CENG 491 – COMPUTER ENGINEERING DESIGN
FINAL DESIGN REPORT
[CSTAR PROJECT]
“KROMPILER”
BY

Osman Paşalak
Fatih Akdoğan
Alperen Yeşil
M. Ozan Koloğlu

Fall 2007
# Table of Contents

1. **INTRODUCTION** ........................................................................................................... 4  
   1.1 Project Title ................................................................................................................. 4  
   1.2 Project Definition .......................................................................................................... 4  
   1.3 Project Scope ............................................................................................................... 4  
   1.4 Project Estimates ......................................................................................................... 6  
      1.4.1 ESTIMATION CALCULATIONS ................................................................................. 6  
      1.4.1.1 FP Based Estimation Calculations .............................................................................. 6  
      1.4.1.2 LOC Calculation Using FP ......................................................................................... 7  
      1.4.2 THE COCOMO II MODEL .......................................................................................... 8  
2 – **REQUIREMENTS** ..................................................................................................... 10  
   2.1 Functional Requirements ............................................................................................ 10  
      2.1.1 Software Requirements ......................................................................................... 10  
      2.1.2 Hardware Requirements ....................................................................................... 11  
   2.2 Non-functional Requirements ...................................................................................... 11  
      2.2.1 Performance ......................................................................................................... 11  
      2.2.2 Usability .............................................................................................................. 11  
      2.2.3 Reliability ........................................................................................................... 12  
3- **ARCHITECTURAL DESIGNS** .................................................................................. 12  
   3.1 Optimization Manager ............................................................................................... 12  
      3.1.1 Language and Grammar Design ............................................................................. 12  
      3.1.2 Grammar .......................................................................................................... 15  
   3.2 Test Case Generator Module ....................................................................................... 22  
   3.3 Optimizations Module ............................................................................................... 24  
4- **Class Diagrams and Explanations** ......................................................................... 24  
   4.1 Framework Class Diagrams ....................................................................................... 24  
      4.1.1 IR Base Classes ................................................................................................... 24  
      4.1.2 Framework Core Utilities Class Diagrams ........................................................... 26  
   4.2 Test Case Generator Module ....................................................................................... 30  
      4.2.1 Test Case Generator Detailed Design .................................................................... 31  
   4.3 Optimizations ........................................................................................................... 33  
      4.3.1 Constant Folding .................................................................................................. 34  
      4.3.2 Global Copy Propagation ....................................................................................... 34  
      4.3.3 Local Copy Propagation ......................................................................................... 35  
      4.3.4 Local Common Subexpression Elimination .......................................................... 35  
      4.3.5 Global Common Subexpression Elimination ........................................................ 36  
      4.3.6 If Simplifications .................................................................................................. 37  
      4.3.7 Unreachable Code Elimination .............................................................................. 37  
      4.3.8 Tail Merging ....................................................................................................... 38  
      4.3.9 Jump Optimizations ............................................................................................ 39  
      4.3.10 Global/Local Forward Substitution .................................................................... 40  
      4.3.11 Strength Reduction ............................................................................................. 41  
      4.3.12 Basic Block Ordering ......................................................................................... 42
4.3.13 Dead Object Elimination

4.4 Loop Optimizations

4.5 Algorithms of Optimizations

4.5 Optimization Manager Module

5- System Model

5.1 DFD and Explanation

5.1.1 DFD level 0

5.1.2 DFD level 1

5.1.3 Explanation of DFD

5.2 Use Case Diagram

5.3 Activity Diagrams

5.3.1 Optimization Manager

5.3.2 Test Case Generator

6 – Testing Strategy

6.1 Unit Testing

6.2 - Integration Testing

7 – Project Schedule

APPENDIX A
1. INTRODUCTION

1.1 Project Title

Our project title is ‘Krompiler’

1.2 Project Definition

As you know many new hardware architectures are designed each year, they all need compilers. That is why compiler technology is very important. It is for sure that compiler will produce the right output but moreover it is very important for a compiler to use less amount of time. Today many developers are interested in optimizing compilers. Handling these optimizations is very significant. Our aim in this project is to do optimizations for CSTAR compiler framework, to write a code generator for the optimizations and to make an optimization manager.

1.3 Project Scope

Krompiler project consists of mainly 3 parts which are optimizations, test case generator, and optimization manager. Here are some general features that will be in these parts:

OPTIMIZATIONS

Optimization is the process of modifying a system to make some aspect of it work more efficiently or use fewer resources which can be hardware specific or hardware independent. For example using reduced amount of memory stack, cpu cycles are kinds of optimizations.

Optimizations are the most important part of the compiler.

Here are the optimizations we will write in our Krompiler project

1) Constant folding
2) Dead code elimination
3) Basic block ordering
4) Local\Global forward substitution
5) Strength reduction
6) Unreachable code elimination
7) Dead object elimination
8) Local/Global Copy Propagation
9) Local/Global Common Sub expression Elimination
10) Jump Optimizations
11) If Simplifications
12) Tail merging
13) Loop unrolling
14) Loop reversal
15) Loop splitting / peeling
16) Loop fission (distribution)
17) Loop fusion (combine)
18) Loop switching (interchange)
19) Loop unswitching
20) Loop skewing

TEST CASE GENERATOR

Testing is one of the most important parts of a software project. For a compiler it is not acceptable to generate a wrong code. So we should test the compiler with a set of tests since it may not be convenient to use hand-written tests. So it is better to use a test case generator. In our project we will implement a test case generator which will be helpful especially for the optimizations. The developer will be able to generate a test code as he/she wants. He can generate a test code for just one optimization or more. He/she also will be able to choose the ratio of the operations (e.g. %20 multiplication, %10 if conditions). By the help of the test case generator developer will have good test-codes and this will better help to see the bugs in the system.

OPTIMIZATION MANAGER

Many compiler optimizations are actually transformations that compiler writers hope that it would optimize the input. There are few optimizations that always result in better code. Therefore a sequence of optimizations is good for some input sources and bad in some other. So it is hard for a compiler to choose what to do. In this project we will write
an optimization manager which will help compiler users to apply optimizations as many times and in any order. Optimization manager will read an external file which specifies the execution order of the optimizations and each optimization’s options (e.g. number of times that the optimization will be applied).

### 1.4 Project Estimates

We are going to apply the following estimation techniques to our project:

1. Function Point (FP) Based Estimation
2. Lines of Code LOC – COCOMO II Model

#### 1.4.1 ESTIMATION CALCULATIONS

**1.4.1.1 FP Based Estimation Calculations**

In order to calculate Function Point Value, we start with the estimation of the domain values:

<table>
<thead>
<tr>
<th>Information domain value</th>
<th>Optimistic</th>
<th>Likely</th>
<th>Pessimistic</th>
<th>Estimated Count</th>
<th>Weight</th>
<th>FP count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of external inputs</td>
<td>5</td>
<td>8</td>
<td>10</td>
<td>8</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>Number of external outputs</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>Number of external inquiries</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>24</td>
</tr>
<tr>
<td>Number of internal logical files</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>21</td>
</tr>
<tr>
<td>Number of external logical files</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>21</td>
</tr>
<tr>
<td><strong>Count Total</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>123</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feature</td>
<td>Weight</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>--------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System Backup and Recovery</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specialized Data Communications</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distributed Processing Functions</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Critical Performance</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operational Environment</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>On-line Data Entry System</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input Transaction Over Multiple Screens</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>On-line Internal Logical File Update</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complexity of Input, Output and Files</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex Internal Processing</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reusable Code Design</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conversion and Installation in Design</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple Installation in Organizations</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Changeable Facilitates and User Friendly Design</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \sum(F_i) = 43 \]

**FP_{Estimation}** = \text{Count Total} \times [0.65 + 0.01 \times \sum(F_i)]

\[ = 123 \times [0.65 + 0.01 \times 43] \]

\[ = 123 \times 1.08 \]

\[ = 132.8 \]

### 1.4.1.2 LOC Calculation Using FP

We will use C++ to implement our project since all the framework is implemented in C++. We will assume that average value of LOC will be used in our project. This value is as follows:

- 60 LOC/FP for C++

Finally by using FP and LOC/FP the estimated LOC value is:

\[ 60 \times 132.8 = 8710 \text{ LOC} = 7.97 \text{ KLOC} \]
Calculating the Effort

LOC-Oriented Effort Estimation

i) Waltson-Felix Model
\[ E = 5.2 \times (\text{KLOC})^{0.91} \]
\[ = 5.2 \times (7.97)^{0.91} \]
\[ = 34 \text{ person-month} \]

ii) Bailey-Basili Model
\[ E = 5.5 + 0.73 \times (\text{KLOC})^{1.16} \]
\[ = 5.5 + 0.73 \times (7.97)^{1.16} \]
\[ = 14 \text{ person-month} \]

iii) Boehm-Simple Model
\[ E = 3.2 \times (\text{KLOC})^{1.05} \]
\[ = 3.2 \times (7.97)^{1.05} \]
\[ = 28 \text{ person-month} \]

FP-Based Effort Estimation

a) Kemerer Model
\[ E = 60.62 \times 7.728 \times 10^{-8} \times \text{FP}^3 \]
\[ = 60.62 \times 7.728 \times 10^{-8} \times 132.8^3 \]
\[ = 11 \text{ person-month} \]

1.4.2 THE COCOMO II MODEL

<table>
<thead>
<tr>
<th>COST</th>
<th>RELY</th>
<th>DATA</th>
<th>CPLX</th>
<th>RUSE</th>
<th>DOCU</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.75</td>
<td>0.88</td>
<td>0.75</td>
<td>0.91</td>
<td>0.89</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>0.88</td>
<td>1.00</td>
<td>0.88</td>
<td>0.91</td>
<td>0.95</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>RATING</td>
<td>1.15</td>
<td>1.10</td>
<td>1.15</td>
<td>1.14</td>
<td>1.06</td>
<td>1.06</td>
</tr>
<tr>
<td></td>
<td>1.39</td>
<td>1.19</td>
<td>1.66</td>
<td>1.49</td>
<td>1.13</td>
<td>1.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.00</td>
<td>1.06</td>
<td>1.21</td>
<td>1.57</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>STOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PVOL</td>
<td>0.87</td>
<td>1.00</td>
<td>1.15</td>
<td>1.30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACAP</td>
<td>1.50</td>
<td>1.22</td>
<td>1.00</td>
<td>0.83</td>
<td>0.67</td>
<td></td>
</tr>
<tr>
<td>PCAP</td>
<td>1.37</td>
<td>1.16</td>
<td>1.00</td>
<td>0.87</td>
<td>0.74</td>
<td></td>
</tr>
<tr>
<td>PCON</td>
<td>1.24</td>
<td>1.10</td>
<td>1.00</td>
<td>0.92</td>
<td>0.84</td>
<td></td>
</tr>
<tr>
<td>AEXP</td>
<td>1.22</td>
<td>1.10</td>
<td>1.00</td>
<td>0.89</td>
<td>0.81</td>
<td></td>
</tr>
<tr>
<td>PEXP</td>
<td>1.25</td>
<td>1.12</td>
<td>1.00</td>
<td>0.88</td>
<td>0.81</td>
<td></td>
</tr>
<tr>
<td>LTEX</td>
<td>1.22</td>
<td>1.10</td>
<td>1.00</td>
<td>0.91</td>
<td>0.84</td>
<td></td>
</tr>
<tr>
<td>TOOL</td>
<td>1.24</td>
<td>1.12</td>
<td>1.00</td>
<td>0.86</td>
<td>0.72</td>
<td></td>
</tr>
<tr>
<td>SITE</td>
<td>1.25</td>
<td>1.10</td>
<td>1.00</td>
<td>0.92</td>
<td>0.84</td>
<td>0.78</td>
</tr>
<tr>
<td>SCED</td>
<td>1.29</td>
<td>1.10</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
</tr>
</tbody>
</table>

**RELY:** Required Software Reliability  
**DATA:** Data base size  
**CPLX:** Process Complexity  
**RUSE:** Required Reuse  
**DOCU:** Documentation Match to Lifecycle  
**TIME:** Time Constraint for CPU  
**STOR:** Main Memory Constraint  
**PCAP:** Programmer Capability  
**PCON:** Personnel Continuity  
**AEXP:** Application Experience  
**PEXP:** Platform Experience  
**LTEX:** Language and Tool Experience  
**TOOL:** Use of Tools  
**SITE:** Multisite Development  
**SCED:** Required Development Schedule

\[
EAF = 0.93 \times 1.15 \times 1.14 \times 1.00 \times 1.11 \times 1.06 \times 1.00 \times 1.00 \times 0.84 \times 1.10 \times 1.00 \times \\
0.91 \times 1.12 \times 1.00 \times 1.00 = 1.35
\]

If we consider the mode definitions of the projects our models can not be the organic mode since we won’t be working in a highly familiar, in-house environment but instead we will have more strict schedules and deadlines to be obeyed. However we can’t be considered to be working in embedded mode too. Finally we can conclude that we are working in the semidetached mode.

We will be using the following formulas to calculate the effort, time and number of people from the COCOMO II Model:

\[
\text{EFFORT} \rightarrow E = a \times (\text{KLOC})^b \times \text{EAF}
\]
TIME → D = c * E^d

NUMBER OF PEOPLE → N = E / D

The values for the variables a, b, c and d will be the following according to the Semidetached mode:

\[ a = 3.00 \quad b = 1.12 \quad c = 2.5 \quad d = 0.35 \]

\[ E = 3.0 \times (7.97)^{1.12} \times 1.35 = 39 \text{ person-month} \]
\[ D = 2.5 \times (39)^{0.35} = 9 \text{ month} \]
\[ N = 39/9 = 4 \text{ people} \]

The above calculations are given as approximate calculations rather than exact floating point descriptions.

2 – REQUIREMENTS

2.1 Functional Requirements

2.1.1 Software Requirements

1) LINUX: QUICKC compiler framework works under Linux. However under Windows Cygwin and a Linux virtual machine environment, the framework should also work.

2) GCC: QUICKC compiler framework is build with GCC version 4. So, GCC version should be over 4.0. Developers also need GCC 4 to generate executable from framework assembly output. In our project we will write some of the optimizations for the QUICKC compiler framework. Because the entire framework is written in C++, we have to work in GCC version 4, also.

3) ANTLR: In optimization manager module, we will interpret the code that will be entered by the user. The language of the input code will be designed for us, so we have to write lexer and parser for our language. We are planning to create the
lexer and parser classes by some automatic compiler generator like ANTLR or another program.

2.1.2 Hardware Requirements

1) X86 HOST MACHINE: The compiler framework is designed for X86 machines. To execute the framework and to see the test results an X86 Host Machine is needed.

2.2 Non-functional Requirements

2.2.1 Performance

In a compiler framework, compiler optimizations consists extensive number of cases to handle as a matter of fact design of these optimizations effects compilers performance very much. Our aim is to increase performance as much as possible. For us, CSTAR provide two benchmark programs (Dhrystone, Whetstone). We will use these programs for enough time to inspect an optimal performance.

2.2.2 Usability

In a compiler framework, testing a produced compiler is an important issue and we will inspect this in test case generator part. Test codes for compilers should be as rich as possible and contain all functionality of input language. Krompilier will contain a complex test case generator. Also this generator will get options in human-readable form and produce highly complicated test code. And the usability of optimization manager is
very important. We design a user friendly language for input codes. And for all modules, we are planning to create informing documents. Especially for the optimizations, telling users what are them, where and when they are optimally used is so important.

2.2.3 Reliability

Firstly, when we finish writing the optimizations we will use benchmarks to test the performance of the optimizations. Secondly we will create a test case generator for the framework as a part of the project. This module will also give us the possibility to efficiently testing the optimizations.

3- ARCHITECTURAL DESIGNS

3.1 Optimization Manager

Before starting the details of the design, we want to talk about how the user will use the optimization manager module. They will enter the command “-optmanager filename.txt” to run the optimization manager. Now, we will first give information about the design of language and its grammar. Later we will try to tell the class diagram design.

3.1.1 Language and Grammar Design

We designed a C-similar language. In our design the program is divided into three main parts. First are the declarations. The second is the setting options and the last one is executions. We thought that in the design we should give user the permission of using the three main parts in any order and in anywhere. The reason of this was that otherway we will make the user going continuously between top and down of the code which is not a well design. And also the grammar has no scopes.
Declarations are done in the language as “ATOx var1;”. Here ATOx must be an existing anatrop. We thought that there is no need to allow redefinition. Here we will read all available optimizations from a file. If ATOx optimization does not exist in the framework we will give an error.

Setting options are done by the statement “var1.opt_name = true”. Here opt_name must be an existing option of that anatrop. Otherwise we will give an error. In addition we will give type checking errors. If an option is int option it can not be set to boolean value.
And finally we divided executions into two. One is expressions. We needed the expressions constants, and- or expressions, comparison expressions function-calls and paranthesized expressions, because we needed their values and types which we will use in syntax-directed evaluation and type-checking. So our expressions return \{type,value\} structs. Constants are integer, boolean and string which we need them in setting options in other expressions. And or expressions are also needed, because they will be used at if statements. Comparison expressions are similarly.

Our function calls are execute(var_name,scope), enterinteractive(), leaveinteractive() and dumpIR(). Execute function is called with parameters optimization variable and scope. Since the user can set the options of variables anywhere, they will not give options as inputs. Enterinteractive and leaveinteractive functions are used to enter and leave the interactive mode.

In interactive mode we will start reading from stdin. And the dumpIR function will be used to dump IR at the interactive mode. That function is needed because the users should see the IR, if they will give scopes like statements or loops.

And our statements are if and while statements. We designed if statement as: if(expr) program else program. We evaluate the expression and choose what we should do. However here we use a variable named parseandexecute for example we have to parse else part but shouldn’t execute the statements in it. And for the while statement we designed as while(20) program. We thought it is enough for using int_constants here. We use the tokenindex here. For n times after we execute the loop part we decrement the token index and parse the code again.

And finally, for the interactive mode, our grammar works slightly differently. We designed that in the interactive mode the user also can do whatever thing he wants,. So we designed that in the interactive mode the user can write as much as programs until he leaves the mode. But there is a problem in reading from stdout. When we will finish reading input and start parsing and evaluating. Whenever user enters a statement, we will parse it, give errors and evaluate that statement. So until leaveInteractive() function is called, we will read the tokens from the lexer statement by statement. To understand when a statement ends, the lexer will read the input until it sees a left brace for if and while statements or semicolon for function calls. When it sees one of the characters it will extract the tokens and the parser will parse the tokens.
And in the language the comments are like the line comments in C language. “//comment” is used for comments. Also whitespaces like tabspaces, newlines are not important in the language as in the C language.

### 3.1.2 Grammar

Below grammar will be used in ANTLR for input parsing of optimization manager.

**Lexer**:

- **ID**: letter (digit | letter)*
- **IF**: “if”
- **ELSE**: “else”
- **WHILE**: “while”
- **OPTIONS**: “options”
- **SMCLN**: “;”
- **ASSIGN**: “=”
- **COMMA**: “,”
- **POINT**: “:”
- **LPAREN**: “(“
- **RPAREN**: “)”
- **AND**: “&&”
- **OR**: “||”
- **COMP**: “==” | “<” | “>” | “!” | “<=” | “>=”
- **LBRC**: “{“
- **RBRC**: “}”
- **INTconst**: (digit)+
- **STRINGconst**: “” (~"“)+ “”
- **BOOLconst**: “true” | “false”
- **WS**: (’ ‘ | ‘\n’ | ‘\t’) { (Token.SKIP); } ;
- **COMMENT**: ”/” (~”n”)* ”n” { (Token.SKIP); }
Parser:

\[
\text{program} \rightarrow (\text{declarations} \mid \text{set\_options} \mid \text{executions})^* \\
\{ \text{if (isInteractive==true) (program); } \} ;
\]

\[
\text{declarations} \rightarrow \text{optname:ID optvar:ID SMCLN} \\
\{ \\
\quad \text{if(parseandexecute==true)} \\
\{ \\
\quad \text{if(isDefined(optname) == false)} \\
\quad \text{giveerror();} \quad \text{// if there is not such an optimization} \\
\quad \text{anatropManager.Create(optname,scope);} \\
\quad \text{symboltable.addSymbol(optname,optvar);} \quad \text{// adds symbol to symtable} \\
\quad \text{// also reads all the options of optimization} \\
\quad \text{// and sets them their default values at symtable} \\
\}
\}
\]

\[
\text{set\_options} \rightarrow \text{optvar: ID POINT option\_name:ID ASSIGN c:constant SMCLN} \\
\{ \\
\quad \text{if(parseandexecute==true)} \\
\{ \\
\quad \text{if(isDefined(optvar\_option\_name) == false)} \\
\quad \text{giveerror();} \quad \text{// if there is not such a variable or the optimization variable} \\
\quad \text{// does not have such an option} \\
\quad \text{if (symboltable.setOption (optvar,option\_name,c) == false)} \quad \text{//sets to symtable} \\
\quad \text{giveError(“type error”);} \quad \text{// the constant type is not the type of option} \\
\quad \text{optvar.GetOptions().obSet(option\_name, c);} \quad \text{// sets an option in the options} \\
\quad \text{// object of optimization} \\
\}
\}
\]
executions \rightarrow (expression \text{(SMCLN)}? | statement);

expression returns \{ \{ string value; string type; \} \} \rightarrow
\{ value,type \} = and_expr
(OR \{ tempval,temp_type \} = and_expr
{
    if(isTrue(value,type) || isTrue(tempval,temp_type))
        value="true"
    else
        value="false";
    type="bool";
}
)*;

and_expr returns \{ \{ string value="false"; string type="bool"; \} \} \rightarrow
\{ value,type \} = comp_expression
(AND \{ tempval,temp_type \} =comp_expression
{
    if(isTrue(value,type) \&\& isTrue(tempval,temp_type))
        value="true"
    else
        value="false";
    type="bool";
}
)*;
comp_expression returns [ {string value; string type; } ] →
{ value,type } = unit_expressions
( oper:COMP {val2,type2} =unit_expression
{

    if(type="int" || "type2!="int")
        give_error();
    else if(compare(value, val2, oper) == true)
    {
        value = "true";
    }
    else if(compare(value, val2, oper) ==false)
    {
        value = "false";
    }
    type= "bool";
}
)? ;

unit_expressions returns [ {string value; string type; } ] →
(   | { value,type } =function_call
    | { value,type } =constant
    | { value,type } =par_expression ) ;

par_expression returns [ {string value; string type; } ] →
LPAREN expression RPAREN
{
    {value, type} = expression;
}
function_call returns [ {string value=""; string type=""; } ] →
{ value,type } = execute_func | { value,type } = other_funcs;

execute_func returns [ {string value=""; string type=""; } ] →
funcname:ID LPAREN varname:ID COMMA scope:ID RPAREN
if(funcname=="execute" && parseandexecute==true)
  s{
    value=anatropManager.execute(varname,scope,varname.getOptions());
    type="int";
  }

other_funcs returns [ {string value=""; string type=""; } ] →
funcname:ID LPAREN RPAREN
{
  if(funcname=="enterInteractive" && parseandexecute==true)
    isInteractive=true;
    ///////// interactive mode //////////
    while (isInteractive ==true)
      {
        Myparser.getStdoutTokens(); // get tokens until you see a “;”
        // or “}” which is the end of if and while
        // statements from the lexer
        Myparser.program(); // parse and evaluate the input for each statement
        // so at each statement the code will be parsed
        // until user leaves the interactive mode
      }
  type="void";
}
if (funcname=="leaveInteractive" && parseandexecute==true) {
    isInteractive=false;
    type="void";
}
if (funcname=="dumpIR" && parseandexecute==true && isInteractive=true;) {
    debug(context,ostr); // dump the intermediate code
    type="void";
}

constant returns [ {string value; string type; } ] →
value=INT_constant {type="int";} |
value=STRING_const {type="string";} |
value=BOOL_const {type="bool";} ;

statement → if-statement | while-statement

if-statement → IF LPAREN { val,type } = expression RPAREN { }
parseandexecute="false”;
if(isTrue(value,type))
    parseandexecute="true”;
}
LBRC program RBRC { } if(isTrue(value,type))
while-statement  {int k = 0; } \rightarrow \text{WHILE LPAREN n: INTconst RPAREN}
{
    for(int k=0;k<n;k++)
    {
        int start_token = currenttokenindex;
        LBRC program RBRC;
        currenttokenindex = start_token;
    }

    parseandexecute=false;
    LBRC program RBRC;
    parseandexecute=true;
} ;
3.2 Test Case Generator Module

Main purpose of test case generator module is generating random C codes. Our design goal is mainly positioned at probabilistic issues. Briefly it can be explained like these,

All expressions and constants in statements can have a different probability to occur.

In order to use test case generator CStar framework should be called with command line argument “-test SpecificationFileName” and target file name should be given with command argument “-testfile OutputFileName”. Input of the test case generator must be an empty C file containing an empty main.

Here is a sample configuration file for test case generator module. This file will be included in Framework as a template user can change its name and options in it.

```c
# #
#test Case Generator configuration file #
#If 'X's didn’t changed a default value will be used #
# #

#random seed
# Double constant

X

#threshold value
# 0 < int constant < 100

X

#lines of code range
#integer constant - integer constant

X - X

#number of functions range
#integer constant - integer constant
```
# Probability of total if-statement and switch statements
# 0 < integer < 100
X

# Probability of loops
# 0 < integer < 100
X

# Constant types probabilities in given context
# contexts If-Stmt For-stmt Switch-stmt
# BasicBlock Expr (Add,Mult,Neg,Sub...)
# int
# float
# double
# char
# all are 0 < integers < 100
X X X X X X
X X X X X X
X X X X X X
X X X X X X
X X X X X X

# Expression type probabilities in functions (all are integers with 0 < #X < 100)
# Addition - Subtraction
X
# Multiplication - Division
X
# Comparison
X
# Assignment
X
# Unary
X

Test Case Generator will get number of lines in output target code, random seed, number of different functions range, and some probabilities of expressions and statements. This constrains will have default values to for user profit. Threshold value is a
helper value for performance of background probability issues. It helps module to ignore some probability values in estimations.

Test case generator also has a query method in “ProbManager “class for developer to learn probability of expression in a given context.

### 3.3 Optimizations Module

Optimizations are known algorithms by every one so we thought that they shouldn’t be in architectural design part. We show their class diagrams and give brief explanations about optimizations in Class diagrams and explanations part.

### 4- Class Diagrams and Explanations

#### 4.1 Framework Class Diagrams

Our project contains mainly 2 parts .One part is the framework classes which we will use , other part is the classes that we will write. We will show class diagrams of both part .But important issue that we aren’t going to show all methods and attributes of framework classes .We will only show methods we will use in our project and also we wont give explanations about methods and attributes as it is huge and beyond our scope in report. On the other hand we will explain all our methods and attributes of own classes and give detailed information about them. We decided to follow a symbolic notation for class diagrams .Main reason is it becomes chaotic if we use traditional way.

#### 4.1.1 IR Base Classes

These classes handle Intermediate Representation of source code and they are already in framework. We will use these classes as needed. These classes are named as IR PACKAGE will be used in our class diagrams with this name for reasons we already told.
4.1.2 Framework Core Utilities Class Diagrams

These classes will be used by our classes several times. Again they will be used as symbolic names for easy to read.

IRBBuilder class is responsible of intermediate representation. Extract class is responsible gathering information about intermediate representation. Predicate class has methods those query information about intermediate representation. All of them uses IR classes explained above. These are the main classes of CStar Framework. In our project we will use their methods and attributes to implement optimizations, optimization manager and test case generator. In our class diagrams we will show them symbolically for easiness.
**PREDICATE**

```c++
+<<const>> pAnyPathFromEntry (IRBB *bb_): bool
+<<const>> pHasPred (const IRBB *bb_): bool
+<<const>> pHasSucc (const IRBB *bb_): bool
+<<const>> pIsAPred (const IRBB *bb_, const IRBB *pred_): bool
+<<const>> pIsAssign (const IRStmt *stmt_): bool
+<<const>> pIsASucc (const IRBB *bb_, const IRBB *succ_): bool
+<<const>> pIsConst(const IRExpr *expr_): bool
+<<const>> pIsConstAssign (const IRStmt *stmt_): bool
+<<const>> pIsDead (const IRStmt *stmt_): bool
+<<const>> pIsDead (const IRBB *bb_): bool
+<<const>> pIsEmpty (const IRDd *bb_): bool
+<<const>> pIsEntryBB (const IRBB *bb_): bool
+<<const>> pIsDead (const IRExpr *expr_): bool
+<<const>> pIsEntryOrExitBB (const IRBB *bb_): bool
+<<const>> pIsGlobal (const IRObj *obj_): bool
+<<const>> pIsIf (const IRStmt *stmt_): bool
+<<const>> pIsAdd (const IRExpr *expr_): bool
+<<const>> pIsEmpty (const IRBB *bb_): bool
+<<const>> pIsInLoop (const IRLoop *loop
  const IRStmt *stmt_): bool
+<<const>> pIsAddrOf(const IRExpr *expr_): bool
+<<const>> pIsIfOrSwitch (const IRStmt *stmt_): bool
+<<const>> pCanKill (const IRExpr *expr_,
  const IRStmt *stmt_): bool
+<<const>> pIsAddrOfObj (const IRExpr *expr_): bool
+<<const>> pIsDeletedOrNull (const IRBB *bb_): bool
+<<const>> pIsDeletedOrNull (const IRStmt *stmt_): bool
+<<const>> pIsEndOfBB (const InsertContext *ic_): bool
+<<const>> pIsEpilog (const IRStmt *stmt_): bool
+<<const>> pIsFunc (const IRTyp *type_): bool
+<<const>> pIsGlobalOrAddrTaken (const IRObj *obj_): bool
+<<const>> pIsIntConst (const IRExpr *expr_): bool
+<<const>> pIsJump (const IRStmt *stmt_): bool
+<<const>> pIsLocal (const IRObj *obj_): bool
+<<const>> pIsMember (const IRBB *bb_, const IRStmt *stmt_): bool
+<<const>> pIsMul (const IRExpr *expr_): bool
+<<const>> pIsObj (const IRExpr *expr_): bool
+<<const>> pIsParentExpr (const IRExpr *expr_): bool
+<<const>> pIsParentStmt (const IRExpr *expr_): bool
+<<const>> pIsPrologOrEpilog (const IRStmt *stmt_): bool
+<<const>> pIsReal (const IRTyp *type_): bool
+<<const>> pIsSub (const IRExpr *expr_): bool
+<<const>> pIsSwitch (const IRStmt *stmt_): bool
+<<const>> pIsLandOr (const IRExpr *expr_): bool
+<<const>> pIsFirstStmt (const IRStmt *stmt_): bool
+<<const>> pIsDiv (const IRExpr *expr_): bool
+Predicate()
+Predicate (): virtual
```
4.2 Test Case Generator Module

MyBuilder Class

Attributes:
Vector<IRFunc *> &globalFuncList
A list of functions which will be inserted to output code
vector<InsertContext *> &globalIC
A structure for tracing insertContexts.

Methods:
BuildFunc() : this function builds an IRFunction with given probabilities.
BuildForLoop() : constructs an IRLoop with given probabilities.
BuildIfstat() : builds an if statement with given probabilities.
BuildReturn() : builds a return statement with given probabilities.
BuildBB() : builds a basic block with given probabilities also uses its own methods.
BuildExpr() : constructs an expression with given context and type.
GetNextContext() : returns an insert context available next.

ato_TCG

AnatropDo() : this function is responsible generating code with given ranges by using
MyBuilder and ProbManager classes.
InsertFunctionListToMain() : this function inserts function calls to main in IR structure
and returns number of functions inserted.

ProbManager class

Attributes:
hUln64 seed : random seed.
hReal32 threshold : threshold value for ignored probabilities.
vector<MyContextType, vector<MyExprType> > exprProb; this list contains expression
probabilities in given contexts.

Methods:
P() : returns probability of given expression in given context.
4.2.1 Test Case Generator Detailed Design

Get seed from file
Generate probabilities for expressions due to contexts
Read other options from files
Generate number of functions
Generate number of statements for each function
Generate number of expressions for each function
Generate number of lines in functions
  Generate functions with names and attributes
  Create a function list

FOR number of functions in the list , FUNC
  FOR number of statements in FUNC , STMT
    Get type of statement
    Create statement
    Push to function body
  END
FOR number of expressions in STMT
  Get a random type expression
  Create expression
  Push to Function body
END
END
FOR all functions
  FOR all statements
    FOR all objects
      Put “printf” function
    END
  END
END

FOR all expressions
  FOR all objects
    Put “printf” function
Create “main” function
Push function calls via function list

FOR all function in function list
    Put optimization specific pattern
END
    Put optimization specific pattern

**Explanation of algorithm:**

Test case generator, first of all, needs a seed value. Also user will be able to give some specific options from our configuration file. This seed will be used in generating some attributes of functions. After distribution of probabilities, it starts to generate functions with predefined attributes.

Attributes only determines number quantities, type of expressions or other type based attributes are derived from a probability class which dynamically calculates and decide on type property.

All variables will be printed to the screen with a second pass in IR. Also main will be written via help of global function list.

Last section is not a must but in our opinion it will be useful for testing optimizations. Some characteristic type of code patterns which are predetermined and constructed independent of probability or other qualifiers. This patterns will be inserted randomly into function scope or global scope. Although some of optimizations can not be constructed with this approach, it will be useful if we thought loop optimizations which are mostly non-algorithmic optimizations.

**4.3 Optimizations**

All optimizations are derived from Anatrop Class in framework. We will override necessary methods. AnatropDo() is main method of this class. It is responsible for operations in Anatrop Class. Our explanations will be done in AnatropDo() unless another method is given.
4.3.1 Constant Folding

We will use predicate class from framework for learning type of expressions and also we will learn if they have constant operands. According the type of expression we will do operations.

4.3.2 Global Copy Propagation
To begin Global Copy Propagation we first get copy assignments using data-flow analysis. We put the copy pairs in a set (Copy) and use this set while we go inside basic blocks. By using this pairs when we see the copied expression in the basic blocks we replace by it' pair that comes from copy pairs from set Copy. We also define another set (Kill) that holds the information about the pairs that we will kill in a basic block. We will not define another method for this optimization we will use the framework DFA class for the copy pairs and we will also use framework replacement method.

4.3.3 Local Copy Propagation

The same manner in global copy propagation but it is done in one basic block but can be extended.

4.3.4 Local Common Subexpression Elimination
To do local common subexpression elimination we iterate through the basic block and add the entries or remove the entry from a set that has a elements having tuples \{pos,op1,op,op2,temp\} where pos is position of the expression in the basic block op1 is the op is the operation and op1,op2 are the operands .tmp is the temporary variable. While we are iterating in the basic block we put the expressions in this set with a nil valued tmp variable. If we op is commutative and if we find the exact op1,op2 in another expression in the basic block we change create a new tmp that it holds the expression. And we put this new tmp tupled element before the position where expression is first added to this set and delete the first added element .we plan to write a method (insert_new()) to insert temporary variable to basic block and all other operations will be done using the framework utilities.

### 4.3.5 Global Common Subexpression Elimination
We will use the same manner that we do in Global Copy Propagation. We will use the DFA class from the framework to find the available expressions. We will use again sets to add the available expressions and kill them. Because it is global we use the basic block numbers too. First we have to look backward if the expression is not assigned in the block. If it is assigned then expression is not a global common subexpression and we will proceed with another expression. If not assign it to a new temp variable. Again we will use these temp variables to replace these with the ones that have the same expression in another basic block. Kill procedure will be done in an appropriate basic block.

4.3.6 If Simplifications

We will define two extra methods for this optimization. One for deleting the unnecessary if statement (remove_branch()) and the other for combining the combining two if statement into one if statement(combine_branch()). Algorithm based on the value of the variable in if. If the variable is known to be either true or false then we will delete this if and only the expressions in if statement will remain. If there are several if statements with the same variable and that variable never changes in the flow of the code then we will put all the expressions in the ifs into just one if statement

4.3.7 Unreachable Code Elimination
Because unreachable code elimination is the code that can nor be executed to do this optimization we will look at the successor and predecessor of the basic blocks. When there is no non-empty path from the ‘entry’ block to a block then we will delete this block. Then we will look for the non-empty path for the successors of the deleted block. We will not implement any other method for this algorithm. We will use framework methods for finding predicate object for non-empty paths and extract object for the successors of the block.

4.3.7 Dead Code Elimination

For the dead code elimination we will hold a set (work list) that each instruction is placed in this work list accordingly if they are essential or not. By saying essential it
means that if the instruction is returned, outputted by a method or affects a storage location. If the instruction is essential we will mark that instruction. After all we will remove all the instructions that are unmarked. For this optimization we will implement 3 methods. First we will implement a method that looks for the essential instructions in a function and put them in a work list. (**FindEssentials()**). Then we will write a method that removes all the instructions in the function that is unmarked (**RemoveUnmarked()**). And finally we will write a method that it removes a basic block which is empty. (**RemoveBasicBlock()**)

### 4.3.8 Tail Merging

For tail merging we will search for a basic block and its successors. We will put the instructions of a basic block in a list and then we will give this list to a function (**SearchForSameInst()**) with its successors. If the last instructions of the list and the last instructions of the successors basic block then we will replace the instructions and make the arrangement of the blocks.
4.3.9 Jump Optimizations

In the jump optimization we will plan to solve the problem that if the unconditional branch whose target is the next instruction in this situation we will delete the branch. We plan to write a function (RemoveBranch()) to delete branch.
4.3.10 Global/Local Forward Substitution:

It is the inverse of the common-subexpression elimination. We use the expression in temp variable (not the temp variable) in another instruction. Of course to do that substitution there must not have been a modification of the variables in the temp variable. We will not implement any other method for this optimization we only use framework objects and functions.

4.3.11 Strength Reduction
4.3.12 Basic Block Ordering

4.3.13 Dead Object Elimination

4.4 Loop Optimizations
An important aspect of loop optimizations in compiler theory is the dependence relation or dependence graph, that includes, for a given sequential program, a collection of statement-to-statement execution orderings that can be used as a guide to select and
apply transformations that preserve the meaning of the program. Each pair of statements in the graph is called dependence. Given a correct dependence graph for the program, any ordering-based optimization that does not change the dependences of a program will be guaranteed not to change the results of the program.

In CStar Framework this dependencies can be controlled by two main classes these are:

- InstDataDependence
- CFG

These classes are responsible from dependencies of source code. Here are explanations of class diagrams.

**Loop Reversal**

- GetLoopDirectionVector() : Find the direction vector of given loop construction. (By vectorization algorithm).
- FindInnerMostLoop() : Finds the innermost loop in given loop nest.
- TestDirectionForReversal() : This method checks condition for reversal is suitable.

**Loop Fission**

- CheckLoopForFission() : Controls loop data dependencies for availability to fission operation.
- SplitLoop() : this method splits loop in to several parts properly.

**Loop Skewing**

- GetDependenceGraph() : Build an array of dependencies of given nested loops.
- GetDirectionVector() : returns the direction vector of given loop in nested loops.
**Loop fusion**

GetDependenceGraph () : Build an array of dependencies of given nested loops.

CompareInnerExprDep() : Compares the dependencies of given two loops .
CombineLoops () : Construct a new loop with given loops and deletes given loops.

**Loop unroll**

Checkloop() : tests whether loop is in simple form and contains no dependencies outside loop context.

UnrollLoop () : this method produces a new loop with same context but do it in less iterations.

**Loop Unswitching**

CheckLoop() : checks if loop has a conditional statement and is it simple.

ExtractConExpr() : finds conditional expression in given loop context.

GetDependenceGraph () : Build an array of dependencies of given loop .

AnatropDo () : main function which carries out conditional expression outside of the loop.

**Loop interchange**

FindInnerMostLoop () : find the inner most loop in given loopnest.
GetDirectionVector() : returns the direction vector of given loop in nested loops.

InterChangeLoops() : inter change places in IR structure of given loops.

GetDependenceGraph() : Build an array of dependencies of given loop.

**Loop Splitting**

GetThreshold() : a value for looking how many iterations will be checked.

CheckForSplit() : controls the loop for availability for splitting with given threshold value.

SplitLoop() : splits loop and returns expression which will be dumped out of loop structure.

### 4. 5 Algorithms of Optimizations

Here are the algorithms designed by us. Most of compiler optimizations are NP-Complete but we determined several cases which we handle. Our criterion is capabilities of framework and performance issues.

**Procedure** : Constant Folding *(Scope:: Basic Block)*

Collect Expressions in Basic Block // ExprCollector Class

IF Expr->left is constant AND Expr->right is constant //Predicate

Create Integer Constant //IRExpr class

IF Expr->type is addition

Expr->value = Expr->left + Expr->right

IF Expr->type is subtraction

Expr->value = Expr->left – Expr->right

IF Expr->type is division
Expr->value = Expr->left / Expr->right

IF Expr->type is multiplication

Expr->value = Expr->left * Expr->right

Replace old expression with new one //IRBuilder Class

Procedure : Unreachable Code Elimination (Scope:: Basic Block)

IF there is NOT a path from entry to basic block //predicate

IF basic block is NOT entry //predicate

IF basic block is NOT exit //predicate

Remove basic block //IRBB class

Procedure: Strength Reduction (Scope:: Statement)

Collect Expression of multiplication //ExprCollector

IF expr->left OR expr->right constant //predicate

IF expr->left->value OR expr->right->value is power of 2
    Get basis of constant value
    Replace multiplication with shift left operation //IRBuilder

Collect Expression of division

IF expr->left OR expr->right constant //predicate

IF expr->left->value OR expr->right->value is power of 2
    Get basis of constant value
    Replace multiplication with shift right operation //IRBuilder
**Procedure**: `IF_Simplifications (Scope:: Statement)

Case 1:

**IF** statement is **IF** statement

**IF** condition expression is constant      //Predicate

  **IF** constant evaluates true      //Predicate

  Delete false case basic block     //IRBuilder

  **ELSE**

  Delete true case basic block     //IRBuilder

Case 2:

Collect true case statements      //StmtCollector

Collect false case statements     //StmtCollector

  **IF** statement is in both lists      //predicate

  **IF** statement isn’t used in basic block      //predicate

    Move statement before **if** statement

**Procedure**: Local Copy Propagation (Scope:: Basic Block)

FOR all statements in basic block     //ListIterator class

  **IF** statement is a copy assignment      //predicate

  Push statement->LHS to tuplelist

  Push statement->RHS to tuplelist

FOR all tuples in list

  **IF** statement->LHS is equal in another tuple->LHS

    Remove tuple

END

Collect binary expression      //ExprCollector

FOR all expressions in expression list

  FOR all tuples in list

    **IF** tuple->LHS is used in expression->RHS

      Replace expression->RHS with tuple->RHS      //IRExpr

    **IF** tuple->LHS is used in expression->LHS


Procedure: Global Copy Propagation (Scope::Function)

FOR all statements in function //StmtCollector
    IF statement is copy assignment
        Push statement->LHS to tuple
        Push statement->RHS to tuple
    END
FOR all basic blocks in function //ListIterator
    Do local copy propagation with tuple
END

Procedure: Dead Code Elimination (Scope::Function)

Do unreachable code elimination //AnatropManager
Collect all statements in function //StmtCollector
FOR all statements
    Collect expression //ExprCollector
    FOR all expressions
        IF expression is object definition //IRObject
        IF definition isn’t used //IRDUUDchairs
            Remove expression //IRBuilder
Procedure : Local Common SubExpression Elimination (Scope::Basic Block)

Create AvailableExpr object //AvailableExpr
Collect assignment statements //StmtCollector
FOR all statements
   Collect Expressions //ExprCollector
   FOR all expressions in statement
      FOR all expressions in AvailableExpr
         IF expr is NOT in list
            Mark unused
            Add binary expressions to AvailableExpr object //AvailableExpr
         ELSE
            IF not marked //AvailableExpr
               Create a temporary object //IROTmp
               Push value of expression to object //IROTmp
               Put object before expression //IRBuilder
               Mark used //AvailableExpr
            ELSE
               Change expression with temporary object //IRExpr
         END
      END
   END
END

Procedure : Global Common Subexpression Elimination (Scope::Function)

Collect basic blocks in function //Extract
FOR all basic blocks in function
   DO local common subexpression elimination //AnatropManager
END

Procedure : Tail Merging (Scope::Function)
Collect basic blocks in function

FOR all basic blocks BB1

   FOR all basic blocks BB2

      IF BB1->successor EQUAL BB2->successor

      Push basic blocks to available list

   END

END

Reverse available list

FOR all basic blocks in available list

   IF statements match in list

   Push statements into match list

END

FOR all basic blocks in available

   remove statements from BB1 in available list

   Create local label beginning of statements in BB2

   Put goto statement into BB1

END

Procedure : Local forward Substitution (Scope::BasicBlock)

Collect statements in basic block

FOR all statements

   Collect assignment expressions EXPR1

   Collect assignment expressions EXPR2

   FOR all expressions in EXPR1

      FOR all expressions EXPR2

      IF EXPR1->LHS is used in EXPR2->RHS

      Replace EXPR1->LHS with EXPR2->RHS

   END

END

Procedure : Global forward Substitution (Scope::Function)

FOR all basic blocks in function

   //Extract
DO local forward substitution //AnatropManager

END

**Procedure**: Jump Optimizations aka Branch Elimination (**Scope**: Basic Block)

Collect statements in basic block //StmtCollector

**IF** number of statements is **EQUAL** 1

Find successor of basic block //extract

Find predecessor of basic block //extract

Connect successor and predecessor //IRBuilder

Remove basic block //IRBuilder

**Procedure**: Basic Block ordering (**Scope**: Function)

Get entry basic block of function //extract

Push to physical order list

Find successors of basic block //predicate

Order successor by minimum jumps

**FOR** all basic blocks in queue

pop basic block of minimum jumps

push successors of popped basic block into physical order list

order blocks by minimum jumps

**END**

Set physical placement of basic blocks as in physical order list //CFG

**Procedure**: Loop Unrolling (**Scope**: Loop)

Collect all statements of Loop //StmtCollector

Get induction variable //IRLoop

**IF** induction variable is **NOT** integer

QUIT

**FOR** all statements

**IF** statement is function call AND induction variable is parameter

Set available true
END
IF available is NOT TRUE
QUIT

Reduce induction variable
Insert function calls with new values    //IRBuilder

Procedure:: Loop Interchange (Scope::Loop)

IF loop has inner Loop    //predicate
IF iteration of inner loop is bigger than outer loop    //IRLoop
IF induction variables are integer constant    //Predicate
IF induction variable isn’t dependent    //IRInductionVar
Replace induction variables    //IRInductionVar,IRLoop

Procedure:: Loop Fusion (Scope::Function)

Collect all loops    //StmtCollector
FOR all loops, loop1
  FOR all loops, loop2
    IF loop1 and loop2 has same header    //IRLoop
    Push loops into list
END

FOR all loops in list
  Insert statements from loop1 to loop2    //IRBuilder,ICStmt
END

Procedure:: Loop Fission (Scope::Loop)

Collect all statements    //StmtCollector
IF induction variable isn’t dependent    //IRInductionVar
FOR all statements
  IF statement is constant copy assignment    //predicate
Push to list
ELSE
QUIT
END

FOR size of list
Create loop      //IRBuilder
Insert statement from list to basic block of loop   //IRLoop
END

Procedure : Loop Unswitching (Scope::Loop)

IF loop has inner Loop       //Predicate
QUIT
Collect statements        //StmtCollector
FOR all statements
    IF statement is type of "IF" statement
        Push statement to list
END
IF size of list is NOT EQUAL 1
QUIT
Build IF statement       //IRBuilder
Insert loop into true case basic block   //IRBuilder , IRIF
Insert loop into false case basic block   //IRBuilder , IRIF
Replace loop with IF statement     //IRIF

Procedure: Loop Reversal (Scope::Loop)

Get header of loop   //IRLoop
IF loop has dependent variables   //IRInductionVar
QUIT
Set induction variable start to its END
4.5 Optimization Manager Module
Now, we will give short explanations of all the classes in the class diagram and their members.

**Anatrop** : This class is the template class of all anatrops in the framework. Because of optimization manager is also an anatrop, it is inherited from base anatrop class.

**ato_OPTIMIZATION** : We used this class to show the relation between the anatropManager and the optimization anatrops. So this class holds for any optimization anatrop.

**Generic Options** : We also showed the Generic Options class in the diagram, because it is used by the ato_OPTMANAGER to hold the option sets of an anatrop variable, to give them as arguments to the execute operation.

**AnatropManager** : anatropManager object is also used by the optimization manager anatrop to do operations to the optimization anatrops with execute, create, etc functions.

**ato_OPTMANAGER**: This class is the anatrop for the optimization manager. It is inherited from Anatrop class as others. AnatropDo() function is the class that performs management of optimizations. It uses the mylexer,myparser,anatropManager and generic options objects to manage the optimization anatrops.

**MyLexer** : It does the lexing of the grammar. Its member attributes are:

- **isInteractive** → Boolean variable that holds whether we are in interactive mode.
- **tokens** → it is the vector of Token class objects that hold the tokens of language.

And its functions are:

- **Mylexer** → constructor function is called with isinteractive boolean and filename if we are in noninteractive mode.
- **setTokens** → this method collects the tokens in the code in a token vector.
- **getTokens** → this method returns the tokens vector in the end of lexer operation.
- **nextToken** → this method is called to find the next token in a iterative mode.
- **matchID,matchIF,matchWHILE.....** → these methods are used to match tokens by looking at the lookahead.
getError → this method holds the possible lexer error strings.

**MyParser**: Myparser does the parsing evaluating, giving errors, symboltable construction...

Its member attributes are;

isInteractive → boolean variable that holds whether we are in interactive mode.

filetokens → the vector of tokens returned from the lexer in noninteractive mode.

stdout tokens → the tokens vector returned from the lexer in interactive mode.

filetokenindex-stdouttokenindex → the index of the token to get the next token.

parseandexecute → boolean value that is used for the if-else statement, whether we should do evaluation while parsing.

And the methods are;

MyParser → constructor function is called with the file_name;

getFileTokens-getStdoutTokens → get the tokens by calling the lexer in interactive or noninteractive mode.

matchToken → gets the token type as input and matches it with the token value. It is also used to report some syntax errors.

program,declarations,setOptions,..... → These are the rules of the grammar. While parsing this is called in a top-down behave.

getError → this method holds the possible parser and type checking error strings.

**Sdirected**: This class is needed by the parser class to hold an {value,type} tuple in syntax directed evaluation.

**Token**: This class holds the tokens with {tokentype,tokenvalue} tuple.

**SymbolTable**: It is the class that holds the symbol table of the anatrop variables that are defined by the user in the code. It also holds the option sets that can be changed.

Its member attribute is;

symtable → vector of Symbol class objects

And the methods are;

addSymbol → It adds a symbol when it is defined correctly. It adds the option set by taking it from the optimizations table.

setOption → It sets the option that want to be set with the given value.
isSymbolThere ➔ It is used to query, when a symbol is used.
isOptionThere ➔ It is used to query whether that variable has such an option.
getOptionSet ➔ It gets the option set when an anatrop variable is defined.

Symbol : It holds a symbol in the symbol table.
Its member attributes are;
optimzname ➔ name of the optimization anatrop.
varname ➔ the name given by the user in the declaration.
options ➔ option set of the variable.
Symbol ➔ the constructor function is called with the optimization name, variable name,
and the option set.

OptimizationsTable : It is used to hold the option sets of each optimization anatrop with
the default values. Its member attributes are;
optimtable ➔ vector of Optimizations object which holds the table.
And the methods are;
addOptim ➔ adds an Optimization object to the table.
constructTable ➔ It constructs the table by adding all the current optimizations.
getOptionSet ➔ It is called from the symbol table to get the option set of a optimization.

Optimization : It is the object that holds the option set of an optimization.
Its member attributes are;
optim_name ➔ name of the optimization anatrop.
options ➔ vector of options that hold the option set of an anatrop

Option : It holds an option with the \{optionname,optiontype \} tuple.
5- System Model

5.1 DFD and Explanation

5.1.1 DFD level 0
5.1.2 DFD level 1

5.1.3 Explanation of DFD

For our project, most of the data flows are function calls, object returns etc. So we think we couldn’t exactly show the DFD diagram. So we will try to explain the interaction of the main classes in our project, in the detailed DFD explanation.
1) Anatrop Manager

Anatrop manager is a singleton object named anatropManager that creates and executes anatrops. Analysis, transformations and optimizations are called anatrop in the framework. Before writing an anatrop, first it should be registered in RegisterAnatrop function of AnatropManager class. AnatropManager creates anatrops with the CreateAnatrop function with the scope of the anatrop that will be defined. Execution of anatrops is also the anatropManager’s job. In execution, the option set is also in the parameter of ExecuteAnatrop function. While setting the options list, if we will define an option other than the basic options, we have to create an Options class object. Also there is a third main function called Trigger. It adds an anatrop to the triggers list of the current anatrop. If anatrops execute_triggers option is true the trigger will be executed after the anatrop.

2) Optimizations

Optimizations are a list of anatrops that construct the needed compiler optimizations. There will be 20 optimizations class for the optimizations. As each anatrop, optimization anatrops also have to register themselves to the singleton anatropManager object. Optimizations are in the back end of the compilation process. So we will work on the IR of the code. We will optimize the IR code when it is needed. Firstly we will use the ExprCollector and StmtCollector classes, when we want to get specific expressions or statements in given scope. We will also use pred and extr singleton objects to get information about any IR structure. More importantly, we have to use the IR class functions to work on the needed IR structures like GetValue, GetRightExpr, GetChildren, etc). In the end we have to change some IR structures with others. So we have to create new IRstructures. And to do this, we have to use IRBuilder class.

3) Test Case Generator

Test case generator is one of the anatrops in the framework. It will create first the intermediate representation. So mostly it will use the IRBuilder class. To create IR
structures, we will have to use most of the functions in the IRBuilder class. The design of
the code will be defined from the anatropManager class. (number of loops, if
statements, etc). Secondly, while constructing the C code we will use the Debug/Trace
utility.

4) Optimization Manager

Optimization manager is also an anatrop in the framework. It uses
anatropManager object to control the option sets and executions order of the anatrops. It
will read an input text file which is given from the compiler developer. And then it will
interpret the code. It will use the Parser class to evaluate and to check errors. And the
Parser class will get the tokens from the Lexer class. This module has to report the errors
because the input file can be erroneous.

And while parsing the code a symbol table has to be defined to check errors and holding
the defined anatrop variables and their option sets which can be changed by the user. And
in the interactive mode we will read the input from the standard input. And in interactive
mode we should dump IR. So we have to use debug-trace utility at this step. Finally to
hold the options of anatrop objects, we have to use a GenericOptions object.

5) Options Class

There is an “options” object which has the general options of the framework. And
each anatrop has an option object. The standard anatrop options are disabled, repeat-on-
change, force, max-repeat-count, mem-clean-up and execute-triggers. To install options,
the member functions of Options class should be used like addIntOption,
addStringOption, etc. Except the generic options, the options should be initialized. The
generic options should be in the actual options to be used, however. All anatrops have
InitOptions function to do initialization. While doing this the GetOptions() function
should be used to get the current options list of the anatrop and to make an addition to the
list. And OnOptions() function is called from the anatropManager object.

6) Extract Class
It is used via ‘extr’ singleton object. This class is defined for getting information about any IR structure.

7) Predicate Class

As in the extract class, predicate class functions can only be used by the singleton ‘pred’ object. By using predicate functions we can make queries about the IR structures. So predicate class functions IR structures and return boolean values.

8) Expression and Statement Collectors

ExprCollector template class is based on ExprVisitor. We use ExprCollector, when we want to get expressions of given type in the scope. Then ExprCollector expressions will be placed in a vector, which we can traverse. StmtCollector does the same job for the statements.

9) IR

IR class is the base class for all the IR constructs like functions, expressions, statements, types, etc. When we have all the IR structures in our hand, we will need to use the functions of IR objects (may be the type of the structure). One important subclass of IR main class is IRTree which is the base class of IRExpr and IRStmt expressions. IRExpr hold the IR expressions like int constants and IRStmt hold the IR statements like if statements. So all the IR structures as IRModule, IRfunctiondefinition and Irprogram are in the IR classes.

10) IR Builder

IRBuilder can be used directly using irBuilder Singleton object. However if we want to create IR objects, an IRBuilder object has to be constructed. IR objects can only be created by IRBuilder class. To add statements to the IR, the insertion place has to be defined by the IRBuilder.(like end of basic block context).
5.2 Use Case Diagram
5.3 Activity Diagrams

5.3.1 Optimization Manager
5.3.2 Test Case Generator

Get the command to run Test Case Generator

Parse configuration file

Probability calculations

Abstraction of IRbuilder

Change IR structure

Insert new functions to main

Dump C code
Although we did not make the whole implementation, we thought about a rough test strategy. That is, the following information is likely to be changed.

We plan to test all units of the project individually first. Afterwards, we will make integration tests.

### 6.1 Unit Testing

**Optimizations**

We will check whether the optimizations give correct results or they fail to give desired results. In order to make a performance tuning, first we should know that the reliability of the optimizations we implemented.

These tests can be made manually but we will be having a test case generator module which is specially developed for this purpose.

The test cases that are generated by the module or we supplied manually will include different C code pieces that does a specific job. The code pieces can include different densities of statements and blocks.

In this part of testing we will also look at the orderings of optimizations and which order gives the best result. But this is more related to performance side of the project.

**Test Case Generator**

Test case generator itself is a testing module. Therefore it must be bug-free to make the tests of the other modules correctly.

We will first check its generated C code fragments: Are they syntactically and semantically correct? To make sure of this, we can feed the output of the generator to a well-known C compiler such as GCC.

We will also check the command line functionality of the module in given hardware and operating system criteria.
Optimization Manager

This module supplies user to make selections and orders of implemented optimizations that he/she wishes to be performed. It reads the selections from a file in a special language specification.

It is possible for language grammar to contain some bugs. We will check this firstly. The language must show correct behavior. That is, it must do what the user wants to do.

6.2 - Integration Testing

Integration of the three modules we will implement must be provided without error. These modules will be integrated to the QuickC framework also. They must be fully compatible to the classes of the framework. So, this phase of testing will be proceeded very carefully.

The integration testing step will contain performance comparison. We think that a comparison with GCC will give us some idea about the performance of the project. Though we do not expect better or same performance, it will be useful to have feedback and see where we are at.

We will also use benchmarks to test the performance of the software. We will use Whetstone and Dhrystone benchmarks that are given to us.

In case we have undesired results during integration testing, we will be able to turn back to unit testing of specific module and debug the module if needed.

7 – Project Schedule

Gannt chart is given in appendix A.
<table>
<thead>
<tr>
<th>ID</th>
<th>Task Name</th>
<th>Start</th>
<th>Finish</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Team Gathering and Determining Project Topic</td>
<td>2/9/2007</td>
<td>0/1/2007</td>
<td>4d</td>
</tr>
<tr>
<td>2</td>
<td>Determining Project Topic Details and Proposal</td>
<td>2/9/2007</td>
<td>0/1/2007</td>
<td>8d</td>
</tr>
<tr>
<td>3</td>
<td>Analysis of the Project</td>
<td>0/5/2007</td>
<td>0/2/2007</td>
<td>25d</td>
</tr>
<tr>
<td>4</td>
<td>Requirement Analysis</td>
<td>0/5/2007</td>
<td>0/9/2007</td>
<td>5d</td>
</tr>
<tr>
<td>5</td>
<td>Literature Survey and Market Research</td>
<td>0/9/2007</td>
<td>1/9/2007</td>
<td>1d</td>
</tr>
<tr>
<td>7</td>
<td>Project Scheduling</td>
<td>1/9/2007</td>
<td>2/9/2007</td>
<td>4d</td>
</tr>
<tr>
<td>10</td>
<td>Milestone I</td>
<td>2/2/2007</td>
<td>2/2/2007</td>
<td>0d</td>
</tr>
<tr>
<td>11</td>
<td>Practicing Framework</td>
<td>2/2/2007</td>
<td>0/5/2007</td>
<td>4d</td>
</tr>
<tr>
<td>12</td>
<td>Initial Design of the Project</td>
<td>0/5/2007</td>
<td>3/3/2007</td>
<td>26d</td>
</tr>
<tr>
<td>13</td>
<td>Design Discussion</td>
<td>0/5/2007</td>
<td>0/9/2007</td>
<td>4d</td>
</tr>
<tr>
<td>14</td>
<td>Information Gathering about Optimizations, Test Case Generator and Optimization Manager</td>
<td>0/5/2007</td>
<td>2/6/2007</td>
<td>22d</td>
</tr>
<tr>
<td>16</td>
<td>Developing Algorithm of Optimizations</td>
<td>1/9/2007</td>
<td>1/9/2007</td>
<td>8d</td>
</tr>
<tr>
<td>17</td>
<td>Specifying Class Diagrams</td>
<td>1/9/2007</td>
<td>2/6/2007</td>
<td>9d</td>
</tr>
<tr>
<td>18</td>
<td>Test Case Generator Design</td>
<td>1/9/2007</td>
<td>2/6/2007</td>
<td>17d</td>
</tr>
<tr>
<td>19</td>
<td>Test Case Algorithm Extraction</td>
<td>1/9/2007</td>
<td>1/9/2007</td>
<td>9d</td>
</tr>
<tr>
<td>20</td>
<td>Specifying Class Diagram</td>
<td>1/9/2007</td>
<td>2/6/2007</td>
<td>8d</td>
</tr>
<tr>
<td>22</td>
<td>Optimization Manager Language Design</td>
<td>1/9/2007</td>
<td>1/9/2007</td>
<td>8d</td>
</tr>
<tr>
<td>24</td>
<td>Specifying Class Diagram</td>
<td>2/2/2007</td>
<td>2/6/2007</td>
<td>5d</td>
</tr>
<tr>
<td>29</td>
<td>Optimizations Design</td>
<td>7/7/2007</td>
<td>3/9/2008</td>
<td>26d</td>
</tr>
<tr>
<td>32</td>
<td>Main Class Abstraction</td>
<td>7/7/2007</td>
<td>1/7/2007</td>
<td>1d</td>
</tr>
<tr>
<td>33</td>
<td>Examining Probability Distribution</td>
<td>1/9/2007</td>
<td>2/7/2007</td>
<td>10d</td>
</tr>
<tr>
<td>35</td>
<td>Optimization Manager Design</td>
<td>7/7/2007</td>
<td>3/9/2008</td>
<td>26d</td>
</tr>
<tr>
<td>36</td>
<td>Learning and Using Linear Parser Generator Software for Our Language</td>
<td>7/7/2007</td>
<td>1/7/2007</td>
<td>13d</td>
</tr>
<tr>
<td>38</td>
<td>Final Design Report</td>
<td>0/4/2008</td>
<td>1/10/2008</td>
<td>8d</td>
</tr>
<tr>
<td>39</td>
<td>Milestone III</td>
<td>1/10/2008</td>
<td>1/10/2008</td>
<td>0d</td>
</tr>
<tr>
<td>40</td>
<td>Prototype Design and Implementation</td>
<td>3/3/2007</td>
<td>16/10/2008</td>
<td>45d</td>
</tr>
<tr>
<td>41</td>
<td>Project Prototype and Demo</td>
<td>18/10/2008</td>
<td>18/10/2008</td>
<td>3d</td>
</tr>
<tr>
<td>ID</td>
<td>Task Name</td>
<td>Start</td>
<td>Finish</td>
<td>Duration</td>
</tr>
<tr>
<td>----</td>
<td>-----------------------------------------------</td>
<td>-------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>1</td>
<td>General Optimizations Implementation</td>
<td>20020208</td>
<td>15032008</td>
<td>25d</td>
</tr>
<tr>
<td>2</td>
<td>Local Optimizations Implementation and Testing</td>
<td>20020208</td>
<td>02032008</td>
<td>12d</td>
</tr>
<tr>
<td>3</td>
<td>Loop Optimizations Implementation and Testing</td>
<td>20020208</td>
<td>04032008</td>
<td>14d</td>
</tr>
<tr>
<td>4</td>
<td>Global Optimizations Implementation and Testing</td>
<td>20020208</td>
<td>03032008</td>
<td>13d</td>
</tr>
<tr>
<td>5</td>
<td>Interprocedural Optimizations Implementation and Testing</td>
<td>20020208</td>
<td>04032008</td>
<td>14d</td>
</tr>
<tr>
<td>6</td>
<td>Whole Program Optimizations Implementation and Testing</td>
<td>05032008</td>
<td>15032008</td>
<td>11d</td>
</tr>
<tr>
<td>7</td>
<td>Integration and Testing</td>
<td>12032008</td>
<td>19032008</td>
<td>8d</td>
</tr>
<tr>
<td>8</td>
<td>Test Case Generator Implementation</td>
<td>20020208</td>
<td>27032008</td>
<td>37d</td>
</tr>
<tr>
<td>9</td>
<td>Probability Distribution Feature Implementation and Testing</td>
<td>20020208</td>
<td>09032008</td>
<td>19d</td>
</tr>
<tr>
<td>10</td>
<td>Main Class Implementation and Testing</td>
<td>29022008</td>
<td>18032008</td>
<td>20d</td>
</tr>
<tr>
<td>11</td>
<td>Extracted Code Patterns Implementation and Testing</td>
<td>10032008</td>
<td>27032008</td>
<td>18d</td>
</tr>
<tr>
<td>12</td>
<td>Integration and Testing</td>
<td>25032008</td>
<td>04042008</td>
<td>8d</td>
</tr>
<tr>
<td>13</td>
<td>Implementation of Optimization Manager</td>
<td>02042008</td>
<td>04052008</td>
<td>30d</td>
</tr>
<tr>
<td>14</td>
<td>Optimization Manager Language Lexer Implementation and Testing</td>
<td>02042008</td>
<td>12042008</td>
<td>11d</td>
</tr>
<tr>
<td>15</td>
<td>Optimization Manager Language Parser Implementation and Testing</td>
<td>10042008</td>
<td>22042008</td>
<td>13d</td>
</tr>
<tr>
<td>16</td>
<td>Embedding Parser of the Language to Framework</td>
<td>20042008</td>
<td>04052008</td>
<td>12d</td>
</tr>
<tr>
<td>17</td>
<td>Integration and Testing</td>
<td>25042008</td>
<td>05052008</td>
<td>8d</td>
</tr>
<tr>
<td>18</td>
<td>Alpha Release</td>
<td>07052008</td>
<td>07062008</td>
<td>0d</td>
</tr>
<tr>
<td>19</td>
<td>Unit Testing</td>
<td>07062008</td>
<td>21062008</td>
<td>15d</td>
</tr>
<tr>
<td>20</td>
<td>Components White-Box Testing</td>
<td>13062008</td>
<td>25062008</td>
<td>13d</td>
</tr>
<tr>
<td>21</td>
<td>Components Black-Box Testing</td>
<td>15062008</td>
<td>29062008</td>
<td>11d</td>
</tr>
<tr>
<td>22</td>
<td>Beta Release</td>
<td>30062008</td>
<td>30062008</td>
<td>0d</td>
</tr>
<tr>
<td>23</td>
<td>End User Testing</td>
<td>30052008</td>
<td>14062008</td>
<td>16d</td>
</tr>
<tr>
<td>24</td>
<td>Final Release</td>
<td>15062008</td>
<td>15062008</td>
<td>0d</td>
</tr>
</tbody>
</table>