Optimizing program performance

for C/C++

Pham Ngoc Tan   Vo Khanh An

Faculty of Computer Science
University of Information Technology, VNU HCM

April, 2021

Reference

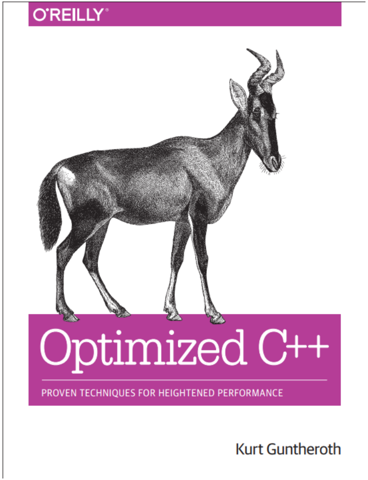

O'REILLY
Optimized C++
PROVEN TECHNIQUES FOR HEIGHTENED PERFORMANCE
Kurt Guntheroth

Table of Contents

Table of Contents

What is optimization ?

What is optimization ?

- A coding activity

What is optimization ?

- A coding activity
- Previously take place after code complete, during the integration and testing phase of a project

The goal of optimization

The goal of optimization

- Improve the behavior of a correct program to meet customer needs

The goal of optimization

- Improve the behavior of a correct program to meet customer needs
  $\Rightarrow$ As important to the development process as coding features is

Bug fixing versus Performance tuning

Bug fixing versus Performance tuning

- Performance is a **continuous** variable
- Bug is a **discrete** variable *present or absent*

Bug fixing versus Performance tuning

- Performance can be either good or bad or something in between
- Optimization is also an iterative process in which each time the slowest part of the program is improved

Optimizing Category

Optimizing Category

When you write code for C/C++ or any programming language, your first and foremost goal is to make your program **executable** and **correct**.

Optimizing Category

When you write code for C/C++ or any programming language, your first and foremost goal is to make your program **executable** and **correct**.

After that, we consider a few things below:

Optimizing Category

When you write code for C/C++ or any programming language, your first and foremost goal is to make your program **executable** and **correct**.

After that, we consider a few things below:

- Security of the program
- Memory consumption
- Speed of the program (Performance improvement)

Optimizing Category

When you write code for C/C++ or any programming language, your first and foremost goal is to make your program **executable** and **correct**.

After that, we consider a few things below:

- Security of the program
- **Memory consumption**
- **Speed of the program** (Performance improvement)

Table of Contents

Problems in optimizing a program

- Speed optimizing through all possible techniques, but with a tremendous memory
- Get conflict due to the use of 2 different optimizing goals

Problems in optimizing a program

- Speed optimizing through all possible techniques, but with a tremendous memory
- Get conflict due to the use of 2 different optimizing goals
- Avoid cheap optimizing tricks for a better program and not receive bad consequences
- Despite efforts in optimizing, the program might not be completely optimized

Problems in optimizing a program

"We should forget about small efficiencies, say about 97 percent of the time: premature optimization is the root of all evil."

Donald Knuth, *Structured Programming with go to Statements*, ACM Computing Surveys 6(4), December 1974, p268. CiteSeerX: 10.1.1.103.6084

Ahmdal's Law

$$Speedup = \frac{time_{old}}{time_{new}} = \frac{1}{(1 - f_{cost}) + \frac{f_{cost}}{f_{speedup}}}$$

Ahmdal's Law

$$Speedup = \frac{time_{old}}{time_{new}} = \frac{1}{(1 - f_{cost}) + \frac{f_{cost}}{f_{speedup}}}$$

s.t.

- f_{cost}: percentage of the program runtime used by the function f
- $f_{speedup}$: the factor to speed up f

Ahmdal's Law

$$Speedup = \frac{time_{old}}{time_{new}} = \frac{1}{(1 - f_{cost}) + \frac{f_{cost}}{f_{speedup}}}$$

If a function takes the program 40% of total runtime and we have optimized it with a double speed, then the program will be 25% faster

$$Speedup = \frac{1}{(1 - f_{cost}) + \frac{f_{cost}}{f_{speedup}}} = \frac{1}{(1 - 0.4) + \frac{0.4}{2}} = 1.25$$

Ahmdal's Law

- An infrequently code might be not a need for optimizing
- *"Make the common case fast and the rare case correct"*

Table of Contents

Table of Contents

Use Better Data Structures

- Manipulation, e.g. inserting, iterating, sorting or retrieving entries, has a runtime cost depending on data structures
- Using different data structures make differing use of memory costs

Use Better Data Structures

Array case:

- Fixed memory, must declare number of items
- Access to a random position in $O(1)$
- Add/remove one element in $O(N)$

Use Better Data Structures

Linked list case:

- Non-fixed memory, no need to declare number of items
- Access to a random position in $O(N)$
- Add/remove first/last element in $O(1)$

Table of Contents

Optimize Algorithms

- Efforts in optimizing algorithms might increase program's performance impressively
- Changing into a more optimized algorithms makes more difference towards the current one when there is a huge dataset
- The more optimized algorithm could even work better with a small amount of data sets if we use that algorithm sufficient enough

Example

Linear search

```
int LinearSearch(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}
```

Linear search

- $O(n)$, is expensive, but extremely general
- It can be used on an unsorted table.
- If the table is sorted, it's still $O(n)$

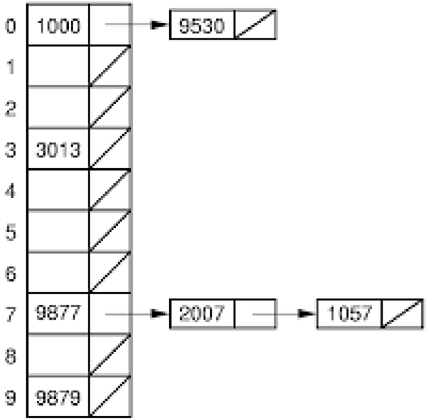
Binary search

```
int BinarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x)
            return mid;

        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        return binarySearch(arr, mid + 1, r, x);
    }
    return -1;
}
```

Binary search

- $O(log_2 n)$, has good performance, but it's not the best possible search
- Binary search requires input data that is sorted on the search key
- Keys that can be compared not only for equality, but for an ordering relation such as less-than.

Hash table



Hash table

- Hashing has worst-case performance of O(n), and may require more hash table entries than there are records to search for.
- However, when the table has fixed contents (like month names or programming language keywords)
  $\Rightarrow$ It is possible to find a record in average O(1)

Table of Contents

Use Better Libraries

"A great library is one nobody notices because it is always there, and always has what people need."

- **Vicki Myron**, author of *Dewey, the Small Town Library Cat*, and librarian of the town of Spencer, Iowa

Use Better Libraries

- The Standard C++ Template *(STL)* is such a powerful library that it may come as surprise with its nearly-optimized speed in comparison with the others
- Mastering STL is a critical skill for C++ developers
- Benefits of this STL is that they may be use in a project instantly, which reduces coding time but sustain the quality of the project

Use Better Libraries

- In *STL*, there is a function named *sort*() and in *stdlib.h* library, there is a function named *qsort*()

Use Better Libraries

- In *STL*, there is a function named *sort*() and in *stdlib.h* library, there is a function named *qsort*()
- About comparison, the *C* standard does not talk about complexity of *qsort*(), while the complexity of *sort*() in the worst case is $O(NlogN)$

Use Better Libraries

- In *STL*, there is a function named $sort()$ and in *stdlib.h* library, there is a function named $qsort()$
- About comparison, the C standard does not talk about complexity of $qsort()$, while the complexity of $sort()$ in the worst case is $O(NlogN)$
- About runtime, STL's $sort()$ runs 20% to 50% faster than the hand-coded Quick Sort and 250% to 1000% faster than the C $qsort()$ library function. C might be the fastest language but $qsort()$ is very slow.

Use Better Libraries

- About flexibility, STL can work on many different data types, e.g. array, vector, deque. This flexibility is quite harder to achieve in C

Use Better Libraries

- About flexibility, STL can work on many different data types, e.g. array, vector, deque. This flexibility is quite harder to achieve in C
- About safety, STL's $sort()$ is safer due to require no access to data via pointer as C's Standard $qsort()$ does

Table of Contents

Optimize Dynamically Allocated Variables

- Except for the use of less-optimal algorithms, the naïve use of dynamically allocated variables is the greatest performance killer in C++ programs

Optimize Dynamically Allocated Variables

- Except for the use of less-optimal algorithms, the naïve use of dynamically allocated variables is the greatest performance killer in C++ programs
- Improving a program's use of dynamically allocated variables is so often that a developer can be an effective optimizer knowing nothing other than how to reduce calls into the memory manager.

Optimize Dynamically Allocated Variables

- Except for the use of less-optimal algorithms, the naïve use of dynamically allocated variables is the greatest performance killer in C++ programs
- Improving a program's use of dynamically allocated variables is so often that a developer can be an effective optimizer knowing nothing other than how to reduce calls into the memory manager.
- C++ to use dynamically allocated variables, like smart pointers, and strings, make writing applications in C++ productive. But there is a dark side to this expressive power

Optimize Dynamically Allocated Variables

- Except for the use of less-optimal algorithms, the naïve use of dynamically allocated variables is the greatest performance killer in C++ programs
- Improving a program's use of dynamically allocated variables is so often that a developer can be an effective optimizer knowing nothing other than how to reduce calls into the memory manager.
- C++ to use dynamically allocated variables, like smart pointers, and strings, make writing applications in C++ productive. But there is a dark side to this expressive power
- When performance matters, *new* is not your friend.

Create Class Instances Statically

```
MyClass* myInstance = new MyClass("hello", 123);
```

```
MyClass myInstance("hello", 123);
```

Create Dynamic Variables Outside of Loops

```cpp
for (auto& filename : namelist) {
    std::string config;
    ReadFileXML(filename, config);
    ProcessXML(config);
}
```

```cpp
std::string config;
for (auto& filename : namelist) {
    config.clear();
    ReadFileXML(filename, config);
    ProcessXML(config);
}
```

Disable Unwanted Copying In The Class Definition

- Not every object in a program should be copied
- Some tremendous objects, e.g. a vector of $1,000$ strings, are brought into function meant to examine it to function properly, but the runtime cost of the copy may be considerable
- Forbidding copying is a must by declaring the copy constructor and assignment operator private if copying a class is undesirable. The declaration alone is enough.

Disable Unwanted Copying In The Class Defination

```cpp
// pre-C++11 way to disable copying
class BigClass {
private:
    BigClass(BigClass const&);
    BigClass& operator=(BigClass const&);
public:
    ...
};
```

```cpp
BigClass(BigClass const&) = delete;
BigClass& operator=(BigClass const&) = delete;
...
```

Table of Contents

Optimize Hot Statements

- Optimizing at the statement level can be modeled as a process of removing instructions from the stream of execution.
- There is no need to focus on small-scale instructions *no statement consumes more than a handful of machine instructions* $\rightarrow$ not worth
- We would rather find factors that magnify the cost of the statement, making it hot enough to be worth optimizing.

Remove Invariant Code from Loops

```
int i,j,x,a[10];
   ...
for (i=0; i<10; ++i) {
    j = 100;
    a[i] = i + j * x * x;
}
```

```
int i,j,x,a[10];
   ...
j = 100;
int tmp = j * x * x;
for (i=0; i<10; ++i) {
    a[i] = i + tmp;
}
```

Remove Unneeded Function Calls from Loops

```
char* s = "sample data with spaces";
    ...
for (size_t i = 0; i < strlen(s); ++i)
    if (s[i] == ' ')
        s[i] = '*'; // change ' ' to '*'
```

```
char* s = "sample data with spaces";
    ...
size_t end = strlen(s);
for (size_t i = 0; i < end; ++i)
    if (s[i] == ' ')
        s[i] = '*'; // change ' ' to '*'
```

Table of Contents

Use Better I/O

- Read and write are time-consuming instructions
- Thus, when needed, we had better use read and write from file to save time

Use Better I/O

```
int n;
int a[1000000];

cin >> n;
for (int i = 0; i < n; i++)
    cin >> a[i];
```

```
int n;
int a[1000000];

freopen("input.txt", "r", stdin);

cin >> n;
for (int i = 0; i < n; i++)
    cin >> a[i];
```

The end.