Scientists create Memristor, missing fourth circuit element

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:

```c
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

```c
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!

```c
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!

```c
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

<table>
<thead>
<tr>
<th>Test</th>
<th>Totally Random number time</th>
<th>Random power of 2 time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitcount_std</td>
<td>830 ms</td>
<td>790 ms</td>
</tr>
<tr>
<td>Bitcount_op</td>
<td>660 ms</td>
<td>273 ms</td>
</tr>
<tr>
<td>Bitcount_preprocess</td>
<td>720 ms</td>
<td>700 ms</td>
</tr>
</tbody>
</table>

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 trouble spot
• As it happened, my code was pretty inefficient
• Won’t always be as easy. Good forensic skills are a must!

Profiling analysis

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

• The same function in assembly:
  ```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.
- 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**

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

For B and C, let’s try it out:

Test data is random data injected into arrays equal to SIZE (duplicate entries filtered out).

<table>
<thead>
<tr>
<th>Size</th>
<th># matches</th>
<th>Hash Speed</th>
<th>Qsort speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>0</td>
<td>23 ms</td>
<td>10 ms</td>
</tr>
<tr>
<td>2 million</td>
<td>1,837</td>
<td>7.7 s</td>
<td>1 s</td>
</tr>
<tr>
<td>20 million</td>
<td>184,835</td>
<td>Started thrashing – gave up</td>
<td>11 s</td>
</tr>
</tbody>
</table>

**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++

```c++
int I = 0, j = 0, 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++) { //for each entry in array2
    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

```c++
int I = 0, j = 0, k = 0;
int *array1, *array2, *result; //already allocated (array are set)
qsort(array1, SIZE, sizeof(int*), comparator);
qsort(array2, SIZE, sizeof(int*), comparator);
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])
        i++;
    else
        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 calls to malloc:
- Allocating a large block of memory a single time is much faster than multiple calls to malloc:

```c
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)
```