Help Session votes

I realized that I have conflicts on Monday 2/27 9-2pm
And I also have a meeting on Friday 2/24 9-12 noon
Based on the votes received, this leaves us with Friday 2/24 at 1pm

Review

Techniques to improve cache performance
Multilevel caches
Global and local miss rates
Inclusive vs Exclusive caches

Virtual to Physical address translation
page tables
Segment tables
Multiple page tables
Translation Lookaside Buffer (TLB)

For 64-bit addresses we may use 3 or more levels of page tables to translate an address
and with each added level, an extra memory access

Place tables in cache memory
(sharing with data cache or separate)
CSCE 4610: Computer Architecture

Putting the pieces together

memory layout and a snapshot of physical map

Virtual addresses

Physical Memory

Process 1
Higher Memory

Main Memory
Physical Higher Memory

STACK

HEAP

Code

Data

Lower Memory

CSCE 4610  Feb 16, 2017

Let us only consider the data memory (not instructions). The first data address referenced is 0x7fffffffece8 = 0000 1111 1111 1111 1111 1111 1111 1111 1110 1100 1110 1000

A hardware register points to L1 table. This table is empty. Allocate a page L1 table for 01111111

Store the address of this page in base table at entry 01111111

There will no entry in this new L1 table for entry 11111111. Allocate a page for L2 table

Store the address of this page in L1 table at entry 11111111

There will be no entry in the new L2 table for 11111111. Allocate a page for L3 table

Store the address of this page in L2 table at entry 11111111.

There will be no entry in this new L3 table for the page 11111111. Allocate a physical page

In our example we allocate page B23C. Store this address in L3 page table at entry 11111110

Let us only consider the data memory (not instructions). The first data address referenced is 0x7fffffffece8 = 0000 xxxx 1111 1111 1111 1111 1111 1111 1111 1111 1110 1100 1110 1000

A hardware register points to L1 table. This table is empty. Allocate a page L1 table for 01111111

Store the address of this page in base table at entry 01111111

There will no entry in this new L1 table for entry 11111111. Allocate a page for L2 table

Store the address of this page in L1 table at entry 11111111

There will be no entry in the new L2 table for 11111111. Allocate a page for L3 table

Store the address of this page in L2 table at entry 11111111.

There will be no entry in this new L3 table for the page 11111111. Allocate a physical page

In our example we allocate page B23C. Store this address in L3 page table at entry 11111110
CSCE 4610: Computer Architecture

Need multiple memory accesses to translate (all tables are in memory or cache) – as many as 7 access in 64 bit processors

To avoid the repeated address translation
we can use a fully associative cache called TLB
Search the TLB for a match with Virtual address;
if no match use page tables

So, to improve cache hit time, we should use virtual addresses

*note that the offset to a page is the same in both virtual and physical addresses*

Two issues: virtual or physical address for indexing?
virtual or physical address for tag comparison?

If use virtual indexes, the “size” of cache is small
4K pages → 4K caches
Or we can use higher associativity so we can use larger cache
8-way → use 32K cache
16-way → use 64K cache
32 way → use 128K cache

This is one of the reasons for higher associative cache even at L1 level

Or we can use larger pages – same 32KB pages
Then we can use direct mapped 32KB cache
or use 256KB 8-way associative cache and still use virtual indexes

Or eliminate some of the virtual to physical translation (our work for AMD)
CSCE 4610: Computer Architecture

TLB misses and Page faults
These impact CPI

An example
Consider a 16K byte direct mapped L-1 cache with write-back strategy. Assume 10% miss rate. Assume that the access time to DRAM is 40 cycles.

Assume that 50% of all transfers are dirty – so must be written back if evicted

Also assume that it takes 20 clock cycles on a TLB miss (remember the use of TLB for address translation) and there is no penalty on a TLB hit.

Also assume that 10% of references to TLB are not found in TLB.

Compute the effective CPI assuming both Physical addresses and Virtual addresses.

Assume that it takes 1.0 cycle to access cache on a hit
CSCE 4610: Computer Architecture

Virtual Address Cache. *Here we can ignore the TLB.*

For each miss, whether read or write, the miss may cause a write-back if the victim is dirty.

Average Access time per access = 1 + (miss rate)*(miss penalty)

On miss, if the victim needs to be written back, *we have to pay for two memory accesses to DRAM.* Since this happens 50% of the time, on average we have 1.5 DRAM accesses or 1.5 * 40 = 60 cycle penalty

Average time per memory access = 1 + (10%) * 60 = 7 cycles

If we assume Physical addresses, if there is a hit in TLB, we do not pay any penalty. But if there is a TLB miss we pay 20 cycle penalty

Since we have 10% miss rate in TLB, on average we pay an extra 2 cycles

Average time per memory access = 7 + 2 = 9 cycle

(7 when there is a hit in TLB, but still have account for write-backs)

---

CSCE 4610: Computer Architecture

So, to improve cache hit time, we should use virtual addresses

**Why do we not use virtual addresses for all caches?**

1. Protection – usually at page level and this information is stored in TLB and page tables
   - Access rights per page
   - R/W/E
   - Base and limit registers
   - page table registers

   We also need to define protected states of execution (or protection levels)
   - User/Supervisor
   - Modifications to page tables done only in supervisor mode

   We can extend this concept to more than 2 states – we will refer them as rings
   - Intel IA-32 processors support 4 levels
     - Level 0 is most protected
     - Level 3 is user level
   - We may comeback to protection if time permits in this class
CSCE 4610: Computer Architecture

2. Second reason for not using virtual addresses for Caches

On context switch a virtual page may be relocated to a different physical page
    May need to either flush cache
    and bring data from new physical addresses

3. Third Reason: Aliasing
    May use two different virtual addresses for the same page
    Shared memory segments
    libraries and OS functions

So, most modern processors prefer to use Physical addresses or
    L1: Physical address for tag, virtual address for index
    L2: Physical address for tag and physical address for index

CSCE 4610/4610: Computer Architecture

Way prediction technique

to improve hit time (similar to pseudo associative caches)

*Compare only one tag out of k-tags in k-way associative caches*

In a 8 way associative cache, you pick one of the ways and compare
    tag of the address with the tag of selected way
    If tag does not match, then search all 8 ways

How to predict which way?
    based on previous hit
    higher bits of the address

Do you think this works?
Also on replacement, do you also use a strategy to improve the prediction
CSCE 4610/4610: Computer Architecture

For 2 way it has been shown that way prediction is correct 90% of the time
for 4 way it is 80% correct
Better prediction for I cache
2-way prediction can result in 10% faster access to cache (on hit)

Used in ARM Cortex-A8 architecture

See example on page 82 – to understand the impact of way prediction

For every I cache access there are 0.5 D cache accesses
D cache is responsible for 15% and I cache for 25% of power consumption

Way prediction increased access times as follows (on a way mis-prediction need an extra
cycle for tag match)
for D cache by 13% and for I cache by 4% way mis-prediction

Way prediction also reduced power consumption as follows
D cache consumes 35% less than regular 4-way D cache
I cache consumes 28% less than as regular 4-way I cache

CSCE 4610  Feb 16, 2017

Using way prediction for for I and/or D cache
Net power savings for I cache = 25%*0.28 = 0.07 (or 7%) of total power
D cache = 15%* 0.35 = 0.05 (or 5%) of total power
Total savings = 12%

We need to also find out if the increased access time leads to more energy consumption

Increased access time: 1.04 (for I cache) + 0.5*1.13 (for D cache) =1.605

Compared to standard 4-way  (50% data accesses): 1+0.5 = 1.5

So, how much energy is saved?

\[
\frac{\text{original energy}}{\text{new energy}} = \frac{(\text{original execution})\ast(\text{original power})}{(\text{new execution})\ast(\text{new power})}
\]

\[
\frac{1.5\ast1}{1.605\ast0.88} = \frac{1.5}{1.4124} = 1.062
\]

6.2% energy savings

Note the difference between power and energy
Another technique to improve cache performance is Non-blocking Caches (see the figure on page 84).

1 miss pending reduces miss penalty by 9% for integer applications.

2 misses pending reduces miss penalty by 10%.

More misses pending do not show any more improvements.

An Example: compare 2-way set associative with hit on one miss non-blocking.

Given: for Floating Point (data partly from the figure 2.5)

- Direct mapped cache: Miss rate = 5.2%
  Miss penalty = 10 cycles

- 2-way cache: Miss rate = 4.9%
  Miss penalty = 10 cycles

Average stall due to caches:
1-way = 5.2% * 10 = 0.52 cycles
2-way = 4.9% * 10 = 0.49 cycles
Ratio = 0.49/0.52 = 94%

- note in reality, we should use time in ns for miss penalty
- but here we are assuming the same clock frequency for direct and 2-way

From the figure on page 84, we are told that hit under one miss reduces latency by 87.5% over direct mapped blocking cache. So, for floating point it is better to have direct mapped cache with hit under one miss instead of 2-way associative.

Repeat this with integer benchmarks.
Multi-banked caches (page 86)

Different ways of interleaving: low order and high order

**Hardware prefetching**
- On every cache miss, in addition to missing cache line, fetch next \( m \) lines (counting on spatial locality)
- Strided prefetch (for arrays)

Issues: Should we give priority to demand request or prefetch request?
Will come back to this soon

More issues with hardware prefetching
- Prefetch may prematurely evict needed data (cache pollution)
- Use separate prefetch buffers

**Software prefetching**
- Insert prefetch instructions
- Assumption: need non-blocking caches

Consider this example (slightly modified version of the example on page 93)

\[
\text{for } (k=0; k<3; k=k+1) \\
\text{ } \quad \text{for } (j=0; j < 100; j = j+1) \\
a[k][j] = b[j][0]*b[j+1][0];
\]

How many cache misses are caused?
Consider 16 byte blocks in a 8KB cache (for now large enough for the application)
Each array element is 8 bytes long
How many misses are caused by this code?

For $a$, we have 3 rows and 100 columns
but each line contains 2 elements
Total misses for $a = (3 \times 100) / 2 = 150$

For $b$, we are accessing 101 elements of column 0
And this will cause 101 misses

A total of 251 misses