International Journal of Engineering Research-Online
A Peer Reviewed International Journal
Articles available online http://www.ijoer.in

Vol.3., Issue.6., 2015
(Nov.-Dec.,)

RESEARCH ARTICLE

ISSN: 2321-7758

AN INTEGRATED APPROACH TOWARDS PERFORMANCE ENGINEERING OF SOFTWARE SYSTEMS: A STUDY

MIRZA MOHD AILEEYA QASIM
Research Scholar, Department of Computer Science and Engineering, Dr. A. P. J. Abdul Kalam Technical University, Lucknow, Uttar Pradesh, India.

MIRZA MOHD AILEEYA QASIM

ABSTRACT
Improving the quality of code in order to make it run efficiently on the hardware is central to code optimization. Some code optimizers may prioritize memory efficiency; some may prioritize time while others may prioritize power consumption. Some might involve a combination of the above three. A general purpose software system can never be fully optimized. This is because a small amount of performance improvement might be left out during the optimization phases or the optimizer might not fully utilize the resources provided. On the other hand, special purpose software systems can be truly optimal. If one tries to reduce the code size, it is possible that the execution time may increase. Vice versa could also happen. So, a tradeoff should be maintained as to what parameter is being given priority. It is also possible that if one uses a faster algorithm, it may drain more power of the system. Thus, all the constraints must be taken into consideration before designing a software system. As soon as an optimal solution for the code optimization problem is found, the process of optimization can be stopped.

Key Words—Automatic Optimization, Manual Optimization, Performance Engineering of Software.

INTRODUCTION

Performance can be improved at various levels of abstraction. Typically, optimization should start from a higher level to a lower level. The highest level being the design level in which the optimizations are performed at a higher level of granularity. Optimizations at this level cannot be changed afterwards because they are the decision making aspects in the beginning. The performance improvements at higher levels are also not easy to change once the decision is made but contribute towards major performance improvements. As we move on towards the lower levels, we have to refine our results continuously. The latter optimizations require more work but are less fruitful whereas the former optimizations require less time but are more fruitful. A good decision made in the beginning can lead to an overall performance improvement once the entire system is optimized at various levels. For very big projects, performance engineering has to be performed after every incremental development.

A program with no performance engineering is very slow and is not always fit for a specific purpose. A PC game which is running on a monitor with refresh rate of 72Hz and giving a frame rate of 15 frames per second will obviously give a very bad performance. The same computer with a graphics card of 2GB and with 120 cores is not being utilized by a program which is otherwise running on a quad core CPU will not give good performance. So,

a program should try to extract every drop of available performance from the hardware. Initially at the time of software development, performance should not be taken into consideration. However, once the software is developed, the second phase should be performance engineering. If we take an example of the Java virtual machine [6], its performance was improved after about a decade.

LEVELS OF OPTIMIZATION

From the highest level of abstraction to the lowest level, the optimization techniques should be applied in a top down manner. A study of how to perform these optimizations is listed as follows:

Design Level

At the design level, the design needs to quantify all of the resources, targets and limitations of the system and once analyzed, the designer has a clear picture in what way he has to optimize. First of all, the high level architecture of the system is designed, and thoroughly analyzed whether it will give a good performance or not. Suppose that the performance of the system which is being developed is overall restricted by the network. Here, the network channel is the main cause of decrease in performance. In such a scenario, the design decision should be such that the developer should minimize the network usage as much as possible. He needs to design the application in such a way which uses less network. As discussed earlier, it all depends on the goal and priority of what is taken into performance consideration. If we are designing a compiler which should compile quickly, a single pass compiler is a good option. On the other hand, if the performance of generated code is our concern, then a slow compiler is beneficial because it will try to optimize the code in all possible ways. The programming language in which we are implementing should also be chosen wisely. It should provide good performance.

Algorithms and Data Structures

Once the high level design is ready, next we choose the algorithms and data structures for our problem. Utmost care should be taken that the data structures we choose should be as efficient as possible. A good and carefully chosen data structure will give good performance as compared to an otherwise unsuitable data structure. Once decided, it will be very difficult to change it afterwards. Abstract data types (ADT) can also be used if changing in near future is required. This will ensure flexibility.

The selection of a good algorithm is also necessary. Either it should work in O(1) time or O(Log n) time or worse i.e. O(n) or O(n Log n) but not more worse. Space and time complexities should also be kept on track. The problems with algorithms of complexity greater than O(n) usually do not scale well. Some algorithms can also cause problems when repeatedly called over and over again.

As soon as the we are done with the asymptotic complexity, we need to check for tradeoff between algorithms. It should be checked whether some hybrid algorithms are available or not so that it can improve performance of the code. Code should also be sampled for smaller inputs to check their performance.

Predefined programming patterns must be adopted in order to make the common case fast instead of wasting much time on it. All unnecessary load on the algorithm should be removed and performance critical places are quantified. With the help of this we get a system which is fast performing in nature. The algorithm should efficiently use the memory, cache and should not perform duplicate calculations.

Source Code Level

After the algorithms and data structures are decided, there comes final implementation in the programming language. The source code forms the basis of all software development. There are some constructs in programming languages that are slower than others, some are faster than others. For example in case of C programming language, while (1) { … } is faster than for(;;) { … } when a loop without condition is to be executed. The first one will always be true whereas the other one requires an unconditional branch. Many of such types of optimizations can be performed by modern optimizing compilers [1]. The overall performance

depends upon the programming language and the target architecture. Compilers also utilize branch predictions to predict the outcome of an execution. Many optimization techniques remove the temporary variables to improve performance.

Build Level

At the time of building the software on a specific hardware, special flags can be used to further tune up the performance of the source code on that specific hardware. Unneeded software features can be easily removed with the help of these flags. Many of the source base Linux distributions such as Gentoo uses the Portage package manager to optimize performance for a specific architecture. Through this feature, advanced branch prediction capabilities can be utilized which are specific to that hardware.

Compile Level

By using a high performance optimizing compiler one can assure himself that the code will be optimized to such an extent such as it will not be optimized beyond the reach of the compiler.

Assembly Level

Once the optimizing compiler has fully optimized the source code, there are still some room for improvement which the compiler just cannot do. The task of the compiler is just to convert the high level language program into assembly language. The assembly program also contains many useless execution and overheads. They all can be removed properly without any loss of consistency to give good performance. This is because the instruction set architecture is specific to the hardware. So, with the help of assembly language, code can be tweaked to produce good performing code. This technique is also known as hand optimization or the ultimate optimization step. Nowadays, compilers are becoming more and more sophisticated and the processors are becoming more complex, so this type of optimization becomes very difficult and is not performed in all of the cases. But the thing to be kept in mind is that it is the most efficient code which can ever run on the hardware.

Nowadays, the code written by software developers is deemed to run on many different processors. As a result of this, the developers do not always optimize for a specific processor. This is because assembler code needs to be optimized for a specific hardware but it will still give worse performance on other processors because the instruction set architecture is different.

So, rather than hand tuning the code, a disassembler is used through which the code is checked and necessary modifications are made in the high level source code itself. The compiled code will be efficient.

Runtime

Java introduced a runtime environment known as HotSpot virtual machine. This featured a nice just in time compiler. Such compilers are used in many other domains as well. They are used in adaptive optimizations in which a just in time compiler is used to dynamically compile portions of code to hardware.

In case of profile directed optimizations [2], the optimizations are performed before the execution of the code and then profiled suitably. Based on the information received from the static analysis, a somewhat better performance can be achieved. Dynamic profile guided optimizations are called adaptive optimizers.

New techniques of self-modifying code are capable of changing itself at runtime depending upon the circumstances. This type of optimizations was popular in case of assembler programs.

There are some processors which perform out-of-order execution, specular execution and branch prediction on the source code. They possess features such as instruction pipelines which modern compilers can easily take advantage of and can perform instruction scheduling.

Platform Dependent and Independent Optimizations

There are two main categories of code optimization. One is platform dependent and the other one being platform independent. The good thing about platform independent optimizations is that these optimizations can be used for any platform. But this is not the case with platform dependent. Platform independent techniques might include loop unrolling, loop, reductions etc. whereas

platform dependent need to take advantage of the features specific to a particular architecture and different code is needed to be produced for different architectures. Machine independent optimizations reduce the number of instructions to be executed by the processors whereas machine dependent techniques use instruction scheduling, data level parallelism, instruction level parallelism, data locality optimizations etc. These techniques might be different on different architectures.

FACTORS INVOLVED IN OPTIMIZATION

There are many factors involved in optimizing a program. Some may be programming language dependent, some hardware dependent and some mathematical.

Strength Reduction

A computation can take place in more than one way. Strength reduction is a way to increase the efficiency while still maintaining the same functionality. Consider an example to add numbers from 1 to N:

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

Assume that there is no arithmetic overflow, the same code can be written more efficiently ad:

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

To make the task computationally efficient, an optimizing compiler automatically does this type of optimization. The compiler selects an algorithm through which it is able to do this. This introduces a significant degree of performance. It is not guaranteed that applying optimization processes as in the above case will actually increase performance. The technique should be selected carefully so that rather than increasing performance, it is in fact slowing down the system if the value of N is very small. It may be also possible that the hardware is very fast in computing loops other than multiplications.

Tradeoffs

Specialized algorithms can also be used which utilize tricks and tweaks and using more comprehensive algorithms and doing tradeoffs. A program which is optimized fully is very difficult to understand because it may contain more faults than the previous version. And also as previously discussed optimization of code results in decreased maintenance capability.

There are many parameters on which performance can be improved. It can be either power, time, bandwidth, memory or any other resource. There are chances that increase in performance of one factor decreases the other, so a tradeoff has to be maintained. For example, memory consumption is increased by increasing the size of the cache but also improves the performance of the runtime system.

Some instances are there in which the programmer must decide to make software better while performing optimizations by making other parameters less efficient. Suppose that a new benchmark is released and the programmers start testing the performance of their software on the benchmarks. Just only to make their software better, they make benchmark specific optimizations which are of course fake but they beat the benchmark.

Bottlenecks

A bottleneck is a region which is a cause of degradation in performance. An optimization technique may also find bottlenecks in the system. This can also be termed an optimization technique. In the language of code, it is called as a hotspot or a primary consumer. In case of network, a bottleneck may be a type of latency in the network bandwidth.

As per Pareto distribution principle, 80% of the resources are used by around 20% of the operations. If we try to get a better approximation of the case, it will be found that 90% of the execution time is being spent on only 10% of the code. It may be also possible that an algorithm with less instructions to execute and too much overhead is induced on that program. The time taken to input data, setup time and complex scheduling algorithm can outweigh the benefit of good algorithm. This is

the reason why adaptive and hybrid algorithms are used instead of these slow performing ones.

It has also been seen that just adding more amount of RAM has actually speeded up execution. For instance, if a disk is used which performs poorly, and our algorithm is reading text from the hard disk continuously, then in that case even we are using a better algorithm, the performance is not going to get any better. However, if we increase the memory size and read a file at once from the memory, it will consume less time. Cache performance can also be significantly increased just by adding enough memory.

When to optimize

As already mentioned, optimizing the code reduces its readability and it can also include code which can increase performance. Because of this significant change in the code, the program becomes even harder to debug. Thus, the performance tuning and optimizations should be applied at the end of the development process.

Donald Knuth says, "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%" [3].

This quote is also attributed to Tony Hoare but he claims that he has not said this.

Donald Knuth also says that, "In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal and I believe the same viewpoint should prevail in software engineering" [3].

Performance improvement should have no place in the design of the code. The resultant design produced seems unclean and may also produce suboptimal results in some cases. The process of including performance improvement at the time of design is termed as premature optimization. The code can become complicated and less intelligible if the programmer introduces optimization at lower level.

Amdahl's Law [5] should always be referred when optimizing a specific part of the application because as per the law, the overall performance depends upon a specific sequential fraction of the code. Performance can only be improved if and only if the sequential fraction is optimized well. Performance will never reach above the sequential fraction. This theoretical law helps in getting performance considerations without executing on hardware.

A good approach will be to first compile the code, then profile or benchmark the code, and based on the results of the profile, optimize the code again. At this stage optimization is quite easy. Profiling may expose potential capabilities of the code and also may reveal possible bugs which may be removed in order to get performance in the final execution rather than premature optimization. Performance goals should be kept in mind at all times while developing software but in the initial stages, the programmer should not prioritize performance rather than design.

Macros

While the code is being developed programmers use macros in different forms. Programming languages such as C/C++ use macros. Macros are particularly developed using mainly token substitution or parse time substitution. Type safety can also be ensured by using inline functions. Inline functions can be further optimized at compile time and inserted in the code. Many programming languages which are functional in nature macros are safer to use with macros because computations are performed at parse time and not link time.

In many functional programming languages macros are implemented using parse-time substitution of parse trees/abstract syntax trees, which it is claimed makes them safer to use. Since in many cases interpretation is used, that is one way to ensure that such computations are only performed at parse-time, and sometimes the only way.

The Lisp language first started providing macros. Therefore, macros are also called lisp like macros. Using template metaprogramming in C++, same effect can be achieved.

Current trends have moved all the work to compile time. There is a difference between C macros, lisp like macros and C++ metaprogramming

templates. The difference is that C++ macros can perform complex calculations whereas lisp like macros can only be used for substitution. Also, iteration and recursion are not supported by C macros there they aren't Turing complete. In conclusion, it is very difficult to predict the impact on tools on software systems.

Automated and Manual Optimizations

Optimizations can be either manual or automatic. If a person performs it by hand, it becomes manual optimization. If it is being optimized by a compiler, then it is termed as automatic optimization. Profits in optimizations are a little less in local optimizations and much more in global optimizations. Expert algorithmic methods are devised in order to find superior algorithms.

The optimizations which a compiler cannot perform are performed manually by programmers. This include the entire software system which is taken into consideration. The code may be changed manually here or there, code size may be increased or otherwise made to use more memory in order to increase performance. The cost of manual optimization is much higher than automatic optimization by compilers.

In order to find out the bottlenecks, a profiler or a performance analyzer might be used. These profilers give useful information regarding the amount of resources being consumed. If an unimportant piece of code is optimized, it might even lead to no performance improvement. People claim that they have a clear idea about the performance of the system but in fact they are not always accurate.

As soon as the bottlenecks are removed, then the algorithm is designed so that bottlenecks no longer remain. Usually, the algorithms are redesigned keeping in mind the bottlenecks while also, not deviating from the application objective and also giving good performance. The algorithm can be transformed from a generic one to a specific one. If a quick sort algorithm is being used and it is known that the elements of the array are already arranged in a specific pattern, then a custom made algorithm can be used which can exploit the features effectively.

Once the programmer is absolutely certain that he has used the most efficient algorithm, from there onwards, he can start to optimize the code. He can apply loop unrolling, data locality optimization, clever selection of data types and also can transform the way in which the computations are taking place. Normally, a compiler automatically does all things for the programmer, but still there are some optimizations that cannot be performed by the compiler and needs manual attention.

It is also possible that due to programming language limitations, some optimizations just cannot be performed. There are some performance critical portions of code that need to be written in assembly language. An example of such a programming language is C in which many of the low-level routines can be written in assembly language so that the code gives good performance. Adding inline assembler routines also allows direct access to the hardware and thus increased efficiency.

An expert programmer would argue about rewriting portions of code for performance. However, many studies have suggested that rewriting the code pays off and it should be considered as a rule of thumb or more appropriately, the 90/10 law which states that 90% of the time is spend optimizing 10% of the code [4]. So, in a nut shell, applying 90% of the effort in 10% of the code may result in huge performance improvements if the correct bottlenecks are found.

Manual optimization can also affect the readability of code because clever optimization techniques can make the code tricky to understand. Thus, whenever the code is being optimized, each and everything should be thoroughly documented using comments so that future developments can be easy to accomplish.

A special program named "optimizer" can also be used for code optimization. These optimizer modules are in-built inside compilers which apply various optimization techniques on the code. These optimizers can either be machine dependent or independent.

Nowadays, only compilers allow automated optimizations. Because compiler optimizations cannot go beyond a certain extent, some special optimizers are used which take in a programming language description and generate a custom built optimization step. These optimizers are better known as code transformers rather than optimizers. These transformers are most suitable in languages like C/C++.

Many programming languages use an intermediate representation to optimize the code in a machine independent manner. Distributed computing, grid computing and cloud computing systems allow optimizations of the entire system by moving tasks from one node to the another at idle time.

Time Taken for Optimization

It is also possible that the time taken for optimization is quite large and becoming an issue in itself.

There is a high probability that the code generated after optimization might become worse or some new bugs might have crept in there. In this case, a code which was running perfectly fine previously might stop working. Other drawback is that optimized code is not easier to maintain because it is less intelligible and difficult to read. So, every step should be taken wisely whether optimization is required or not because often there is a tradeoff.

An optimizer should itself be optimized suitable and care should be taken that it does not take much time in optimizing other programs. Therefore, the compilation time should be compensated by optimization time if the source code is quite large. Particularly for just in time compilers, the performance of the runtime combined with execution of target code can give good performance.

Conclusion

In this study, a specific top-down approach was adopted to improve performance of software at various levels of abstraction. The techniques specific to compilers were not included because that requires another article to explain. Only those techniques which can be done manually outside the compilation process were discussed. It is widely accepted that applying these approaches right from the design phase can lead to significant improvements in performance.

References

[1]. Ken Kennedy and John R. Allen, Optimizing compilers for modern architectures: a dependence-based approach, Morgan Kaufmann Publishers Inc. San Francisco, CA, USA 2002.

[2]. Dehao Chen, "Taming hardware event samples for fdo compilation", Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization, 2010.

[3]. Donald Knuth, "Structured Programming with go to Statements". ACM Journal Computing Surveys 6 (4): 268 (December 1974).

[4]. Bob Wescott, "Every Computer Performance Book", April 2013.

[5]. Gene M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities”, AFIPS spring joint computer conference, 1967.

[6]. Bill Venners, Inside the Java Virtual Machine, McGraw-Hill, Inc. New York, NY, USA 1996.