ISSN:0975-9646

Optimal Code Compiling in C

Nitika Gupta
M.Tech (I.T) Student
Banasthali Vidyapith
Rajasthan, India

Nistha Seth
M.Tech (C.S) Student
Banasthali Vidyapith
Rajasthan, India

Prabhat Verma
Assistant Professor
CSE Department
HBTI, U.P, India

Abstract- **In the present era, developing high level application programs has been a major concern for the programmers. During the design phase of software application, the functionality of developed programs has always been the area of consideration. Different compilers may have in built mechanism to speed up the programs, but these compilation process may adversely affect the length of the code and hence slower the execution of highly developed applications. Thus, to enhance the performance of the large complex applications, several code optimization techniques are being provided by the compilers in C. These code expansion techniques are predominated over manual coding techniques as they help in speeding up the process of program execution in such a way that both time and reserved memory space can be effectively utilized. In this paper we discuss about our implemented work on Code Optimization using two techniques: "Dead Code Elimination" and "Inlining". Our implemented work is automatic procedure which eliminates the chances of errors that are quite possible in manual procedures. We have verified the code optimization performance using code complexity measurement tools. Results obtained are quite satisfactory as compare to existing methods.**

Index terms- **Code Optimization techniques, complexity, compilers, CCCC tool, functions, software.**

I. INTRODUCTION

In compiler design, Optimization is the process of transforming a piece of code (un-optimized code) to make more efficient without changing its output or side effects. A program may be optimized so that it becomes of a smaller size, consumes less memory, executes more rapidly, or performs fewer input/output operations. Optimization can be performed by automatic optimizers or programmers. An optimizer is either a specialized software tool or a built-in unit of a compiler. Optimization is classified into high-level and low-level optimization. High-level optimization are generally performed by the programmer who handles abstract entities (functions, procedures, classes, etc.) and performed at the level of elementary structural blocks of source code- loops, branches, etc. Low-level optimizations are performed at the stage when source code is compiled into a set of machine instructions, and it is at this stage that automated optimization is usually employed. Optimization includes finding a bottleneck, a critical part of the code which is the primary consumer of the needed resources [1]. The code optimization mainly concerns on correctness that means optimization does not change the correctness of generated code. The criterion of code optimization must preserve the semantic equivalence of the program and the algorithm should not be modified. Transformation, on average should speed up the execution of program. In compiler optimization theory, the compiler optimization basically refers to the program optimization to achieve performance in the execution. Program optimization refers to the three aspects:-

i. Frontend: a programming language code.
ii. Intermediate code: an assembly language code generated by the compiler appropriate to the programming language.
iii. Backend: the specific machine or object code generated from the assembly language code for the actual execution by the compiler.

II. EXISTING APPROACHES OF OPTIMIZATION

A. Need for Optimization?

Since process of optimization takes extra time than the actual time involved in coding, our area of focus should be more on that 10% executed time- critical program rather than considering of implementation of whole program These fragments of code created a congestion in the process of implementation and hence can be detected by special utilities profilers which can measure the execution time of various parts of the program. Such optimization methods are done only during the stage of "complex programming" and therefore is a mixture of several optimization approaches such as, refactoring and debugging: simplification of "queer" constructs like strlen(path.c_str()), logical conditions like (a.x != 0 && a.x != 0), and so on. Profilers are of little help with this kind of optimization and hence such issues can be resolved by the usage of statistics analytics tools. Such inefficient code may result due to programming errors and hence code fragmentation is done to detect every part of program. These tools analyze each fragment of the code and hence generate the warning messages.

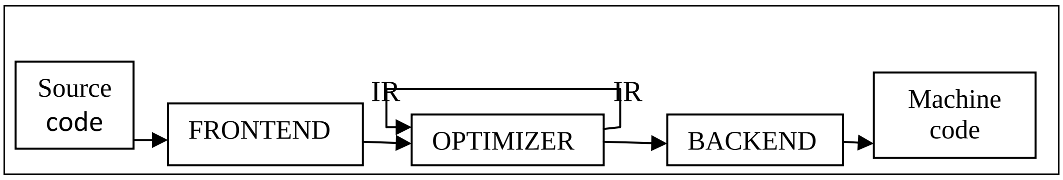


Figure 1.Code Optimization

B. Which code to optimize?

Code optimization done manually creates problem: one doesn't only need to know how exactly optimization should be done, but also what particular part of the program should be optimized. Due to various reasons (slow input operations, the difference in the working speed of a human operator and a computer, and so on), 90% of the execution time of a program is spent executing only 10% of the code. Since optimization takes more time developing the program, one should better focus on optimizing these time-critical 10% of code rather than try to optimize the whole program[11]. These code fragments are known as bottlenecks and can be detected by special utilities - profilers - which can measure the time taken by various parts of the program to execute.

The process of optimization can reduce readability and include more of coding thus enhancing the strength of program. However, in case of complex programs such process of optimization is difficult to achieve leading to slower debugging. Thus optimization is done at the end of process of development of application. Initial optimization of code lead to increase in the level of programming making it more complicated which interrupt the programmer. The process of code expansion should be done keeping in mind the time involved and influence of code on the system performance [9]. The better approach is to perform designing before coding leading to code fragmentation and hence avoid the unexpected programming problems. Such form of developed code after design phase is being clear and precise, thus outperforming the cost involved in maintaining it.

C. Merits of Optimization

- Code optimization is a set of methods of code modification to improve code quality and efficiency.
- A program may be optimized so that it becomes of a smaller size.
- A program may be optimized so that it consumes less memory.
- An optimized code executes more rapidly.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Performs fewer input/output operations.
- Optimization gives high quality code with best complexity (time and space) without affecting the exact result of the code.

.

Table I. PREVALENT TECHNIQUES OF OPTIMIZATION

Techniques	Definition	Examples	
		Before	**After**

DEAD CODE ELIMINATION	• Repeated instructions considered 'Dead'
	• Can be removed[14]
	• tmp1 = tmp2 + tmp3;
	• tmp1 dead
		int f(int x) {
		return x+1;
		...
		}
			int f (int x)
			{
			return x+1;
			}

INLINING	• contents of a function are "inlined", basically, copied and pasted instead of a traditional call to that function[5]
	• avoids the overhead of function calls.
		int add (int x, int y)
		{ return x+y;
		}
		int sub(int x,int y)
		{
		return add(x , -y);
		{
			int sub (int x, int y)
			{
			return x-y;
			}

CODE MOTION	• identifying bits of code that occur within loops, but need only be executed once during that particular loop.[5]
	• expensive re-evaluation will be potentially avoided
		void f(int a, int b)
		{ int i;
		for (i=1; i<10; i++)
		{ ar[i]=a+b;
		}
		}
			void f (int a, int b)
			{ int i;
			int tmp =a+b;
			for (i=1; i<10; i++)
			{ ar[i]=tmp; }
			}

COMMON SUB-EXPRESSION	• common function: two operations produce same results[2]
	• Recomputing the expression can be eliminated
		i = x + y + 1;
		j = x + y;
			t1 = x + y
			i = t1 + 1;
			j = t1;

STRENGTH REDUCTION	• Computationally expensive operations by simpler ones.
	• Application: simplify multiplication by index variables to additions within loops[13]
		t := b * c
		FOR i := 1 to 10000 DO
		BEGIN
		
		a := t
		
		...
		END
			t := b * c
			d := 0
			FOR i := 1 TO 10000 DO
			BEGIN
			d := i * 3
			a := t
			d := d + 3
			…
			END

LOOP UNROLLING	• Loop unwinding
	• Interpreting the iterations into a sequence of instructions which will reduce the loop overhead.
		int i = 0;
		while (i < num) {
		a_certain_function(i);
		i++;
		}
			int i = 0;
			while (i < num) {
			a_certain_function(i);
			a_certain_function(i+1);
			a_certain_function(i+2);
			a_certain_function(i+3);
			i += 4;
			}

CONSTANT FOLDING	• Evaluate constant expressions at compile time. • Only possible when side-effect freeness guaranteed.	C:=1+3 True not	C:=4 False

D. COMMON OPTIMIZATION TECHNIQUES

Optimization techniques are used to improve the speed of computer program. It focuses on minimizing time spent by the CPU and gives sample source code transformations that often yield improvements. Table 1gives a brief overview of these optimization techniques

E. CCCC TOOL

CCCC is a tool for the analysis of source code in various languages (primarily C++), which generates a report in HTML format on various measurements of the code processed. Although the tool was originally implemented to process C++ and ANSI C, the present version is also able to process Java source files, and support has been presented in earlier versions for Ada95. The name CCCC stands for 'C and C++ CodeCounter'. Measurements of source code of this kind are generally referred to as 'software metrics', or more precisely 'software product metrics'. CCCC has been developed as freeware, and is released in source code form. Users are encouraged to compile the program themselves, and to modify the source to reflect their preferences and interests. The simplest way of using CCCC is just to run it with the names of a selection of files on the command line like this:

```
cccc my_types.h big.h small.h *.cc
```

For each file, named, CCCC will examine the extension of the filename, and if the extension is recognized as indicating a supported language, the appropriate parser will run on the file. As each file is parsed, recognition of certain constructs will cause records to be written into an internal database. When all files have been processed, a report on the contents of the internal database will be generated in HTML format. By default the main summary HTML report is generated to the file cccc.htm in a subdirectory called .cccc of the current working directory, with detailed reports on each module (i.e. C++ or Java class) identified by the analysis run. In addition to the summary and detailed HTML reports, the run will cause generation of corresponding summary and detailed reports in XML format, and a further file called cccc.db to be created. [7]
The report contains a number of tables identifying the modules in the files submitted and covering:

1. Measures of the procedural volume and complexity of each module and its functions;
2. Measures of the number and type of the relationships each module is a party to either as a client or a supplier;
3. Identification of any parts of the source code submitted which the program failed to parse; and
4. A summary report over the whole body of code processed of the measures identified above.

1. Features of CCCC

The main features of CCCC 3.0 which are working as of today include:

- Internal database recoded using STL(much faster. No hard_ coded limits on run size).
- Persistent file format allows analysis outcomes to be saved across runs, which is reloaded and read by other tools.
- All output files are now generated into a single directory.
-

2. Counting Methods

CCCC calculate each of the measures by implementing simple algorithm. These algorithms are intended to give a useful approximation to the underlying quantities.

a. **Number of Modules (NOM):**
CCCC defines modules in terms of grouping of member functions : C++ classes and namespace, java classes and interfaces and Ada packages are all defined as modules.

b. **Line of Code (LOC):**
It includes industry standard of counting non-blank, non-comment lines of source code. Class and Function declaration are counted, but declarations of global data are ignored.

c. **Comment Lines (COM):**
Any line which contains any part of a comment for the language concerned is treated as a comment by CCCC. The leading comments are treated as part of the function or class definition which follows them.

d. **Cabe's cyclomatic complexity (MVG):**
It is the count of linearly independent paths through a flow of control derived from a subprogram. In case of C++, the count is incremented for each of the following tokens: 'if', 'while', 'for', 'switch', 'break', '&&', '||'.

e. **Weighted methods per class (WMC):**
This is a count of the member functions known to exist in a class. Knowledge of existence of a function is only gained from declarations or definitions directly contained in files processed by CCCC.

When applying these existing approaches we have to face some problems. Sometimes, it changes the meaning of the code and final output. So that we built an interface in which we embed optimization techniques in order to optimize the code. This is an automatic process in which time, space and cost is reduced in comparison to manual code optimization, will give quiet satisfactory result and easy to optimize with complexity measures.

III. PROPOSED TECHNIQUE

This section presents the details of our proposed technique for optimizing the code using automatic tool. Optimization scheme chooses the portion of the code to compile/recompile and then compile the code with set of optimizations. This work is similar in the respect that a separate optimization phase is done concurrently with program execution. However, our work is more aggressive and adaptive in that our optimization phase includes the full spectrum of optimization techniques. Recently, research effort have been exploring the use of search techniques to identify the interaction as well as the value of determining the best order to apply optimizations at different portions of the code [12]. Two optimization techniques "Dead Code Elimination" and "Inlining" are implemented using .NET framework that is connected internally with CCCC tool. We built an interface in which we embed optimization techniques in order to optimize the code. It imports a code from your computer and then some operations are performed according to the techniques implemented and then it calls the CCCC tool automatically to find the complexity of an optimized code.

Our design works on two modes-

i. First mode: - Program is typed already in the program file and we will let the user open the existing file.

ii. Second mode: - User will type the program in the space provided.

A. DEAD CODE ELIMINATION

Dead Code is code that is either never executed or, if it is executed, its result is never used by the program. Unreachable code can be eliminated. Code that follows a return, break, continue, goto and has no label, can be eliminated.[2]

```
int f(int x){                    int f(int x){

 return x+1;  ...}      ⇒        return x+1; }
```

Figure 2. Example Dead code elimination

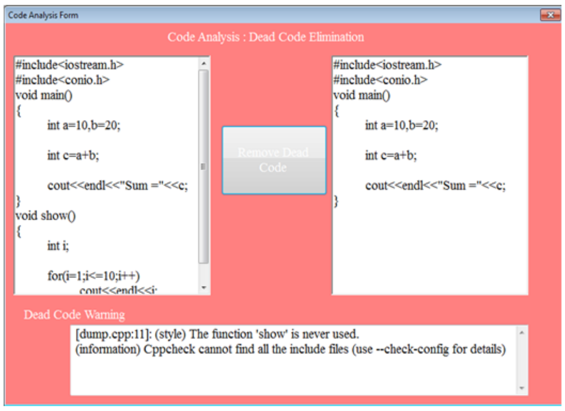
Code that appears in functions that are never called can be eliminated. This process is sometimes describes as “tree-shaking”. In another example, the value assigned to i is never used, and the dead store can be eliminated. The first assignment to app is dead, and the third assignment to global is unreachable; both can be eliminated.

```
int app ;
void  f()
{    int i;
     i=1;
/*dead store*/
     app=1;
/*dead store*/
     app=2;
     return ;
     app=3;
/*unreachable*/
}
```

⇒

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

Using .NET framework that is interconnected with the CCCC tool, we produce an optimized code. Firstly , an existing program is analysed on the platform we provided. In the menu bar three options are there- Code Analysis, Program Check and Help**. In First phase**, Code analysis provides two options local code file and Type code. Dead code elimination can be done on both types of code either it is previously existed file or it is manually typed. When code analysis is done, it gives the dead code warning(as shown in **figure 3**). After removing the dead code, in the bin folder an optimized code file will be automatically generated with the name dump.cpp**. In Second phase**, Program check is used to analyse the performance of optimized code by using the CCCC Tool which was already linked with code analysis link(as shown in **figure 4**) and after that it will give the resultant table of complexity measures(shown in **figure 5**).

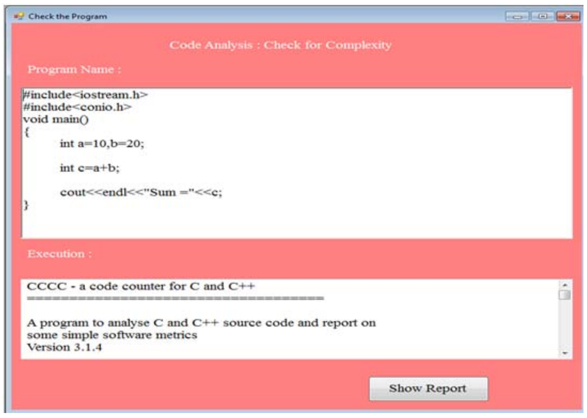


```
#include<iostream.h>
#include<conio.h>
void main()
{
        int a=10,b=20;

        int c=a+b;

        cout<<endl<<"Sum ="<<c;
}
void show()
{
        int i;

        for(i=1;i<=10;i++)
             cout<<endl<<i;
```

```
#include<iostream.h>
#include<conio.h>
void main()
{
        int a=10,b=20;

        int c=a+b;

        cout<<endl<<"Sum ="<<c;
}
```

```
[dump.cpp:11]: (style) The function 'show' is never used.
(information) Cppcheck cannot find all the include files (use --check-config for details)
```

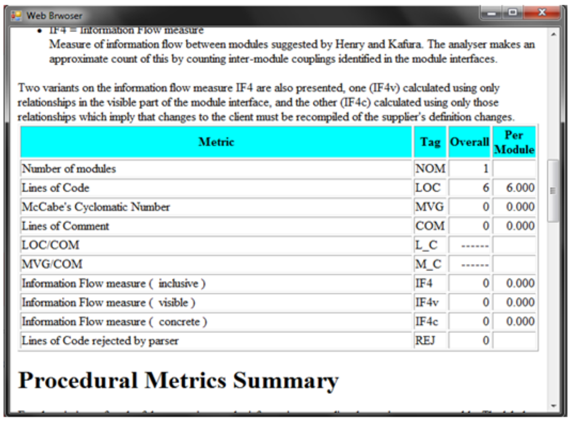
Figure 3. Code Analysis



```
#include<iostream.h>
#include<conio.h>
void main()
{
        int a=10,b=20;

        int c=a+b;

        cout<<endl<<"Sum ="<<c;
}
```

```
CCCC - a code counter for C and C++
=====================================

A program to analyse C and C++ source code and report on
some simple software metrics
Version 3.1.4
```

Figure 4. Complexity Check



- IF4 = Information Flow measure
  Measure of information flow between modules suggested by Henry and Kafura. The analyser makes an approximate count of this by counting inter-module couplings identified in the module interfaces.

Two variants on the information flow measure IF4 are also presented, one (IF4v) calculated using only relationships in the visible part of the module interface, and the other (IF4c) calculated using only those relationships which imply that changes to the client must be recompiled of the supplier's definition changes.

Metric	Tag	Overall	Per Module
Number of modules	NOM	1	
Lines of Code	LOC	6	6.000
McCabe's Cyclomatic Number	MVG	0	0.000
Lines of Comment	COM	0	0.000
LOC/COM	L_C	------	
MVG/COM	M_C	------	
Information Flow measure (inclusive)	IF4	0	0.000
Information Flow measure (visible)	IF4v	0	0.000
Information Flow measure (concrete)	IF4c	0	0.000
Lines of Code rejected by parser	REJ	0	

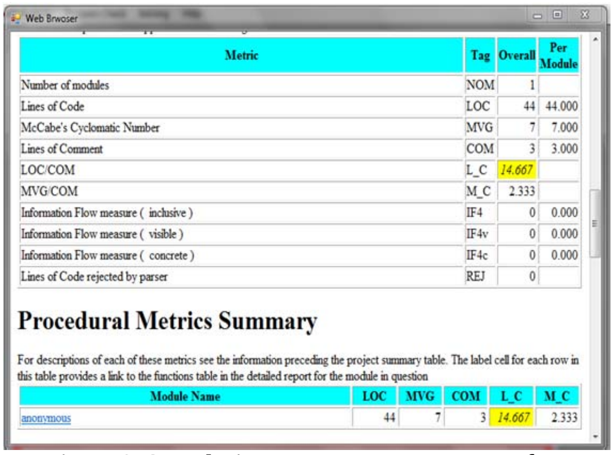
Procedural Metrics Summary

Figure 5. complexity measures after optimization

1. ***Comparision of complexity in between unoptimized code and optimized code using Dead Code Elimination:*** In our project there is a menu option Program check, which is used for Report Generation. The comparison is made using the following example of c code.

Before optimization

```
#include<stdio.h>
#include<conio.h>
/* function prototype */
float average(float , float,float);
void main()
{           float a,b,c,avg;
            clrscr();
            printf("\nEnter the value of a,b,c :");
            scanf("%f %f %f",&a,&b,&c);
            avg=average(a,b,c);
        /*function call */
            printf("\nAverage = %f",avg);
            getch();
}
/* function defintion */
float average(float i,float j,float k)
{           float avg;
            avg=(i+j+k)/3;
            return avg;
}
int sum(int a,int b)
{           int c;
            c=a+b;
            return c;
}
int power(int x,int y)
{           int i,result;
            result=1;
            for(i=1;i<=y;i++)
                        result=result*x;
            return result;
}
int factorial(int num)
{           int fact=1,i;
           for(i=1;i<=num;i++)
                        fact=fact*i;
            return fact;
}
int square(int num)
{           int result;
            result=num*num;
            return result;
}
```



Metric	Tag	Overall	Per Module
Number of modules	NOM	1	
Lines of Code	LOC	44	44.000
McCabe's Cyclomatic Number	MVG	7	7.000
Lines of Comment	COM	3	3.000
LOC/COM	L_C	14.667	
MVG/COM	M_C	2.333	
Information Flow measure (inclusive)	IF4	0	0.000
Information Flow measure (visible)	IF4v	0	0.000
Information Flow measure (concrete)	IF4c	0	0.000
Lines of Code rejected by parser	REJ	0	

Procedural Metrics Summary

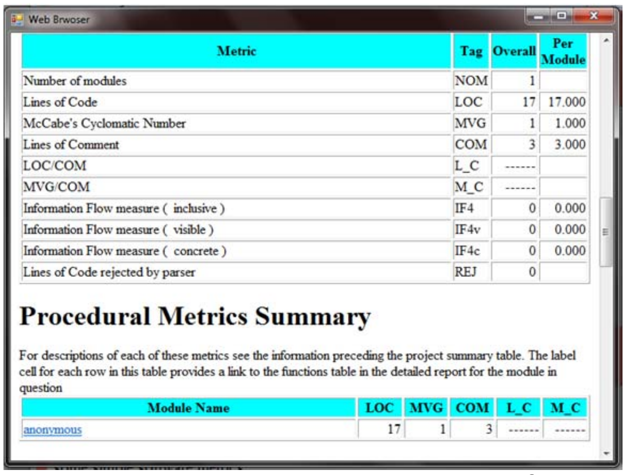
For descriptions of each of these metrics see the information preceding the project summary table. The label cell for each row in this table provides a link to the functions table in the detailed report for the module in question

Module Name	LOC	MVG	COM	L_C	M_C
anonymous	44	7	3	14.667	2.333

Figure 6. Complexity Measurement Report Before Optimization

after optimization

```
 #include<stdio.h>
#include<conio.h>

/* function prototype */

float average(float , float,float);

void main()
{
            float a,b,c,avg;
            clrscr();
         printf("\nEnter the value of a,b,c :");
            scanf("%f %f %f",&a,&b,&c);
            avg=average(a,b,c);
 /*function call */
            printf("\nAverage = %f",avg);
            getch();
}
/* function defintion */
float average(float i,float j,float k)
{
            float avg;
            avg=(i+j+k)/3;
            return avg;
}
```



Metric	Tag	Overall	Per Module
Number of modules	NOM	1	
Lines of Code	LOC	17	17.000
McCabe's Cyclomatic Number	MVG	1	1.000
Lines of Comment	COM	3	3.000
LOC/COM	L_C	------	
MVG/COM	M_C	------	
Information Flow measure (inclusive)	IF4	0	0.000
Information Flow measure (visible)	IF4v	0	0.000
Information Flow measure (concrete)	IF4c	0	0.000
Lines of Code rejected by parser	REJ	0	

Procedural Metrics Summary

For descriptions of each of these metrics see the information preceding the project summary table. The label cell for each row in this table provides a link to the functions table in the detailed report for the module in question

Module Name	LOC	MVG	COM	L_C	M_C
anonymous	17	1	3	------	------

Figure 7. Complexity Measurement Report After Optimization

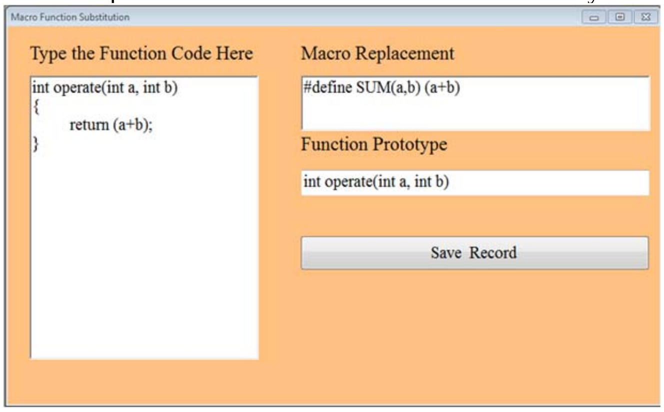
B. INLINING

Inlining refers to compile-time optimization where a small function of code will be injected into the calling function rather than require a separate call. It solves the performance and maintainability issue by letting you declare the function as inline (at least in C++), [3]so that when you call that function - instead of having your app jumping around at runtime - the code in the inline function is injected at compile time every time that given function is called. There is an example of inlining i.e.

```
 void swap(int & m, int & n)
{   int temp = m;
  m = n;
  n = temp;
}
```

After Inlining -
int temp = x;
x = y;
y = temp;

When implementing a sorting algorithm doing lots of swaps, this speeds things up a lot. By using .NET framework, firstly we have to check that the SQL server express is installed then we have to follow some steps one by one, In Inlining process we have some phases for producing final result from unoptimized code to optimized code. First phase is **'Input of unoptimized code'(**shown in **fig 8)**, second phase is **'code analysis and Macro substitution'** in this phase code is analysed and code will process to database(database in **fig 9)** and match the code with the function prototype, and substitute the macros according to the function prototype with the help of stored data and replace it with the function code(shown in **fig 10)**. And last phase of this process is **'resultant code'**. In this if we want to replace it the click 'yes', and then click on 'perform inlining' after that resultant code will display on the screen and give the optimized code (see **fig11)** .If we want to measure its complexity we can use **cccc** tool also. These snapshots will describe its function automatically.

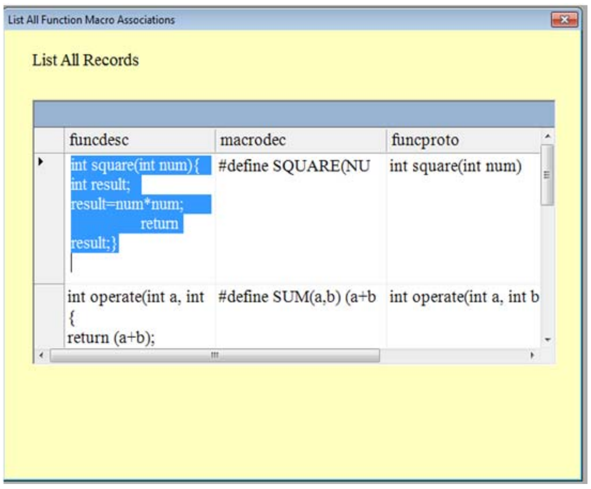


```
int operate(int a, int b)
{
        return (a+b);
}
```

```
#define SUM(a,b) (a+b)
```

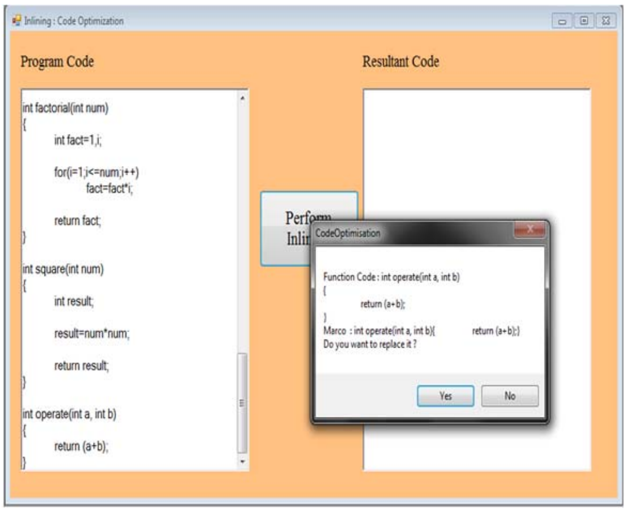
```
int operate(int a, int b)
```

Figure 8. Input of Unoptimized Code



funcdesc	macrodec	funcproto
int square(int num){ int result; result=num*num; return result;}	#define SQUARE(NU	int square(int num)
int operate(int a, int { return (a+b);	#define SUM(a,b) (a+b	int operate(int a, int b

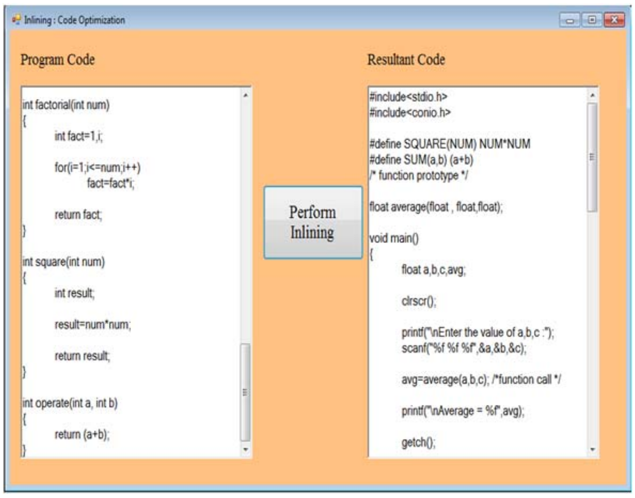
Figure 9. Database of Functions



```
int factorial(int num)
{
        int fact=1,i;

        for(i=1;i<=num;i++)
                fact=fact*i;

        return fact;
}

int square(int num)
{
        int result;

        result=num*num;

        return result;
}

int operate(int a, int b)
{
        return (a+b);
}
```

```
Function Code : int operate(int a, int b)
{
        return (a+b);
}
Marco : int operate(int a, int b){        return (a+b);}
Do you want to replace it ?
```

Figure 10. Substitution & Replacement of Macros



```
int factorial(int num)
{
        int fact=1,i;

        for(i=1;i<=num;i++)
                fact=fact*i;

        return fact;
}

int square(int num)
{
        int result;

        result=num*num;

        return result;
}

int operate(int a, int b)
{
        return (a+b);
}
```

```
#include<stdio.h>
#include<conio.h>

#define SQUARE(NUM) NUM*NUM
#define SUM(a,b) (a+b)
/* function prototype */

float average(float , float,float);

void main()
{
        float a,b,c,avg;

        clrscr();

        printf("\nEnter the value of a,b,c :");
        scanf("%f %f %f",&a,&b,&c);

        avg=average(a,b,c); /*function call */

        printf("\nAverage = %f",avg);

        getch();
```

Figure 11. Final Result

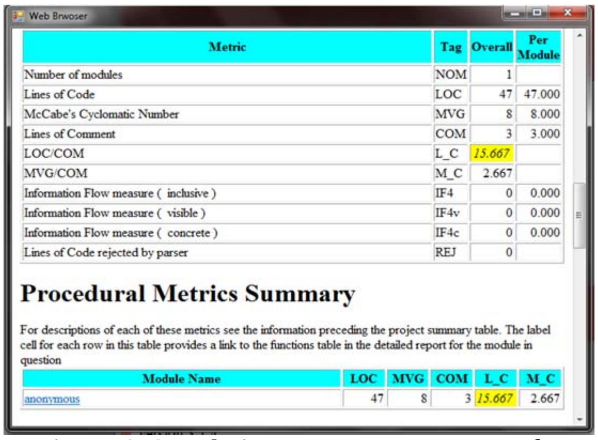
*1. **Comparision of complexity in between unoptimized code and optimized code using Inlining-*** In our project there is a menu option Program check, which is used for Report Generation. The comparison is made using the following example of c code.

Before Optimization

```
#include<stdio.h>
#include<conio.h>
/* function prototype */
float average(float , float,float);
void main()
{           float a,b,c,avg;
   clrscr();
   printf("\nEnter the value of a,b,c :");
  scanf("%f %f %f",&a,&b,&c);
  avg=average(a,b,c);        /*function call */
  printf("\nAverage = %f",avg);
  getch();
}
/* function definition */
float average(float i,float j,float k)
{           float avg;
            avg=(i+j+k)/3;
            return avg;
}
int sum(int a,int b)
{           int c;
            c=a+b;
            return c;
}
int power(int x,int y)
{           int i,result;
            result=1;
            for(i=1;i<=y;i++)
            result=result*x;
            return result;
}
int factorial(int num)
{           int fact=1,i;
            for(i=1;i<=num;i++)
            fact=fact*i;
            return fact;
}
int square(int num)
{           int result;
            result=num*num;
            return result;
}
int operate(int a, int b)
{           return (a+b);
}
```

After optimization

```
#include<stdio.h>
#include<conio.h>

#define SQUARE(NUM) NUM*NUM
#define SUM(a,b) (a+b)
/* function prototype */
float average(float , float, float);
void main()
{           float a, b ,c, avg;
            clrscr();
    printf("\nEnter the value of a,b,c :");
    scanf("%f %f %f",&a,&b,&c);
    avg=average(a,b,c)
 /*function call */
    printf("\nAverage = %f",avg);
    getch();
}
/* function definition */
float average(float I,float j,float k)
{           float avg;
            avg=(i+j+k)/3;
            return avg;
}
int sum(int a,int b)
{           int c;
            c=a+b;
            return c;
}
int power(int x,int y)
{           int I,result;
            result=1;
            for(i=1;i<=y;i++)
            result=result*x;
            return result;
}
int factorial(int num)
{           int fact=1,I;
            for(i=1;i<=num;i++)
            fact=fact*I;
            return fact;
}
```



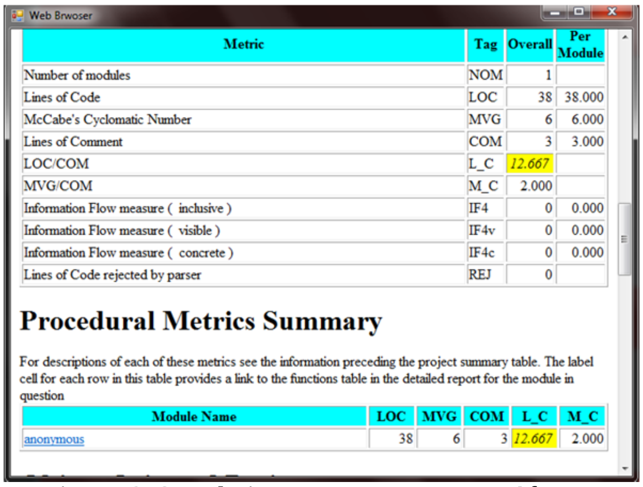
Metric	Tag	Overall	Per Module
Number of modules	NOM	1	
Lines of Code	LOC	47	47.000
McCabe's Cyclomatic Number	MVG	8	8.000
Lines of Comment	COM	3	3.000
LOC/COM	L_C	15.667	
MVG/COM	M_C	2.667	
Information Flow measure (inclusive)	IF4	0	0.000
Information Flow measure (visible)	IF4v	0	0.000
Information Flow measure (concrete)	IF4c	0	0.000
Lines of Code rejected by parser	REJ	0	

Procedural Metrics Summary

For descriptions of each of these metrics see the information preceding the project summary table. The label cell for each row in this table provides a link to the functions table in the detailed report for the module in question

Module Name	LOC	MVG	COM	L_C	M_C
anonymous	47	8	3	15.667	2.667

Figure 12. Complexity Measurement Report Before Optimization



Metric	Tag	Overall	Per Module
Number of modules	NOM	1	
Lines of Code	LOC	38	38.000
McCabe's Cyclomatic Number	MVG	6	6.000
Lines of Comment	COM	3	3.000
LOC/COM	L_C	12.667	
MVG/COM	M_C	2.000	
Information Flow measure (inclusive)	IF4	0	0.000
Information Flow measure (visible)	IF4v	0	0.000
Information Flow measure (concrete)	IF4c	0	0.000
Lines of Code rejected by parser	REJ	0	

Procedural Metrics Summary

For descriptions of each of these metrics see the information preceding the project summary table. The label cell for each row in this table provides a link to the functions table in the detailed report for the module in question

Module Name	LOC	MVG	COM	L_C	M_C
anonymous	38	6	3	12.667	2.000

Figure 13. Complexity Measurement Report After Optimization

Table II. PERFORMANCE MEASUREMENT FOR DEAD CODE ELIMINATION

Lines of Code (LOC)	44	17
McCabe's Cyclomatic Number	7	1
Lines of Comment	3	3
LOC/COM	14.667	----
MVG/COM	2.333	----

Table III. PERFORMANCE MEASUREMENT FOR INLINING

Lines of Code (LOC)	47	38
McCabe's Cyclomatic Number	8	6
Lines of Comment	3	3
LOC/COM	15.667	12.667
MVG/COM	2.667	2.000

IV. RESULTS AND DISCUSSION

In this paper, we have proposed our approaches for code optimization using techniques like Dead Code Elimination and Inlining for programming code written in C / C++ language. In Dead Code Elimination we are using cccc tool which was already linked with function and it will give result of complexities when the dead code found and give the optimized code with complexity measures. While using Inlining technique, we have used SQL server for making a database of code function, and when we click on "optimize the code" after writing an un-optimized code, it will go to database and match the code function, if it matched, the database gives the macro function as per the code function requirement and optimize the code with minimum line of codes. We have verified the code optimization performance using code complexity measurement tools. The results confirm a significant enhancement in quality of optimized code as computed using performance measuring tools.

V. CONCLUSION AND FUTURE WORK

There are different types of code optimization techniques that can be used to make code effective without affecting its final output. The CCCC tool provides the way to find the complexity of optimized and un-optimized codes and comparison in between them. This paper describes our methods for optimization using techniques like Dead Code Elimination and Inlining for programming code written in C / C++ language. Our future work shall incorporate other methods for code optimization in our automated tool for this purpose.

REFERENCES

[1] Michael E. Lee, "Optimization of Computer Programs in C", Ontek Corporation, USA "Code Optimization" article. Available: http://leto.net/docs/C-optimization.php Forman, G. 2003.

[2] Maggie Johnson, "Code Optimization", Handout 20, August 04, 2008.

[3] Mohammed Fadle Abdulla, "Manual and Fast C Code Optimization", Anale. Seria Informatica. Vol. VIII fasc. I-2010.

[4] Mr. Chirag H. Bhatt, Dr. Harshad B. Bhadka, "Peephole Optimization Technique for analysis and review of Compile Design and Construction", IOSR Journal of Computer Engineering (IOSR-JCE), Volume 9, Issue (4 Mar. - Apr. 2013).

[5] "Optimization Techniques in C", Fall, 2013. Available: http://cs.brown.edu/courses/cs033/docs/guides/c_optimization_notes.pdf.

[6] C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, "Source–Level Execution Time Estimation of C Programs", Proceedings of the ninth international symposium on Hardware/software code design.

[7] "CCCC User Guide" available at http://www.stderr.org/doc/cccc/CCCC%20User%20Guide.html

[8] Tips for "Optimizing C/C++ Code". Available: http://people.cs.clemson.edu/~dhouse/courses/405/papers/optimize.pdf

[9] "Writing Efficient C and C Code Optimization". Article: http://www.codeproject.com/Articles/6154/Writing-Efficient-C-and-C-Code-Optimization

[10] "Optimizing C++/ Code Optimization/ Faster operations"Available: http://en.wikibooks.org/wiki/Optimizing_C%2B%2B/Code_optimization/Faster_operations.

[11] "Continuous Compilation: A New Approach to Aggressive and Adaptive Code Transformation". Available:http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=1213375&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D1213375\

[12] S.K.Srivastava, Deepali Srivastava, "C in Depth" 3rd edition.

[13] Programming language translation http://web.cs.wpi.edu/~cs544/PLT10.2.3.html

[14] www.sm.luth.se_csee_courses_smd_163_lecture11.pdf Viktor Leijon & Peter Jonsson with slides by Johan Nordlander Contains material generously provided by Mark P. Jones