Lecture #14

****
Review -- 1 min
****
Last 4 weeks: Improving ILP
  HW
  - pipelining
  - forwarding (reduce data dependencies)
  - branch prediction (reduce control dependencies)
  - register renaming (avoid false dependencies)
  - multiple issue
  - speculative execution
  - in-order completion
  - trace cache
  - annulling instructions

  SW
  - speculation and branch prediction
  - loop unrolling
  - trace scheduling
  - sw pipelining
  - ...

Started with "as natural as laundry"
we added considerable complexity

ended up with tools that could -- in theory and given infinite HW resources -- execute all instructions in parallel limited only by true data dependencies

And proposals to go even beyond that!

****
Outline - 1 min
****
3 issues
1) what limits performance/ilp in practice

2) Memory: Motivation and Overview
3) 4 Questions of Memory Design

****
Lecture - 20 min
****
Limits to ILP

Motivation: in first lecture -- electrical engineers --> 35% performance per year via better processes
Past few years -- 55% per year -- the 20% comes from taking advantage of better ILP

pipelining (SPARC1, MIPS R2000)
superpipelining (Alpha 21064, Mips R4000)
superscaler (IBM Power, Ultrasparc, Alpha 21164,
Mips R10000)
...

What will it take to get more ILP?
Where does it end?

ILP Studies

Goal: Find out how much parallelism
Difficulty: depends on
1. Benchmarks used
2. hardware assumptions
3. compiler assumptions
(e.g. no conditional execution instruction)

This study

Aggressive HW model
1. Infinite Functional Units (1 cycle latency for all)

2. Infinite 1 cycle cache
3. Inifinite renaming registers
    -- not WAR, WAW hazards
4. Perfect branch and jump prediction
5. Infinite instruction issue
6. perfect alias analysis in memory

Picture: you have trace of entire program
(you know whether all branches are taken/not taken before program starts running)
load the trace into an infinite tomasulo engine
at every cycle, execute all instructions that no longer have data hazards

Notice, this is *really* aggressive, but, surprizingly, we can imagine a machine that does even better – HW2b, superspeculative

Also, standard MIPS compiler (could imagine a compiler that does high-level optimizations to get rid of unnecessary calculations, etc.)
As we restrict hardware, compiler limitations may become more important

Start with aggressive hardware then make more realistic assumptions.

SLIDES:
- upper limit to ILP (fig 4.38 p 319)
    - ideal processor
- window (fig 4.40 p 323)
    - dependencies (issue)^2

      Think about logic for renaming registers
      This is part of why machines will issue "2FP, 2 integers, and a branch, but not 5 integer instructions – limit dependencies to check)

branch and jump prediction (fig 4.42)

"ideal processor" – results of all branches known before first instruction executed!

For restricted cases, rule is that if you mispredict you have to stop until dependency resolved (no other penalty, just restricts parallelism)

register impact (fig 4.44 p328)

limit number of renaming registers

surprizingly much impact from name dependencies – need to look at lots of live variables to get full parallelism
(today's processors <32)

alias (4.46 p 330)

conclude: need either sophisticated compiler analysis or dynamic HW

realistic (vy aggressive) (4.48 p 332)

- <= 64 instructions issued per clock
    - of any type (e.g. all Loads? OK)
- selective predictor with 1K entries; 16-entry return predictor
- perfect disambiguation of memory refs (HW)
- register renaming w/ 64 integer and 64 FP
- vary window size
- perfect cache; 1-cycle latencies for all instructions (XXX not realistic)
-

Bottom line:

FP limited by how many instructions we can issue
Integer: limited by branch prediction

****
Admin - 3 min
****
Projects -- construct a plan
    (the plan will change, but think things out ahead of time)

Wednesday – Doug Burger will guest lecture

****
Lecture - 24 min
****

Motivation and Overview
    slide: CPU-DRAM gap ---> it's important

    1980 -- no cache on microprocessors
    1995 -- > 1/2 of transistors on microprocessor go to cache
        -- Alpha 21164~80%
    >2000 – 90%? (lipasti paper ~90%)

    slide: Alpha memory hierarchy --> people build complex systems

Basics

    Smaller is Faster
        spend more $'s per bit
        spend more W (watts) per bit
        electrical considerations (e.g. speed of light; RC delay)

    Locality
        Temporal -- referenced again soon
        Spatial -- nearby items referenced soon

    Locality + smaller is faster --> memory hierarchy
        levels -- each smaller/faster/more expensive than level below

inclusive -- data found on top also found on bottom

Definitions

upper == closer to processor

block == minimum unit that is present/not in upper level

address = block frame address + block offset

hit time == time to acces upper level

(including hit determination)

hit rate == fraction found at that level

hit rate high, misses expensive --> talk about miss rate

(95% hit rate v. 90% hit rate -- doesn't sound too different

5% miss rate v. 10% miss rate -- sounds much different

second one is closer to performance -- gives better intuition.

miss rate fallacy

look at avg memory access time, not just miss rate

(miss rate for caches ~MIPS for CPU performanc)

local v. global miss rate

local = #misses in cache / #requests to cache

global = #misses in cache / #requests CPU

in hierarchical caches, higher levels of cache skim "cream" --> lower levels look poor

--> talk about global miss rate

Implications

L1 cache: optimize hits

on CPU critical path

Unpredictable Mem Access Time

L1: 1-3 cycles
L2: 10-20 cycles
Memory: 100+ cycles
Disk: millions of cycles

10's of cycles -- wait
1000's of cycles -- do something else

Performance

CPU time = (CPU execution clock cycles
+ memory stall cycles) * clock cycle time

Memory stall cycles =reads * read miss rate * read miss penalty
+ writes * write miss rate * write miss penalty

sometimes simplify
= mem access * miss rate * miss penalty

CPU time = IC * (CPI_execution
+ misses per instruction * miss penalty)

* clock cycle time

Misses Per Instruction = mem access per instruction
* miss rate

4 Questions for Memory Hierarchy Designers

Q1: Block Placement: Where can a block be placed in the upper level?
    Picture: fully associative, direct mapped, set associative

Q2: Block identification: How is a block found if it is in the upper level?

```
|            Block Address          |  Block offset |
|     Tag             |   index     |               |
```

PICTURE: cache -- indexed by index
                    select block with offset
                    compare with tag

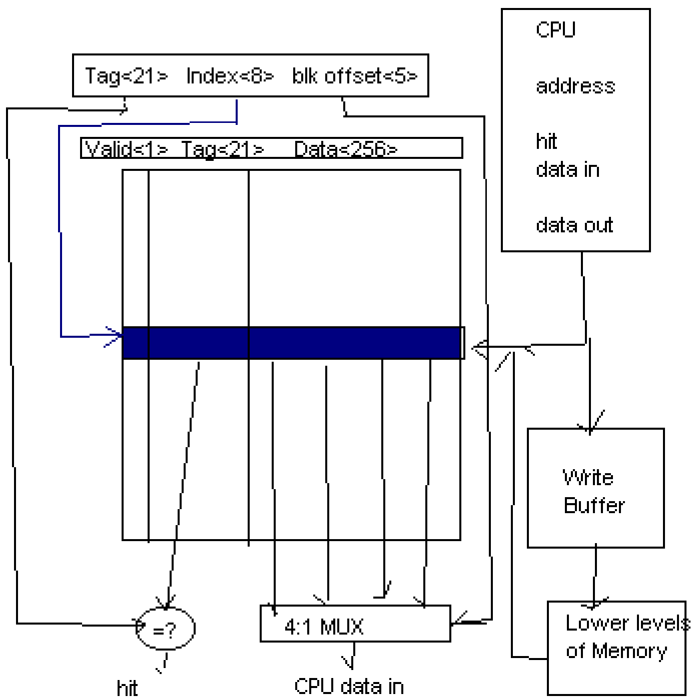
Q3: Block replacement: Which block should be replaced on a miss?
    Easy for direct mapped (only 1 choice)

    Otherwise several choices:
        Optimal: replace block used furthest into future
        LRU -- works well , but lots of HW
        Random works pretty well
        Random but not MRU often a good compromise

Q4: Write Strategy: What happens on a write?
    write back
    write through
    (w or w/o write buffer)

Example: Alpha 21064 Data Cache

8192K Byte
Direct Mapped
block = 4 x 64-bit words
--> 256 blocks (8K / 32 bytes per line)



Improving Cache Performance

Mem Access Time = Hit time + miss rate * miss penalty

Improve performance by
1. reduce miss rate
2. reduce miss penalty
3. reduce time to hit in cache

****
Summay - 1 min
****

o CPU memory gap --> obstacle for performance

o Take advantage of program behavior -- locality

o time of program is still only reliable performance measure

- 4Q's of memory hierarchy