


MK
MORGAN KAUFMANN

Computer Architecture
A Quantitative Approach, Fifth Edition



Chapter 3

**Instruction-Level Parallelism
and Its Exploitation**

MK
MORGAN KAUFMANN

Copyright © 2012, Elsevier Inc. All rights reserved.

1

Introduction

- Pipelining become universal technique in 1985
 - Overlaps execution of instructions
 - Exploits “Instruction Level Parallelism”
- Beyond this, there are two main approaches:
 - Hardware-based dynamic approaches
 - Used in server and desktop processors
 - Not used as extensively in PMP processors
 - Compiler-based static approaches
 - Not as successful outside of scientific applications

MK
MORGAN KAUFMANN

Copyright © 2012, Elsevier Inc. All rights reserved.

2

Introduction

Instruction-Level Parallelism

Introduction

- When exploiting instruction-level parallelism, goal is to maximize CPI
 - Pipeline CPI =
 - Ideal pipeline CPI +
 - Structural stalls +
 - Data hazard stalls +
 - Control stalls
- Parallelism with basic block is limited
 - Typical size of basic block = 3-6 instructions
 - Must optimize across branches



Copyright © 2012, Elsevier Inc. All rights reserved.

3

Data Dependence

Introduction

- Loop-Level Parallelism
 - Unroll loop statically or dynamically
 - Use SIMD (vector processors and GPUs)
- Challenges:
 - Data dependency
 - Instruction j is data dependent on instruction i if
 - Instruction i produces a result that may be used by instruction j
 - Instruction j is data dependent on instruction k and instruction k is data dependent on instruction i
- Dependent instructions cannot be executed simultaneously



Copyright © 2012, Elsevier Inc. All rights reserved.

4


Introduction

Data Dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it causes a stall


- Data dependence conveys:
 - Possibility of a hazard
 - Order in which results must be calculated
 - Upper bound on exploitable instruction level parallelism

- Dependencies that flow through memory locations are difficult to detect


Copyright © 2012, Elsevier Inc. All rights reserved.
5

Data Dependence

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


Copyright © 2012, Elsevier Inc. All rights reserved.
6

Name Dependence

Introduction

- Two instructions use the same name but no flow of information
 - Not a true data dependence, *but is a problem when reordering instructions*
 - *Antidependence*: instruction j writes a register or memory location that instruction i reads
 - Initial ordering (i before j) must be preserved
 - *Output dependence*: instruction i and instruction j write the same register or memory location
 - Ordering must be preserved
- To resolve, use renaming techniques



Copyright © 2012, Elsevier Inc. All rights reserved.

7

Other Factors

Introduction

- Data Hazards
 - Read after write (RAW)
 - Write after write (WAW)
 - Write after read (WAR)
- Control Dependence
 - Ordering of instruction i with respect to a branch instruction
 - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
 - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch




Copyright © 2012, Elsevier Inc. All rights reserved.

8

Control Dependence

- Must preserve exception behavior.
- We should not change the exception behavior of the program.
- We often relax this to “reordering of instruction must not raise new exceptions”

| | |
|--------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> ■ DADDU R2,R3,R4 ■ BEQZ R2,L1 ■ LW R1,0(R2) ■ L1: | <ul style="list-style-type: none"> ■ No data dependence prevents us from exchanging BEQZ and LW, but might result in memory protection exception |
|--------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|



Copyright © 2012, Elsevier Inc. All rights reserved.

9

Introduction

Examples

- Example 1:


| | | |
|--------|----------|--|
| DADDU | R1,R2,R3 | |
| BEQZ | R4,L | |
| DSUBU | R1,R1,R6 | |
| L: ... | | |
| OR | R7,R1,R8 | |

 - OR instruction dependent on DADDU and DSUBU
 - Preserving the order alone is not sufficient (must have the correct value in R1)

- Example 2:

| | | |
|-------|----------|--|
| DADDU | R1,R2,R3 | |
| BEQZ | R12,skip | |
| DSUBU | R4,R5,R6 | |
| DADDU | R5,R4,R9 | |
| skip: | | |
| OR | R7,R8,R9 | |

 - Assume R4 isn't used after skip
 - Possible to move DSUBU before the branch



Copyright © 2012, Elsevier Inc. All rights reserved.

10

Compiler Techniques

Compiler Techniques for Exposing ILP

- Pipeline scheduling
 - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction
- Example:


```

                for (i=999; i>=0; i=i-1)
                    x[i] = x[i] + s;
            
```

No dependence between iterations
MIPS code?

| 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 |

Copyright © 2012, Elsevier Inc. All rights reserved.

11

Compiler Techniques

Pipeline Stalls

```

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 (assume integer load latency is 1) 8
      BNE   R1,R2,Loop          9
    
```

| 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 |

Copyright © 2012, Elsevier Inc. All rights reserved.

12

Pipeline Scheduling


Compiler Techniques

Scheduled code:

```

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

| 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 |



Copyright © 2012, Elsevier Inc. All rights reserved.

13

Loop Unrolling

Compiler Techniques

- Loop unrolling
 - Unroll by a factor of 4 (assume # elements is divisible by 4)
 - Eliminate unnecessary instructions


```

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
    
```

1 stall (pointing to the first ADD.D instruction)

2 stalls (pointing to the first and second S.D instructions)

- note: number of live registers vs. original loop



Copyright © 2012, Elsevier Inc. All rights reserved.

14

Loop Unrolling/Pipeline Scheduling

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

Loop iterations are independent



Strip Mining

- Unknown number of loop iterations?
 - Number of iterations = n
 - Goal: make k copies of the loop body
 - Generate pair of loops:
 - First executes $n \bmod k$ times
 - Second executes n / k times
 - “Strip mining”



Loop Level Parallelism

- Loop-Level Parallelism (LLP) analysis focuses on whether data accesses in later iterations of a loop are data dependent on data values produced in earlier iterations and possibly making loop iterations independent.

```
For(i=0;i<100;i++)  
  x[i]=x[i]+A;
```

- the computation in each iteration is independent of the previous iterations and the loop is thus parallel. The use of `x[i]` twice is within a single iteration.
 - ⇒ Thus loop iterations are parallel (or independent from each other).



Copyright © 2012, Elsevier Inc. All rights reserved.

17

Loop Level Parallelism

- **Loop-carried Dependence:** A data dependence between different loop iterations (data produced in earlier iteration used in a later one).
- LLP analysis is important in software optimizations such as loop unrolling since it usually requires loop iterations to be independent.
- LLP analysis is normally done at the source code level or close to it since assembly language and target machine code generation introduces loop-carried name dependence in the registers used for addressing and incrementing.
- Instruction level parallelism (ILP) analysis, on the other hand, is usually done when instructions are generated by the compiler



Copyright © 2012, Elsevier Inc. All rights reserved.

18

Loop Level Parallelism

```

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 */
}

```

Dependency Graph

Iteration # → i i+1

Not Loop Carried Dependence

Loop-carried Dependence

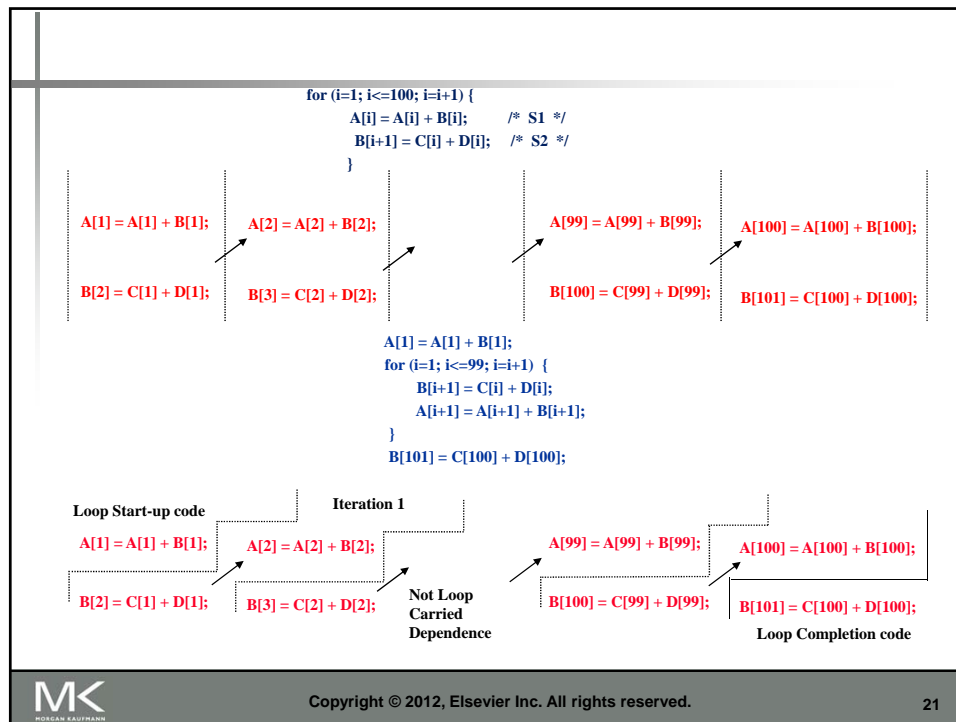
- S2 uses the value $A[i+1]$, computed by S1 in the same iteration. This data dependence is within the same iteration (not a loop-carried dependence).
 - ⇒ does not prevent loop iteration parallelism.
- S1 uses a value computed by S1 in an earlier iteration, since iteration i computes $A[i+1]$ read in iteration $i+1$ (loop-carried dependence, prevents parallelism). The same applies for S2 for $B[i]$ and $B[i+1]$
 - ⇒ These two dependencies are loop-carried spanning more than one iteration preventing loop parallelism.

Copyright © 2012, Elsevier Inc. All rights reserved.
19

Loop Level parallelism

- `for(i=0; i<=100; i++)`
- `A[i] = A[i] + B[i]; /* S1 */`
- `B[i+1] = C[i] + D[i]; /* S2 */`
- S1 uses the value calculated by S2 in the previous iteration (loop carried dependence)
- The dependence is not circular, S2 does not depend on S1 in the previous iteration

Copyright © 2012, Elsevier Inc. All rights reserved.
20



Finding Dependence

- Finding dependences in the program is very important for renaming and executing instructions in parallel.
- Arrays and pointers makes finding dependences very difficult.
- Assume array indices are *affine*, which means on the form $a_i + b_j + c_k$ where a and c are constant.
- GCD test can be used to detect dependences.

Finding Dependence

- Assume we stored an array with index value of $ai+b$ and loaded an array with an index value of $cj+d$
- Are they pointing to the same location?
- Assume the loop limit is m, n
- Are there

$$j, k \quad m \leq j, k \leq n \text{ such that } a \times j + b = c \times k + d$$



Copyright © 2012, Elsevier Inc. All rights reserved.

23

GCD test

- A simple and sufficient test for absence can be found.
- If a loop dependence exists, then

$$GCD(c, a) \text{ divides } (d - b)$$



Copyright © 2012, Elsevier Inc. All rights reserved.

24

GCD Test -- Example

```
for(i=1; i<=100; i=i+1) {
    x[2*i+3] = x[2*i] * 5.0;
}
```

a = 2 b = 3 c = 2 d = 0

GCD(a, c) = 2

d - b = -3

2 does not divide -3 \Rightarrow No dependence
is not possible.

5,7,9,11,13,15,17,19,21,23,....

4,6,8,10,12,14,16,18,20,22,....



Copyright © 2012, Elsevier Inc. All rights reserved.

25

Dependence Analysis

- Dependence analysis is a very important tool for exploiting LLP, it can not be used in these situations
- Objects are referenced using pointers
- Array indexing using another array $a[b[i]]$
- Dependence may exist for some values of input, but in reality the input never takes these values.
- When we want to more than the possibility of dependence (which write causes it?)
- Dependence analysis across procedure boundaries



Copyright © 2012, Elsevier Inc. All rights reserved.

26

Dependence Analysis

- Sometimes, *points-to* analysis might help.
- We might be able to answer *simpler* questions, or get some hints.
- Do 2 pointers point to the same list?
- Type information
- Information derived when the object was allocated
- Pointer assignments

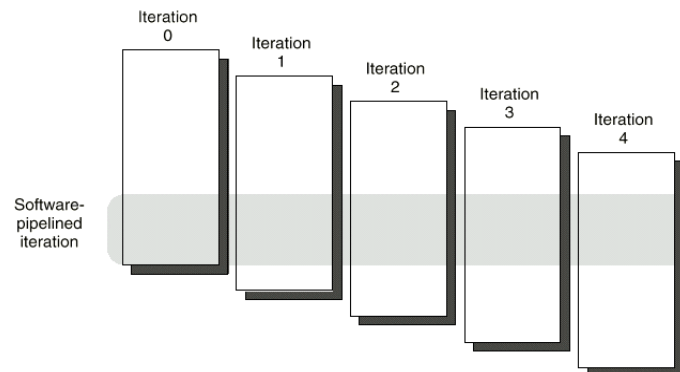


Copyright © 2012, Elsevier Inc. All rights reserved.

27

Software Pipelines

- Software pipelined loop chooses instructions from different loop iterations, thus separating the dependent instructions within one iteration of the original loop



Copyright © 2012, Elsevier Inc. All rights reserved.

28

Software Pipelines

```

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

Before: Unrolled 3 times


```

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 Version

```

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




Copyright © 2012, Elsevier Inc. All rights reserved.

29

Software Pipelines

4 Software Pipelined loop iterations (2 iterations fewer)

Loop Body of software Pipelined Version



Copyright © 2012, Elsevier Inc. All rights reserved.

30

Branch Prediction

- Dynamic scheduling deals with data dependence improving, the limiting factor is the control dependence.
- Branch prediction is important for processors that maintains a CPI of 1, but it is crucial for processors who tries to issue more than one instruction per cycle ($CPI < 1$).
- We have already studied some techniques (delayed branch, predict not taken), but these do not depend on the dynamic behavior of the code.



Copyright © 2012, Elsevier Inc. All rights reserved.

31

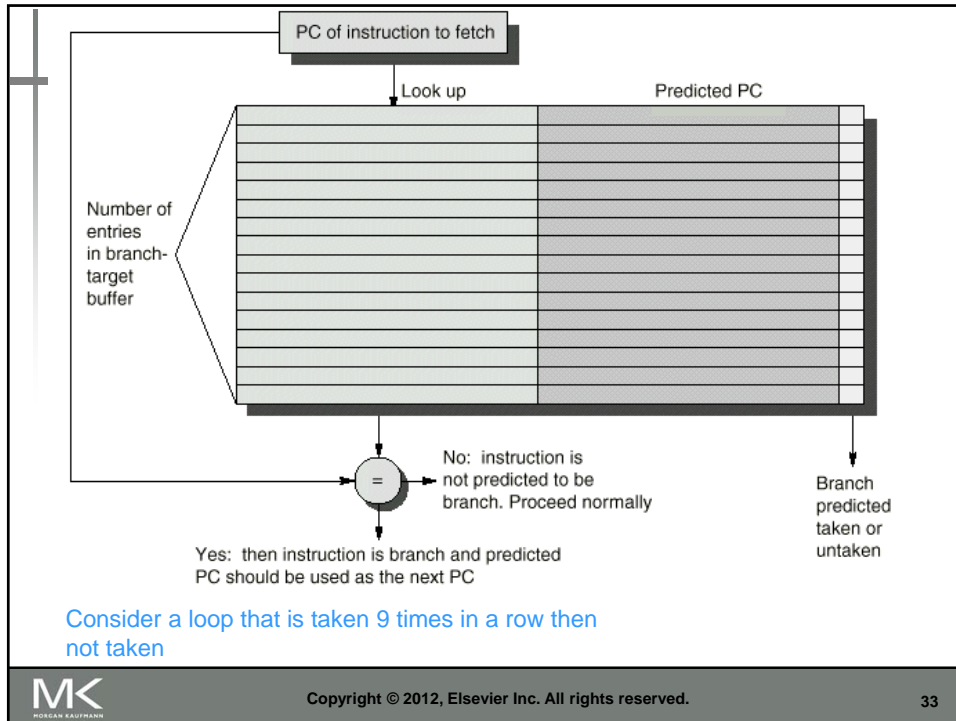
Branch History Table

- A small memory indexed by the lower portion of the address of the branch instruction.
- The memory contains only 1-bit, to predict taken or untaken
- If the prediction is incorrect, the prediction bit is inverted.
- In a loop, it mispredicts twice
 - End of loop case, when it exits instead of looping as before
 - First time through loop on *next* time through code, when it predicts exit instead of looping



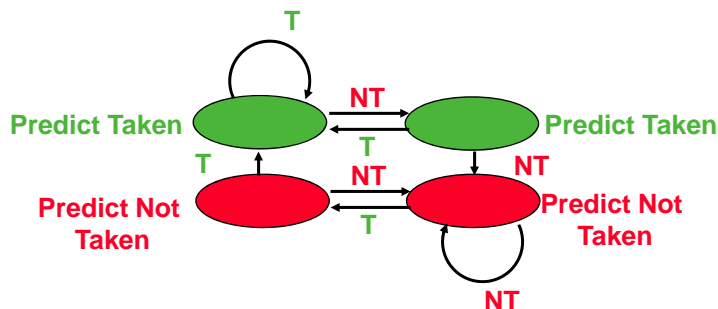
Copyright © 2012, Elsevier Inc. All rights reserved.

32



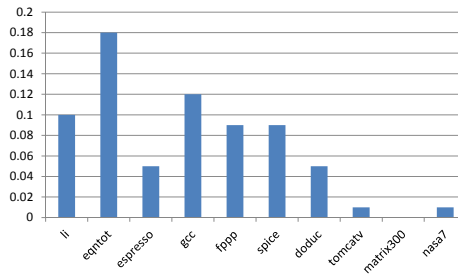
2-Bit Predictor

- Uses 2 bits to add some hysteresis to the prediction – Compare with 1 bit?
- 2 bits are as good as N bits (approx.)



2-bit Predictor

- 4096 entries 2-bit predictor miss rate



Copyright © 2012, Elsevier Inc. All rights reserved.

35

Correlating Branch Predictors

| | | | |
|----|--------------|-------|------------------------------|
| | | | DSUBUI R3, R1, #2 |
| B1 | if (aa==2) | → | BNEZ R3, L1 ; b1 (aa!=2) |
| | aa=0; | | DADD R1, R0, R0 ; aa==0 |
| B2 | if (bb==2) | → L1: | DSUBUI R3, R1, #2 |
| | bb=0; | | BNEZ R3, L2 ; b2 (bb!=2) |
| | | | DADD R2, R0, R0 ; bb==0 |
| B3 | if (aa!=bb){ | → L2: | DSUBUI R3, R1, R2 ; R3=aa-bb |
| | | | BEQZ R3, L3 ; b3 (aa==bb) |

If the condition is true → B1,B2 branch NOT TAKEN

If the condition is true → B3 NOT taken

If B1 and B2 both NOT TAKEN B3 → TAKEN

There is a correlation between B3 and both B1 and B2



Copyright © 2012, Elsevier Inc. All rights reserved.

36

Correlating Branch Predictors

- Correlating predictors (rw0-level predictors) use the behavior of other branches to make prediction.
- Simplest (1-bit) has 2 predictions, one if the last branch is take, the second is when the last branch is not taken
- The prediction is on the form **NT/T**



Copyright © 2012, Elsevier Inc. All rights reserved.

37

Example

```

B1   if (d==0)
        d=1;
B2   if (d==1)
        {

```

```

        BNEZ   R1, L1    ; d == 0 ?
        DADD   R1, R0, #1 ; YES d==1
L1:    DADD   R3, R1, #-1
        BNEZ   R3, L2    ; b2 (bb!=2)
L2:

```

If b1 not taken, b2 is taken for sure

| Initial d | d==0? | B1 | d before b2 | d==1 | b2 |
|-----------|-------|-------|-------------|------|-------|
| 0 | Y | NO | 1 | Y | NO |
| 1 | N | Taken | 1 | Y | NO |
| 2 | N | Taken | 2 | N | Taken |



Copyright © 2012, Elsevier Inc. All rights reserved.


38

Example

| Initial d | d==0? | B1 | d before b2 | d==1 | B2 |
|-----------|-------|-------|-------------|------|-------|
| 0 | Y | NO | 1 | Y | NO |
| 1 | N | Taken | 1 | Y | NO |
| 2 | N | Taken | 2 | N | Taken |

| d | B1 Pred | B1 action | newB1 pred | B2 pred | B2 action | new B2 pred |
|---|---------|-----------|------------|---------|-----------|-------------|
| 2 | NT | T | T | NT | T | T |
| 0 | T | NT | NT | T | NT | NT |
| 2 | NT | T | T | NT | T | T |
| 0 | T | NT | NT | T | NT | NT |

Miss on every prediction



Copyright © 2012, Elsevier Inc. All rights reserved.
39

Example

| Initial d | d==0? | B1 | d before b2 | d==1 | b2 |
|-----------|-------|-------|-------------|------|-------|
| 0 | Y | NO | 1 | Y | NO |
| 1 | N | Taken | 1 | Y | NO |
| 2 | N | Taken | 2 | N | Taken |

| d | b1 Pred | b1 action | newb1 pred | b2 pred | b2 action | new b2 pred | | |
|---|---------|-----------|------------|---------|-----------|-------------|------|------|
| 2 | NT/NT | ✗ | T | NT/NT | ✗ | T | NT/T | |
| 0 | T/NT | ✓ | NT | T/NT | NT/T | ✓ | NT | NT/T |
| 2 | T/NT | ✓ | T | T/NT | NT/T | ✓ | T | NT/T |
| 0 | T/NT | ✓ | NT | T/NT | NT/T | ✓ | NT | NT/T |

Misprediction on first try only


Copyright © 2012, Elsevier Inc. All rights reserved.
40

Correlating Predictors

- The previous predictor is called (1,1) predictor.
- It uses one bit for history (last branch), to choose among two (2^1) 1-bit branch predictors.
- In general a predictor could be (m,n) predictor.
- It uses the last m branch to choose among 2^m branch predictors each is n -bit predictor.



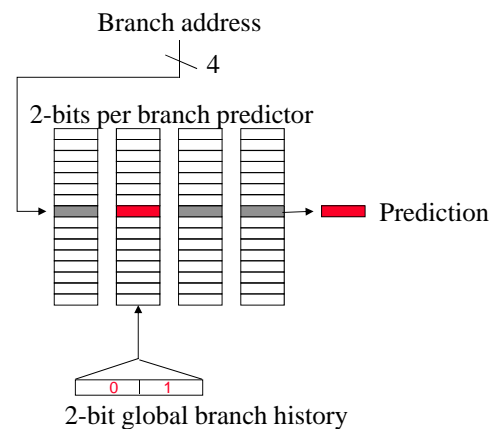
Copyright © 2012, Elsevier Inc. All rights reserved.

41

(2,2) Correlating Predictors

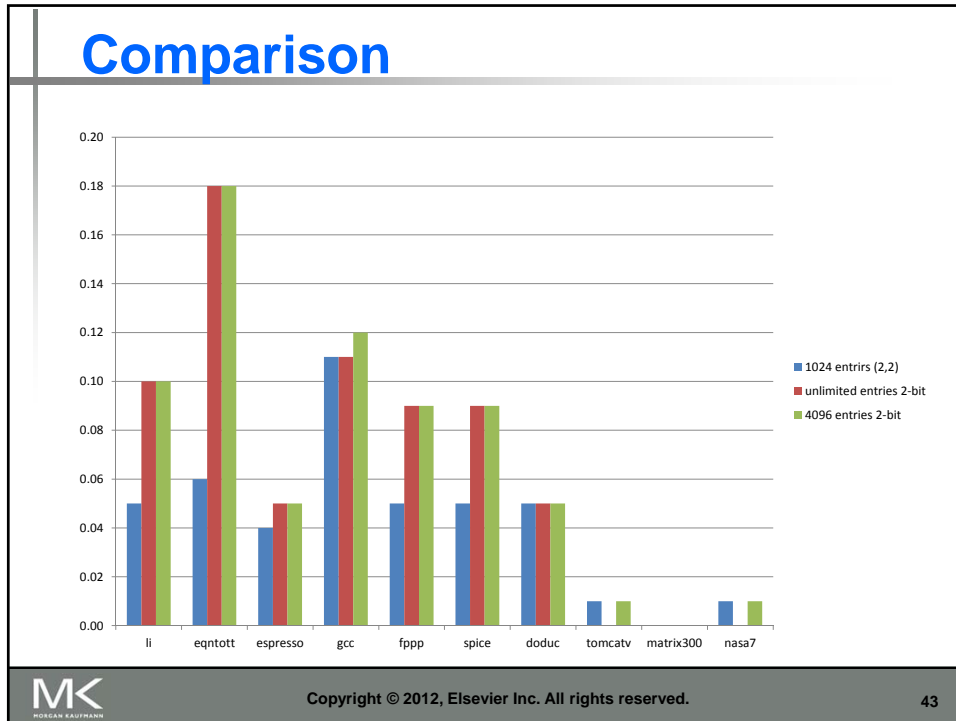
(2,2) predictor

- Behavior of recent branches selects between four predictions of next branch, updating just that prediction



Copyright © 2012, Elsevier Inc. All rights reserved.

42



Branch Prediction

Branch Prediction

- Basic 2-bit predictor:
 - For each branch:
 - Predict taken or not taken
 - If the prediction is wrong two consecutive times, change prediction
- Correlating predictor:
 - Multiple 2-bit predictors for each branch
 - One for each possible combination of outcomes of preceding n branches
- Local predictor:
 - Multiple 2-bit predictors for each branch
 - One for each possible combination of outcomes for the last n occurrences of this branch


MK
MORGAN KAUFMANN

Copyright © 2012, Elsevier Inc. All rights reserved.

44

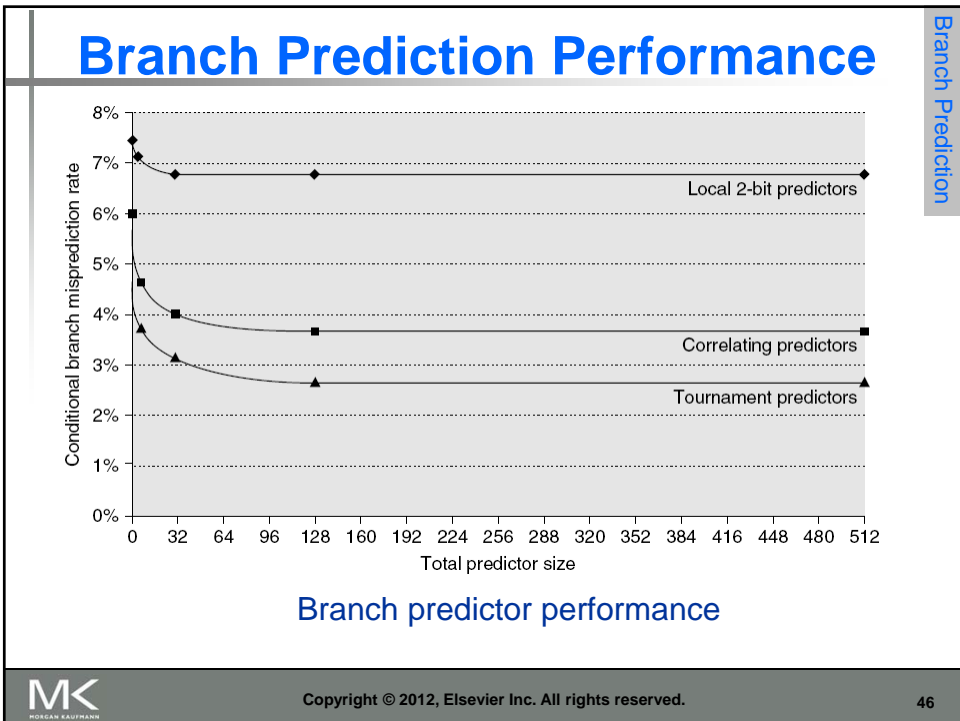
Tournament Predictor

- Tournament predictor:
 - Combine correlating predictor with local predictor
 - A selector is used to decide which one of these to use
- The selector could be similar to a 2-bit predictor
 - A saturating 2-bit binary counter with 2 outcomes P1/P2



Copyright © 2012, Elsevier Inc. All rights reserved.

45



Alpha 21264 Branch Predictor

- Tournament predictor using, 4K 2-bit counters indexed by local branch address.
- Global predictor
 - 4K entries index by history of last 12 branches ($2^{12} = 4K$)
 - Each entry is a standard 2-bit predictor
- Local predictor
 - Local history table: 1024 10-bit entries recording last 10 branches, index by branch address
 - The pattern of the last 10 occurrences of that **particular** branch used to index table of 1K entries with 3-bit saturating counters



Copyright © 2012, Elsevier Inc. All rights reserved.

47

Intel Core i7 Branch Predictor



Copyright © 2012, Elsevier Inc. All rights reserved.

48

Dynamic Scheduling

Branch Prediction

- Rearrange order of instructions to reduce stalls while maintaining data flow
- Advantages:
 - Compiler doesn't need to have knowledge of microarchitecture
 - Handles cases where dependencies are unknown at compile time
- Disadvantage:
 - Substantial increase in hardware complexity
 - Complicates exceptions



Copyright © 2012, Elsevier Inc. All rights reserved.

49

Dynamic Scheduling

Branch Prediction

- Dynamic scheduling implies:
 - Out-of-order execution
 - Out-of-order completion
- Creates the possibility for WAR and WAW hazards
- Tomasulo's Approach
 - Tracks when operands are available
 - Introduces register renaming in hardware
 - Minimizes WAW and WAR hazards



Copyright © 2012, Elsevier Inc. All rights reserved.

50

Branch Prediction

Register Renaming

- Example:

```

DIV.D F0,F2,F4
ADD.D F6,F0,F8
S.D F6,0(R1)
SUB.D F8,F10,F14
MUL.D F6,F10,F8
    
```

+ name dependence with F6

Copyright © 2012, Elsevier Inc. All rights reserved.
51

Branch Prediction

Register Renaming

- Example:

```

DIV.D F0,F2,F4
ADD.D S,F0,F8
S.D S,0(R1)
SUB.D T,F10,F14
MUL.D F6,F10,T
    
```

- Now only RAW hazards remain, which can be strictly ordered

Copyright © 2012, Elsevier Inc. All rights reserved.
52

Register Renaming

Branch Prediction

- Register renaming is provided by reservation stations (RS)
 - Contains:
 - The instruction
 - Buffered operand values (when available)
 - Reservation station number of instruction providing the operand values
 - RS fetches and buffers an operand as soon as it becomes available (not necessarily involving register file)
 - Pending instructions designate the RS to which they will send their output
 - Result values broadcast on a result bus, called the common data bus (CDB)
 - Only the last output updates the register file
 - As instructions are issued, the register specifiers are renamed with the reservation station
 - May be more reservation stations than registers



Copyright © 2012, Elsevier Inc. All rights reserved.

53