate register destinations
		compare immediate	3	1	8	2 predicate register destinations
I	29	shift R/L variable	9	3	0	Many multimedia instructions use this format.
		test bit	6	3	6-bit field specifier	2 predicate register destinations
		move to BR	6	1	9-bit branch predict	branch register specifier
M	46	integer/FP load and store, line prefetch	10	2	0	speculative/ nonspeculative
		integer/FP load and store, and line prefetch and post-increment by immediate	9	2	8	speculative/ nonspeculative
		integer/FP load prefetch and register postincrement	10	3		speculative/ nonspeculative
		integer/FP speculation check	3	1	21 in two fields	
B	9	PC-relative branch, counted branch	7	0	21	
		PC-relative call	4	0	21	1 branch register
F	15	FP arithmetic	2	4		
		FP compare	2	2		2 6-bit predicate regs
L+X	4	move immediate long	2	1	64	

Summary

Chapter 4 Exploiting Instruction-Level Parallelism with Software Approaches

4.1 Basic Compiler Techniques for Exposing ILP

4.2 Static Branch Prediction

4.3 Static Multiple Issue: The VLIW Approach

4.4 Advanced Compiler Support for ILP

4.7 Intel IA-64 Architecture and Itanium Processor