

CMSC 216
Introduction to Computer Systems
Lecture 26
Optimizing Performance

Jan Plane & Alan Sussman
{jplane, als}@cs.umd.edu

Administrivia

- Project 6 due Friday
 - questions?
- Practice final exam posted, with answers soon
- Read Chapter 14 of Reek
- Please do course evaluation, at www.CourseEvalUM.umd.edu

CMSC 216 - Wood, Sussman, Herman

2

Chapter 5, Bryant & O'Hallaron

OPTIMIZING PROGRAM PERFORMANCE

CMSC 216 - Wood, Sussman, Herman

3

Issues in conducting measurements

- Number of runs:
 - a single run of a program to time its performance is not sufficient - many things go on in a computer, such as operating system functions and other programs running
 - multiple runs provide increased accuracy
 - take the mean of the K fastest runs
- Workload:
 - what data is the program given for measurement runs- does it look like a "typical" use of the program?
 - many algorithms might look good if the measured workload is too small – for small n , $O(n^2)$ algorithms are similar to $O(n)$
 - many algorithms might also look good if the measured workload is too large (not representative of the typical usage pattern)

CMSC 216 - Wood, Sussman, Herman

4

Sources of performance problems

- I/O operations that are too small
 - reading one or a few characters at a time
- Debugging `printf`'s were left in the program
- Poor basic algorithms
 - $O(n^2)$ algorithms were used in cases where n was large, or for frequently performed operations
- Algorithms were used that compose poorly
- Cache memory performance
 - Reuse each cached item at least once if possible

CMSC 216 - Wood, Sussman, Herman

5

Improving code performance

- This is **not** about algorithms, but rather:
 - how to convert algorithms into efficient code
 - how to refine code to make it run faster
- The key ideas are:
 - be systematic and data-driven in what you do (it's easy to just make random code changes without improving things)
 - programs spend their time in loops or recursion (a sequence of code executed once is fast), and in doing I/O

CMSC 216 - Wood, Sussman, Herman

6

Know your compiler

- Compilers can be asked to "optimize" your code
 - the **gcc** `-O` flag (and `-ON`) enables the optimizer, which makes modifications to the compiled program if possible
 - optimization is supposed to never break your code—only safe changes to the program are permitted
 - optimization should never alter correct programs
 - but it may discover **latent** bugs
- Limits on compiler optimizations:
 - the compiler has a limited understanding of the program
 - there's a need to compile programs quickly

CMSC 216 - Wood, Sussman, Herman

7

Sample compiler optimization

- Original code:


```
int i, j, k;
...
for (i = 0; i < 200; i++)
    a[2 * j + i] = j * k + i;
```
- Optimized code:


```
int i, j, k;
...
int prod_2j, prod_jk;
prod_2j = 2 * j;
prod_jk = j * k;
for (i = 0; i < 200; i++)
    a[prod_2j + i] = prod_jk + i;
```

CMSC 216 - Wood, Sussman, Herman

8

Typical compiler optimizations

- Code motion
 - previous example
- Loop unrolling
 - things can often be faster if a loop body is a bit bigger, because modern processors can perform several instructions (or at least parts of them) simultaneously
 - convert:

```
for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```
 - to:

```
for (i = 0; i < n; i += 2) {
    c[i] = a[i] + b[i];
    c[i + 1] = a[i + 1] + b[i + 1];
}
```
- Dead code elimination
 - if the compiler can infer that some code can never be executed, the code can be removed

CMSC 216 - Wood, Sussman, Herman

9

Limits to compiler optimization

- Consider:


```
void func1(int *xp, int *yp) {
    *xp += *yp;
    *xp += *yp;
}
```

Could this be replaced by the following version?

```
void func1(int *xp, int *yp) {
    *xp += 2 * *yp;
}
```
- If a function ends with


```
return f(x) + f(x) + f(x) + f(x);
```

 can it be simplified to the following?


```
return 4 * f(x);
```

CMSC 216 - Wood, Sussman, Herman

10

Common code changes- loop bound checks

- Consider:


```
for (i = 0; i < strlen(data); i++)
    if (data[i] >= 'a' && data[i] <= 'z')
        data[i] -= ('a' - 'A');
```
- **strlen()** is called each time, but its result doesn't change inside the loop
 - this results in $O(n^2)$ performance
- Change to:


```
int difference = 'a' - 'A';
int limit = strlen(data);
for (i = 0; i < limit; i++)
    if (data[i] >= 'a' && data[i] <= 'z')
        data[i] -= difference;
```
- A compiler may not be able to figure the **strlen()** optimization out

CMSC 216 - Wood, Sussman, Herman

11

Common code changes, con't.

- Recall:


```
int difference = 'a' - 'A';
int limit = strlen(data);
for (i = 0; i < limit; i++)
    if (data[i] >= 'a' && data[i] <= 'z')
        data[i] -= difference;
```
- The **strlen()** call can be eliminated entirely:


```
int difference = 'a' - 'A';
ptr = data;
while (*ptr != '\0') {
    if (*ptr >= 'a' && *ptr <= 'z')
        *ptr -= difference;
    ptr++;
}
```

CMSC 216 - Wood, Sussman, Herman

12

Common code transformation- don't write small functions

- Functions calls take time
 - to set up the runtime stack when calling a function, and to return when done
- Small functions of one or two lines are not that useful:
 - they take more time to set up things for the call and to return afterward, than the time it takes to perform the function's instructions
 - they also provide relatively little isolation of ideas
- Try to define abstractions so the work to be done is larger than a few lines
- Sometimes you can use macros in place of small functions

CMSC 216 - Wood, Sussman, Herman

13

Approach to applying these techniques

- For a large program it's hard to know where to start to optimize
- Start with **gprof** data and concentrate on the parts of the program that take the most time
- Amdahl's Law:
 - if you're tuning a function that takes α fraction (%) of the program execution time T
 - and you can make it k times faster
 - $T_{\text{new}} = (1 - \alpha) T_{\text{old}} + (\alpha T_{\text{old}})/k$
 - you're basically limited by how big α is

CMSC 216 - Wood, Sussman, Herman

14

Types of optimizations

- Optimizations you should do regardless of the processor/compiler:
 - code motion (moving code out of a loop)
 - reducing function calls (e.g., inlining)
 - reduce/remove unneeded memory usage
 - share common subexpressions
- Machine-dependent optimization
 - pointer code
 - loop unrolling
 - enabling instruction-level parallelism

CMSC 216 - Wood, Sussman, Herman

15

Storage and access order matters

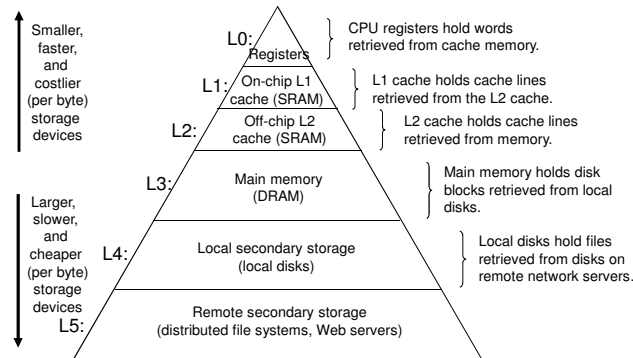
- The layout of data items in memory means that if we access things in a different order, we can get wildly different execution times
- Memory hierarchy:

Type of storage	Access time (CPU cycles)
registers	1
cache (1 st level)	~2-3
memory	~10-100

CMSC 216 - Wood, Sussman, Herman

16

More detailed memory hierarchy



CMSC 216 - Wood, Sussman, Herman

17

Two kinds of locality

- Principle of locality:
 - programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves
 - **temporal locality**: recently referenced items are likely to be referenced again in the near future
 - **spatial locality**: items with nearby addresses tend to be referenced close together in time

CMSC 216 - Wood, Sussman, Herman

18

Locality example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- Data
 - reference array elements in succession (in the same order they're stored in memory)
 - reference **sum** each iteration
- Instructions
 - reference instructions in sequence
 - cycle through loop repeatedly

CMSC 216 - Wood, Sussman, Herman

19

Aside: Multidimensional array storage

- In C, array elements are stored in row major order in memory, meaning each row's elements are stored in adjacent locations in order by column subscript, and all of the rows are adjacent in memory, stored in order by row subscript.

```
int a[2][3] = {{1, 3, 5}, {2, 4, 6}};
```

conceptually →

1	3	5
2	4	6

memory layout →

1	3	5	2	4	6
---	---	---	---	---	---

CMSC 216 - Wood, Sussman, Herman

20

A dramatic locality example

```
#define ROWS 20000
#define COLS 20000

int main() {
    int arr[ROWS][COLS];
    int i, j, sum = 0;
    struct rusage usage1, usage2;

    for (i = 0; i < ROWS; i++)
        for (j = 0; j < COLS; j++)
            arr[i][j] = rand();
    getrusage(RUSAGE_SELF, &usage1);
    for (i = 0; i < ROWS; i++)
        for (j = 0; j < COLS; j++)
            sum += arr[i][j];
    getrusage(RUSAGE_SELF, &usage2);
    printf("sum = %d\n", sum);
    /* calculate user time, store in usage2 */
    printf("User time %d.%06ds\n", (int) usage2.ru_utime.tv_sec,
           (int) usage2.ru_utime.tv_usec);
    return 0;
}
```

CMSC 216 - Wood, Sussman, Herman

21

A dramatic locality example

```
#define ROWS 20000
#define COLS 20000

int main() {
    int arr[ROWS][COLS];
    int i, j, sum = 0;
    struct rusage usage1, usage2;

    for (i = 0; i < ROWS; i++)
        for (j = 0; j < COLS; j++)
            arr[i][j] = rand();
    getrusage(RUSAGE_SELF, &usage1);
    for (j = 0; j < COLS; j++)
        for (i = 0; i < ROWS; i++)
            sum += arr[i][j];
    getrusage(RUSAGE_SELF, &usage2);
    printf("sum = %d\n", sum);
    /* calculate user time, store in usage2 */
    printf("User time %d.%06ds\n", (int) usage2.ru_utime.tv_sec,
           (int) usage2.ru_utime.tv_usec);
    return 0;
}
```

CMSC 216 - Wood, Sussman, Herman

22

Other caching optimizations

- Memoization
 - we can store intermediate results to speed up our calculations
 - C requires us to do this explicitly (some languages have support for doing this automatically)

CMSC 216 - Wood, Sussman, Herman

23

Standard Fibonacci code

```
#include <stdio.h>
#include <stdlib.h>

/* Return the nth Fibonacci number
 * (fib(0) = fib(1) = 1)
 */
unsigned long fib(unsigned int n) {
    if (n == 0 || n == 1)
        return 1;
    return fib(n-1) + fib(n-2);
}

int main(int argc, char *argv[]) {
    int num = atoi(argv[1]);
    printf("%lu\n", fib(num));
    return 0;
}
```

CMSC 216 - Wood, Sussman, Herman

24

Pruning the recursion tree

- This version of the Fibonacci function is quite slow
- We recompute values we've already computed!
- If we cache these computed values explicitly, we can speed things up dramatically

Memoized Fibonacci code

```
#include <stdio.h>
#include <stdlib.h>

/* Return the nth Fibonacci number
 * (fib(0) = fib(1) = 1)
 */
unsigned long fib(unsigned int n) {
    static unsigned int table[100] = {0};
    if (n == 0 || n == 1)
        return 1;
    if (table[n])
        return table[n];
    return table[n] = fib(n-1) + fib(n-2);
}

int main(int argc, char *argv[]) {
    int num = atoi(argv[1]);
    printf("%lu\n", fib(num));
    return 0;
}
```