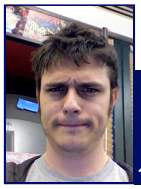
inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture #39
Writing *Really* Fast Programs



2008-5-2

Disheveled TA Casey Rodarmor

inst.eecs.berkeley.edu/~cs61c-tc

Scientists create Memristor, missing fourth circuit element

May be possible to create storage with the speed of RAM and the persistence of a hard drive, utterly pwning both.

http://blog.wired.com/gadgets/2008/04/scientists-proj.html

Speed

- Fast is good!
- But why is my program so slow?
  - Algorithmic Complexity
  - Number of instructions executed
  - Architectural considerations
- We will focus on the last two – take CS170 (or think back to 61B) for fast algorithms

Minimizing number of instructions

- Know your input: If your input is constrained in some way, you can often optimize.
  - Many algorithms are ideal for large random data
    - Often you are dealing with smaller numbers, or less random ones
    - When taken into account, "worse" algorithms may perform better
- Preprocess if at all possible: If you know some function will be called often, you may wish to preprocess
  - The fixed costs (preprocessing) are high, but the lower variable costs (instant results!) may make up for it.

Example 1 – bit counting – Basic Idea

- Sometimes you may want to count the number of bits in a number:
  - This is used in encodings
  - Also used in interview questions
- We must somehow 'visit' all the bits, so no algorithm can do better than O(n), where n is the number of bits
- But perhaps we can optimize a little!

Example 1 – bit counting - Basic

- The basic way of counting:

```
int bitcount_std(uint32_t num) {
    int cnt = 0;

    while(num) {
        cnt += (num & 1);
        num >>= 1;
    }

    return cnt;
}
```


Example 1 – bit counting – Optimized?

- The "optimized" way of counting:
- Still O(n), but now n is # of 1's present

```
int bitcount_op(uint32_t num) {
    int cnt = 0;
    while(num) {
        cnt++;
        num &= (num - 1);
    }
    return cnt;
}
```

This relies on the fact that
num = (num – 1) & num
changes rightmost 1 bit in num to a 0.
Try it out!

Example 1 – bit counting – Preprocess

- **Preprocessing!**

```
uint8_t tbl[256];

void init_table() {
    for(int i = 0; i < 256; i++)
        tbl[i] = bitcount_std(i);
}

// could also memoize, but the additional
// branch is overkill in this case
```

Example 1 – bit counting – Preprocess

- **The payoff!**

```
uint8_t tbl[256];// tbl[i] has number of 1's in i

int bitcount_preprocess(uint32_t num) {
    int cnt = 0;
    while(num) {
        cnt += tbl[num & 0xff];
        num >>= 8;
    }
    return cnt;
}
```

The table could be made smaller or larger; there is a trade-off between table size and speed.

Example 1 – Times

Test: Call bitcount on 20 million random numbers. Compiled with -O1, run on 2.4 Ghz Intel Core 2 Duo with 1 Gb RAM

Test	Totally Random number time	Random power of 2 time
Bitcount_std	830 ms	790 ms
Bitcount_op	860 ms	273 ms
Bitcount_ preprocess	720 ms	700 ms

Preprocessing improved (13% increase). Optimization was great for power of two numbers.

With random data, the linear in 1's optimization actually hurt speed (subtracting 1 may take more time than shifting on many x86 processors).

Profiling demo

- Can we speed up my old 184 project?
- It draws a nicely shaded sphere, but it's slow as a dog.
- Demo time!

Profiling analysis

- Profiling led us right to the touble spot
- As it happened, my code was pretty inefficient
- Won't always be as easy. Good forensic skills are a must!

Administrivia

- Lab14 + Proj3 grading. Oh, the horror.
- Project 4 Due yesterday at 11:59pm
- Performance Contest submissions due May 9th
  - No using slip days!

Inlining

- **A function in C:**

```
int foo(int v){
      // do something freaking sweet!
}
foo(9)
```

- **The same function in assembly:**

```
foo: # push back stack pointer
     # save regs
     # do something freaking sweet!
     # restore regs
     # push forward stack pointer
     jr $ra
#elsewhere
     jal foo
```

Inlining - Etc

- Calling a function is expensive!
- C provides the inline command
  - Functions that are marked inline (e.g. inline void f) will have their code inserted into the caller
  - A little like macros, but without the suck
- With inlining, bitcount-std took 830 ms
- Without inlining, bitcount-std took 1.2s!
- Bad things about inlining:
  - Inlined functions generally cannot be recursive.
  - Inlining large functions is actually a bad idea. It increases code size and may hurt cache performance

Sorting algorithms compared

Quicksort vs. Radix sort!

- **QUICKSORT – O(N*log(N)):**

  Basically selects "pivot" in an array and rotates elements about the pivot

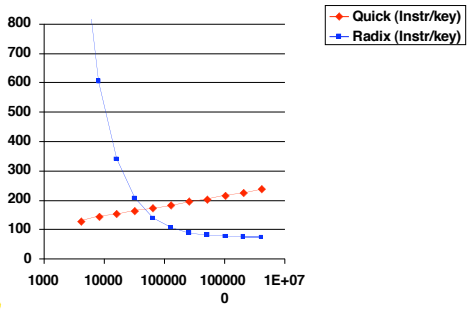
  Average Complexity: O(n*log(n))

RADIX SORT – O(n):

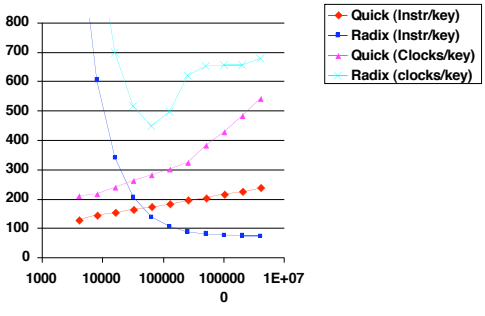
  Advanced bucket sort

  Basically "hashes" individual items.

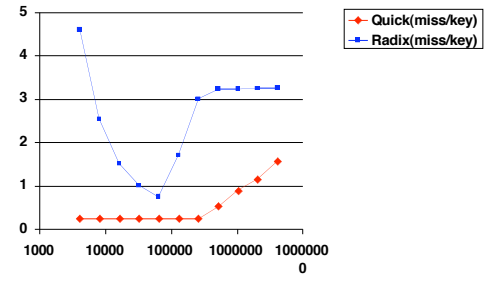
Complexity holds true for instruction count



Yet CPU time suggests otherwise…



Never forget Cache effects!



Other random tidbits

- Approximation: Often an approximation of a problem you are trying to solve is good enough – and will run much faster
  - For instance, cache and paging LRU algorithm uses an approximation
- Parallelization: Within a few years, all manufactured CPUs will have at least 4 cores. Use them!
- Instruction Order Matters: There is an instruction cache, so the common case should have high spatial locality
  - GCC's –O2 tries to do this for you
- Test your optimizations. You generally want to time your code and see if your latest optimization actually has improved anything.
  - Ideally, you want to know the *slowest* area of your code.
- Don't over-optimize! There is no reason to spend 3 extra months on a project to make it run 5% faster.

Case Study - Hardware Dependence

- You have two integers arrays A and B.
- You want to make a third array C.
- C consists of all integers that are in both A and B.
- You can assume that no integer is repeated in either A or B.

A	6	4	8
B	8	3	4
C	8	4	

Case Study - Hardware Dependence

- You have two integers arrays A and B.
- You want to make a third array C.
- C consists of all integers that are in both A and B.
- You can assume that no integer is repeated in either A or B.
- There are two reasonable ways to do this:
  - Method 1: Make a hash table.
    - Put all elements in A into the hash table.
    - Iterate through all elements n in B. If n is present in A, add it to C.
  - Method 2: Sort!
    - Quicksort A and B
    - Iterate through both as if to merge two sorted lists.
    - Whenever A[index_A] and B[index_B] are ever equal, add A[index_A] to C

Peer Instruction

Method 1 – Make a hash table.
Put all elements in A into the hash table.
Iterate through all elements n in B. If n is in A, add it to C

Method 2 – Sort!
Quicksort A and B
Iterate through both as if to merge two sorted lists.
If A[index_A] and B[index_B] are ever equal, add A[index_A]

A. Method 1 is has lower average time complexity (Big O) than Method 2
B. Method 1 is faster for small arrays
C. Method 1 is faster for large arrays

	ABC
0:	FFF
1:	FFT
2:	FTF
3:	FTT
4:	TFF
5:	TFT
6:	TTF
7:	TTT

Peer Instruction

A. Hash Tables (assuming little collisions) are O(N). Quick sort averages O(N*log N). Both have worse case time complexity $O(N^2)$.

For B and C, let's try it out:
Test data is random data injected into arrays equal to SIZE (duplicate entries filtered out).

Size	# matches	Hash Speed	Qsort speed
200	0	23 ms	10 ms
2 million	1,837	7.7 s	1 s
20 million	184,835	Started thrashing – gave up	11 s

So TFF!

Analysis

- The hash table performs worse and worse as N increases, even though it has better time complexity.
- The thrashing occurred when the table occupied more memory than physical RAM.

And in conclusion…

- **CACHE, CACHE, CACHE. Its effects can make seemingly fast algorithms run slower than expected. (For the record, there are specialized cache efficient hash tables)**
- **Function Inlining: For frequently called CPU intensive functions, this can be very effective**
- **Malloc: Less calls to malloc is more better, big blocks!**
- **Preprocessing and memoizing: Very useful for often called functions.**
- **There are other optimizations possible: But be sure to test before using them!**

Bonus slides

- **Source code is provided beyond this point**
- **We don't have time to go over it in lecture.**

Method 1 Source – in C++

```
int I = 0, int j =0, int k=0;
int *array1, *array2, *result; //already allocated (array are set)
map<unsigned int, unsigned int> ht; //a hash table
for (int i=0; i<SIZE; i++) {  //add array1 to hash table
     ht[array1[i]] = 1;
}
for (int i=0; i<SIZE; i++) {
   if(ht.find(array2[i]) != ht.end()) { //is array2[i] in ht?
      result[k] = ht[array2[i]];  //add to result array
      k++;
   }
}
```

Method 2 Source

```
int I = 0, int j =0, int k=0;
int *array1, *array2, *result; //already allocated (array are set)
qsort(array1,SIZE,sizeof(int*),comparator);
qsort(array2,SIZE,sizeof(int*),comparator);
//once sort is done, we merge
while (i<SIZE && j<SIZE){
      if (array1[i] == array2[j]){ //if equal, add
            result[k++] = array1[i] ; //add to results
            i++;   j++; //increment pointers
      }
      else if (array1[i] < array2[j]) //move array1
            i++;
      else  //move array2
            j++;
      }
```

Along the Same lines - Malloc

- Malloc is a function call – and a slow one at that.
- Often times, you will be allocating memory that is never freed
  - Or multiple blocks of memory that will be freed at once.
- Allocating a large block of memory a single time is much faster than multiple calls to malloc.

```
int *malloc_cur, *malloc_end;
//normal allocation:
malloc_cur = malloc(BLOCKCHUNK*sizeof(int*));
//block allocation - we allocate BLOCKSIZE at a time
malloc_cur += BLOCKSIZE;
 if (malloc_cur == malloc_end){
      malloc_cur = malloc(BLOCKSIZE*sizeof(int*));
      malloc_end = malloc_cur + BLOCKSIZE;
 }
Block allocation is 40% faster
 (BLOCKSIZE=256; BLOCKCHUNK=16)
```