### Basic Cache Optimizations

- **Goal is to improve average memory access time.**
  - hit_time + miss_rate * miss_penalty
- **miss rate optimizations**
  - larger block size, bigger caches, higher associativity
- **miss penalty optimizations**
  - multi-level caches, giving priority to read misses before write misses.
- **reducing hit time**
  - virtual caches, virtually indexed and physically tagged caches

### Further Improving Caches

- It is also possible to improve cache bandwidth and energy usage.
- **advanced cache optimizations**
  - small and simple first-level caches
  - way prediction
  - pipelined cache access
  - nonblocking caches
  - multibanked caches
  - critical word first and early restart
  - merging write buffer
  - compiler optimizations
  - hardware prefetching
  - compiler controlled prefetching

### Concepts Introduced in Chapter 2

- advanced cache optimizations
- memory technology
- protection
- trends

### Cache Review

- **organizations**
  - direct mapped, fully associative, set associative
- **replacement policy**
  - random, LRU, FIFO
- **handling writes**
  - write policy
    - write through, write back
  - write miss policy
    - write allocate, no-write allocate
Small and Simple First-Level Caches

- Faster
  - Less propagation delay due to smaller size.
  - Direct-mapped organization allows tag check to overlap transmission of data to the CPU.
- More energy efficient
  - Smaller size results in both lower dynamic and static power.
  - Direct-mapped organization means fewer tag comparisons and fewer data words accessed on loads.

Way Prediction

- In each cache set are predictor bits indicating which block (way) in the set to check on the next access to that set.
- A way hit is about as fast as a direct-mapped organization.
- A way misprediction requires checking the other block(s) in the set on the subsequent cycle.
- Decreases misses as in a set-associative organization.
- Decreases power due to checking fewer tags and accessing fewer data words in parallel.
- Increases complexity as will have fast and slow hits.
- The miss penalty is increased by one cycle.

Pipelined Cache Access

- Can simply have the cache access pipeline stages (IF and MEM) take multiple cycles.
- Allows for a fast clock cycle time.
- Typically will result in a greater branch misprediction penalty and increased delay between a load and the first being able to use the data.
- Example of pipelining loads:

Nonblocking Data Caches

- Allows accesses for cache hits while a cache miss is being serviced on an out-of-order processor.
- Some caches allow hit under multiple misses, which implies the use of a queue of outstanding misses.
- Need a status bit for each block to indicate if it is being filled.
- Improves performance by reducing the effective miss penalty.
Multibanked Caches

- Divide the cache into independent banks to support simultaneous accesses.
- Usually sequential interleaving is applied where the block or word address modulo \( n \) is used to determine which of the \( n \) banks to access.
- Provides support for multiple issue processors for both instructions and data.
- At the L2 or lower levels can also support simultaneous access of instructions and data in the unified L2 cache.
- Reduces energy usage when only a single bank (as opposed to the entire cache) is accessed for a single instruction or data reference.

Critical Word First and Early Restart

- Critical word first means to request the missed word first from the next memory hierarchy level to allow the processor to continue while filling in the remaining words in the block, usually in a wrap-around fill manner.
- Early restart means to fetch the words in the normal order, but allow the processor to continue once the requested word arrives.
- Both approaches effectively reduce the miss penalty.
- Status bits must be used to indicate how much of the block has arrived.
- More beneficial for caches with large block sizes.

Merging Write Buffer

- Due to the latency penalty, multiword writes are faster than writing words in separate requests to the next memory hierarchy level.
- Multiple words in the write buffer can be associated with the same block address.
- Valid bits can be used to indicate if a word within the block should be written.
- Reduces the number of accesses to the next memory hierarchy level.
- Reduces the miss penalty due to fewer write buffer stalls for a given buffer size.

Compiler Optimizations

- Compiler optimizations can be used to reduce the cache miss rate.
- Improves spatial locality.
  - code positioning
  - array merging
  - loop interchange
- Improves temporal locality.
  - blocking
Code Positioning

- Permutes the order of the basic blocks in a program so that the most frequently executed transitions between basic blocks are fall-through transitions.
- Some processors have a penalty associated with each taken branch and this optimization can minimize this penalty.
- Also improves spatial locality, which can lower L1 IC miss rate.

![Diagram of code positioning](image)

(Array Merging)

- Merges multiple arrays together as an array of structs when corresponding elements are accessed near in time to improve spatial locality.

```c
/* Before */
int x[SIZE];
int y[SIZE];
for (i=0; i < 100; i++)
    sum += x[i] + y[i];

/* After */
struct merge {
    int x;
    int y;
} s[SIZE];
for (i=0; i < 100; i++)
    sum += s[i].x + s[i].y;
```

Loop Interchange

- Reorders loop iterations to improve spatial locality.

```c
/* Before */
for (j = 0; j < 100; j++)
    for (i = 0; i < 5000; i++)
        x[i][j] = x[i][j]*2;

/* After */
for (i = 0; i < 5000; i++)
    for (j = 0; j < 100; j++)
        x[i][j] = x[i][j]*2;
```

Blocking

- Operate on submatrices or blocks that fit into cache.

```c
for (i = 0; i < N; i++) /* Before */
    for (j = 0; j < N; j++) {
        r = 0;
        for (k = 0; k < N; k++)
            r += y[i][k]*z[k][j];
        x[i][j] = r;
    }

for (jj = 0; jj < N; jj += B) /* After */
    for (kk = 0; kk < N; 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 += y[i][k]*z[k][j];
                x[i][j] += r;
            }
```
Hardware Prefetching

- Idea is to fetch two blocks on a miss. The referenced block is loaded into a cache. The next consecutive block is prefetched into a stream buffer.
- A valid bit is set only after the prefetched block has been loaded into the buffer, which is after the missed block was loaded into the cache.
- On a miss the stream buffer is checked. Again, there would be a one cycle miss penalty if the item was not in cache, but in the buffer.
- A hit in the stream buffer causes it to be copied into the cache and another prefetch to occur.
- Prefetching exploits spatial locality.
- Can have multiple stream buffers. The stream buffer replaced is the one that is least recently accessed.

Compiler-Controlled Prefetching

- Idea is for the compiler to insert instructions into the program that will initiate prefetch operations.
  ```
  prefetch O(R1), #number_of_words
  ```
- Typical prefetch operation is to load data into cache, not a register, and to not fault if an exception would be generated by a conventional load from that address.
- The caches have to be nonblocking. This means that cache hits can be serviced while the prefetched data is being loaded.
- If the hit ratio was already very high, then the execution of the prefetch instructions may degrade performance.

SRAM

- SRAM - Static Random Access Memory
- Used in caches.
- Requires 6 transistors per bit to prevent data from being corrupted when read and requires 4 times the amount of space than DRAM.
- SRAM access time for the L3 cache is 3 to 5 times faster than DRAM.
- SRAM is perhaps 20 times more expensive than DRAM.

DRAM

- DRAM - Dynamic Random Access Memory
- Used for main memory.
- Requires a single transistor per bit, which is lost after being read, so each read requires that the data be written back.
- DRAM requires that the data be refreshed periodically, about 5% of the cycles.
- Access time - time between when a read is requested and the desired word arrives.
- Cycle time - minimum time between requests to memory.
- Address pins to DRAM chip are decreased by two by multiplexing the address lines.
- Row Access Strobe (RAS) - first half of the address is sent for the row
- Column Access Strobe (CAS) - second half of the address is sent for the column
**DRAM Optimizations**

- DRAMs allow repeated accesses to the same row without another RAS, which is called fast page mode.
- Synchronous DRAM (SDRAM) adds a clock signal to avoid overhead of synchronizing with the memory controller and allows a variable number of bytes to be sent over multiple cycles per memory request.
- DRAMs were made wider (in 2010 has 16-bit memory buses) to simultaneously transfer more bits.
- Double data rate (DDR) transfers data on both the rising and falling edge of the DRAM clock signal.
- SDRAMs introduced 2 to 8 banks that can operate independently and simultaneously service independent requests, which also reduces power.
- SDRAMs have a low power mode, which disables the SDRAM except for the internal refresh. Returning to an active power mode requires about 200 cycles.

**Flash Memory**

- Used as the backup storage for PMDs.
- Is nonvolatile and uses less power when in standby mode.
- Must be erased before written and is erased in large blocks, often 64KB to 256KB.
- There are a limited number of writes for each block, typically at least 100,000.
- Cheaper than SDRAM, but more expensive than disks ($20-$40/GB for SDRAM, $2/GB for Flash, $0.09/GB for disk).
- Read accesses are four times slower than SDRAM, but about 1000 times faster than disk.
- SDRAM is about 10 to 100 times faster that Flash for writes.

**Enhancing Dependability**

- Larger memories can have errors from the fabrication process or dynamic errors primarily from cosmic rays (soft errors).
- All DRAMs, Flash memory, and many SRAMs have spare rows to accommodate errors from manufacturing defects.
- Dynamic errors are detected by parity bits and fixed by error correcting codes (ECCs).

**Protection via Virtual Memory**

- Virtual memory is also used to provide process protection.
- Page protection bits (read, write, and execute access) are included in each page table (and TLB) entry.
- Only the OS is allowed to update the page table for a process.
- Can switch between a user (user mode) and the OS (kernel mode) by using a system call.
- Additional protection can be provided by virtual machines (VMs).
Where Does I/O Occur?

- between the I/O device and the cache
  - Solves the cache-coherency problem as the cache would always have the latest data.
  - Would interfere with the CPU.
- between the I/O device and main memory
  - Little interference with the CPU.
  - Need to resolve the stale data problem.

Introducing Stale Data

- Copies of data can be in the data cache(s), main memory, and disk (virtual memory).
- Stale data occurs when these copies are not consistent.
- Two problems with stale data:
  - CPU may not read the most recently input data since the cache may not be updated after an I/O input operation.
  - An I/O output operation may not have the correct data because memory is not up-to-date.

Solutions to Dealing with Stale Data

- Input
  - Not allow any I/O input buffers to be cachable.
  - Flush appropriate lines from cache that are to be updated from the I/O input operation.
- Output
  - Use a write-through cache.
  - Write back all dirty lines in cache before performing an I/O output operation.

Avoiding the Memory Wall

- Delays due to increasing memory latency have been avoided.
  - multiple levels of cache
  - prefetching of data and instructions
  - compiler optimizations to improve locality
  - use of out-of-order pipelines with nonblocking caches supporting multiple outstanding misses
  - use of multithreading and thread-level parallelism
Fallacies and Pitfalls

- Fallacy: Predicting cache performance of one program from another.
- Pitfall: Simulating enough instructions to get accurate performance measures of the memory hierarchy.