A

Project Report On

GCC Extension for Detecting Critical Sections in a Multithreaded Environment

Submitted By

Vishal Dawange	**B80058525**
Mayur Jadhav	**B80058549**
Akash Kothawale	**B80058564**
Prasad Muley	**B80058576**

Under the guidance of

Prof. K.K.Nandedkar

In partial fulfilment of

Bachelor of Engineering

[B. E. Information Engineering]

[May 2013]

AT

Department of Information Technology

Pune Institute of Computer Technology

Dhankawadi, Pune 411043

Affiliated to

University of Pune

Pune Institute of Computer Technology

Department of Information Technology

Dhankawadi, Pune 411043

CERTIFICATE

This is certify that the Dissertation entitled **"GCC Extension for Detecting Critical Sections in a Multithreaded Environment"**, submitted by **Vishal Dawange** is a record of bonafide work carried out by him, in the partial fulfilment of the requirement for the award of Degree of Bachelor of Engineering (Information Technology) at Pune Institute of Computer Technology, Pune under the University of Pune. This work is done during year 2012-2013, under our guidance.

____

(Prof. K.K.Nandedkar)

Project Guide

________________________	________________________-
Prof. Emmanual M.	Dr. P. T. Kulkarni
HOD, IT Department	**Principal PICT**

Examination:

Examiner __________________

Date:

Acknowledgements

I am profoundly grateful to **Prof. K.K.Nandedkar** for her expert guidance and continuous encouragement throughout to see that this project rights its target since its commencement to its completion.

I would like to express deepest appreciation towards **Dr. P. T. Kulkarni**, Principal PICT, Pune, **Prof. Emmanual M.** HOD Information Technology Department and **Prof. Manish R. Khodaskar** (Project Coordinator) whose invaluable guidance supported me in completing this project.

I am particularly grateful to **Mr.Vedang Manerikar** (Helpshift Inc.) and **Mr.Gaurav Jain** (Marvell Semiconducor) who allows me to work in the company.

At last I must express my sincere heartfelt gratitude to all the staff members of Information Technology Department who helped me directly or indirectly during this course of work.

Vishal Dawange

Mayur Jadhav

Akash Kothawale

Prasad Muley

CERTIFICATE

This is to certify that the project report entitled

GCC Extension for Detecting Critical Sections in a Multithreaded Environment

Submitted by

Vishal Dawange	**B80058525**
Mayur Jadhav	**B80058549**
Akash Kothawale	**B80058564**
Prasad Muley	**B80058576**

is a bonafide work carried out by them with the Sponsorship from DREAMZ Group under the supervision of Mr. Vedang Manerikar, Mr. Gaurav Jain and has been completed successfully .

(Mr. Vedang Manerikar)	(Mr. Gaurav Jain)
(Designation) External Guide	(Designation) External Guide

Place : Pune
Date:

ABSTRACT

Concurrent programming is gaining popularity, since Moore's law is nearing its end of life. The significant truth is that processors aren't getting faster anymore, but you get more cores.

In concurrent / parallel programming, one of the toughest tasks for programmers is synchronizing the access of shared memory to avoid basic problems of concurrency. While working on a large code base, programmers may miss out synchronizing certain critical section leaving the code vulnerable for race conditions to occur.

We propose an approach that extends the capability of GCC to identify all the critical sections in multithreaded programs which has synchronization bugs, i.e. race conditions may occur due to incorrect/no use of locking and unlocking mechanisms, which will warn the programmers by pointing out the exact location where the problems would occur.

Furthermore, it also provides synchronization to that section of code, by introducing proper locking methods. This stage is optional, and programmers may synchronize the section themselves too.

Keywords : Critical Section, Race Condition, GCC, Locks

Contents

List of Figures

Chapter 1

Introduction

In the recent years, from late 2005, clocks speeds havent advanced anymore. Instead, core counts have increased at the pace clock speed used to, and that has affected more and more programmers [1]. As programmers started migrating from single core to multicore architectural programming, many problems arose.

One of the most difficult parts of multi-threaded programming is when programmers need to handle critical sections (A critical section of a multithreaded program is a section of code where shared data are accessed by the multiple threads) which are responsible for causing data races and are also one of the reasons for causing deadlocks. Data races occur due to incorrect or no synchronization of the threads in the programs.

In order to avoid data races, synchronization is achieved by using lock/unlock mechanism (semaphores, monitors, pthread_mutex, etc.) to limit a threads concurrent access to shared resources, if it is being used by other thread. To resolve the issue of manually handling each and every critical section or to debug large programs with synchronization bugs we are proposing an idea with which we identify critical sections which may cause data races, and avoid them by introducing proper synchronizations.

1.1 Background

In multithreaded programs, we often need to handle cases where critical sections exist, as the programs are at risk of producing unexpected outputs. This is done by introducing locking mechanisms so that only one thread may alter the data at once. This is necessary to avoid race conditions, which generally lead to unexpected results if the locking mechanism doesn't exist.

1.1.1 Critical Section

In concurrent programming, a critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution. A critical section will usually terminate in fixed time, and a thread, task, or process will have to wait for a fixed time to enter it (aka bounded waiting).

By carefully controlling which variables are modified inside and outside the critical section, concurrent access to that state is prevented. A critical section is typically used when a multi-threaded program must update multiple related variables without a separate thread making conflicting changes to that data. In a related situation, a critical section may be used to ensure a shared resource, for example a printer, can only be accessed by one process at a time.

1.1.2 Race Condition

Race conditions arise in software when separate processes or threads of execution depend on some shared state. Operations upon shared states are critical sections that must be mutually exclusive. Failure to do so opens up the possibility of corrupting the shared state.

Race conditions are notoriously difficult to reproduce and debug, since the end result is nondeterministic, and highly dependent on the relative timing between interfering threads. Problems occurring in production systems can therefore disappear when running in debug mode, when additional logging is added, or when attaching a debugger, often referred to as a Heisenbug. It is therefore highly preferable to avoid race conditions in the first place by careful software design than to fix problems afterwards.

Race conditions occur especially in multithreaded, concurrent, parallel or distributed programs [2].

1.1.3 Synchronization

Thread synchronization or serialization, strictly defined, is the application of particular mechanisms to ensure that two concurrently-executing threads or processes do not execute specific portions of a program at the same time. If one thread has begun to execute a serialized portion of the program, any other thread trying to execute this portion must wait until the first thread finishes.

Synchronization is used to control access to state in smallscale multiprocessing systems - in multithreaded environments and multiprocessor computers - and in distributed computers consisting of thousands of units [2].

1.2 Need

When large multithreaded programs are written, its difficult to keep a track of the critical sections in it. This may inturn lead to the occurence of unexpected results or may lead to synchronization bugs. Our project aims to find out these critical sections and introduce a locking mechanism to avoid race conditions in the program.

And current version of GCC does not provide or support any feature to find out the critical sections or synchronization bugs in the multitheaded programming.

1.3 Motivation

Motivation behind Critical Section Detection tool is contribution to open source technology, because now a days there are many technologies which are there already implemented based on GCC or Open Source technologies but each of them was developed considering different parameters in front of them.

Our system is motivated not only considering detecting critical sections but different parameter which are mandatory now a day, like initially we are checking for error free C source code and after that tokenizing and parsing the same source code. So it seems that we need to study all the phases compiler in details and as fat our knowledge about GCC concerns, GCC doesn't provide this featue of detecting critical sections for given source code.

GCC Extension for detecting Critical Sections in a Multi threaded Environment is motivated with respect to many parameter under one basic idea, considering one scenario where one peraon have written a lorge multithreaded code say 1 KLOC and another person tries to work on the same code but unfortunately there are some synchronization bugs, so in such a case it is so difficult to new person to browse that code and detect the suspected critical section or race condiiton. Hence if the tool that we have developed is used by that new person, would become very easy to detect suspected critical section.

1.4 Objective

To develop a GCC Extension for detecting Critical Sections and providing locking and unlocking mechanism or synchronization mechanism which may lead to race Condition or Synchronization bugs in a Multithreaded Environment.

The main objective of the system is, the tool should automatically detect the suspected critical sections which may lead to race conditions or synchronization bugs in a multitheaded program.

Chapter 2

Proposed Work

2.1 Problem Statement

The compiler is made to identify the critical section in a multi-threaded program for the synchronization bugs, which currently is not a feature in gcc. Also, the compiler should automatically take care of the critical sections by introducing helper statements (comments) for Lock and Unlock function calls in a multi-threaded program without involvement of the programmers.

2.2 Scope

The project has been split up in two modules.

Firstly, the project aims to nd out critical sections in a program which may cause race conditions. The reasons for Race conditions occurence are many. For example, simaltaneuos writing on shared variables(like global, extern, volatile), les, pipes. We shall be nding the critical sections of these.

There are situations where prior locks exist and they need to be taken care of. The approach generally used is either a lock free program, or the existing locks are removed and then put back. This causes lot of processing overhead, and hence is ineffecient.

We also intended to contentrate only on the critical sections of the code since doing a point-to analysis of the whole program proves out the be process intensive. In some cases wrong Lock and Unlock may the reason for Race Condition so in such cases detecting CS and inserting Lock and Unlock on proper location.

Chapter 3

Project Design

3.1 Software Requirement Specification

3.1.1 Hardware Requirements

- 2.4GHz Celeron Processor
- 256MB Memory
- 10GB Disk Space

3.1.2 Software Requirements

- [Operating System]: Linux Based Operating System
- [Build-essential]: Build-essential is required to build Debian packages, starting with dpkg (>= 1.14.18)
- [Necessary packages for building GCC]:
  `http://gcc.gnu.org/install/prerequisites.html`

3.1.3 Technologies Used

- C, lex, yacc.

3.2 U.M.L. Diagrams

UML was meant to be a unifying language enabling IT professionals to model computer applications. The primary authors were Jim Rumbaugh, Ivar Jacobson, and Grady Booch, who originally had their own competing methods (OMT, OOSE, and Booch). Eventually, they joined forces and brought about an open standard. (Sound familiar? A similar phenomenon spawned J2EE, SOAP, and Linux.) One reason UML has become a standard modeling language is that it is programming-language independent. (UML modeling tools from IBM Rational are used extensively in J2EE shops as well in .NET shops.) Also, the UML notation set is a language and not a methodology. This is important, because a language, as opposed to a methodology, can easily fit into any company's way of conducting business without requiring change.

Since UML is not a methodology, it does not require any formal work products (i.e., ”artifacts” in IBM Rational Unified Process lingo). Yet it does provide several types of diagrams that, when used within a given methodology, increase the ease of understanding an application under development. There is more to UML than these diagrams, but for my purposes here, the diagrams offer a good introduction to the language and the principles behind its use. By placing standard UML diagrams in your methodology's work products, you make it easier for UML-proficient people to join your project and quickly become productive. The most useful, standard UML diagrams are: use case diagram, class diagram, sequence diagram, state chart diagram, activity diagram, component diagram, and deployment diagram.

A **Use Case Diagram** in the Unified Modeling Language (UML) is a type of behavioral diagram defined by and created from a Use-case analysis. Its purpose is to present a graphical overview of the functionality provided by a system in terms of actors, their goals (represented as use cases), and any dependencies between those use cases. The main purpose of a use case diagram is to show what system functions are performed for which actor. Roles of the actors in the system can be depicted.

A **Sequence Diagram** in Unified Modeling Language (UML) is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart. A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Sequence diagrams typically are associated with use case realizations in the Logical View of the system under development.

Activity Diagram in Unified Modeling Language (UML) is a kind of interaction diagram that shows how activities are carried out in synchronization with one another and in what order. It is a construct of a Flow Chart. A activity diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the activities exchanged between the objects needed to carry out the functionality of the scenario.

3.2.1 Use Case

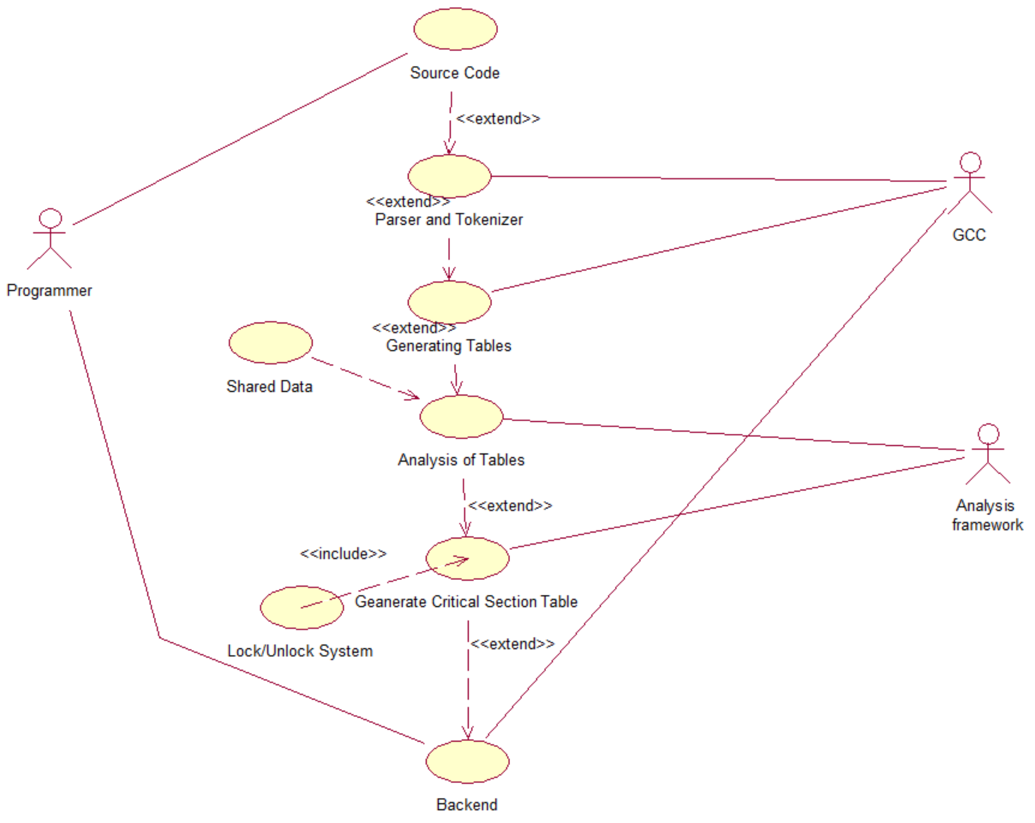

Figure 3.1: Use case diagram

From use case point of view our system contains one actor as user or programmer, in which programmer has to specify a C source code (Source code should be error free and Multithreaded for better results). And on the other hand GCC does the parsing and tokenization with the help of patterns and grammar that we have written, Analysis framework does the detection of critical sections in a source code provided by the Programmer as a input.

3.2.2 State Chart

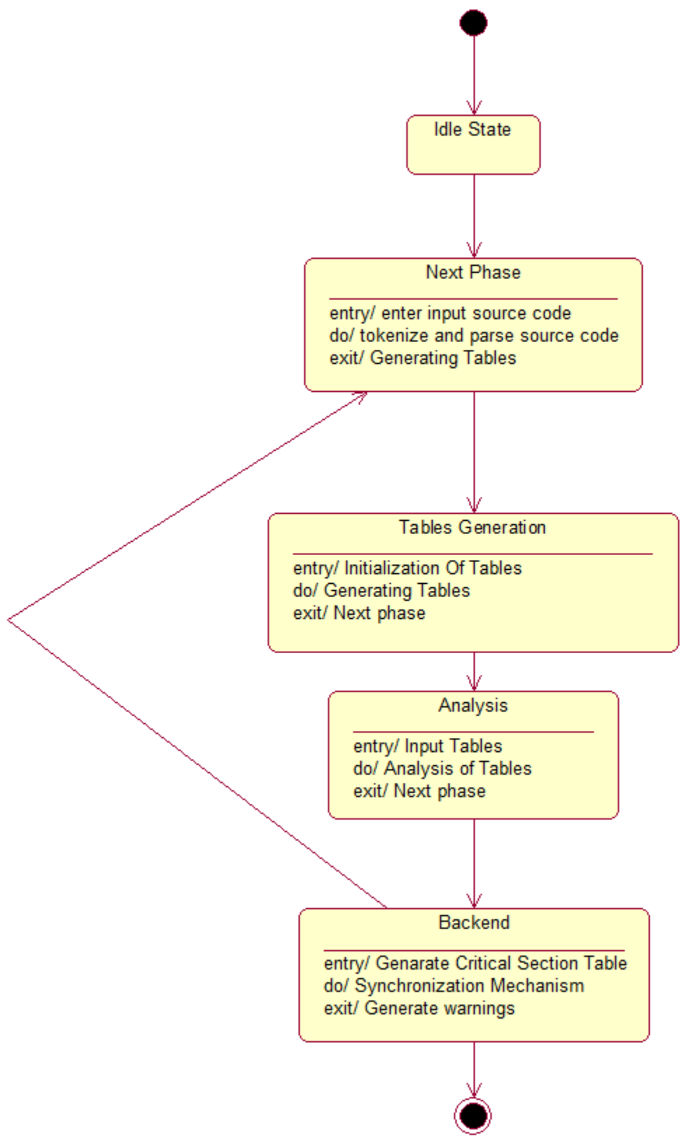

Figure 3.2: State chart diagram

Control Flow of the Critical Section Detection System. Above figure shows the state transisions of our system, this shows that how actually the system is going to transfer from eact step to next step after providing certain input. There are certain steps in each state. Flow of the Critical Section Detection System.

3.2.3 Sequence Diagram

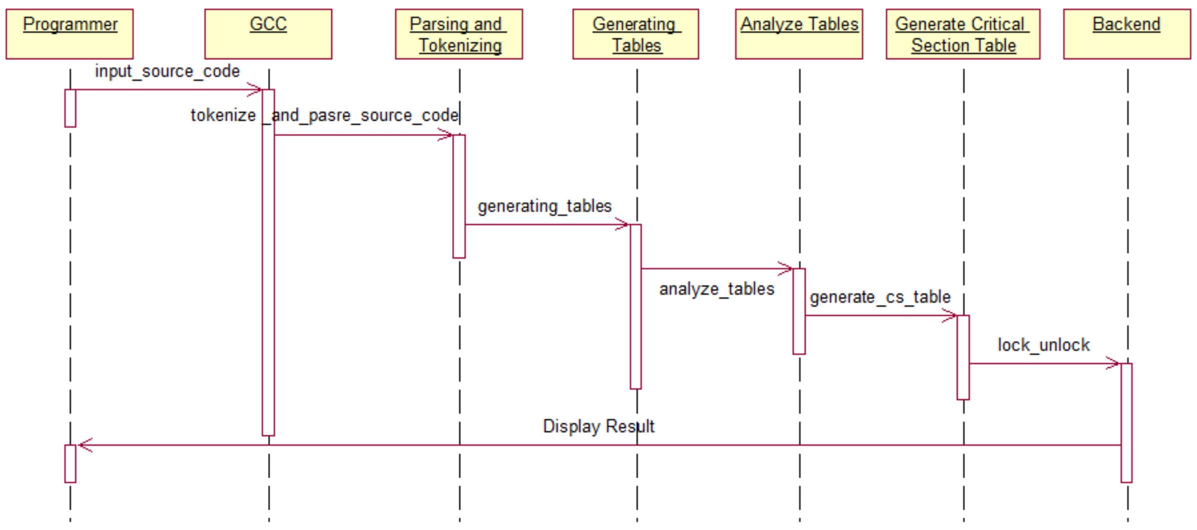

Figure 3.3: Sequence Diagram

Shows the sequences of operations from starting to last one. A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario i.e Programmer, GCC, Analysis Framework etc. and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Scenario could be whole sequence of operations till the end result.

3.2.4 Collaboration Diagram

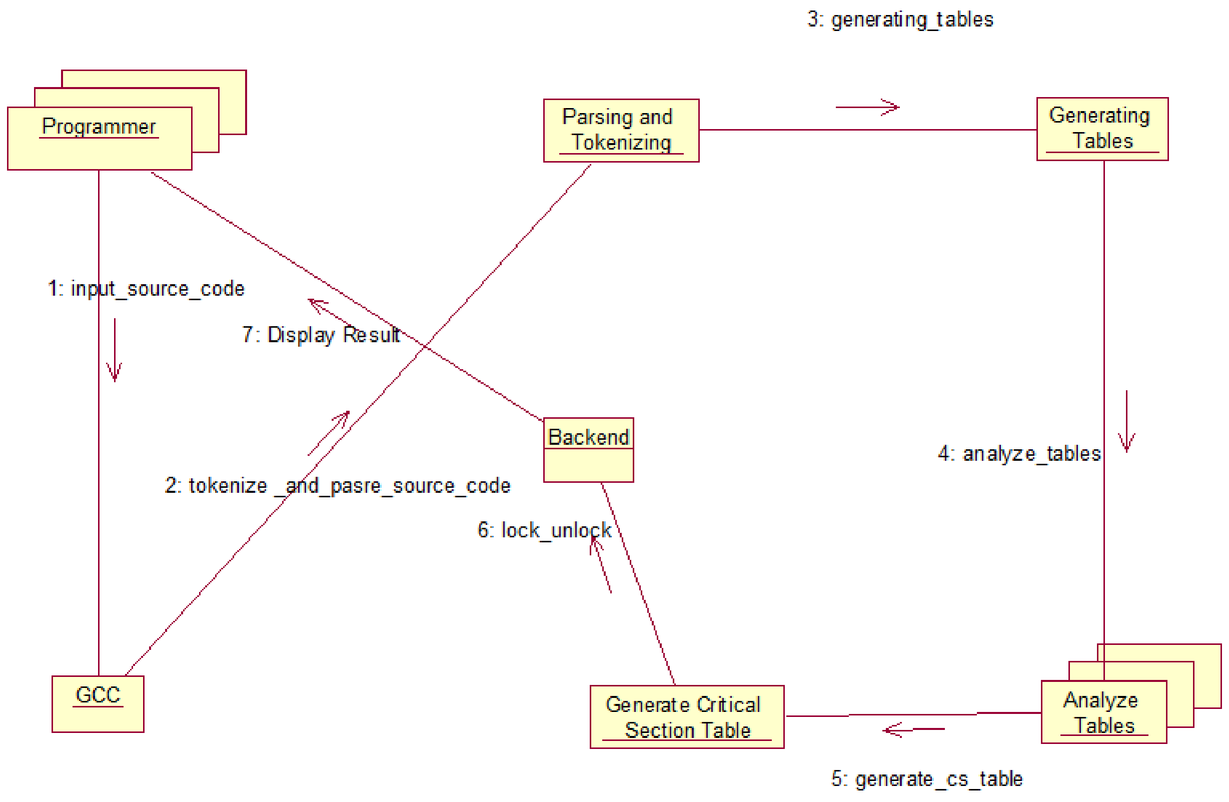

Figure 3.4: Collaboration Diagram

From Collaboarative perspective of our system, it covers the same scenario as shown in Sequence diagram.Indeed the collaborative view of the system is generated by using sequence diagram.

3.2.5 Activity Diagram

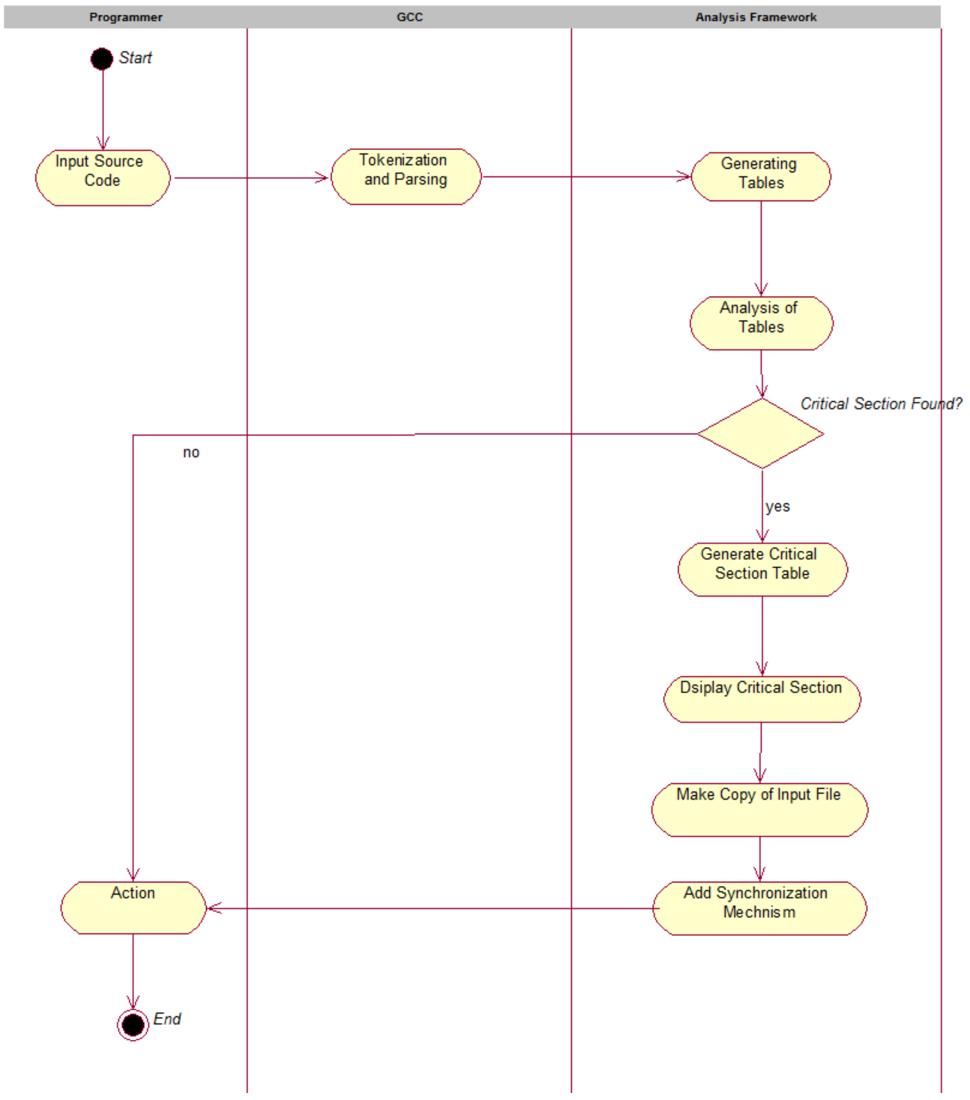

Figure 3.5: Activity Diagram

From programmers prespective activity diagram shows which are the diffrent activities performed by Programmer, GCC and the Analysis Framework.

3.2.6 Component Diagram

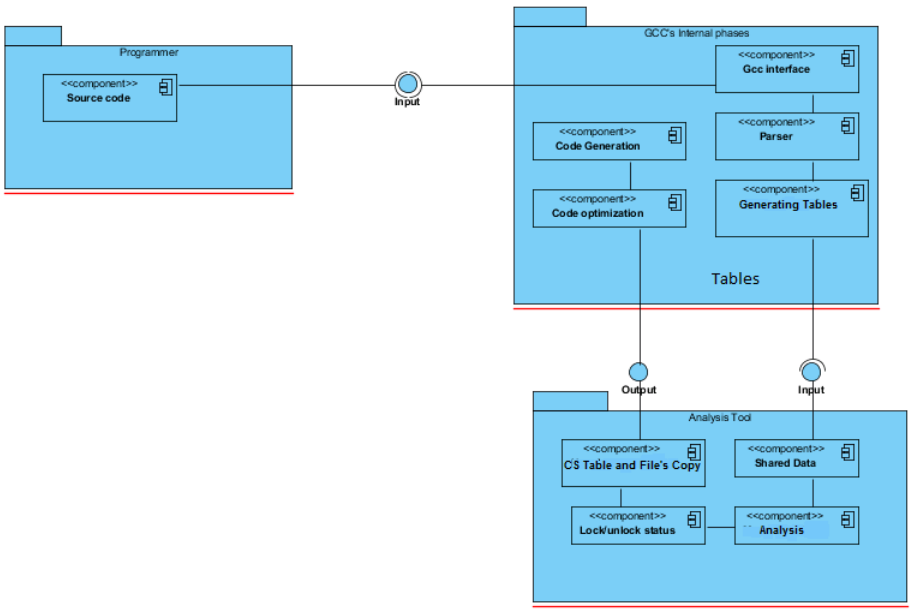

Figure 3.6: Component Diagram

From Component point of view our system having different components used in system like system is Table Generator which generates the tables like Global Symbol's Table, Local Symbol's Table, Log Table, Semaphore Table, Function Table and Thread Table etc.

3.2.7 Deployment Diagram

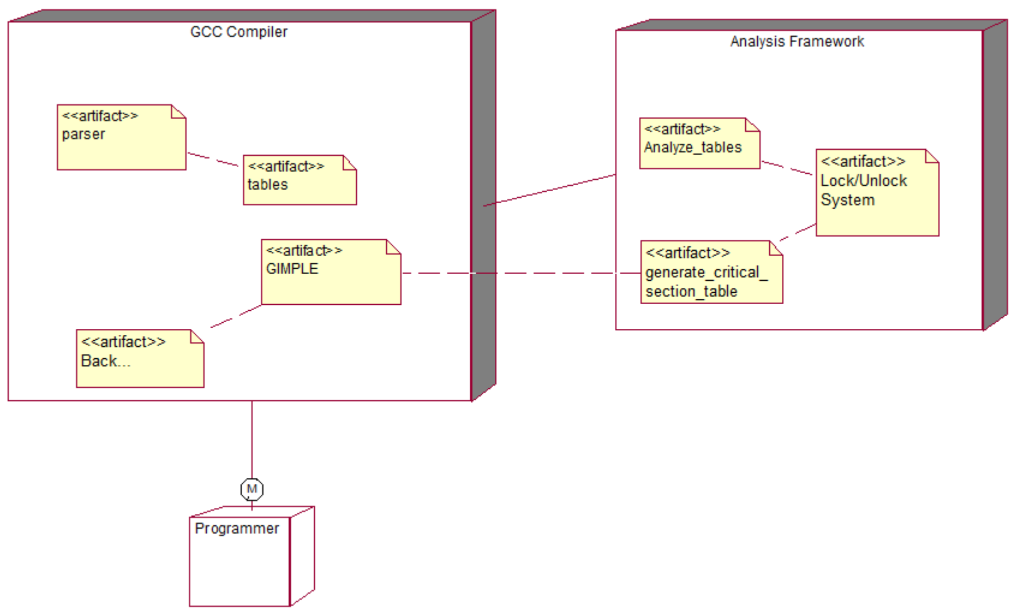

Figure 3.7: Deployment Diagram

From deployment point of view our system contains following hardware and software component which are required for system execution are like GCC Compiler for compiling the source code, lex and yacc for Tokenization and Parsiing and the analysis frmaework for detecting the critical sections and adding synchronization mechanism.

Chapter 4

Implementation

The important aspect of the approach is that we are analyzing the programs statically. This helps us check each and every possible case that may occur during execution of the programs. To implement the current idea of detecting critical sections, we have designed architecture as follows.

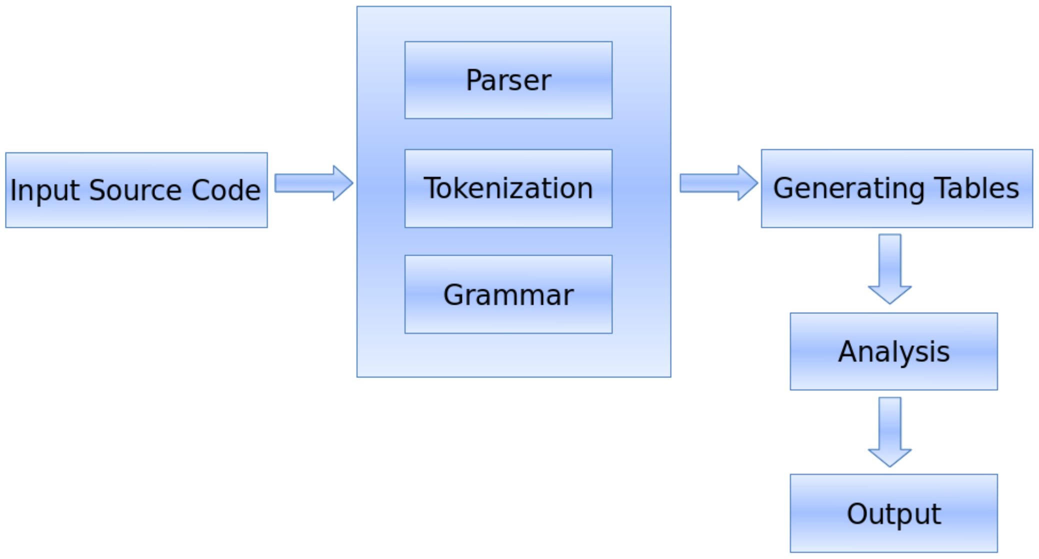

Figure 4.1: Architectural Diagram

4.1 Architecture

4.1.1 Input Source Files

A normal multithreaded program is given as input with critical sections. This file is first compiled to check for errors and only when it is compiled error free; it heads to the next stage. _source.png _source.pdf _source.jpg _source.mps _source.jpeg _source.jbig2 _so
c

4.1.2 Parser, Tokenizer, Grammar

Flex (The Fast Lexical Analyser) is a tool for generating scanners. A scanner, sometimes called a tokenizer, is a program which recognizes lexical patterns in text. The flex program reads user-specified input files, or its standard input if no file names are given, for a description of a scanner to generate [4].

With it we separate all the unwanted text (ex. Comments, pre-processors) and tokenize the program to find out shared memory, functions, threads, and locks. In order to find the above from the token, it analyzes its input for occurrences of text matching the regular expressions for each rule. Whenever it finds a match, it executes the corresponding code [3].

The different in-built functions were monitoring are:

- pthread_create
- pthread_join
- pthread_cond_wait
- pthread_cond_signal
- sem_wait
- sem_post

c

4.1.3 Generating Tables

Yacc (Yet Another Compiler Compiler) provides a general tool for describing the input to a computer program. We specify the structures of the input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a usersupplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification [5].

Once all the unwanted text is ignored from the code, with the help of Yacc we generate the following data structures:

- Global variables table (Id, access, variable_name, line_number)
  Stores the entries of all the shared variables that exist in the program
- User-defined Functions (Id, function_name, return_type, number_of_parameters, line_number)
  Stores the entries of all the functions defined by the user in the program.
- Functions Local symbol table (Id, access_type, name, data_type, function_id, line_number)
  Stores all the local variables associated with a function.
- Semaphore table (id, name, function, location)
  Stores all the locations of sem_wait() and sem_post functions used in the code
- Thread table (Id, thread_name, function_name, function_id, thread_attribute, parameters, parent_thread)
  Stores all the threads associated with which functions and their attributes
- Shared Memory Log table (Id, function_name, shared_object, scope, line_number, thread-function_id, thread_id)
  Stores a log of all variables which may cause data races and helps analysis the proof.
- Critical Section table (Id, shared_object, thread_function_index, first_location, last_location, function_name)

4.1.4 Analysis

Once all tables are generated we analyze them. If any shared variable exists in multiple functions and is unhandled i.e. it is not protected with the function calls sem_wait(), sem_post(), then we categorize that shared variable as unhandled critical section and add its entry to the critical section table.

After getting a table having all the unhandled critical sections, we analyze it for the location where we need to provides locks and unlocks.

From the Critical Section Table, we get the line numbers of first and last occurrence of shared variables usage. Depending on their closeness, we decide the number of times we need to lock, and unlock.

4.1.5 Output

Once all tables are analyzed and we display all the unhandled critical sections that exist. The various attributes of the critical sections that are found are:

- The shared object which is being accessed my multiple threads
- The threads which are accessing the same shared object
- The functions in which the critical section exist
- The location pertaining to the line number of the critical section

We then decide if the programmer handles the critical sections himself, or we have to handle it. Once the permissions are given, we add sem_wait() sem_post(), recompile the code to produce bug free programs.

4.2 Placement of our tool in GCC hierarchy

Our tool exists adjacent to the current architecture of GCC but runs once the compilation is over and no errors are generated. It keeps a copy of the original source code, works on it, and pushes its warnings along with the warnings of the compiler.

Chapter 5

Scheduling

5.1 Proposed Modules

- Parse, Tokenize
- Tables Generation
- Analysis
- Testing

5.2 Scheduling

Chapter 6

Conclusion and Future Scope

6.1 Conclusion

The overall approach helps solve many practical problems.
Bugs that are notoriously difficult to find in concurrent programming are handled by the compiler itself. In large code bases data races possibilities be will perfectly identified and this framework will help automatically detect critical section and provide Lock/Unlock system without involvement of the programmer.

6.2 Future Scope

Although this concept if pretty much full proof, the capabilities can extended further.
As of now, we are only supporting POSIX thread library and semaphore locking mechanism, this project can be extended for other thread libraries and locking mechanism.

- OpenThreads
- Boost C++ libraries http://www.boost.org/
- OpenMP http://openmp.org/wp/
- Intel Cilk Plus http://software.intel.com/en-us/intel-cilk-plus
- ZThreads http://zthread.sourceforge.net/

Since this approach is generic, we can port the same for compilers of other languages. Since critical sections are one of the most important factors of concurrent programming, this port would help programmers of other domain incorporate it with their development and benefit from it.

For now we have only concentrated on shared memory in User space. A possible future enhancement could be that we can tackle shared memory in the kernel space, (for example: kernel level pipes) or handle files as well.

A very difficult (but a possible) concept of adding synchronization mechanisms automatically could be done. However, many complications are associated with it as locking mechanism depends on the application logic, which is practically impossible to identify programmatically.

References