Introduction To Computer Systems

15-213/18-243, Spring 2011
Recitation 7 (performance)
Monday, February 21
Agenda

• Performance review
  • Program optimization
  • Memory hierarchy and caches
Performance Review

• Program optimization
  • Efficient programs result from:
    • Good algorithms and data structures
    • Code that the compiler can effectively optimize and turn into efficient executable
  • The topic of program optimization relates to the second
Performance Review (cont)

• Modern compilers use sophisticated techniques to optimize programs
• However,
  • Their ability to understand code is limited
  • They are conservative
• Programmer can greatly influence compiler’s ability to optimize
Optimization Blockers

• Procedure calls
  • Compiler’s ability to perform inter-procedural optimization is limited
  • Solution: replace call by procedure body
    • Can result in much faster programs
    • Inlining and macros can help preserve modularity

• Loop invariants
  • Expression that do not change in loop body
  • Solution: code motion
Optimization Blockers (cont)

• Memory aliasing
  • Accessing memory can have side effects difficult for the compiler to analyze (e.g., aliasing)
  • Solution: scalar replacement
    • Copy elements into temporary variables, operate, then store result back
    • Particularly important if memory references are in innermost loop
Loop Unrolling

• A technique for reducing loop overhead
  • Perform more data operations in single iteration
  • Resulting program has fewer iterations, which translates into fewer condition checks and jumps
  • Enables more aggressive scheduling of loops
  • However, too much unrolling can be bad
    • Results in larger code
    • Code may not fit in instruction cache
Other Techniques

• Out of order processing
• Branch prediction
• Less crucial in this class
Caches

• Definition
  • Memory with short access time
  • Used for storage of frequently or recently used instructions or data

• Performance metrics
  • Hit rate
  • Miss rate (commonly used)
  • Miss penalty
Cache Misses

• Types of misses
  • Compulsory: due to cold cache (happens at beginning)
  • Conflict: When referenced data maps to the same block
  • Capacity: when working set is larger than cache
Locality

• Reason why caches work
• Temporal locality
  • Programs tend to use the same data and instructions over and over
• Spatial locality
  • Programs tend to use data and instructions with addresses near to those they have recently used
Memory Hierarchy
Cache Miss Analysis Exercise

• Assume:
  – Cache blocks are 16-byte
  – Only memory accesses are to the entries of grid

• Determine the cache performance of the following:

```c
struct algae_position {
    int x;
    int y;
};
struct algae_position_grid[16][16];
int total_x = 0, total_y = 0, 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)
• Analyze the cache miss rate for the following:
  • Assume 32-byte lines, array elements are doubles

```c
void ijk(A[], B[], C[], n) {
   int i, j, k; double sum;
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         sum += A[i][k]*B[k][j]
      C[i][j] += sum
}

void kij(A[], B[], C[], 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)
• Analyze the cache miss rate for the following:
  • Assume 32-byte lines, array elements are doubles

```c
void naive(A[], B[], C[], 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 (A[], B[], C[], n, b) {
    int i, j, k, i1, j1, k1;
    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?

- Program optimization
- Writing friendly cache code
- Cache lab