Improving on Caches

CS448

1

#4: Pseudo-Associative Cache

- Also called column associative
- Idea
  - start with a direct mapped cache, then on a miss check another entry
    - A typical next location to check is to invert the high order index bit to get the next try
  - Similar to hashing with probing
- Initial hit fast (direct), second hit slower
  - may have the problem that you mostly need the slow hit
  - in this case it's better to swap the blocks
  - like victim caches - provides selective on demand associativity

2

#5: Hardware Prefetch

- Get proactive!
- Modify our hardware to prefetch into the cache instructions and data we are likely to use
  - Alpha AXP 21064 fetches two blocks on a miss from the I-cache
    - Requested block and the next consecutive block
    - Consecutive block catches 15-25% of misses on a 4K direct mapped cache, can improve with fetching multiple blocks
  - Similar approach on data accesses not so good, however
- Works well if we have extra memory bandwidth that is unused
- Not so good if the prefetch slows down instructions trying to get to memory

3

#6 Compiler-Controlled Prefetch

- Two types
  - Register prefetch (load value into a register)
  - Cache prefetch (load data into cache, need new instr)
- The compiler determines where to place these instructions, ideally in such a way as to be invisible to the execution of the program
  - Nonfaulting instructions – if there is a fault, the instruction just turns into a NOP
- Only makes sense if cache can continue to supply data while waiting for prefetch to complete
  - Called a *nonblocking* or *lockup-free* cache
- Loops are a key target

4

Compiler Prefetch Example

Using a Write-Back cache

```
for (i=0; i<3; i++)
   for (j=0; j<100; j++)
       a[i][j]=b[j][0]+b[j+1][0];
```

Spatial locality
Say even j's miss, odd hit
Total of 300/2 = 150 misses

Temporal locality
Hits on next iteration
Misses on j=0 only
Total of 101 misses

Prefetched version, assuming we need to prefetch 7 iterations in advance to avoid the miss penalty. Doesn't address initial misses:

```
for (j=0; j<100; j++) {
   prefetch(b[j+7][0]);
   prefetch(a[0][j+7]);
   a[0][j]=b[j][0]+b[j+1][0];
}
for (i=0; i<3; i++)
   for (j=0; j<100; j++) {
       prefetch(a[i][j+7]);
       a[i][j]=b[j][0]+b[j+1][0]; }
```

Fetch for 7 iterations later
Pay penalty for first 7 iterations

Total misses = (3*7/2) + 1 + 7
= 19

5

#7 Compiler Optimizations

- Lots of options
- Array merging
  - allocate arrays so that paired operands show up in same cache block
- Loop interchange
  - exchange inner and outer loop order to improve cache performance
- Loop fusion
  - for independent loops accessing the same data
  - fuse these loops into a single aggregate loop
- Blocking
  - Do as much as possible on a sub-block before moving on
  - We'll skip this one

6

Array Merging

Given a loop like this:

```
int val1[SIZE], val2[SIZE];
for (i=0; i<1000; i++) {
    x += val1[i] * val2[i];
}
```

For spatial locality, instead use:

```
struct merge {
   int val1, val2;
} m[SIZE];

for (i=0; i<1000; i++) {
    x += m[i].val1 * m[i].val2;
}
```

For some situations, array splitting is better:

```
struct merge {
   int val1, val2;
} m1[SIZE], m2[SIZE];

for (i=0; i<1000; i++) {
  x += m1[i].val1 * m2[i].val1;
}
```

val2 unused, getting in the way of spatial locality. First version could actually be better!

Objects can be good or bad, depending on access pattern

7

Loop Interchange

```
for (i=0; i<100; i++) {
   for (j=0; j< 5000; j++)
       x[i][j]++;
}
```

Say the cache is small, much less than 5000 numbers
We'll have many misses in the inner loop due to replacement

Switch order:

```
for (i=0; i<5000; i++) {
   for (j=0; j< 100; j++)
       x[i][j]++;
}
```

With spatial locality, presumably we can operate on all 100 items in the inner loop without a cache miss

Access all words in the cache block before going on to the next one

8

Loop Fusion

```
for (i=0; i<100; i++) {
   for (j=0; j< 5000; j++)
       a[i][j]=1/b[i][j] * c[i][j];
}
for (i=0; i<100; i++) {
   for (j=0; j< 5000; j++)
       d[i][j]=a[i][j] * c[i][j];
}
```

Merge loops:

```
for (i=0; i<100; i++) {
   for (j=0; j< 5000; j++)
       a[i][j]=1/b[i][j] * c[i][j];
       d[i][j]=a[i][j] * c[i][j];
}
```

Freeload on cached value!

9

Reducing Miss Penalties

- So far we've been talking about ways to reduce cache misses
- Let's discuss now reducing access time (the penalty) when we have a miss
- What we've seen so far
  - #1: Write Buffer
    - Most useful with write-through cache
    - no need for the CPU to wait on a write
      - hence buffer the write and let the CPU proceed
      - needs to be associative so it can respond to a read of a buffered value

10

Problems with Write Buffers

- Consider this code sequence
  - SW 512(R0), R3 ← Maps to cache index 0
  - LW R1, 1024(R0) ← Maps to cache index 0
  - LW R2, 512(R0) ← Maps to cache index 0
- There is a RAW data hazard
  - Store is put into write buffer
  - First load puts data from M[1024] into cache index 0
  - Second load results in a miss, if the write buffer isn't done writing, the read of M[512] could put the old value in the cache and then R2
- Solutions
  - Make the read wait for write to finish
  - Check the write buffer for contents first, associative memory

11

#2 Other Ways to Reduce Miss Penalties

- Sub-Block Placement
  - Large blocks reduces tag storage and increases spatial locality, but more collisions and a higher penalty in transferring big chunks of data
  - Compromise is Sub-Blocks
  - Add a "valid" bit to units smaller than the full block, called sub-blocks
    - Allow a single sub-block to be read on a miss to reduce transfer time
    - In other modes of operation, we fetch a regular-sized block to get the benefits of more spatial locality

12

#3 Early Restart & Critical Word First

- CPU often needs just one word of a block at a time
  - Idea : Don't wait for full block to load, just pass on the requested word to the CPU and finish filling up the block while the CPU processes the data
- Early Start
  - As soon as the requested word of the block arrives, send it to the CPU
- Critical Word First
  - Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling in the rest of the block

13

#4 Nonblocking Caches

- Scoreboarding or Tomasulo-based machines
  - Could continue executing something else while waiting on a cache miss
  - This requires the CPU to continue fetching instructions or data while the cache retrieves the block from memory
  - Called a nonblocking or lockup-free cache
  - Cache could actually lower the miss penalty if it can overlap multiple misses and combine multiple memory accesses

14

#5 Second Level Caches

- Probably the best miss-penalty reduction technique, but does throw in a few extra complications on the analysis side…
- L1 = Level 1 cache, L2 = Level 2 cache

$$Average\_Memory\_Access\_Time = Hit\_Time(L1) + Miss\_Rate(L1) \times Miss\_Penalty(L1)$$

$$Miss\_Penalty(L1) = Hit\_Time(L2) + Miss\_Rate(L2) \times Miss\_Penalty(L2)$$

- Combining gives:

$$Average\_Memory\_Access\_Time = Hit\_Time(L1) + Miss\_Rate(L1) \times \left(Hit\_Time(L2) + Miss\_Rate(L2) \times Miss\_Penalty(L2)\right)$$

  - little to be done for compulsory misses and the penalty goes up
  - capacity misses in L1 end up with a significant penalty reduction since they likely will get supplied from L2
  - conflict misses in L1 will get supplied by L2 unless they also conflict in L2

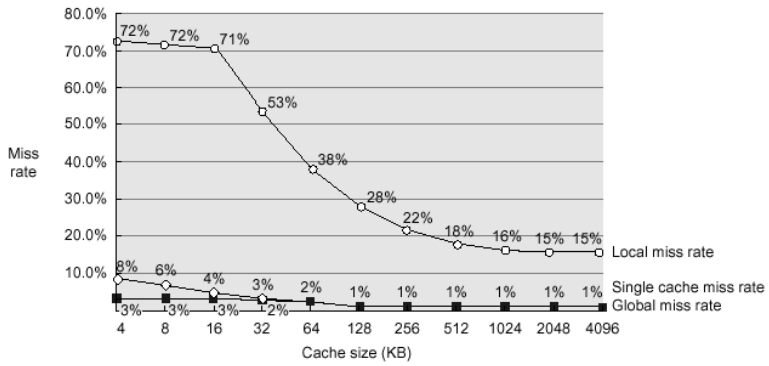
15

Second Level Caches

- Terminology
  - Local Miss Rate
    - Number of misses in the cache divided by total accesses to the cache; this is Miss Rate(L2) for the second level cache
  - Global Miss Rate
    - Number of misses in the cache divided by the total number of memory accesses generated by the CPU; the global miss rate of the second-level cache is
      - Miss Rate(L1)*Miss Rate(L2)
    - Indicates fraction of accesses that must go all the way to memory
  - If L1 misses 40 times, L2 misses 20 times for 1000 references
    - 40/1000 = 4% local miss rate for L1
    - 20/40 = 50% local miss rate for L2
    - 20/40 * 40/1000 = 2% = global miss rate for L2

16

Effects of L2 Cache



L2 cache with 32K L1 cache
Top: local miss rate of L2 cache
Middle: L1 cache miss rate
Bottom: Global miss rate

Takeaways:
Size of L2 should be > L1
Local miss rate not a good measure

17

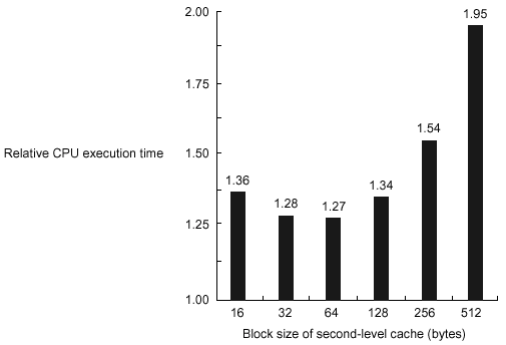
Size of L2?

- L2 should be bigger than L1
  - Everything in L1 likely to be in L2
  - If L2 is just slightly bigger than L1, lots of misses
- Size matters for L2, then..
  - Could use a large direct-mapped cache
    - Large size means few capacity misses, compulsory or conflict misses possible
  - Set associativity make sense?
    - Generally not, more expensive and can increase cycle time
  - Most L2 caches made as big as possible, size of main memory in older computers

18

L2 Cache Block Size

- Increased block size
  - Big block size increases chances for conflicts (fewer blocks in the cache), but not so much a problem in L2 if it's already big to start with
  - Sizes of 64-256 bytes are popular



19

L2 Cache Inclusion

- Should data in L1 also be in L2?
  - If yes, L2 has the *multilevel inclusion property*
  - This can be desirable to maintain consistency between caches and I/O; we could just check the L2 cache
  - Write through will support multilevel inclusion
- Drawback if yes:
  - "Wasted" space in L2, since we'll have a hit in L1
  - Not a big factor if L2 >> L1
  - Write back caches
    - L2 will need to "snoop" for write activity in L1 if it wants to maintain consistency in L2

20

Reducing Hit Time

- We've seen ways to reduce misses, and reduce the penalty.. next is reducing the hit time
- #1 Simplest technique: Small and Simple Cache
  - Small → Faster, less to search
  - Must be small enough to fit on-chip
    - Some compromises to keep tags on chip, data off chip but not used today with the shrinking manufacturing process
  - Use direct-mapped cache
    - Choice if we want an aggressive cycle time
    - Trades off hit time for miss rate, since set-associative has a better miss rate

21

#2 Virtual Caches

- Virtual Memory
  - Map a virtual address to a physical address or to disk, allowing a virtual memory to be larger than physical memory
  - More on virtual memory later
- Traditional caches or Physical caches
  - Take a physical address and look it up in the cache
- Virtual caches
  - Same idea as physical caches, but start with the virtual address instead of the physical address
  - If data is in the cache, it avoids the costly lookup to map from a virtual address to a physical address
    - Actually, we still need to the do the translation to make sure there is no protection fault
- Too good to be true?

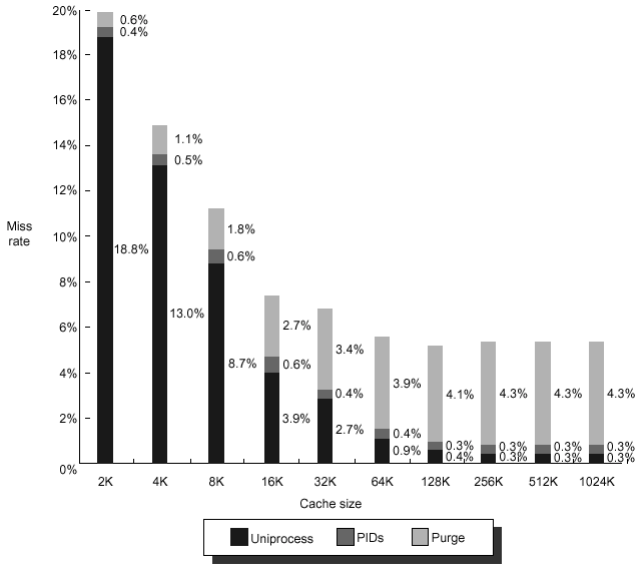
22

Virtual Cache Problems

- Process Switching
  - When a process is switched, the same virtual address from a previous process can now refer to a different physical addresses
    - Cache must be flushed
    - Too expensive to safe the whole cache and re-load it
    - One solution: add PID's to the cache tag so we know what process goes with what cache entry
  - Comparison of results and the penalty on the next slide

23

Miss Rates of Virtually Addressed Cache



24

More Virtual Cache Problems…

- Aliasing
  - Two processes might access different virtual addresses that are really the same physical address
  - Duplicate values in the virtual cache
  - Anti-aliasing hardware guarantees every cache block has a unique physical address
- Memory-Mapped I/O
  - Would also need to map memory-mapped I/O devices to a virtual address to interact with them
- Despite these issues…
  - Virtual caches used in some of today's processors
    - Alpha, HP…

25

#3 Pipelining Writes for Fast Hits

- Write hits take longer than read hits
  - Need to check the tags first before writing data to avoid writing to the wrong address
  - To speed up the process we can pipeline the writes (Alpha)
    - First, split up the tags and the data to address each independently
    - On a write, cache compares the tag with the write address
    - Write to the data portion of the cache can occur in parallel with a comparison of some other tag
      - We just overlapped two stages
    - Allows back-to-back writes to finish one per clock cycle
- Reads play no part in this pipeline, can already operate in parallel with the tag check

26

Cache Improvement Summary

Technique	Miss rate	Miss penalty	Hit time	Hardware complexity	Comment
Larger block size	+	–		0	Trivial; RS/6000 550 uses 128
Higher associativity	+		–	1	e.g., MIPS R10000 is 4-way
Victim caches	+			2	Similar technique in HP 7200
Pseudo-associative caches	+			2	Used in L2 of MIPS R10000
Hardware prefetching of instructions and data	+			2	Data are harder to prefetch; tried in a few machines; Alpha 21064
Compiler-controlled prefetching	+			3	Needs nonblocking cache too; several machines support it
Compiler techniques to reduce cache misses	+			0	Software is challenge; some machines give compiler option
Giving priority to read misses over writes		+		1	Trivial for uniprocessor, and widely used
Subblock placement		+		1	Used primarily to reduce tags
Early restart and critical word first		+		2	Used in MIPS R10000, IBM 620
Nonblocking caches		+		3	Used in Alpha 21064, R10000
Second-level caches		+		2	Costly hardware; harder if block size L1 ≠ L2; widely used
Small and simple caches	–		+	0	Trivial; widely used
Avoiding address translation during indexing of the cache			+	2	Trivial if small cache; used in Alpha 21064
Pipelining writes for fast write hits			+	1	Used in Alpha 21064