MASTER

Binary instrumentation with QEMU

Aslam, M.J.

Award date:
2016

Link to publication
Department of Mathematics and Computer Science
Electronic Systems Group

Graduation Project Thesis

**Binary Instrumentation with QEMU**

Muhammad Junaid Aslam

Supervisors:

Lech Jozwiak (TU/e)
Jos van Eijndhoven (Vector Fabrics)

Eindhoven, July 2016
Abstract

The subject of this MSc project is binary instrumentation with QEMU. QEMU is an open source processor emulator that can be used at the operating system level to run various guest operating systems. It can also be used at the user application software level for emulation of application-level processes. Instrumentation is a process of adding extra lines of code in the original user code at the compile time for the purpose of dynamic code analysis. Such analysis aims to detect bugs in the application coding that are hard to detect otherwise. This project was proposed by Vector Fabrics, the host company, where research and development has been carried out. Vector Fabrics aims at expanding their set of products to address customer needs which cannot be served with their current set of tools. The aim of this project is to investigate if it is feasible to perform code instrumentation during the JIT (Just in Time) compilation phase in QEMU. In addition, does the ARM processor binary code as input provide sufficient information to create traces that allow appropriate error/bug uncovering and reporting to the application programmer. To answer these questions, it was required to study the operational flow of QEMU, as well as to make an experimental implementation of the instrumentation and perform the analysis of the generated trace file efficiency.

The Vector Fabrics tool Pareon is a state-of-the art tool, which is comprised of VF Compiler and VF Analyzer, for the application code analysis and bug finding based on compile time instrumentation. Some of its key competitors are the Valgrind and PIN instrumentation frameworks. Each of these tools have some advantages over the other, but they have some limitations as well. The project was conducted in three phases. In the first phase of the project, ARM instruction set architecture and QEMU source code was explored. In the second phase, instrumentation was performed on some of the selected ARM instructions in the JIT (just in time) compilation inside QEMU. The selection of instructions was based on the analysis requirements. In the third phase, experimental research is carried out for analyzing the feasibility and efficiency of the performed instrumentation.

A prototype is developed for the runtime trace creation based on the compile time instrumentation. The research has verified the hypothesis that the proposed idea of the binary instrumentation with QEMU for dynamic code analysis is feasible. Apart from few problems related to multithreaded programs, most of the questions posed in the aim of the project are answered positively. The trace generated by the instrumentation provides sufficient information to the application programmer for the main error/bug analysis, which proves that the proposed idea is feasible and appropriate for further development.
# Table of Contents

Abstract .......................................................................................................................... 2

Table of Contents .......................................................................................................... 3

List of Figures ............................................................................................................... 5

List of Tables ............................................................................................................... 6

Chapter 1 ................................................................................................................... 7
  1.1 - Introduction ........................................................................................................ 7
  1.2 - Problem Statement .......................................................................................... 11
  1.3 - Aim .................................................................................................................. 12
  1.4 - Objectives ........................................................................................................ 12

Chapter 2 ................................................................................................................... 14
  2.1 - Related Work .................................................................................................. 14
  2.2 - Analysis of Vector Fabrics Development Tool ............................................. 14
  2.3 - PIN (Intel’s Dynamic Binary Instrumentation Framework) ....................... 15
    2.3.1 - Cache Simulation (CMP$IM) ................................................................. 16
    2.3.2 - Instruction Instrumentation ................................................................. 17
  2.4 - Valgrind Binary Instrumentation ................................................................. 17
    2.4.1 - Memcheck ............................................................................................ 18
    2.4.2 - Cachegrind ........................................................................................... 18
    2.4.3 - Callgrind ............................................................................................... 18
    2.4.4 - Helgrind: ............................................................................................... 18
  2.5 – Comparison of Pareon, PIN and Valgrind ................................................... 18

Chapter 3 ................................................................................................................... 20
  3.1 - Research Context and Problem Analysis .................................................... 20
  3.2 - Study of QEMU as a platform for binary instrumentation ....................... 22
    3.2.1 - Function Call Instructions ................................................................. 24
    3.2.2 - Function Return Instructions ............................................................ 24
    3.2.3 - Load/Store Instructions ..................................................................... 24
    3.2.4 - Marking of Specific Functions (Malloc/Free, Pthread_create/Join) .... 25

Chapter 4 ................................................................................................................... 27
  4.1 - Solution Approach ......................................................................................... 27
  4.2 - QEMU Emulator ........................................................................................... 27
  4.3 - Sub-Solutions to Sub-Problems ................................................................... 32
4.3.1 - Function Call and Function Return Instruction Instrumentation .............................................. 32
4.3.2 - Load Store Instruction Instrumentation .................................................................................. 37
4.3.3 - Marking of Specific Functions (Malloc/Free, Pthread_create/Join) ........................................ 40
Chapter 5 ........................................................................................................................................ 45
  5.1 - Experimental Research ........................................................................................................... 45
  5.2 - Experimental Setup ............................................................................................................... 45
    5.2.1 - Function Call and Return Experimental Results ................................................................. 46
    5.2.2 - Load Store Instruction Experimental Results ...................................................................... 49
    5.2.3 - Experimental Result for Marking of Functions .................................................................. 51
  5.3 – Performance Evaluation ......................................................................................................... 56
Chapter 6 ........................................................................................................................................ 61
  6.1 - Conclusion ............................................................................................................................ 61
  6.2 - Future Work .......................................................................................................................... 61
References: ....................................................................................................................................... 65
Appendix (Image Processing Application Used for Performance Evaluation) ............................ 68
List of Figures

Figure 1: Vector Fabrics Systems Operational Overview ............................................. 8
Figure 2: Vector Fabrics State of the ART ................................................................. 14
Figure 3: PIN Instrumentation Tools ...................................................................... 16
Figure 4: Valgrind Infrastructure ........................................................................... 17
Figure 5: QEMU Operational Structure ................................................................ 22
Figure 6: Final Expected System Overview .............................................................. 26
Figure 7: QEMU Operational Overview with added instrumentation .................. 27
Figure 8: QEMU TCG (Tiny Code Generator) Processes ...................................... 28
Figure 9: Front-End and Back-End Output Sample Codes .................................. 29
Figure 10: Sample Test Code .................................................................................. 31
Figure 11: Output without Instrumentation ............................................................. 31
Figure 12: Output with Instrumentation ................................................................. 31
Figure 13: Function Call Instruction (BL) Instrumentation ..................................... 33
Figure 14: Function Call Instruction (BLX) Instrumentation ................................ 33
Figure 15: Body of custom function for Function Call Instruction Instrumentation ..... 34
Figure 16: HELPER function to generate trace messages for function call instructions 34
Figure 17: Prototypes for all HELPER functions ..................................................... 34
Figure 18: Function return instruction (BX) Branch and Exchange instrumentation [B] ... 35
Figure 19: Function return instruction (BXJ) Branch and Exchange to Jazelle instrumentation [B] ... 35
Figure 20: Function return instruction instrumentation through (POP) [B] .......... 36
Figure 21: Function return instruction instrumentation through (LDM) [B] ........... 36
Figure 22: Body of Helper Function for Function Return Trace Message Generation at The runtime [C] ................................................................. 36
Figure 23: Load Store Instruction Instrumentation for Data in the range of (8 to 32) bits [B] 38
Figure 24: Load Store Instruction Instrumentation for Data of 64 bits’ size [B] .......... 39
Figure 25: Helper Function Body for Load Store Instructions [C] ............................ 40
Figure 26: Wrap Function Bodies for marking functions [A] ................................... 42
Figure 27: Code Patch for Identifying content of C14 coprocessor Register [B] ....... 44
Figure 28: Example Test for HELPER function and Return Test .......................... 46
Figure 29: Objdump for user executable to test Function Call and Return Instrumentation ... 47
Figure 30: Trace Results for Function Call and Return Instructions ....................... 48
Figure 31: Program to Test Load Store Instrumentation ...................................... 49
Figure 32: Trace Load Store Instructions ............................................................... 50
Figure 33: Libc Shared Library Opened and Returned Base Address .................... 52
Figure 34: Pthread Library Opened at The runtime .............................................. 52
Figure 35: Libwrap custom shared library trace information ................................... 52
Figure 36: Trace for Marking Function with Base Address Approach .................. 53
Figure 37: Marking of fprintf using base address approach .................................. 54
Figure 38: Source Code for experimentation of Function Marking ....................... 54
Figure 39: Example Wrap Function for Malloc Function Marking [A] ................. 55
Figure 40: Trace message for marking of malloc and free .................................... 56
Figure 41: Performance Measurement with Figure 31 code ........................................ 57
Figure 42: Performance Measurement with Figure 28 Code ........................................ 58
Figure 43: Performance measurement with Image processing application ..................... 58

List of Tables

Table 1: Pareon, PIN, Valgrind Comparison .................................................................. 19
Table 2: Performance Measurement ........................................................................... 59
Chapter 1

1.1 - Introduction

This report discusses the research and development work performed during my MSc graduation project in embedded systems, and the results of this work. With the advent of embedded systems and rapidly increasing system complexity the need for sophisticated design automation tools is increasing rapidly. Researchers and developers are using design and automation tools to make their work effective, efficient and speed up the development process. To get a higher performance multi-threaded programming has become important for software developers. This has resulted in increase in the demand for tools supporting verification of such systems. The main reason for their increasing demands is that the embedded systems are often safety critical systems which need to be error free.

One of the methods of embedded software analysis aiming at finding errors in the application software is a dynamic code analysis through emulation. Vector Fabrics is one of those companies which are developing state-of-the-art methods and tools for dynamic code analysis. They develop design automation tools for C and C++ programmers. The current tool set of Vector Fabrics, are the Pareon tools which are comprised of Vector Fabrics C/C++ Compiler (VFCC) and Vector Fabrics analysis (VFANA) tools. These tools enable dynamic code analysis for multi-threaded programs, while at the same time Pareon tools can perform parallelization of the existing single threaded programs. Currently, Vector Fabrics tools perform the dynamic code analysis with the help of a trace file which is generated at the runtime. The trace file is generated at the runtime during execution of a program which is instrumented at the compile time. An overview of Vector Fabrics system operation can be seen in Figure 1.

The goal of the M.Sc. project described in this report was feasibility and efficiency analysis of the binary code instrumentation with QEMU. Vector Fabrics proposed to investigate the binary code instrumentation for dynamic code analysis in the scope of an M.Sc project.
Instrumentation is a procedure in which additional lines of code are added to an existing code during the compilation phase. This enables the dynamic code analysis, which means that the codes can be analyzed for bug tracing at runtime for bugs that are hard to find at compile time. Instead of instrumentation during compilation of the application source code, in this project binary instrumentation is performed inside translation engine, tiny code generator (TCG), of QEMU. Quick Emulator (QEMU) is an open source emulator which performs the just in time (JIT) compilation of blocks of code for emulation. The key operation of QEMU is to translate instructions of code, aimed to be executed in the target CPU architecture, into host CPU architecture compatible instruction set. For this project target CPU is ARM, whose instructions are emulated using QEMU, while the host platform is based on x86 CPU architecture and experiments are performed when using Linux operating system.

Now, it is important to mention the reason for the binary code analysis, instead of the source code analysis, before explanation of the methods which are used to perform the instrumentation. One of the main reasons is that often the user is not provided with the actual source code when it comes to the use of the object files or libraries in a program. It helps to do dynamic analysis of shared libraries, because developers are mostly equipped with binary code files of shared libraries. Next paragraph explains the need of instrumentation and dynamic code analysis.
When some code is developed usually there are some bugs in it. It is very important that those bugs are found and eliminated before the release of the code. Bugs in a program can have different nature depending upon the kind of mistake made by the programmer while developing the code. Earlier programmers relied only on compilers to detect errors in their programs. To find and fix errors compilers use a technique known as static analysis. For example, if a variable is assigned a value through another variable whose data type is not compatible then the compiler can spawn an error during compilation. But, errors which contain complex memory access at runtime cannot be resolved by compiler. There are a lot of reasons for that; for example, compilers do not have information about the codes of shared libraries which are accessed at runtime. The access to memory locations through some pointer referencing at the runtime can lead to complex memory access problems which cannot be handled at compile time. So, the static analysis, performed by a compiler, has a major deficiency in analyzing a code for complex memory access and data race errors etc. In order to fulfill the deficiency of static analysis tools it is important to have some mechanisms which can detect errors in programs at runtime. One of the mechanisms is known as dynamic code analysis which does the analysis of code instructions at runtime.

Dynamic code analysis is made possible with the help of additional information about the source code apart from the source code itself. The additional information, which is required to perform the binary code analysis, is generated using instrumentation in the original source code. This instrumentation leads to the generation of a trace file, at runtime, which contains the additional information about the application behavior. For example, if it is needed to analyze which memory locations were accessed at the runtime to load and store values in memory then information about the load and store instructions will be recorded in the trace file. This additional information is used by the analysis software to analyze for well-behaved memory access and to suggest where an error may occur. The analysis software needs additional information in the form of the CPU architecture type of the instructions, the available memory and the kind of instruction set used in generating the binary executable. With all of this information given in a trace file analysis tool can trace complex errors and report them to user.

Now, the reason for the selection of QEMU to perform the instrumentation for binary code is such that QEMU performs the emulation of the binary code, but not of the source code. As explained before, QEMU has a binary code translation engine which does the just in time binary translation. Just in time (JIT) is a process in which blocks of code are fetched from a binary executable and translated from one type of binary instructions to another type of binary instructions and then executed once the whole block is translated [3]. For this project we will
use QEMU to translate ARM binary application code to an X86-based machine, so that the developer can test ARM applications on his desktop development computer. Therefore, it is very suitable to be used for the purpose of this project. QEMU translation engine already contains decoding of almost all the required instructions to be instrumented, but the decoding of the instructions is not distinguished explicitly for the type of instructions selected for instrumentation in this project.

To achieve the goal of this project, the work to be performed has been decomposed into several activities, including the following:

- To learn QEMU environment and the source code of TCG as it is the main engine for the binary code translation.
- To understand the mechanism of the instrumentation for the binary instructions of the ARM instruction set architecture.
- To understand the ARM instruction set architecture itself, specifically targeting those instructions which were selected for instrumentation.
- To propose the needed instrumentation and emulation techniques.
- To perform the binary instrumentation within the QEMU translation engine, when using the proposed technique.
- Implement emulation to generate the trace file, containing sufficient information to allow application error analysis.
- To perform experiments to verify the feasibility and efficiency of the instrumentation.
- To record performance measurement, with respect to the elapsed time of the application, by doing comparison between instrumented code and code without instrumentation. In practice, the slowdown of the code due to instrumentation and emulation inevitably represents a severe issue.

The following are the main initial assumptions of the project:

- Instrumentation of the Binary code will be performed instead of the higher level C code.
- Instrumentation will be performed in the tiny code generator section of QEMU
- QEMU will be used in user application mode, as opposed to its operating systems mode.
- This Master project will be performed on a x86 based Linux platform
- To support the explorative nature of this project, the trace file shall contain sufficient information for error analysis, but does not need to exactly fit into the current VF analysis tool.
In this report a thorough explanation is given of the above mentioned goals and activities of the project and their realization while meeting all the requirements. Also, a comparative analysis of some competitive tools with Pareon tools is made to show the difference in operational requirements and other features presented. The following section will describe the problem statement in more detail.

1.2 - Problem Statement

Before explaining the problem, it is important to mention that the QEMU system is studied in the context of Vector Fabrics Pareon tools which represents the current state-of-the-art. As mentioned in the introductory section many application development tools require target embedded hardware to be attached to the host development platform at the runtime for debugging critical errors in the code. The same is the case with the VFCC. The indulgence of target embedded hardware, for example the ARM based Android or Linux embedded systems, during development often increases the development time. One of the reasons is that often it takes a significant time to upload the application to the target embedded hardware before running it. Moreover, for profiling the code additional trace files are generated at the runtime. Since the trace file is generated at the runtime at the target hardware itself, an overhead is involved to transfer the contents of the trace file back to the host development platform for the analysis and error message generation.

This whole process of data transmission between the two platforms using a socket or wired medium slows down the actual process of development and requires additional setup in doing the development of the projects. For instance, the Vector Fabrics compiler performs the instrumentation in the original source code, which as a result generates a trace file at the runtime. The trace file is subsequently fed to the VFANA which is another Vector Fabrics tool for analyzing the runtime code behavior. These trace files often cannot be stored on small embedded target platforms due to their memory limitations. For example, the 32-bit ARM evaluation kits, like STM32F100, contain very small on chip flash or EEPROM memories to store the code and data; hence, in this case it becomes extremely difficult for the target embedded hardware to support the generation of such trace files during the runtime.

If Vector Fabrics wants to expand its Pareon solution to also cover small embedded platforms, then a new approach must address the following issues:

- Extracting the trace files out of small embedded platforms; for example, evaluation kits containing 32bit ARM Cortex processors.
• Limiting the code and data memory size overhead of the run-time instrumentation library on the small target hardware.

• Limiting the speed penalty of the run-time trace generation.

Due to the above mentioned issues Vector Fabrics wants to investigate an approach to address these issues around small targets, so that it can expand its customer base in that domain. Pareon is explained in more detail in the analysis of the state-of-the-art section of related work of the report. Application emulation on the developer desktop machine is a promising approach to address above issues. QEMU currently is a popular and widely adopted emulation system [8], and its open-source nature allows for the required instrumentation modifications.

1.3 - Aim
The main aim of the Master project described in this report was to investigate if cross platform emulation and instrumentation techniques are feasible for the analysis of the cross platform emulated code. Moreover, if it is feasible to perform the code instrumentation during the JIT (Just in Time) compilation phase in QEMU. In addition, in the case of the ARM, processor to be emulated, if the binary input provides sufficient information to create traces that allow the error/bug uncovering and reporting of such information to the application programmer. Realization of this aim required analysis of the QEMU source code, instrumentation feasibility and the ARM instruction set architecture.

1.4 - Objectives
The more specific objectives of this master project, with reference to the aim described above, are summarized below:

• Perform the problem analysis and comparative analysis of different binary instrumentation techniques.

• Perform instrumentation of the ARM Binary code in QEMU

• Develop and implement identification of Function Entry / Exit Points by Instrumentation

• Develop and implement identification of Addresses of Load Store Instructions by Instrumentation

• Develop and implement marking of certain functions and generation of trace messages
• Demonstrate the feasibility and efficiency of the proposed approach, and techniques through performing the experimental research when using a prototype implementation of the technique.
Chapter 2

2.1 - Related Work

In this section the study is reported that was performed to understand and compare different frameworks related to binary instrumentation and dynamic code analysis. The analysis of the state-of-the-art is performed with respect to Vector Fabrics Pareon which is comprised of a compiler and trace analysis tool. For comparison purposes two other major instrumentation tools will be discussed. These tools are:
1 – PIN Dynamic Binary Instrumentation Tool (Intel)
2 – Valgrind (Cambridge)
Each of these tools is explained below.

2.2 - Analysis of Vector Fabrics Development Tool

Vector Fabrics Pareon Verify consists of two parts. One of the parts is the VF compiler (VFCC) and the other part is the VF analyzer (VFANA). Compiler does the code compilation and instrumentation of the source code, while the analysis tool analyzes the trace file generated at the runtime. This trace file is generated by executing the application which is instrumented by the Vector Fabrics tools at compile time. Currently, the VF Analyzer tool supports trace analysis for x86 and ARM 32-bit and 64-bit CPU architectures in an operating system environment of Linux or Android. VF Compiler has a separate trace instrumentation library which is integrated into the compiler to perform instrumentation at compile time.

A block diagram of the Vector Fabrics tools operation is shown in Figure 2.

From Figure 2 it is clear that there are two distinct sections of the development tool which are VF Compiler and VF Analyzer. The instrumentation takes place after the generation of the
intermediate representation of the assembly instructions during compilation of the source code. The trace library reads IR instructions and generates tokens which are then used to generate executable file which contains additional tokens to generate trace messages in the trace file at the runtime. This instrumented binary final is run over a target embedded platform which generates the trace file of the code during the runtime. This trace file is then fed to the VFANA (VF analyzer) which generates the (error) messages for the user.

VF Analyzer is a tool which is quite resource intensive, which means that it requires a lot of CPU computation power. Hence it supports only a limited set of larger platforms. Currently, it can only be run on x86 based Linux platforms. On the other hand, the code, through VFCC (VF Compiler), can be compiled in cross-compilation mode to run on a large number of CPU and Operating system platforms. For example, these are ARM32 and ARM64 bit Android and Linux platforms. In addition, it can also be run over x86 based 32 and 64 bit Linux operating systems. It can support some of the small embedded targets as well hosting ARM based hardware, for example the code compiled through VFCC can also run over Raspberry pi and Beagle-bone Black, which are relatively small target platforms. The executable runs on the target hardware platform and the trace file which is generated at the runtime is sent to the x86 Linux based desktop via a wire or socket where VFANA is running. VFANA then generates error messages for the user based on the trace file information. The Pareon tools are in use at Vector Fabrics’ customers, and successfully analyze C and C++ programs up to 10M lines of source code, such as the Android Chrome web browser.

2.3 - PIN (Intel’s Dynamic Binary Instrumentation Framework)

PIN is a binary instrumentation framework for dynamic code analysis by Intel. It currently supports IA-32, Intel64 and IA-64 architectures with Linux, Windows, and MAC-OS platforms. PIN can be used to create further instrumentation tools which are called Pin-tools. It can instrument all sorts of applications, databases and web browsers. It gives the facility to write programs in C, C++ and Assembly languages. It can also instrument multi-threaded programs. The concept of dynamic binary instrumentation refers to the mechanism in which instrumentation of the code is performed just before execution through just in time compilation. [1]

PIN provides APIs to write your own custom functions; and these custom functions can be used to replace the user application functions. For example, if user wants to trace the performance of shared library functions like “malloc” or “free” then he can make use of PIN’s API to write his own “malloc” which will be called before calling the actual “malloc” function.
A similar mechanism will be used in this Master project to do a particular task for marking of functions which will be explained in later sections of this thesis. There are a lot of similarities in functionalities between Vector Fabrics Tools and PIN, but the main difference between two of them is that Vector Fabrics Tools support code instrumentation for ARM target architecture applications, but Intel's PIN supports x86 target architecture applications only. In addition, PIN provides developers the opportunity to add instrumentation functions or even create instrumentation tools while vector fabrics tools internally do all the instrumentation at compile time which then generates trace files at the runtime. PIN is an instrumentation framework which gives opportunity to create even more new instrumentation tools with support for x86 while Pareon tools are very different in this regard because Pareon tools don't provide any library to create new instrumentation tools, rather everything is done internally within the compiler. [1]

Currently, at Intel PIN is used to create profiling and analysis tools for the C language programs. In addition, it is used to analyze the new instructions which are developed for x86 architecture. The examples of PIN tools are given in Figure 3.

![Figure 3: PIN Instrumentation Tools](attachment:image.png)

An introduction to the few PIN Tools is given in subsequent sections.

2.3.1 - Cache Simulation (CMP$IM)

In order to simulate the cache behavior, PIN captures the type of memory operation, the address of the memory, memory operation size and memory reference thread id. The application runs on top of PIN while PIN fetches and provides the memory instruction
information to CMP$IM for cache behavior simulation. The memory information is
dynamically sent to the model of the cache, which is to be observed, defined by the user. The
user defines the size of the cache, associativity, and the replacement policy of the data when
cache lines are filled. [5]

2.3.2 - Instruction Instrumentation

PIN also supports instruction instrumentation. In instruction instrumentation each of
the instructions are instrumented one by one. This is similar to trace instrumentation tool of the
PIN in which each trace begins from the target of a jump taken and ends at the unconditional
branch. PIN divides the traces in the form of basic blocks. Normally each of the basic blocks
are single entry single exit sequences of instructions, but if there is some jump inside a basic
block a new basic block is formed with a new trace in the result of the jump. [4]

2.4 - Valgrind Binary Instrumentation

Just like PIN, Valgrind is another binary instrumentation and dynamic code analysis
framework. It was developed in 2006 at Cambridge by Julian Seward. Valgrind is used to find
errors in application programs. It is used to do the code analysis, find bugs and memory leaks
in the programs. In the default mode it analyses for memory related errors. There are several
more tools associated with Valgrind as plug-in extensions; few of them are introduced below.
In addition, Valgrind lets its users define their own instrumentation tools as well. The problem
with Valgrind is that it gives an overhead of almost 5x-10x even without doing any
instrumentation. Valgrind infrastructure is given in Figure 4.

Valgrind can also instrument running programs; for example, web browsers and database
servers. In addition, it can also instrument self-modifying code. [2]
Valgrind has a modular architecture; hence it is easier to create new modules without disrupting the whole system. Since it is a framework to create instrumentation tools, hence it provides a standard set of tools as well. A few of them are mentioned below:

2.4.1 - Memcheck: Memcheck is a tool to find memory errors. For example, it can find uninitialized dangerous variables which can cause problems in the program. In addition, it also finds dangerous variables which are derived from uninitialized variables. In this way it helps to make the programs, written particularly in C and C++, more correct.

2.4.2 - Cachegrind: Cachegrind is very helpful in making the program more robust. It simulates the different levels of caches of a system for the application and makes the program run faster based on the results of the application's interaction with the simulated caches. Right now, it can simulate three levels of cache which matches most of the modern machines, but for the machines having four levels of cache it simulates only first and last level of cache as the last level of cache masks access to main memory.

2.4.3 - Callgrind: Callgrind does the profiling of the functions called inside functions. It keeps the history of those calls inside a file which can be read using certain type of commands. The history of function calls is related to the source code lines. A GUI representation of those call graphs to functions can also be received which is QT based. QT is a tool which is used to create User Interface using C++.

Similarly, there are more tools available to make the programs robust and error free. These tools are extensively used by developers to make their programs correct during development phase.

2.4.4 - Helgrind: Helgrind is a plug-in that detects and reports data race errors in multi-threaded programs. It is aware of the Pthread API semantics and tracks its effects as accurately as it can.

A comparison between all three of the above mentioned tools is given in Table 1: Pareon, PIN, Valgrind Comparison).

2.5 - Comparison of Pareon, PIN and Valgrind
A comparison between PIN, Pareon and Valgrind is shown in Table 1: Pareon, PIN, Valgrind Comparison).
Table 1: Pareon, PIN, Valgrind Comparison

<table>
<thead>
<tr>
<th>Tool</th>
<th>Type of the Tool</th>
<th>User Customization</th>
<th>Operation</th>
<th>Supported Platforms</th>
<th>Cross Platform Emulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pareon</td>
<td>Compiler and Analyzer</td>
<td>Not Possible</td>
<td>Dynamic Code Analysis</td>
<td>(x86, ARM) (Linux, Android)</td>
<td>NO</td>
</tr>
<tr>
<td>PIN</td>
<td>Instrumentation Framework</td>
<td>Yes</td>
<td>Custom code Instrumentation and Dynamic Code Analysis</td>
<td>x86 (Linux, Windows, MAC)</td>
<td>NO</td>
</tr>
<tr>
<td>Valgrind</td>
<td>Instrumentation Framework</td>
<td>No</td>
<td>Custom code Instrumentation and Dynamic Code Analysis</td>
<td>(X86, AMD, ARM, PPC) (Linux, Android, Solaris)</td>
<td>NO</td>
</tr>
</tbody>
</table>

The above given table shows a comparison between different instrumentation and dynamic code analysis tools. It is clear from Table 1: Pareon, PIN, Valgrind Comparison) that Pareon is the only tool out of the three which does not support custom code instrumentation, but everything is rather performed by the compiler itself. On the other hand, PIN and Valgrind provide this feature to the developers. In addition, Valgrind provides support for a lot of CPU architectures and operating systems, then comes Pareon and the least support is provided by PIN, which only supports x86 based platforms. It is important to mention that none of the tools support cross platform emulation. However, Pareon is the only tool that provides cross-platform compilation and off-target analysis: PIN and Valgrind must do the expensive analysis on the target itself, whereas Pareon allows to offload the analysis to a development host. Due to the high compute requirements of the targeted memory and thread-based errors, offloading the analysis to another host is a major advantage.
Chapter 3

3.1 - Research Context and Problem Analysis

This MSc project was performed at Vector Fabrics in collaboration with the Electronic Systems group of Technical University of Eindhoven. This chapter describes research context of this project. Vector Fabrics develops and serves the industry with tools for the multi-threaded applications development. Their tools currently support Linux and Android operating systems for ARM and x86 CPU architectures. Their current platforms run with a considerable overhead and within limitation of embedded hardware dependency; for example, as mentioned before an embedded hardware, like an android hand held device, is needed to be attached to the development platform at the time of running the program. Vector Fabrics proposed this M.Sc. project to investigate a new method of dynamic code analysis, which is to be used later in collaboration with their current setup. For this purpose, a new approach of instrumenting the binary code had to be proposed to make the dynamic binary code analysis possible.

Instrumentation is the mechanism of adding extra lines of code at the compile time to cause the generation of a trace file at the runtime. That trace file, which has to be transferred to the desktop system or a larger server computer, is analyzed by VFANA. The bandwidth for the data transfer between the target embedded hardware and host desktop is relatively low, hence the bandwidth is the bottleneck caused by the target embedded hardware. For example, if one out of four instructions is a load/store instruction, for which a trace message is to be generated, and each of the trace messages is of 6 bytes then the required bandwidth for the 1GHz clock of target platform, is given as:

\[
\text{Trace Bandwidth} = 1\text{GHz} \times \frac{1}{4} \times 6 \text{ bytes} = 1.5\text{GB/s}
\]

Achieving this high bandwidth for the data transfer over the network using USB cables or even socket connections is not possible. In consequence it is one of the major bottlenecks of the system.

However, the purpose of Vector Fabrics to propose this research was not to replace their current pool of product with a totally new invention. But, rather to enrich their pool of products by including additional tools for the binary code instrumentation and analysis in their system. Vector Fabrics current tools basically do the user source code analysis while instrumenting the source code at the compile time. In the current implementation of Vector
Fabrics tools the Assembly IR (Intermediate representation) is instrumented to generate the trace file at the run time.

For this Master project it is assumed that the instrumentation of the binary code will be performed instead of the higher level C code. A detailed description of QEMU with a sample implementation of the idea will be presented in the next chapter.

One of the problems which will be analyzed through this research is the binary code analysis instead of the high level user code analysis. Sometimes it happens that user has no access to the source code of shared libraries, but rather only to the binary files of libraries. In this case the binary code analysis can reveal a lot of information regarding the shared libraries as well. In this way, the third party binary code files can also be analyzed to find bugs in the programs. So, potentially, a QEMU based setup could address two important issues:

1. Support the analysis for errors also where binary libraries are part of the application
2. Transfer of the trace-file from the emulated system to the analysis host inside the same computer so that plenty of bandwidth is available.

The problem of analyzing the binary code is not very simple. It is much more difficult than simple source code syntax which can be semantically and syntactically analyzed to identify the instruction type. There are numerous kinds of encoding and decoding schemes related to the instructions of ARM architecture, and all of them need study before deciding upon their instrumentation.

In order to make this research possible within the scope of this Master project it is decided to instrument only certain most important types of instructions and the research was dedicated accordingly. The main objective of the dedicated instruction analysis and instrumentation was to discover if and how binary instrumentation can efficiently be performed by the QEMU JIT compilation system. In addition, will the proposed binary instrumentation be useful and sufficient for the dynamic code analysis, to support, with accurate error messages to the developer

Following are the major partial problems which had to be solved:

- Instrumentation of the Function Call and Return Instructions
- Instrumentation of the Load Store Instructions
- Instrumentation for the Marking of certain types of functions
There are distinct reasons behind selection of the above mentioned instruction types for the analysis. These reasons are explained in the subsequent sections, but first QEMU architecture and work are more precisely explained below.

3.2 - Study of QEMU as a platform for binary instrumentation

The proposal of binary instrumentation brought a question of how the binary code itself can be instrumented. There are two main methods to do that; either by decoding of each instruction out of the binary files one by one and then instrumenting the instructions or by using an emulator which already does the work of instruction decoding. The first option would lead to the development of an emulator itself which is certainly out of the scope of this Master project. Therefore, the second approach was used in this project. For the purpose of the binary instrumentation QEMU is used. QEMU already has established itself as a highly popular emulation system. There is a clear structure of how QEMU works which is shown in Figure 5.

The above picture shows the overall program flow to emulate a binary application built for ARM processor.

The application is to be emulated on an x86 desktop machine. One of the major reasons behind the study of QEMU was to understand the possibility to instrument the code in such a way that the instrumentation would work to generate trace file at the runtime rather than compile time.

There are different types of instructions which are decoded within QEMU to enable emulation of ARM instruction on top of x86 based machine. The study of each type of instruction was
important to cover the complete range of the selected instructions within the domain of each type of instruction set architecture.

For example, the following types of instruction sets are available on an ARM microprocessor:

1. ARM Instruction Set
2. THUMB Instruction Set
3. THUMB2 Instruction Set
4. NEON Instruction Set
5. iWMMXT Instruction Set
6. ARM Coprocessor Instructions

The above mentioned instruction sets have their own special functions as well as generic implementations on the top of the ARM microprocessor. For example; ARM, THUMB and THUMB2 are generic instruction set architectures used in ARM. But, NEON, iWMMXT and Co-processor instructions have special functions associated with them. NEON is a special kind of instruction set designed to support graphics processing more efficiently. NEON instructions are basically vector instructions and these instructions operate on contiguous memory locations or multiple registers at the same time to make graphics processing faster. Similarly, iWMMXT instructions are special instructions to support wireless multimedia instruction processing. Finally, co-processor instructions are used to support writing and reading of the co-processor registers.

The implementation of these instruction set architectures is also dependent upon the processor version, as well as, the type of operations specified within the program itself; but for efficiency and complete analysis of the code, all of the instruction sets are needed to be analyzed for dedicated instruction types which are to be instrumented.

Following is the set of instruction types which were selected for the instrumentation:

- Function Call Instructions
- Function Return Instructions
- Load/Store Instructions
- Marking of Specific Functions (Malloc/Free, Pthread_create/Join)

The above mentioned instructions are the most commonly used instructions in the high level languages like C and C++. They are selected for instrumentation because they are key for the analysis and reporting of memory and data-race application errors.
3.2.1 - Function Call Instructions

In ARM architecture there are special instructions to call a subroutine which are known as Function call instructions. A subroutine is a patch of code which is either compiled separately, which could be part of the shared library, or within the same C source code. But, a subroutine is not part of instructions within the main block of a code. A subroutine can be called at the runtime from anywhere within its scope in the overall program. A scope of a function or subroutine is the boundary in a program within which a function can be called by any other function or main part of the program. A function can be called by writing its prototype while providing proper argument types. An example of a function call is given below:

\[ \text{function_test (int time, int frequency);} \]

\[ \text{function_test (int time, int frequency);} \] is a proper syntax to call a function named “function_test” while time and frequency are its arguments where “int” is the type of those arguments. An application written in C language normally contains hundreds of such functions which are called from different places within the program.

3.2.2 - Function Return Instructions

Function return instructions are one of the most critical instruction types because there is no fixed place in a function from where the function returns to its caller. Apart from uncertainty in returning from a fixed location in a function another uncertainty is in choosing the type binary instruction to return from the function. The normal syntax to return from a function is given below:

\[ \text{Return sum;} \]

“Sum” in the above instruction syntax can be a variable or pointer type as well. There can be many different types of return statements; for example, it can include simple variable return types, pointer of a data type and then void type returns as well. In void type return statements none of the variables or integer values is returned.

The tracing of function call and return statements allows that if any errors are found in an application run, these errors can be reported as occurring in a detected call-stack. Such information will help the developer to derive the cause of the error.

3.2.3 - Load/Store Instructions

Load and Store Instructions represent the second type of the most widely used instructions in a program after the use of normal “MOV” instructions. A load or store instruction loads data from memory or stores it to memory respectively. Load and store instructions are extremely important to instrument because these instructions access memory instead of the registers only.
With the information of load and stores the analysis tool can identify if the program is accessing memory within the boundary of the program or out of bound memory location which creates bugs and uncertain problems in the programs at the runtime. Normally a load or store instruction comes into play when global data is accessed in a high level language like C or C++.

3.2.4 - Marking of Specific Functions (Malloc/Free, Pthread_create/Join)
Since function calls are already one of the target instructions to be instrumented a question would arise why certain functions like malloc, free, pthread_create and pthread_join are needed to be marked. One of the objectives of marking is that there are certain functions with their unique importance which are very critical to analyze the behavior of the program and find bugs in it. For example, malloc is used to allocate certain number of bytes in memory. Similarly, marking of function “free” will give information about the beginning address in memory from which it is to be wiped out. Analysis of malloc is important because it gives us basic information about the memory bound to certain data sets and if there is an access to memory out of its bounds then it is a runtime error.

Similarly, in multi-threaded applications it is very important to know when a certain thread is spawned and joined in the process. This helps to understand the completion of tasks under the action of certain threads which are recorded. This helps to get the record of total number of threads created at the runtime. In addition, it helps to know how many threads did not join while creating very large memory leaks.

Therefore, for these special functions, we need tracing of some of their run-time argument or return values.

The solution approach to these problems is an important factor in the overall performance of the system. The final system overview will look like Figure 6.
There are multiple approaches to come up with the solutions which are described in the next chapter.
Chapter 4

4.1 - Solution Approach

This section discusses different solution approaches considered for each of the sub problems of the binary instrumentation in QEMU. Before discussing the solution approach to each of the sub problems, it is important to introduce an overall introduction to QEMU and its operation.

4.2 - QEMU Emulator

QEMU is an open source emulator and virtualizer. QEMU can run user programs as well as operating systems. It can run programs, written for one type of CPU architectures, on the top of different host CPU architecture. For example, if a program is written for the ARM processor, QEMU can run it on the top of the x86 based host platform. It can run one type of operating system on the top of another operating system of different type. For example, it can run Linux on the top of Windows. The overview of the QEMU in Figure 7.

![Figure 7: QEMU Operational Overview with added instrumentation](image-url)
Figure 7 shows QEMU operational flow in a broader view with added instrumentation which will be explained later. In the user mode, which is the mode in which a simple user application runs through QEMU instead of a whole operating system, the program has to be first compiled using an appropriate compiler. In Figure 7 it is shown that the application source code is first compiled through ARM-Linux-GCC compiler. Once compiled it produces a binary file which is specific to the ARM architecture.

That binary file is fed to the QEMU which does the just in time (JIT) compilation of the target binary. The instructions of this binary file are converted in JIT fashion to the instructions of the host CPU architecture. After a single block of instructions is translated it is stored inside a Translation Block cache, which is 16MB, and it is run over the host platform. Currently QEMU does not support translation of floating point operations; hence these are performed using the host CPU instructions in a special way which will be explained in following paragraphs. A single block of binary instructions is finished when a conditional jump takes place or when a page boundary is reached in a way that the next instruction starts at the beginning of the next page. [3] The translation of a block can also be terminated due to reasons like exception generating instructions and hardware watch points.

The just in time compilation mechanism works in such a way that when the application execution makes a jump into a new block of code, the mechanism looks in the translation cache if the block was already translated or not. If there is already translation available to that block it is executed by fetching from the cache otherwise the new block of code is translated and stored in the cache. Once the cache is filled up the cache is flushed totally for simplicity. There are two main parts of the QEMU translation engine. One of the parts is known as front-end and the other is back-end. The abstract view of the translation engine (TCG) which is tiny code generator is shown in Figure 8.
The above picture shows two distinct sections of TCG which are front-end and back-end. The front-end converts the target binary into its intermediate representation which is hardware independent. Micro-ops are the intermediate representation which are then converted to native binary instructions by the back-end process.

Figure 9 shows the results of backend and front end processes. The letter “C” is displayed to show correlation of the marked call instruction with the block in QEMU overall system overview. The Micro-ops are generated in the result of frontend processing on the target binary file, while the backend produces respective native binary code as shown on the left side of Figure 9. The frontend functions of TCG are responsible for the Just in Time compilation of the user binary code into Micro-ops, while TCG backend does the same Just in Time compilation process on the Micro-ops to convert them to the host specific binary instructions. The custom helper functions which are used to generate the trace messages at the runtime are part of the TCG frontend; however, they are converted into Micro-ops, as shown in Figure 8 in the form of call instruction to function load_test_trace, to be called from the host specific binary instructions.
For each of the target CPU architectures to be emulated there is a separate translation file corresponding to the target CPU architecture. Similarly, for each of the host CPU architectures there is a dedicated Micro-op to the host compatible binary instruction conversion file. So, basically it’s a two steps process. First step is related to conversion of target binary instruction to intermediate representation (Micro-ops) and second step is to convert those Micro-ops to the host compatible binary instructions. There are a lot of instructions which cannot be simply converted to micro-ops because of their complex operations. For example, there are NEON instructions which cannot be directly converted into micro-ops. These are complex instructions specially introduced to process graphical applications. NEON instruction set is more like an additional instruction set architecture incorporated in original instruction set architecture of ARM. NEON instruction set architecture is complex and it is executed by using the host CPU architecture specific code written in C language.

In order to execute these instructions a special set of functions is called within the program. These functions are called Helper functions. The call to the helper functions is converted into the corresponding micro-op calls. These helper functions are then called at the runtime and every time program executes these helper functions are executed at the appropriate points. For each of the instruction set architecture type there is a separate disassembly function in the translate file. Each of the binary instructions is translated into multiple micro-ops. There is a predefined set of micro-ops which are the same for all sorts of CPU architectures. QEMU also supports many different hardware simulations; for example, network and sound devices. In addition, the support is constantly increasing, and in future QEMU for parallel programs will be released. In this Master project the front end part of QEMU is used to perform the instrumentation, as it gives the clear access to each of the instruction types of the ARM processor. The problem with the backend is such that a lot of information is lost if the instrumentation is tried at the back end of the process. Below execution of an example will be discussed, after doing a simple instrumentation in the front end part of QEMU. The results show both types of output; one produced before the instrumentation and another one which is produced after the instrumentation.
```c
// preprocessor_token_pasting.cpp
#include <stdio.h>

int functiontest(int);

int main(void)
{
    int i, sum;
    sum = 0;
    for (i=0; i<30; i++)
        sum += functiontest(i);
    i--;
    i++;
    return (sum == 0);
}

int functiontest(int i)
{
    printf("JTest: call %d\n", i);
    return (int)((float)i / 0.33f);
}
```

Figure 10: Sample Test Code

Figure 11: Output without Instrumentation

Figure 12: Output with Instrumentation
The example code calls a function named “functiontest” 30 times from the main function of the program. The program does some floating point calculations, conversions and returns the result while printing the function call number. The first output in Figure 11 only shows the output of the program while it is executed through QEMU where there is no instrumentation performed to get the function entry address. In the second figure, Figure 12, additional messages are created which is result of the instrumentation. At the top of the window the first address is 0x00008498, which is the entry point address of the function functiontest with additional messages containing one of the shared library function addresses which will be explained later.

After the instrumentation in QEMU has been performed, it found the address, which was specified manually for testing purpose and prints the additional lines in the output as can be seen in Figure 12. The application code calls functiontest 30 times and instrumentation also shows same number of calls in the print message. The first line of the output in Figure 12, where count is 0, shows the address of the function which is called 30 times. It is first translated, and once it is translated, then it is executed from the cache where the translated block is stored.

4.3 - Sub-Solutions to Sub-Problems

This section will describe solution approaches to each of the sub problems analyzed in the previous chapter.

4.3.1 - Function Call and Function Return Instruction Instrumentation

For instrumentation of function call and return instructions there were not many choices except, for the choices related to, following the decoding of each of the instruction types for calling a subroutine and returning from that subroutine. There are multiple instruction types for calling a subroutine for ARM, THUMB and THUMB2 instruction set architectures. They are similar in their Assembly mnemonics, but differ in encoding of the instructions.

Whenever a subroutine is to be called a special type of “Branch and Link” instruction is used in all of the instruction set architectures. There are two variants of Branch and Link instructions; one of the variants is simple “BL” instruction which means Branch and Link and the other variant is “BLX” which means Branch and Link with Exchange. The second variant is used to switch between ARM to THUMB instruction set and vice versa after calling the function. The decoding for such instructions is shown below:
Figure 13: Function Call Instruction (BL) Instrumentation

Figure 13 shows decoding for the THUMB2 type *Branch and Link* Instruction which, after calling the function, exchanges to ARM instruction set. The letter “B” shows correlation of instrumentation code segments with block of QEMU system overview, denoting code that is executed during JIT compilation. In future pictures all of the instrumentation code segments can be correlated similarly. The decoding shown within the “if” statement is to differentiate between normal *Branch* instruction and *Branch and Link* instruction. The decoding of the “*Branch and Link with Exchange*” instruction, BLX, is shown in Figure 14 for calling a subroutine.

This instruction calls the subroutine and exchanges the execution to be continued within the ARM instruction set architecture domain. The decoding within “if” statement is to make sure that correct instruction is decoded for function calling.

Similarly, BL and BLX instructions for ARM and THUMB instruction set architectures are decoded and instrumented. The statement “vftrace_func_call (offset_trace, __LINE__);” is used to call a custom function which takes the arguments “offset_trace” which is address of function to be called, and “__LINE__” which is the line number within the translation engine for debugging purposes. The body of the function is shown in Figure 15.
Within this custom function, which is described above, Helper function is invoked by passing the same arguments which are given to the custom functions. The helper functions generate the trace messages at the runtime. The arguments of the helper functions should be in the form of TCGv_i32 or TCGv_i64 types which are "typedef" variables used within the translation engine of the QEMU. This is the reason why the "line_number" argument is converted into its TCG variant before passing to helper function. The body of the helper function for the generation of trace messages for function call stack is shown in Figure 16.

```c
void uTrace_func_call(TCGv_i32 function_call, uint32_t line_number)
{
    TCGv_i32 line_no = tcg_const_i32(line_number);
    #if ENABLE_CALL
        gem_helper_func_call(function_call, line_no); // Call to Helper Function call
    #endif
    tcg_temp_free_i32(line_no);
}
```

Figure 15: Body of custom function for Function Call Instruction Instrumentation

The letter “C” is to show correlation of helper code segments with “C” block of QEMU system overview, and represents code that is invoked every time that a translated block of code is executed. Rest of the other similar code segments in this report can also be correlated subsequently. The prototype of the helper functions which is defined in separate HEADER file within QEMU which is shown in Figure 17.

```c
/* .... Helper Instrumentation Function Prototype Declaration */
void HELPER(func_call)(uint32_t junval, uint32_t line_number) {ponce;
    fprintf(junaid_trace,"func_call: 0x%08x @ line:%d\n",junval&0xffffffffc,line_number);
    fflush(junaid_trace);
}
```

Figure 16: HELPER function to generate trace messages for function call instructions

The HELPER function prototypes shown in Figure 17 are used throughout the whole QEMU during the whole project to generate trace messages at the runtime. The very first
prototype for the helper function is used for generating trace messages for function call. Similarly, subsequent prototypes are used for generating trace messages for some of the other instruction types which will be explained in the following sections.

Function call instruction is fairly simple to be instrumented as there are dedicated assembly instructions to call a subroutine, but for function return instructions there is a lot of ambiguity. There are multiple ways to return from a function using many different kinds of instructions. According to the ARM manual there are two distinct instruction types to return from functions which are “BX LR” and “POP” instructions. The first instruction type, “BX LR”, means Branch and exchange to ARM or THUMB instruction type. LR in this instruction is the R14 (Link Register) register which always contains the return address. Another variant of the BX is BXJ which exchanges to the Jazelle state after doing the branch to the given address in LR. Figure 18 shows the decoding of LR register for selection between BX instruction for returning from subroutine and normal BX instruction respectively.

```
/* branch/exchange thumb (bx). */
ARCH(4T);
tmp = load_reg(s, rm);
//.... Vftrace Function Return Instrumentation
if(rm==14):
    vftrace_func_return(tmp, __LINE__, insn);
//.... End of Vftrace
```

Figure 18: Function return instruction (BX) Branch and Exchange instrumentation [B]

```
ARCH(5J); /* bxj */
/* Trivial implementation equivalent to bx. */
tmp = load_reg(s, rm);
//.... Vftrace Function Return Instrumentation
if(rm==14):
    vftrace_func_return(tmp, __LINE__, insn);
//.... End of Vftrace
```

Figure 19: Function return instruction (BXJ) Branch and Exchange to Jazelle instrumentation [B]

It is clear in the Figure 18 and Figure 19 that in the “if” condition “Rm” register is decoded and checked for a value of 14 which means that Rm should correspond to R14 Link Register otherwise the instrumentation will not be done as R14 is the only register which contains the return address for BX and BXJ instructions. Once it is verified that Rm is R14 “vftrace_func_return” function is called which contains the call to helper function for generating the trace messages for function returns just like it is done previously with “Function Call instruction” instrumentation.
The prototype definitions in Figure 17 contain a certain number associated to their definition; for example, DEF_HELPER2 contains number “2” which suggests that this helper function expects two arguments in its prototype declaration.

In addition to “BX” and “BXJ”, another popular method to return from a function is by using POP instruction or by moving the contents of LR register into PC (Program counter) using MOV instruction. The method to return using “MOV PC, LR” instruction is almost obsolete now and POP is a frequently used instruction. POP instruction works similarly to LDM instruction which loads multiple registers from stack including PC (program counter). POP is always used whenever a function pushes its arguments or registers on the stack before calling a nested subroutine. In order to get the maximum number of return instructions instrumented both, POP and LDM, type instructions are instrumented. The instrumented codes are given in Figure 20 and Figure 21 respectively.

```c
/* pop pc */
tmp = tcg_temp_new_132();
gen_aas2_ld32u(tmp, addr, get_mem_index(s));
//..... Vftrace Function Trace Instrumentation
   vftrace_func_return(tmp, __LINE__, insn);
//..... End of Vftrace
```

Figure 20: Function return instruction instrumentation through (POP) [B]

```c
//..... Vftrace Function Return Instrumentation
   if (i == 15)
   {
      vftrace_func_return(tmp, __LINE__, insn);
   }
//..... End of Vftrace
```

Figure 21: Function return instruction instrumentation through (LDM) [B]

In Figure 21 decodes for PC register in “if (i == 15)” condition before instrumenting for return instruction. The helper function body for the function return instruction instrumentation are given in Figure 22.

```c
void HELPER(func_ret)(uint32_t junval, uint32_t line_number, uint32_t ret_from)
{
   fprintf(junaid_trace,"FUNC_RET TO: 0x%08x @ line:%d\n",junval6/xffffffff, line_number);
   fflush(junaid_trace);
}
```

Figure 22: Body of Helper Function for Function Return Trace Message Generation at The runtime [C]

Instructions in Figure 22 generate trace messages for the function return instructions. The AND operation with the variable “junval” is always executed for the half word access as
given in ARM Reference Manual. Similarly, for the full word access the address is calculated by executing the AND operation with “0xffffffff” instead of “0xfffffffe” as shown in Figure 22.

Before wrapping up this section it is worth mentioning that different compilers generate different instructions for returning from functions when it comes to popping registers from stack at the end of the subroutine.

4.3.2 - Load Store Instruction Instrumentation

As explained before, Load and Store instructions are one of the most critical instructions with respect to the creation of the runtime bugs related to invalid memory accesses. For example, if a load instruction accesses the memory, to fetch the data element of a global array, which is out of the boundary of the memory allocated for a particular array then it causes an error and it can lead to an undefined program behavior and can have dramatic consequences. Similarly, storing any data element to an invalid memory location leads to critical program errors, sometimes resulting in crashing the whole program.

There are two approaches to instrumentation of the load store instructions. In the first approach, the user could get information regarding the instruction set type as well to which a load or store instruction belongs to. While in the second approach no such information can be extracted, rather only the size of the data loaded or stored and address of the memory where it is stored or loaded from. First approach leads to increased coding and complex instruction decoding for individual instruction set architecture. The second approach has more simplistic method of instrumentation within QEMU and it is fairly easy to do instrumentation for load and store instructions. This is because all of the load and store instructions pass through the same frontend TCG functions for related Micro-op generation. It is noticeable that within the translation engine of QEMU all of the load and store instructions are executed through the same abstraction level which is one level below the identification of a load and store instruction itself. The requirement of this Master thesis was to simply instrument any load store instruction regardless of any particular instruction set architecture type; hence the second approach was taken in this case and all of the load and store instructions were instrumented without discriminating between different types of instruction set architectures. The reason for not specifying the instruction set architecture type for a load of store instruction is that all types of load and store instructions do the same task of accessing memory for storing or loading data, and only the access type (load or store) and the address are relevant for application error analysis. The only exception in the load and store instruction instrumentation was
discrimination of load exclusive and store exclusive instructions. This discrimination is made because these instructions have special semantics to analyze multi-threading data-race errors.

The following are the types of Load and Store instructions which are regularly used in programs which are built for ARM microprocessor.

- LDR (Loads a constant address value from a Register)
- LDD (loads contents of two consecutive memory locations if the architecture is of 32bits otherwise one double word of 64bits)
- LDM (loads multiple registers, which also normally works as POP instruction to return register from stack including program counter to return from function).

In addition to above mentioned types of load instructions there are more variants or load instructions depending upon the size of data being loaded. Similarly, there are different variants of store instructions as well. These variants are analogues to the load instruction variants, with the operation of storing the data at a specified memory address instead of loading the data.

- STR (Store Register contents to memory location)
- STM (Stores non empty sub set of general purpose registers to memory location)
- STRB (Stores least significant byte of Register to the memory address)
- STRH (Stores half word from register to the memory location)

The code corresponding to load and store instructions instrumentation which is applicable to all type of instruction set architecture is given in Figure 23.

```c
void tcg_gen_qemu_ld_i32(TCGv_i32 val, TCGv addr, TCGArch idx, TCGMemOp memop)
{
    int size = size_of_memop(memop);
    genu_log("tcg_gen_qemu_ld_i32: memop=%d, size=%d\n", (int)memop, size);
    TCGv_i32 byte_type = tcg_const_i32(size);
    gen_helper_load_test_trace(addr, byte_type);
    tcg_insn_gen_i32(THC refill, byte_type);
    memop = tcg_canonicalize_memop(memop, 0, 0);
    gen_insn_i32(INDEX_op_qemu_ld_i32, val, addr, memop, idx);
}

void tcg_gen_qemu_st_i32(TCGv_i32 val, TCGv addr, TCGArch idx, TCGMemOp memop)
{
    int size = size_of_memop(memop);
    genu_log("tcg_gen_qemu_st_i32: memop=%d, size=%d\n", (int)memop, size);
    TCGv_i32 byte_type = tcg_const_i32(size);
    gen_helper_store_test_trace(addr, byte_type);
    tcg_insn_gen_i32(THC refill, byte_type);
    memop = tcg_canonicalize_memop(memop, 0, 1);
    gen_insn_i32(INDEX_op_qemu_st_i32, val, addr, memop, idx);
}
```

Figure 23: Load Store Instruction Instrumentation for Data in the range of (8 to 32) bits [B]
The `qemu_log()` instruction generates the log message at compile time while “gen_helper_load_test_trace” function generates trace messages for the load instructions and `gen_helper_store_test_trace` generates trace messages for the store instructions. In the helper function call “byte_type” argument tells the helper function about the size of data which is loaded or stored at the address specified by the first argument “addr”. This helper function prototype can handle loads and stores from 8bit to 32bit data elements while for 64bit data elements the instrumentation functions are given in Figure 24.

```c
void tcg_gen_qemu_id_i64(DECL TCGV i64 val, TCGv addr, TCGArg idx, TCGMemOp memop)
{
    qemu_log("tcg_gen_qemu_id_i64: memop=%d\n", (int)memop);
    TCGV_i32 byte_type = tcg_const_i32(64);
    gen_helper_load_test_trace(addr, byte_type);
    tcg_temp_fres_i32(byte_type);
    if (TCG_TARGET_REG_BITS == 32 && (memop & NO_SIZE) < NO_64)
    {
        tcg_gen_qemu_id_i32(TCGV_LOW(val), addr, idx, memop);
        if (memop & NO_SIGN)
        {
            tcg_gen_sari_i32(TCGV_HIGH(val), TCGV_LOW(val), 31);
        } else {
            tcg_gen_movu_i32(TCGV_HIGH(val), 0);
        }
        return;
    }
    memop = tcg_canonicalize_memop(memop, 1, 0);
    gen_ldst_i64(INDEX_op_qemu_id_i64, val, addr, memop, idx);
}

void tcg_gen_qemu_st_i64(DECL TCGV i64 val, TCGv addr, TCGArg idx, TCGMemOp memop)
{
    qemu_log("tcg_gen_qemu_st_i64: memop=%d\n", (int)memop);
    TCGV_i32 byte_type = tcg_const_i32(64);
    gen_helper_store_test_trace(addr, byte_type);
    tcg_temp_fres_i32(byte_type);
    if (TCG_TARGET_REG_BITS == 32 && (memop & NO_SIZE) < NO_64)
    { return; }
    memop = tcg_canonicalize_memop(memop, 1, 1);
    gen_ldst_i64(INDEX_op_qemu_st_i64, val, addr, memop, idx);
}
```

**Figure 24: Load Store Instruction Instrumentation for Data of 64 bits’ size** [B]

The functions shown in Figure 24 work just like 32bit functions but the difference is that value in this case is 64bit which is either loaded or stored. In 64bit variant of load store functions it is always known that instrumentation is done for 64bit values but for previous functions (8bit to 32bit) a differentiation is made by getting “size” information of the “memop” argument by using the function “size_of_memop (memop)” as shown in Figure 23. The helper
function body of load and store instructions for generating trace messages is given in Figure 25.

```c
void HELPER(load_test_trace)(uint32_t junval, uint32_t memop)
{
    fprintf(junaid_trace,"Load From: 0x%x Size %d Bits\n", junval, memop);
    fflush(junaid_trace);
}

void HELPER(store_test_trace)(uint32_t junval, uint32_t memop)
{
    fprintf(junaid_trace,"Store At: 0x%x Size %d Bits\n", junval, memop);
    fflush(junaid_trace);
}
```

Figure 25: Helper Function Body for Load/Store Instructions [C]

The first argument contains the address from which the data is either loaded or stored at, while the second argument gives the information about the size of the data byte loaded or stored. There is a special kind of load store instruction which needs special attention for instrumentation. These instructions are, as explained earlier, load and store exclusive. Whenever there is a load or store exclusive data is loaded from or stored at physical memory location therefore is of special interest. The instrumentation functions and helper function bodies are almost the same like mentioned before, but the instructions were explicitly identified out of the translation function to explicitly instrument them in order to generate distinct trace messages for exclusive load and store instructions.

**4.3.3 - Marking of Specific Functions (Malloc/Free, Pthread_create/Join)**

After the instrumentation of function call, function return and load/store instructions the next challenge was to mark some specific functions of interest at the runtime. This task was very critical and computationally intensive depending upon the approach taken to perform the task. The purpose of the instrumentation for marking the functions is already described in the previous chapter. There are two major approaches to do the instrumentation of marking the functions which are discussed here.

One of the approaches was tedious in its implementation and required a lot of research before confirming to its implementation. Since the functions selected for marking belong to shared libraries, the runtime trace messages only produce absolute addresses of function calls. An absolute address is the address in virtual memory where a piece of code or subroutine starts with respect to the beginning address of virtual memory where the complete shared library is
loaded at the runtime. The relative address is the beginning address of a subroutine or function body with respect to the beginning address in the library file itself. So, since in the first approach we had absolute addresses, it was extremely important to have the base address of memory where a shared memory was loaded at the runtime. Since the relative addresses of the function bodies always stay constant in an ELF that information is always at hand without any instrumentation. The only information required at this point would be then the base address of the memory where the shared library is loaded.

Once the base address is acquired, this base address can be subtracted from the absolute address acquired from instrumentation of function calls. The result of the subtraction gives the relative address of a subroutine beginning which can be used to identify and mark the required functions. However, this process is very tedious and cannot be used during translation and generation of trace messages as it will introduce high overhead. This method could be useful for the trace file analysis program itself which might use some sort of script to perform the above mentioned subtraction and address comparison for each function call to identify and mark required functions. Although this approach was not used to perform the actual implementation for marking the required functions, the experimental research will further explain this approach in detail.

To make the things quicker and simpler, the second approach was adopted to perform the function marking task. This approach involves use of one of the dynamic linker features. There is a special feature of dynamic linker which can be used to invoke certain wrap functions. In this process the instrumentation is indirectly involved for generation of trace messages. What basically happens is that in the result of calling a function through _wrap_ functions custom assembly instructions are invoked at compile time which are identified, during JIT compilation, and instrumented. These wrap functions are called first before calling the original function. The body of these wrap functions can contain additional required instructions as well. There is a specific syntax for using the wrap functions which is recognized by the dynamic linker at the runtime and this feature is required to be invoked at the compile time by giving appropriate linker commands. When compiling the program, a certain argument, as shown in the list below, to the compile command makes linker to recognize certain wrap functions.

1- -Wl,-wrap,pthread_create
2- -Wl,-wrap,pthread_join
3- -Wl,-wrap,free
4- `-Wl,-wrap,malloc`

Above shown 4 arguments, given to compiler at compile time, have certain syntax similarities with differences in the names of functions to be marked. Without these arguments the compiler gives errors about wrap functions whose body is shown in the Figure 26.

Figure 26: Wrap Function Bodies for marking functions [A]
As it can be seen in the Figure 26, there is a similarity in the function prototype declaration where "__wrap_" syntax is common. Rest of the prototype definition contains the actual functions and their arguments which are to be marked. At the end of the body of all of the wrap functions the statement "__real_" is again common while rest of the syntax corresponds to the marked function. The letter "A" shows correlation of this code segment with related block in Figure 6: Final Expected System Overview.

So, whenever a function is defined as shown Figure 26, upon a call to that shared library function the linker first invokes the wrap function of that particular functions; for example, when "malloc" is called, the linker invokes "__wrap_malloc". After the call to the wrap function the linker identifies the statement "__real_malloc" to call the original shared library malloc function. However, the question is that how this approach is useful in marking the functions. Also, what are those weird looking assembly instructions within the C language code?

The use of assembly instructions helps to generate custom messages within the trace file. Figure 26 shows similar usage of assembly instructions throughout all the wrap functions. The reason for the use of such instructions is that they are used in writing values to co-processor registers. By writing data to dedicated co-processor registers it becomes easier to identify the instruction for instrumentation within the translation engine of QEMU. It is like invoking custom instructions through user code to efficiently perform the instrumentation. If these custom assembly instructions are not inserted in the program, there will be no use of wrapping the functions. These custom assembly instructions give us the luxury to insert custom data which can be identified at the compile time within QEMU. For the purpose of recognition, a specific coprocessor register C14 is written with custom data using general purpose register R2, as shown in Figure 26. Whenever this register is written the disassembly function for coprocessor instructions is invoked and within that function our custom code patch identifies the given information against specified registers to display it in the trace messages. Let's take the example of malloc to explore these assembly instructions one by one. The combination of instructions `asm("mov r2, #0x0100");` and `asm("MCR P15, 0, r2, C14, C0, 0");` write value 0x100 to coprocessor register C14 to let the HELPER function know that it is malloc function which was called by the subroutine call instruction. Then the combination of instructions `asm("mov r2, %[value]":[value]"r"(c):);` and `asm("MCR P15, 0, r2, C14, C0, 0");` writes the number bytes of memory which was allocated. Finally the combination of instructions `asm("mov r2, %[value]":[value]"r"(begin_addr):);` and `asm("MCR P15, 0, r2, C14, C0, 0");`
writes the beginning address in memory from where the memory was allocated for a certain `malloc` function call. The above mentioned set of information is written in the trace file at the runtime, while marking the function name “`malloc`” itself. Similarly, the rest of the wrap functions are invoked and inline assembly instructions are used to generate the required information for the trace file generation which can be then used later for the analysis. The coprocessor register identification code patch used in translation phase is given in Figure 27.

```c
//... VFTrace code patch for Function Marking
if(crn == 14 && rt == r2)
{
    TCGv_i32 vf_arg = load_reg(s, rt); // loading content of r2
    gen_helper_func_mark(vf_arg, __LINE__); // Calling HELPER for trace msgs
    tcg_temp_free_i32(vf_arg); // Free VF temporary
}
//... End of VFtrace code patch
```

Figure 27: Code Patch for Identifying content of C14 coprocessor Register [B]

The “`if(crn == 14 && rt == r2)`” condition statement filters out for C14 register and R2 general purpose register which is preselected for this purpose. The selection of C14 is made specifically for this purpose because C14 register of coprocessor P15 is not used by the compilers, with reference to the ARM manual, hence it is used for instrumentation purposes.

Next chapter shows the experimental research and its results, which is performed, using above mentioned solution approaches.
Chapter 5

5.1 - Experimental Research

In this chapter experimental research and its results will be discussed for each of the sub solution approaches described in the previous chapter. The aim of the experimental research is to analyze if the instrumentation results are efficient enough to prove the feasibility of the approach. The experiments will be performed by using custom C functions which are developed in such a way that each function contains dedicated instructions for testing of the selected instruction instrumentation. For example, the testing of the load/store instruction instrumentation is performed by developing a code which contains all types of data access instructions. Similarly, different variants of other selected instructions are used in their respective test programs for analyzing the efficiency of instrumentation. However, before the explanation of the experimental research and its results experimental set up for the project is described in next section.

5.2 - Experimental Setup

Experimental setup of the binary instrumentation in QEMU did not require presence of any of the hardware components. This experimental setup aims at testing the instrumentation results themselves and not the performance evaluation between two different versions of the program. For testing the instrumentation results QEMU, the open source emulator, is used. There were two different versions of QEMU used for the experimentation along with two different ARM Cross compilers for testing a range of instrumentations and a variation in the use of instruction set of ARM for executing different instructions. The following set of QEMU and ARM Cross compilers was used for the experimentation:

1- QEMU 2.5v
   ARM Linux GCC 4.5v
2- QEMU 2.6v
   ARM Linux GCC 4.9v

Both experimental setups have their advantages and little disadvantages, but the advantage mainly comes from the use of different compiler versions and not from QEMU itself, as QEMU processes almost all types of instructions very well.

It has been observed that ARM Linux GCC 4.5v compiler generated more conventional instructions for function return statements than the newer compiler. By conventional function
return statements, it means that the compiler generates instructions which are referred more directly in the manual. For example, in the newer compilers the instructions to return from a subroutine would follow the ARM reference manual less. The manual mentions explicitly instructions to return from a subroutine, and all of the other supplementary instructions, which could also be used, are mentioned in footnotes. It is sometimes very complex to correctly instrument an instruction due to the dependencies between certain instructions and registers. It was observed that the compiler used at Vector Fabrics premises, which is version 4.5v, provided 99% efficiency in total number of subroutine call and return instruction decoding, while the compiler used in developer’s own laptop, which was 4.92v, gave efficiency in between 90 – 95 percent. Following sections will show the experimental results for each of the sub problems selected for verification of QEMU’s feasibility for binary instrumentation.

5.2.1 - Function Call and Return Experimental Results

For experimental research for function call and return instructions multiple experiments are performed. All of the experiments contained use of shared libraries apart from user application code. Figure 28 shows a sample program which was used to test the function call and return instructions.

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

int values[] = {88,56,100,25};
int cmpfunc(const void *a, const void *b)
{
    return (*(int*)a - *(int*)b);
}

void check_func(void)
{
    return;
}

int functiontest(int, void (*ptr_func)());
int main(void)
{
    int i, sum;
    for(i=0;i<5;i++)
    {
        sum += functiontest(i, check_func);
    }
    check_func();
    return (sum--);
}

int functiontest(int i, void (*ptr_func)())
{
    printf(values, i, sizeof(int), cmpfunc);
    if(i&1)
    |
        ptr_func();
    int sum = 0;
    sum += (int)(floor(1)/0.333);
    return sum;
}
```

Figure 28: Example_Test for Function Call and Return Test
The test for function call and return in Figure 28 contains multiple subroutines. The subroutine “functiontest” and “check_func” are two local user subroutines used within the user executable program while the subroutine “qsort” is a shared library subroutine. As we know that shared library subroutines are difficult to tackle because we get the absolute addresses for those subroutines rather than the relative addresses as in the case of user executable. So, the shared library function call and return testing is ignored for this moment and that will be shown in the later section. Below, the results of functiontest and check_func are shown. In order to show that right addresses are called at the runtime and not some random addresses are called the Figure 29 shows the “objdump” output for the executable of example_test program. Objdump is a Linux tool which presents the hexadecimal form of an executable in the proper format. We will focus at the section required to display our results instead of showing complete objdump output which is very huge. Figure 29 shows section of functiontest function obtained using objdump command to the executable file of example_test.c.

![Figure 29: Objdump for user executable to test Function Call and Return Instrumentation](image-url)

Figure 29 shows the hexadecimal form of the section of the whole executable which contains “main” body of the program of C language shown in Figure 27. It can be seen that within the “main” body of the HEX program the subroutine “functiontest” is called using the
Binary Instrumentation with QEMU

instruction “bl” which lands at the address “00008468” which is shown at the bottom of Figure 28. Whenever the call to subroutine is executed the program flow jumps from that point of instruction to “00008468” and starts to execute this section of the code. When the execution of the “functiontest” is over the program should return to the instruction next to “bl” which is “MOV” at the address “0000843a”. Now, we look in our trace file, which was generated at runtime, if we have the correct call stack or not. By correct, it means that the call to subroutine and return from the subroutine should be at the right addresses and for the time being we ignore shared library subroutines called within the body of “functiontest”. Figure 30 shows results obtained in the trace file for call and return of subroutines.

Figure 30: Trace Results for Function Call and Return Instructions

Figure 30 shows the results obtained within the trace file for function call and return instructions with appropriate addresses as expected. The load and store instruction instrumentation results can be ignored at this time. The line: 10055 and line: 11110 suggest that these instructions were executed in the result of instrumentation done at those line numbers in translation engine of QEMU. This feature is useful for debugging purpose during the development of instrumentation only.

A similar effect was observed for other subroutines found within the program, namely “check_func” and “cmpfunc”. The trace file contains trace messages containing addresses of function entry point and returns which is in accordance with the addresses shown in Figure 29, which holds for all of the other subroutines as well. Similarly, for the shared library function calls the section of marking shared library functions will explain more details about experimentation. It is important to mention that for user application the subroutine addresses which are printed in trace file are the relative addresses, while addresses for the shared library functions are absolute. The reason for this is that user application is directly read by the QEMU; however, shared library functions are read by QEMU after getting loaded to specific location.
in memory which becomes the base address of the shared library. All of the relative addresses of the shared library functions are added to that base address of memory, where the library is loaded at the runtime, and thus when QEMU reads those shared library functions at the runtime it shows absolute addresses rather than the relative addresses as is the case with the user application.

The next section will describe the experimental results for the Load Store instructions.

5.2.2 - Load Store Instruction Experimental Results

As explained, before, the use of a certain compiler results in significant changes in the implementation of particular instructions. Load store instruction instrumentation was fairly simple as compared to other sub problems, as the approach taken to instrument load store instructions did not include distinction between different instructions set types. Figure 31 shows the program used to test the load store instruction instrumentation in the program.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
volatile int inx = 0;
void break_pnt(void)
{
    return;
}
int main(void)
{
    char *c = strdup("hello\n");
    break_pnt();
    char  c1 = c[1];
    double c3 = c[1];
    short c4 = c[3];
    char  c2 = c[3];
    break_pnt();
    printf("%d\n", (unsigned long)c);
    return 0; //c1+c2+c3+c4);
}
```

Figure 31: Program to Test Load Store Instrumentation

Figure 31 contains a piece of code which contains 4 instructions which load data with three different sizes and stores it in appropriate sized variables. The trace file should display trace messages for all of different types of data being loaded and stored. According to the above shown C program the trace file should display load instructions for the data sizes of 8 bits, 16 bits, 32 bits and 64 bits.

It can be observed that load and store instructions, to be observed, are only used in between two function calls namely “break_pnt”. This type of calling was used to identify
certain instructions within the trace file as the overall trace file size is very large to scan through the whole file. The trace file section containing the load store instruction trace messages is displayed in Figure 32.

```plaintext
func_call:0x00008424 @ line:10055
Stored_At:0xf6ffeef34 Size:32 Bits
Load_From:0xf6ffeef34 Size:32 Bits
func_ret:0x00008444 @ line:10742
Load_From:0xf6ffeef3c Size:32 Bits
Load_From:0x00012008 Size:6 Bits
Stored_At:0xf6ffeef30 Size:8 Bits
Load_From:0xf6ffeef3c Size:32 Bits
Load_From:0x0001200a Size:8 Bits
Stored_At:0xf6ffeef40 Size:64 Bits
Load_From:0xf6ffeef3c Size:32 Bits
Load_From:0x0001200d Size:6 Bits
Stored_At:0xf6ffeef3a Size:16 Bits
Load_From:0x0001102c Size:32 Bits
Load_From:0xf6ffeef3c Size:32 Bits
Load_From:0xa001200b Size:8 Bits
Stored_At:0xf6ffeef39 Size:8 Bits
func_call:0x00008424 @ line:10055
Stored_At:0xf6ffeef34 Size:32 Bits
Load_From:0xf6ffeef34 Size:32 Bits
func_ret:0x00008444 @ line:10742
```

Figure 32: Trace Load Store Instructions

The addresses with higher values, for example “0xf6ffeef34”, can be ignored as they belong to the addresses on the stack where local variables are stored. At this moment we are only interested in analyzing the addresses with lower values. The addresses “0x00012009, 0x0001200a” correspond to lower addresses, which mean that they are the relative addresses within the executable file. A closer look at the sizes of data at the right side of trace file shows that we have all 4 different sizes of data stored at four different types of variables as declared in the C program itself. This trace file content clearly shows that the approach taken to instrument the load store instructions worked absolutely fine. But, the use of compiler version 4.92 in this case has increased the overall load and store instruction number as compared to a lower version of compiler.

The difference is that the newer version of ARM cross compiler loaded the data directly from an, “register + offset”, address into the required location. But, in this case data is loaded indirectly by use of multiple registers which is cause of the generation of extra trace messages. In both cases the result is accurate and according to the requirement but the only exception is of the extra load and store instructions generated by the compiler.
5.2.3 - Experimental Result for Marking of Functions

As explained earlier some of the shared library functions are very important as compared to other ones. Their importance is due to the fact that they deal with memory allocation and spawning of threads at the runtime. In addition, if those functions are used in incorrect manner the function can easily lead to crashing the application. This is also due to their frequency utilization as well as their interaction with hardware resources like memory and other peripherals. For example, malloc is used to reserve the memory with a certain number of bytes for a certain data array. Similarly, “free” is used to free the memory allocated by the use of malloc. On the other hand, pthread_create and pthread_join functions give information about the spawning of threads at the runtime and joining of threads once they are finished with their work. The approach taken to instrument such function marking was by the use of special linker option explained in the previous chapter, but here the results of both of the approaches introduced in previous chapter will be discussed.

Before discussing the final proposed method, the first approach will be discussed which was tested but not finally adopted for the tracing of the special functions. We still must this method to find the names for all functions to display call-stacks for detected errors. According to the first approach it was very vital to have the base address of memory where the shared library was loaded at the runtime. With this base address and absolute address, which is obtained in the result of function call instrumentation, the name of the function can easily be marked. It is important to mention that QEMU, after reading the binary file, fetches the required shared libraries in memory at the runtime. It looks for the shared library locations and if not found it gives an error. For custom built shared libraries, including the shared libraries built for ARM architecture, the path has to be given explicitly while running the program through QEMU. In order to get the base address of the shared libraries from QEMU to the trace files a special option will have to be given to QEMU which is “strace”. Once QEMU has the argument “-strace” it recognizes that it needs to print the memory map of files opened at the runtime. It also displays the memory map of some of the other operations like writing of strings. It displays the memory map information by getting information about the system calls made at the runtime. Following figures show the memory map of shared libraries open in this experimentation.
Figure 33: Libc Shared Library Opened and Returned Base Address

The last line in the Figure 33 displays the returned base address of memory, which is 0xf66b3000, where the shared library is loaded at the runtime. Figure 33 shows the information about the Libc shared library for ARM architecture loaded at the runtime. Similarly, for pthread library the strace information is given in Figure 34.

Figure 34: Pthread Library Opened at The runtime

The returned base address for pthread library is 0xf679a000 in this run. Similarly, for custom built shared library which is used for marking the specific functions, the strace information is given in Figure 35.

Figure 35: Libwrap custom shared library strace information

All the images shown above related to the strace information give key information for marking the functions at the runtime, when using a custom script to read the individual function call addresses from the complete trace file. In this mechanism the script will have to subtract the base address from the address obtained from each of the function calls to the shared library function to get the relative address for extracting the name of the function called at the runtime. We take the example of strace information of libc.so.6 which is displayed in Figure 33 to
explain the first approach of marking the functions. First look at the following piece of trace file shown in Figure 36.

```plaintext
func_call: 0x00008644 @ line:10068
Load_From: 0x00011024 Size:32 Bits
func_call: 0xf66ec038 @ line:8995
Stored_At: 0xf6ffef2c Size:32 Bits
Stored_At: 0xf6ffef30 Size:32 Bits
```

Figure 36: Trace for Marking Function with Base Address Approach

In the Figure 36 the highlighted address 0xf66ec038 is the absolute address in memory for the function “printf”. The problem is that with only this address in the trace file it is impossible to judge which function is called. Hence there is some more information required to fulfill this task. It is very important to mention the fact that “relative addresses within the executable always remain constant, whether it is some user compiled executable file or some shared library executable, unless the library itself is upgraded or changed”. With this information we can always assume that as long as a library is not changed, which is normally not a frequent case, the relative addresses of functions of the shared library will always correspond to the selected function for marking. So, if the base address, of any shared library, in memory is known then it can easily be found which function is called by calculating the absolute address using the base address. In the above picture we have the absolute address of a shared library function call which is 0xf66ec038 and we know from Figure 33 that the base address of Libc shared library is 0xf66b3000. With the following arithmetic operation, we get a relative address of a function inside the shared library:

Relative Address: Absolute Address (function) – Base Address (SharedLibrary)

Relative Address: 0xf66ec038 – 0xf66b3000 = 0x00039038

The value “0x00039038” corresponds to a function, which is called at the runtime, in shared library. The Figure 37 shows the function, which is called, with respect to the address “0x00039038”.

Figure 37: Marking of fprintf using base address approach

Figure 37 clearly shows that the absolute address, which was displayed in trace file in result of a function call, corresponds to the fprintf function call.

Figure 38: Source Code for experimentation of Function Marking

It is clear in the Figure 38 the function fprintf is called several times and each time fprintf is called the address 0xf66ec038 will be displayed in the trace file which is the absolute address in memory. The first approach, which was discussed in previous chapter and displayed above, will need subtraction of base memory address form this address to get relative address.
of fprintf to identify the name of the function for better analysis. Similarly, using the above mentioned approach information about all the functions called, which belong to shared libraries, can be obtained. But, this process is very tedious because in this process the program analysis the trace file will have to call a script to do this arithmetic operation and comparison of addresses for each function call. This will put considerable amount of overhead on the overall program and timing of the analyzing the program.

In the previous chapter an alternative approach to do the function marking was discussed. In this approach special linker options were invoked along with additional piece of code which was built in the form of custom shared library, libwrap.so, which was opened by QEMU in Figure 35. With this approach the linker identifies custom functions defined using keywords __wrap_ and __real_ to call the shared library functions. The body of functions wrapped using __wrap_ keyword is called first before calling original shared library functions using __real_ keyword. In this approach custom instructions were added to the source code of __wrap_ functions to gain required information to be displayed in trace file. These instructions moved some data into some specially selected coprocessor registers, so that can be identified at the compile time within QEMU to generate the related custom trace messages at the runtime in the trace file. An example of a wrap function is shown again in the Figure 39.

```
void *__wrap_malloc (size_t c)
{
    int *begin_addr = 0;
    begin_addr = (int*) __real_malloc (c);
    asm("mov r2, @(0x0100)\n");
    asm("mv r15, 0, r2, C14, C0, 0\n");
    asm("mv r2, %[value]::\[value]::\{value\}::begin_addr::\n");
    asm("mv r2, %[value]::begin_addr::\n");
    fprintf(stderr, "begin_addr: 0x%08x\n", begin_addr);
    return begin_addr;
}
```

Figure 39: Example Wrap Function for Malloc Function Marking [A]

The contents of the Figure 39 are already explained in a great detail in previous chapter. The body of the above function enables identification of function malloc at the runtime, and it provides information about the beginning address in memory from where the memory is allocated. In addition, it also provides information about the number of bytes reserved in the memory. The number “0x100” is a special identification code for the function malloc. For other functions like “free”, “pthread_create” and “pthread_join” the codes are “0x102”, “0x104” and
“0x106” respectively. Whenever a certain function out of these four functions is called this identification code is scanned to get the information on the function type followed by additional information required to be printed in trace file. The output of functions marking for malloc and free is shown in the Figure 40.

```
In the highlighted parts of the trace messages shown in Figure 40 the marking of functions “malloc” and “free” is done at the runtime. In addition to their names the starting location in memory, from where the memory is allocated, and the number of bytes which are reserved by malloc are also displayed. In the second message the highlighted part free shows the starting address in memory from where memory is freed. These messages are correct in relation to each other and the results are correct. Similarly, pthread_create is marked with the thread id and pthread_join with reference to the thread in memory instead of the thread id. Line number, 7170, shows place of the instrumentation in translation engine for the function marking.

5.3 – Performance Evaluation

This Master project did not include performance optimized solution in its scope, yet runtime performance of dynamic analysis tools is a challenging and critical issue. To demonstrate the benefits of dynamic analysis with the help of QEMU a comparison is performed with different setups. In order to measure the performance of the system for recording the slowdown factors several experiments were conducted. Out of these experiments,
results of three experiments are mentioned in Table 2. These experiments were conducted on two different platforms for the comparison purposes. It is worthy to mention that mere emulation, without any instrumentation, involves some overhead as well. This is because the binary code translation also introduces some overhead in the process, yet it was interesting to measure the performance of applications which were emulated vs applications which were running on a target embedded device. For running applications on the target embedded hardware Raspberry Pi Model B was selected which contains a 900MHz processor clock with Linux running on it. For emulation QEMU was running on x86 based Linux platform whose CPU model was i5 with clock rate up to 2.5GHz. Selection of the platforms was made so as to show that the actual applications, after dynamic code analysis, will be running on top of the embedded target hardware like Raspberry Pi. However, the dynamic code analysis will be performed on the host development platform, like x86 based Linux machine. The purpose of comparison is to show the difference in time between the emulation of instrumented code vs running of the application without instrumentation on a target embedded platform. Additionally, several experiments are performed with Valgrind which was running on the target embedded hardware (Raspberry pi). The objective of using Valgrind on raspberry pi is to show the inefficiency of dynamic code analysis when performed on the target embedded hardware with respect to slowdown factor. The images shown below only contain a comparison of QEMU vs Raspberry pi without Valgrind, but the results gathered from experimentation are tabulated in Table 2. The comparison of applications is performed with respect to the real time clock values and all other time values, like CPU time and user application time, are ignored.

![Figure 41: Performance Measurement with Figure 31 code](image)
Figure 42: Performance Measurement with Figure 28 Code

Figure 43: Performance measurement with Image processing application
Each of the Figures 41 - 43 contains results for the emulation, as well as, for the application running on raspberry pi. It can be observed that there is a clear difference in the elapsed time of both type of application executions. The time of the application running on target embedded hardware is significantly less as compared to the one being emulated. There are obvious reasons for the slowdown of the application which are:

1. Overhead of the emulation itself in the form of the code translation and relocation
2. Overhead created by the instrumentation (which is less significant)
3. Overhead of the printing of the trace messages

Out of the three main reasons of overhead, number 1 and number 3 factors result in a significant overhead to the system, while number 2 is only related to the additional lines of code added to the overall process. The code for the image processing application, which is used in the third experiment, is given in Appendix of this report. The image processing application takes up the data of a 400x400 ppm image and performs multiple operations on it. It converts the color image into greyscale, then compresses it by ¼, then finds edges of that image and finally display the compressed and converted image edges in the form of ASCII characters on the console. The table of comparisons between elapsed time of applications is given in Table 2.

<table>
<thead>
<tr>
<th>Application</th>
<th>Valgrind on Target (Raspberry Pi)</th>
<th>QEMU Emulation</th>
<th>Raspberry Pi without Valgrind</th>
<th>Slowdown (QEMU vs Raspberry Pi)</th>
<th>Slowdown (RPi with Valgrind vs Native RPi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code in Figure 28</td>
<td>4.65 seconds</td>
<td>244 milliseconds</td>
<td>12ms</td>
<td>20X</td>
<td>387.5X</td>
</tr>
<tr>
<td>Code in Figure 31</td>
<td>4.80 seconds</td>
<td>269 milliseconds</td>
<td>14ms</td>
<td>19X</td>
<td>342.8X</td>
</tr>
<tr>
<td>Image Processing Application</td>
<td>385 seconds</td>
<td>89.711 seconds</td>
<td>Approx. 5.636 seconds</td>
<td>16X</td>
<td>68.3X</td>
</tr>
</tbody>
</table>

The trend of lower slowdown factors in QEMU vs Raspberry pi performance analysis, in Table 2, with respect to the complexity of application is because of the fact that once QEMU translates a block of code it keeps it in its cache memory and no further translation for that particular code block is performed. If that block of code is needed again it simply ruses that
code from its cache memory. This reduces the overall overhead and decreases the slowdown factor in complex applications as compared to those applications which make less reuse of their codes. Applications which make less reuse of existing codes put more overhead of translation on the applications and increase the slowdown factor.

For “RPi with Valgrind vs Native RPi” performance values experiments are performed using Valgrind’s Helgrind plugin on Raspberry pi. Table 2 shows a drastic decrease in slowdown factor. This is due to the fact that Valgrind takes prestart time before starting the code analysis which is almost same for all applications, hence a smaller code example would not yield a reasonable result for comparison. But, this does not imply to the QEMU vs Raspberry pi (without any analysis tool) comparison. The selected image processing application, apart from other smaller code examples, was reasonable enough for a fair comparison between Valgrind and RPi native performance. The comparison of RPi vs Valgrind for image processing application clearly indicates that even if a fairly complex and resource enriched target embedded hardware like Raspberry pi is used still dynamic code analysis on the target embedded hardware is not feasible. In the light of above results it can be said that the dynamic code analysis is very much feasible using QEMU. Small target embedded hardware platforms cannot even afford to have an on board analysis tool due to their memory limitations. Clearly, small targets often run at even lower clock rates. That means that the QEMU instrumented emulation can reach a speed that is more close to the native application runtime, because the incurred overhead is compensated by the faster X86 processor speed.
Chapter 6

6.1 - Conclusion

The main aim of the Master project described in this report was to investigate if proposed emulation and instrumentation techniques are feasible and efficient, and specifically, if it is feasible to perform the code instrumentation during the JIT (Just in Time) compilation phase in QEMU. In addition, in the case of the ARM, processor being emulated, to investigate if the binary input provides sufficient information to create traces that allow the error/bug uncovering and reporting of such information to the application programmer. Realization of these aims required analysis of the QEMU source code, the ARM instruction set architecture, and instrumentation feasibility.

In this section of the report, the conclusions of the whole research and development will be discussed. In order to verify and demonstrate the feasibility and efficiency of the proposed idea it was necessary to analyze the instrumentation results. As mentioned previously, several critical sub problems of the dynamic binary code analysis were selected to show the feasibility of the proposed approach. If the results are found to be sufficient for the selected sub-problems, it can be said with a high confidence that the approach is feasible and efficient.

In the previous chapter the results of the instrumentation were discussed one by one for each of the particular sub-problems and all the results are very positive. On the other hand, it was observed that the selection of a compiler to compile the program before dynamic analysis might affect the overall efficiency. In the beginning of this report a couple of major questions were posed which were to be answered by performing the research and development in this Master project. These questions were:

- Will the dynamic code analysis for the cross platform emulated code be feasible?
- Is it feasible to do the code instrumentation during the JIT compilation phase in QEMU?
- In the case of ARM processor to be emulated, does the binary input provide sufficient information to create traces that allow the error/bug uncovering and reporting of such information to the application programmer?

In the light of the research and development performed to assess the feasibility of the proposed approach following are the answers to the main questions posed in this thesis.
- The instrumentation of the ARM binary code during the JIT compilation phase is confirmed feasible.
- The created trace information is sufficient to do main error/bug analysis (load/store address tracing, malloc/free and thread create/join call tracing).
- The created information is sufficient for feedback to the programmer, regarding the error location (keeping track of call-stacks and tracking dynamic loading of shared libraries).

It was observed that QEMU provides a very flexible and feasible environment for performing the binary code instrumentation. The study of the QEMU source code revealed that there were several layers of the binary code translation which could be used for the binary instrumentation. However, the most suitable layer for the instrumentation was the top layer of TCG where translation of instructions takes place. Moreover, QEMU provides decoding for almost all types of ARM instruction set architecture which was very handy in further decoding of instructions for performing the instrumentation. QEMU does not discriminate between normal branch instructions and branch & link instructions for calling subroutines which had to be performed apart from instrumentation. It was handy in finding the right kind of instruction set architecture and it also provided clues to where a particular instruction decoding can be performed in the source code for the instrumentation. The instrumentation of instructions provided sufficient information for feedback to the programmer for error/bug reporting.

The efficiency of the call stack for the function call and function return statements was almost 99%. The difference in the total number of function calls to the function return statements was used to measure the efficiency of the function call stack efficiency. The efficiency of the load store instruction instrumentation was almost 100%. It is important to mention that instrumentation of the load/store instructions with QEMU was comparatively easier as compared to the function call stack and marking of functions. The instrumentation showed very encouraging results for the load/store instructions which are very critical for finding errors related to memory accesses.

The trace file demonstrates very good results for function marking. The results of the function marking showed that the total number of marked functions were the same as the total number of functions dedicated for the test purpose. The approach to mark required functions, with additional required details, was feasible and very efficient, and the results of the trace file show 100% accuracy in marking of functions. This implies that the marking of functions was
correctly performed and the results were very good. The results of the instrumentation show that the use of HELPER functions for runtime trace message generation was a feasible approach. Apart from the many positive results deduced out of this research there were few negative results as well which are given below:

1. In few cases the call-stack cannot be accurately reconstructed. That is a pity, but it is a known problem also for general debuggers such as GDB.

2. Doing this analysis on only same plain binary applications was not sufficient, because for marking of the functions high level source code relinking with “_wrap_” specific functions is performed as explained in solution approach and experimental research sections. This can be acceptable as compromise, but might also be solvable in other ways.

3. Tracing of multi-threaded behavior, with thread-id's inserted in the trace file, was not yet confirmed. This is partly because the required support in QEMU itself is still in flux. This was left for the future work.

Referring to point 1, it was observed that the overall efficiency in the percentage of the call stack for the function calls and returns is variable depending on the selection of compiler. The selection of a compiler effects the call trace due to the difference in the selection of the function return statements in few cases, and specifically, for the subroutines which are part of the shared libraries. It has been observed that functions which are called at the beginning of the program run and at the end of the program are special functions, known as glue code used by linker, whose return statements are varied depending on the version of a compiler. A more recent version of the ARM cross compiler might generate different results. For example, the version of the compiler which was used at Vector Fabrics, version 4.5, was better in selection of instructions with respect to the manual for returning from subroutines. On the other hand, the compilers used in another desktop system with version 4.92 produced less efficient results. The call stack efficiency of the function call and returns was almost 99% when 4.5v compiler was used to build the programs and with latest version of compiler in a second machine the results floated in between 90 – 95 percent.

As for the point 2, where function marking is concerned, the method used to perform the instrumentation for the function marking was not a part of the instrumentation performed inside QEMU’s translation engine. This implies that instrumentation on the plain binaries for marking of functions with additional details in trace messages was not possible inside of the
QEMU’s translation engine (TCG). For this purpose, a special wrapping library was used to re-link the program in a way which could generate the required results as shown in Figure 26. This approach is also acceptable since it will still not be a headache for the application programmer to relink the application, rather it will be a part of the tool itself. There is another approach to perform the same task mentioned in this report, but that requires additional work for the analysis software to properly provide feedback to the programmer about the required marked functions.

There has been some difficulty in getting the complete information about the multithreaded programs. For example, one of the major bottlenecks, in getting the complete information about the multithreaded programs, has been tracing of the thread ids. By measure of function marking mechanism it can be traced which POSIX functions are called at the runtime for spawning or joining a thread for a user function. However, QEMU does not help much in getting the thread id information at the runtime. Much of this is because the current versions of QEMU do not support multithreaded execution of user applications. A multithreaded application is emulated in such a way that the application behaves like a single threaded program. In this case the trace file only recorded a single thread id against all instructions which were corresponding to the native binary instructions of x86, though in the target application there were multiple threads spawned. Normally, the co-processor register C13 is used to store the thread ids for the operating system, but decoding of C13 in QEMUs environment did not prove to be fruitful, as it only showed fixed thread ids which was not as sufficient. The reason for this is already mentioned, which is that QEMU emulates the multithreaded application as if it were a single threaded application and in that case at the runtime the trace file only records a single value of the thread id. However, future versions of QEMU will contain the facility to emulate the multithreaded applications in the multithreaded manner which will be helpful in achieving the same kind of results for multithreaded programs as well.

The performance measurement of the system shows an interesting trend. It has been observed that there is a decrease in the slowdown factor of the application emulation with respect to its complexity. The performance measurement was performed with respect to the elapsed time of the emulation of application vs the application running on the target embedded hardware (Raspberry pi). The reason for this trend of decrease in slowdown factor is explained in performance measurement section. However, as for the conclusion of performance
measurement and quality of the work is concerned it is clear that the performance of QEMU is better as compared to the Valgrind (on target analysis tool) performance.

This MSc project has successfully instrumented applications for the binary code analysis on QEMU which supports cross platform emulation. This work is very valuable because it enables binary code analysis which is currently not available in Pareon tools, and it does not require presence of any target embedded hardware for the dynamic code analysis. In addition, all of the development and code analysis can be performed on a single host platform. The research has verified that the binary instrumentation idea proposed by this MSc project is feasible. The binary instrumentation within QEMU is feasible and efficient and can result in substantial information as feedback to the programmer for error reporting. Moreover, the performance of QEMU for generation of the trace file at the runtime is better than of the tools which run on the target embedded hardware. Summing up this MSc project resulted in:

- A comprehensive analysis of the binary instrumentation problem with QEMU
- A feasible and efficient prototype for the binary instrumentation for the trace file generation.
- The experimental research results demonstrate that the proposed instrumentation is feasible and efficient.

This demonstrates that the aims of the MSc project have been completely achieved.

6.2 - Future Work

Currently the size of the trace file is larger than possible due to much text in the representation of the trace messages. As a future work the size of the trace file can be substantially decreased by use of the appropriate binary representation to the textual trace messages. The application of image processing which was used to do performance measurement generated a trace file of almost 7GB which is huge for a small-size application. As a future work the trace file contents can be converted into binary values instead of text strings, this will also make the trace file compatible with the Vector Fabrics tool for the trace message analysis or to be used for visualization purposes.

The instrumentation can also be extended to account for more instructions. QEMU has a complete set of ARM instruction set architecture disassembly which can be decoded and instrumented just as it is demonstrated in this master thesis. There are speculations that in near future QEMU support of the multi-threaded programs will be launched. Once it is launched,
better results can be achieved for the multithreaded programs. Currently, there has been a limitation in gathering complete information about the multithreaded programs as explained in conclusion section. With newer versions of QEMU, the future instrumentation for multithreaded programs can be done as efficiently as it can be done for sequential programs.

In this Master thesis QEMU was used in user application mode and not in the system mode. In this mode of QEMU system calls were not instrumented or dealt with, though the research has shown that there is a considerable potential in QEMU to get the information regarding the system calls as well. It was observed in one of the experiments, during the experimentation for using the first approach for function marking, that the system calls can also be traced for a later analysis. QEMU has a big potential, as it is very flexible. One of the biggest advantages of QEMU over Valgrind and PIN is that it supports the cross platform emulation. With the cross platform emulation support one can instrument many different instruction set architectures, for example PowerPC, and not only x86 or ARM.
References:


  https://www.usenix.org/legacy/events/usenix05/tech/freenix/full_papers/bellard/bellard.html/


  https://software.intel.com/sites/landingpage/pintool/docs/65163/Pin/html/index.html#Pintools


Appendix (Image Processing Application Used for Performance Evaluation)

```c
#include <stdio.h>
#include <stdlib.h>
#define DIM_X 400 // Maximum Size X-Dimension
#define DIM_Y 400 // Maximum Size Y-Dimension
//void generate_p3(unsigned char size_x,unsigned char size_y,unsigned char max_color);
void generate_and_operate_p3(void);
void rgbtogray(unsigned int x, unsigned int max_val);
void perform_sobal(unsigned int x);
void vCompressImg(unsigned int dimension);
char filename[6]="0";
unsigned char img_data[160000]; // grayscale image data
unsigned int shared[480200]; // initial image data color
unsigned char cmp_array[40200]; // compressed data
unsigned char edge_img[40000]=0; // edge detected data of image
void gather_image_data(void);
void vCnvPictoAscii(unsigned int); // converting to ASCII ART
void main(int argc, char *argv[])
{
    char count=0, charc=100, getargument[12]={0}, xcount=0;
    // sprintf(filename,"%s", argv[1]);
    sprintf(getargument,"%s", argv[0]);
    while(charc!='\0')
    {
        printf("Character is:%c\n", getargument[xcount]);
        if (isalpha(getargument[xcount]))
        {
            puts("it's a letter");
        }
        else if (isdigit(getargument[xcount]))
        {
            printf("Found digit:%d", getargument[xcount]);
        }
    }
}```
filename[count] = charc;
    count++;
} xcount++;
charc = getargument[xcount];
}
filename[count++] = '.'; filename[count++] = 't'; filename[count++] = 'x';
filename[count++] = 't';
printf("File:%s\n", filename);
generate_and_operate_p3();
}

void generate_p3(unsigned char size_x,unsigned char size_y,unsigned char max_color) {
    unsigned char r[DIM_X][DIM_Y];
    unsigned char g[DIM_X][DIM_Y];
    unsigned char b[DIM_X][DIM_Y];
    unsigned int h=3;
    // unsigned char* shared;
    typedef struct
    {
        unsigned char r;
        unsigned char g;
        unsigned char b;
    } rgb;
    const rgb bar_colour[8] =
    {
        {255, 255, 255}, // White
        {255, 255, 0 }, // Yellow
        {0 , 255, 255}, // Cyan
        {0 , 255, 0 }, // Green
        {255, 0 , 255}, // Magenta
        {255, 0 , 0 }, // Red
        {0, 0 , 255}, // Blue
        {0, 0 , 0}, // Black
    };
    int x, y;
for (y = 0; y < size_x; y++)
    for (x = 0; x < size_y; x++)
    {
        unsigned char colour;
        colour = y / (size_y / 8);
        r[x][y] = bar_colour[colour].r;
        g[x][y] = bar_colour[colour].g;
        b[x][y] = bar_colour[colour].b;
    }
shared[0] = size_x;
shared[1] = size_y;
shared[2] = max_color;
    for(y = 0; y < size_y; y++)
    {
        for(x = 0; x < size_x; x++)
        {
            shared[h++] = r[x][y];
            shared[h++] = g[x][y];
            shared[h++] = b[x][y];
        }
    }
}
generate_and_operate_p3(void)
{
    unsigned int i=0,j=0,k=0;
    unsigned char value=0;
    for (i = 400; i <= 400; i++)
    {
        gather_image_data();
        rgbtogray(i,255);
        vCompressImg(i);
        perform_sobal(i);
        // Convert and Print ASCII ART
        vCnvPictoAscii(i);
        printf("ASCII Conversion Done\n");
void rgbtogray(unsigned int x, unsigned int max_val)
{
    unsigned char i=0;
    unsigned int j=0,d=0,k=0;
    float ctog[3]={0.3125,0.5625,0.125};
    unsigned int array_index=0,h=3,z=0;
    for(j=0;j<x;j++)
    {
        for(k=0;k<x;k++)
        {
            for(i=0;i<=2;i++)
            {
                shared[h] = ((shared[h])*(ctog[i]));
                d = (shared[h])+d;
                h++;
            }
            //printf(" ");
            img_data[z] = d;
            // printf("%d",img_data[z]);
            z++;
            d=0;
        }
        //printf("\n");
    }
}
void vCompressImg(unsigned int dimension)
{
    unsigned int array_index=0,k=0,array_limit=0,j=0;
    unsigned int i=0;
    array_limit = (dimension*dimension/4);
    k=dimension;
    for(j=0;j<dimension;)
    {
        for(j;j<k;j=j+2)
        {
            shared[h] = ((shared[h])*(ctog[i]));
            d = (shared[h])+d;
            h++;
            //printf(" ");
            img_data[z] = d;
            // printf("%d",img_data[z]);
            z++;
            d=0;
        }
        //printf("\n");
    }
cmp_array[array_index] = ((img_data[j]+img_data[j+1]+img_data[j+dimension]+img_data[j+dimension+1])/4);
    array_index++;
}
k = k+dimension*2;
j = j+dimension;
if(k>(dimension*dimension))
    break;
}
/*....SOBEL ALGORITHM STARTS HERE....*/

void perform_sobal(unsigned int dimensions)
{
    unsigned int dimension = 0;
    unsigned char *edgeptr = (unsigned char *)edge_img;
    unsigned int X=0,Y=0;
    int GX[3][3];
    int GY[3][3];
    unsigned int array_index=0;
    unsigned char *cmparray = (unsigned char *)cmp_array;
    int I, J, SUM;
    int sumX, sumY;
    char breaki;
    FILE *fp;
    fp = fopen("flagimg.txt","w");
dimension = dimensions/2;
for(Y=0; Y<=(dimension-1); Y++)
{
    for(X=0; X<=(dimension-1); X++)
    {
        sumX = 0;
        sumY = 0;

        // image boundaries
        if(Y==0 || Y==dimension-1)
{  
    SUM = 0;
}
else if (X==0 || X==dimension-1)
{
    SUM = 0;
}
// Convolution starts here
else
{
    //--------X GRADIENT APPROXIMATION--------
    sumX = sumX - (int)( *(cmparray + X -1 + (((Y - 1)*dimension))) );
    sumX = sumX + (int)( *(cmparray + X -1 + (((Y +1)*dimension))) );
    sumX = sumX - (int)( *(cmparray + X + (((Y -1)*dimension))) );
    sumX = sumX + (int)( (*(cmparray + X + (((Y+1)*dimension))))<< 1 );
    sumX = sumX - (int)( *(cmparray + X + (1) + (((Y -1)*dimension))) );
    sumX = sumX - (int)(*(cmparray + X + (1) + (((Y+1)*dimension))));
    //--------Y GRADIENT APPROXIMATION--------
    sumY = sumY + (int)( *(cmparray + X -1 + (((Y -1)*dimension))) );
    sumY = sumY + (int)((*(cmparray + X -1 + (((Y)*dimension))))<< 1);
    sumY = sumY + (int)(*(cmparray + X -1 + (((Y+1)*dimension))) );
    sumY = sumY - (int)((*(cmparray + X + (1) + (((Y -1)*dimension))) )<< 1);
    sumY = sumY - (int)( *(cmparray + X + (1) + (((Y+1)*dimension))));
    //---GRADIENT MAGNITUDE APPROXIMATION (Myler p.218)---
    SUM = abs(sumX) + abs(sumY);
}
if(SUM>255)
{ 
    SUM=255;
}
if(SUM<0) 
{
    SUM=0;
}
*(edgeptr+X+Y*dimension) = 255- (unsigned char)(SUM);
printf("%d ",*(edgeptr+X+Y*dimension));
fprintf(fp,"%d ",*(edgeptr+X+Y*dimension));
} fputc('
',fp);printf("\n");
}
}

void gather_image_data()
{
    unsigned int h=3; unsigned char i=0,breaki=0;
    FILE *fp;
    fp = fopen(filename,"r");
    if(fp==NULL) 
    
    
    {
        printf("Image does not exist\n");
    }
    else 
    
    
    {
        printf("Gathering Image %s\n",filename);
        while(!feof(fp))
        
        
        {
            fscanf(fp,"%d",&shared[h++]);
            if(feof(fp))
            
            
            {
                printf("\nYou have reached end of file\n");
            }
        } 
fclose(fp);
    }
}
void vCnvPictoAscii(unsigned int dimension)
{
    char symbols[16] = {'.','!','m','#','d','k','e','x','?','+','=',';','-','@','p'};
    unsigned int h=0;
    unsigned int dimensions = dimension/2;
    unsigned int i=0,j=0;
    FILE *fp,*fp1;
    fp = fopen("original.txt","w");
    fp1 = fopen("bootup.txt","w");
    fclose(fp1);
    printf("\n");
    for(h=0;h<dimensions*dimensions;h++)
    {
        edge_img[h] = symbols[(edge_img[h]%16)];
    }
    h=0;
    for(i=0;i<dimensions;i++)
    {
        for(j=0;j<dimensions;j++)
        {
            fprintf(fp, "%c ", edge_img[h]);
            printf("%c ", edge_img[h]);
            h++;
        }
        fputc('\n',fp);printf("\n");
    }
    fprintf(fp,"%s is Executed",filename);
    fclose(fp);
}