Code Optimization Techniques

By

VYOM JAIN
15BIT067

DEPARTMENT OF INFORMATION TECHNOLOGY
Ahmedabad 382481

CODE OPTIMIZATION TECHNIQUES

Seminar

Submitted in fulfillment of the requirements

For the degree,

Bachelor of Technology in Information Technology

By

VYOM JAIN
15BIT067

Guided By
PROF. MALARAM KUMHAR
[DEPARTMENT OF INFORMATION TECHNOLOGY]

DEPARTMENT OF INFORMATION TECHNOLOGY
Ahmedabad 382481

CERTIFICATE

This is to certify that the Seminar entitled “Code optimization Techniques” submitted by Vyom Jain (15BIT067), towards the partial fulfillment of the requirements for the degree of Bachelor of Technology in Information Technology of Nirma University is the record of work carried out by him/her under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination.

Prof. Malaram Kumhar
Assistant Professor,
Department of Computer Engineering,
Institute of Technology,
Nirma University,
Ahmedabad

Dr. Madhuri Bhavsar
Dept. of Information Technology,
Institute of Technology,
Nirma University,
Ahmedabad

ACKNOWLEDGEMENT

I would like to thank Prof. Malaram Kumhar for guiding me at each step every week about when to do what. This helped me to progress in the correct direction at the correct pace. I would also like to express my gratitude toward Dr. Ankit Thakkar, who always kept a check on my progress every week and for forcing me to work only on the seminar and not on any other thing, during the lab hours. It helped me to focus on the seminar and to give it the attention that it needed.

ABSTRACT

This report includes to all the progress that I made in order to understand the need and implementation of the different kinds of code optimization techniques used in the industry. This includes the improvements done at programmer level, source code level and compiler level. The basic requirement of the technique is that the output should be exactly the same and thereby it should offer some positive change in the performance parameters, viz. time and space.

CONTENTS

CHAPTER - 1
INTRODUCTION

1.1 Code optimization:

Code optimization can be any method of modification in the code to improve its quality and efficiency. Optimization may be done so that it becomes smaller in size, demands less memory, executes in less time, or performs less number of input/output operations.

1.2 Rules to be followed:

1.2.1 The basic requirement is that an optimized program must have the same output as its non-optimized version.

1.2.2 The optimization technique should decrease the duration of execution of the program or the program should use less storage, memory or operations.

1.2.3 Optimization itself should not slow the program down.

1.3 Efforts before optimization

1.3.1 The programmer can rearrange the code or employ a different algorithm to develop the code.

1.3.2 Compiler can modify the intermediate code by improving loops and address calculations, after the intermediate code has been generated.

1.3.3 The compiler can make use of memory hierarchy and CPU registers, while it produces the machine code for the target system.

CHAPTER - 2
TYPES OF CODE OPTIMIZATIONS

2.1 Peep-hole Optimization:

In this procedure, the programmer examines the parts of code which can be replaced by a single instruction. This may reduce the accuracy of the result but this can be done when accuracy is less important in the situation.

E.g. - multiplying by 2 can replaced by a left-shift operation.

2.2 Local Optimization:

In this procedure, we tackle errors on a locality basis, i.e. on a block criteria. This reduces the storage requirements and also the time taken for execution. But there is a compromise that no information is saved for future reference.

2.3 Global Optimization:

These are also called “intraprocedural methods”, this provides all the functions with all the information without the need of passing them as arguments. But while computation worst case scenario is needed to be considered.

2.4 Loop Optimization:

These optimizations act on the statements having a loop. These techniques play a very important role in optimization of code because algorithms spend most of the time in executing loops.

2.5 Prescient store optimizations:

Stored operations are allowed to take place earlier than they would occur. Intention is to perform code rearrangements that preserve the meaning of program.

2.6 Interprocedural, whole-program or link-time optimization:

Greater information extracted from the source code is more effective than having only local information. This kind of optimization can enable new techniques like function and class inlining.

2.7 Machine Code Optimization:

Limited scope optimization techniques, such as macro compression are more effective when the entire machine code is available for analysis. This is used in that use case.

CHAPTER - 3
FACTORS AFFECTING OPTIMIZATION

3.1 Architecture of the target CPU:

3.1.1 RISC vs CISC: CISC instruction have more variation with variable addressing modes and various ways to manipulate the data, hence its instruction code cannot be assumed to have a constant length. Whereas, RISC instructions are limited only to read and store when communicating with the memory. Therefore, instructions are of constant length. Compilers need to know the respective costs of the process in each type of the processors and then choose the best instruction sequence.

3.1.2 Pipelines: this allows the CPU to carry out multiple operations at once. According to the number of channels available in the pipeline structure the compiler needs to allocate and schedule the tasks.

3.1.3 Number of Functional units: Different CPUs have different configuration and number of functional units, i.e., ALU and FPU. The compiler needs to schedule and deploy processes according to the number of ALUs or FPUs available.

3.2 Architecture of the Machine

3.2.1 Cache Size and types: the compiler should have an idea of the size of the cache memory as it becomes an issue of concern when techniques like loop unrolling and inline expansion are used. These increase the size of the generated machine code.

3.2.2 Cache/memory transfer rates: this is rarely important but is useful for high accuracy and big data programs.

3.3 Intended use of the generated code:

3.3.1 Debugging: when debugging the code, the user will run it again and again, when the step by step procedure is considered and the optimization technique reorders the code, then it becomes very difficult to associate a line of code with the output. This confuses both the debugging tool and the programmer.

3.3.2 General purpose use: these require flexible optimization techniques as the packaged softwares are used on a variety of devices with very different configurations, i.e. with different cache, memory, architecture and timing.

3.3.3 Special purpose use: we can employ heavy optimization techniques as the code is to be used only on a limited type of devices all whose specifications can be known beforehand.

3.3.4 Embedded systems: these are common to special usecase in terms of usage, but these have more priority on system reliability than speed, hence the techniques should try to reduce the code size at the cost of speed.

CHAPTER - 4

LOOP OPTIMIZATION

4.1 Induction variable analysis:

The variables used for iterations or for any other purpose inside a for or while loop, needs to be paid a close attention as more of these will increase the complexity of the program manifold.

```
int A[n];
int B[n];
int i, j, k;

for(i=0, j=0, k=0;
i<n; i1++)
     A[j++]=B[k++];
```

```
int A[n];
int B[n];
int i;

for(i=0; i<n; i++)
     A[i] = B[i];
```

4.2 Loop fission or loop distribution:

This technique breaks or distributes the loop into segments with the same index but using only part of the loop. This improves locality of reference.

```
for(J=0;J<N;J++)
  {
   A[J]=A[J]+B[J-1];
   B[J]=[J-1]*X + Y;
   C[J]=1/B[J];
   D[J]=SQRT(C[J]);
  }
```

```
     for(I=0; I<N-1;
I++)
      {
      B[I+1]=C[I]*X +
Y;
      C[I+1]=1/B[I+1];
      }

     for(I=0; I<N-1;
I++)
      A[I+1]=A[I+1] +
B[I];
     for (I=0; I<N-1;
I++)
      D[I+1]=
SQRT(C[I+1]);

I=N+1;
```

4.3 Loop fusion:

When two loops iterate over the same length then they can be combined to reduce loop overhead.

```
     for (I = 0; I <
300; I++)
      A[I] = A[I] + 3;

     for (I = 0; I <
300; I++)
      B[I] = B[I] + 4;
```

```
for (I = 0; I < 300;
I++){
 A[I] = A[I] + 3;
 B[I] = B[I] + 4;
 }
```

4.4 Loop inversion:

The jumps caused in loops usually lead to pipeline stall, to reduce this we should reduce the number of jumps. Using a do…while loop instead of a while loop wrapped in an if condition can serve our purpose.

```
int i = 0;
while (i++ < 1) {
     //do something
}
```

```
int i = 0;
if (i++ < 1) {
     do {
          //do
something
     } while (i++ < 1);
}
```

4.5 Loop interchange:

Exchanging the inner loop with the outer one sometimes improves the locality of reference. Particularly useful when loop variables index into an array.

```
for (i1 = 0; i2 < 100;
i1++)
 for (i2 = 0; i2 <
100; i2++)
 for (i1 = 0; i1 <
5000; i1++)
 x[i1][i2] = 2 *
x[i1][i2];
```

```
for (i1 = 0; i1 < 100;
i1++)
 for (i1 = 0; i1 <
5000; i1++)
 for (i2 = 0; i2 < 100;
i2++)
 x[i1][i2] = 2 *
x[i1][i2];
```

4.6 Loop-invariant code handling:

If a quantity inside a loop remains the same or has the same impact on every iteration, can be removed to hugely improve the performance of the code.

```
for (i3 = 0; i3< 100;
i3++){
     int a = 2;
     x[i1][i2]=2 *
x[i1][i2];
}
```

```
for (i3= 0; i3< 100;
i3++){
 x[i1][i2] = 2 *
x[i1][i2];
}
int a = 2;
```

4.7 Loop nest optimization:

Some algorithms have too many direct memory references, this optimization technique improves the usage through efficient usage of cache and by using loop interchange.

4.8 Loop unrolling:

This one duplicates the operations to be performed inside the loop thereby reducing the number of times the condition is checked and the number of times the jump instruction is called, thereby improving performance at the pipeline level.

```
for (i = 0; i < 100;
i++)
 g ();
```

```
for (i = 0; i < 100; i
+= 2)
{
 g ();
 g ();
}
```

4.9 Software pipelining:

The loop is restructured so that the work to be done in the loop is split into many parts and are executed over many iterations. This loop improves performance where the values are loaded and used again and again.

```
for (i=1, i<100, i++)
{
x = A[i];
x = x+1;
A[i] = x
}
```

```
for (i=1, i<98, i++) {
store A[i]
incr A[i+1]
load A[i+2]
}
These can be executed
in parallel,
no dependency exists.
```

4.10 Loop unswitching:

This method moves the conditional statements from inside to outside of the loops so as to reduce the number of times the conditional statement is called.

```
for (j = 0; j < N;
j++)
 if (a)
 A[j] = 0;
 else
 B[j] = 0;
```

```
if (a)
 for (j = 0; j < N;
j++)
 A[j] = 0;
else
 for (j = 0; j < N;
j++)
 B[j] = 0;
```

CHAPTER - 5
DATA FLOW OPTIMIZATIONS

5.1 Common subexpression elimination:

```
int a;
int b;

double c = (a+b)-(a+b)/4
```

In the above snippet, (a+b) is a common subexpression. The compiler will understand this and will reuse the previously computed value.

5.2 Constant folding and propagation:

This technique suggests to replace subexpressions of constants with constants which will decrease the number of computations.

```
x = 3;
y = x+4;
```

```
x = 3;
y = 7;//propagation
```

```
int f (void)
{
 return 3+5;
}
```

```
int f (void)
{
 return 8;//folding
}
```

5.3 Alias optimization:

Remove unrelated pointers by specifying which variables can alias which variables.

```
void f (short *p1, int
*p2)
{
 int a, b;
 a = *p2;
 *p1 = 3;
 b = *p2;
 g (a, bb);
}
```

```
void f (short *p1, int
*p2)
{
 int a;
 a = *p2;
 *p1 = 3;
 g (a, a);
}
```

5.4 Dead Store elimination:

Remove the lines of code which will not eventually get executed or which are redundant and don't cause any change in the variables or the functions in the program.

```
int glob;
void f ()
{
 int a;
 a = 1;     //dead store
 glob = 1; //dead store
 glob = 2;
 return;
 glob = 3; //unreachable
}
```

```
int global;
void f ()
{
 glob = 2;
 return;
}
```

CHAPTER - 6

SSA (STATIC SINGLE ASSIGNMENT) BASED OPTIMIZATIONS

6.1 Global value numbering:

It determines which values are computed via identical expressions. It can identify more redundancy than common subexpression elimination.

6.2 Sparse conditional constant propagation:

This one combines dead code elimination, constant propagation and folding and improves on what could have been the result on running them separately.

CHAPTER - 7
CODE GENERATOR OPTIMIZATIONS

7.1 Register Allocation:

An interference graph is created which determines which variables are used the most often. The ones which are used the more frequently should be placed in the processor registers.

7.2 Instruction selection:

The instruction selector needs to make an efficient selection of operations specially in CISC architecture which provides many-many addressing options as well.

7.3 Instruction scheduling:

Specially for new pipelined systems, this one avoids empty spaces called bubbles from the cycle of execution of macro operations.

7.4 Code factoring:

If sequence of instructions in several code snippets is identical then these can be called through a shared subroutine.

7.5 Trampolines:

Some systems may have relatively smaller subroutine instructions for accessing a low memory. Space can be saved by calling subroutines in the main function/method of the code. Jump instruction can use memory routines at any address. This helps to save space.

CHAPTER - 8
FUNCTIONAL LANGUAGE OPTIMIZATIONS

8.1 Removing recursion:

Recursion is often expensive both in terms of size and time. As it uses stack to store the activation record of each call, as the number of calls increases, it increases the size of the stack. These can be converted to iterative, which do not use the stack phenomena and hence offer very less space complexity.

8.2 Deforestation (data structure fusion):

It is possible to combine different recursive functions which produce and consume some temporary structure so that the data is passed without wasting the time used in constructing the data structure.

CHAPTER - 9
OTHER OPTIMIZATIONS

9.1 Bounds checking elimination:

Languages like Java check for ArrayOutOfBounds Exception every time the program is compiled, to see that no incorrect indexing is encountered at runtime. This check is very time consuming and can act as a bottleneck in scientific codes. This technique eliminates that check at compile time and is useful when the coder has already checked that no incorrect indexing is occurring.

```
{
int A[], not_used;
A = new int[100];
not_used = A[0];
not_used = A[1];
...
}
```

```
{
//once checked for
ArrayOutOfBounds other
// statements can be
optimized
away.
}
```

9.2 Factoring out of invariants:

This technique prevents the use of statements which will be executed irrespective of a condition being true or false. Such a statement can be written just once out of the if…else block. If a constant assignment expression comes up in a loop then it can be written just once out of the loop.

9.3 Inline expansion or macro expansion:

When the number of times a function called is too few, it can be expanded in the main body of the code, it increases space but saves the overhead related to procedure calls and returns. This is also useful when small procedures are called a large number of times.

9.4 Macro Compression:

if you are running low on space and willing to lose some performance at its cost, then this is really helpful. Divide your program into subprograms and call them wherever necessary. This makes the code modular, reusable and saves space.

CHAPTER - 10
PROBLEMS WITH OPTIMIZATION

#1. Optimized compilers do not improve algorithmic complexity.

#2. Compilers usually have to make a trade-off for a variety of conflicting objectives such as space vs. time.

#3. A compiler only deals with a part of the full code, often a single class or c file not the entire project.

#4. Optimization is a little time consuming itself.

#5. Complex phases of optimization, often make it difficult to find an optimal sequence for application of these optimization phases.

CHAPTER – F
SUMMARY AND CONCLUSION

This report discussed the need and implementation of code optimization. We first came to define what are the ground rules for a fair optimization technique. Then we classified them in a broader sense. After having an overview of the techniques, we dived into the factors which can influence our choice of the techniques that we make.
A detailed description of each technique and its subtypes were discussed with examples explaining the usage and the actual implementation of most of them. The problems related to it were discussed after that, which comprises of the challenges and shortcomings of the currently used practices for code optimization.

As a conclusion, code optimization is a process which requires the actual context of the problem and the priorities of the intended use case. It requires a tradeoff between space and time requirements. What looks like an optimized code or compiler for a program may give disastrous results in some other application. Therefore, it is instrumental to choose wisely.

APPENDIX

Below is a list of the useful websites that were used in the making of this project under the guidance of the concerned faculty.

1. https://en.wikipedia.org/wiki/Optimizing_compiler

2. All the examples of code optimizations were inspired from the website http://www.compileroptimizations.com

3. Some content is taken from my own presentation: https://docs.google.com/a/nirmauni.ac.in/presentation/d/1BJ0Eem_j1FmFcJUYIXJIi4SphLO6XzS8pAmF37SsHJk/edit?usp=sharing