The Solutions People

White Paper

A PRACTICAL APPROACH TO OPTIMIZE CODE IMPLEMENTATION

ABSTRACT

In the era of rapidly changing technology, there has been a huge emphasis on making lightweight, low power consuming and responsive applications. A few examples of applications include graphic editors on handheld and mobile devices, video calling/conferencing applications, live broadcasting station on wearable device, and many others. This requirement directly links to the fact that application / program should utilize the CPU resources like computation engine, video processing unit, crypto engine, audio processing unit efficiently. This raises a unique challenge for developers to write the program in an efficient manner, so that each CPU clock cycle is utilized to its maximum limit. This application program or embedded software often runs on processors with limited computational power, and hence raises a need for Code Optimization.

Most of the existing compilers are optimizing the program code at the lower level. But manual optimization remains predominantly important for performing optimization at the source code level. Compiler is often limited to general-purpose optimization that should apply to many programs, while the programmer can be creative and can use his/her knowledge and experience to perform optimization. It is often difficult to identify areas where more optimization is required, especially in case of large and complex applications. In such cases, we can use profiler to identify those areas and verify the result of optimization. Several different techniques exist by which a program can run faster, but it often increases the size of the program as well.

In this article, we will look into different methods of manual optimization. These methods focus on minimizing the time spent by the CPU, and provides transformed source codes that often yield improvements. Various methods to achieve optimization and time are represented, which includes from data type usage to complicated loop optimization.

INTRODUCTION:

Software applications are designed to achieve a specified set of functionalities. The performance consideration may add to the design later on.

There are various techniques which can help a software run faster. However, improvement in speed can also cause an increase in the program size. So the approach to improving the performance is to write the code in such a way that both the memory and speeds are optimized. There are different ways to get performance data, but the best way is to use a good performance profiler, which shows the time spent in each function and provide analysis on the data. In this article, we will use an example of an image recognition program and the same can be applied to any other signal processing code.

Once we have the data, the next step is to analyze and identify the routines that consumes more than desired time to execute. These areas are known as Hotspots. Before starting optimization, it is better to have a look at the code and choose a better algorithm, if any, rather than optimizing existing code for performance.

GOALS OF CODE OPTIMIZATION

1. Remove redundant code without changing the meaning of the program.
2. Reduce execution speed.
3. Reduce memory consumption.

A few points to keep in mind before optimizing the code:

- Time based optimization will give faster output, but it can lead up to a bigger code base. So to optimize the code for time performance may actually conflict with memory and size consumption. For that, you have to find the balance between time and memory, in consideration to your requirement.
- Performance optimization is a never-ending process. There are always chances to improve and make your code run faster.
- Sometimes, we can be tempted to use certain programming methods to run faster at the expense of not following best practices like coding standards. Try to avoid any such kind of inappropriate methods.

CATEGORY OF OPTIMIZATION

A) Space optimization

B) Time optimization

You can optimize your code by either reducing space or time. Because, both space and time optimization categories are correlated with each other.

A.) Space Optimization:

You can optimize your code using below techniques:

1. Data Types Usage

The use of correct data type is important in a recursive calculation or large array processing. Smaller data types are usually faster. It will improve memory utilization and take lesser CPU cache. This can help us to improve the speed by decreasing number of CPU cycles. Also, when we use smaller data type, it will require less number of CPU cycles for operation.

2. Data Alignment: Arrangement and Packing

The declaration of the component in a structure will determine how the components are being stored. Due to the memory alignment, it is possible to have dummy area within the structure. It is recommended to place all similar sized variables in the same group which is as shown below.

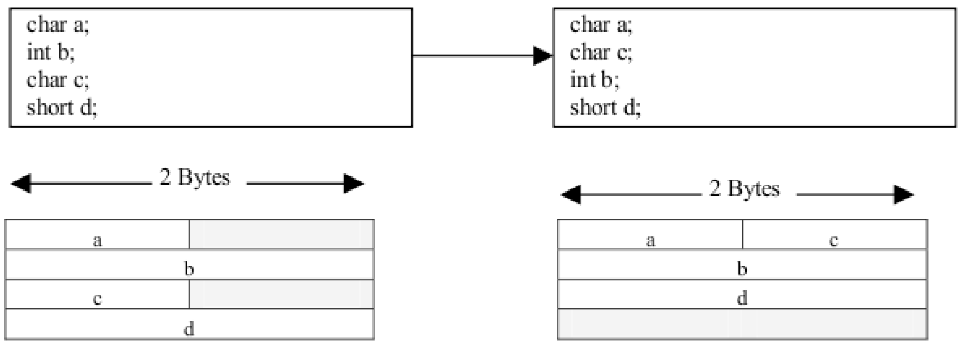

Figure 1 : Data alignment

As the structure is packed, integer b will not be aligned. But in the second approach, it will improve RAM, but degrade the operational speed, as the access of ‘b’ will take up two cycles.

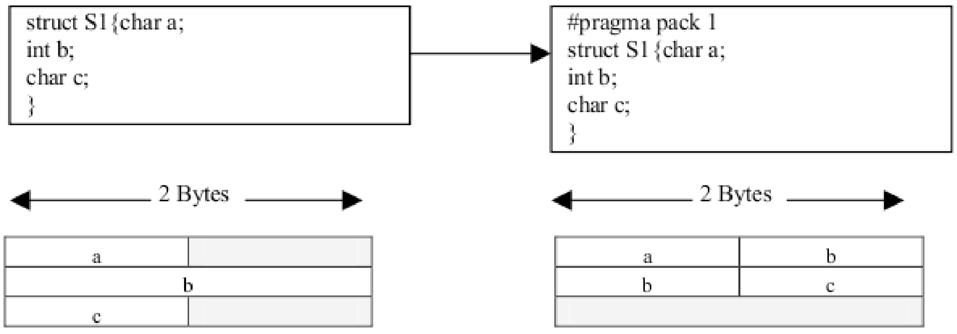

Figure 2: Data alignment

3. Pass by Reference

When we pass a large structure or class to a function, pass by value will make a copy of the arguments. In most cases, this is not required and it may create performance issues. Also, when we pass arguments by value, the only way to return a value back to the caller is via the function's return value. It is often suitable to use pass by reference and modify the argument to give the value back to caller function.

If we use pass by value, when there are large number of arguments, it may increase processing time and RAM usage due to the number of pushing and popping action on each function call. Especially, when there is a need to pass a large structure or class, it will take a long time.

With pass by reference, it allows us to pass large structures and classes with a minimum performance penalty and RAM usage.

4. Return Value

The return value of a function will be stored in a register. If this return data have no intended usage, time and space are wasted in storing this information. Programmer should define the function as "void" to minimize the extra handling in the function.

B) Time Optimization:

1. Optimize Program Algorithm

For any code, you should always allocate some time to think the right algorithm to use. So, the first task is to select and improve the algorithm which will be frequently used in the code.

2. Avoid Type Conversion

Whenever possible, plan to use the same type of variables for processing. Type conversion must be avoided. Otherwise, extra machine cycles will be wasted to convert from one type to another.

3. Loop Related Optimization

If you identify that a loop is executed thousands of cycles in your code and is taking most of execution time, the best thing is to redesign code for lesser loop execution count. This is more effective than making the loop run faster.

Below are various techniques that can be implemented to optimize the loop:

• Inline Function

Functions can be instructed to compiler to make them inline so that the compiler can replace those function definitions wherever called. It saves overhead of variable push/pop on the stack, which was required for function calling and also reduces overhead of return from function. At the same time inline function will tend to increase the code size.

Example:

```
inline int max (int x, int y) {
                        return (x > y)? x : y;
}
```

• Loop Unrolling

Loop unrolling is to replicate the code inside a loop for number of cycles. The number of copies is called the loop unrolling factor.

In this technique, code size increases with reducing number of counts of loop as well as number of jumps. This results in faster execution time.

Old Code:

```
     for (i = 0; i < 100; i++)
             {
               test (i);
             }
```

New Code:

```
     for (i = 0; i < 100; i++)
             {
                test(i);   i++;
       test (i);   i++;
       test (i);   i++;
                test (i);   i++;
       test (i);   i++;
       test (i);   i++;
       test (i);   i++;
       test (i);   i++;
       test (i);   i++;
       test (i);   i++;
    }
```

• Loop Jamming or Loop Fusion

Loop jamming is the replacement of multiple loops with a single one. It is possible when two adjacent loops iterate over the same range. So that, the loop overhead is reduced, resulting in a speed-up in execution, as well as a reduction in code space. Loop jamming does not always improve run-time speed.

Old Code:

```
for (j = 0; j < 1000; j++)   /* initialize array to 0's */
      a[j] = 0;
 for (i = 0; i < 1100; i++)   /* put 1 to other array */
```

```
        b[i] = 1;
```

New Code:

```
for (i = 0; i < 1000; i++)
  {
     a[i] = 0;      /* initialize array to 0's */
      b[i] = 1;         /* put 1 to other array- 0f 1000 times */
  }
i++;
while(i<1100)                    // rest of 100 times
                b[i]=1;  i++;   //reduce iteration from 2100 to 1100
```

• Loop Inversion

Loop inversion improves the performance in boundary conditions. Rather than checking loop condition at the beginning, it is better to check at the end, so that it does not require to jump back to the top even when the condition fails, . For that purpose, one can use (if + do...while) instead of only while.

Old Code:

```
int i, a[100];
  i = 0;
  while (i < 100) {
    a[i] = 0;
    i++;
  }
```

New Code:

```
 int i, a[100];
  i = 0;
  if (i < 100) {
    do {
      a[i] = 0;
      i++;
    } while (i < 100);
  }
```

When the value of i is 99, the processor need not jump. (This is required in the old code) Processors have a special instruction for decrement, which is faster to test if compared with zero than comparing two different numbers. So it reduces execution time. e.g.

```
for (i=10; i--;)        and         i = MAX+1;
                                                  while(--i) {..}
```

• Strength Reduction

Strength reduction is the replacement of a more expensive expression by less expensive expression, so that it now becomes cheaper to compute. E.g. pow (a, 2) = a * a.

Old Code:

```
int i, sum = 0;
for (i = 1; i <= N; ++i) {
  sum += i;
}
printf("sum: %d\n", sum);
```

New Code:

```
int sum = N * (1 + N) / 2;
printf ("sum: %d\n", sum);
```

• Loop Invariant Computations

Instructions that give the same value each time when defined with an infinite loop should be kept outside the infinite loop to protect that line from calculating repeatedly inside a loop.

Old Code:

```
for (int i=0; i<n; ++i) {
    x = y+z;            //contains constant value on each iteration
    a[i] = 6*i + x*x;
}
```

New Code:

```
x = y+z;
t1 = x*x;
for (int i=0; i<n; ++i) {
    a[i] = 6*i + t1;
}
```

4. Use Lookup Table

Look-Up-Table (LUT) is an array that holds a set of pre-computed results for a given operation. It provides access to the results in a way that is faster than computing each time the result of the given operation. Look-Up Tables are a tool to accelerate operations that can be expressed as functions of an integer argument. They store pre-computed results that allow the program to immediately obtain a result without performing same time-consuming operation repeatedly.

Old Code:

```
/*this code is used at multiple places to calculate same result*/
 for (i = 0; i < 256; i++)
{
            result = sqrt (i);
}
```

New Code:

```
double pre_cal_sqrt [256];    /* declared globally */
for (i = 0; i < 256; i++)
{
             pre_cal _sqrt[i] = sqrt (i);
}
  /* during the normal execution it can be used multiple time */
  result= pre_cal _sqrt [sample];                /* instead of result = sqrt (sample); */
```

However, we have to be careful before deciding to use LUT's to solve a particular problem. We should carefully evaluate the cost associated with their use (in particular, the memory space that will be required to store the pre-computed results).

EXAMPLE OF CODE OPTIMIZATION

Here, we have taken a sample code snippet for Image Recognition. Below is the code where the same function is used multiple times and called by boundingRect().

Old Code:

```
vector<vector<Point> > validContours;
for (int i=0;i<contours_poly.size();i++){
        Rect r = boundingRect(Mat(contours_poly[i]));
        ...
        for(int j=0;j<contours_poly.size();j++){
                Rect r2 = boundingRect(Mat(contours_poly[j]));
                //Do operation on both rectangle and find if it is valid contour
                if(condition)
                        validContours.push_back(contours_poly[i]);
        }
}
```

In another function, it is required to create rectangle for valid contours.

```
for(int i=0;i<validContours.size();i++){
                                boundRect[i] = boundingRect( Mat(validContours[i]) );
```

In the old code, we have created boundRect multiple times, whenever it is required. But in the optimized code, with expense of memory, we have created it for a single time and used it whenever required. This will increase memory size as we are keeping all boundRect, but we can reduce execution time as we are calculating it only once.

New Code:

```
for (int i=0;i<contours_poly.size();i++){
        boundRect[i] = boundingRect( Mat(contours_poly[i]) );
        Rect r = boundRect[i];
        ...
        for(int j=0;j<contours_poly.size();j++){
                Rect r2 = boundRect[j];
                //Do operation on both rectangle and find if it is valid contour
                if(condition)
                        valid[i]=1;
}
```

In one of the image processing code, we have used tessaract and SVM methods for detection. These methods need to initialize once and hence we can move it outside the loop. So it is important to identify areas in the loop, which gives same output and need not be calculated every time inside the loop. We can move these codes outside of loop.

Also, we have revisited code and used new tesseract API which takes argument in different format. With this new API, we can use data that are in available format and we do not need to convert these data to make it fit in old API argument format.

Old code:

```
Pix *imaget = pixCreate(i_roi.size().width, i_roi.size().height, 8);
api->SetImage(imaget);
```

New code:

```
api->SetImage(i_roi.data, i_roi.size().width, i_roi.size().height, i_roi.channels(), i_roi.step1() );
```

These are some of the examples which we can apply to code for optimization.

OBSERVATION

After applying above optimizing techniques, below is the observation table for the original code and optimized code execution time.

	Original code execution time	Optimized code execution time	Performance improvement (%)
Code 1	55356.9	35668.7	28.83%
Code 2	18820.4	12871.9	31.61%
Code 3	544.772	397.68	27%
Code 4	457.161	341.912	25.21%

Table 1: Observation table

CONCLUSION

As it is depicted in the observation table, that by using optimization technique the improvement in execution time is evident. This experiment was carried out on an image identification program, but it is not limited to the usage within the image identification program. The scope extends across the entire software engineering projects, including IoT, Cloud Computation, Signal Processing, ADAS, machine learning, neural networks, deep learning and others. Careful selection of optimization technique can improve the performance drastically. Microprocessor's architecture can also be exploited to enhance the overall timing improvements.

ABOUT AUTHORS

Rudrik Upadhyay

Rudrik Upadhyay is a Senior Embedded Engineer at eInfochips. He has worked in various domains of software programming, including Embedded, Network, IoT, streaming and Algorithm optimizations. He contributes to data analytics and neural networks projects within eInfochips.

Julsi Nagarbandhara

Julsi is an Embedded Engineer at eInfochips. She works in various domains of software processing like Embedded Software, Neural Networks, Algorithm Optimization and IoT..

Samir Bhatt :

Samir Bhatt is a Senior Technical Lead at eInfochips and has over a decade of experience in various domains of software programming like Video/Graphic processing, Automotive, Medical Instrumentation, Streaming, and Algorithm Optimizations, etc. He has been guiding a team of focused engineers to execute Neural Networks projects for many engineering solutions at eInfochips

REFERENCES:

http://www.thegeekstuff.com/2015/01/c-cpp-code-optimization/
http://leto.net/docs/C-optimization.php
https://www.slideshare.net/EmertxeSlides/embedded-c-optimization-techniques
https://www.mochima.com/articles/LUT/LUT.html
https://www.d.umn.edu/~gshute/arch/loop-unrolling.xhtml
http://anale-informatica.tibiscus.ro/download/lucrari/8-1-09-Abdulla.pdf

ABOUT eInfochips

eInfochips is a full-fledged product engineering and software services company with over 20 years of experience, 500+ product developments, and over 10M deployments in 140 countries across the world. Majority of our clients and partners are Fortune 500 companies and 80% of our business is generated from solutions for connected devices. From silicon to embedded systems to software, from deployment to sustenance, we map the journey of our customers.
We have the expertise and experience to deliver complex, critical, and connected products across multiple domains, for projects as small as a one-time app development to a complete turnkey product design. We continuously invest and fuel innovations in the areas of Product Engineering, Device Lifecycle Management, IoT & Cloud Frameworks, Intelligent Automation, and Video Management.