## ECE 252 / CPS 220 Advanced Computer Architecture I

## Lecture 13 Memory – Part 2

Benjamin Lee Electrical and Computer Engineering Duke University

www.duke.edu/~bcl15 www.duke.edu/~bcl15/class/class\_ece252fall11.html



#### 20 October – Project Proposals Due

One page proposal

- 1. What question are you asking?
- 2. How are you going to answer that question?
- 3. Talk to me if you are looking for project ideas.

#### 25 October – Homework #3 Due

#### 25 October – Class Discussion

Roughly one reading per class. Do not wait until the day before!

- 1. Jouppi. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers."
- 2. Kim et al. "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches."
- 3. Fromm et al. "The energy efficiency of IRAM architectures"
- 4. Lee et al. "Phase change memory architecture and the quest for scalability"



#### History of Memory

- Williams tubes were unreliable. Core memory reliable but slow
- Semiconductor memory competitive in 1970s.

### **DRAM** Operation

- DRAM accesses require (1) activates, (2) reads/writes, (3) precharges
- Performance gap between processor and memory is growing.

### Managing the Memory Hierarchy

- 1. Predictable patterns provide (1) temporal and (2) spatial locality
- 2. Exploit predictable patterns with small, fast caches.
- 3. Cache data placement policies include (1) fully-associative, (2) setassociative, and direct-mapped. Choice of policy determines cache effectiveness.







### AMAT = [Hit Time] + [Miss Prob.] x [Miss Penalty]

- Miss Penalty equals AMAT of next cache/memory/storage level.
- AMAT is recursively defined

#### To improve performance

- Reduce the hit time (e.g., smaller cache)
- Reduce the miss rate (e.g., larger cache)
- Reduce the miss penalty (e.g., optimize the next level)

#### Simple design strategy

- Observe that hit time increases with cache size
- Design the largest possible cache with a hit time of 1-2 cycles.
- For example, design 8-32KB of cache in modern technology
- Design trade-offs are more complex with superscalar architectures and multi-ported memories



#### **Serial Access**

- Check cache for addressed data. If miss (probability 1-p), go to memory
- AMAT = Tcache + (1-p) x Tmem



#### **Parallel Access**

- Check cache and memory for addressed data.
- $AMAT = p \times Tcache + (1-p) \times Tmem$
- AMAT reductions from parallel access often small. Tmem >> Tcache and p is high. Parallel access consumes memory bandwidth Parallel access increases cache complexity and Tcache.



#### Compulsory

- First reference to a block. May be caused by "cold" caches as application begins execution.
- Compulsory misses would occur even with infinitely sized cache

### Capacity

- Cache is too small and cannot hold all data needed by the program.
- Capacity misses would occur even under perfect replacement policy.

#### Conflict

- Cache line replacement policy causes collisions
- Conflict misses would not occur with full associativity



#### Larger Cache Size

- Benefit: reduces capacity and conflict misses
- Cost: increases hit time

#### Higher Associativity

- Benefit: reduces conflict misses
- Cost: increases hit time

#### Larger Line Size

- Benefit: reduces compulsory and capacity misses
- Cost: increases conflict misses and increases miss penalty



What happens when a cache line is written? If write hits in cache (i.e., line already cached)

- <u>Write-Through</u>: Write data to both cache and memory. Increases memory traffic but allows simpler datapath and cache controllers.
- <u>Write-Back</u>: Write data to cache only. Write data to memory only when cache line is replaced (e.g., conflict).
- <u>Write-Back Optimization</u>: Insert "dirty bit" per cache line, which indicates whether line is modified. Write-back only if replaced cache line is dirty.

#### If write misses in cache

- <u>No Write Allocate</u>: Write data to memory only.
- <u>Write Allocate</u>: Fetch data into cache. Write data into cache. Also known as fetch-on-write.

#### Common combinations

- Write-through and no-write allocate
- Write-back and write allocate.







#### Problem: Writes take two cycles in memory

- Access cache and compare tags to generate HIT signal (1 cycle)
- Perform cache write if HIT enables a write (1 cycle)

#### Solutions

- Design data SRAM that can perform read/write in one cycle, restoring old value if HIT is false.
- Design content-addressable data SRAM (CAM), which enables word line only if HIT is true.
- Pipeline writes with a write buffer. Write the cache for j-th store instruction during the tag check for (j+1)-th store instruction

## Pipelining Cache Writes





#### With buffers, writes do not stall datapath

- Introduce a write buffer between adjacent levels in cache hierarchy.
- After writes enter buffer, computation proceeds.
- For example: Reads bypass writes.

#### Problem

- Write buffer may hold latest value for a read instruction's address

#### Solutions

- Option 1: If a read misses in cache, wait for the write buffer to empty
- Option 2: Compare read address with write buffer addresses. If no match, allow read to bypass writes. Else, return value in write buffer.



#### Problem

- Memory technology imposes trade-off between speed and size.
- A memory cannot be both large and fast.

### Solution

- Introduce a multi-level cache hierarchy.
- As distance from datapath increases, increase cache size.





#### Use smaller L1 cache if L2 cache is present

- Reduce L1 hit time, but increase L1 miss rate.
- L2 mitigates higher L1 miss rate by reducing L1 miss penalty
- May reduce average time (AMAT) and energy (AMAE)

### Use write-through L1 if write-back L2 cache is present

- Write-through L1 simplifies pipeline, cache controller.
- No write-backs for dirty lines reduces complexity.
- Write-back L2 absorbs write traffic. Writes do not go off-chip to DRAM.

#### Inclusion Policies

- <u>Inclusive Multi-level Cache</u>: Smaller cache (e.g., L1) holds copies of data in larger cache (e.g., L2). Simpler policies.
- <u>Exclusive Multi-level Cache:</u> Smaller cache (e.g., L1) holds data not in larger cache (e.g., L2). Example: AMD Athlon with 64KB primary and 256KB secondary. Reduces duplication.

# Power7 On-Chip Caches (2009)

32KB L1 I\$/core 32KB L1 D\$/core 3-cycle latency

256KB Unified L2\$/core 8-cycle latency

32MB Unified Shared L3\$ Embedded DRAM 25-cycle latency to local slice





#### Speculate about future memory accesses

- Predict likely instruction and data accesses.
- Pre-emptively fetch instructions and data into caches.
- Instructions accesses likely easier to predict than data accesses.
- Mechanisms might be implemented in HW, SW or both.
- What type of misses does prefetching affect?

#### Challenges in Prefetching

- Prefetching should be <u>useful</u>. Prefetches should reduce misses.
- Prefetching should be <u>timely</u>. Prefetches will pollute the cache if too early and will be useless if too late.
- Prefetching may pollute caches and consume memory bandwidth.



#### Example: Alpha AXP 21064

- Prefetch instructions
- Fetch two lines on a cache miss. Fetch the requested line (i) and the next consecutive line (i+1).
- Place requested line in instruction L1 cache. Place next line in an instruction stream buffer.
- If an instruction fetch misses in L1 cache but hits in stream buffer, move stream buffer line into L1 cache. And prefetch next line (i+2)





#### Prefetch after a cache miss

- Prefetch line (i+1) if an access for line (i) misses in the cache

### One Block Look-ahead (OBL)

- Blocks also known as lines.
- Initiate prefetch for block (i+1) when block (i) is accessed.
- Generalizes to N-block look-ahead
- How is this different from increasing the block or line size by N times?

#### Strided Prefetch

- Observe sequence of accesses to cache lines.
- Suppose a sequence (i), (i+N), (i+2N) is observed. Prefetch (i+3N).
- <u>N</u> is the stride.
- Example: IBM Power 5 (2003) supports eight independent streams of strided prefetches per processor, prefetching 12 lines ahead of the current access.



```
for(i=0; i < N; i++) {
    prefetch( &a[i + P] );
    prefetch( &b[i + P] );
    SUM = SUM + a[i] * b[i];
}</pre>
```

#### Challenges in Software Prefetching

- Timing is the biggest difficulty, not predictability.
- Prefetch too late if prefetch instruction too close to data use.
- Prefetch too early and cause cache/bandwidth pollution.
- Requires estimating prefetch latency: time from issuing prefetch to filling L1 cache line.
- Why is this hard to do? What is the correct value of P above?



#### Restructuring code affects data access sequences

- Group data accesses together to improve spatial locality
- Re-order data accesses to improve temporal locality

#### Prevent data from entering the cache

- Useful for variables that are only accessed once
- Requires SW to communicate hints to HW.
- Example: "no-allocate" instruction hints

#### Kill data that will never be used again

- Streaming data provides spatial locality but not temporal locality
- If particular lines contain dead data, use them in replacement policy.
- Toward software-managed caches



What type of locality does this improve? What does it assume about x?



```
for (i=0; i < N; i++)
    a[i] = b[i] * c[i];
for(i=0; i < N; i++)
     d[i] = a[i] * c[i];
for(i=0; i < N; i++)
{
       a[i] = b[i] * c[i];
       d[i] = a[i] * c[i];
}
```

What type of locality does this improve?

## Matrix Multiply (X=YZ) – Naïve



#### <u>Notes</u>

- 1. Iterate through matrix (y) by row.
- 2. Iterate through matrix (z) by col.
- 3. Update matrix (x) by col.







# Matrix Multiply (X=YZ) – Blocked

```
for(jj=0; jj < N; jj=jj+B)
for(kk=0; kk < N; kk=kk+B)
for(i=0; i < N; i++)
for(j=jj; j < min(jj+B,N); j++) {
    r = 0;
    for(k=kk; k < min(kk+B,N); k++)
        r = r + y[i][k] * z[k][j];
        x[i][j] = x[i][j] + r;
}</pre>
```



#### <u>Notes</u>

- 1. Organize matrix into BxB submatrices.
- 2. Iterate through blocks (jj, kk).
- 3. For each block, iterate through its elements (i, j).

What type of locality does this improve? Hint: Track the re-use of matrix elements during computation.







Old access





#### Caches

- Quantify cache/memory hierarchy performance with AMAT
- Three types of cache misses: (1) compulsory, (2) capacity, (3) conflict
- Cache structure and data placement policies determine miss rates
- Write buffers improve performance.

#### Prefetching

- Identify and exploit spatial locality
- Prefetchers can be implemented in hardware, software, or both

#### Caches and Code

- Restructuring SW code can improve HW cache performance
- Data re-use can improve with code structure (e.g., matrix-multiply)



These slides contain material developed and copyright by

- Arvind (MIT)
- Krste Asanovic (MIT/UCB)
- Joel Emer (Intel/MIT)
- James Hoe (CMU)
- John Kubiatowicz (UCB)
- Alvin Lebeck (Duke)
- David Patterson (UCB)
- Daniel Sorin (Duke)