Memory and cache

CPU Memory I/O

The Memory Hierarchy

- ~2ns Registers
- ~4-5ns Primary cache
- ~30ns Secondary cache
- ~220ns+ Main memory
- >1ms (~6ms) Magnetic disk

Cache & Locality

- Cache sits between the CPU and main memory
  - Invisible to the CPU
  - Only useful if recently used items are used again
  - Fortunately, this happens a lot. We call this property locality of reference.

Locality of reference

- Temporal locality
  - Recently accessed data/instructions are likely to be accessed again.
    - Most program time is spent in loops
    - Arrays are often scanned multiple times
- Spatial locality
  - If I access memory address n, I am likely to then access another address close to n (usually n+1, n+2, or n+4)
    - Linear execution of code
    - Linear access of arrays

How a cache exploits locality

- Temporal - When an item is accessed from memory it is brought into the cache
  - If it is accessed again soon, it comes from the cache and not main memory
- Spatial - When we access a memory word, we also fetch the next few words of memory into the cache
  - The number of words fetched is the cache line size, or the cache block size for the machine

Cache write policies

- As long as we are only doing READ operations, the cache is an exact copy of a small part of the main memory
- When we write, should we write to cache or memory?
- Write through cache – write to both cache and main memory. Cache and memory are always consistent
- Write back cache – write only to cache and set a “dirty bit”. When the block gets replaced from the cache, write it out to memory.

When might the write-back policy be dangerous?
Cache mapping

- Direct mapped – each memory block can occupy one and only one cache block
- Example:
  - Cache block size: 16 words
  - Memory = 64K (4K blocks)
  - Cache = 2K (128 blocks)

Direct Mapped Cache

- Memory block $n$ occupies cache block ($n$ mod 128)
- Example:
  - Cache block size: 16 words
  - Memory = 64K (4K blocks)
  - Cache = 2K (128 blocks)

- Consider address $2EF4$
  - $0010111011110100$
  - block: $2EF = 751$ word: 4
- Cache:
  - $0010111011110100$
  - tag: 5 block: 111 word: 4

Fully Associative Cache

- More efficient cache utilization
  - No wasted cache space
  - Slower cache search
  - Must check the tag of every entry in the cache to see if the block we want is there.

Set-associative mapping

- Blocks are grouped into sets
- Each memory block can occupy any block in its set
- This example is 2-way set-associative
- Which of the two blocks do we replace?

Replacement algorithms

- Random
- Oldest first
- Least accesses
- Least recently used (LRU): replace the block that has gone the longest time without being referenced.
  - This is the most commonly used replacement algorithm
  - Easy to implement with a small counter...

Implementing LRU replacement

- Suppose we have a 4-way set associative cache...
  - Hit:
    - Increment lower counters
    - Reset counter to 00
  - Miss:
    - Replace the 11
    - Set to 00
    - Increment all other counters
Interactions with DMA

- If we have a write-back cache, DMA must check the cache before acting.
  - Many systems simply flush the cache before DMA write operations
- What if we have a memory word in the cache, and the DMA controller changes that word?
  - Stale data
  - We keep a valid bit for each cache line. When DMA changes memory, it must also set the valid bit to 0 for that cache line.
  - "Cache coherence"

Typical Modern Cache Architecture

- L0 cache
  - On chip
  - Split 16 KB data/16 KB instructions
- L1 cache
  - On chip
  - 64 KB unified
- L2 cache
  - Off chip
  - 128 KB to 16+ MB

Measuring Cache Performance

- No cache: Often about 10 cycles per memory access
- Simple cache:
  - \( t_{\text{ave}} = hC + (1-h)M \)
  - \( C \) is often 1 clock cycle
  - Assume \( M \) is 17 cycles (to load an entire cache line)
  - Assume \( h \) is about 90%
  - \( t_{\text{ave}} = 9(1) + (1).17 = 2.6 \text{ cycles/access} \)
  - What happens when \( h \) is 95%?

Multi-level cache performance

- \( t_{\text{ave}} = h_1C_1 + (1-h_1)h_2C_2 + (1-h_2)h_3M \)
  - \( h_1 \) = hit rate in primary cache
  - \( h_2 \) = hit rate in secondary cache
  - \( C_1 \) = time to access primary cache
  - \( C_2 \) = time to access secondary cache
  - \( M \) = miss penalty (time to load an entire cache line from main memory)

Memory Interleaving

- Memory is organized into modules or banks
- Within a bank, capacitors must be recharged after each read operation
- Successive reads to the same bank are slow

Virtual Memory
Virtual Memory: Introduction

- Motorola 68000 has 24-bit memory addressing and is byte addressable
  - Can address 2^24 bytes of memory = 16 MB
- Intel Pentiums have 32-bit memory addressing and are byte addressable
  - Can address 2^32 bytes of memory = 4 GB
- What good is all that address space if you only have 256MB of main memory (RAM)?
  - How do you run a program that is LARGER than 256MB?
  - Examples:
    - Unreal Tournament - 2.4 gigabytes
    - Neverwinter Nights - 2 gigabytes

Not all of a program needs to be in memory while you are executing it:
- Error handling routines are not called very often.
- Character creation generally only happens at the start of the game... why keep that code easily available?

Examples:
- Unreal Tournament - 100MB (requires 128MB)
- Internet Explorer - 20MB

Why does paging work?
- Locality model
  - Process migrates from one locality to another.
  - Localities may overlap.

Why does thrashing occur?
- Size of locality > total memory size

What should we do?
- Suspend processes!

If a process does not have enough frames to hold its current working set, the page-fault rate is very high

Thrashing
- a process is thrashing when it spends more time paging than executing
- w/ local replacement algorithms, a process may thrash even though memory is available
- w/ global replacement algorithms, the entire system may thrash
- Less thrashing in general, but is it fair?

How should we arrange memory references to large arrays?
- Is the array stored in row-major or column-major order?

Example:
- Array A[1024, 1024] of type integer
  - Page size = 1K
  - Each row is stored in one page
- System has one frame
  - Program 1
    for i = 1 to 1024 do
    for j = 1 to 1024 do
      A[i][j] := 0;
  - 1024 page faults
  - Program 2
    for j = 1 to 1024 do
    for i = 1 to 1024 do
      A[i][j] := 0;
  - 1024 x 1024 page faults

Memory Matters
- Memory is not unbounded
  - It must be allocated and managed
  - Many applications are memory dominated
- Memory referencing bugs especially pernicious
  - Effects are distant in both time and space
  - Memory performance is not uniform
    - Cache and virtual memory effects can greatly affect program performance
    - Adapting program to characteristics of memory system can lead to major speed improvements
Memory Referencing Errors

- C and C++ do not provide any memory protection
  - Out of bounds array references
  - Invalid pointer values
  - Abuses of malloc/free
- Can lead to nasty bugs
  - Whether or not bug has any effect depends on system and compiler
  - Corrupted object logically unrelated to one being accessed
  - Effect of bug may be first observed long after it is generated

How can I deal with this?
- Program in Java, Lisp, or ML
- Understand what possible interactions may occur
- Use or develop tools to detect referencing errors

Memory Performance Example

- Implementations of Matrix Multiplication
  - Multiple ways to nest loops

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

```
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Matmult Performance

```
0 20 40 60 80 100 120 140 160
matrix size (n)
```

```
too big for L1 Cache
too big for L2 Cache
```

IA32 and GCC

- IA32 processors, like most other processors, have special memory elements (registers) for holding floating-point values as they are being computed and used.
- IA32 uses special 80-bit extended precision floating-point registers
- IA32/gcc stores doubles as 64 bits in memory. Numbers are converted as they are stored in memory
- Whenever a function call is made, register values may be stored in memory (callee save)

```
double recip (int denom) {
  return 1.0/(double) denom;
}
void do_nothing () {};
void test (int denom) {
  double r1, r2;
  int t1, t2;
  r1 = recip(denom);
  r2 = recip(denom);
  if (r1 == r2)
    do_nothing();
  t1 = (r1 == r2);
  do_nothing();
  t2 = (r1 == r2);
  printf("t1: r1 %c= r2 \n, t1 ? '=' : '!'\n");
  printf("t2: r1 %c= r2 \n, t2 ? '=' : '!'\n");
}
```

```
t1: r1 != r2;
t2: r1 = = r2;
```