# Lecture 17—High-Level Language Performance: Showdown

ECE 459: Programming for Performance

March 11, 2014

# Last Time

More profiling tools:

- Single-process: gperftools;
- Systemwide: oprofile, perf, DTrace, WAIT.

Reentrancy vs thread-safety.

Inlining.

# Introduction

So far, we've only seen C—we haven't seen anything complex.

C is low level, which is good for learning what's really going on.

Writing compact, readable code in C is hard.
Common C sights:

- **#define macros**
- **void***

C++11 has made major strides towards readability and efficiency (it provides light-weight abstractions).

# Goal

Sort a bunch of integers.

In **C**, usually use qsort from `stdlib.h`.

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

- A fairly ugly definition (as usual, for generic C functions)

# How ugly? 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 sort with a much nicer interface[1]...

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

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

[1] ...nicer 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 a function or a functor.
By default, `sort` uses `operator<` on the objects being sorted.

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

# Timing Various Sorts

[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 about the data.
- We're throwing away useful information about what's being sorted.
- A C function-pointer call prevents inlining of the compare function.

OK. What if we write our own sort in C, specialized for the data?

# Custom Sort

[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 (who) do you trust more?

Abstractions will not make your program slower.

They allow speedups and are much easier to maintain and read.

# Lecture Fun

Let's throw Java-style programming (or at least collections) into the mix and see what happens.

# Vectors vs. Lists: Problem

1. 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

2. 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.
- Lists use far more memory.
  **On 64 bit machines:**
  - Vector: 4 bytes per element.
  - List: At least 20 bytes per element.
- Memory access is slow, and results arrive in blocks:
  - Lists' elements are all over memory, hence many cache misses.
  - A cache miss for a vector will bring a lot more usable data.

# Performance Tips: Bullets

- 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.

# Programming for Performance with the Compiler

- Often, telling the compiler more gives you better code.
- Data structures can be critical, sometimes more than complexity.
- **Low-level code != Efficient**.
- Think at a low level if you need to optimize anything.
- Readable code is good code—
  different hardware needs different optimizations.