Introduction to Computer Systems

15-213/18-243, Spring 2009

Recitation 6 (performance), March 2nd

Agenda

- **Announcements**
- **Performance review**
  - Program optimization
  - Memory hierarchy and caches
  - With hints on perf lab

Announcements

- **Exam 1**
  - Graded exams were returned in class on Thursday
  - Statistics
  - If you find a grading error
    - Follow the procedure on page 6 of the syllabus
- **Labs**
  - Perf lab
    - Due March 5 (this Thursday)
  - Shell lab
    - Out immediately after perflab
      - Official start date is March 17
      - But can work on it over spring break if you want
    - Due March 31

Performance Review

- **Program optimization**
  - Efficient programs are the result of two things
    - Good algorithms and data structures
    - Code that the compiler can effectively optimize and turn into efficient executable
      - How to write such code is the topic of program optimization
  - Modern compilers use sophisticated techniques to detect and exploit opportunities for optimization
    - However,
      - their ability to understand code is limited, and
      - they are conservative
  - Programmer can greatly influence the compiler's ability to optimize

Optimization Blockers

- **Procedure calls**
  - Compilers' ability to perform inter-procedural optimizations is limited
  - Solution: replace by procedure body
    - Perhaps comes at cost in program modularity
      - Inlining and using macros help mitigate this
    - Can result in much faster programs
- **Loop invariants**
  - Expressions that do not change in loop body
  - Solution: code motion

Optimization Blockers (cont.)

- **Memory aliasing**
  - Accessing memory can have side effects
    - Difficult for compiler to analyze
  - Two different memory reference expressions can reference the same memory location (i.e. alisases)
    - Very difficult for compiler to detect aliases
  - Solution: scalar replacement
    - Copy elements that are reused into temporary variables, operate on them, then store result back
    - Basic scheme:
      - Load: tmp_var1 ← *ptr1; tmp_var2 ← *ptr2; ...
      - Compute: tmp_var1 ← tmp_var1 OP tmp_var2; ...
      - Store: *ptr1 = tmp_var1;
    - Particularly important if memory references are in inner most loop (e.g. in perflab!)

Loop Unrolling

- **A technique for reducing loop overhead**
  - Perform more data operations in single iteration
    - E.g. access and compile multiple array elements in each iteration
  - Resulting program has fewer iterations, which translates into fewer condition checking and jumps
  - Enables more aggressive scheduling of loops
  - However, too much unrolling is bad
    - Results in larger code
    - Code may not fit in instruction cache

Others

- Out of order processing
- Branch prediction
- Can be used in perf lab, but less crucial

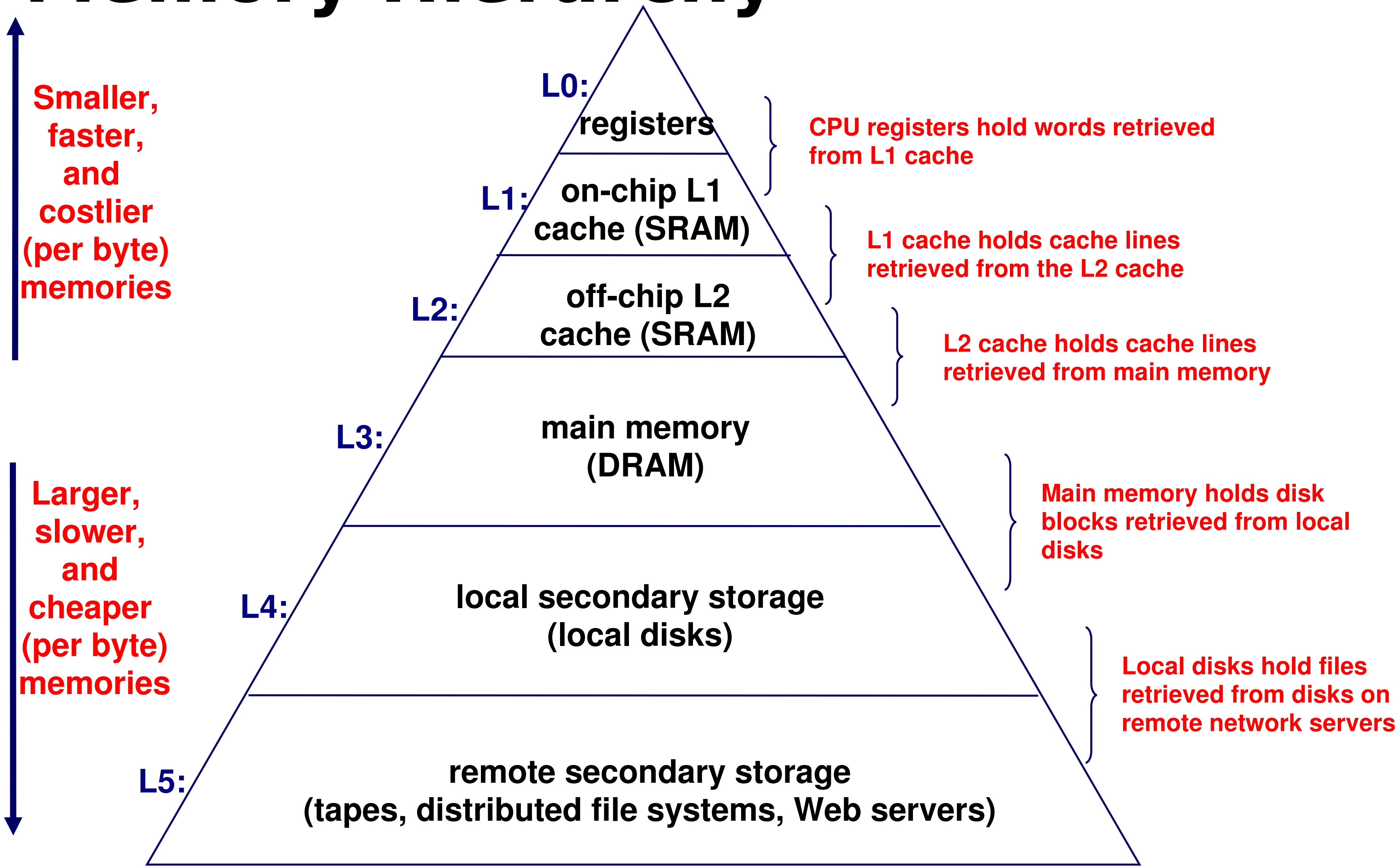
Caches

- **Definition**
  - Memory with short access time
  - Used for storage of “frequently” or “recently” used instructions or data
- **Concepts**
  - Hits, misses
- **Performance metrics**
  - Hit rate, miss rate (commonly used), miss penalty
- **Types of misses**
  - Compulsory: due to “cold” cache (happens at beginning)
  - Conflict: when referenced data map to the same block
  - Capacity: when “working set” is larger than cache

Locality

- **Programs tend to use data and instructions with addresses near or equal to those they have recently used**
- **Reason why caches work**
- **Two types**
  - Temporal
  - Spatial

Memory Hierarchy



Cache Miss Analysis Exercise

- **Assume:**
  - Cache blocks are 16-byte
  - Only memory accesses are to the entries of grid; index variables are stored in registers
- **Determine the cache performance of the following**

```
struct algae_position {
  int x;
  int y;
};

struct algae_position grid[16][16];
int total_x = 0, total_y = 0;
int i, j;
```

```
for (i = 0; i < 16; i++) {
  for (j = 0; j < 16; j++)
    total_x += grid[i][j].x;
}

for (i = 0; i < 16; i++) {
  for (j = 0; j < 16; j++)
    total_y += grid[i][j].y;
}
```

Techniques for Increasing Locality

- **Rearranging loops**
  - Increases spatial locality
  - Important in perf lab
  - Analyze the cache miss rate for the following assuming
    - Array elements are doubles
    - Cache line is 32 bytes

```
void ijk(array A, array B, array C,
         int n)
{
    int i, j, k;
    double sum;

    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;
      }
}
```

```
void kij(array A, array B, array C,
         int n)
{
    int i, j, k;
    double r;

    for (k = 0; k < n; k++)
      for (i = 0; i < n; i++) {
         r = A[i][k];
         for (j = 0; j < n; j++)
             C[i][j] += r*B[k][j];
      }
}
```

Techniques for Increasing Locality (cont.)

- **Blocking**
  - Increases temporal locality
  - Important in perf lab
  - Analyze the cache miss rate for the following assuming
    - Array elements are doubles
    - Cache line size is 32 bytes (i.e. can hold 4 doubles)

```
void naiive(array a, array b, array c,
            int N)
{
  int i, j, k;

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

```
void blocking(array a, array b, array c,
              int N, int B)
{
  int i, j, k;

  for (i = 0; i < N; i += B)
   for (j = 0; j < N; j += B)
    for (k = 0; k < N; k += B)
     for (i1 = i; i1 < (i + B); i1++)
      for (j1 = j; j1 < (j + B); j1++)
       for (k1 = k; k1 < (k + B); k1++)
        c[i1][j1] += a[i1][k1] * b[k1][j1];
}
```

Questions?

- Exam 1
- Program optimization
- Writing cache friendly code
- Perf lab