

# COSC 6385

## Computer Architecture

### - Memory Hierarchy Design (III)

Edgar Gabriel

Fall 2006



COSC 6385 – Computer Architecture  
Edgar Gabriel



# Reducing cache miss penalty

- Five techniques
  - Multilevel caches
  - Critical word first and early restart
  - Giving priority to read misses over writes
  - Merging write buffer
  - Victim caches



# Reducing miss rate

- Five techniques to reduce the miss rate
  - Larger cache block size
  - Larger caches
  - Higher associativity
  - Way prediction and pseudo-associative caches
  - Compiler optimization



# Reducing Cache Miss penalty/rate via Parallelism

- Nonblocking caches
- Hardware prefetch of Instructions and Data
- Compiler controlled prefetching



# Non-blocking caches

- Allow data cache to continue to supply cache hits during a cache miss
- Reduces effective miss penalty
- Further optimization: allow overlap of multiple misses
  - Only useful if the memory system can service multiple misses.
- Increases complexity of cache controller



# Hardware prefetching

- Prefetch items before they are requested by the processor
  - E.g. processor prefetches two blocks on a miss, assuming that the block following the missing data item will also be requested
  - Requested block is placed in the instruction cache
  - Prefetched block is placed in an instruction stream buffer
  - In case of a request to the prefetched block, the original cache request is cancelled by the hardware and the block is read from the stream buffer
- Downside: wasting memory bandwidth if block is not required



# Compiler-controlled prefetch

- Register prefetch: load value into register
- Cache prefetch: load data into cache
- *Faulty* and *non-faulting* prefetches: a non-faulting prefetch turns into a no-op in case of a virtual address fault/protection violation



# Compiler-controlled prefetch (II)

- Example:

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

- Assumptions:

- Each element of  $a$ ,  $b$  are 8 byte ( double precision fp)
- 8 KB direct mapped cache with 16 byte blocks
  - Each block can hold 2 consecutive values of either  $a$  or  $b$



# Compiler controlled prefetching (III)

- No. of cache misses for a:
  - Since a block contains  $a[i][j]$ ,  $a[i][j+1]$  every even value of  $j$  will lead to a cache miss and every odd value of  $j$  will be a cache hit
  - $3 \times 100/2 = 150$  cache misses because of a
- No. of cache misses because of b:
  - b does not benefit from spatial locality
  - Each element of b can be used 3 times (for  $i=0,1,2$ )
  - Ignoring cache conflicts, this leads to a
    - Cache miss for  $b[0][0]$  for  $i=0$
    - cache miss for every  $b[j+1][0]$  when  $i=0$
  - $1 + 100 = 101$  cache misses



# Compiler controlled prefetching (IV)

- Assuming that a cache miss takes 7 clock cycles
- Assuming that we are dealing with non-faulting prefetches
- Does not prevent cache misses for the first 7 iterations

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



# Compiler controlled prefetching (V)

- New number of cache misses:
  - 7 misses for  $b[0][0]$ ,  $b[1][0]$ , ... in the first loop
  - $7/2 = 4$  misses for  $a[0][0]$ ,  $a[0][2]$ , ... in the first loop
  - $7/2 = 4$  misses for  $a[1][0]$ ,  $a[1][2]$ , ... in the second loop
  - $7/2 = 4$  misses for  $a[2][0]$ ,  $a[2][2]$ , ... in the second loop
  - total number of misses: 19
  - reducing the number of cache misses by 232!
- A *write hint* could tell the cache: I need the elements of  $a$  in the cache for speeding up the writes. But since I do not read the element, don't bother to load the data really into the cache – it will be overwritten anyway!



# Reducing Hit time

- Small and simple caches
- Avoiding address translation
- Pipelined cache access
- Trace caches



# Small and simple caches

- Cache hit time depends on
  - Comparison of the tag and compare it to the address
  - No. of comparisons having to be executed
  - Clock-cycle of the cache
    - On-chip vs. off-chip caches
- Smaller and simpler caches are faster ‘by nature’



# Avoiding address translation

- Translation of virtual address to physical address required to access memory/caches
- Can a cache use virtual addresses?
  - In practice: no
    - Page protection has to be enforced, even if cache would use virtual addresses
    - Cache has to be flushed for every process switch
      - Two processes might use the same virtual address without meaning the same physical address
      - Adding of a process identifier tag (PID) to the cache address tag would solve parts of this problem
  - Operating systems might use different virtual addresses for the same physical address (aliases)
    - Could lead to duplicate copies of the same data in the cache



# Pipelined cache access and trace caches

- Pipelined cache access:
  - Reduce latency of first level cache access by pipelining
- Trace caches:
  - Find a dynamic sequence of instructions to load into a instruction cache block
  - The cache gets ‘dynamic traces of the executed instructions’ of the CPU and is not bound the spatial locality in the memory
  - Works well on taken branches
  - Used e.g. by Pentium 4



# Memory organization

- How can you improve memory bandwidth?
  - Wider main memory bus
  - Interleaved main memory
    - Try to take advantage of the fact that reading/writing multiple words use multiple chips (=banks)
    - Banks are typically one word wide
    - Memory interleaving tries to utilize all chips in the system
  - Independent memory banks
    - Allow independent access to different memory chips/banks
    - Multiple memory controllers allow banks to operate independently
      - Might required separate data buses, address lines
    - Usually required by non-blocking caches

