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
  SW
    speculationa 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

****
Outline - 1 min
****
Memory: Motivation and Overview
Feedback
4 Questions of Memory Design
Translation
    multilevel translation
    paged page tables
    inverted page tables

****
Lecture - 20 min
****

Motivation and Overview

slide: fig 5.1 CPU-DRAM gap ---> it's important

1980 -- no cache on microprocessors
1995 -- > 1/2 of transistors on microprocessor go to cache
-- Alpha ~80%
-- PPro --
-- 2 chip package
-- ~20% chip 1 (1st levl cache)
-- 100% of chip 2 (> chip 1)

slide: fig5.47 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 bottome

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: 10-100+ cycles

Disk: millions of cycles

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

****
Admin - 3 min
****

HW3 ready -- more emphasis on "lab" work
-- seeing how real computers work
-- using real tools

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

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

Performance

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

Memory stall cycles = reads * read miss rate * read mis 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

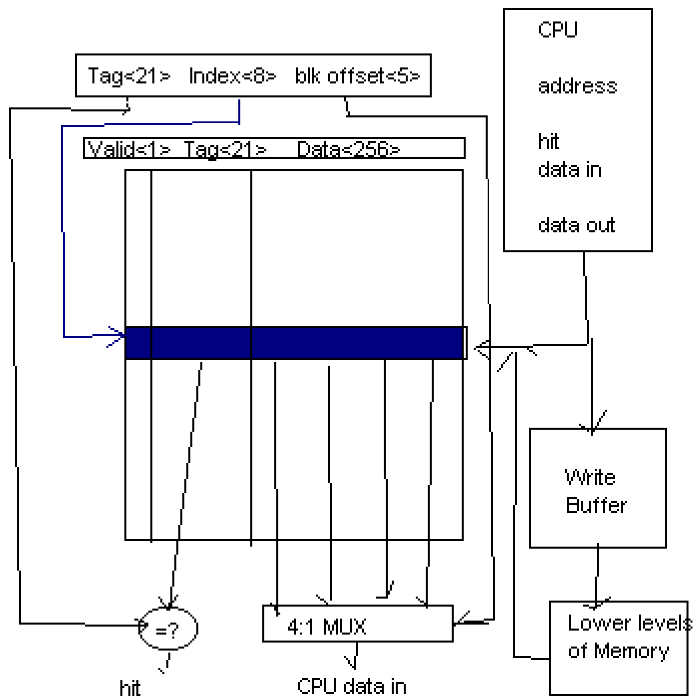
Q3: Block replacement: Which block should be replaced on a miss?
Easy for direct mapped (only 1 choice)

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 hapens 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

****
****

Translation

****
****

Really usefult to have programs talk in terms of *virtual addresses* – relocation etc. And want hardware to enforce address spaces for security.

Several ways to accomplish.

Paging and segmentation

pro: simple, easy to share entire segment or page

Segmentation

```
if(va.offset < table[segID].size){
    pa = table[segID].base + va.offset;
}
```

Paging

```
if(va.vpage < table.npages){
    pa = table[va.vpage] .ppage * pageSize || va.offset;
}
```

con:

segmentation – fitting odd sized fragments into odd-sized regions of free memory

paging – size of page table (esp for sparse address space)

e.g. unix code, heap at 0, stack at 2^31-1

2^31 bytes of VA / 2^13 bytes/page → 264K entries

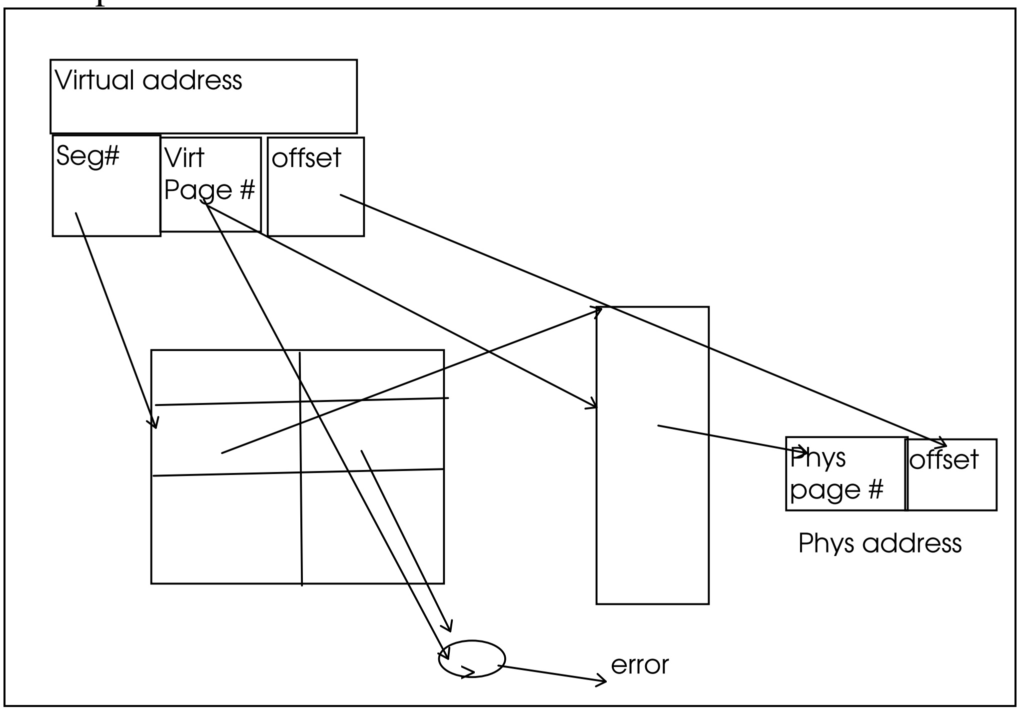
Multi-level translation

Use a tree of tables

Lowest level is page table so that physical memory can be allocated via bitmap

Higher levels (typically) segmented

example: 2-level translation



Just like recursion, can have any number of levels in tree

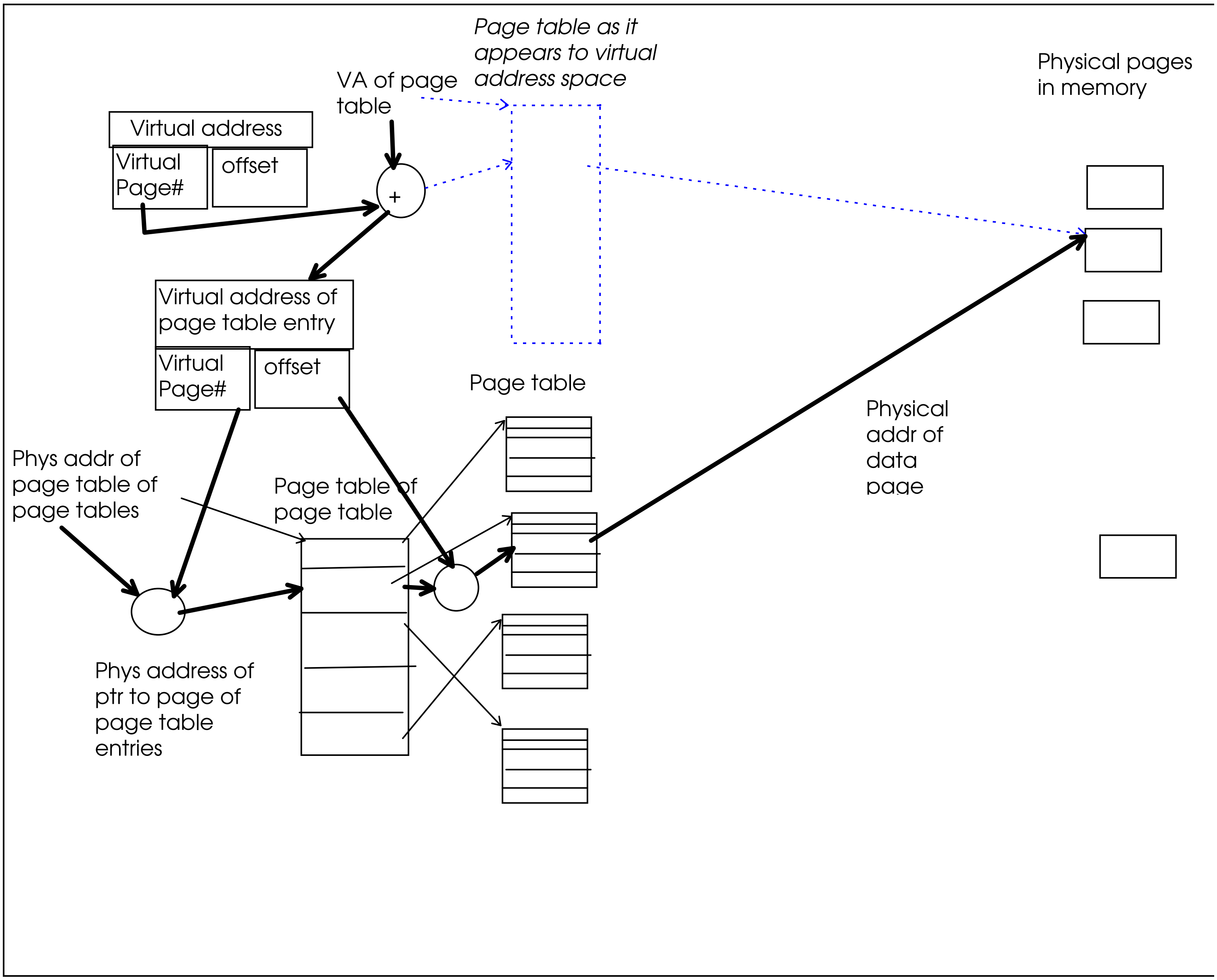
Multilevel translation pros & cons
+ only need to allocate as many page table entries as we need
+ easy memory allocation
+ share at segment or page level
- pointer per page (typically 4KB - 16 KB pages today)
- each page tables needs to be physically contigous
    128 MB executable w 4KB pages
    → 32K entries at at least 4 bytes / entry → 128KB contigous needed
- two (or more) hops per memory reference

2 problems with segemented paging
1) need large contigous memory → page page tables
2) multiple memory references to do lookup → inverted page table

Paged page tables

Another way to solve sparse address spaces is to allow page tables to be paged; only allocate physical memory for page table entries you actually use

Top level page table is in physical memory (stored in contigous phys mem)



all lower levels are in virtual memory (stored in contiguous virt mem but in fixed-sized frames in physical memory) notice – although page table stored in virtual memory, it is stored in kernel virtual address space, not process virtual address space (otherwise process could modify it)

****
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

o 4Q's of memory hierarchy