ILP Basics & Branch Prediction

Today's topics:

Compiler hazard mitigation

- loop unrolling
- SW pipelining

Branch Prediction

ILP

- Parallelism → independent enough
  - e.g. avoid stalls
    - » control – correctly predict decision
      - or use branch delay slots via proper scheduling
    - » data – forwarding or instruction scheduling
    - » structural – duplicate resources
      - or avoid conflict via scheduling
  - hmm – scheduling looks like the key
- What schedules?
  - compiler
    - » knows pipeline and latencies
    - » and source code
      - note: programmers can help by writing clean code
    - » can't know some run time status however
      - e.g. how data dependent conditions resolve
  - HW
    - » needs to pitch in where the compiler can't

Basic Block Problems

- **Avg. dynamic branch frequency = 15%-25%**
  - **→ branch every 3-6 instructions**
    - » **ILP is going to be hard to find**
- **Focus on loops**
  - **major part of the execution time in the common case**
    - » **Amdahl's shouts in our ears here**

Pipeline CPI = Ideal Pipeline CPI + Struct. Stalls + RAW Stalls + WAR Stalls + WAW Stalls + Control Stalls

- » **optimize this equation over loops**

- **Some loops are easy (basically vector ops)**

```
for (i=1, i<=1000, i++)      loop:    L.D       F0, 0(R1)      ;R1 array ptr
       x[i] = x[i] + s                ADD.D     F4, F0, F2     ;F2 = s
                                      S.D       0(R1), F4      ;put result back
                                                               ;
                                      DADDUI    R1, R1, #-8    decr. R1
                                      BNE       R1, R2, loop   ;again if not done
```

- » **known vector size & immediate value known to compiler**
- » **10 cycles: L.D, RAW stall, ADD.D, 2 RAW stalls, S.D, DADDUI, RAW stall, BNE, branch delay stall**

School of Computing
University of Utah 3 **CS6810**

Smarter Schedule

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

→

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

Note key issue – S.D → L.D potential for RAW stall if target is same memory addr. e.g. disambiguation problem.
No intervening R1 mod and 0!=8 so no problem

- **6 cycles but still 1 stall**
  - **need larger loop body in order to have a chance**
  - **consider**
    - » **DADDUI, BNE are loop overhead – 40% of total**
    - » **rest are the actual work**

School of Computing
University of Utah 4 **CS6810**

Loop Unrolling → Bigger Basic Block

- **Basic idea**
  - take n loop bodies and catentate them
    - » can't use the same target registers or Wax stalls are a problem
      - increased register pressure limits value of n
    - » adjust termination code
    - » adjust offset values
      - only possible if value is immediate or a known constant in a register
- **Next idea**
  - schedule instructions to avoid existing stalls
    - » a common case is shuffle rather than catenate

4x Unroll

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

Simple Unroll: 12 work instructions, 2 overhead instructions
How many cycles per loop?

5x Unroll

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

$\longrightarrow$

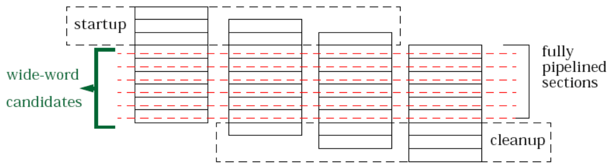
```
Loop:   L.D      F0, 0(R1)
        L.D      F6, -8(R1)
        L.D      F10,-16(R1)
        L.D      F14, -24(R1)
        ADD.D    F4, F0, F2
        ADD.D    F8, F6, F2
        ADD.D    F12, F10, F2
        ADD.D    F16, F14, F2
        S.D      F4, 0(R1)
        S.D      F8, -8(R1)
        DADDUI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

Simple Unroll
How many cycles per loop?
4x(1 post L.D. stall + 2 post ADD.D stalls) = 12
+ 1 post DADDUI and 1 post BNE stall 14 total
+ 14 instructions ➔ crap still only 50% efficient

Schedule – mostly a shuffle
no stalls 14 instructions & 4 loops
3.5 cycles per iteration =
2.857x speedup over 10 cycle original loop
1.7x speedup over scheduled unrolled loop

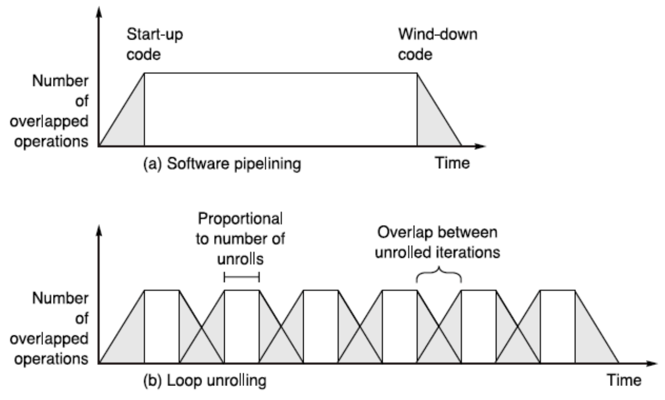
Software Pipelining

- **Similar to loop unrolling but shuffle first**
  - **often referred to as symbolic loop unrolling**



  - **register/name management can be tricky**
    - **» but same idea – create a single loop body**
  - **add this to an already unrolled loop**

Benefit Idea


Start-up code
Wind-down code
Number of overlapped operations
(a) Software pipelining
Time
Proportional to number of unrolls
Overlap between unrolled iterations
Number of overlapped operations
(b) Loop unrolling
Time

Dependency Tactic Synopsis

- **Consider when scheduling and unrolling**
  - **data/RAW**
    - » **unrolling can provide more independent instructions**
      - **up to register availability limit**
    - » **schedule to remove RAW stalls**
  - **name/Wax**
    - » **rename to use different target registers**
    - » **removes WAx stalls**
  - **control**
    - » **the tricky part: scheduling across branches**
      - **simple in this example since there were no loop carried dependencies**
      - **easy when iteration count and offset values are known constants**
    - » **much harder when things aren't vector ops**

Control Dependence Worries

- Conditional branches
  - instructions before the branch are "uncontrolled"
  - instructions after the branch are "controlled"
- Scheduling constraints
  - must preserve controlled and uncontrolled nature of the original instructions
  - note: control over multiple branches is transitive
- Simple in-order pipelines
  - instruction order is preserved
    - » so compiler can handle the schedule
      - except for branch direction and memory latency uncertainties
  - out of order completion of EX stage introduces complexity
    - » increased book-keeping either by the compiler/HW or both

Loop Carried Dependence

- Consider

```
for (i=1; i<=1000; i++) {
        A[i+1] = A[i] + C[i];           /* S1*/
        B[i+1] = B[i] + A[i+1];}        /* S2*/
```

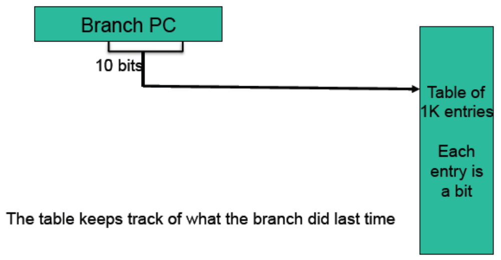
  - S1 depends on an earlier instance of S1
    - » same with S2
    - » ➔ now order matters unlike the vector-scalar add example
- In general there are lots of loop carried dependencies
  - large variety of types
    - » some have work arounds and some don't
    - » save these issues for a bit later
      - since branches come into play
- Hence – take on branch prediction next
  - filling the branch delay slots helps but correct prediction is even better
    - » speculation

Branch Prediction

- **Simple Idea**
  - **let history predict the future**
  - **can be simple – Baskett bit idea**
    - » **arbitrarily complex if you want to be accurate**
- **Static prediction**
  - **compiler can help**
    - » **predict taken (loop bias) has 34% error for SPEC**
      - **wide range depending on app however**
    - » **profile code to get better probability**
      - **average mispredict improves to 9%**
      - **good enough?**
        - **given the penalty for blowing it – probably not**
      - **actual mispredict varies from 5% - 22% for the SPEC benchmarks**
      - **REMEMBER – benchmarks are not real apps**
        - **so reality is likely worse**
- **Enter dynamic prediction**
  - **track actual history in the HW and use as a prediction base**

Baskett Bit Expanded

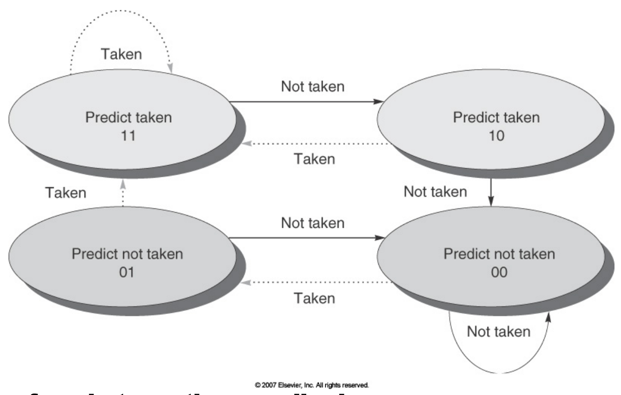
- **Bimodal 1-bit entry in BHT (Branch History Table)**



- **Problem**
  - **for loops – 2 mispredictions per loop**
    - » **exit is always a surprise**
    - » **unless loop count is static**
      - **common in DSP's so HW exists for this**
      - **not common enough in GP CPU's so need something better**
  - **high order bit alias problem (how likely is the problem?)**
  - **how many bits above and below the 10 shown?**

2 Bit Predictor

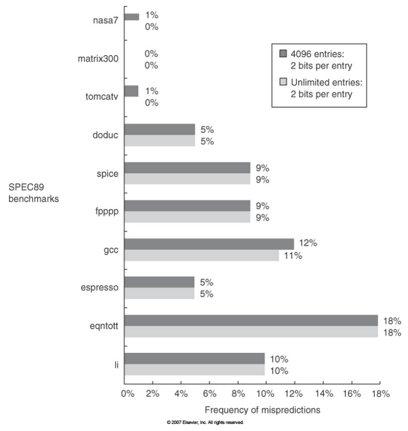
- **Saturating counter**



» **allows bias for whatever the compiler knows**
- **loops vs. others – how does the compiler tell the HW?**
- **which state should a loop branch start in?**
- **what's wrong with this state machine? why is it reasonable?**

Is Bigger Better

- **2 options**
  - **more than 2 bit predictor**
    » **studies show that this isn't a win**
  - **more entries in the BHT**
    » **4K good enough for SPEC89**
    » **a bit more needed for real codes or more modern benchmarks**
    - **lower instruction locality**
    » **bigger BHT**
    - **reduces alias problem**
    » **experiment to find the sweet spot**
- **Note**
  - **integer codes are a bigger problem**
  - **reality is even a bit worse than BM's**



Correlating Predictors

- As ILP and Issue width goes up
  - need to predict over multiple branches
    - » trace scheduling and trace caches come into play
- Fortunately branches exist in a context
  - e.g.

```
if (aa==2) aa=0;
if (bb==2) bb=0;
if (aa!-bb) { … }
```

    - » if first 2 fail then 3rd will be taken
      - dumb code for sure but simple example of correlation
    - » non-correlating predictor will never capture this behavior
- 2-level correlating predictors
  - take global information
    - » what happened over some previous set of branches
      - if set has m members then it's an m-bit vector
      - HW is a simple shift register
  - (m,n) predictor
    - » m bits of global, and n-bit predictor

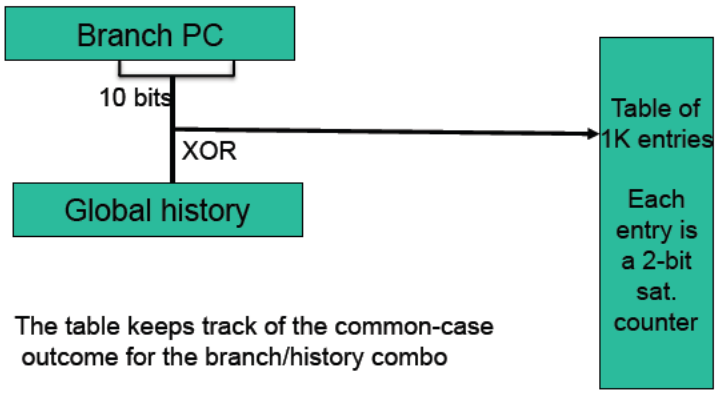
(m,n) Predictor Problem

- Assume
  - m=10, n=2
  - and branch ID is 10 bits
- If we use all 20 bits
  - need a 4M x 2-bit = 1MB BHT
  - TOO EXPENSIVE
- What should we do?

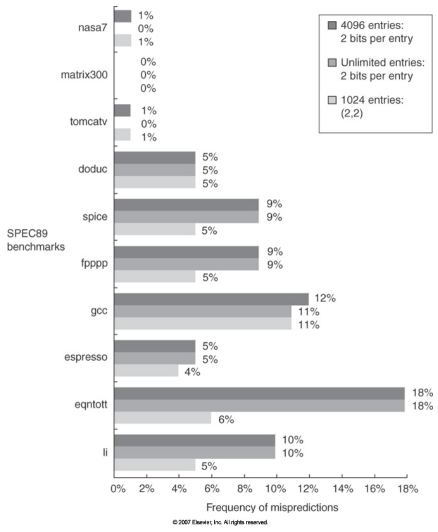
(m,n) Predictor Problem

- **Assume**
  - **m=10, n=2**
  - **and branch ID is 10 bits**
- **If we use all 20 bits**
  - **need a 4M x 2-bit = 1MB BHT**
  - **TOO EXPENSIVE**
- **What should we do?**
  - **hash the 20 bits into something smaller**
  - **XOR is a good hash function**
    - » **cheap and fast**

(10,2) Global Predictor (Gshare)

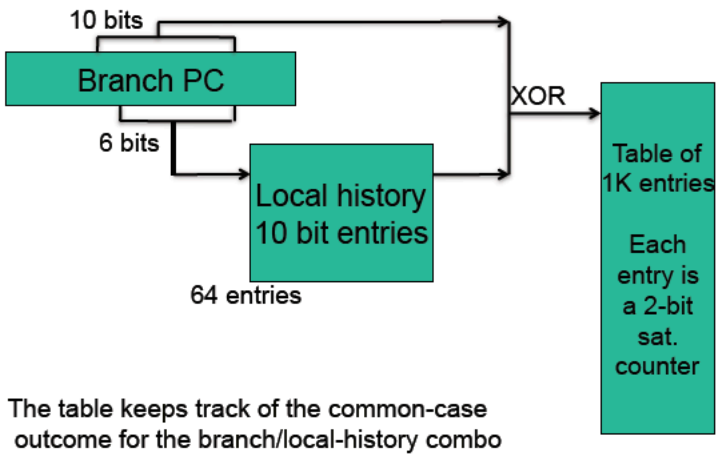

Branch PC
10 bits
XOR
Global history
Table of 1K entries
Each entry is a 2-bit sat. counter
The table keeps track of the common-case outcome for the branch/history combo

How Well Does it Work?



Even a (2,2) predictor: significantly smaller BHT and a 2-bit shift register works better

Local Predictor (Gselect)



The table keeps track of the common-case outcome for the branch/local-history combo

How is the local history set?

Which is Better?

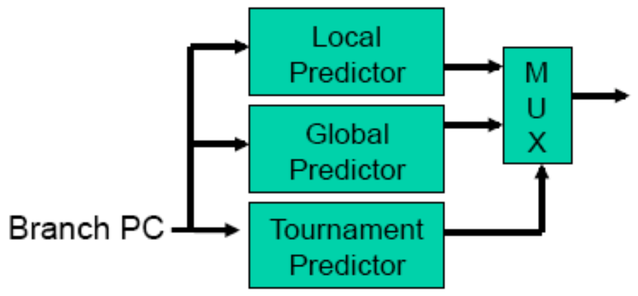
- Simple bi-modal (0,2) is the worst
  - both Gshare and Gselect are an improvement
  - Gshare is better than Gselect for table sizes > 256 bytes
- But neither work all the time
  - How can we fix this?

Which is Better?

- Simple bi-modal (0,2) is the worst
  - both Gshare and Gselect are an improvement
  - Gshare is better than Gselect for table sizes > 256 bytes
- But neither work all the time
  - How can we fix this?
  - track both and see which one would have worked best
    - » use a 2-bit saturating counter for this prediction as well
  - result is a predictor predictor
    - » since it sounds bogus it's called a predictor selector
    - » book calls it a tournament predictor
      - competition between local vs. global predictor
      - selector uses history to make the choice
    - » see Scott McFarling's 1993 paper if you want it from the source
      - link to .pdf is on the class web page
      - note renaming – original name "Combining Predictors"

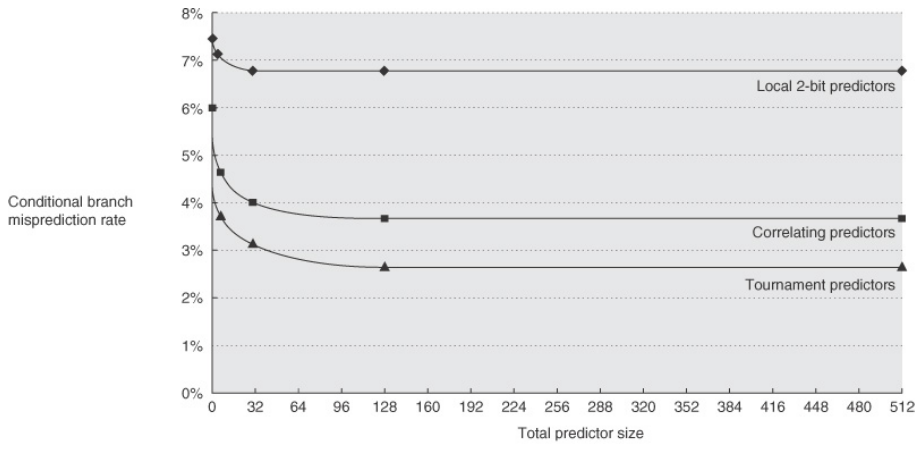
Tournament Predictor

- **Basic idea**
  - **TP is table of 2-bit counters**
    - » **decoded into taken/not-taken**
      - **e.g. high order bit is the MUX select line**



Summary

- **Compare based on number of bits of state that needs to be kept (not counting final 2-bit predictor table)**


Conditional branch misprediction rate
Total predictor size
Local 2-bit predictors
Correlating predictors
Tournament predictors