Sub-Sahara African Academic Research Publications
December, 2020, Editions

JPAS
Journal of Pure and Applied Science
Vol..17, No.9, ISSN 2318-9315

A SIMPLE CODE OPTIMIZING TOOL

VICTOR IBOYI[1]; FATIMA SHITTU[2]; & MUSTAPHA LAWAL ABDULRAHMAN3

[1,2]Department of Computer Science, Federal Polytechnic, Damaturu
[3]Mathematical Science, Abubakar Tafawa Balewa University Baushi, Nigeria

ABSTRACT

Compilers can be designed to provide code optimization. An optimizing compiler is a compiler that tries to minimize or maximize some attributes of an executable computer program. The most common requirement is to minimize the time taken to execute a program; a less common one is to minimize the amount of memory occupied. Users should only focus on optimizations not provided by the compiler such as choosing a faster and/or less memory intensive algorithm. The need to enhance code for efficient use of computer resources today has called for the need of code optimization. In this work, we look at the various level of optimization. We also describe the different optimizations performed by the compiler as an automated optimizer which include In lining small functions, Code hoisting, Dead store elimination, Eliminating common sub-expressions, Loop unrolling, Loop optimizations: Code motion, Induction variable elimination, and Reduction in strength. Java programming language was used to analyse the code which performed the same operation. The performance metrics (speed, memory consumed and code size) were compared between the optimized code and the un-optimized code.

Keywords: *Optimizer, platform independent optimization, code size, code hoisting, memory utilization*

INTRODUCTION

In computer science, optimization is assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order

to speed up the program execution without wasting resources like CPU, memory. The output of this phase is an optimized intermediate code (Annal and Jesintha, 2017). The program that performs an automated optimization is called an optimizer. Most optimizers are embedded in compilers and operate during compilation. Optimizers can often tailor the generated code to specific processors (Hoos, 2012).

Today, automated optimizations are almost exclusively limited to compiler optimization. However, because compiler optimizations are usually limited to a fixed set of rather general optimizations, there is considerable demand for optimizers which can accept descriptions of problem and language-specific optimizations, allowing a programmer to specify custom optimizations. In general, a computer program may be optimized so that it achieves maximum efficiency in storage capacity or time (executing more rapidly) or cost (Srikant and Shanker, 2018)..

Although the word "optimization" shares the same root as "optimal", it is rare for the process of optimization to produce a truly optimal system (Rubinstein, 1997). The optimized system will typically only be optimal in one application or for one audience. One might reduce the amount of time that a program takes to perform some task at the price of making it consume more memory. In an application where memory space is at a premium, one might deliberately choose a slower algorithm in order to use less memory. Often there is no "one size fits all" design which works well in all cases, so programmers make trade-offs to optimize the attributes of greatest interest.

A Code optimizer sits between the front end and the code generator.

Works with intermediate code.

Can do control flow analysis.

Can do data flow analysis.

Does transformations to improve the intermediate code.

A code optimizing process must follow the three rules given below:

The optimize code must not, in any way, change the meaning of the program.

Optimization should increase the speed of the program and if possible, the program should demand a smaller number of resources.

Optimization should itself be fast and should not delay the overall compiling process.

AIMS AND OBJECTIVES

Goal of this work is to develop a novel code optimizer to remove redundant code without changing the meaning of program. This can be achieved via the objectives listed below:

- Reduce execution speed
- Reduce code size Achieved through code transformation while preserving semantics.
- To reduce memory resources consumed by programs
- To reduce other resources consumed such as bandwidth, power consumption etc.
- To have fewer lines of code doing the same job.

LITERATURE REVIEW

The most common requirement is to minimize the time taken to execute a program; a less common one is to minimize the amount of memory occupied. The growth of portable computers has created a market for minimizing the power consumed by a program. Compiler optimization is generally implemented using a sequence of optimizing transformation algorithm which takes a program and transforms it to produce a semantically equivalent output program that uses fewer resources.

It has been shown that some code optimization problems are NP-complete (Schiex, 2016) or even un-decidable (Cheng, Wang, Li, and Rudolph, 2019). . In practice, factors such as the programmer's willingness to wait for the compiler to complete its task places upper limits on the optimizations that a compiler implementer might provide. Optimization is generally a very CPU- and memory-intensive process (Wescott, 2013). In the past, computer memory limitations were also a major factor in limiting which optimizations could be performed. Because of all these factors, optimization rarely produces "optimal" output in any sense, and in fact an "optimization" may impede performance in some cases; however, heuristic

methods for improving resource usage in typical programs are available to combat this deficiency (Zhang, Wang, Yang, Ding, Li, and Hu (2017).

Levels of Optimization

Optimization can occur at a number of levels. Typically, the higher levels have greater impact, and are harder to change later on in a project, requiring significant changes or a complete rewrite if they need to be changed. Thus, optimization can typically proceed via refinement from higher to lower, with initial gains being larger and achieved with less work, and later gains being smaller and requiring more work. However, in some cases overall performance depends on performance of very low-level portions of a program, and small changes at a late stage or early consideration of low-level details can have outsized impact.

Design level optimization

At the highest level, the design may be optimized to make best use of the available resources, given goals, constraints, and expected use/load. The architectural design of a system overwhelmingly affects its performance.

Algorithms and data structures

Given an overall design, a good choice of efficient algorithms and data structures, and efficient implementation of these algorithms and data structures comes next. After design, the choice of algorithms and data structures affects efficiency more than any other aspect of the program. Generally, data structures are more difficult to change than algorithms, as a data structure assumption and its performance assumptions are used throughout the program, though this can be minimized by the use of abstract data types in function definitions, and keeping the concrete data structure definitions restricted to a few places.

Source code level

Beyond general algorithms and their implementation on an abstract machine, concrete source code level choices can make a significant difference. For example, on early C compilers, while (1) was slower than for (;;) for an unconditional loop, because while (1) evaluated 1 and then

had a conditional jump which tested if it was true, while for (;;) had an unconditional jump.

Build level

Between the source and compile level, directives and build flags can be used to tune performance options in the source code and compiler respectively, such as using preprocessor defined features to disable unneeded software features, optimizing for specific processor models or hardware capabilities, or predicting branching, for instance. Source-based software distribution systems such as BSD's Ports and Gentoo's Portage can take advantage of this form of optimization.

Compile level

Use of an optimizing compiler tends to ensure that the executable program is optimized at least as much as the compiler can predict.

Assembly level

At the lowest level, writing code using an assembly language, designed for a particular hardware platform can produce the most efficient and compact code if the programmer takes advantage of the full repertoire of machine instructions. Many operating systems used on embedded systems have been traditionally written in assembler code for this reason. Programs (other than very small programs) are seldom written from start to finish in assembly due to the time and cost involved. Most are compiled down from a high level language to assembly and hand optimized from there. When efficiency and size are less important large parts may be written in a high-level language.

Runtime level

Just-in-time compilers can produce customized machine code based on run-time data, at the cost of compilation overhead. This technique dates to the earliest regular expression engines, and has become widespread with Java HotSpot and V8 for JavaScript. In some cases adaptive optimization may be able to perform run time optimization exceeding the capability of

static compilers by dynamically adjusting parameters according to the actual input or other factors.

Platform dependent and independent optimization

Optimization can be categorized broadly into two types: machine independent and machine dependent. In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. For example:

```
Do
{
        Item = 10;
            Value = value + item;
} while (value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
Item = 10;
Do
{
            Value = value + item;
} while (value<100);
```

Should not only save the CPU cycles, but can be used on any processor

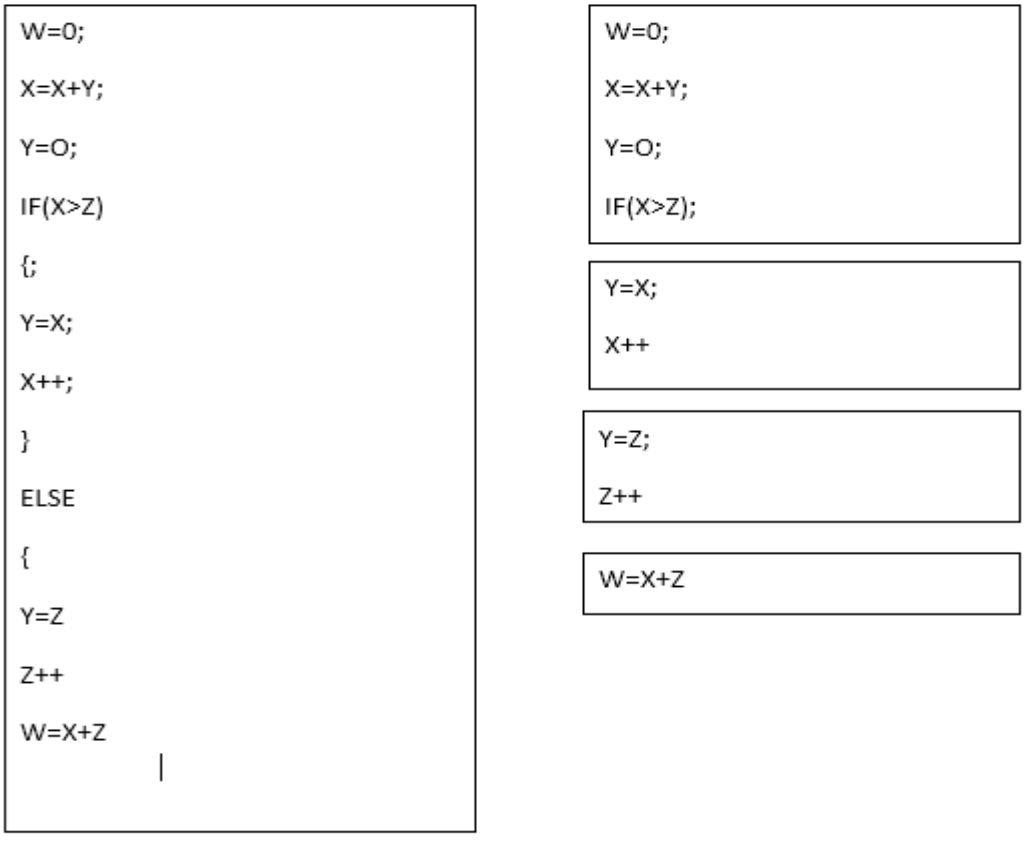
On the other hand, machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy

Basic Block

Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow

control of the program (Wescott, 2013). A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.
We may use the following algorithm to find the basic blocks in a program:
1. Search header statements of all the basic blocks from where a basic block starts:
i. First statement of a program.
ii. Statements that are target of any branch (conditional/unconditional).
iii. Statements that follow any branch statement.

2. Header statements and the statements following them form a basic block.
3. A basic block does not include any header statement of any other basic block. Basic blocks are important concepts from both code generation and optimization point of view. An illustration is shown below:



Source Code Basic Blocks

Figure 1: Source code and Basic block

Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

In lining small functions

This method involves repeatedly inserting the function code instead of calling it. This saves the calling overhead and enable further optimizations. In lining large functions will make the executable too large.

Strength Reduction

There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.

Code hoisting

This is the Moving of computations outside loops, it saves computing time In the following example (2.0 * PI) is an invariant expression there is no reason to recompute it 100 times.

```
DO I = 1, 100
        ARRAY (I) = 2.0 * PI * I
END DO
```

By introducing a temporary variable 't' it can be transformed to:

```
t = 2.0 * PI
DO I = 1, 100
        ARRAY (I) = t * I
END DO
```

Dead store elimination

If the compiler detects variables that are never used, it may safely ignore many of the operations that compute their values. Eliminating common sub-expressions

Optimization compilers are able to perform quite well:

X = A * LOG(Y) + (LOG(Y) ** 2)

Introduce an explicit temporary variable t:

t = LOG(Y)
X = A * t + (t ** 2)

Saves one 'heavy' function call, by an elimination of the common sub-expression LOG(Y), the exponentiation now is:

X = (A + t) * t

Loop unrolling

The loop exit checks cost CPU time.

Loop unrolling tries to get rid of the checks completely or to reduce the number of checks.

If you know a loop is only performed a certain number of times, or if you know the number of times it will be repeated is a multiple of a constant you can unroll this loop.

Example:

```
// old loop
for (int i=0; i<3; i++) {
        color_map[n+i] = i;
}
// unrolled version
int i = 0;
    colormap[n+i] = i;
        i++;
        colormap[n+i] = i;
            i++;
                colormap[n+i] = i;
```

Code Motion

Any code inside a loop that always computes the same value can be moved before the loop.

Example:

```
While (i<= limit-2)
        do {loop code}
```

Where the loop code doesn't change the limit variable. The subtraction, limit-2, will be inside the loop. Code motion would substitute:

```
t = limit-2;
    While (i<= t)
        Do {loop code}
```

Challenges Encountered with optimization

Optimization may include finding a bottleneck in a system – a component that is the limiting factor on performance. In terms of code, this will often be a hot spot a critical part of the code that is the primary consumer of the needed resource though it can be another factor, such as I/O latency or network bandwidth (Klemmt, Horn, Weigert, & Wolter, 2009).

When to perform optimization

Optimization can reduce readability and add code that is used only to improve the performance. This may complicate programs or systems, making them harder to maintain and debug. As a result, optimization or performance tuning is often performed at the end of the development stage. Premature optimization" is a phrase used to describe a situation where a programmer lets performance considerations affect the design of a piece of code. This can result in a design that is not as clean as it could have been or code that is incorrect, because the code is complicated by the optimization and the programmer is distracted by optimizing.

A better approach is therefore to design first, code from the design and then profile/benchmark the resulting code to see which parts should be optimized. A simple and elegant design is often easier to optimize at this stage, and profiling may reveal unexpected performance problems that would not have been addressed by premature optimization. In practice, it is often necessary to keep performance goals in mind when first designing software, but the programmer balances the goals of design and optimization.

METHODOLOGY

In this work, two different codes that perform the same function were analyzed. Here, two different codes are run and observed, two different

codes for a simple Body Mass Index (BMI) calculator that perform the same function on java compiler. The first code used is bulky and memory consuming yet it performed the simple calculations. The second code which is the optimized code is afterward used to perform same operations. The two codes are then be evaluated based on the performance metrics.

PERFORMANCE METRICS

The performance metrics considered in this work are as follows:

- Processing Speed
- Code Size
- Memory Utilization.

IMPLEMENTATION

In this work, code optimization was shown by implementing two programs in java environment that performs the same functions of calculating the BMI of a person. The two programs (BMI calculator) were analyzed based on the number of code lines, execution speed and memory space consumed by each of the program.

RESULT

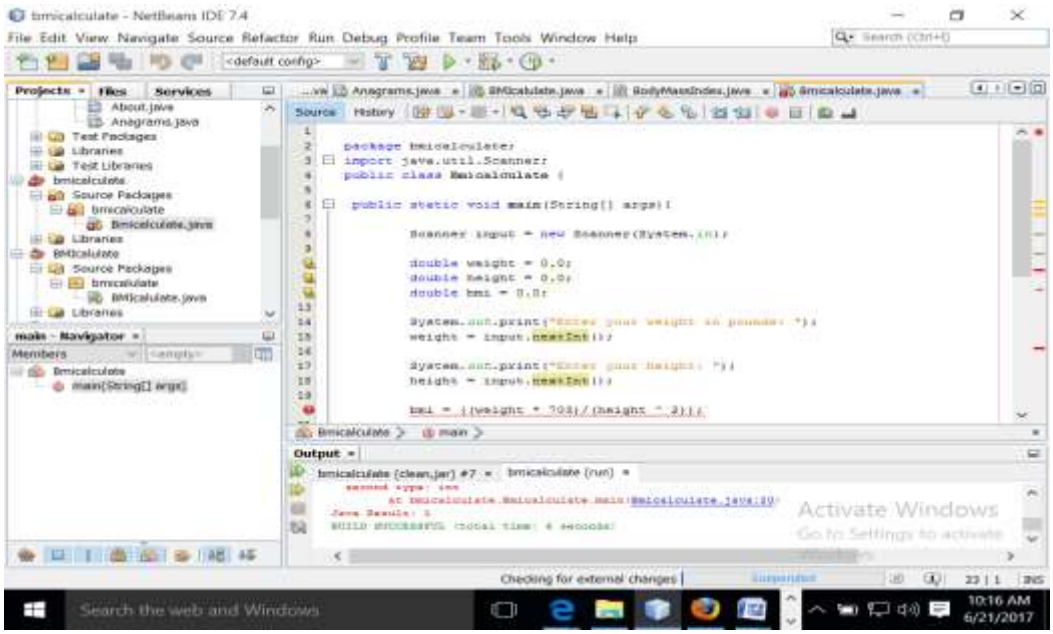


Figure 2: output result for the un-optimized calculator program.

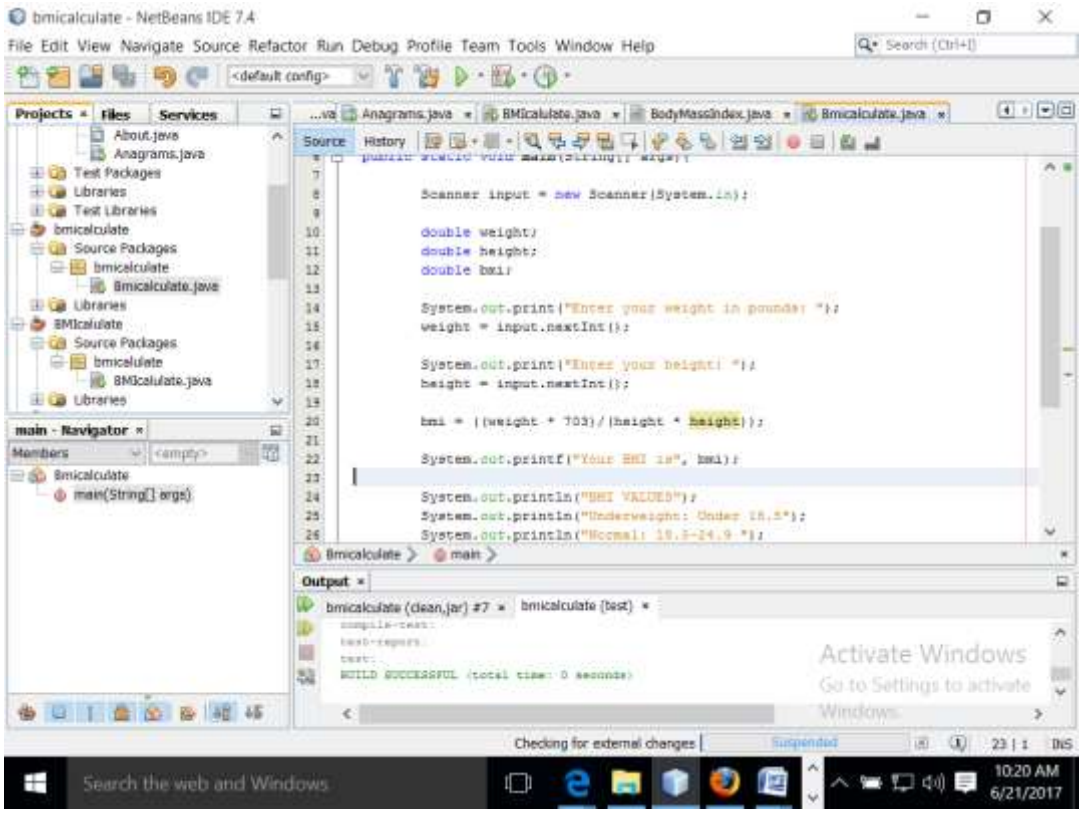


Figure 3: output result for the optimized calculator

DISCUSSION OF RESULTS

Memory utilization: the memory space consumed by the un-optimized program is 268kb while the memory space consumed by the optimized program is 144kb. This indicates that the optimized version of the calculator program consumed less amount of memory space.

Processing Speed: the time taken for the un-optimized program to execute is longer than that of the optimized program. On the average, the un-optimized code takes six (6) seconds slower than the time of the optimized program to complete execution in seconds.

Codes Size: the un-optimized program uses more lines of codes than the optimized version.

CONCLUSION

This work has been able to prove that the value of optimization in programming cannot be over-emphasized. Based on this work we were

able to bring out some major benefits of optimization in programming such as reduced lines of codes, decreased processing time and better utilization of memory resource in java programming language.

RECOMMENDATION

This research work performs code optimization on a simple BMI calculator program. It is recommended that such technique be adopted in the implementation of more complex real life software like online learning and banking software as a measure of improving performance.

References

Annal, E.S and Jessintha, J. P. (2017). Compiler design concepts, worked out examples and mcqs for net/set. Availaible online at https://www.researchgate.net/publication/316560026. L ordine nuovo publication. ISBN: 978-81-93368-26-6

Cheng, Z., Wang, Q., Li, Z. and Rudolph, G. (2019). "Computation offloading and resource allocation for mobile edge computing," 2019 IEEE Symposium Series on Computational Intelligence (SSCI), Xiamen, China, 2019, pp. 2735-2740, doi: 10.1109/SSCI44817.2019.9003106.

Hoos, H. H. (2012). Programming by optimization. Communications of the ACM, 55(2), 70-80.

Klemmt, A., Horn, S., Weigert, G., & Wolter, K.-J. (2009). Simulation-based optimization vs. mathematical programming: A hybrid approach for optimizing scheduling problems. Robotics and Computer-Integrated Manufacturing, 25(6), 917-925.

Rubinstein, R. Y. (1997). Optimization of computer simulation models with rare events. European Journal of Operational Research, 99(1), 89-112.

Schiex, T. (Oct, 2016). Optimization in Graphical Models - NP-complete optimization and its applications. Assemblée générale de l'Association Française d'Intelligence Artificielle, Paris, France. hal-02794159

Srikant, Y. N and Shanker, P (2018). The compiler design handbook: Optimizations and machine code generation. USA. CRC Press.

Wescott, B. (2013). Every computer performance book: how to avoid and solve performance problems on the computers you work with. CreateSpace Independent Publishing platform

Zhang, Q., Wang, R., Yang, J., Ding, K., Li,Y. and Hu, J. (2017). Collective decision optimization algorithm: A new heuristic optimization method. Neurocomputing. 221, 123-137. doi:10.1016/j.neucom.2016.09.068.