DETECTION OF BUGS IN MPI PROGRAMS USING A HYBRID APPROACH

By

SONAM SHERPA

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Engineering and Computer Science, Vancouver

DECEMBER 2020

To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of SONAM SHERPA find it satisfactory and recommend that it be accepted.

Xinghui Zhao, Ph.D., Chair

Xuechen Zhang, Ph.D.

Scott Wallace, Ph.D.

ACKNOWLEDGMENTS

First I'd like to express my sincere gratitude to my advisor Dr. Xinghui Zhao, for her guidance, patience, motivation and immense knowledge. Her guidance was instrumental to making my research and writing of this thesis a success. I could not have imagined having a better advisor and mentor for my Masters degree at Washington State University.

Besides my advisor I'd like to thank the rest of my thesis committee members: Dr. Xuechen Zhang and Dr. Scott Wallace for their suggestions that helped shape the research and resources that helped evaluate the proposed solution.

I thank my fellow colleagues for the simulating discussions, for providing me with there precious time, for the sleepless nights we had as we tried to meet deadlines and for the fun we had in the last two years.

Lastly, I'd like to thank my parents and siblings who have been instrumental in ensuring the thesis was a success through their encouragement, unending inspiration and support every step of the way.

DETECTION OF BUGS IN MPI PROGRAMS USING A HYBRID APPROACH

Abstract

by Sonam Sherpa, M.S.
Washington State University
December 2020

Chair: Xinghui Zhao

MPI bugs are difficult to diagnose and fix. Traditional approaches for diagnosing MPI bugs attempt to reproduce the exact execution schedule which reveals the bug, resulting in high run-time overhead. In this thesis, we present our work in identifying bugs in MPI programs using a hybrid approach that leverages both static and dynamic analysis. The static approach is based on the LLVM IR (Intermediate Representation) file that we generate using the LLVM compiler [1]. Once we retrieve the LLVM IR we then parse the IR to gather information to trace the buggy code regions. For the dynamic analysis we have two approaches (Machine Learning-based approach and Rule-based approach). For the machine learning-based approach, we observed that resource access and consumption patterns are critical indications of the run-time behavior of concurrent software, and can be used as a powerful mechanism to guide the software debugging process. In our process to build a dynamic approach we first monitored the resource footprints at run-time and found that it can effectively help detect software bugs. Specifically, for MPI programs, a simple SVM classifier can detect deadlocks with high accuracy using only the CPU usage patterns. Therefore, in our dynamic analysis, we explore ways to detect bugs using resource consumption patterns.

Whereas, for the rule-based approach we do the run-time analysis of each of the children's functions and the main function as well, by the aid of the profiling tool in-order to understand the measure performance bugs. For this approach we take user-defined rules such as how much time a particular function is to run for (lower bound and upper bound) and how many times it should be called, and based on this rules, we compare the rules with the actual run-time and give result whether the rules passed or fail. To evaluate our approach, we have carried out experiments to detect various bugs in different benchmarks such as CombBLAS, OpenFFT, and NAS Parallel benchmarks [2–4]. We were able to detect both the deadlock [5–8] and performance bugs [9–13] using our approach.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS iii

ABSTRACT iv

LIST OF TABLES viii

LIST OF FIGURES ix

1 INTRODUCTION 1

2 RELATED WORK 7

1 Static Analysis 7

2 Dynamic Analysis 8

3 Hybrid Analysis 10

3 SYSTEM DESIGN 12

1 Static Approach Design and Implementation 12

1.1 Type Mismatch 14

1.2 Unreachable Call Check 14

1.3 Unmatched Point To Point Call Check 15

2 Dynamic Approach Design And Implementation 18

2.1 Machine Learning-Based Approach 18

2.2 Rule Based-Approach 19

4 EVALUATION 23

1 Experimental System 23

1.1 Static Approach 23

1.2 Dynamic Analysis 25

5 CONCLUSION AND FUTURE WORK 31

Bibliography 37

List of Tables

2.1 Comparison of existing tools. 11

3.1 Rule Book where the user sets the rules. 21

3.2 Output displaying the result. 21

3.3 Static and Dynamic Bug Detection with cause and effects. 21

3.4 Different Static and Dynamic techniques for Bug Detection: 22

4.1 Run-time analysis of bug-free NPB program with different problem size: . . 26

4.2 Run-time analysis of NPB program with different problem size after bug injection: 26

4.3 Deadlock Detection Accuracy 30

5.1 Run-time analysis of different NPB benchmarks written in FORTRAN: . . . 32

List of Figures

1.1 Static approach for bug detection in MPI programs. 3

1.2 Dynamic approach for bug detection in MPI programs. 6

3.1 Type Mismatch 14

3.2 Unreachable Call Check 15

3.3 Unmatched Point To Point Call Check 16

3.4 Static Approach Work Flow 17

3.5 Machine Learning-based approach. 19

4.1 Type Mismatch 24

4.2 Bug Injection 24

4.3 Deadlock Injection 27

4.4 Resource Consumption Traces 28

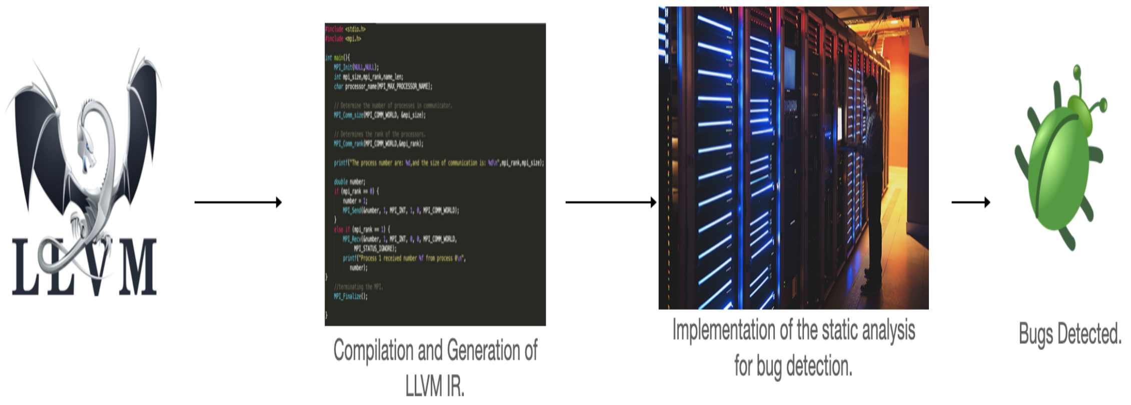
Chapter 1

INTRODUCTION

Recent advances in hardware architectures, particularly multicore and manycore systems, implicitly require programmers to write concurrent programs that can be executed in parallel on multiple cores [14]. However, writing correct and efficient concurrent programs is challenging, and programmers make mistakes easily when writing concurrent programs. In addition, the notorious non-determinism of concurrent programs makes it difficult to diagnose and fix concurrency bugs. For decades, concurrency bug detection has been a challenging problem in the software community. Existing work has been using both static and run-time approaches to address this problem, such as code analysis and thread interleaving monitoring [15–17]. Most of the existing work only utilizes either the static approach [18–21] that detects bugs by searching specific performance anti-patterns in software, such as inefficient call sequences or loop patterns; or only the dynamic approach [8, 22–27] that closely monitors run time application behaviors to infer root causes of performance problems in detecting of the bugs in the concurrent programs. Both approaches have their advantages and limitations.

Concurrent programs can be written using different techniques. In our work, we focus on MPI programs, which are widely used in HPC systems. The MPI [28] is a standard form of a library specification for passing data between processes. Concrete implementations of this specification make it possible to distribute a computational task across nodes that communicate to collaboratively find a solution. Communication schema used in message passing can easily get complex and hard to survey. Even for a simple schema, a flaw can be introduced by overlooking a tiny detail. It is quite hard to deal with this sort of error, especially when the line of code increases. Also if we run these buggy programs and try to figure out the problem it might be hard to detect, as sometimes the program may compile and run normally without showing errors and provide us with results that we are not expecting of. One of the threats of MPI programs is the bugs that are hard to trace or detect. In this thesis, we present our work in identifying concurrency bugs in MPI programs using both the static (LLVM Compiler) [1] and dynamic (run-time analysis) approach. The static approach is the analysis of the program without actually executing programs. Whereas, dynamic approach is the analysis performed on programs while they are executing.

Static Approach: In the static analysis, we aim to detect bugs at compile-time, without having to read the source code. The static approach presented in the thesis is based on the LLVM IR file generated using the LLVM compiler [1]. The LLVM compiler is a set of compiler and toolchain technologies, which can be used to develop a front end for any programming language and a back end for any instruction set architecture. LLVM is designed around a language-independent intermediate representation (IR) that serves as a portable, high-level assembly language that can be optimized with a variety of transformations over multiple passes. LLVM is written in C++ and is designed for compile-time, link-time, run-time, and "idle-time" optimization. Through this approach, we present a feasible solution for bug detection by analyzing the LLVM IR (Intermediate Representation) of the source code. The main advantage of having such a bug detection technique is that the system run-time dependency and code dependency is reduced. As in the case of LLVM IR, we depend on the IR representation of the source code rather than depending on the actual code. This technique can be useful in the scenario where we do not have much information (e.g, no error log, core dump). For this approach, we build checkers that help us in detecting the root cause of bugs without running the program. The checkers consists of rules for detecting bugs. One of the main reason why we want to develop these methods to find errors is that even before running the program (during compile time), we would like to identify the cause of the bug, which in return would save us the system resources and time as we would not be running the buggy program. As in the case of HPC, running such a bug might cause a huge loss as it may fail to provide us results with proper accuracy, or in the worst scenario, it may hang the whole system. We may also generate the LLVM IR from the binary code of the program generated during the compile time. Once we retrieve the LLVM IR we then parse the IR to gather information to trace the buggy code regions. The buggy code regions are identified by the aid of the rules, each rule consists of checkers that identify the bugs in the program. Figure 1.1 shows the steps followed in a static approach for bug detection in MPI programs.


LLVM
Compilation and Generation of LLVM IR.
Implementation of the static analysis for bug detection.
Bugs Detected.

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

int main(){
    MPI_Init(NULL,NULL);
    int mpi_size,mpi_rank,name_len;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    // Determine the number of processes in communicator.
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

    // Determines the rank of the processors.
    MPI_Comm_rank(MPI_COMM_WORLD,&mpi_rank);

    printf("The process number are: %d,and the size of communication is: %d\n",mpi_rank,mpi_size);

    double number;
    if (mpi_rank == 0) {
        number = 1;
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }
    else if (mpi_rank == 1) {
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        printf("Process 1 received number %f from process 0\n",
          number);
}
    //terminating the MPI.
    MPI_Finalize();

}
```

Figure 1.1: Static approach for bug detection in MPI programs.

Dynamic Approach: In the dynamic approach, we detect bugs based on two approaches that are Machine Learning and Rule-based approach. For the Machine Learning approach, we record the pattern of resources consumption footprint of the system during the run-time of the program to detect bugs. Later the recorded pattern of resources consumption footprint is fed to the machine learning module to detect the buggy pattern. This approach is useful to detect bugs in the system which have stable workflow and known pattern from the past [13, 29]. It can also be beneficial in case of detecting bugs in big servers and data centers, where we could mine system logs [30–33] to find similar issues from the past. For the Rule-based approach, we analyze the run-time of each function calls in-order to trace the buggy functions. For this, we set up conditions such as for how long the program functions are supposed to run for if called for a particular number of times. We set up the upper bound and lower bound range for the run-time of the functions that need to be analyzed. We analyze the actual run-time of each function in the program by the aid of profiling tools and compare them with the conditions we had setup. Depending on the comparison between the conditions and the actual run-time result we pass or fail the condition. Our dynamic approach is very user friendly and interactive. It provides the user or developer an interface to set up their own set of conditions or rules to detect bugs. In this approach, we assume that the users or developers have some domain knowledge about the run time of the program, so that they may set up their own rules for detecting bugs. Figure 1.2 shows the steps followed in a dynamic approach for bug detection in MPI programs.

Thesis Contribution: In this section, we present our contributions to bug detection in MPI programs. Most of the existing tools for bug detection in MPI are either static or dynamic. Both Static and Dynamic approaches have its ways of bug detection. The bugs detected by using only Static may not be able to detect bugs detected by using a dynamic approach and vice-versa. Keeping it in mind we build a hybrid approach for bug detection in MPI programs. The hybrid approach that we developed provides a better way to detect bugs in MPI programs. Firstly, it is beneficial in case of detecting bugs in systems which have stable workflow and known bug pattern from the past. It also provides the user or developer with the flexibility to generate their own set of rules to detect bugs assuming that the user has domain knowledge about the run-time of the program. Moreover, it also provides a feature to detect bugs at the compile-time as well. Most of the existing static approach directly depends on the source code of the program, but in our case, the static approach relies on the LLVM IR.

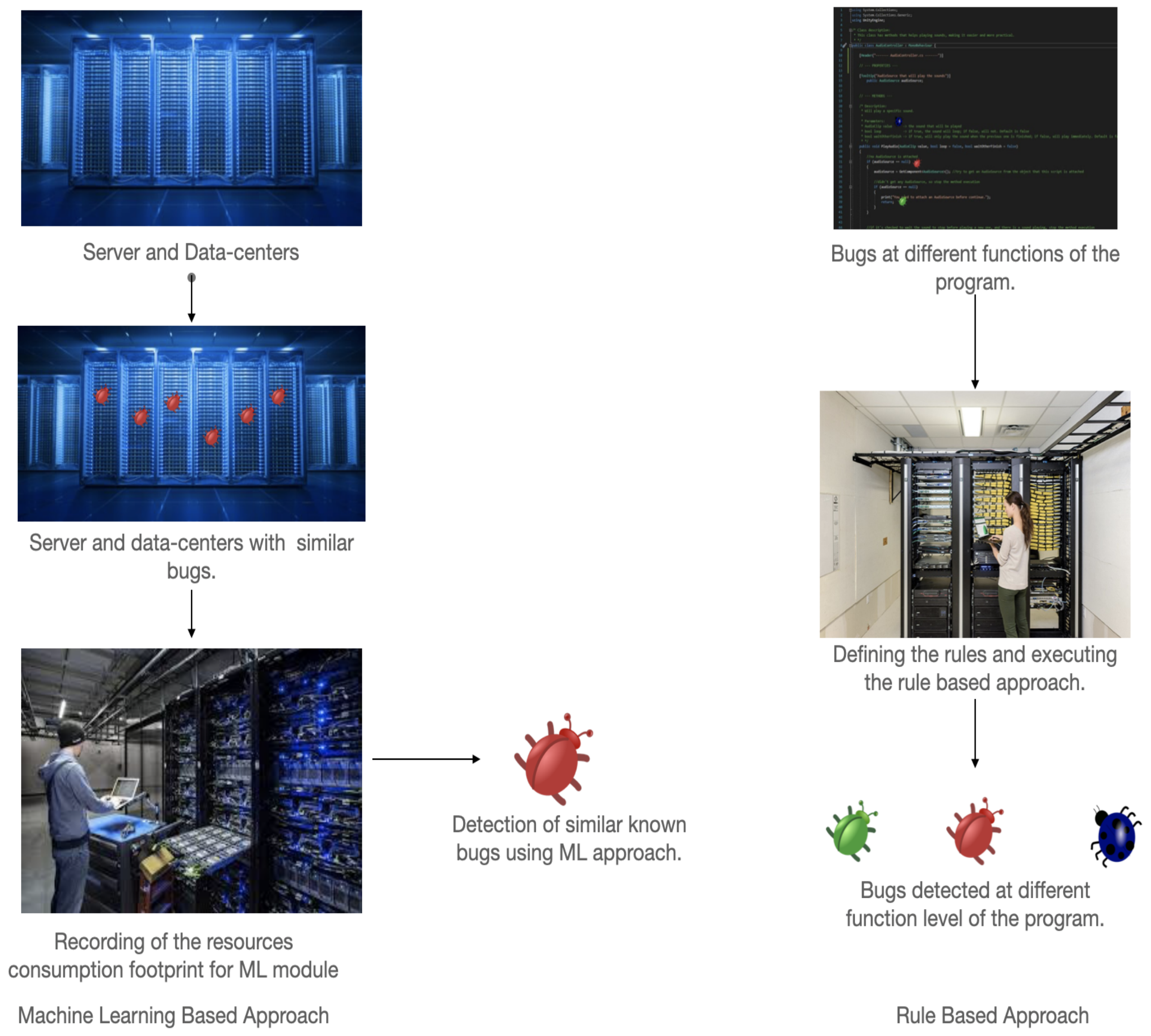

Server and Data-centers
Server and data-centers with similar bugs.
Recording of the resources consumption footprint for ML module
Detection of similar known bugs using ML approach.
Machine Learning Based Approach
Bugs at different functions of the program.
Defining the rules and executing the rule based approach.
Bugs detected at different function level of the program.
Rule Based Approach

Figure 1.2: Dynamic approach for bug detection in MPI programs.

Chapter 2

RELATED WORK

Bug detection in the MPI program can be achieved by using either the Static method or Dynamic method. In this section, we discuss related work on different static and dynamic methods used to detect bugs in MPI programs.

1 Static Analysis

The static analysis does the correction of the code based on static patterns of the source code. They do not consider any sort of run time evaluation. MPI checker [18] is a static bug detection tool for MPI programs. Static approaches have unique benefit over dynamic approaches, as they do not waste system resources as the correction or analysis of the program is done even before running the program. Moreover, in the case of a large parallel system, the run time analysis can be hectic as it seems to contain lots of synchronization among processes which might consume lots of time. The computational cost of static analysis on the other hand is unrelated to the number of processes and does not depend on results calculated at run time. Bugs are caught at compile time. The difference in the approach that we take is we do not directly analyze the code instead we emit the LLVM IR of the code by converting the bitcode of the program using the Clang LLVM compiler [1]. The reason behind doing so is that server applications running inside production cloud infrastructures are prone to various bugs (eg deadlock, performance slowdown). When those problems occur, developers often have little clue to diagnosing those problems. But if we use our approach we do not require the source code and we may also apply our method of static analysis on both compiled and interpreted programs such as C/C++. LLVM IR is similar to that of assembly language and it provides us with much information to find bugs statically. MPI Check is another existing tool that does static analysis of bugs in MPI but it only aids in detecting bugs in MPI programs that are written in free or fixed format of FORTRAN 90 and FORTRAN 77. Whereas using our hybrid approach we can detect bugs for MPI programs written in any language. MPI-CHECK [34] is restricted to FORTRAN code and performs argument type checking or find problems like deadlocks. The disadvantages of these tools are that they are limited to special platforms or language bindings.

2 Dynamic Analysis

Dynamic analysis is the testing and evaluation of a program by executing data in runtime. The objective is to find errors in a program while it is running, rather than by repeatedly examining the code offline. In PFix [23] the bugs are correlated with the memory access patterns. It aids in most of the cases in detecting the root cause of the bugs. The idea is to identify a well-designed locking which makes the failure-inducing memory-access patterns impossible. But what makes our approach different from it is that our approach looks into every aspect, but PFix only focuses more on locking policies and does seem to avoid small issues such as what if there are type mismatch scenario. But in the case of the dynamic analysis performed by us, this is not the case it looks into a scenario such as execution time of each of the functions along with its function calls and provides the user with information about which particular function call could be the cause of error or performance issue. While for known types of bugs from the past it records the resources consumption pattern and aids in bug detection. ParaStack [22] is a parallel tool based on a stack trace that judges a hang by detecting dynamic manifestation of the following pattern of behavior –persistent existence of very few processes outside of MPI calls. Since processes iterate between computation and communication phases, a persistent dynamic variation of the count of processes outside of MPI calls indicates a healthy running state while a continuous small count of processes outside MPI calls strongly indicates the onset of a hang. Based on execution history, ParaStack builds a runtime model of count that is robust even with limited history information and uses it to evaluate the likelihood of continuously observing a small count. A hang is verified if the likelihood of persistent small count is significantly high. The approach differs from our approach as para stack heavily depends on MPI in and MPI out time which can cause a lot of issues if we don't have the exact information also they are only able to trace only deadlock scenario but in our case, we not only look at deadlock but also into performance bugs caused in the program. Moreover, in case of Parastack, it also seems that we need to run the program and analyze at each step the MPI calls, which can be bit tedious and hard to do every time as a large parallel program may have many MPI calls and different scenarios that lead to cause problems, and para stack being heavily dependent on runtime can waste time and resources but instead what we do in case of our approach is that we would analyze the program runtime and provide the user with feasibility to set up their own rules for detecting bugs. Umpire [35] is a tool that monitors the MPI operations of an application by interposing itself between the application and the MPI runtime system using the MPI profiling layer. Umpire then checks the application's MPI behavior for specific errors. Umpire is limited to shared memory platforms. Performance-wise Umpire takes much time to detect bugs. It would also slow down the applications. While our approach does not lower the application performance as it gathers information that is only required by users. Our approach provides the user to set up rules on which specific part of the application function do they want to test, as a result, our approach does not slow down the application or the program.

3 Hybrid Analysis

The HYTRACE [36] is a hybrid approach which uses both the approach of having static analysis and dynamic analysis. But what makes our hybrid analysis better than the HYTRACE is the fact that we do not only focus on certain program bugs but we look into MPI bugs and bugs related to all the other programs as our dynamic analysis provides the freedom for the user to set rules of their own to check bugs. Moreover, HYTRACE depends on the conformation of both dynamic and static analysis to detect or approve that the code is buggy. This usually is because their static analysis is not precise enough to detect the bugs during the compile-time period. Whereas in our case, we believe that our approach can precisely detect what is the root cause or what may be the cause of the buggy scenario of the application. Our approach is a mixer of the state of the art of the MPI checker and Hytrace, which seems to overcome the drawbacks of both by providing more precise results. Our approach also records the past bugs resources consumption, which provides an easy bug fix of the known bugs from the past if encountered in the future.

Table 2.1: Comparison of existing tools.

Tools	MPI_Checker	PFix	Parastack	HYTRACE	Our Tool
Static	✓	×	×	✓	✓
Dynamic	×	✓	✓	✓	✓
User defined rules	×	×	×	×	✓
Language Independency	×	×	×	×	✓

The table 2.1 shows the comparison of the existing tools with our tool.

Chapter 3

SYSTEM DESIGN

The Message Passing Interface (MPI) [28] is commonly used to write parallel programs for distributed memory parallel computers. One of the threats of parallel programs is the bugs that are hard to trace or detect. In high-performance computing, clusters are built by interconnecting a large number of computing nodes to aggregate computing power. Based on this hardware framework, a model for parallel programming called message passing evolved which realizes communication between nodes. During the communication between processes in MPI programs, we encounter bugs. In this section, we present the system design of the Hybrid (Static and Dynamic) approach developed by us for detecting bugs in MPI programs.

1 Static Approach Design and Implementation

For the static approach, we rely on the LLVM IR generated by the LLVM compiler and rules that help us in detecting the root cause of buggy issues without running the program. One of the main reasons why we want to develop this method to find errors is that even before running the program, we would like to identify the cause of the bug, which in return would save us the system resources and time as we would not be running the bugged program. As in the case of HPC, running such a bug might cause a huge loss as it may fail to provide us results with proper accuracy, or in the worst scenario, it may hang the whole system. Cloud computing infrastructures have become increasingly popular by allowing users to access computing resources cost-effectively. In such cases, developers often have a little clue (such as error log, core dump report, etc.) to localize or to diagnose those problems. The other factor that guided us to perform such a method was, what if the server applications running inside production cloud infrastructures are prone to such various performance problems (Software hang, Performance issues, etc.). The method described does not require source code and can be applied to both compiled and interpreted programs such as C/C++.

LLVM [1] is a compiler infrastructure that allows developers to examine code as it's being compiled. We have implemented the LLVM compiler to our static analysis. LLVM can provide the middle layers of a complete compiler system, taking intermediate representation (IR) code from a compiler and emitting an optimized IR. This new IR can then be converted and linked into machine-dependent assembly language code for a target platform. LLVM can accept the IR from the GNU Compiler Collection (GCC) toolchain, allowing it to be used with a wide array of extant compilers. LLVM can also generate relocatable machine code at compile-time or link-time or even binary machine code at run-time. Using the LLVM compiler we can also generate the LLVM IR even from the binary code of source code. Depending on the availability of the source code or binary code, the approach firstly converts the source code or the bit code of the source code into an optimized IR. After the IR is generated we implement our checkers on the LLVM IR to check for bugs. Below we present the implementation of rules on LLVM IR to detect bugs.

1.1 Type Mismatch

For the data to be transferred between the processes in MPI we need to always define an MPI_datatype. Using the MPI datatype tags, which enable communication between distinct processors which run on machines that have different memory representations of a type. If there is a mismatch between the actual data type of the buffer and MPI data type it would lead to some serious miscalculation or data transfer. All datatype and MPI_Datatype are represented by special numbers and characters in the IR representation. The rule Type Mismatch checks through the IR of the source code and detects if there are any buffer data type and MPI data type mismatch while we do send and receive between processes.If any mismatch is detected the rule notifies the user about it.

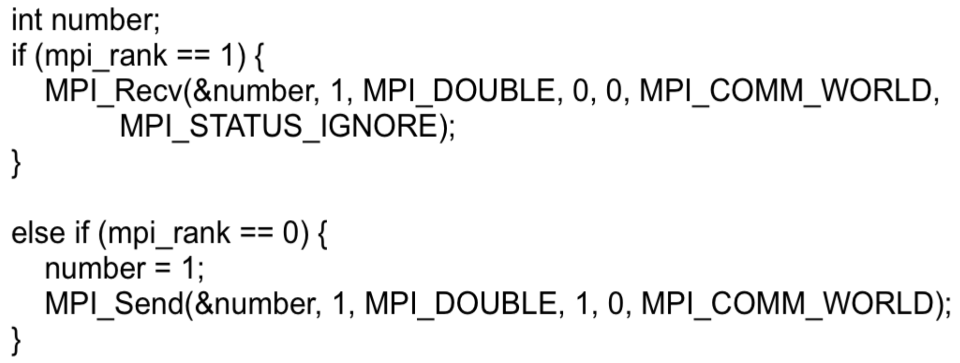
```
int number;

if (mpi_rank == 0) {
   number = 1;
   MPI_Send(&number, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
}

else if (mpi_rank == 1) {
   MPI_Recv(&number, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
          MPI_STATUS_IGNORE);
}
```

Figure 3.1: Type Mismatch

Figure 3.1 shows how a type mismatch looks like. In the figure, we can see a buffer number with datatype integer, but when a send and receive is done in MPI, the MPI_datatype and buffer data type does not match which leads towards the transfer of incorrect value in MPI.

1.2 Unreachable Call Check

The unreachable calls can lead to the deadlock scenario in the case of MPI. In the unreachable call check rule, we trace for cases where send calls are being not acknowledged by receive calls. For this rule to function, we first scan through the LLVM IR and check if the send and receive calls are equal in numbers, if there are an equal number of sends and receive calls made. The next factor we look into is whether the send calls are made before receive calls or not. As in case of this calls the first operation needs to be a send operation whereas the second one must be a receive operation. When a send operation is matched with a reception, the operation is marked and the algorithm continues to search a matching receive for the next send. If in case there is no matching receive and send calls or if the receive call was made before sending call we discontinue searching and report the user or developer about the issue, which is whether there was uneven send and receive calls made or a receive call was being made before sending the call.



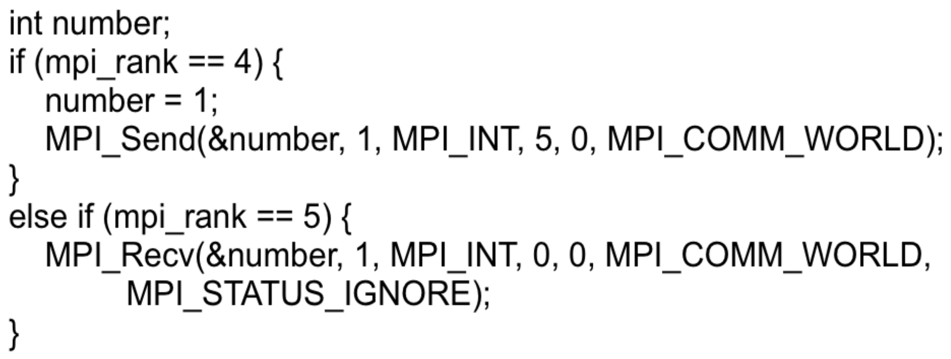
```
int number;
if (mpi_rank == 1) {
   MPI_Recv(&number, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
          MPI_STATUS_IGNORE);
}

else if (mpi_rank == 0) {
   number = 1;
   MPI_Send(&number, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
}
```

Figure 3.2: Unreachable Call Check

Figure 3.2 shows how an unreachable call check could occur in an MPI program. Here we can observe that the receive call is made before a send call which leads towards the deadlock scenario.

1.3 Unmatched Point To Point Call Check

The unmatched point to point call rule aids us in tracing which process is sending and which process is in the receiving end. It validates if any point to point operations are without a matching partner. This is the case if a send call has no matching receive call. The algorithm constructed for this rule first searches for a send call being done from a process, after that it searches for its matching receive call. If the match is not found then we report it to the user or developer that there was no matching receive call for a send call.



```
int number;
if (mpi_rank == 4) {
   number = 1;
   MPI_Send(&number, 1, MPI_INT, 5, 0, MPI_COMM_WORLD);
}
else if (mpi_rank == 5) {
   MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
}
```

Figure 3.3: Unmatched Point To Point Call Check

Figure 3.3 shows how an unmatched point to point call check may occur. In the above figure, process 4 is trying to send the value of the buffer to process 5 but process 5 is waiting to receive buffer value from process 0 which results in a deadlock scenario.

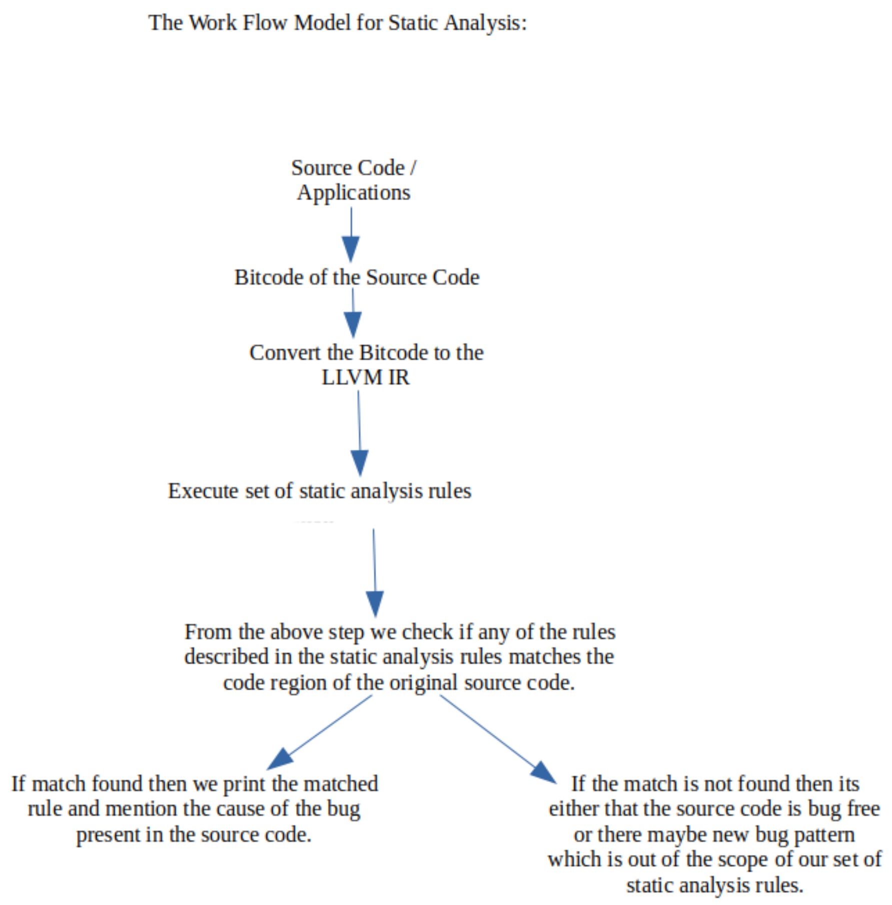

The Work Flow Model for Static Analysis:
Source Code / Applications
Bitcode of the Source Code
Convert the Bitcode to the LLVM IR
Execute set of static analysis rules
From the above step we check if any of the rules described in the static analysis rules matches the code region of the original source code.
If match found then we print the matched rule and mention the cause of the bug present in the source code.
If the match is not found then its either that the source code is bug free or there maybe new bug pattern which is out of the scope of our set of static analysis rules.

Figure 3.4: Static Approach Work Flow

Figure 3.4 shows how our Static Analysis approach works. It represents the overall workflow of our static approach. Firstly, we compile the source code of the program using the LLVM compiler, which generates the bitcode of the source code. After doing so we convert the bitcode to the LLVM IR in which we apply the static analysis rules to trace buggy regions. Once the buggy region is detected we inform the user about the type of bugs detected while performing the static analysis.

2 Dynamic Approach Design And Implementation

The Dynamic approach presented here is based on two approaches Machine Learning-based approach and rule-based approach. The Machine learning-based approach is useful for the servers and data centers that have stable workflow and known bugs pattern from the past. While the rule-based approach is useful in cases where the user has more domain knowledge about the run-time of the programs and application.

2.1 Machine Learning-Based Approach

In this section, we present the machine learning-based approach. The dynamic analysis for this approach is based on a resource consumption footprint. MPI is widely used in the high performance computing community. One of the common bugs in MPI programs is a deadlock, due to the fact that several usage patterns of MPI can lead to this type of bug. Therefore, we choose deadlocks in MPI programs to demonstrate how resource consumption traces [37,38] can be used to identify bugs at run-time. To explore the potentials of detecting deadlocks using run-time resource consumption patterns, we train various machine learning models [12, 39] using a dataset containing both normal and deadlocked executions. The dataset is obtained as follows: 1) we trigger the system profiling tool; 2) we execute the MPI program, and record the run-time resource consumption in a file; 3) we preprocess the data file and use three main parameters: process id (PID), CPU usage in percentage (%CPU), and Timestamp. Using this method, we collect 200 execution traces, among which 100 are normal executions, and another 100 are executions with deadlocks. These traces form the dataset we use to train and test the machine learning models. To investigate the correlations between resource consumption patterns and the deadlock, we choose two different machine learning methods, SVM and K-nearest neighbors, both are widely used for supervised classification purposes. For training, we use 80% of the data and use the rest of 20% for testing.

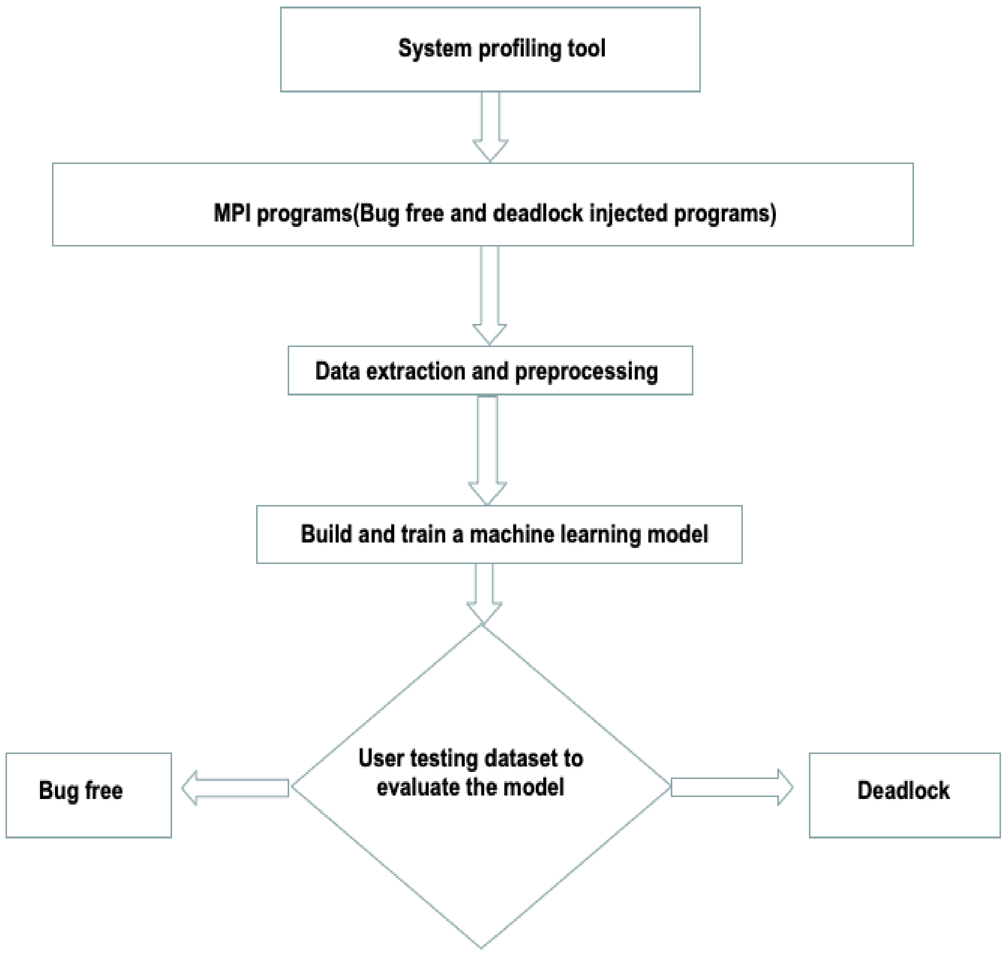

System profiling tool
MPI programs(Bug free and deadlock injected programs)
Data extraction and preprocessing
Build and train a machine learning model
User testing dataset to evaluate the model
Bug free
Deadlock

Figure 3.5: Machine Learning-based approach.

The figure 3.5 represents the steps in the machine learning-based approach. Firstly we run the system profiling tool to trace the system resources consumption during the execution of MPI programs(bug-free and bug injected program). After recording the data we preprocess it to build and train the machine learning model. Later we use the trained model to identify similar known bugs from the past.

2.2 Rule Based-Approach

In this section, we discuss the other part of the dynamic approach which is rule based-approach. In the rule based-approach, we trace the execution time [22,38] and function calls made in the program. The dynamic analysis we present here aids in the tracing of MPI bugs that exist in the source code, which could sometime be failed to be understood by the user or developer. We trace the execution time and each function calls of the user through the profiling tool. Profiling allows us to investigate both the execution trace in terms of both the execution time and function calls. This information can show us which pieces of our program are slower than we expected, and might be candidates for rewriting to make our program execute faster. It can also tell us which functions are being called more or less often than we expected. This may help us spot bugs that had otherwise been unnoticed.

For tracing the MPI bug using the above approach the algorithm works as follows. The user or developer first sets the rules on the rule book text file (The rule book consists of which function to check for, the min and max range of the function execution time and the number of times a particular function being called for.) about how long or how precisely do they believe that the particular function should execute for, given a particular number of times the function is being called in the program. After doing so we execute the program, along with the profiling tool, and compare the rule set up by the user with the data extracted using the profiling tool. Based on the comparison between the rule set up by the user and the data extracted from the profiling tool we get output for the rules. The output would consist of whether the rules passed or failed. Above we present two tables one representing the rule book file 3.1 and the other representing the output 3.2. The field Function_name represents the name of the function which the user wants to test. The Min and Max run-time is used to set up the range of execution period of the particular function. The run-time is measured in seconds. The calls field defines how many times a function has been called. Lastly, the result column declares whether there are any differences between the user and the actual run-time range. On this basis, the user gets notified of whether the rule passed or failed.

Table 3.1: Rule Book where the user sets the rules.

Serial No.	Function_Name	Min_run-time	Max_run-time	Calls
1	main	0.00	0.00	0
2	Pro5	2	3	1
3	dboard	0.02	0.08	100
4	sumup	0.00	0.05	1

Table 3.2: Output displaying the result.

Serial No.	Function_Name	Min_run-time	Max_run-time	Calls	Result
1	main	0.00	0.00	0	Passed
2	Pro5	2	3	1	Passed
3	dboard	0.02	0.08	100	Passed
4	sumup	0.00	0.07	2	Failed

Table 3.3: Static and Dynamic Bug Detection with cause and effects.

Serial No.	Static/Dynamic	Bug Type	Root Cause	Effects
1	Static	TypeMismatch	Buffer_DataType and MPI_Datatype mismatch.	Transfer of Wrong data between processes.
2	Static	Unreachable call check	A receive call is made before send call.	Hang
3	Static	Unmatched point to point call check	A send call has no matching receive call	Hang
4	Dynamic	Deadlock	Incorrect use of send and receive calls.	Hang
5	Dynamic	Performance	Use of infinite loops or unnecessary funtion calls.	Execution Time increases along with system resources use.

Table 3.4: Different Static and Dynamic techniques for Bug Detection:

Serial No.	Static/Dynamic	Bug Type	Approach
1	Static	TypeMismatch	Checks for the mismatch between Buffer_DataType and MPI_Datatype.
2	Static	Unreachable call check	Checks if send calls are made before receive calls or not.
3	Static	Unreachable call check	Validates if any point to point operations are without a matching partner.
4	Dynamic	Deadlock	Traces the CPU usage during the run-time of the program which aids in deadlock detection.
5	Dynamic	Performance	Traces the run-time and function calls of each function and compares it with the user defined rules(run-time,function call) to check if there is any performance bug at a function level.

Chapter 4

EVALUATION

1 Experimental System

In this section, we present the use of our hybrid approach to detecting bugs in the MPI programs through a series of experiments. The experimental system and experimental results are explained, respectively.

1.1 Static Approach

For the bug detection in the MPI program using the static approach, we tested the idea onto benchmarks such as CombBLAS (The Combinatorial BLAS) [3] and OpenFFT [4]. CombBLAS is an extensible distributed-memory parallel graph library offering a small but powerful set of linear algebra primitives specifically targeting graph analytics. While OpenFFT is an open-source parallel package for computing multi-dimensional Fast Fourier Transforms (3-D and 4-D FFTs) of both real and complex numbers of arbitrary input size. In the case of the **CombBLAS** we were able to trace bug related to Type mismatch. At first, when we compiled the program, it compiled completely fine. But then when we diagnosed the program with the help of our static approach rules, we found out that instead of the MPI LONG LONG the MPI datatype tag MPI INT64 T should have been used.

```
int64_t * localcounts = new int64_t[nprocs];
localcounts[rank] = totrecv;
MPI_Allgather(MPI_IN_PLACE, 1, MPI_LONG_LONG, localcounts, 1, MPI_LONG_LONG, World);
int64_t glenuntil = std::accumulate(localcounts, localcounts+rank, static_cast<int64_t>(0));
```

Figure 4.1: Type Mismatch

The figure 4.1 displays the MPI bug. The bug that got detected was, the buffer has an int64_t datatype which when converted to MPI_datatype should be (MPI_INT64_T), and in LLVM IR the code name for it is 1275070522 but when we compile the program and run the rule, instead of getting 1275070522 we see that the LLVM IR check detects 1275070473 which is an IR code for MPI_LONG_LONG, hence there was a type mismatch.

For the second experiment, we tested our static approach in the **OpenFFT** benchmark. For OpenFFT no MPI related bugs were found. We followed the same procedure for the OpenFFT test case as well. Since no bugs were detected in the OpenFFT, we decided to inject some of the known bugs into it and explore if we could detect the injected bugs. Below we present the types of bugs injected into the OpenFFT benchmark.

```
int Num_Snd_Grid_CA2CB;

if(MPI_Comm_rank == 10){

     MPI_Send(&Num_Snd_Grid_CA2CB,1,MPI_INT,12,0,mpi_comm_level1);
  }

elseif(MPI_Comm_rank == 12){

     MPI_Recv(&Num_Snd_Grid_CA2CB,1,MPI_INT,14,0,mpi_comm_level1);
  }
```

Figure 4.2: Bug Injection

In figure 4.2 we present the bug injected to the OpenFFT benchmark. From the above bug, we can see that process 10 is trying to send the buffer value of Num_Snd_Grid_CA2CB to process 12 with a tag of 0. Whereas process 12 is expecting to receive the value of the buffer Num_Snd_Grid_CA2CB from process 14 which does not even exist as a result a deadlock scenario is detected, as both the sending process and receiving process do not have a matching send or receive calls.

For the detection of the bug in both the benchmark tool, the algorithm works as follow: 1) First we compile the program using the LLVM compiler (The LLVM compiler provides us with the bit code of the program, which is generated based on the compiled source code.) 2) On the next step, we generate the LLVM IR (Intermediate representation of the source code.) through the bit code of the program. The LLVM IR is similar to that of the assembly language. 3) Once the LLVM IR is generated we scan for the buggy code region in the LLVM IR of the source code by executing our rules for bug detection. 4) After execution of the rules for bug detection, if any sort of bug is detected, it gets reported to the developer, or a flag is raised to inform about which bug was detected during the bug scanning. The only difference between the CombBLAS and the OpenFFT benchmark bug detection testing was the addition of the stage of bug injection. Since no bugs were detected in the OpenFFT benchmark, we had to self inject the bug into the OpenFFT benchmark. Figure 4.2 represents the bug we injected into the OpenFFT benchmark. The effect of the bug that we injected to the OpenFFT benchmark was, a deadlock scenario was created as both the processes were not able to acknowledge the send or receive calls.

1.2 Dynamic Analysis

For the dynamic analysis, we tested our tool on NAS Parallel Benchmarks [2]. The NAS Parallel Benchmarks (NPB) are a small set of programs designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications and consist of kernels and pseudo-applications in the original "pencil-and-paper" specification. Problem sizes in NPB are predefined and indicated

Table 4.1: Run-time analysis of bug-free NPB program with different problem size:

Min-Runtime	Max-Runtime	Size	Class
0.75	1	8388608	A
3	4	33554432	B

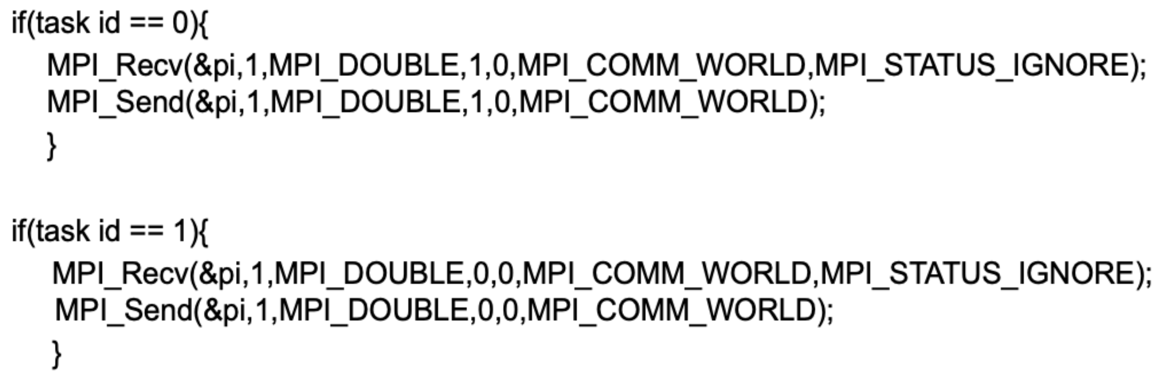
Table 4.2: Run-time analysis of NPB program with different problem size after bug injection:

Bug-injected	Min-Runtime	Max-Runtime	Size	Class
Type mismatch	0	0	8388608	A
Type mismatch	0	0	33554432	B
Performance bug	1.03	1.53	8388608	A
Performance bug	4.12	4.64	33554432	B

as different classes. Since no bugs were detected in the NAS Parallel Benchmark, we injected bugs to it. For the problem size of the NAS Parallel Benchmark, we used two classes problem size (A and B). Before injecting the bug to the benchmark we traced the execution time of the program. The traced range of the execution time of the program was provided to the rule book. Then in the next step, we injected the bug to the program and compared the execution time with the bug-free program execution time. Two types of bugs were injected into the MPI program. The first bug we injected was the type mismatch bug and the second was the performance bug. As we ran our dynamic analysis to test for the detection of the bug in both the scenario, we were able to detect the bug. The maximum and minimum range was traced for all the testing process.

In the Table 4.1 we present the bug free NPB program exectuion time. All the run time are measured in seconds. Here we have two problem size of class A and B. The size column denotes the actual size of the problem. In the Table 4.2 we present the run time of the NPB program after injecting bug to it. It is clearly visible from both the Table that there is an effect in the run time of the program before and after the bug is injected.

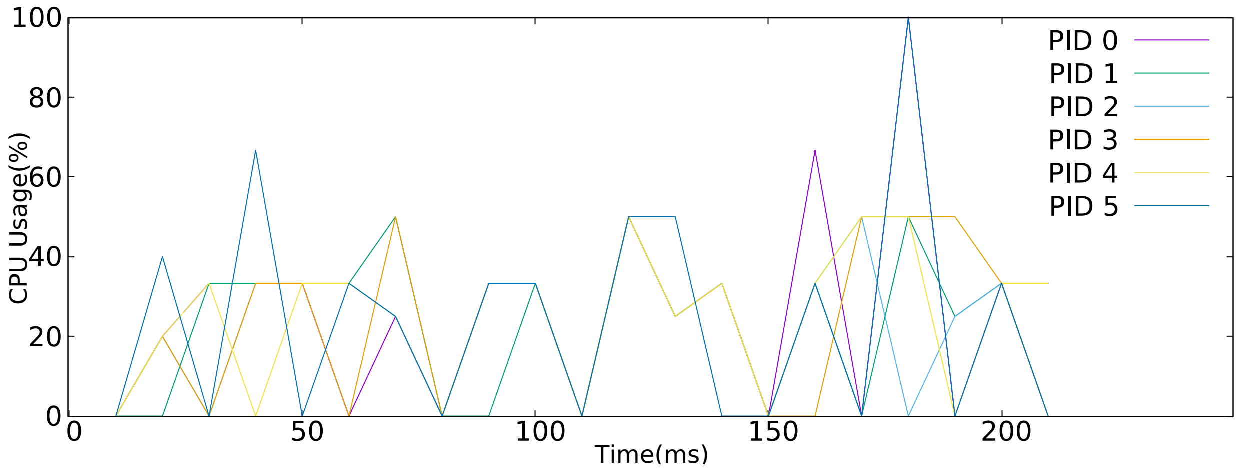
Below we present another part of the dynamic analysis. The approach is based on a resource consumption footprint. MPI is widely used in the high performance computing community. One of the common bugs in MPI programs is a deadlock because several usage patterns of MPI can lead to this type of bug. Therefore, we choose deadlocks in MPI programs to demonstrate how resource consumption traces can be used to identify bugs at runtime. To this end, we use MPI to implement a Monte Carlo simulation [40] for calculating mathematical constant π. The algorithm works as follows. We first generate a large number (n) of random points with coordinates within a unit square, i.e., a square with side length 1. We then calculate the number (m) of points that fall into the incircle of the unit square. Statistically, when n is large enough, we can use $4(m/n)$ to estimate the value of π. This algorithm allows flexible configuration for the amount of computation and level of concurrency.



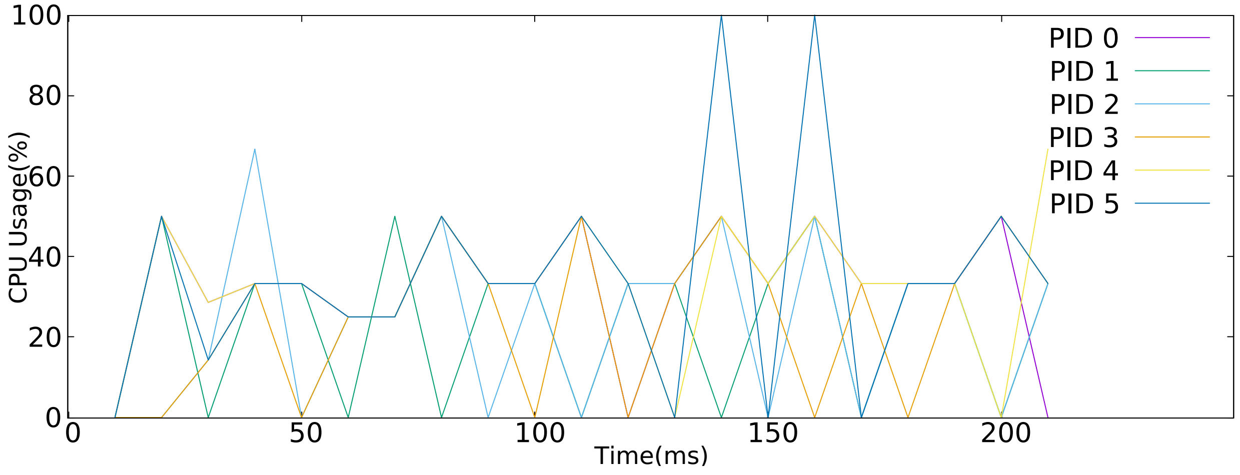
```
if(task id == 0){
   MPI_Recv(&pi,1,MPI_DOUBLE,1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
   MPI_Send(&pi,1,MPI_DOUBLE,1,0,MPI_COMM_WORLD);
   }

if(task id == 1){
    MPI_Recv(&pi,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    MPI_Send(&pi,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
   }
```

Figure 4.3: Deadlock Injection

We then inject a deadlock [41–43] to the MPI program. Figure 4.3 shows how deadlock is implemented. A recv-recv deadlock occurs due to incorrect use of `send` and `receive`. Using system profiling tools, we can retrieve the system resource consumption traces for both the correct program and the program with a deadlock injected. To illustrate the effect of the injected deadlock on resource consumption at runtime, we use CPU usage as an example. Figure 4.4 shows the runtime CPU usage traces for both the correct program and the program



(a) Normal Execution



(b) Execution with Deadlock

Figure 4.4: Resource Consumption Traces

with deadlock. It can be seen that the execution with deadlock has a different CPU usage pattern, comparing to the normal execution. We are interested in investigating whether the resource consumption patterns can be used in detecting bugs.

To explore the potentials of detecting deadlocks using runtime resource consumption patterns, we train various machine learning models using a dataset containing both normal and deadlocked executions of the π calculation. The dataset is obtained as follows: 1) we trigger the system profiling tool; 2) we execute the MPI program, and record the runtime resource consumption in a file; 3) we preprocess the data file and use three main parameters: process id (PID), CPU usage in percentage (%CPU), and Timestamp [44,45]. Using this method, we collect 200 execution traces, among which 100 are normal executions, and another 100 are executions with deadlocks. These traces form the dataset we use to train and test the machine learning models. To investigate the correlations between resource consumption patterns and the deadlock, we choose two different machine learning methods, SVM and K-nearest neighbors, both are widely used for supervised classification purposes. For training, we use 80% of the data and use the rest of 20% for testing.

Several approaches were attempted to create a structure which accurately portrayed the usage of resources throughout the entire run-time of a given file. Considering the wide variability of CPU usage of individual processes within each run-time, statistical measurements were observed to best display the differing patterns between deadlocked and bug-free files. The overall CPU usage of a file during a run-time was used as a unit to compute these measurements and their values were added as features, in addition to the raw features provided in the data file. The measurements include the median, mean, standard deviation, variance, and overall sum.

Many classification algorithms, including SVM and K-nearest neighbors, provide a more accurate prediction when features are normalized. Often datasets will contain feature values that highly vary in range and magnitude, which can cause miscalculations. Therefore, feature scaling is a common practice of preprocessing data in machine learning. Scaling transforms feature values so that they are centered around zero which allows the variance of the features to be in the same range. For comparison purposes, we have carried out two sets of experiments, one without scaling, one with scaling. The scaled values were computed using StandardScaler, which standardizes the features by subtracting the mean from each value and then dividing it by the standard deviation of the whole set.

Table 4.3 shows the experimental results. For both learning methods, the scaling [46] process helps to improve the prediction performance, achieving an overall 95%+ accuracy.

Table 4.3: Deadlock Detection Accuracy

Learning Methods	**Preprocessing settings**	**Accuracy (deadlock)**	**Accuracy (overall)**
SVM	No scaling	100%	70%
	Scaling	100%	95%
K-Nearest Neighbors	No scaling	80%	90%
	Scaling	100%	97.5%

It is worth noting that even without scaling, SVM can detect all deadlocked traces using the CPU usage patterns. These results illustrate that resource consumption patterns are promising features for detecting software bugs at run-time. The above approach is useful in the case where the pattern [47] of resource consumption is known or in scenarios where we encounter similar bugs from the past.

Chapter 5

CONCLUSION AND FUTURE WORK

Diagnosing and fixing MPI bugs has long been a challenging problem in the software community. In this thesis, we present our work in detecting MPI bugs by using a hybrid approach. For the static approach, we detect bugs at compile time with the help of LLVM IR. This technique of handling the bugs would reduce both the run-time and the dependency on source code which results in a completely new method to detect bugs in MPI. Moreover having such a technique provides the developer with a lot of information as the optimized LLVM IR almost look-alike to the source code. It is fairly readable. The optimized LLVM IR also provides us with information on the target triple stating on which machine we are running our code. It provides us with more details on the compile level to understand what's going on during our program compilation as well. While for the dynamic approach we introduce two types of approach, Machine Learning-based approach, and rule-based approach. In the case of a rule-based approach, we trace the execution period of the program and compare it with the user estimated execution period to notify if any bug was detected during the run time of the program. While in Machine Learning-based approach we look into the system

Table 5.1: Run-time analysis of different NPB benchmarks written in FORTRAN:

Benchmarks	Min-Runtime	Max-Runtime	Class
CG	1.21	2.4	A
EP	12.54	13.98	A
BT	61.21	65.67	A
LU	40.23	43.54	A

resources consumption pattern at runtime. The approach is beneficial in case of detecting bugs in systems which have stable workflow and known bug pattern from the past. It also provides the user or developer with the flexibility to generate their own set of rules to detect bugs assuming that the user has domain knowledge about the run-time of the program. For future work, we would like to add more rules for static bug detection. Moreover, we also want to make our approach not to be used only for detecting bugs in the MPI program but for all sorts of programming languages. Whereas for the dynamic approach we would like to look into what other resources patterns that could be helpful in the detection of the bugs in the MPI programs.

The total time it takes for our rule-based approach to analyze the bugs in the program is equivalent to the total run-time of the program and the time taken for the comparison of the user-defined run-time with real run-time.

Table 5.1 shows the run-time of different NPB benchmarks. All the benchmarks (CG, EP, BT, and LU) are written in Fortran. The max-runtime and min-runtime are represented in seconds. Class A represents a standard test problem size. With our tool, we were even able to analyze the programs written in Fortran.

Bibliography

[1] LLVM, “www.llvm.org.”

[2] N. P. Benchmarks, “www.nas.nasa.gov/publications/npb.html.”

[3] CombBLAS, “gauss.cs.ucsb.edu/ aydin/combblas/html/.”

[4] OpenFFT, “www.openmx-square.org/openfft/.”

[5] V. Forejt, D. Kroening, G. Narayanaswamy, and S. Sharma, “Precise predictive analysis for discovering communication deadlocks in mpi programs,” in *International Symposium on Formal Methods.* Springer, 2014, pp. 263–278.

[6] G. R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva, “Deadlock detection in mpi programs,” *Concurrency and Computation: Practice and Experience*, vol. 14, no. 11, pp. 911–932, 2002.

[7] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller, “A graph based approach for mpi deadlock detection,” in *Proceedings of the 23rd international conference on Supercomputing*, 2009, pp. 296–305.

[8] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller, “Mpi runtime error detection with must: advances in deadlock detection,” *Scientific Programming*, vol. 21, no. 3-4, pp. 109–121, 2013.

[9] I. Laguna, D. H. Ahn, B. R. De Supinski, S. Bagchi, and T. Gamblin, “Probabilistic diagnosis of performance faults in large-scale parallel applications,” in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012, pp. 213–222.

[10] I. Laguna, D. H. Ahn, B. R. de Supinski, T. Gamblin, G. L. Lee, M. Schulz, S. Bagchi, M. Kulkarni, B. Zhou, Z. Chen *et al.*, “Debugging high-performance computing applications at massive scales,” *Communications of the ACM*, vol. 58, no. 9, pp. 72–81, 2015.

[11] I. Laguna, D. H. Ahn, B. R. de Supinski, S. Bagchi, and T. Gamblin, “Diagnosis of performance faults in largescale mpi applications via probabilistic progress-dependence

inference," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 5, pp. 1280–1289, 2014.

[12] B. Zhou, M. Kulkarni, and S. Bagchi, "Vrisha: using scaling properties of parallel programs for bug detection and localization," in *Proceedings of the 20th international symposium on High performance distributed computing*, 2011, pp. 85–96.

[13] E. Karrels and E. Lusk, "Performance analysis of mpi programs," *Environments and Tools for Parallel Scientific Computing*, pp. 195–200, 1994.

[14] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008).* New York, NY, USA: ACM, 2008, pp. 329–339.

[15] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: fault localization in concurrent programs," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 245–254.

[16] M. S. Ayub, W. U. Rehman, and J. H. Siddiqui, "Experience report: Verifying mpi java programs using software model checking," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE).* IEEE, 2017, pp. 294–304.

[17] X. Ouyang, K. Gopalakrishnan, T. Gangadharappa, and D. K. Panda, "Fast checkpointing by write aggregation with dynamic buffer and interleaving on multicore architecture," in *2009 International Conference on High Performance Computing (HiPC).* IEEE, 2009, pp. 99–108.

[18] A. Droste, M. Kuhn, and T. Ludwig, "Mpi-checker: Static analysis for mpi," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–10.

[19] A. J. McPherson, V. Nagarajan, and M. Cintra, "Static approximation of mpi communication graphs for optimized process placement," in *International Workshop on Languages and Compilers for Parallel Computing.* Springer, 2014, pp. 268–283.

[20] E. Gabriel, G. E. Fagg, and J. J. Dongarra, "Evaluating the performance of mpi-2 dynamic communicators and one-sided communication," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting.* Springer, 2003, pp. 88–97.

[21] R. Preissl, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan, "Using mpi communication patterns to guide source code transformations," in *International Conference on Computational Science.* Springer, 2008, pp. 253–260.

[22] H. Li, Z. Chen, and R. Gupta, "Parastack: Efficient hang detection for mpi programs at large scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[23] H. Lin, Z. Wang, S. Liu, J. Sun, D. Zhang, and G. Wei, "Pfix: Fixing concurrency bugs based on memory access patterns," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 589–600.

[24] G. Florez, Z. Liu, S. M. Bridges, A. Skjellum, and R. B. Vaughn, "Lightweight monitoring of mpi programs in real time," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 13, pp. 1547–1578, 2005.

[25] T. Hilbrich, M. Schulz, B. R. de Supinski, and M. S. Müller, "Must: A scalable approach to runtime error detection in mpi programs," in *Tools for high performance computing 2009*. Springer, 2010, pp. 53–66.

[26] T. Hilbrich, B. R. de Supinski, F. Hänsel, M. S. Müller, M. Schulz, and W. E. Nagel, "Runtime mpi collective checking with tree-based overlay networks," in *Proceedings of the 20th European MPI Users' Group Meeting*, 2013, pp. 129–134.

[27] J. Protze, T. Hilbrich, B. R. de Supinski, M. Schulz, M. S. Müller, and W. E. Nagel, "Mpi runtime error detection with must: Advanced error reports," in *Tools for High Performance Computing 2012*. Springer, 2013, pp. 25–38.

[28] V. . Message Passing Interface Forum. MPI : A Message-Passing Interface Standard, "www.mpi-forum.org/docs/mpi-3.1/ mpi31-report.pdf."

[29] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008, pp. 329–339.

[30] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 117–132.

[31] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan, "Mining console logs for large-scale system problem detection." *SysML*, vol. 8, pp. 4–4, 2008.

[32] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," in *2009 Ninth IEEE International Conference on Data Mining*. IEEE, 2009, pp. 588–597.

[33] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*. IEEE, 2003, pp. 119–126.

[34] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou, "Mpi-check: a tool for checking fortran 90 mpi programs," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 2, pp. 93–100, 2003.

[35] J. S. Vetter and B. R. De Supinski, "Dynamic software testing of mpi applications with umpire," in *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE, 2000, pp. 51–51.

[36] T. Dai, D. Dean, P. Wang, X. Gu, and S. Lu, "Hytrace: a hybrid approach to performance bug diagnosis in production cloud infrastructures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 1, pp. 107–118, 2018.

[37] Z. Chen, Q. Gao, W. Zhang, and F. Qin, "Flowchecker: Detecting bugs in mpi libraries via message flow checking," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.

[38] Z. Chen, X. Li, J.-Y. Chen, H. Zhong, and F. Qin, "Syncchecker: detecting synchronization errors between mpi applications and libraries," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 342–353.

[39] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, "Diagnosing performance variations in hpc applications using machine learning," in *International Supercomputing Conference*. Springer, 2017, pp. 355–373.

[40] K. Binder, D. Heermann, L. Roelofs, A. J. Mallinckrodt, and S. McKay, "Monte Carlo Simulation in Statistical Physics," *Computers in Physics*, vol. 7, no. 2, pp. 156–157, 1993.

[41] J.-W. Lee, L. R. Bachega, S. P. Midkiff, and Y. C. Hu, "Ant: a debugging framework for mpi parallel programs," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2012, pp. 220–233.

[42] K. Sato, D. H. Ahn, I. Laguna, G. L. Lee, M. Schulz, and C. M. Chambreau, "Noise injection techniques to expose subtle and unintended message races," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 89–101.

[43] Z. Chen, J. Dinan, Z. Tang, P. Balaji, H. Zhong, J. Wei, T. Huang, and F. Qin, "Mc-checker: Detecting memory consistency errors in mpi one-sided applications," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 499–510.

[44] C. Gottbrath, "Eliminating parallel application memory bugs with totalview," in *Proceedings of the 2006 ACM/IEEE conference on supercomputing*, 2006, pp. 210–es.

[45] Q. Gao, F. Qin, and D. K. Panda, "Dmtracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements," in *SC'07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. IEEE, 2007, pp. 1–12.

[46] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, “Using automated performance modeling to find scalability bugs in complex codes,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[47] D. J. Quinlan, R. W. Vuduc, and G. Misherghi, “Techniques for specifying bug patterns,” in *Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, 2007, pp. 27–35.