Lecture 18 - Performance Examples

ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

February 15, 2012

Previous Lecture

- A laundry list of compiler optimizations

- Your code should be as **readable** as possible
  - The compiler is likely to do a better job
  - The optimization may not even matter in the big picture (we'll see were to focus our efforts when we do profiling)

Don't waste yourself

- You should give the compiler as much information* as possible
  - *correct information
  - Using `restrict` and `__builtin_expected`

Introduction

- So far, we've only seen C, since we haven't seen anything terribly complex
- Writing compact, readable code in C is hard, common things you see are:
  - **define macros**
  - **void***
- Mainly C, because it's low level, and we want to learn whats really going on
- C++11 has made major strides towards readability and efficiency (light-weight abstractions)

Problem

All we want to do is sort a bunch of integers

- In **C** our standard option is to use qsort in `stdlib.h`

```
void qsort (void* base, size_t num, size_t size,
            int (*comparator) (const void*, const void*));
```

- A fairly ugly definition (as is standard with generic C functions)

qsort Usage

```
#include <stdlib.h>

int compare(const void* a, const void* b)
{
    return (*((int*)a) - *((int*)b));
}

int main(int argc, char* argv[])
{
    int array[] = {4, 3, 5, 2, 1};
    qsort(array, 5, sizeof(int), compare);
}
```

- This looks like a nightmare and is more likely to have bugs

C++ sort

C++ has a version of sort that is much nicer interface*...

```
template <class RandomAccessIterator>
void sort (
    RandomAccessIterator first,
    RandomAccessIterator last
);

template <class RandomAccessIterator, class Compare>
void sort (
    RandomAccessIterator first,
    RandomAccessIterator last,
    Compare comp
);
```

* To use, after you get over templates (they're useful, I swear)

C++ sort Usage

```
#include <vector>
#include <algorithm>

int main(int argc, char* argv[])
{
    std::vector<int> v = {4, 3, 5, 2, 1};
    std::sort(v.begin(), v.end());
}
```

Note: Your compare function can be function or a functors, by default it's `operator<`

- Which is less error prone?
- Which is **faster**?

Standard Algorithms Results

[Shown actual runtimes of `qsort` vs `sort`]

- The C++ version is **twice** as fast, why?
  - The C version just operates on memory, it has no clue what the data is
  - We're throwing away useful information about what's being sorted
  - A C function call will prevent inlining of the compare function
- What if we write our own sort in C, specialized for the data?

Results and Conclusion

[Shown actual runtimes of custom sort vs `sort`]

- The C++ version is still faster (although it's close)
- However, this is quickly going to become a maintainability nightmare
  - Would you rather read a custom sort or 1 line?
  - What do you trust more?

- Abstractions will not make your program slower, they can actually allow speedups and is much easier to maintain and read

Lecture Fun

Let's throw Java in the mix and see what happens

Problem

- Generate **N** random integers and insert them into (sorted) sequence
  Example: 3 4 2 1
  - 3
  - 3 4
  - 2 3 4
  - 1 2 3 4
- Remove **N** elements one at a time by going to a random position and removing the element
  Example: 2 0 1 0
  - 1 2 4
  - 2 4
  - 2
  -

For which **N** is it better to use a list than a vector (or array)?

Complexity

- **Vector**
  - Inserting
    - $O(\log n)$ for binary search
    - $O(n)$ for insertion (on average, move half the elements)
  - Removing
    - $O(1)$ for accessing
    - $O(n)$ for deletion (on average, move half the elements)
- **List**
  - Inserting
    - $O(n)$ for linear search
    - $O(1)$ for insertion
  - Removing
    - $O(n)$ for accessing
    - $O(1)$ for deletion

Therefore, based on their complexity lists should be better

Reality

[Shown actual runtimes of vectors and lists]

Vectors dominate lists performance wise, why?

- Binary search vs. linear search complexity dominates
- The amount of memory lists use is far higher
  64 bit machines:
  - Vector: 4 bytes per element
  - List: At least 20 bytes per element
- Memory access is slow, and comes in blocks
  - Lists elements are all over memory, so there is a large number of cache misses
  - A cache miss for a vector will bring a lot more usable data

Conclusion

- Don't store unnecessary data in your program
- Keep your data as compact as possible
- Access memory in a predictable manner
- Use vectors instead of lists by default
- Programming abstractly can save a lot of time

Summary

- More cases were giving the compiler more information gives you better code
- Data structures can be very important, more so than complexity
- **Low-level code != Efficient**
- You should think at a low level if you need to optimize anything
- Readable code is good code (different hardware will have different optimizations)