Performance Examples

2024 Fall ECE454: Computer Systems Programming — Lecture 4

Jon Eyolfson

1.0.0

Takeaways from Last Lecture

We saw 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
  - (you'll see were to focus your efforts when profiling)

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

Writing Readable C Code is Hard

Common things you see are:

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

The language itself is not very large, it’s also low level

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

One of the Most Impactful Compiler Optimization is Inlining

```
class Point {
public:
    int getX() {
      return x;
    }
private:
    int x;
};

int main(void) {
  Point p = /* ... */;
  std::cout << p.getX() << std::endl;
}
```

would get optimized to:

```
int main(void) {
  Point p = /* ... */;
  std::cout << p.x << std::endl;
}
```

Inlining has a Tradeoff

You will avoid the overhead of a function call and return

However, the program size may increase

- At runtime you may have worse performance due to the instruction cache

Inlining may allow other compiler optimizations to happen as well

The `inline` keyword is just a suggestion to the compiler, it can ignore you

Vecor vs List 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)?

Theoretical 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 of Vectors and Lists

[Shown in class]

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 indirect, and comes in blocks
  Lists 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

We Can Also Use `perf`

`pref` is a Linux specific profiler that lets you access hardware counters

You can run it before any command, some good arguments:

```
perf stat -B -e task-clock,cycles,instructions
perf stat -B -e cache-references,cache-misses,branches,branch-misses,page-faults
```

You may use it in this course, but your devcontainer likely won’t support it

`perf` Example using Vectors on a Raspberry Pi 4

I ran: `perf stat -B build/vector-vs-list 20000 --vector`

```
         508.40 msec task-clock:u                     #    0.994 CPUs utilized
    762,299,327      cache-references:u               #    1.499 G/sec
      6,834,952      cache-misses:u                   #    0.90% of all cache refs
    852,207,596      cycles:u                         #    1.676 GHz
    570,993,225      instructions:u                   #    0.67  insn per cycle
<not supported>      branches:u
      1,103,866      branch-misses:u
            280      page-faults:u                    #  550.746 /sec

    0.511387243 seconds time elapsed

    0.502540000 seconds user
    0.008040000 seconds sys
```

`perf` Example using Lists on a Raspberry Pi 4

I ran: `perf stat -B build/vector-vs-list 20000 --list`

```
       21,107.95 msec task-clock:u                     #    1.000 CPUs utilized
   1,529,772,607      cache-references:u               #   72.474 M/sec
   1,409,358,582      cache-misses:u                   #   92.13% of all cache refs
  37,873,673,434      cycles:u                         #    1.794 GHz
   6,070,348,884      instructions:u                   #    0.16  insn per cycle
  <not supported>      branches:u
       2,004,950      branch-misses:u
             304      page-faults:u                    #   14.402 /sec

    21.110834897 seconds time elapsed

    21.105720000 seconds user
     0.004000000 seconds sys
```

More Performance Tips

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

Takeways

Giving the compiler more information produces 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 to start with
(different hardware will have different optimizations)