Runtime Support for Improving Reliability in System Software

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Qi Gao, B.E., M.S.

Graduate Program in Computer Science and Engineering
Ohio State University
2010

Dissertation Committee:

Prof. Feng Qin, Advisor
Prof. Atanas Rountev
Prof. Xiaodong Zhang
ABSTRACT

As software is becoming increasingly complex, software reliability is getting more and more important. In particular, the reliability of system software is critical to the overall reliability of computer systems since system software is designed to provide a platform for application software running on top of. Unfortunately, it is very challenging to ensure the reliability of system software and the defects (bugs) in it can often cause severe impact.

This dissertation proposes to use runtime support for improving system software reliability. Runtime support here refers to the technique to extend the runtime software system with more functionalities useful for reliability-oriented tasks, such as instrumentation-based profiling, runtime analysis, checkpointing/re-execution, scheduling control, memory layout control, etc. Leveraging runtime support, this dissertation proposes novel methods for bug manifestation, bug detection, bug diagnosis, failure recovery and error prevention in multiple phases in the software development and deployment cycle.

The most preferable phase to detect and fix software bugs is pre-release testing phase. To improve the testing effectiveness and efficiency, this dissertation proposes the first method to help manifest the bugs hidden in system software. Facing the real-world fact that there are always some bugs making their way to deployment sites no matter how rigorous the software testing is, this dissertation proposes the second method to help monitor the system software and detect runtime errors. To handle the runtime errors caused by software bugs, this dissertation proposes the third method to help diagnose the failure, recover the program, and prevent future errors due to the same bugs.

Specifically, we propose a software testing method called 2ndStrike to manifest hidden concurrency typestate bugs in multi-threaded system software. 2ndStrike first profiles certain
program runtime events related to the typestate and thread synchronization. Based on the logs, 2ndStrike then identifies bug candidates that would cause typestate violation if event order is reversed. Finally, 2ndStrike re-executes the program in multiple iterations with controlled thread interleaving for manifesting bug candidates.

In addition, we propose a deployment-time monitoring and analysis method called **DM-Tracker** to detect anomalies in distributed system software running on parallel platforms during production runs. Based on the observation that data movements in parallel programs typically follow certain patterns, our idea is to extract data movement (DM)-based invariants at program runtime and check the violations of these invariants. These violations indicate potential bugs such as data races and memory corruption bugs that manifest themselves in data movements. Utilizing the data movement information, we propose a statistical-rule-based approach to detect anomalies for finding bugs.

Finally, we propose a deployment-time fault tolerance method called **First-Aid** to recover failures in system software due to common memory bugs during production runs and prevent future errors caused by the same bugs. Upon a failure, First-Aid diagnoses the bug type and identifies the memory objects that trigger the bug. To do so, it rolls back the program to previous checkpoints and uses two types of environmental changes that can prevent or expose memory bug manifestation during re-execution. Based on the diagnosis, First-Aid generates and applies runtime patches to avoid the memory bug and prevent its reoccurrence.

We have designed and implemented software prototypes for the proposed methods and evaluated them with real world bugs on large open-source system software packages, such as Apache, MySQL, Mozilla, MVAPICH, etc. The experimental results show that the methods proposed in this dissertation can provide great help in improving reliability of system software in various scenarios. In addition, the results also demonstrate that the runtime support in these methods can bring key advantages such as high efficiency, high accuracy, and high usability.
Dedicated To

My Love, Ying Tu
ACKNOWLEDGMENTS

I am truly lucky to have Professor Feng Qin as my advisor, who is kind, generous, and inspiring. On top of many good traits that an excellent advisor can have, his most admirable quality that distinguishes him from other good professors I have seen or heard is that he genuinely cares and respects his students. His insights and guidance have made me a better researcher, while more importantly, his open mind and optimism have influenced me to become a stronger person.

I am also deeply indebted to Professor Xiaodong Zhang and Professor Atanas Rountev, not only for their time and effort in serving in my dissertation committee, but also for their generous help and insightful feedbacks during my study. Thanks also go to other professors who have worked with me and/or passed their knowledge to me. An incomplete list of them are Professor Panda, Professor Sadayappan, Professor Agrawal, Professor Xuan, Professor Arora, Professor Lai, Professor Long, and Professor Soundarajan.

I would also like to thank my groupmates and fellow graduate students. Wenbin Zhang, Zhezhe Chen and Mai Zheng, thanks for extending help in the research projects. I thank Wei Huang, Lei Chai, Matthew Koop, Sayantan Sur, Pavan Balaji for their help in improving my research skills, engineering skills, and English speaking and writing skills. I thank Guoqiang Shu, Zhaozhang Jin, Qian Zhu, Lifeng Sang, Guoqing Xu, Ping Lai, Na Li and Hui Gao for their help in various aspects.

I am grateful to Dr. Min Xu in VMware., Dr. Jason Yang in Microsoft, and Dr. Pete Beckman in Argonne National Lab for being my mentor while I was doing internships. I feel lucky working with them and other excellent researchers/engineers in their teams. I have really learned a lot and greatly broadened my horizon from these valuable experiences.
Those who I want to thank the most will not mind that I put them the last. I am deeply thankful to my parents, Weishi Gao and Xiaoli Liu, who have been providing me a cultivating and nurturing environment for my grow-up. They always believe in me and support me in every possible way. Finally, my very special thanks are given to my dearest wife-to-be, Ying Tu, who has been with me during the whole journey of going to a distant country for Ph.D. study. She is smart, diligent, caring, inspiring and she creates so much joy in my life. It is her love and support that unleashed my full potential.
VITA

2004 ........................................ Bachelor in Engineering
Zhejiang University
Hangzhou, China

2009 ........................................ Master of Science
The Ohio State University
Columbus, OH, USA

2004 – Present ................................ Graduate Student
The Ohio State University
Columbus, OH, USA

PUBLICATIONS

Research Publications


**FIELDS OF STUDY**

Major Field: Computer Science and Engineering
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td></td>
<td>i</td>
</tr>
<tr>
<td>Dedication</td>
<td></td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td></td>
<td>iv</td>
</tr>
<tr>
<td>Vita</td>
<td></td>
<td>vi</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.2 Contributions</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1.2.1 Manifesting hidden concurrency bugs in multi-threaded system software in software testing</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>1.2.2 Runtime monitoring and anomaly detection in distributed system software during production runs</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>1.2.3 Failure recovery and future error prevention for common memory bugs in system software during production runs</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>1.3 Outline</td>
<td>10</td>
</tr>
<tr>
<td>2.</td>
<td>RELATED WORK</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>2.1 Static and Dynamic Bug Detection</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>2.2 Software Testing and Model Checking</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>2.3 Fault Tolerance</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>2.4 Debugging and Problem Diagnosis</td>
<td>17</td>
</tr>
<tr>
<td>3.</td>
<td>2NDSTRIKE</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>3.1 Overview</td>
<td>20</td>
</tr>
</tbody>
</table>
3.2 Background: Typestate Bugs ........................................... 25
3.3 Our Approach: 2ndStrike ............................................. 26
  3.3.1 State Machines for Detecting Concurrency Typestate Bugs .... 27
  3.3.2 Runtime Profiling to Monitor Typestate ......................... 29
  3.3.3 On-the-fly Analysis to Identify Bug Candidates ................. 30
  3.3.4 Scheduling Control to Manifest Bugs .......................... 33
  3.3.5 Issues and Discussion ........................................... 34
3.4 Evaluation Methodology ............................................. 35
3.5 Experimental Results .............................................. 37
  3.5.1 Effectiveness .................................................... 37
  3.5.2 Efficiency ...................................................... 38
  3.5.3 Usability ....................................................... 41
3.6 Summary ............................................................ 42

4. DMTRACKER .......................................................... 44
  4.1 Overview .......................................................... 45
  4.2 Background: Statistical Rule Based Bug Detection ............... 50
  4.3 Data Movement-Based Invariants .................................. 51
    4.3.1 Frequent Chain Invariants ................................ 52
    4.3.2 Chain Distribution Invariants .............................. 54
  4.4 Design of DMTracker ............................................. 56
    4.4.1 Lightweight Data Movement Tracking ....................... 57
    4.4.2 Preprocessing: DM-Chain Formation ....................... 58
    4.4.3 Invariants Generation ..................................... 60
    4.4.4 FC-invariants Based Anomaly Detection ................... 63
    4.4.5 CD-invariants Based Anomaly Detection ................... 65
    4.4.6 Issues and Discussions .................................... 66
  4.5 Evaluation and Case Studies ..................................... 67
    4.5.1 Case 1: Data Corruption in Communication ................ 68
    4.5.2 Case 2: Deadlock in Connection Setup .................... 69
    4.5.3 Runtime Overhead ......................................... 70
    4.5.4 Sensitivity Study and False Positives ..................... 71
  4.6 Summary .......................................................... 73

5. FIRST-AID .......................................................... 74
  5.1 Overview .......................................................... 75
  5.2 First-Aid Approach and Working Scenario ....................... 78
  5.3 First-Aid System Design ......................................... 83
  5.4 Bug Diagnosis .................................................... 86
    5.4.1 Phase 1: Identify the checkpoint for patching ............ 86
    5.4.2 Phase 2: Identify the bug type and patch application points 88
    5.4.3 Comparison of First-Aid and Rx bug diagnosis ............. 90
CHAPTER 1

INTRODUCTION

1.1 Motivation

As software is becoming more and more complex, software reliability becomes increasingly important. From Internet server programs to desktop programs, from scientific applications to financial applications, high reliability is critical to many aspects of our lives. Almost every large organization nowadays depends on the smooth operation of its Internet servers such as mail servers, web servers, etc. Many people feel frustrated and can not work productively when there are problems with their desktop programs such as browser, text editor, calendar, etc. For scientific applications, reliability problems can lead to huge waste on computing cycles or even wrong results in simulation. For financial applications, down time directly means cost. According to a report from Gartner Group [1], the cost of one hour downtime of financial applications exceeds six million US dollars.

In particular, system software is the computer software designed to provide and maintain a platform for application software to run on top of. As shown in Figure 1.1, system software runs on top of operating system (OS) kernels or directly works with computer hardware in order to provide application software higher level functionalities. There are many categories of system software, including common libraries such as C library and MPI [2] library, server programs such as web servers and database servers, utility software such as graphical user
interface, modern web browser engines, etc. As the glue between the simple and condensed interface provided by OS kernels and rich functionalities required by applications, system software is extremely important to the overall capability of computer systems.

Because of its importance, the reliability of system software is critical to the overall reliability of computer systems. First of all, system software is generally very widely deployed and commonly used based on the requirements from application software and users. Therefore, if one system software has a bug (or defect), it will affect potentially millions of users. On the other hand, due to its popularity, malicious attackers often attempt to exploit the defects in system software to launch security attacks. For example, in 2003 the Slammer worm exploiting the defects of Microsoft SQL servers brought down tens of thousands of machines installed with this system software within several minutes [3]. Even for high-end systems, a recent study [4] with more than 20 different high-performance computer systems at Los Alamos National Laboratory has shown that bugs in system software account for as many as 24% of failures.

In addition, since system software is the backbone of the many applications, the reliability problem of system software often causes severe impacts. A bug in system software may cause the whole system to crash, hang, or exhibit other unexpected behaviors, and thus leads to
financial loss. Even worse, due to its complexity, the bugs in system software typically require experts to diagnose and fix, and meanwhile it is costly for users to avoid using the system software totally. For example, the Slammer worm incident mentioned above caused hundreds of millions of dollars loss.

Unfortunately, it is very challenging to ensure the reliability in system software. Firstly, system software is required to provide a rich set of functionalities used by applications, and thus it usually has a very complex design and is not easy to ensure correctness. Coding errors can happen in any corner of a complex framework and cause problems in production when the particular functionality is exercised. In addition, system software is often required to be high-performing and portable on many platforms, which usually introduces non-intuitive optimizations and adjustments on the implementations of system software. Even a very experienced programmer can make subtle mistakes in writing system software dealing with all the trickiness. Furthermore, system software evolves with the underlying platforms, and changes often lead to errors. It generally requires a collective effort from generations of developers, testers, and operators to improve the reliability of one piece of system software.

Specifically, there are challenging issues involved in each part of software development and deployment cycle.

- In software testing phase, manifesting hidden bugs in programs can be quite challenging. This is mainly because: a) the possible cases to test is usually in huge number and it is impractical to cover all cases; and b) software bugs are often introduced when handling the corner cases which are often not covered adequately by tests as well. For example, multi-threaded programs have more than one threads running concurrently during the execution. Even if the threads execute the same set of statements, the different runtime interleavings could lead to both correct and incorrect results. Since the number of possible interleavings is often astronomically large, it is impractical to
cover all cases. Also, the bugs tend to manifest when a rare interleaving case happens during execution, which may be extremely difficult to achieve using natural scheduling. As a result, many bugs still slip through the testing phase and left uncaught.

- After deployment, monitoring and detecting runtime errors can be challenging in the production runs. This is mainly because: a) the method must be very lightweight and only impose very small overhead for not causing noticeable performance degradation; b) a lightweight method often can only provide very limited information, and how to effectively detect problems based on this information is difficult; and c) the platform where the software is deployed may also become a barrier for monitoring and error detection. For example, many system software run on distributed machines. Only monitoring their individual activities may not be helpful in detecting errors, hence the inter-communication among them also needs to be monitored. Monitoring communication is challenging because the naive way of logging all the messages will often incur unacceptable overhead. In addition, how to correlate the logs from distributed machines and detect runtime errors is even more challenging.

- When a bug happens in production runs, tolerating the fault and keeping the software running healthily can be challenging. This is mainly because: a) the method must be fully automatic and do not involve human in critical path because the software can crash at any time in production regardless whether it is in human operators’ office hours; b) the method must be safe and do not introduce new bugs; c) the method must be lightweight and do not incur large overhead; and d) a good method must be able to prevent the error from being repetitively triggered. For example, if a program is crashed due to some input triggering a deterministic memory bug, the same or similar input will trigger the bug again next time. In this case, the common restart approach
used in practice will not be enough for fault tolerance. With all these requirements, finding a good method to tolerate even some specific type of bugs is quite challenging.

Facing these challenges, it is imperative for developers and operators to have effective and efficient methods for detecting and diagnose bugs in system software.

1.2 Contributions

This dissertation proposes to use runtime support for improving system software reliability. Runtime support here refers to the technique to extend the runtime software system with more functionality useful for reliability-oriented tasks, such as instrumentation-based profiling, runtime analysis, checkpointing/re-execution, scheduling control, memory layout control, etc. Leveraging runtime support, this dissertation proposes novel methods for bug manifestation, bug detection, bug diagnosis, failure recovery and error prevention. Specifically, runtime support in this dissertation includes two aspects: system support and dynamic analysis.

Traditionally, the main thing runtime system focuses is how to deliver good performance. In contrast, system support for reliability means extending runtime system to provide additional functionality for reliability-oriented tasks. These tasks range from extracting logs and aggregating runtime information to taking checkpoints of the running program and re-executing the program from a previous checkpoint. Proper system support is the cornerstone of some useful techniques and can often greatly increase the efficiency of the tools with minor performance cost.

Compared with static analysis focusing on the source code, dynamic analysis only focuses on the program behaviors that exhibit at runtime, and therefore is more accurate and provides more detailed information. Because of that, dynamic analysis not only generates more accurate and more informative bug reports, but also allows runtime changes to be
made during the execution. These runtime changes enable more sophisticated tasks such as manifesting hidden bugs and preventing future bug occurrence.

This dissertation proposes to use automated bug detection and diagnosis techniques for improving reliability of system software. In particular, this dissertation focuses on the automated methods that combine system support and runtime analysis to improve the effectiveness and efficiency in many tasks such as bug detection, runtime monitoring, failure diagnosis, fault tolerance, and error prevention.

Combining system support and dynamic analysis, this dissertation proposes novel methods for bug manifestation, bug detection, bug diagnosis, failure recovery and error prevention, with following desirable features:

- High efficiency: The methods proposed in this dissertation for productions runs have very low runtime overhead, often causing negligible impact on performance. The methods for testing imposes moderate performance overhead and greatly increases the testing efficiency based on current state of the art.

- High accuracy: The methods proposed in this dissertation only generate a small number of, if any, false positives when performing bug detection. For bug diagnosis, the methods can accurately identify the root causes of the bugs.

- Easy to use: The methods proposed in this dissertation are easy to use by developers and operators because: a) they are fully automated; b) they do not require source code change on the software, and c) they provide detailed information in bug reports.

Specifically, this dissertation proposes novel methods to address issues in multiple parts of the software development and deployment cycle. The most preferable phase to detect and fix software bugs is pre-release testing phase. To improve the testing effectiveness and efficiency, this dissertation proposes the first method to help manifest the bugs hidden in
system software. Facing the real-world fact that there are always some bugs making their way to deployment sites no matter how rigorous the software testing is, this dissertation proposes the second method to help monitor the system software and detect runtime errors. To handle the runtime errors caused by software bugs, this dissertation proposes the third method proposed to help diagnose the failure, recover the program, and prevent future errors due to the same bugs. More specific descriptions are provided as follows.

1.2.1 Manifesting hidden concurrency bugs in multi-threaded system software in software testing

For performance and responsiveness, system software often adopt a multi-threaded design. Unfortunately, concurrency bugs in multi-threaded programs are notoriously difficult to detect in testing, mainly due to the huge interleaving possibilities to cover. Stress testing, the de facto method used in current software testing practice, can only cover a small fraction of the interleaving space with non-trivial usage on computing resource, hence not a very desirable solution.

Based on our observation, a large number of concurrency bugs are related to the typestate [5] of a certain object, e.g. invalid reads to a file handler closed by a different thread. To address this, we propose a method called 2ndStrike for manifesting these concurrency typestate bugs based on runtime profiling and actively changing thread scheduling.

Given a state machine describing correct program behavior on certain object typestates, 2ndStrike tool first profiles certain program runtime events related to the typestate and thread synchronization. Based on the logs, 2ndStrike then identifies bug candidates that would cause typestate violation. A bug candidate here refers to a pair of runtime events that happen in different threads and could cause error if the order between them are changed. Finally, 2ndStrike re-executes the program in multiple iterations with controlled thread interleaving for manifesting bug candidates.
We have implemented 2ndStrike prototype and evaluated it for two types of concurrency typestate bugs, i.e., invalid pointer dereference and lock operation violation and with three real world bugs from three open-source servers and desktop programs (i.e., MySQL, Mozilla, pbzip2) on a eight-core machine. Our experimental results have shown that 2ndStrike can effectively and efficiently manifest all three tested software bugs, i.e., within 255 seconds, 178 times faster than stress testing. Additionally, 2ndStrike provides detailed bug reports and can consistently reproduce the bugs after manifesting the bugs once.

1.2.2 Runtime monitoring and anomaly detection in distributed system software during production runs

Detecting runtime errors in distributed or parallel environment is especially challenging. In such environment, only tracing a single process running on a specific node or monitoring individual processes separately can provide little help in detecting and diagnosing bugs at runtime.

For monitoring and detecting anomaly in distributed parallel system software during production runs, we propose a method called DMTracker. Facing the special challenge of parallel computing platforms, we take the distributed processes in parallel program as a whole and focus on the data movements among parallel processes. Utilizing the data movement information, we propose to a statistical-rule-based approach to detect anomalies for finding bugs.

Based on the observation that data movements in parallel programs typically follow certain patterns, our idea is to extract data movement (DM)-based invariants at program runtime and check the violations of these invariants. These violations indicate potential bugs such as data races and memory corruption bugs that manifest themselves in data movements.

We have implemented DMTracker prototype and evaluated it on a cluster with 64 CPUs. Our experiments with two real-world bug cases in MVAPICH/MVAPICH2 [6], a popular
MPI library, have shown that DMTracker can effectively detect them and report abnormal data movements to help programmers quickly diagnose the root causes of bugs. In addition, DMTracker incurs very low runtime overhead, from 0.9% to 6.0%, in our experiments with High Performance Linpack (HPL) [7] and NAS Parallel Benchmarks (NPB) [8], which indicates that DMTracker can be deployed in production runs.

1.2.3 Failure recovery and future error prevention for common memory bugs in system software during production runs

Common memory management bugs such as buffer overflow and dangling pointer access are one of the major causes of software failures and exploits in production runs. For example, a recent large-scale security attack on several large companies including Google was exploiting a dangling pointer bug [9], and it takes quite a long time for developers to generate a fix.

We propose an automated on-site diagnosis and patching method for memory bugs to avoid the long delays for developers to fix the bugs that manifest in production runs. By using fully automated diagnosis and patching method, the memory bugs manifested in production runs will be treated immediately online without the need to wait for developers to manually reproduce and fix.

We have designed and implemented First-Aid, a lightweight runtime system that help software survive failures caused by common memory management bugs and prevents future failures by the same bugs during production runs. Upon a failure, First-Aid diagnoses the bug type and identifies the memory objects that trigger the bug. To do so, it rolls back the program to previous checkpoints and uses two types of environmental changes that can prevent or expose memory bug manifestation during re-execution. Based on the diagnosis, First-Aid generates and applies runtime patches to avoid the memory bug and prevent its reoccurrence. Furthermore, First-Aid validates the consistent effects of the runtime patches and generates on-site diagnostic reports to assist developers in fixing the bugs.
We have evaluated First-Aid prototype with seven applications that contain various types of memory bugs, including buffer overflow, uninitialized read, dangling pointer read/write, and double free. The results show that First-Aid can quickly diagnose the tested bugs and recover applications from failures (in 0.084 to 3.978 seconds). The results also show that the runtime patches generated by First-Aid can prevent future failures caused by the diagnosed bugs. Additionally, First-Aid provides detailed diagnostic information on both the root cause and the manifestation of the bugs. Furthermore, First-Aid incurs low overhead (0.4-11.6% with an average of 3.7%) during normal execution for the tested buggy applications, SPEC INT2000, and four allocation intensive programs.

1.3 Outline

The rest of this dissertation is organized as follows: In Chapter 2, we discuss previous studies that are related to our work. In Chapter 3, we describe the method to manifesting hidden concurrency bugs in multi-threaded system software in software testing. In Chapter 4, we describe the method for runtime monitoring and anomaly detection in distributed system software during production runs. In Chapter 5, we describe the method for failure recovery and future error prevention for common memory bugs in system software during production runs. At last in Chapter 6, we provide a summary and draw the conclusion.
CHAPTER 2

RELATED WORK

2.1 Static and Dynamic Bug Detection

Generally tools in this category [10–18] scan the program source code and use various static analysis techniques to detect potential software bugs. Evans’s LCLint [10] detects the inconsistency between the source code and properties inferred from user-provided annotations. Foster et al. proposed CQual [11], a general framework for checking program invariants specified by customized type qualifiers. Similarly, METAL [13] checks the source code and detect violations of programming rules, either provided by programmers [19] or automatically inferred from the source code itself [20]. Some tools are designed to detect certain types of bugs such as memory leaks and memory corruption. Clouseau [16] detects memory leaks using an object ownership and inference model, while CSSV [15], proposed by Sagiv et al., detects unsafe string operations in C programs with the aid of procedure summaries. In [17,18], a static race detector framework is proposed for Java programs using a special property called “conditional must not aliasing” to reduce overhead. While static tools do not impose runtime overheads, they may miss some bugs and/or generate many false positives because accurate runtime information is unavailable at compile time. Additionally, many static tools do not scale well to large programs.
Dynamic tools, either purely based on software [21–25] or relying on hardware extension [26–28], detect software bugs at runtime. The state-of-the-art tools Purify [21] and Valgrind [22] detect memory-related bugs such as memory leaks and memory corruption by intercepting every memory access and monitoring every dynamically allocated memory objects through binary instrumentation. Jones and Kelly’s tool [25], PointGuard [29], SafeC [24] and CRED [30] can detect buffer overflows by dynamically checking each pointer dereference. While these tools do not suffer from the same limitation as static tools, most software-based detection tools usually incur high runtime overhead, up to 40 times [23, 28] due to interception of every memory/pointer access.

Some hybrid schemes combine static and dynamic technique to alleviate the high overhead problem to some extent. CCured [23] is such a hybrid bug detection tool. It first attempts to enforce a strong type system in C programs via static analysis. Portions of the program that cannot be guaranteed by the CCured type system are instrumented with run-time checks to monitor the safety of executions. Cyclone [31] is very similar. It changes the pointer representation to detect pointer dereference error.

Another direction to reduce runtime overhead incurred by software-based dynamic bug detection tools is to rely on hardware support. Some tools, such as ReEnact [26], iWatcher [27], AccMon [28], SafeMem [32] etc., propose hardware extension to support bug detection.

Besides the bug detection tools based on programming rules, there is another category of tools detecting bugs using statistical methods. They extract rules statistically at program runtime, and check the violations of the extracted rules. Statistical rules, or dynamic invariants [33, 34], are properties that likely hold at a certain point or points in a program. One can extract invariants at runtime over a single run or across multiple runs. Several recent works [28, 33, 34] have demonstrated that statistics-rule-based approaches are very promising due to their effectiveness in detecting bugs that do not violate any programming
rules. Daikon [33] and DIDUCE [34] extract value-based invariants (i.e., the possible values of a given variable are always within a certain range) at runtime and can detect bugs that generate abnormal values. Similarly, AccMon [28] captures the runtime program-counter-based invariants (i.e. a given variable is typically accessed by only a few instructions), and use them to detect memory-related bugs. Mirgorodskiy et al. have conducted an initial study of using statistics-rule-based approach to diagnose software problems in large-scale systems [35]. Their approach extracts the invariant of function time distribution in control flow, and then identify the abnormal process among a large number of parallel processes.

For detecting concurrency bugs in multi-threaded programs, there are many existing approaches on data race detection [36–40]. Most of the data race detection methods are based on two categories of algorithms: the happen-before algorithm [36] and the lockset algorithm [37], or a combination of both [39, 40]. However, data races are not necessarily concurrency bugs. As pointed out in [39, 40], many data races are intentionally introduced to reduce locking cost and improve performance. There is also some method proposed to distinguish benign races from harmful races [41].

There are other approaches focusing on the atomicity violation problem [42–44]. This problem is based on the fact that developers write programs with conceptual atomic regions in mind. If during the execution, interleaved accesses from other threads break the atomic region and lead to unserializable results, the program invariants are likely be violated. Some approaches like Atomizer [43] require programmers’ annotation while other approaches [42, 44] are based on heuristics gained during execution. In common, these tools report atomicity violations that happen at runtime to programmers.
2.2 Software Testing and Model Checking

Typically, model checking tools check system properties at the protocol level [45–47] or the implementation level [48–51] by exhaustively searching the system state space. Traditional model checking tools such as SMV [45] and SPIN [46] focus on verifying hardware and software protocols. Although they can detect non-trivial bugs, the requirement of building a model for the system in another language is the major drawback due to the required significant amount of manual effort that can easily lead errors.

Some recent software model checking tools such as Verisoft [48] and CMC [51] systematically execute and check the systems in the implementation level. They have been used to check systems for concurrency bugs, e.g., deadlock, and assertion failures. Yang et al. applied the model checking method to widely-used, heavily-tested file systems and found serious software bugs [52]. KLEE [53] applies symbolic execution to core utilities in Unix systems and discovers several deeply-buried bugs in these commonly used programs. Without requiring an abstract model for checking a target system is a big advantage. However, the state explosion problem is still a major obstacle for them, especially for large and complex systems, due to enormous state space need to be explored and limited computation resources.

Recently, there are several approaches [54–57] proposed for testing concurrent programs, mainly focusing on how to effectively and efficiently expose concurrency bugs. The common technique these approaches share is controlling on scheduling, which also used in our work for exposing deadlock bugs. However, these works are based on different ways to control scheduling in testing. CHESS [55] systematically explores the thread interleaving space based on pre-defined criteria. RaceFuzzer [57] is a testing tool for Java programs that triggers the potential data races reported by any imprecise race detector. The key idea is that based on the potential data race cases, RaceFuzzer forcefully stops the thread at the statement where
it can be involved in a data race and wait for other threads coming to the corresponding statement in the data race. When all threads are blocked, it randomly chooses one thread to unblock to maintain progress. CTrigger [56] is a testing tool for C programs that exposes the atomicity violation bugs which can only be triggered by very rare-happening scheduling cases. The key idea is first to profile potential unserializable access interleavings in stress testing and then to forcefully insert delay at special moments so that the unserializable access interleavings can be exercised.

2.3 Fault Tolerance

Whole program restart [58,59] is usually the first attempt to handle software failures with the hope that they are caused by non-deterministic bugs since non-deterministic bugs may disappear during re-execution. However, it may cause a long period of service unavailability [60,61] for server programs that buffer significant amount of state in main memory (e.g., data buffer caches). Software rejuvenation [62–64] is an interesting approach to reduce the period of unexpected service outage by rejuvenating/restarting the software to a fresh state after a certain period and before it fails. Candea et al. proposed Micro-rebooting [65,66] to address this problem to some extent by only rebooting the failed components.

General checkpointing and recovery mechanisms [67–69] have been proposed for surviving failures for a while. Typically, they checkpoint the program state, roll back the program upon failures, and then re-execute the program. The checkpoints may be done to disk [70–73], or even remote memory [74–76]. These checkpoints can be provided with relatively low overhead. If there are messages and operations in flight, logging is also needed [71,77–79]. To deal with resource exhaustion or operating system crashes, monitoring, logging and recovery can be done remotely via support by special network interface cards [80].
Various fault tolerance mechanisms can be used to survive software failures caused by deterministic bugs, a major cause of software failures [81]. The recovery blocks approach [82,83] extends a conventional block with a means of error detection and additional alternates, i.e., different implementation versions of the same block. The alternate will be executed upon a failure detected at the end of a block execution. Similarly, n-version programming [84–86] relies on different implementation versions of the same software unit. Unlike recovery blocks, it executes different versions concurrently and the result is a consensus result from all the versions. Both mechanisms can address software failures caused by deterministic bugs assuming that different implementation versions fail independently. However, they are too expensive to be deployed in the normal software development process.

Several recent proposals help programs survive failures caused by memory bugs. Failure oblivious computing [87] discards out-of-bound writes and manufactures an arbitrary value for out-of-bound reads. While this approach may survive failures for certain types of applications, the speculation on programmers’ intention could easily lead to programs’ misbehavior. DieHard [88] and Exterminator [89] probabilistically prevent failures caused by memory bugs via a randomized memory runtime system. However, large time and space overheads restrict them from being adopted for production runs. Archipelago [90] scatters memory objects to different pages for preventing buffer overflow, but the memory consumption becomes non-trivial for many applications. Rx [91] quickly recovers programs from failures by re-executing program from previous checkpoints with program execution environmental changes applied. While being effective and safe to avoid the occurring memory bugs, Rx cannot prevent future failures caused by the same bugs. This is because it will disable the environmental changes after surviving the current failures due to potentially large overhead.
Recently, several studies have been proposed for avoiding concurrency bugs. Programming on transactional memory [92] prevents data races. Isolator [93] and ToleRace [94] exploit data replication to tolerate transactional memory for surviving atomicity violation bugs during production runs. Yu et al. have proposed a shared-memory multi-processor design to avoid untested thread interleavings that may cause concurrency bugs [95]. Dimmunix [96] prevents programs from re-encountering previously-seen deadlocks. Complementary to these production-phase tools, our work is applied at development phase.

2.4 Debugging and Problem Diagnosis

Many studies on debugging support focus on off-site tools, either incurring large overhead or relying on extensive human effort. Examples of such tools are delta debugging [97, 98], program slicing [99–101], interactive debugging [102], and deterministic replay [103–108].

A recent tool called Triage [109] focuses on detailed on-site failure diagnosis. It automates several debugging technologies, such as delta debugging, program slicing, memory bug detection, etc., and combines them into one framework for generating useful on-site diagnostic information. Although Triage provide deep and comprehensive analysis to one single failed process, it cannot help much for distributed or parallel programs running in complex environment. In addition, it can only shorten the developers diagnosis delay to some extend but cannot help patch generation and application, therefore the downtime still could be long.

Many research efforts have been made to help detect bugs in parallel programs, including parallel debuggers [110–113], technologies to support interactive parallel debugging [114–117], and automatic bug detection tools [35, 118–123]. Most of these focus on helping interactive debugging by using automated information collection/aggregation technologies and visualization technologies.
Problem diagnosis in large scale systems has been studied for many years [35,124–130]. These works mainly study how to analyze and locate the root causes of system failures or performance problems. The root causes can be hardware failures, configuration problems, software bugs, operator mistakes, etc. Although being helpful to some extend in bug detection and diagnosis, these methods tend to focus on monitoring environments or configurations, instead of the problems in programs’ semantics.
CHAPTER 3

2NDSTRIKE

This chapter talks about manifesting hidden concurrency bugs in multi-threaded system software in software testing. Naturally, being able to catch bugs in testing phase is the most preferable way to deal with the bugs. However, testing is not always effective, especially when facing the concurrency bugs in multi-threaded programs. Unfortunately, for better performance on the emerging multi-core platforms and better responsiveness for serving multiple application software programs, more and more system software adopts multi-threaded design. As a result, the effectiveness and efficiency of testing multi-threaded system software is an increasingly problematic issue. In order to address this challenge, we propose a testing framework called 2ndStrike, which helps manifest a common type of hidden concurrency bugs, concurrency typestate bugs. It focuses on a concept in programming languages called typestate, and detects the concurrency bugs caused by the synchronization errors that could result typestate violation, e.g., accessing a file descriptor which was closed earlier by another unsynchronized thread. Based on runtime profiling interested objects and the associated operation events during program execution, 2ndStrike identifies candidates of concurrency typestate bugs and then re-execute the program with controlled thread scheduling for manifesting the bug candidates.

This chapter is organized as follows: In Section 3.1, we give an overview of the motivation and our method. In Section 3.2, we describe the background on typestate bugs. In Section 3.3
we present the main idea and the design of 2ndStrike. Then we describe the evaluation methodology and present experimental results in Section 3.4 and Section 3.5, respectively. After that, we provide a brief summary in Section 3.6.

3.1 Overview

Concurrency bugs are becoming increasingly prevalent in the multi-core era. This is mainly because more programs are written or rewritten in a multi-threaded fashion to better utilize multi-core systems. However, it is extremely difficult for developers to clearly reason all possible thread interleavings of program execution at the coding phase. Consequently, concurrency bugs (e.g., improperly synchronized accesses to shared resources) hidden in some corner cases of thread interleavings are inevitable, which often occur sporadically and cause runtime errors.

Due to their non-deterministic nature, concurrency bugs are notoriously hard to pinpoint at the testing phase. The \textit{de facto} way to expose and detect concurrency bugs is stress testing, i.e., running a multi-threaded program repetitively for a long time with test inputs. While easy to conduct, stress testing is not efficient due to large consumption of computing resources. Furthermore, stress testing is usually not very effective because it tends to only exercise a fraction of possible thread interleavings [55,56].

Much research effort has been spent on detecting common concurrency bugs including data races and atomicity violations. These works include data race detectors [37,40,131] and atomicity violation detectors [42–44]. They mainly focus on detecting memory accesses to shared variables that are not protected by common locks or not serializable under exercised thread interleavings. The philosophy is that programmers should protect the accesses to shared variables via locks and achieve atomicity in order to guarantee correctness. However,
these tools can only detect concurrency bugs that occur under exercised thread interleavings during program execution.

Recently, several active testing techniques [56,57,132] are proposed to expose data races or atomicity violations in multi-threaded programs even if the bugs do not manifest themselves during program execution. Based on information from external detectors or runtime profiling, these testing techniques actively manipulate the runtime thread scheduling to favor the interleavings that can trigger data races [57] or atomicity violations [56,132]. As a result, these techniques help expose data races or atomicity violations hidden in uncommon thread interleavings.

However, previous study has shown that large system programs contain a significant number of concurrency bugs that do not involve data races or atomicity violations [133]. Instead of accessing shared variables unprotected or in a unserializable way, many of these concurrency bugs are caused by accessing some general runtime resources in their illegal states. Figure 3.1 shows a real concurrency bug of this type in MySQL, a popular database server [134]. For many iterations of program execution, one thread (Thread 1) dereferences a pointer object and later another thread (Thread 2) invalidates the pointer by setting it to NULL. Due to improper synchronization, Thread 2 can set the pointer to NULL before Thread 1 dereferences the pointer, leading to a runtime error. This bug occurs even though both the dereference operation and the nullify operation are protected by the same lock and made atomic.

These concurrency bugs are related to typestates [5]. By definition, a typestate is associated to an object (or resource) and determines the subset of operations permitted on the object. In each typestate of an object, it is legitimate to apply some operations to the object but not others. For example, read operation is allowed to be applied to a file handler object when the object is in the typestate of OPEN, while read is not allowed for a CLOSED
file handler object. In multi-threaded programs, there may be multiple threads accessing an object, atomically or not, which may or may not change object typestates permittedly only in certain typestates. Concurrency bugs occur when the state manipulating thread is not well synchronized with other threads that access the same object, causing the object being applied with unpermitted operations in some typestates. In this chapter, we use the term *concurrency typestate bugs* to refer to this type of bugs.

Concurrency typestate bugs are important yet challenging to address. First of all, they are common mistakes in large system programs [133]. In addition, concurrency typestate bugs are often relevant to program semantics because it involves high-level program invariants. Furthermore, existing tools that help expose and detect data races or atomicity violations are not very effective in dealing with this type of bugs. Since these tools focus on the synchronization at memory access level, they do not take program semantics into consideration. Even there are data races or atomicity violations involved in a concurrency typestate bug, existing tools still provide few insights for understanding root causes of bugs since they do not capture high-level program invariants.

In this work, we propose a method called *2ndStrike* to manifest potential concurrency typestate bugs hidden in multi-threaded programs. Our main idea is to profile interested
objects and the associated operation events during program execution, identify candidates of concurrency typestate bugs based on the runtime profile, and then re-execute the program with controlled thread scheduling for manifesting the bug candidates.

Specifically, based on a state machine that specify object typestates and allowed operations, 2ndStrike performs three main steps. First, it monitors the typestates of objects at runtime and logs corresponding events, including operations performed on the objects and state transitions of the objects. For example, in Figure 3.1, 2ndStrike logs the events of pointer dereference in Thread 1 and pointer nullify events in Thread 2. In normal execution, the pointer dereference event often happens before the pointer nullify event so the bug is very difficult to be manifested.

Then 2ndStrike analyzes the logged events continuously fed from the profiling step while the program is still running. Specifically, based on the given state machine, 2ndStrike identifies potential bug candidates, i.e., operation pairs in two threads that may cause concurrency typestate bugs if the order of the operation pairs is reversed. In Figure 3.1, a potential bug candidate is a pointer dereference operation from one thread that occurs before the nullify operation on the same pointer from another thread.

In the last step, with the generated potential bug candidates, 2ndStrike re-executes the program and applies scheduling changes during the re-execution that can help manifest the bugs. In Figure 3.1, the scheduling change is to delay the pointer dereference event in Thread 1 and wait for the nullify event in Thread 2 to happen first. If successfully manifesting the bug, 2ndStrike reports the bug with detailed debugging information. If the bug cannot be manifested due to unmonitored synchronization events, 2ndStrike continues to try other bug candidates with the process above.
We have implemented a prototype of 2ndStrike framework on Linux, and used two common types of concurrency typestate bugs, i.e., invalid pointer dereference and lock operation violation, as examples. We have evaluated 2ndStrike with three real world bugs from three open-source server and desktop software programs (i.e., MySQL [134], Mozilla [135], pbzi2 [136]) on an eight-core machine. Our experimental results show that 2ndStrike can effectively and efficiently manifest all three tested software bugs. In addition, it provides detailed bug reports and can consistently reproduce the bugs after manifesting them.

In summary, our work has made following contributions:

- We have proposed a general solution to manifest program semantics related concurrency bugs, i.e., *concurrency typestate bugs*, in software testing. To the best of our knowledge, 2ndStrike is the first method for such purpose.

- 2ndStrike is effective. It is able to manifest all three tested concurrency bugs, which are difficult to trigger by stress testing.

- 2ndStrike is efficient. Equipped with a stand-alone analyzer running in parallel with the testing program, it analyzes the streaming event logs and performs bug manifestation. Furthermore, with targeted instrumentation, it only needs to instrument a small set of events, which is lighter weight than many existing tools that instrument every memory accesses.

- 2ndStrike is easy to use. It does not require source code changes and can be easily executed with existing test suites packaged with the software. In addition, it does not report false positives. Moreover, it provides reproducibility as well as a bug report to help developers diagnose the manifested bugs.
3.2 Background: Typestate Bugs

First introduced in [5], typestate is a temporal extension of the concepts of types in programming languages. More specifically, typestate defines a finite set of states and the state transitions corresponding to different stages of objects' lifetime along with permitted operations on objects with certain states. For example, a file handler object can be in two states, i.e. \texttt{OPEN} if the handler refers to an open file and \texttt{CLOSED} if the handler refers to a closed file. Operations such as \texttt{read} and \texttt{write} are only permitted to be applied to an \texttt{OPEN} file handler, not a \texttt{CLOSED} file handler.

Much research has been done on detecting typestate bugs in programs. Some approaches detect typestate bugs via program analysis [5, 137, 138] or symbolic execution [139], while others check typestate properties at runtime [140–143]. These approaches are proposed for detecting typestate bugs in sequential programs.

Situation gets more complicated when concurrency meets typestate bugs. In this work, \textit{concurrency typestate bugs} are typestate bugs caused by ill-synchronized threads in multi-threaded programs. In general, concurrency typestate bugs are difficult to deal with due to several reasons. First, when involving multiple threads, the possible sequences of state transitions can grow much faster than single thread cases. This often results in negligence when programmers try to enforce the program ordering invariants. Additionally, in relatively well tested programs, the operations performed on the objects are mostly in the correct orders, even if when the order is not strictly enforced. This makes the corner case bugs extremely hard to manifest in testing and easily slip into production runs. Furthermore, traditional typestate bug detection tools are not effective to detect concurrency typestate bugs. On one hand, it is challenging for static tools [137–139] to model threading very well and thus they may report a huge number of false positives. On the other hand, dynamic
tools [140–143] are only effective to the bugs actually manifested in the execution, which itself is often not easy in limited testing environments.

3.3 Our Approach: 2ndStrike

Different types of concurrency typestate bugs are related with different aspects of semantics of the program. In order to manifest a specific type of concurrency typestate bugs, 2ndStrike needs a state machine to capture the typestate property and identify what thread-interleaving conditions can cause typestate violation (i.e., potential concurrency typestate bugs). Given a state machine for a certain type of concurrency typestate bugs, 2ndStrike performs three steps to manifest the target bugs. As shown in Figure 3.2, 2ndStrike first performs profiling to monitor the typestates of runtime objects and generate logs. As logs being generated, the 2ndStrike analyzes the logs on-the-fly to generate bug candidates. A bug candidate is a pair of operations from different threads that can cause concurrency typestate bugs if the order of the operation pairs is reversed. After finishing the steps of profiling and analysis, 2ndStrike performs multiple additional executions with controlled thread scheduling to manifest the potential bugs indicated by the bug candidates. If a bug is successfully
manifested, 2ndStrike generates a bug report and remembers the bug candidate information to help programmers later reproduce the bug.

In the rest of this section, we first present state machines for two types of common concurrency typestate bugs in Section 3.3.1, followed by the design and implementation of the three components (i.e., Profiler, Analyzer, and Manifestor) that perform the above-mentioned three steps in Sections 3.3.2-3.3.4. Finally, we discuss several key design issues.

3.3.1 State Machines for Detecting Concurrency Typestate Bugs

There are several ways to provide state machines for detecting concurrency typestate bugs. Generally, programmers can provide such state machines since they know high-level program semantics better than others. To reduce programmers’ manual effort, researchers have proposed static methods [144] as well as dynamic methods [145] for automatically inferring the likely typestate property of objects. Furthermore, we can derive certain state machines based on common programming rules, such as “cannot read on a closed file handler” and “cannot dereference an invalid pointer”.

In this work, we derive two state machines for two types of concurrency typestate bugs, i.e., concurrent invalid pointer dereference and concurrent invalid lock operation, respectively,
as examples to illustrate how 2ndStrike works. Figure 3.3 and Figure 3.4 show the state machines for pointers and locks, respectively. They are relatively simple state machines good for illustration purposes.

As shown in Figure 3.3, a pointer has three possible states: **VALID**, **DANGLE**, and **NULL**. A pointer in its **VALID** indicates it points to an object in memory. In this state, there are four operations legal to be applied on the pointer:

- **Dereference**: Dereference the pointer, such as access a field in a structure.

- **Assign Valid**: Assign the pointer using the memory address of a valid object in memory. Examples include allocating a new memory object.

- **Free**: Free the object pointed to by the pointer. This operation will affect states of other pointers pointing to the same object.

- **Assign NULL**: Assign the NULL value to the pointer.

Among these operations, **Dereference** and **Assign Valid** do not cause state change of the pointer. **Free** causes the pointer state changing to **DANGLE**, i.e., becoming a dangling pointer. Similarly, **Assign NULL** causes the pointer state changing to **NULL**, i.e., becoming a NULL
pointer. In DANGLE and NULL states, the legal operations on the pointer only contains Assign Valid and Assign NULL, while Dereference and Free are illegal. Attempting to perform an illegal operation (e.g., Dereference in the NULL state) indicates a runtime error, such as NULL pointer dereference, which is likely caused by a bug.

Similarly in Figure 3.3, a lock has three states: INVALID, UNLOCKED, and LOCKED. A lock in state of INVALID, meaning it is not ready to use, can be initialized (via operation Initialize) and change to UNLOCKED state. The states UNLOCKED and LOCKED can change back and forth via operations Acquire and Release, respectively. In addition, a lock in either of LOCKED or UNLOCKED states can be invalidated via an operation Invalidate (e.g., destroying a lock) and change its state back to INVALID.

In multi-threaded programs, synchronization errors can happen when performing state-changing operations on runtime objects. As shown in the example in Figure 3.1, the order between the operation Dereference by thread Thread 1 and the operation Assign NULL by thread Thread 2 is not strictly enforced. Therefore in some rare cases, the pointer state could first change to NULL before the operation Dereference is applied, resulting in a typestate violation.

### 3.3.2 Runtime Profiling to Monitor Typestate

In the profiling step, 2ndStrike monitors the typestates of runtime objects in multi-threaded programs. More specifically, 2ndStrike logs the state transition information and operations made on objects for later analysis.

The runtime profiling is typestate-specific since it is for manifesting specific types of concurrency typestate bugs. For example, for invalid pointer dereference bugs, 2ndStrike logs the following runtime events: a) creation/destroy of runtime objects, such as malloc and free, to determine whether a memory address is valid or not, b) assignment to pointers,
and c) dereference of pointers. Similarly, for invalid lock operations, 2ndStrike logs: a) initialization/invalidation of locks, and b) lock acquire/release. The runtime overhead for profiling largely depends on the specific typestate being monitored. Since the profiling is targeted, its overhead is in general less than the overhead for monitoring each memory access (See Section 3.5 for details).

Additionally, 2ndStrike logs general synchronization events to prune false bug candidates in later analysis step (Section 3.3.3). These events include the following categories: a) thread fork/join, b) lock activities, and c) semaphore or condition variable activities.

In addition to logging, 2ndStrike needs to force specific scheduling order between events at the bug manifestation step (Section 3.3.4). For this purpose, we introduce the concept of instrumentation point (InstPt) to unify the logging and thread scheduling control. An InstPt is a specific source code location where 2ndStrike either emit a log entry or insert delay depending on the steps. In current prototype of 2ndStrike, we use a compile framework called LLVM [146] to statically instrument programs.

Each log entry profiled by 2ndStrike includes the following information: a) event type, such as pointer dereference and lock release, b) the thread ID on which the event happens, c) additional event information, e.g. to indicate which pointer is dereferenced or to indicate which lock has been released, d) the InstPt corresponding to the event, and e) the callsite to indicate the runtime context of the event.

3.3.3 On-the-fly Analysis to Identify Bug Candidates

With runtime logs, the analyzer performs log analysis in order to identify the bug candidates. We first present criteria for potential concurrency typestate bugs, followed by algorithms for identifying bug candidates, then pruning and ranking methods used in 2ndStrike.
Criteria for potential concurrency typestate bugs. To manifest concurrency typestate bugs, 2ndStrike needs to identify the potential problems in synchronization between state-changing operations and the operations that are only permitted on a subset of states.

Specifically, suppose we have a state machine with three states $S_p$, $S_q$, and $S_r$. In this state machine, operation $O_x$ is allowed to be applied only on $S_p$, while operation $O_y$ transfers state $S_p$ to state $S_q$ and operation $O_z$ transfers state $S_r$ to state $S_p$. Figure 3.5 shows this state machine. In this scenario, a bug candidate can be operation pairs of $O_x$ by a thread and $O_y$ by another thread, where $O_x$ occurs earlier than $O_y$. This is because the order of $O_x$ and $O_y$ from two threads can be potentially switched and the order switch will cause concurrency typestate violation. Similarly, a bug candidate can also be operation pairs of $O_z$ by a thread and $O_x$, where $O_z$ occurs earlier than $O_x$, since their order switch will cause concurrency typestate violation.

Taking the pointer typestates as an example, the Dereference operation is only permitted on the state VALID but not on the state NULL, and the Assign NULL operation changes the pointer state from VALID to NULL. Therefore, if there is a runtime operation Dereference on a pointer by one thread, followed by another operation Assign NULL by another thread,
and the orders of them can potentially be switched, there exists a invalid pointer dereference bug.

**Identifying bug candidates.** Each bug candidate includes the information on the pair of events corresponding to the operations needed to be reordered. The purpose of such information is to allow 2ndStrike later accurately identify the runtime operation pair to reorder. We call the two events in each bug candidate *preceding event* and *subsequent event*, reflecting the order they have in normal execution. For each event, it includes the InstPt and the callsite information. Additionally it can be configured to include the thread id and object id as well for stricter matching but since thread id and object id can easily change across multiple runs, 2ndStrike does not use them by default.

For many typestate monitoring there are potentially a huge number of log entries emitted at runtime. The log storage can be an issue and may cause significant performance degradation of testing. To increase the efficiency, we devise an on-the-fly design in which the log is consumed by the analyzer as soon as being generated. In this design, the analyzer runs in parallel with the program being profiled and uses a stream processing algorithm to analyze the runtime logs. In this algorithm, the event information is stored in hash tables indexed by both thread id and object id. The lockset and vector clock for the each log entry is maintained by the analyzer as well. As a new log entry comes in, the analyzer can quickly look up the table and find related events to analyze together. The old event information is periodically cleaned up to limit the memory usage and maintain efficiency of table lookup. In addition, the analyzer holds a buffer of log entries that have just been received to avoid the delay on profiling execution due to jitter in logging. Only when the buffer is full, the profiling execution will temporarily block to wait for the analyzer. With this design, the runtime logs only stay in memory and only a small list of bug candidates are written to disk.
**Pruning false bug candidates.** With identified bug candidates, some operation pairs cannot be re-ordered. One main reason is strict orders between operation pairs enforced by thread synchronization events. To filter out such strictly-ordered operation pairs, 2nd-Strike uses similar techniques as previous works [56]. Specifically, 2ndStrike profiles general synchronization events, such as thread fork/join and condition variable wait/signal, and computes a vector clock associated with each operation based on these events and happens-before relation between operation pairs. Certainly, it is possible for threads to synchronize in customized ways that are not generally traceable, such as via atomic operations on shared variables. (See Section 3.3.5 for details)

### 3.3.4 Scheduling Control to Manifest Bugs

After a list of bug candidates being generated, 2ndStrike performs one round of execution for each bug candidate to try to manifest the bug. In other words, 2ndStrike attempts to reorder the preceding event and the subsequent event in this execution by inserting delay before the preceding event.

Specifically, during a bug manifesting run, 2ndStrike first checks the operation at each InstPt against the preceding event in the bug candidate by comparing their InstPt ids and callsites. If their InstPt ids and callsites match, 2ndStrike blocks the thread on a semaphore, lets other thread proceed as normal, and looks for the matching subsequent event at each InstPt. In addition to matching InstPt ids and callsites with the subsequent event in bug candidate, the object id in the operation has to match the one in the delayed operation. If a match is found, 2ndStrike unblocks the blocked thread and let it commit the typestate violation. Moreover, 2ndStrike records the runtime information at this point for generating bug report.
It is also common that the matching subsequent event never happens after one thread is blocked before the preceding event. This is mainly because other synchronization mechanisms such as flag variables are used in the program to prevent such reordering. In this case, 2ndStrike will timeout after a certain threshold, unblock the blocked thread, and keep looking for the preceding event. After a certain number of timeouts, 2ndStrike aborts the reordering attempt and stops this execution.

### 3.3.5 Issues and Discussion

**Un-monitored synchronization.** Similar to previous active testing tools [56, 57], 2ndStrike cannot monitor all possible synchronizations happened in the program execution, especially the ones based on shared flag variables. This will cause 2ndStrike analyzer to generate some operations pairs that are impossible to be reordered as bug candidates. Although these candidates will be pruned later since 2ndStrike cannot manifest them, this do reduce the test efficiency. To address this issue, we can either ask developers for hints on their customized synchronization mechanisms or derive such information.

How to more accurately capture the synchronization behavior in multi-threaded program is a research problem and will be left in our future work.

**Non-determinisim.** Multi-threaded programs are naturally non-deterministic in their execution, which means the profiling result based on previous executions may not be valid for subsequent executions. Same as previous active testing tools [56, 57], 2ndStrike assumes for one input, the code executed at different runs are mostly the same. In practice, unfortunately, there are certain degree of non-determinism involved in program executions especially for large complex software. The result is that in some program executions that are very different from the profiling execution, 2ndStrike may fail to manifest the concurrency typestate
bugs because the object is not operated in the same way. To address this issue, we can integrate 2ndStrike with existing deterministic replay tools [104,147].

### 3.4 Evaluation Methodology

We have implemented a prototype of 2ndStrike on Linux and conducted our experiments on a Intel Xeon machine with eight 2GHz processing cores and 16GB memory. The Linux kernel version is 2.6.16. We have evaluated the manifesting capability of 2ndStrike for two types of concurrency typestate bugs used as examples (i.e., invalid pointer dereference and invalid lock operation). The software benchmarks include three multi-threaded open-source software packages (MySQL, Mozilla and pbzip2) with three real-world bugs, as shown in Table 3.1. For MySQL, we focus on testing its Falcon storage engine, which is a new storage engine for highly concurrent workloads, and for Mozilla, we focus on testing its JavaScript engine, JS.

To evaluate the effectiveness, for each bug, we ran 2ndStrike with the input which can potentially trigger the bug with certain thread interleavings. When possible, we chose to use the default testing framework for the program such as mysql-test, and the default testing inputs such as JavaScript files provided in the Mozilla JavaScript engine release package. This allows us to evaluate how 2ndStrike integrates with the existing test suites, which we consider as an important issue for usability. For pbzip2, since it does not have test inputs

<table>
<thead>
<tr>
<th>App.</th>
<th>Bug id</th>
<th>Bug description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>MySQL-ptr</td>
<td>An invalid pointer dereference causing crash when handling transactions</td>
</tr>
<tr>
<td>Mozilla</td>
<td>Mozilla-ptr</td>
<td>An invalid pointer dereference causing crash when destroying contexts</td>
</tr>
<tr>
<td>pbzip2</td>
<td>pbzip2-lck</td>
<td>An invalid lock usage causing crash when decompressing a file</td>
</tr>
</tbody>
</table>

Table 3.1: Evaluated applications and concurrency typestate bugs
included in the package, we used a medium-size file as input (193KB bz2 file decompressed into 5MB text file). For comparison, we also performed a stress testing by running the tested programs with the same input repetitively.

To evaluate the efficiency of 2ndStrike, we have conducted experiments for measuring both profiling/analyzing overhead, and bug manifesting overhead (i.e., for controlled thread interleavings). For profiling/analyzing overhead, we measure the program execution time under the following four configurations:

- Baseline, executing the programs without 2ndStrike.
- 2ndStrike’s profiling only, executing the programs with 2ndStrike’s profiling functionality for both types of concurrency typestate bugs and logs not recorded (fed into /dev/null).
- 2ndStrike’s profiling and analyzing, executing the programs with 2ndStrike profiling turned on along with analyzer for stream log processing.
- Memory access profiling, executing the programs with profiling every memory access and logs not recorded. This is for comparing 2ndStrike’s profiling with data race or atomicity violation detectors’ profiling.

For bug manifesting overhead, we measure the execution time for a manifesting run with the bug successfully manifested, and the execution time for bug manifesting run without bug being manifested, e.g., the tested bug candidate does not corresponding to a real bug.

In addition, we have evaluated each real bug’s reproducibility. With the bug candidate identified by 2ndStrike after initial successful manifestation, we execute the 2ndStrike in manifesting mode for multiple times to measure how well the bug can be reproduced. The reproducibility is important to developers when they try to understand and fix a bug.
3.5 Experimental Results

3.5.1 Effectiveness

<table>
<thead>
<tr>
<th>Bug id</th>
<th>2ndStrike</th>
<th>Stress</th>
<th>Data race directed</th>
<th>Atomicity violation directed</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL-ptr</td>
<td>255</td>
<td>45,535</td>
<td>Maybe</td>
<td>Yes</td>
</tr>
<tr>
<td>Mozilla-ptr</td>
<td>45</td>
<td>No</td>
<td>Maybe</td>
<td>No</td>
</tr>
<tr>
<td>pbzip2-lck</td>
<td>1</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
</tbody>
</table>

Table 3.2: Overall effectiveness of 2ndStrike: The ‘2ndStrike’ and ‘Stress’ columns show the time in seconds needed to manifest the bug (No means bug not manifested in 24 hours). The last two columns show the code analysis results on whether the data race and atomicity violation directed tools can effectively manifest the bug.

Table 3.2 describes our evaluation results on effectiveness of 2ndStrike. The second column shows the overall time in seconds needed for manifesting the tested bug, including profiling, analysis, and bug manifesting execution. The third column shows the time in seconds needed to manifest the bug using stress testing. ‘No’ means the bug cannot be manifested by stress testing after executed for 24 hours.

Our results show that 2ndStrike is effective in manifesting the tested concurrency type-state bugs. With 2ndStrike, all three tested bugs are manifested within 255 seconds. On the contrary, the stress testing cannot manifest two out of the three bugs (Mozilla-ptr and pbzip2-lck). For MySQL-ptr, stress testing is 178 times slower than 2ndStrike for manifesting the bug.

The forth and fifth columns show the code analysis results on whether the data race and atomicity violation directed tools can effectively manifest the bug. ‘Maybe’ means there could be data races or atomicity violations related with the bug-triggering object, however
just triggering any data race or unserializable access interleaving does not guarantee the manifestation of the bug. This is because there are usually non-trivial number of accesses on the bug-triggering objects considered as data races or unserializable accesses. Since the bug can be manifested only when specific orders are met in the accesses, only a subset of cases are effective in manifesting the bug. Furthermore, if the data race or atomicity violation directed tools failed to manifest the bug, they may even categorize the bug as ‘benign’.

The main reason why 2ndStrike can be effective in manifesting these bugs is that it focuses on the semantic errors instead of only on memory accesses. Taking the Mozilla-ptr bug as an example. This bug was caused by an ‘order violation’ in using and destroying runtime objects in `js_DestroyContext`, as analyzed in a previous study [133]. Since it does not involve clear unserializable accesses to shared variables, atomicity directed bug manifesting tools are not effective in manifesting it.

### 3.5.2 Efficiency

![Graphs showing efficiency of 2ndStrike](image)

Figures 3.6: Efficiency of 2ndStrike: ‘N’ means normal execution time without 2ndStrike being applied; ‘P’ means execution time for 2ndStrike profiling only; ‘PMA’ means execution time for profiling memory accesses, with logs fed into `/dev/null`; ‘PA’ means execution time for 2ndStrike profiling and on-the-fly analysis. ‘MS’ means execution time for bug manifesting run successfully manifests the bug; ‘MP-Min’, ‘MP-Max’, and ‘MP-Avg’ mean the minimal, maximal, and average execution time for bug manifesting run which passes without manifesting any bug, respectively.
Figure 3.6 (a), (b), and (c) show the detailed performance results when using 2ndStrike to manifest the three concurrency typestate bugs, respectively.

We can observe that 2ndStrike cause tolerable runtime overhead for profiling and analysis for software testing purposes. The reason for its profiling overhead (column ‘P’ versus column ‘N’, ranging from 2x to 88x) is two-fold. Firstly, to monitor the pointer typestate, every pointer dereference event is logged, which is quite intensive. For some other typestates such as lock and file descriptor, the number of events needed to be monitored is order-of-magnitude less. In addition, in order to emit runtime events to a single analyzer, all threads need to hold a common lock for doing so, which becomes a bottleneck in profiling executions. Even this, 2ndStrike’s profiling overhead is still less than the overhead for logging every memory access (column ‘PMA’).

Moreover, the results show that 2ndStrike’s stream processing design greatly helps reduce the profiling and analysis overhead (see column ‘PA’ versus column ‘P’). We measured the total number of logged events and the size of logs if dumped to disk. They can be huge for large software. For example, for executing the input on MySQL Falcon storage engine, whose normal execution only takes 6 seconds, the execution generates 31,539,852 events to log and it takes 5.6 GB disk space to store the log. Without using the stream processing design, the overhead for just writing the logs to disk would be very high, causing the overall profiling/analysis time orders of magnitude larger.

Furthermore, the results indicate that overhead for bug manifesting execution is very small. The last four bars in Figure 3.6 show the execution time for a successful manifesting run with the concurrency typestate bug manifested (column ‘MS’) and the minimal/maximal/average execution time for a manifesting run that fails to manifest any bug (columns ‘MP-Min’, ‘MP-Max’, and ‘MP-Avg’). We can see that the execution time for bug manifestation is generally very small because the lock contention issue mentioned above does
<table>
<thead>
<tr>
<th>App.</th>
<th>Sync</th>
<th>Pointer</th>
<th>Lock</th>
<th>Total 2ndStrike</th>
<th>Load</th>
<th>Store</th>
<th>Total Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL Falcon</td>
<td>1,476</td>
<td>52,045</td>
<td>1,449</td>
<td>53,521</td>
<td>43,864</td>
<td>13,676</td>
<td>57,540</td>
</tr>
<tr>
<td>Mozilla JS</td>
<td>124</td>
<td>18,072</td>
<td>122</td>
<td>18,196</td>
<td>124,552</td>
<td>47,666</td>
<td>172,218</td>
</tr>
<tr>
<td>pbzip2</td>
<td>85</td>
<td>483</td>
<td>74</td>
<td>578</td>
<td>982</td>
<td>203</td>
<td>1,185</td>
</tr>
</tbody>
</table>

Table 3.3: Detailed statistics of instrumentation by 2ndStrike: Number of instrumentation points are shown. ‘Sync’ means the instrumentation for synchronization; ‘Pointer’ means the instrumentation for detecting invalid pointer dereference; ‘Lock’ means the instrumentation for detecting invalid lock operations; ‘Total 2ndStrike’ means total instrumentation made by 2ndStrike. ‘Load’, ‘Store’, ‘Total Access’ mean the instrumentation of memory reads, writes, and all memory accesses, respectively.

<table>
<thead>
<tr>
<th>Bug id</th>
<th>Sync</th>
<th>Pointer</th>
<th>Lock</th>
<th>Total 2ndStrike</th>
<th>Load</th>
<th>Store</th>
<th>Total Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL Falcon</td>
<td>1,534,786</td>
<td>30,905,066</td>
<td>1,482,286</td>
<td>31,539,852</td>
<td>31,029,421</td>
<td>15,721,396</td>
<td>46,750,817</td>
</tr>
<tr>
<td>Mozilla JS</td>
<td>219,444</td>
<td>6,476,891</td>
<td>107,646</td>
<td>6,696,335</td>
<td>45,442,055</td>
<td>16,134,868</td>
<td>61,576,923</td>
</tr>
<tr>
<td>pbzip2</td>
<td>77</td>
<td>96,795</td>
<td>53</td>
<td>96,872</td>
<td>193,209</td>
<td>242</td>
<td>193,451</td>
</tr>
</tbody>
</table>

Table 3.4: Detailed statistics of logged runtime events by 2ndStrike, Meanings of columns are same as for Table 3.3

not hold for bug-manifesting executions. The reason why in some cases the execution time for successfully manifesting the bug is shorter than the normal execution time is that when the bug manifested, the program will crash in the middle without finishing the execution.

Moreover, the results indicate that by only instrumenting the relevant runtime events, 2ndStrike help reducing the runtime overhead. Table 3.3 shows the detailed statistics of instrumentation by 2ndStrike for each application. For example for Mozilla JS, 2ndStrike totally has 18,196 InstPts, while to trace all memory accesses, 172,218 places need to be instrumented, which is 9 times higher. The less instrumentation directly result in less runtime events need to be profiled as shown in Table 3.4. For Mozilla JS, the 2ndStrike profiling is 9 times less than the profiling for all memory accesses.
3.5.3 Usability

Integration with testing framework. 2ndStrike can be easily integrated with existing testing framework and test inputs. It does not require source code change or special dynamic instrumentation environments. All it requires is changing the build scripts for the software component. In addition, it is flexible to be applied to only a subset of components in a large software package. In our experiments with MySQL and Mozilla, we only apply 2ndStrike to MySQL’s Falcon storage engine and Mozilla’s JavaScript engine. We consider that this flexibility can help developers perform more targeted testing and is very important for large software with many components.

Bug reporting. After manifesting a bug, 2ndStrike generates an accurate and detailed bug report, which includes following pieces of information: a) the failed test input, b) the typestate violation manifested, c) the bug candidate, and d) the runtime ‘preceding event’ and ‘subsequent event’ that happened during bug manifesting execution. Note that since the bug is manifested when the order between ‘preceding event’ and ‘subsequent event’ is flipped, the ‘subsequent event’ actually happens first in the buggy run while the ‘preceding event’ happens after. For both events, the bug report includes: a) the InstPt information, which corresponds to a unique source code location, b) the runtime callsite, including multiple levels of function names, c) the thread id, and d) the object id. This information can be very helpful for developers to diagnose the bug and pinpoint the root cause.

No false positive. In addition, 2ndStrike reports no false positive in its final results. This is because all bug candidates that are not corresponding to real bugs are pruned automatically in the bug manifesting executions. Only the bug candidates that can trigger real concurrency typestate bugs are reported in the final results. This feature is also critical to developers, who are generally very reluctant to spend efforts in vain.
<table>
<thead>
<tr>
<th>Bug id</th>
<th>Probability to manifest by 2ndStrike</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL-ptr</td>
<td>88.24%</td>
</tr>
<tr>
<td>Mozilla-ptr</td>
<td>100%</td>
</tr>
<tr>
<td>pbzip2-lck</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 3.5: Reproducibility of the manifested concurrency typestate bugs in 2ndStrike

**Bug reproducing.** Moreover, 2ndStrike helps developers further by making reproducing the bug easier. As shown in Table 3.5, the candidate of concurrency typestate bug can be reproduced by 2ndStrike with a high probability after it is being manifested once. The reason why it is not 100% reproducible is the non-determinism (see Section 3.3.5 for details). In some of these multi-threaded program executions, the exercised code path is different from the profiling run.

### 3.6 Summary

In summary, this chapter presents an effective method called 2ndStrike for manifesting concurrency typestate bugs in multi-threaded programs at testing phase. To do so, 2ndStrike profiles key runtime events that are related to object typestates described by a given state machine and synchronization events. Then 2ndStrike identifies bug candidates by checking the profiled events against the state machine with different orders of possible thread interleavings. Finally 2ndStrike re-executes the program with controlled thread interleaving for manifesting bug candidates.

Our evaluation of 2ndStrike with three real world bugs from three open-source server and desktop programs (i.e., MySQL, Mozilla, pbzip2) on an eight-core machine has shown that 2ndStrike can effectively and efficiently manifest the three tested software bugs, i.e., within 255 seconds, at least 178 times faster than stress testing. Additionally, 2ndStrike does not
generate false positive, provides detailed bug reports, and can consistently reproduce the bugs after manifesting the bugs once. Therefore, it offers great assistance for developers to diagnose and fix the bugs as well.
CHAPTER 4

DMTRACKER

This chapter talks about runtime monitoring and anomaly detection in distributed system software during production runs. No matter how carefully the testing is conducted before release, inevitably any non-trivial program especially system software will contain bugs after deployment. Sooner or later the bugs will be triggered causing the system software to exhibit abnormal behaviors. Both operators and developers depend on runtime monitoring and anomaly detection techniques to detect the errors and hopefully find the cause of the bugs. Specifically for system software that supports parallel programs running on distributed machines like clusters, it is quite challenging to effectively collect useful execution information and use the information to find bugs. To address this challenge, we propose a light-weight tool called DMTracker focusing on the data movements among parallel processes. Based on the observation that data movements in parallel programs typically follow certain patterns, our idea is to extract data movement (DM)-based invariants at program runtime and check the violations of these invariants. These violations indicate potential bugs such as data races and memory corruption bugs that manifest themselves in data movements. Based on this idea, DMTracker automatically extracts DM-based invariants and catch the anomalies in parallel program execution.

This chapter is organized as follows: In Section 4.1, we give an overview of the motivation and our method. In Section 4.2, we describe the background on statistical rule based bug
detection. In Section 4.3 and Section 4.4, we present the main idea of (DM)-based invariants and the design of DMTracker, respectively. Then we discuss the evaluation results in Section 4.5, followed by a brief summary in Section 4.6.

4.1 Overview

Bugs in system software greatly affect the reliability of high performance systems. A recent study [4] with more than 20 different high-performance computer systems at Los Alamos National Laboratory has shown that software bugs account for as many as 24% of failures. Furthermore, software bugs may silently corrupt application data and remain unnoticed until termination of the whole task, which leads to incorrect results and significantly affects overall productivity. With peta-scale and many-core architectures becoming a mainstream in High Performance Computing (HPC) systems in the predictable future, software will become much more complex and software bugs may cause more frequent and more severe system problems.

Unfortunately, finding software bugs in high-performance systems is a daunting task due to the systems’ inherent nature of non-determinism and large scale. Bugs such as data races manifested during one execution may not be triggered during another execution because of various non-deterministic events, such as process execution orders, thread interleaving, signal delivery timing, I/O events, etc. Such non-determinism makes it difficult to reproduce bugs and thus renders a significant challenge for detecting and locating a software bug. Furthermore, the problem becomes much more complicated due to the ever-increasing scale of high performance systems. For example, some software bugs can only be triggered in very large-scale systems. However, it would cause a huge resource waste if developers have to occupy the whole large-scale system for inefficient manual debugging. Therefore, it is
imperative to have low-overhead tools deployed in production runs to help automatically locate software bugs.

Previous work on detecting bugs at execution time can be classified into two categories: *programming-rule-based* approach and *statistics-rule-based* approach [28]. While methods in both categories check certain rules during program execution, they focus on different types of rules. Programming-rule-based approaches detect violations of rules imposed by specific languages such as C/C++ or specific interfaces such as Message Passing Interface (MPI). For example, “bounds of the message buffer cannot exceed its allocated bounds” and “all members of one process group must execute collective operations over the same communicator in the same order” are two rules used by MPI-CHECK [121] and Umpire [119], respectively. Much research has been conducted in this category, such as Purify [21], Valgrind [22], Umpire [119], MARMOT [120], MPI-CHECK [121], etc. While they are effective in detecting some types of software bugs at runtime, their rules are either too general to be able to detect semantics-related bugs [21,22] or highly dependent on domain-specific expertise and human efforts [119–121].

Statistics-rule-based methods extract rules statistically at program runtime, and check the violations of the extracted rules. Statistical rules, or dynamic invariants [33,34], are properties that likely hold at a certain point or points in a program. One can extract invariants at runtime over a single run or across multiple runs. Several recent works [28,33,34] have demonstrated that statistics-rule-based approaches are very promising due to their effectiveness in detecting bugs that do not violate any programming rules. Daikon [33] and DIDUCE [34] extract value-based invariants (i.e., the possible values of a given variable are always within a certain range) at runtime and can detect bugs that generate abnormal values. Similarly, AccMon [28] captures the runtime program-counter-based invariants (i.e. a given variable is typically accessed by only a few instructions), and use them to detect
memory-related bugs. Many statistical rules extracted from program execution are related to program semantics, which are usually not accurately documented by programmers and difficult to infer from the code itself [148].

Parallel or distributed programs, the common type of most applications running on HPC systems, are especially suitable for applying statistics-rule-based methods. In addition to the temporal dimension explored by previous works [28,33,34] (i.e., invariants based on program behaviors in multiple runs or multiple phases of a single run), one can explore the spatial dimension (i.e., invariants based on the behaviors of multiple concurrently-running processes) in parallel systems. For example, in scientific parallel applications, usually similar or even identical tasks are performed in multiple iterations (the temporal dimension), as well as by multiple processes (the spatial dimension). Similarly, in some commercial HPC systems such as web server farms, a group of processes concurrently handle tasks in the same way for achieving high throughput.

Recently, Mirgorodskiy et al. have conducted an initial study of using statistics-rule-based approach to diagnose software problems in large-scale systems [35]. Their approach extracts the invariant of function time distribution in control flow, and then identify the abnormal process among a large number of parallel processes. They have shown that this invariant is effective for locating problematic processes and functions. However, their approach has two limitations in detecting software bugs. First, they cannot detect bugs that do not cause abnormal function time distribution across multiple processes. For example, data corruption may only cause wrong results without affecting the function time distribution. Similarly, some bugs manifest themselves in all the processes, resulting in the same distorted function time distribution for all the processes, and thereby their approach cannot detect these bugs. Second, the function time distribution invariant is easily interfered by system-level noises [149] (e.g., process scheduling, signal delivery, network congestion),
and thus, it does not reflect the semantics of programs very accurately. For example, the processes performing identical tasks on different nodes can show very different function time distribution if the network traffic load is unbalanced across nodes.

In this work, we propose a novel statistics-rule-based technique, called data movement (DM)-based invariants, to find hard-to-detect software bugs that can cause severe problems such as data corruptions and deadlocks in large-scale parallel programs. Our idea is based on the observation that data movement in parallel programs typically follow certain patterns. If we can extract such DM-based invariants at runtime, it is possible to detect abnormal data movements that are caused by potential software bugs.

Our idea is inspired by the fact that data movements in parallel programs are pervasive and bug-inducing. Different from sequential programs, parallel programs require multiple processes to coordinate with each other to perform large tasks. Therefore, processes in parallel programs usually communicate with each other very frequently. Unfortunately, programmers can easily make mistakes in various situations when performing data movements. On the application level, many parallel algorithms require data to be exchanged in non-trivial ways due to subtle boundary condition handling. In addition, parallel programming models, such as MPI, provide a variety of communication interfaces with different semantics such as point to point send/receive, collective calls, etc. Often, it is difficult for application programmers to precisely understand the subtle semantic differences and choose the correct interface. Furthermore, to achieve better performance, the applications and libraries may introduce aggressive and error-prone optimizations, which may work correctly for most of the time but affecting correctness in some corner cases. Therefore, bugs can be easily introduced and manifested in pervasive data movements in parallel programs.
More specifically, we propose two types of DM-based invariants: frequent chain (FC)-invariants, i.e., frequently occurring data movement chains, and chain distribution (CD)-invariants, i.e., clusters of data movement chain distributions of multiple processes. FC-invariants and CD-invariants focus on temporal and spatial similarity of data movements in parallel programs, respectively. The violations of them, abnormal data movement chains for FC-invariants or outlier data movement chain distribution in one process for CD-invariants, may indicate the potential software bugs such as data corruptions, livelocks, deadlocks, etc. Note that these two types of invariants are based on data movement chains formed by linking individual data movements where the destination of one data movement is the source of a subsequent data movement. The reason for doing so is that data movement chains reflect better semantic information than individual data movements.

Based on these ideas, we have built a tool, called DMTracker, to extract FC-invariants and CD-invariants at runtime and check for violations of them. Our experiments with two real-world bug cases in MVAPICH/MVAPICH2 [6], a popular MPI library, have shown that DMTracker can effectively detect them and report abnormal data movements to help programmers quickly diagnose the root causes of bugs. Complementary to existing programming-rule-based and statistics-rule-based tools, DMTracker has the following unique advantages, some or all of which are unavailable in other tools.

- To the best of our knowledge, DMTracker is the first automatic tool that utilizes statistical rules based on data movements (i.e., DM-based invariants) for detecting hard-to-detect software bugs that can cause severe problems such as data corruptions and deadlocks in parallel programs. To find software bugs, it focuses on a key aspect in parallel programs – the data movements. Based on DM-based invariants, DMTracker can detect various types of severe bugs such as data corruptions and deadlocks that
manifest themselves in data movements and help programmers to diagnose the root causes.

- DMTracker can detect both deterministic and non-deterministic software bugs that manifest themselves in only a few processes or across all processes. This is because it extracts invariants by exploring both temporal and spatial similarities of data movements. Our experiments have shown that DMTracker is effective in detecting both deterministic bug and non-deterministic bugs. In contrast, previous work [35] cannot handle software bugs manifested across all the processes, which is demonstrated by our first bug case.

- DMTracker can detect software bugs that do not violate the function time distribution since DM-based invariants capture “how data move” instead of “when data move.” Our experiments with the second bug case have shown that the problematic function time distribution can be easily overshadowed by other functions or system-level noises and thus it is hard to be identified by previous work [35].

- DMTracker incurs low overhead due to our system design and usage of a low-overhead dynamic instrumentation tool called Pin [150]. Our experimental results show that the runtime overhead of DMTracker is only 0.9-6.0%. Therefore, it is possible to directly apply DMTracker to production runs.

4.2 Background: Statistical Rule Based Bug Detection

Statistics-rule-based bug detection methods extract statistic rules, (i.e., dynamic invariants), at program runtime and check violations of the extracted invariants, which indicate potential software bugs. These approaches are motivated from an observation that the hard-to-detect bugs are usually those lurking in a corner case that rarely happens. Because the
program behaves correctly for most of the cases, the bugs in corner cases are more difficult to be manifested by testing and thus more likely to cause problems in production runs. Due to their hidden nature, these bugs take much more time to detect and fix.

In recent years, research efforts have been made toward this direction [28,33,34,44]. They have introduced several types of dynamic invariants to detect bugs that manifest themselves in different ways. Daikon [33] and DIDUCE [34] focus on the value ranges of variables and use them as invariants to detect the abnormal values of variables. AccMon [28] focuses on the program counter (PC)-based invariants to detect the abnormal instructions accessing to a certain memory location. AVIO [44] makes use of Access Interleaving (AI) invariants to detect the violation of atomicity execution in multi-thread programs. These works focus on sequential programs, and thus their proposed invariants are mainly to capture the temporal similarity of the programs behavior. Recently in [35], an anomaly-detection method was proposed to use function time based invariants for diagnosing problems in large distributed computing environments. This work focuses on identifying the abnormal process in a parallel application, by capturing the spatial similarity of parallel programs.

In our work, we focus on one of the key aspects of parallel programs, the data movements, and propose two data movement-based invariants to capture both temporal similarity and spatial similarity in parallel programs.

4.3 Data Movement-Based Invariants

Data movement is the movement of a chunk of memory data from a source buffer to a destination buffer. For example, copying a chunk of memory data from buffer $A$ to buffer $B$ corresponds to one data movement $A \rightarrow B$. Typically, we can capture data movements in parallel programs on different levels: on application level by regarding each communication calls such as MPI library calls as one data movement, or on library level by analyzing each
primitive operation such as memory copy, network send/receive, etc., as one data movement. For bug detection purposes, we choose the library level because it provides comprehensive information for finding bugs in applications as well as communication libraries and it is decoupled from particular programming models and communication interfaces.

Individual data movement reveals little information about program semantics. Thus it is difficult to extract invariants from them. To address this issue, we link a series of data movements to form a data movement (DM)-chain in a way that the destination of previous data movement is the source of the subsequent data movement. Figure 1 illustrates a simple DM-chain between two processes. The whole chain in the figure can be caused by a pair of communication calls such as MPI_Send and MPI_Recv on the application level. DM-chains are the basis of our proposed two types of invariants, which are described in the following two sections.

![Figure 4.1: A simple DM-chain: A1→B1→B2→A2](image)

### 4.3.1 Frequent Chain Invariants

FC-invariants are the frequently-occurring DM-chains. This is based on the observation that processes in parallel programs often exhibit temporal similarity, e.g., performing similar or identical tasks in multiple iterations. As a result, similar DM-chains occur many times during program execution. Based on this observation, we can group similar DM-chains together according to their type information, e.g., call sites of data movements and memory
buffers. Then we can use large-sized groups, i.e., frequently-happening DM-chains, as FC-invariants.

Based on the FC-invariants, it is possible to detect abnormal data movements, i.e., similar to a FC-invariant but with slight difference. These abnormal data movements are potentially caused by software bugs and deserve programmers’ attention. Typically, abnormal data movements are caused by buffer misuse and can lead to data corruptions as well as other errors such as crash or deadlock.

Figure 2(a) shows a simplified bug case extracted from a communication library, where data corruption is caused by pointer misuse. While the code path of common case goes through line 11, the bug is at line 7, in the code path to deal with a rare case, where some data need to be piggybacked to the packet to notify its peer about some event, e.g. out of resource. Since the programmer tends to think more of bytes rather than data objects when programming network protocols, \texttt{sizeof} is used to calculate the offset. Unfortunately, without parentheses, the type cast happens in a higher precedence and the address is subtracted in unit of \texttt{sizeof(Piggyback.t)}, so the actual address change is \texttt{sizeof(Piggyback.t)} \times \texttt{sizeof(Piggyback.t)}, which causes the pointer to point to another buffer. This bug can easily slip through normal software tests since it deals with uncommon cases only when some rare event occurs. Furthermore, it is difficult to detect this bug during production runs since it may manifest itself as silently sending incorrect data to the remote receiver.

Figure 2(b) shows data movements related to the \texttt{send()} routine, including both common cases and uncommon cases. It clearly tells that the bug manifests itself in the abnormal data movement. During normal program execution, DM-chains related to the \texttt{send()} routine are $X \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow Y$. When the bug occurs, the DM-chain changes to $A' \rightarrow B \rightarrow C \rightarrow D \rightarrow Y$, similar to the FC-invariant for the common case but with the link from $A \rightarrow B$ broken. Obviously, it violates the FC-invariant.
FC-invariants can be extracted from one run or based on multiple runs. DMTracker does not require reference data to distinguish abnormal chains from normal. But in some cases, the incorrect DM-chains can be the “common cases” in one specific run when the bug happens. To be more effective in detecting bugs in these cases, DMTracker also allows users to provide a “training set”, a number of traces known to reflect correct behaviors, so that it can extract patterns only from those traces. Then DMTracker can be more effective in locating incorrect behaviors in traces from problematic runs.

### 4.3.2 Chain Distribution Invariants

CD-invariants are the clusters of chain distributions that the chain distribution of each process should fit in. It is based on the observation that processes in parallel programs often exhibit spatial similarity, e.g., performing similar or identical tasks and following the same symmetric communication patterns in multiple processes. As a result, the distribution of various groups of DM-chains (grouping DM-chains using the same method as we did for FC-invariants) are similar across multiple processes. Therefore, we can use chain distributions clusters as CD-invariants.

Figure 3 demonstrates the chain distributions in all processes for High Performance Linpack (HPL) benchmark [7] and SP, MG in NAS Parallel Benchmarks (NPB) [8]. (Most
benchmarks in NPB show similar trends, we omit them due to lack of space). The x-axis indicates the process ID of each process in the parallel programs and each column of a graph shows the chain distribution in that process. It is clear that during normal execution, all 64 processes for the tested benchmarks share very similar chain distributions.

Based on CD-invariants, it is possible to capture bugs that happen in a small number of processes. DMTracker compares chain distributions across all processes and automatically locates the manifesting process in a large number of peers, so that the search space for the bug can be greatly narrowed down. Typically, an abnormal chain distribution is caused by some algorithm or protocol error in parallel programs, which can manifest itself as infinite loop (i.e., livelock), deadlock, etc.

In addition to the chain distributions of the whole run, comparing the chain distributions within a certain time period can be more effective for detecting bugs in some cases. For example, to diagnose deadlock bug, chain distributions of all processes in the last phase of execution are especially useful. Since in a deadlock situation, processes usually stop to proceed their executions and also stop communicating, the distorted chain distribution only shows in the last phase and can potentially be overshadowed by the chain distribution in earlier phases.

In cases where processes in a parallel application are designed to perform different tasks (e.g. in master-worker model), the chain distributions for some processes (e.g. rank 0) will
be very different from others even in correct runs. In these cases, CD-invariants are only valid for groups of processes which perform similar or identical tasks. DMTracker can be configured to only analyze chain distributions of a specific subset of processes rather than all processes in the whole parallel program.

4.4 Design of DMTracker

DMTracker consists of two major components, the online tracking component and the offline analysis component, as shown in Figure 4.4. The online tracking component (Section 4.4.1) collects data movement (DM)-traces at runtime by leveraging lightweight binary instrumentation. Based on the collected DM-traces, the offline analysis component forms DM-chains (Section 4.4.2), extracts FC-invariants and CD-invariants respectively (Section 4.4.3) and detects anomalies that violate the extracted invariants (Sections 4.4.4 and 4.4.5).

Figure 4.4: DMTracker Design Overview
4.4.1 Lightweight Data Movement Tracking

The online data movement tracking component records data movement related information into traces, called DM-trace, by instrumenting binary code of the parallel programs. A dynamic instrumentation tool called Pin [150] is used in our current implementation. Unlike traditional bug detection tools such as Purify [21] that track every memory accesses, DMTracker only instruments function calls related to memory management (e.g. allocation/deallocation) and data movements (e.g., memory copy and network operations). Therefore, it incurs very low overhead. Furthermore, by directly instrumenting binary code, DMTracker requires no recompilation of the source code, which can avoid some inconvenience for usage.

To capture data movement semantics, DMTracker records the following information of instrumented function calls: (1) key arguments and return values, (2) call sites that contain stack context when the call is made, (3) thread IDs for multi-threaded processes, and (4) local timestamps when the call is made. More specifically, for memory allocation calls such as malloc, calloc, etc., DMTracker records both request size and the memory object address; for memory copy calls, it records source and destination addresses and the copy length; for network operations, it records the buffer, length, and the information to identify the network endpoint. The call sites are useful in analyzing traces and providing more diagnosis information to programmers. The timestamp is used to order local operations. Note that for the programs using customized memory management module, additional instrumentation needs to be done regarding the customized interfaces.

DM-trace grows moderately since DMTracker only records data movement related functions. Its growth rate largely depends on the communication patterns of parallel applications. In our experiments, typically 20MB of disk space can store traces for several minutes for parallel benchmarks such as HPL and NPB, and thus it is usually not a big problem for
storing the whole traces into local disks. In addition, we can leverage existing techniques such as trace compression [151] and streaming processing [152] to reduce storage overhead if it becomes a big concern.

4.4.2 Preprocessing: DM-Chain Formation

Based on the collected DM-traces, DMTracker forms DM-chains by parsing data movement operations, DM-operations, and linking related data movements. For scalability, DMTracker processes individual traces and forms DM-chains in parallel instead of processing all the traces in a central node.

Parsing DM-Operations. DMTracker parses information of each function call recorded in the DM-traces and correlates each DM-operation to its source and destination buffers’ allocation information. For example, a memory copy $A \rightarrow B$ correlates to the allocation information of $A$ (e.g. malloc at $PC_a$) and $B$ (e.g. memalign at $PC_b$). This correlation information provides contexts for linking data movements and grouping chains, which are discussed in detail later.

In parallel programs, some data movements such as data pack/unpack may contain multiple memory copies. In many cases, data in multiple small non-contiguous buffers are packed to a larger contiguous buffer, or data in a large buffer are unpacked to multiple smaller non-contiguous buffers. Since a memory copy usually requires source and destination to be contiguous, these operations require multiple memory copies. DMTracker aggregates multiple memory copies for a pack or unpack operation and parses them as a single DM-operation to better reflect the semantics of data movements.

For network operations, in this step, DMTracker can only correlate partial allocation information, either the source buffer (for send) or the destination buffer (for receive), to the
data movement within each process. In order to link data movements across the network, DMTracker keeps the connection end-point information temporarily.

**Movement Concatenation.** Based on the parsed DM-operations, DMTracker forms chains by linking related data movements where the destination buffer of a previous DM-operation is the source buffer of a subsequent DM-operation. To form a complete chain, DMTracker performs two steps, intra-process linking and inter-process linking. The intra-process linking is to link DM-operations within one process. For example, it links memory copies with related memory copies or network operations. To efficiently match the source buffer of a DM-operation with the destination buffer of some previous DM-operation, DMTracker maintains an *active chain table* for existing DM-chains that can potentially have successive DM-operations. If matching happens, DMTracker extends the matched DM-chain and updates the table correspondingly. Otherwise, it inserts the DM-operation as a new DM-chain into the table.

The inter-process linking is to link the chains across different processes together by matching the send operations and receive operations. To achieve this, DMTracker maintains the information of network connections (e.g., file descriptors for sockets, queue pairs (QPs) for InfiniBand [153]). For the FIFO communication channel (e.g., TCP and InfiniBand RC), send and receive can be matched simply by the order. For the channels which do not guarantee FIFO, further instrumentation needs to be done to track sequence numbers with the packets. When a send matches with a receive, the chain starting with the receive is linked to the end of the chain ending with the send. This way the DM-chain that reflects the whole process of data movement across multiple processes can be formed.
4.4.3 Invariants Generation

To generate FC-invariants and CD-invariants, DMTracker first groups the same type of chains together based on the information of data movements in each chain since chain groups provide better data movement semantics. More precisely, DMTracker regards two chains as *the same type* and puts them into *the same group* if the corresponding individual data movements within the two chains have the same call sites for the DM-operations and the same allocation call sites for their source and destination buffers. For example, the chain $A_1 \rightarrow A_2 \rightarrow A_3$ and the chain $B_1 \rightarrow B_2 \rightarrow B_3$ are of the same type and belong to the same chain group if the data movement $A_i \rightarrow A_{i+1}$ has the same call site as $B_i \rightarrow B_{i+1}$ and $A_j$ has the same allocation call site as $B_j$, where $i = 1, 2$ and $j = 1, 2, 3$. We use call site information for grouping chains due to two reasons: (1) data movements at different program locations usually handle different cases, capturing program semantics in communication, and (2) the same memory allocation site usually allocates the same type of data, capturing characteristics of memory buffers.

**Generating FC-invariants: Frequent Chain Pattern Selection**

DMTracker extracts FC-invariants from the above-formed chain groups based on two criteria. Obviously, the chains of that group must happen frequently, i.e., the number of chains in the chain group should be relatively large. One naive way is to set an absolute number as the threshold and require all the FC-invariants with chain number larger than that threshold. However, it is difficult to do so in practice since the threshold is highly dependent on the number of chains in the trace. Therefore, DMTracker uses a relative number, the percentage of total number of chains, as the threshold.

Another criteria is that, the chain type for each chain group must preserve unique characteristics so that we can easily determine whether another chain matches the FC-invariant.
or not, and how much it matches. If a chain type contains only one data movement, there is no point to select it as invariant and compare other chains with it because other chains are either 0% match or 100% match, therefore they can not be the violations we are interested in, which are mostly similar to but have some difference from the invariant chain type. The uniqueness of a chain type is determined by the length of the chain as well as the uniqueness of each individual data movement that forms the chain. Therefore, we define the uniqueness value of a chain type $C$, $Uniqueness(C)$, as the sum of the uniqueness values of its DM-operations ($O_1, O_2, ...$):

$$Uniqueness(C) = \sum Uniqueness(O_i)$$ (4.1)

Uniqueness values of different types of data movements should be assigned in different ways since they reflect different semantics. We normalize the single memory copy to 1, and define the uniqueness of a DM-operation $O_i$, $Uniqueness(O_i)$, as follows

$$Uniqueness(O_i) = \begin{cases} 
1, & \text{if } O_i \text{ is a single memory copy;} \\
M, & \text{if } O_i \text{ is a network operation;} \\
\min(N, n), & \text{if } O_i \text{ is a data pack/unpack.}
\end{cases}$$ (4.2)

In Equation 4.2, $n$ represents the number of segments of data to pack/unpack, and $M$ and $N$ are tunable parameters. Since network operation involves two processes, we use $M = 2$ in our experiments. Uniqueness of data pack/unpack is designed that way because: a) it should reflect the number of segments it has, and b) it should not overwhelm other movements. We use $N = 10$ in our experiments.

The thresholds to select FC-invariants should partially depend on the target program. In our experiments, we set the uniqueness threshold to 5, and require the number of chains in a chain group for FC-invariants to account for more than 10% of the total number of chains in the chain groups whose uniqueness value is above the threshold.
DMTracker extracts CD-invariants, i.e., clusters of chain distributions that the chain distribution of individual processes should fit in, based on the chain distributions. We define the chain distribution in a trace $T$ for a process as a vector, $CD(T)$. Each element in the vector represents the percentage of all the chains one chain group accounts for.

$$CD(T) = \left\langle \frac{\text{Count}(C_1, T)}{\text{Count}(T)}, \frac{\text{Count}(C_2, T)}{\text{Count}(T)}, ..., \frac{\text{Count}(C_m, T)}{\text{Count}(T)} \right\rangle$$ (4.3)

where $m$ is the total number of distinct chain groups for all the processes, $\text{Count}(C_i, T)$ represents the number of chains in the $i^{th}$ chain group $C_i$ in the trace $T$, and $\text{Count}(T)$ represents total number of chains occurred in $T$. Even though in many cases a chain goes across multiple processes, to analyze the distribution, we consider a chain to belong to a certain process when the chain starts from that process. In this way, we do not calculate a chain multiple times and still have a symmetric measurement.

To cluster chain distributions, DMTracker uses Manhattan distance [154] to the $k^{th}$ nearest neighbor [155] as our metric, which is also used in previous work [35]. The Manhattan distance between two traces $T_i$ and $T_j$, $\text{Distance}(T_i, T_j)$, is the sum of the absolute differences of each element in the chain distribution vector as defined in Equation 4.4. $\text{Distance}^k$ is the distance to $k^{th}$ nearest neighbor (Equation 4.5), which reflects how well a chain distribution fits into a cluster with multiple peers. A lower value of $\text{Distance}^k$ means the chain distribution fits better into a cluster. Thus, DMTracker uses it to measure how similar a chain distribution in one process is compared to its peers’.

DMTracker uses a certain range of $\text{Distance}^k(T_i)$ as CD-invariants. Similar to [35], in our experiments, the values of parameters $k$ is set to $n/4$, where $n$ is total number of processes.

$$\text{Distance}(T_i, T_j) = \left| \frac{\text{Count}(C_1, T_i)}{\text{Count}(T_i)} - \frac{\text{Count}(C_1, T_j)}{\text{Count}(T_j)} \right| + ... + \left| \frac{\text{Count}(C_m, T_i)}{\text{Count}(T_i)} - \frac{\text{Count}(C_m, T_j)}{\text{Count}(T_j)} \right|$$

$$\text{Distance}^k(T_i) = \text{Distance}(T_i, T_{j_l}) \text{ where Distance}(T_i, T_{j_l}) < \text{Distance}(T_i, T_{j_{l+1}}), i \neq j_l, 1 \leq l \leq n - 1$$ (4.5)
4.4.4 FC-invariants Based Anomaly Detection

Based on the extracted FC-invariants, DMTracker can detect abnormal chain groups potentially caused by software bugs, and validate the reported bug by checking each chain instance in the abnormal groups with its context.

Abnormal Chain Group Detection

DMTracker detects abnormal chain groups by comparing each chain group with the extracted FC-invariants. It considers a chain group $C$ being an abnormal case of a FC-invariant $P$ if they are similar enough and $C$ is relatively rare compared to $P$. On one hand, $C$ needs to be similar to $P$ so that we can determine it as an abnormal case of $P$. The more similar they are, the more likely that $C$ is an abnormal case of $P$, except $C$ is identical to $P$. For example, if $C$ matches 90% with $P$, then we can consider the rest 10% unmatched part to be abnormal. But if $C$ only matches 10% with $P$, it is likely to be just an infrequent type of chains instead of being an abnormal case of $P$.

To measure the similarity between a chain group $C$ and a FC-invariant $P$, DMTracker uses a metrics derived from Jaccard Coefficient [156], a widely used metric to measure similarity, to capture the matched part and unmatched part between $C$ and $P$. Equation 4.6 shows the formal definition of similarity. To find the largest matching part $(C \cap P)$, we symbolize the DM operations in the chain and convert the problem to a longest common substring problem. The longest common substring problem can be solved in $O(m + n)$ time with the help of a generalized suffix tree [157], where $m$ and $n$ are the lengths of the two strings.

$$
\text{Similarity}(C, P) = \frac{\text{Uniqueness}(C \cap P)}{\text{Uniqueness}(C \cup P)} = \frac{\text{Uniqueness}(C \cap P)}{\text{Uniqueness}(C) + \text{Uniqueness}(P) - \text{Uniqueness}(C \cap P)}
$$

(4.6)
On the other hand, $C$ needs to be relatively rare compared to $P$. Otherwise, if $C$ is almost as frequent as, or even more frequent than $P$, it is very likely to be another pattern instead of an abnormal case of $P$. To measure the rareness, DMTracker compares the frequency of $C$, $Frequency(C)$, with the frequency of $P$, $Frequency(P)$. There is no point to compare the $C$ with $P$ if $Frequency(C)$ is larger than $Frequency(P)$. We define the rareness, $Rareness(C, P)$, as below:

$$Rareness(C, P) = \frac{Frequency(P) - Frequency(C)}{Frequency(P)}$$

where $Frequency(P) > Frequency(C)$

(4.7)

We are looking for abnormal chain groups with high similarity, meaning the chain group is very similar to the FC-invariants, and with high rareness, meaning the chain group is relatively rare compared to the FC-invariants. We choose to use harmonic mean to combine similarity and rareness because it is a commonly used way which prefers high scores in both dimensions. Thus the overall metric of abnormality can be defined as follows:

$$Abnormality(C, P) = \frac{2}{\frac{1}{Similarity(C, P)} + \frac{1}{Rareness(C, P)}}$$

(4.8)

For each pattern $P_i$, DMTracker detects a list of chain groups, noted as $C_{i_1}, C_{i_2}, ...$, with $Abnormality(C_{i_k}, P_i) > Threshold_{Abnormality}$, $C_{i_k} \neq P_i$, and $Frequency(C_{i_k}) < Frequency(P_i)$. We call each pair, $(C_{i_k}, P_i)$, a violation. DMTracker then combines lists of violations for all patterns, and ranks them according to the abnormality score. In our experiments, DMTracker reports all violations with an abnormality score more than 0.7

**Chain Instance Checking**

Since not all DM-chains in the abnormal chain groups are necessarily caused by bugs, DMTracker needs to check each chain instance in the context of chain trace for validation and providing more detailed diagnosis information to the programmer.
In this step, DMTracker goes through each chain instance in the abnormal chain groups, and examines the DM-operations that happened immediately before/after the chain instance to see whether they match the previously unmatched part between the abnormal chain and its matching FC-invariant. For instance, assume that a chain group $C$ with three DM-operations $(X, Y, Z)$ matches third to fifth DM-operations in a FC-invariant $P (U, V, X, Y, Z)$ and $c$ is a chain instance of $C$. If some DM-operations closely before $c$ in the chain trace match the DM-operations $U$ and/or $V$ in $P$, it strongly suggests a broken chain. DMTracker will highlight it by marking as “context match.”

Furthermore, DMTracker can provide the detailed information with each abnormal instance into report, and rank the anomalies with “context match” and high abnormality score on the top. The reported information includes buffer address, buffer allocation call sites, data movement call sites, etc., of both the abnormal chains and their contexts (i.e. data movements chains happened before/after it). It is likely to be helpful for diagnosing the problem.

### 4.4.5 CD-invariants Based Anomaly Detection

Based on the CD-invariants, reflected as a certain range of $Distance^k$ scores, it is relatively straightforward to detect the outliers. Different from [155], where the number of outliers is given by users, here DMTracker uses another criterion to find all traces that are not similar enough to their peers. If a trace has a significantly larger value of $Distance^k$ than the average, e.g. $K$ times larger, DMTracker will report it as an abnormal trace. In our experiments, we use 5 as the value of parameter $K$ for fewer false positives. The chain group that contributes the most to the $Distance(T_i, T_{jk})$ in Equation 4.4 is also included as diagnosis information in bug reports.
In some cases, the most interesting part of the chain distribution is about the last phase of execution. For instance, if a parallel program deadlocks and we want to find out which process causes this problem, the chains that happen much earlier in the run may dilute the difference. To deal with this case, we can use a binary search method. When no outliers have been found but users want to do in-depth diagnosis, DMTracker can perform multiple rounds of CD-invariant based detection with the chains in the halved time range.

4.4.6 Issues and Discussions

Additional Inter-process Communication Channels. In communication libraries, in addition to common network send and receive, there are other approaches to perform inter-process communication, such as through a shared memory region on the same host, or using Remote Direct Memory Access (RDMA) to directly access memory on a remote host. For data movements through shared memory, DMTracker tracks them as data movements through non-shared memory. However, our current prototype has not yet supported the shared memory region construction such as memory mapping to a common file. Therefore, DMTracker does not link the chains crossing processes if they are communicated through shared memory regions. This may affect the accuracy in some cases, but is not a major problem in our experiments because the separate chains are still analyzed in each process. We plan to track the construction of shared memory regions in our future work.

RDMA, an advanced feature provided by modern networks such as InfiniBand [153] and Quadrics [158], etc., allows one-sided communication. Our current prototype does not link chains through RDMA channels because only the process on the active side has a DM-operation. To address this issue, we need to modify device drivers or firmware of network interface cards to expose RDMA operation to the user-level process on the passive side.
Online Analysis and On-the-fly Detection. With more computing power provided by multi-core systems, DMTracker could process and analyze traces using dedicated cores in each node. Since the processing cores can directly access the traces in memory instead of via expensive file I/Os, it can achieve high performance. In addition, the storage overhead can be alleviated because much smaller intermediate results are needed for further analysis after preliminary local processing. These will enable us to extend DMTracker for performing on-the-fly detection in future work.

4.5 Evaluation and Case Studies

The experiments described in this section were conducted on a 64-processor cluster with 32 nodes. On each node, there are two 3.6GHz, 2MB L2Cache CPUs and 2GB memory. These nodes are connected using InfiniBand PCI-Ex DDR adapters with 10Gbps peak unidirectional bandwidth. The Operating System is Linux with kernel version 2.6.17.7.

To evaluate DMTracker’s functionality, we use MVAPICH/MVAPICH2 [6], a popular high performance open-source MPI library over InfiniBand, with two real-world bug cases: one data corruption bug causing incorrect results and one deadlock bug causing program to hang. Both MVAPICH and MVAPICH2 packages are large: each has more than 350,000 lines of C code in more than 1,500 source files.
To evaluate runtime overhead incurred by the online tracking component of DMTracker, we compare the performance difference of High Performance Linpack (HPL) benchmark and NAS Parallel Benchmarks (NPB) with and without DMTracker.

4.5.1 Case 1: Data Corruption in Communication

The data corruption bug in MVAPICH2 version 0.9.8 was triggered deterministically by executing a communication library for linear algebra, called BLACS (Basic Linear Algebra Communication Subprograms) [159]. The test program for the BLACS package, called xCbtst, reports “Invalid element” error in all the 64 processes after executing BSBR (broadcast/send and broadcast/recv) test cases. The data corruption bug happens silently (i.e., no hang or system failures) and is shown at the last stage of result verification.

After being applied for this scenario, DMTracker reports six abnormal chain groups that violate two of five extracted FC-invariants. Out of the reported six abnormal cases, the top two ranked anomalies indicate the real bug. In both cases, a FC-invariant (15075 times) is violated by rarely-occurred similar chains (154 times). Figure 4.5 shows the frequently-happening chain (FC-invariant) on the left side and the rare cases (a broken chain caused by the data corruption bug) on the right side. The instance context checking confirms that all the 154 chain instances are caused by the data corruption bug.

DMTracker not only detects this bug, but also provides useful diagnostic information to programmers for quickly locating this bug. With detailed information about the abnormal chain groups, the root cause of the problem, an optimization called header caching for RDMA operations fails to handle a corner case in communication protocols, can be easily identified. This demonstrates that statistical-rule-based tools like DMTracker can be helpful in detecting rarely-happening bugs and locating the root causes, which otherwise would require much more human effort.
Furthermore, this case study shows that DMTracker can also detect software bugs that manifest themselves across all the processes in a similar way. This is because DMTracker exploits temporal similarity within each process as well as spatial similarity across different processes. This data corruption bug has been triggered in all the 64 processes in a similar way, indicating that it is extremely hard, if not impossible, for previous work [35] to detect and diagnose it by only exploring spatial similarity among different processes.

4.5.2 Case 2: Deadlock in Connection Setup

The deadlock bug was triggered by running FT benchmark when testing an internal version of MVAPICH on 64 processes. The program hang non-deterministically with a very small chance during execution.

After being applied, DMTracker detects an anomaly that violates the extracted CD-based invariant. As shown in Figure 4.6, process 43 has a very high $\text{Distance}^k$ score, 1.61, which is significantly higher than the average score 0.23. It strongly indicates that the chain distribution in process 43 is an outlier of any cluster of chain distributions. DMTracker detects this non-deterministic bug via exploiting the spatial similarity across different processes.

DMTracker not only detects this bug, but also reports useful diagnostic information to programmers for identifying the root cause. It reports that the chain group with a network send operation in function cm_post_ud_packet() of the file cm.c is the major contributor to the high $\text{Distance}^k$ score. This information quickly narrows down the root cause to a specific function. In that function, accessing of a variable, which is forgotten to be defined as volatile, causes an intended benign data race to be harmful by a small chance.

Although the time spent on the function cm_post_ud_packet() in bug cases is much longer than that in normal cases, it can not be easily detected by the function time distribution method [35] due to noises caused by other functions. To achieve better performance, most
communication libraries built on current main-stream high performance networks including MVAPICH/MVAPICH2 use a polling-based progressing mechanism. In this mechanism, when waiting for an event, the process will be busy waiting in multiple polling functions. The difference of time spent in abnormal functions is easily overshadowed by the time spent in these polling functions. In our experiments, we measured that the total time spent in the abnormal function in the outlier process is less than 0.05% of the total time in progressing functions, which is not reflected in visible difference of function time distribution. To filter the effect of polling functions, non-trivial work is required with respect to each library to determine which functions should count and which should not.

4.5.3 Runtime Overhead

In this set of experiments, we evaluated the performance impact of DMTracker’s online tracking component with HPL benchmark with different problem sizes and NPB benchmark with class C. We ran these benchmarks based on MVAPICH2 version 0.9.8 natively and with tracking using 32 nodes with 2 processes per node (32x2).

Table 4.5.3 lists the relative performance degradation for HPL when applying DMTracker as compared to native. In general, the runtime overhead is very low, from 2.3% to 3.1%. The low overhead is because DMTracker only tracks a small set of functions related to data movements. In addition, we observe that the overhead decreases as the problem size of HPL benchmark increases. That is because when the problem size becomes larger, the data movements among processes tends to be in larger chunks, resulting in less frequent data movements and thus lower overhead. Therefore, for long-running applications operating on large data sets, the overhead incurred by DMTracker is expected to be low.

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>40000</th>
<th>50000</th>
<th>60000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative Degradation</td>
<td>3.1%</td>
<td>2.7%</td>
<td>2.3%</td>
</tr>
</tbody>
</table>
We also observe that applications with different communication characteristics show different runtime overheads when being tracked by DMTracker. As shown in Table 4.2, the overhead varies largely for different NPB benchmarks. The ones with very frequent communications, like CG, IS, and LU, show slightly higher overhead (3.9%-6.0%); while others, BT, FT, MG, and SP, only show almost negligible overhead (0.9%-1.6%). These results are expected since the overhead is caused by tracking data communications in applications. As demonstrated by both HPL and NPB, DMTracker incurs very low runtime overhead, which indicates that DMTracker can be deployed in production runs.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>BT</th>
<th>CG</th>
<th>FT</th>
<th>IS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative Degradation</td>
<td>1.0%</td>
<td>3.9%</td>
<td>0.9%</td>
<td>4.2%</td>
</tr>
<tr>
<td>Benchmarks</td>
<td>LU</td>
<td>MG</td>
<td>SP</td>
<td></td>
</tr>
<tr>
<td>Relative Degradation</td>
<td>6.0%</td>
<td>1.4%</td>
<td>1.6%</td>
<td></td>
</tr>
</tbody>
</table>

Currently, the trace processing and analysis components of DMTracker are still performed offline. Therefore, this section only discusses the runtime overhead incurred by the online tracking component. In the future, we plan to extend DMTracker to detect software bugs on-the-fly by leveraging stream processing algorithms and multi-core architectures.

### 4.5.4 Sensitivity Study and False Positives

We have also conducted a set of experiments to study the parameter sensitivity of DMTracker. As in most static-rule-based tools, parameters and threshold values can affect the balance between effectiveness and false positives of DMTracker. With lower thresholds, DMTracker can report more potential bugs (also more false positives), while using higher thresholds can reduce the number of false positives with the risk of missing the real bug (false negative).
For FC-invariants, we tried several options to lower the thresholds of parameters in case 1: a) uniqueness requirements for a chain group to be FC-invariant, from 5 to 3 (the minimum meaningful value); b) frequency thresholds for a chain group to be FC-invariant, from 10% to 5% (a very low value); and c) abnormality threshold for violation report, from 0.7 to 0.5. Experimental results were sensitive to these parameters to some extent: up to 3 more statistical invariants and up to 15 more violations were reported, but the real bug cases still ranked at the top and highlighted by the context match. Most false positives for FC-invariants we have encountered during the experiments with case 1, NPB, and HPL, can be summarized as follows.

**Infrequently used communication protocol.** Since the flow control algorithm in the MPI library may choose different protocols according to network resource usage, in some cases a fall-back protocol may be used. Then the data movements in the infrequently used protocol will cause a very small group of chains to be very similar but different from FC-invariants, which will be reported as violation by DMTracker. This type contributes more than half of the false positives. To prune them, we can use more sophisticated checking steps by incorporating semantic information.

**Buffer reuse for control message.** In the MPI library, control messages are passed using the same protocol as small data messages, but the communication buffers for control messages are immediately reused instead of being copied to application buffers. Thus the data movements for control messages may result in a slightly different (usually longer) chain than FC-invariant, which will be considered as a violation.

For CD-invariants, we tried different values of parameter $k$, such as $n/2$, $n/4$, $n/8$, where $n$ is the number of processes. The result of case 2 is not affected by different values. When applying to HPL and NPB, a few false alarms were given for LU and CG indicating that
they have abnormal processes with different data movement statistics. These false alarms can be pruned by checking the data movement statistics about previous known good runs.

Note that to correct suboptimal parameter settings, users only need to rerun the analysis component offline, which is generally much cheaper than rerunning the large-scale parallel program again.

4.6 Summary

In this section, we have discussed an innovative statistical-rule-based approach to automatically find the hard-to-detect bugs that can cause severe problems such as data corruptions and deadlocks in large-scale parallel programs. Our approach extracts the data movement based program invariants at runtime, and detects anomaly based on the extracted invariants. Based on this idea, we have built DMTracker to help programmers locate root causes of software bugs. Our evaluation with two real-world bug cases in MVAPICH/MVAPICH2 shows that DMTracker is effective in detecting them and providing useful diagnosis information. In addition, DMTracker only incurs a very low runtime overhead (0.9%-6.0%) measured by HPL benchmark and NAS Parallel Benchmarks, so that it is possible to be deployed in production runs.
CHAPTER 5

FIRST-AID

This chapter talks about failure recovery and future error prevention for common memory bugs in system software during production runs. Given that memory bugs are a major category of common software defects in programs written in C/C++ languages, it is quite common for a system software program, which are mainly written in these languages for easier interfacing with OS kernels and hardware devices, to encounter a failure due to a memory bug in production runs. When a failure happens, a fully automated runtime diagnosis and patching system is very preferable to the availability because it can completely get human involvement out of the critical path of dealing with the manifested bugs. For that purpose, we propose such a light-weight tool called First-Aid, which allow system software to survive failures caused by common memory management bugs and prevents future errors caused by the same bugs during production runs.

This chapter is organized as follows: In Section 5.1, we give an overview of the motivation and our approach. In Section 5.2, we describe the working scenario of our tool, First-Aid. In Section 5.3, we present the design of First-Aid. Then we describe the diagnosis procedure and validation procedure in Section 5.4 and Section 5.5, respectively, followed by a brief discussion in Section 5.6. In Section 5.7, we discuss the evaluation results. And finally we provide a brief summary in Section 5.8.
5.1 Overview

Memory management bugs, such as buffer overflows and dangling pointers, are a major category of common software defects and severely affect system availability and security. During production runs, these types of bugs can corrupt memory data, leading to program crashes or hangs. Furthermore, malicious users often launch security attacks by exploiting these bugs. According to the US-CERT Vulnerability Notes Database [160], memory management bugs dominate recent security vulnerability reports.

Furthermore, it is a challenging task for developers to diagnose these bugs and release timely fixes because of ever-increasing software complexity and lack of on-site failure information [109]. Previous studies [161, 162] have shown that it takes several weeks on average to diagnose bugs and generate patches. During this long time window, users have to either continue running the software with bugs and tolerate problems such as intermittent crashes and potential attacks, or stop running the software and experience costly system downtime. Neither option is desirable.

Therefore, it is critical to be able to quickly recover programs from software failures caused by memory bugs, protect programs from future failures due to the same bugs, and provide useful on-site failure information for developers to fix the bugs.

While several recent proposals help programs survive failures caused by memory bugs, they suffer one or more of the following limitations: unsafe speculation on programmers’ intentions [87, 163], inability to prevent subsequent failures due to the same bugs [88, 91], little diagnostic information to developers [90], and large time and space overheads [88, 89]. For example, failure oblivious computing [87] discards out-of-bound writes and manufactures arbitrary values for out-of-bound reads. While this approach may survive failures for certain types of applications, its speculation on programmers’ intention could easily lead to program misbehavior. DieHard [88] and Exterminator [89] probabilistically prevent failures caused
by memory bugs via a randomized memory runtime system. However, large time and space overheads prevent them from being adopted for production runs.

Our previous work Rx [91] can quickly recover programs from failures by re-executing a program from previous checkpoints and applying environmental changes to program execution. An example of such an environmental change is adding padding to all memory objects during recovery to avoid buffer overflow bugs. While effectively and safely avoiding the memory bugs, Rx has to disable the environmental changes after surviving a failure due to their potentially large overhead. Therefore, Rx cannot prevent future failures caused by the same bug. Furthermore, the on-site failure recovery information provided by Rx (i.e., whether and what environmental changes can work) is quite limited and may mislead developers because failure-surviving environmental changes may not directly relate to the bugs that have occurred.

This work makes three contributions in order to address the limitations of previous work:

**Contribution 1: A low-overhead method for surviving and preventing memory management bugs.** We propose a system, called First-Aid, that can quickly recover programs from failures caused by memory bugs, prevent future failures due to the same bugs, and provide useful on-site diagnostic information to developers. The main idea is to diagnose the bug type and identify the memory objects that trigger the bug when a failure occurs. Based on results of the diagnosis, First-Aid generates and applies runtime patches (environmental changes for bug-triggering memory objects) to a small set of memory objects that can potentially trigger the bug, during both program recovery and future program execution. As a result, First-Aid not only helps programs survive the current failure but also prevents future failures caused by the same bug. Furthermore, First-Aid validates the runtime patches to make sure their effects are consistent and generates a bug report to help developers fix the bug.
**Contribution 2: Environmental change based failure diagnosis.** We propose an online failure diagnosis method that leverages efficient checkpointing and re-execution mechanisms as well as execution environment changes. Specifically, when a bug causes a failure or an error at runtime, First-Aid rolls back the program to previous checkpoints and re-executes the program. During each re-execution, First-Aid dynamically applies two types of environmental changes: *exposing changes*, which force a certain type of bug to manifest itself, and *preventive changes*, which prevent a certain type of bug from manifesting itself. Based on bug manifestation after re-execution, First-Aid can conclude whether a certain type of bug is occurring. For example, to determine whether a buffer overflow bug is occurring, First-Aid applies the exposing change for buffer overflow, i.e., padding each memory object and filling the padding with canary values, and the preventive changes for all other memory bug types. The term *canary* refers to certain memory content patterns that are unlikely to appear during normal program execution. If the canary values in some padding are corrupted, First-Aid knows that the bug is likely to be a buffer overflow. Exposing changes also help First-Aid identify the bug-triggering memory objects. In the same example, based on the corrupted padding, First-Aid can identify the memory objects that are overflowed.

**Contribution 3: Evaluation with real-world applications.** We have implemented First-Aid on Linux and evaluated its functionality and performance with seven applications: three server applications (Apache, Squid, and CVS) and four desktop applications (Pine, Mutt, M4, and BC). These applications contain various types of memory bugs, including buffer overflow, dangling pointer read/write, double free, and uninitialized read. Additionally, we evaluate First-Aid’s performance with the SPEC INT2000 benchmark [164] and four allocation intensive benchmarks [165]. Our experimental results show that, compared to previous approaches, First-Aid has the following advantages:
• **Fast diagnosis and failure recovery.** First-Aid can quickly identify the bugs and recover programs from failures using its diagnosis algorithm. Our evaluation with the seven tested applications shows that the time for failure diagnosis and recovery ranges from 0.084 to 3.978 seconds with an average of 0.887 seconds.

• **Prevention of bug reoccurrence.** First-Aid applies the runtime patches after a program failure and thereby prevents future failures due to the reoccurrence of the same bug. Furthermore, First-Aid stores the generated patches persistently to prevent the bug from occurring on subsequent runs or on other processes running the same program. This improves the overall reliability of the system.

• **Low normal-run overhead.** First-Aid incurs low runtime overhead during normal program execution, i.e., without bugs being triggered. Our evaluation with applications and benchmarks shows that the runtime overhead ranges from 0.4 to 11.6% with an average of 3.7%.

• **Informative bug reports.** First-Aid provides programmers with accurate information about the bug: the bug type, the memory objects triggering the bug, their allocation or deallocation sites, and the relevant illegal memory accesses. Such diagnostic information helps programmers understand both the root cause and the manifestation of the bug.

### 5.2 First-Aid Approach and Working Scenario

Figure 5.1 shows the working scenario of First-Aid. During normal execution, First-Aid periodically takes checkpoints of a running program. Upon a failure, First-Aid diagnoses the bug and generates the corresponding runtime patches. Then it applies the patches to allow the program to recover from the failure and to prevent future failures due to the same bug. After recovery, First-Aid validates the patches and generates a detailed bug report.
**Bug diagnosis.** First-Aid diagnoses the bug by re-executing the failed process from previous checkpoints for multiple iterations. In each iteration, First-Aid rolls back the program to a previous checkpoint, tentatively applies one or more environmental changes to all or a subset of memory objects, and re-executes the program. Based on the results of execution, it narrows down the possible causes of the failure. If First-Aid can identify the bug type and the bug-triggering memory objects, the diagnostic process stops. Otherwise, First-Aid continues the above process until it identifies the bug or times out.

To accurately diagnose the bug, First-Aid uses a combination of two types of environmental changes: *exposing changes* for forcing a certain type of bug to manifest itself and *preventive changes* for preventing a certain type of bug from manifestation. More specifically, to check the occurrence of a bug type $b$, First-Aid applies the exposing change for $b$ and the preventive changes for all other bug types during re-execution. In this way, First-Aid ensures that only the bug type $b$ can manifest itself. If the manifestation of bugs with type $b$ is observed, First-Aid concludes that a bug of type $b$ has occurred before the failure. Otherwise, First-Aid rules out the bug type $b$. Furthermore, based on the bug manifestation, First-Aid can identify the memory objects that potentially trigger the bug.
<table>
<thead>
<tr>
<th>Bug type</th>
<th>Common reason(s) for the bug</th>
<th>Preventive change/ Runtime patch</th>
<th>Exposing change (Bug manifestation)</th>
<th>Patch application point</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer overflow</td>
<td>1. length underestimation</td>
<td>add padding to objects</td>
<td>pad objects with canary values (canary corruption)</td>
<td>allocation</td>
</tr>
<tr>
<td></td>
<td>2. offset miscalculation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dangling pointer read</td>
<td></td>
<td>delay free</td>
<td>fill objects with canary values (failure)</td>
<td>deallocation</td>
</tr>
<tr>
<td></td>
<td>1. premature buffer free</td>
<td></td>
<td>fill objects with canary values (canary corruption)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. forget to set NULL</td>
<td></td>
<td>check parameters (freed twice)</td>
<td></td>
</tr>
<tr>
<td>dangling pointer write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>double free</td>
<td></td>
<td>fill objects with zeros</td>
<td>fill objects with canary values (failure)</td>
<td>allocation</td>
</tr>
<tr>
<td>uninitialized read</td>
<td>1. assume zeros in buffers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. write</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Memory bug types and corresponding environmental changes

Table 5.1 describes both types of environmental changes for each bug type, including buffer overflow, double free, dangling pointer read/write, and uninitialized read. For example, adding padding to both ends of each newly-allocated memory object can prevent buffer overflow bugs, while adding canary-filled padding can manifest buffer overflow bugs as canary corruption. Delaying the recycling of freed memory objects can prevent dangling pointer read bugs from accessing meaningless data as well as prevent dangling pointer write bugs from corrupting useful data. On the contrary, filling delay-freed memory objects with canary values can manifest dangling pointer read bugs as failures and dangling pointer write bugs as canary corruption. Furthermore, zero-filling new objects can prevent uninitialized read bugs, while canary-filling new objects is likely to manifest uninitialized read bugs as failures.

At the end of the diagnostic process, there are three possible results. First, the failure can be caused by non-deterministic bugs (as indicated by a successful re-execution with only timing-based changes and no memory management changes). In this case, First-Aid lets the program continue the normal execution. Second, the failure can also be caused by deterministic memory management bugs. In this case, First-Aid passes the diagnostic results to the next step for patch generation. The third case is deterministic bugs that First-Aid
cannot handle. Examples of such bugs include non-memory bugs and some types of memory bugs (details in Section 5.6). In this case, First-Aid times out and resorts to other recovery schemes.

**Patch generation and application.** Based on the bug diagnostic information, First-Aid generates runtime patches for recovering the program from a failure and preventing future failures caused by the same bug. A runtime patch is a pair of a preventive change corresponding to the identified bug type and a patch application point. As shown in Table 5.1, the preventive change for buffer overflows is to add padding to both ends of bug-triggering objects when these objects are allocated; the preventive change for dangling pointers and double frees is to delay recycling of deallocated bug-triggering objects for a long time until the memory occupied by these objects reaches a customizable threshold; the preventive change for uninitialized read is to fill the contents of bug-triggering objects with all 0’s.

A patch application point is the allocation or deallocation *call-site* of the bug-triggering memory objects. In this work, the *call-site* is defined as the return addresses of the most recent three functions on the stack. It can serve as the “signature” of the bug-triggering memory objects because memory objects with the same call-site of allocation or deallocation often have similar characteristics such as overflow [32,89].

First-Aid stores the generated patches and applies them to the running process. More specifically, a patch takes effect when a later memory allocation or deallocation call-site matches the patch application point. By applying the identified preventive change at these points, First-Aid protects programs from future failures caused by the same bug. Furthermore, since the patches are specific to the program executable (not only the running process), First-Aid applies them to the subsequent runs of the same program and other processes running the same executable. This effectively protects the programs from future failures caused by the same bug and improves the overall system reliability.
Some of the runtime patches generated by First-Aid increase the application’s memory usage. Filling new objects with 0’s incurs no space overhead. Space overhead for the added padding depends on the sizes of padding and the number of padded memory objects. Delaying frees increases memory usage by accumulating delay-freed objects. For our evaluated applications, First-Aid’s runtime patches incur a small memory space overhead (Section 5.7.6). This is because the patches are only applied to a small number of memory objects whose allocation or deallocation sites match call-sites specified in the patches.

Although the situation never arose during our experiments, the extra memory usage incurred by First-Aid’s runtime patches may theoretically exhaust the memory space of a process. To address this issue, First-Aid can disable runtime patching and even start deallocating the oldest delay-freed objects when the memory usage reaches a user-defined threshold. Although deallocating very old delay-freed objects is usually safe for dangling pointer and double free bugs, it can potentially undermine patch effectiveness – the program may fail again. First-Aid allows users to decide how much extra memory space they are willing to pay for better system reliability.

**Patch validation and bug reporting.** After a runtime patch is generated and applied for a quick recovery, First-Aid performs a further step to validate the patch and collect detailed information for the bug report. This step can be done in parallel on a different processor core based on a snapshot of the program so that it does not delay the failure recovery.

To validate that the applied patch is consistently effective, First-Aid re-executes the buggy program region in multiple iterations with randomized memory allocation, and collects detailed traces on memory management operations, patch triggering, and illegal memory accesses, e.g., accesses through a dangling pointer, reads before initialization, etc. Then it checks the traces to validate that the patch has consistent effects on the program execution,
e.g., neutralizing the same number of illegal accesses. If the validation fails, First-Aid removes the corresponding patch and logs the event of validation failure.

Furthermore, First-Aid assists developers in diagnosing and fixing the bugs by providing detailed on-site bug information. For example, the call-sites of bug-triggering memory objects in the diagnosis log and patch information can help developers locate the memory management code related to the bug. Additionally, the execution traces on memory management operations and illegal accesses can help developers understand the process of bug manifestation.

5.3 First-Aid System Design

Figure 5.2 shows the software architecture of First-Aid. It consists of six main user-level and kernel-level components: (1) a lightweight memory allocator extension for assisting bug diagnosis, patch validation, and patch application, (2) error monitor(s) for detecting errors or failures in applications, (3) a checkpoint/rollback component for taking snapshots of running programs and performing rollbacks for diagnosis and recovery, (4) a diagnostic engine for diagnosing the bugs and generating runtime patches (details in Section 5.4), (5) a patch management module for managing the generated runtime patches and controlling patch application, and (6) a validation engine for validating the consistent effects of the patches and collecting on-site diagnostic information (details in Section 5.5).
**Memory allocator extension.** The memory allocator extension collects memory object information and applies preventive and/or exposing changes for diagnosing bugs and surviving failures. It operates in one of three modes: *normal* mode, *diagnostic* mode, or *validation* mode.

In normal mode, the memory allocator extension checks whether the current call-site of a memory allocation or deallocation request matches any patch application point in the available patches. If so, it applies the preventive change in the patch to the memory object. Note that the memory allocator extension relies on the underlying memory allocator for fulfilling memory management requests.

In diagnostic mode, the memory allocator extension performs three functions during re-execution. First, it applies preventive and/or exposing changes, as instructed by the diagnostic engine, to all or a subset of memory objects when they are allocated or deallocated. Second, it collects multi-level call-site information for each memory object allocation and deallocation. Such information is used for bug diagnosis and future patch application. Third, for each deallocation request, it checks whether the memory object has been freed previously, to detect double free bugs.

In validation mode, as controlled by the validation engine, the memory allocator extension introduces randomization in the memory allocation algorithm and keeps traces of memory allocation and deallocation as well as the patch triggering information.

**Error monitor(s).** The error monitors detect errors or failures at runtime and notify the diagnostic engine upon detection. The cheapest way to detect an error or failure is to catch assertion failures as well as exceptions (e.g., access violation) raised from the kernel. Additionally, one can deploy more sophisticated error detectors such as AccMon [28] if they incur low overhead. Our current implementation is based on assertion failures and exceptions.
Lightweight checkpoint/rollback. First-Aid leverages the lightweight checkpointing and re-execution runtime system provided in Rx [91]. More specifically, it takes in-memory checkpoints using a fork-like operation and rolls back the program by reinstating the saved task state. For handling files, it applies ideas similar to previous work [104, 166] by keeping a copy of each accessed file and file pointers at the beginning of each checkpoint and rein-stating it for rollback. Copy-on-write (COW) is also applied here to reduce the overhead. Additionally, First-Aid leverages a network proxy to record network messages during normal execution and replay them during re-execution. More details can be found in the work on Rx [91] and Flashback [104].

Instead of using fixed checkpointing intervals as in Rx, First-Aid dynamically adjusts the checkpointing intervals for balancing the low normal execution overhead and quick recovery time. It does so by monitoring the copy-on-write (COW) page rate, which directly affects the runtime overhead. If the runtime overhead is higher than the threshold $T_{\text{overhead}}$ specified by the user, i.e., the COW page rate is too high, First-Aid gradually increases the checkpointing interval to control the overhead. On the other hand, the recovery time becomes longer when the checkpoint interval is larger. Thus, once the checkpoint interval reaches the user-specified maximal interval $T_{\text{checkpoint}}$, First-Aid stops increasing it.

Patch management. This component manages the patches and makes them available to all the processes that are running the same program. Once the diagnostic engine generates a patch, the patch management component stores it in a central patch pool based on the call-site information. First-Aid maintains a patch pool for each program so that the patches do not mix for different programs. During normal execution, the memory allocator extension queries the patch pool at each allocation or deallocation request.
5.4 Bug Diagnosis

The diagnostic engine uses two phases to diagnose the bug. The first phase (Section 5.4.1) searches for the best checkpoint from which the patch should be applied for surviving the bug. The second phase (Section 5.4.2) performs in-depth diagnosis to identify the bug type and the patch application points, i.e., allocation or deallocation call-sites of the bug-triggering memory objects. In Section 5.4.3, we compare the First-Aid bug diagnosis with that of Rx.

5.4.1 Phase 1: Identify the checkpoint for patching

In order to be both effective and efficient, the patch should take effect from the latest checkpoint before the bug is triggered. To identify that checkpoint, First-Aid rolls back the program to previous checkpoints in reverse chronological order and re-executes the program. It first re-executes the program without any memory management changes. If the program succeeds, then the failure is likely caused by a non-deterministic bug and First-Aid only logs this event and lets the program continue execution. If the program fails, First-Aid again re-executes the program from the checkpoint with all the preventive changes on all subsequently allocated or deallocated memory objects. If the program succeeds this time, i.e., some preventive change is effective, First-Aid stops searching and reports this checkpoint as the latest one before the bug-triggering point. Otherwise, First-Aid continues searching with the checkpoint right before this one. After trying a certain number of previous checkpoints, First-Aid times out and logs the bug as non-patchable. Note that the criterion of failed or successful re-execution in First-Aid is based on whether the program execution can pass the original failure region. Its end point is conservatively defined as a certain number (3 in our experiments) of checkpoint intervals after the failure point.

One challenge in this scheme is the possible misidentification of the latest checkpoint before the bug-triggering point. In some cases, when being applied to a checkpoint that is after
Figure 5.3: A potential misidentification of the checkpoint for patching and the heap marking technique

the bug-triggering point, the preventive changes can appear to be effective by temporarily avoiding the failure. This is because the manifestations of some memory bugs may rely on the heap layout, which could be disturbed when the preventive changes are applied later.

Figure 5.3 shows one such example. During the original execution, from (a) to (c), the dangling pointer $p$ appears at (b) the bug-triggering point, when the object $B$ is prematurely freed. After (b), a checkpoint $C1$ is taken, and then the freed space of $B$ is re-allocated to another object $E$. At the end, a failure occurs because a write via de-referencing $p$ corrupts the data in the object $E$, illustrated as the black dot in Figure 5.3(c). When the program re-executes from the checkpoint $C1$, the preventive changes avoid the failure. This is because the preventive change for buffer overflow adds padding to $E$, denoted as $Epadded$, making it larger than the original $B$. As a result, shown in Figure 5.3(d), the freed space of $B$ is not reused by $Epadded$, which avoids the memory corruption caused by the dangling pointer write.

To address this issue, we devise a technique called heap marking to verify that the bug indeed occurs after the checkpoint. The key idea is to expose the bugs that occur before
the checkpoint. To this end, First-Aid marks the old heap region before re-executing the
program from the checkpoint. More specifically, as shown in Figure 5.3(e), First-Aid marks
all the free chunks in the heap by filling their contents with canary values. Additionally, it
adds padding filled with canary values after the last memory object in the heap. This heap
marking technique exposes previously triggered dangling pointer write or buffer overflow
bugs as canary corruption even though the failure can be accidentally avoided due to heap
layout disturbance. For dangling pointer read bugs, the heap marking technique makes the
failure still occur during re-execution due to the canary.

5.4.2 Phase 2: Identify the bug type and patch application points

After identifying the latest checkpoint before the bug triggering point, First-Aid starts
phase 2 for more in-depth analysis. It first identifies the types of the bugs and then identifies
the allocation or deallocation call-sites of bug-triggering memory objects. Note that, First-
Aid takes into consideration the case where multiple types of bugs are triggered and the
program will not survive unless all of them are avoided. Therefore, the algorithm carefully
separates each bug type.

The basic procedure is to diagnose each bug type one by one using a combination of
preventive changes and exposing changes. We define two sets of bug types, the undecided
set $S_u$ and the identified set $S_i$. Initially, $S_u$ contains all the bug types and $S_i$ is empty.
For each bug type $b$ from the undecided set $S_u$, First-Aid applies the following changes to
all subsequently allocated or deallocated memory objects: the exposing change for the bug
type $b$, and the preventive changes for all other bug types in the set $S_u \cup S_i - \{b\}$. This
way, the bugs with the type $b$ will manifest themselves during re-execution and the potential
interferences from other types of bugs will be prevented. If the bugs of the type $b$ manifest
themselves during re-execution, First-Aid will move the bug type $b$ from the undecided set
$S_u$ to the identified set $S_i$; otherwise, First-Aid will simply remove it from the undecided set $S_u$. At the end of this procedure, the identified set $S_i$ will contain all the types of the bugs that have occurred.

After each new bug type is identified, First-Aid checks whether the current identified set $S_i$ covers all the types of bugs that have occurred. To do so, First-Aid performs one iteration of re-execution, applying preventive changes for bug types in the identified set along with exposing changes for the undecided set. If no bugs manifest themselves, First-Aid stops searching for more bug types; otherwise, it continues.

After identifying all the bug types, the next step is to identify the call-sites of the bug-triggering memory objects. For buffer overflow and dangling pointer write, First-Aid can directly identify the bug-triggering memory objects by looking for canary corruption in padding and delay-freed memory objects, respectively. For double frees, it identifies the bug-triggering memory objects by checking the parameters passed to memory deallocation operations.

For uninitialized read and dangling pointer read, it is more challenging to identify the call-sites through the bug-triggering memory objects themselves, because these bugs only cause incorrect content reads. To address this problem, First-Aid uses a binary search algorithm to identify such call-sites. Specifically, starting with a search range covering all $N$ call-sites after the checkpoint, in each iteration of re-execution, First-Aid applies the exposing change to half of the call-sites in the search range and the preventive change to the rest of call-sites. Depending on whether the bug is exposed, i.e., whether the program fails, it narrows down the search range by half and starts another iteration until the search range contains only a single bug-triggering call-site. The number of iterations for this algorithm is $O(\log N)$.

By also applying preventive changes in each iteration, First-Aid can prevent interference from undiagnosed bug-triggering call-sites that are outside of the current search range. This
is critical for handling the case where multiple call-sites need to be patched at the same time to prevent a failure. In this case, one bug-triggering call-site is guaranteed to be identified by each round of the above binary search, and First-Aid needs to conduct multiple rounds of search and remove the identified call-site from the whole search range after each round. If there are \( M \) bug-triggering call-sites, the search algorithm uses \( O(M \times \log N) \) re-executions in total.

5.4.3 Comparison of First-Aid and Rx bug diagnosis

Our previous work, Rx [91], can also provide quick recovery for failures caused by memory bugs. However, it (intentionally) does not perform in-depth diagnosis since it aims for fast recovery. First-Aid’s goals, on the other hand, are not only to quickly recover programs from failures, but also to prevent reoccurrence of the same bug and provide on-site diagnostic information to developers. Therefore, First-Aid performs accurate diagnosis on memory bugs in the following two respects.

Correctness: First-Aid will not misdiagnose one type of memory bugs as another. It determines one bug type by observing both failure symptoms and possible bug manifestations such as memory content corruption imposed by the exposing change. In contrast, Rx makes its decision based on whether the program survives or fails after applying preventive changes only. It could thus misdiagnose one type of bugs for another. For example, Rx may end up using padding, which is for avoiding a buffer overflow bug, to cure a dangling pointer write bug when the memory write through the dangling pointer happens to corrupt some padding instead of useful data. This cannot happen in First-Aid because when diagnosing buffer overflow, the “delay free” change is also applied to prevent dangling pointer writes from corrupting any other places, including the padding.
**Exactness:** First-Aid identifies a small set of memory objects that potentially trigger the bug, while Rx applies environmental changes to all memory objects during re-execution. For example, Rx stops diagnosis if padding all the new objects can avoid the bug during re-execution. In contrast, First-Aid pinpoints the exact objects where the buffer overflows occur by checking canary values in the padding of all the objects.

### 5.5 Patch Validation and Bug Reporting

Although First-Aid’s diagnosis algorithm will not misdiagnose one type of memory bugs as another, it may diagnose other types of bugs (e.g. semantic bugs) as memory bugs if the bug manifestation depends on memory layout. For example, if a memory write due to a semantic bug happens at the address right after a newly allocated object, it may be diagnosed as a buffer overflow. Even though the chance of such misdiagnosis is small, it undermines the safety and reliability of the program execution in the long run and can mislead developers when they fix the bug in the source code.

To rule out the possibility of such misdiagnosis, the random side-effects of a patch must be distinguished from the desired effects. First-Aid does this by checking that the effect of a runtime patch is consistent under memory layout randomization. During validation, First-Aid re-executes the buggy region of the program for three more iterations with a randomized allocation algorithm. In each iteration, the memory allocator’s activities are logged by First-Aid’s allocator extension, and illegal memory accesses are traced using the dynamic instrumentation tool Pin [150]. Specifically, for each memory allocation or deallocation, First-Aid logs the object address and whether a patch is triggered at this operation. If a patch is triggered, First-Aid also traces the illegal accesses corresponding to the patch: the memory writes to the padding, the memory reads and writes to delay-freed objects, and the reads before initialization. With these traces, First-Aid checks whether the effects of the
patch are consistent among multiple re-executions, based on the following criteria: a) the patch is triggered for the same number of times; b) there are the same number of total illegal accesses prevented by the patch; and c) each illegal access is made by the same instruction at the same offset in the corresponding memory object (the memory object’s address is randomized though). If the consistency check fails, First-Aid removes the runtime patch and logs the validation failure.

The traces collected in the above validation process are organized into a bug report for developers. Specifically, besides the usual bug report (e.g., core dump, event log), First-Aid provides four pieces of information: a) a diagnosis log, which helps developers understand the diagnostic process; b) the runtime patch information (i.e., the bug type and call-sites of the relevant allocation or deallocation operations), which points developers to the critical source code section related to the bug; c) allocation and deallocation traces in the bug region, which show clearly when the runtime patch takes effect and what memory objects are affected; and d) illegal memory accesses in the bug region, which show the instructions that have made illegal memory accesses. The above information about both the root causes of the bug and the bug manifestation process can help developers fix the bug.

5.6 Discussion

Assumptions on memory bugs: First-Aid’s diagnosis algorithm is based on several assumptions about the common characteristics of memory bugs. These assumptions include: a) buffer overflow bug must corrupt data within a neighboring region of the memory object; canary values must not be coincidentally used as normal values in buggy memory objects; programmers intend to initialize the newly allocated buffers with zeros in the case of an uninitialized read bug. These assumptions cover most common cases in real memory bugs
and are also used in previous work [89,91], although exceptions can happen theoretically and result in failed attempts to patch.

**Bug coverage:** Although First-Aid covers common memory bugs, there are some types of bugs that First-Aid cannot fix. Specifically, First-Aid cannot deal with memory leak bugs, whose negative effects are cumulative and cannot be reverted by simply rolling back to a recent checkpoint. One way to survive or prevent memory leak bugs is to deploy memory leak detectors and swap the leaked objects onto disk [167,168]. Additionally, First-Aid cannot handle memory bugs that slip through the deployed error monitors. One way to address this issue is to deploy more rigorous dynamic integrity and correctness checkers as First-Aid’s error monitors. This is currently an active research area. Moreover, First-Aid cannot deal with latent bugs – bugs whose root causes are far away from the error symptoms. Fortunately, this is a rare case, as a previous study [169] shows that most bugs tend to cause quick crashes.

**Customized memory allocator in applications:** To avoid certain memory bugs or improve performance, some applications use customized memory allocation wrappers or even their own memory allocators. Memory allocation wrappers have little impact on First-Aid because First-Aid’s diagnosis and patching are based on multi-level call-sites. If an application-specific allocator is used, First-Aid’s memory allocator extension should be ported to the application-specific allocator. We do not see any particular challenge for porting.

### 5.7 Evaluation and Experimental Results

#### 5.7.1 Experimental setup

Our experimental platform consists of two machines with Intel Xeon 3.00 GHz processors, 2MB L2 cache, 2GB memory, and a 100Mbps Ethernet connection between them. The
operating system kernel is the Linux 2.4.22 modified with Flashback [104] checkpointing support. We ran servers on one machine and clients on the other. We implemented the memory allocator extension by modifying the Lea allocator [170], the default memory allocator used in the GNU C library.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>2.0.51</td>
<td>dangling pointer read</td>
<td>263K</td>
<td>web server</td>
</tr>
<tr>
<td>Apache-uir</td>
<td></td>
<td>uninitialized read</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache-dpw</td>
<td></td>
<td>dangling pointer write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Squid</td>
<td>2.3</td>
<td>buffer overflow</td>
<td>93K</td>
<td>proxy cache</td>
</tr>
<tr>
<td>CVS</td>
<td>1.11.4</td>
<td>double free</td>
<td>114K</td>
<td>version control</td>
</tr>
<tr>
<td>Pine</td>
<td>4.44</td>
<td>buffer overflow</td>
<td>330K</td>
<td>email client</td>
</tr>
<tr>
<td>Mutt</td>
<td>1.3.99i</td>
<td>buffer overflow</td>
<td>86K</td>
<td>email client</td>
</tr>
<tr>
<td>M4</td>
<td>1.4.4</td>
<td>dangling pointer read</td>
<td>17K</td>
<td>macro processor</td>
</tr>
<tr>
<td>BC</td>
<td>1.06</td>
<td>buffer overflow</td>
<td>14K</td>
<td>calculator</td>
</tr>
</tbody>
</table>

Table 5.2: Applications and bugs used in evaluation.

We evaluated First-Aid with a range of applications including three server applications (Apache, Squid, and CVS) and four desktop applications (Pine, Mutt, M4, and BC), as shown in Table 5.2. The applications contain various types of memory bugs, including buffer overflow, dangling pointer read/write, double free, and uninitialized read. Seven of these bugs were introduced by the original developers. We have not been able to locate applications that contain dangling pointer write or uninitialized read bugs. To evaluate First-Aid’s functionality of handling these two types of bugs, we injected them into Apache httpd server separately – Apache-uir contains an uninitialized read bug and Apache-dpw contains a dangling pointer write bug.

5.7.2 Overall effectiveness

We executed these seven applications with First-Aid. The checkpoint intervals are 200 milliseconds. To simulate bug occurrences in real scenarios, we mixed the bug-triggering
Table 5.3: Overall results for First-Aid in surviving and preventing memory bugs. The recovery time is from when the failure is first caught to when the program changes back to normal mode with the applied runtime patches. The validation time is the extra time taken when enabling a three-iteration validation. Apache-uir and Apache-dpw correspond to the cases with injected uninitialized read and dangling pointer write, respectively.

<table>
<thead>
<tr>
<th>Application</th>
<th>Diagnosed bugs</th>
<th>Runtime patch (Number of call-sites applied)</th>
<th>Recovery time (s)</th>
<th>Avoid future errors?</th>
<th>Number of rollbacks for diagnosis</th>
<th>Validation time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>dangling pointer read</td>
<td>delay free(1)</td>
<td>3.978</td>
<td>Yes</td>
<td>28</td>
<td>9.620</td>
</tr>
<tr>
<td>Squid</td>
<td>buffer overflow</td>
<td>add padding(1)</td>
<td>0.858</td>
<td>Yes</td>
<td>7</td>
<td>14.198</td>
</tr>
<tr>
<td>CVS</td>
<td>double free</td>
<td>delay free(1)</td>
<td>0.121</td>
<td>Yes</td>
<td>6</td>
<td>3.887</td>
</tr>
<tr>
<td>Pine</td>
<td>buffer overflow</td>
<td>add padding(1)</td>
<td>0.722</td>
<td>Yes</td>
<td>7</td>
<td>18.276</td>
</tr>
<tr>
<td>Mutt</td>
<td>buffer overflow</td>
<td>add padding(1)</td>
<td>0.017</td>
<td>Yes</td>
<td>7</td>
<td>10.610</td>
</tr>
<tr>
<td>M4</td>
<td>dangling pointer reads</td>
<td>delay free(2)</td>
<td>1.396</td>
<td>Yes</td>
<td>18</td>
<td>3.407</td>
</tr>
<tr>
<td>BC</td>
<td>two buffer overflows</td>
<td>add padding(3)</td>
<td>0.573</td>
<td>Yes</td>
<td>6</td>
<td>2.625</td>
</tr>
<tr>
<td>Apache-uir</td>
<td>uninitialized read</td>
<td>fill with zero(1)</td>
<td>0.102</td>
<td>Yes</td>
<td>9</td>
<td>5.750</td>
</tr>
<tr>
<td>Apache-dpw</td>
<td>dangling pointer write</td>
<td>delay free(1)</td>
<td>0.084</td>
<td>Yes</td>
<td>7</td>
<td>5.718</td>
</tr>
</tbody>
</table>

First-Aid is effective in diagnosing memory bugs. As shown in Table 5.3, for all tested cases, First-Aid correctly identifies the bug type and the call-sites of bug-triggering memory objects. The diagnosis is accurate because First-Aid leverages both preventive and exposing changes for separating the interferences among different bugs.

First-Aid provides quick failure recovery and thereby hides program failures from users. As shown in Table 5.3, the failure recovery time ranges from 0.084 to 3.978 seconds with an average 0.887 seconds. This is because of First-Aid’s lightweight diagnosis algorithm and checkpointing-and-re-execution mechanism. For example, First-Aid quickly pin-points the bug in seven cases after 6-9 iterations of program re-execution, resulting in a failure recovery time less than 1 second. The recovery time for the dangling pointer read case in Apache is
relatively long because its bug-triggering point is a little farther (3 checkpoints) from the failure point.

The number of rollbacks First-Aid performs for diagnosis depends on the bug type. For buffer overflow, dangling pointer write, and double-free bugs, the number of rollbacks is small (6-7 times in our evaluation). This is because the bug-triggering objects can be directly identified by checking the integrity of the canary or parameters of deallocation requests. For uninitialized read and dangling pointer read bugs, more rollbacks may be needed because the patch application points need to be identified via binary search.

Table 5.3 shows that First-Aid is effective in avoiding future errors caused by the same bug. In the experiments, we repeatedly sent the bug-triggering inputs to the applications. First-Aid successfully prevents future memory errors due to the same bug after applying the runtime patches. This is because First-Aid’s patch is accurate (Section 5.7.3) and thereby can be enabled for preventing future errors during the entire program execution.

As shown in Table 5.3, First-Aid successfully validates the generated patches within a small amount of time, i.e., 3-18 seconds, for the tested cases. The validation is done in parallel with a snapshot of the application and does not delay the recovery. During the validation process, the runtime patches show consistent effects, i.e., patches are applied to the same number of memory objects and each illegal access matches across different runs. The validation time is small because First-Aid re-executes the program from the checkpoint identified by the diagnostic engine instead of from the beginning.

5.7.3 Future error prevention

We evaluated First-Aid’s capability of future error prevention and compared it with two alternatives: Rx [91] and the restart method [58, 59], using two representative server programs, Apache (the dangling pointer read bug) and Squid (the buffer overflow bug). In
the experiments, after a certain period of normal execution, we periodically triggered the real bugs by sending bug-triggering requests mixed with normal inputs.

Figure 5.4 shows that First-Aid effectively prevents future errors caused by the same bug while Rx and the restart approach cannot. In the case of Apache (Figure 5.4 (a)), First-Aid diagnoses the bug and recovers the programs from the first failure in around 4 seconds and then maintains stable performance when facing the same bug-triggering inputs repeatedly. This is because the patch generated by First-Aid is correct and accurate so that it can avoid the same memory bug during future program execution. While Rx recovers the program from failures more quickly than First-Aid does, it suffers the same bug repetitively during subsequent program execution. This is because Rx disables the environmental changes after passing the buggy program region. For the restart approach, it suffers the same bug repetitively since the bug is deterministically triggered during subsequent program execution. Figure 5.4 (b) shows similar results for Squid. One difference is that First-Aid recovers the program from the first failure faster for Squid than for Apache because of Squid’s shorter error propagation distance.
Table 5.4 further illustrates the accuracy of First-Aid’s patches as well as the reason why Rx disables the environmental changes after surviving failures. For the seven tested real bugs, we compare patch application in First-Aid and environmental change application in Rx for the buggy region in terms of the number of call-sites and objects being applied with the changes. As shown in Table 5.4, the patch generated by First-Aid is of much lighter weight. For example, First-Aid only affects 1 to 7 call-sites and 1 to 315 objects, while Rx affects 8 to 380 call-sites and 183 to 5004 objects. Furthermore, after passing the buggy region, First-Aid’s patches are less likely to be triggered by normal user inputs. Therefore, First-Aid can enable the lightweight patches during the entire program execution for preventing future errors due to the same bug.

<table>
<thead>
<tr>
<th>Name</th>
<th>Call-sites</th>
<th></th>
<th></th>
<th>Objects</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>First-Aid</td>
<td>Rx</td>
<td>Ratio</td>
<td>First-Aid</td>
<td>Rx</td>
<td>Ratio</td>
</tr>
<tr>
<td>Apache</td>
<td>7</td>
<td>32</td>
<td>21.88%</td>
<td>315</td>
<td>2567</td>
<td>12.23%</td>
</tr>
<tr>
<td>Squid</td>
<td>1</td>
<td>61</td>
<td>1.64%</td>
<td>1</td>
<td>3626</td>
<td>0.03%</td>
</tr>
<tr>
<td>CVS</td>
<td>1</td>
<td>44</td>
<td>2.27%</td>
<td>17</td>
<td>306</td>
<td>5.56%</td>
</tr>
<tr>
<td>Pine</td>
<td>1</td>
<td>380</td>
<td>0.26%</td>
<td>11</td>
<td>2881</td>
<td>0.38%</td>
</tr>
<tr>
<td>Mutt</td>
<td>1</td>
<td>216</td>
<td>0.46%</td>
<td>2</td>
<td>5004</td>
<td>0.04%</td>
</tr>
<tr>
<td>M4</td>
<td>2</td>
<td>8</td>
<td>25.00%</td>
<td>3</td>
<td>183</td>
<td>1.64%</td>
</tr>
<tr>
<td>BC</td>
<td>3</td>
<td>34</td>
<td>8.82%</td>
<td>5</td>
<td>732</td>
<td>0.68%</td>
</tr>
</tbody>
</table>

Table 5.4: The call-sites and memory objects affected by the runtime patch in the buggy region

5.7.4 Bug report

Our manual inspection of the bug reports shows that the on-site diagnostic information is helpful in fixing the bug. Figure 5.5 shows the report generated by First-Aid for the Apache dangling pointer read bug. It consists of five parts: a failure core-dump, a diagnosis log,
Figure 5.5: The bug report generated by First-Aid for the Apache dangling pointer read bug runtime patch information, memory allocation and deallocation traces, and an illegal access trace. Note that some details of the report are omitted in Figure 5.5.

The patch information (item 3 in Figure 5.5) states that the bug is a dangling pointer read and can be avoided by delaying frees in the functions `util_ald_cache_purge` and `util_ldap_search_node_free`, which are the callers of a wrapper (`util_ald_free`) for `free` in Apache. Additionally, the multi-level call-sites show that all the delayed frees are issued indirectly through `util_ald_cache_purge`. This information indicates that the bug is related to the LDAP (Lightweight Directory Access Protocol) cache. By comparing the allocation/deallocation traces (item 4 in Figure 5.5) with and without the runtime patch applied, we notice that without the patch, the freed memory is reallocated later (as indicated by the italic font at item 4 in Figure 5.5). Furthermore, the illegal access trace (item 5 in Figure 5.5) in the buggy region clearly shows that in a few LDAP cache related functions, the application accesses the memory objects that have been freed by the application (but
not deallocated due to First-Aid’s runtime patches). All of these suggest that the bug is in util_ald_cache_purge: dangling pointers are created in the cache cleanup operation.

5.7.5 Runtime overhead during normal execution

![Figure 5.6: Overhead for First-Aid during normal execution. The ‘allocator’ bars show the overhead imposed by the memory allocator extension. The ‘overall’ bars show the combined overhead for First-Aid including both the allocator extension and checkpointing.](image)

We evaluated the normal execution overhead incurred by First-Aid using three sets of programs: the seven applications in Table 5.3, the SPEC INT2000 benchmarks [164], and four allocation intensive benchmarks [165]. We executed First-Aid with normal user inputs in two configurations: enabling only the memory allocator extension, and enabling both the memory allocator extension and checkpointing. The default checkpointing interval in the adaptive checkpointing scheme is 200 milliseconds. For the SPEC benchmarks, we used the reference data sets as the workload. For the other programs, we either chose a large testing program distributed along with the package or constructed synthesized workloads based on the commonly exercised operations, e.g., fetching various sizes of html pages for Apache and Squid servers, exporting a directory with files for CVS server, going through a mail box and reading each email for Pine and Mutt, etc.
Figure 5.6 shows the overhead of First-Aid during normal execution. For desktop programs, we compare the execution time, while for server programs, we compare the average response time. We show the overhead imposed by the memory allocator extension (the second bar) and the overall overhead incurred by both the memory allocator extension and checkpointing (the third bar).

As shown in Figure 5.6, First-Aid incurs low overhead (0.4-11.6% with an average of 3.7%) for the tested applications. This is because both the memory allocator extension and the checkpointing mechanism (the two main sources of runtime overhead) are lightweight. Specifically, First-Aid incurs less than 5% overhead for 17 out of the 22 applications. We observe that the runtime overhead incurred by the allocator extension and the checkpointing mechanism varies for different programs. For some programs that have large memory working sets, such as the SPEC benchmarks, the checkpointing overhead is generally higher due to the frequent copy-on-write page replication. For some programs that perform intensive allocation and deallocation, such as BC, cfrac, the memory allocator extension imposes a relatively larger overhead. This is mainly due to the time spent checking for available patches and maintaining additional meta data on memory management. Since we did not spend much effort on optimization, there is room for improvement.

5.7.6 Space overhead

We evaluated First-Aid’s space overhead, which mainly comes from the runtime patches, the memory allocator extension, and the checkpointing module. Space overhead of runtime patches We evaluated the space overhead incurred by the patches for bugs in the seven applications. We mixed the bug-triggering inputs and normal inputs to trigger the bugs. After patches were applied, we monitored all the memory object allocations and deallocations to measure the space overhead. For the padding patch, we
measured the maximal memory space occupied by the padding. For the delay free patch, we measured the accumulated memory space occupied by the delay-freed objects.

<table>
<thead>
<tr>
<th>Name</th>
<th>Heap size (Kbytes)</th>
<th>Patch type</th>
<th>Space overhead (Bytes)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Squid</td>
<td>2338</td>
<td>padding</td>
<td>1016</td>
<td>0.04%</td>
</tr>
<tr>
<td>Pine</td>
<td>651</td>
<td>padding</td>
<td>1016</td>
<td>0.15%</td>
</tr>
<tr>
<td>Mutt</td>
<td>353</td>
<td>padding</td>
<td>1016</td>
<td>0.28%</td>
</tr>
<tr>
<td>BC</td>
<td>61</td>
<td>padding</td>
<td>3048</td>
<td>4.96%</td>
</tr>
<tr>
<td>Apache</td>
<td>825</td>
<td>delay free</td>
<td>14512</td>
<td>1.72%</td>
</tr>
<tr>
<td>CVS</td>
<td>292</td>
<td>delay free</td>
<td>1496</td>
<td>0.50%</td>
</tr>
<tr>
<td>M4</td>
<td>16343</td>
<td>delay free</td>
<td>128</td>
<td>0.0008%</td>
</tr>
</tbody>
</table>

Table 5.5: The space overhead for patches

Table 5.5 shows that the space overhead incurred by patches for each tested application is small, ranging from 128 bytes to 14512 bytes. This is mainly because patches are only applied to a small number of memory objects. For the padding patch, since the padded objects are usually freed very soon, the extra memory space for padding only appears for a very short time. Specifically, for Squid, Pine, and Mutt, there is only one padded object in the heap at a time; for BC, there are at most three padded objects at the same time. Even though the padding used in First-Aid is relatively large (almost 1 KB), the space overhead is still small. For the delay free patch, the space overhead is accumulated slowly. For Apache, where we triggered the bug aggressively, delay-freeing 315 objects only causes a space overhead of less than 15 KB. Note that First-Aid keeps track of all the delay-freed objects so that they can be freed once the memory consumption incurred by them reaches a customizable threshold (1 MB in our experiments).

**Space overhead of memory allocator extension** We also evaluated the space overhead incurred by the First-Aid memory allocator extension. Detailed results are shown in Table 5.6. In most cases, the memory allocator extension incurs low space overhead, i.e.,
less than 5%. This is because it only adds 16 bytes of meta data for each memory object. However, the relative heap overhead is large in some cases, e.g., 93.17% for cfrac. These applications have a large number of small objects, which makes the relatively space overhead large. We expect that optimizations can reduce the meta data from 16 bytes to 8 bytes, which would further reduce space overhead incurred by the memory allocator extension.

**Space overhead of checkpointing** Table 5.7 shows the space overhead for keeping the checkpoints in memory. Our checkpointing tool uses copy-on-write (COW) to save the dirty pages, so the space overhead is directly affected by the working set for each application. For many applications we tested, the checkpointing space overhead is low, less than 1 MB for each checkpoint. For example, keeping 100 checkpoints for Apache and Squid only takes 6.8 MB and 21.1 MB, respectively. However, for several SPEC INT2000 benchmarks, such
as vortex and bzip2, the checkpointing overhead is large due to their large working sets. In these cases, First-Aid leverages an adaptive checkpointing scheme to alleviate the space overhead. As a result, the space overhead per second is kept low. When the checkpoint interval is increased and old checkpoints are discarded, First-Aid maintains the same length of history while keeping less data in memory. The downside is that the recovery time is longer when the checkpoint interval is larger.

5.8 Summary

In summary, First-Aid is a lightweight runtime system that provides accurate bug diagnosis, failure recovery, and future failure prevention for common memory bugs, including buffer overflow, uninitialized read, dangling pointer read/write, and double free. Using exposing and preventive environmental changes, First-Aid can accurately identify the bug types and bug-triggering memory objects. Based on such diagnostic information, First-Aid generates runtime patches and applies them to a small set of memory objects to tolerate the bug and prevent future failures caused by the same bug.

Our evaluation based on seven applications shows that First-Aid can successfully diagnose and generate runtime patches for common memory bugs. It provides fast failure recovery, i.e., 0.084-3.978 seconds recovery time with an average of 0.887 seconds. The results also show that First-Aid is effective in preventing future bug reoccurrences. Additionally, First-Aid generates a detailed on-site bug report that helps developers understand the root cause and manifestation of the bug. Furthermore, our evaluation with the seven applications, SPEC INT2000, and four allocation intensive benchmarks shows that First-Aid incurs a low runtime overhead (0.4 to 11.6% with an average of 3.7%) during normal program execution.
CHAPTER 6

FINAL REMARKS

In order to enhance the reliability of system software, this dissertation proposes to use runtime support for improving system software reliability. Runtime support here refers to the technique to extend the runtime software system with more functionalities useful for reliability-oriented tasks, such as instrumentation-based profiling, runtime analysis, checkpointing/re-execution, scheduling control, memory layout control, etc. Leveraging runtime support, this dissertation proposes novel methods for bug manifestation, bug detection, bug diagnosis, failure recovery and error prevention in multiple phases in the software development and deployment cycle.

For system software under testing, we propose a method called 2ndStrike to manifest hidden concurrency typestate bugs in multi-threaded system software. 2ndStrike first profiles certain program runtime events related to the typestate and thread synchronization. Based on the logs, 2ndStrike then identifies bug candidates that would cause typestate violation if event order is reversed. Finally, 2ndStrike re-executes the program in multiple iterations with controlled thread interleaving for manifesting bug candidates.

We have evaluated 2ndStrike prototype for two types of concurrency typestate bugs, i.e., invalid pointer dereference and lock operation violation and with three real world bugs from three open-source servers and desktop programs (i.e., MySQL, Mozilla, pbzip2) on a eight-core machine. Our experimental results have shown that 2ndStrike can effectively
and efficiently manifest all three tested software bugs, i.e., within 255 seconds, 178 times faster than stress testing. Additionally, 2ndStrike provides detailed bug reports and can consistently reproduce the bugs after manifesting the bugs once.

For system software deployed in production, we propose a method called DMTracker to detect anomalies in distributed system software running on parallel platforms. Based on the observation that data movements in parallel programs typically follow certain patterns, our idea is to extract data movement (DM)-based invariants at program runtime and check the violations of these invariants. These violations indicate potential bugs such as data races and memory corruption bugs that manifest themselves in data movements. Utilizing the data movement information, we propose to a statistical-rule-based approach to detect anomalies for finding bugs.

We have evaluated DMTracker prototype on a cluster with 64 CPUs. Our experiments with two real-world bug cases in MVAPICH/MVAPICH2 [6], a popular MPI library, have shown that DMTracker can effectively detect them and report abnormal data movements to help programmers quickly diagnose the root causes of bugs. In addition, DMTracker incurs very low runtime overhead, from 0.9% to 6.0%, in our experiments with High Performance Linpack (HPL) [7] and NAS Parallel Benchmarks (NPB) [8], which indicates that DMTracker can be deployed in production runs.

For system software failed due to runtime error, we propose a method called First-Aid to recover failures caused by common memory bugs during production runs and prevent future errors caused by the same bugs. Upon a failure, First-Aid diagnoses the bug type and identifies the memory objects that trigger the bug. To do so, it rolls back the program to previous checkpoints and uses two types of environmental changes that can prevent or expose memory bug manifestation during re-execution. Based on the diagnosis, First-Aid generates and applies runtime patches to avoid the memory bug and prevent its reoccurrence.
We have evaluated First-Aid prototype with seven applications that contain various types of memory bugs, including buffer overflow, uninitialized read, dangling pointer read/write, and double free. The results show that First-Aid can quickly diagnose the tested bugs and recover applications from failures (in 0.084 to 3.978 seconds). The results also show that the runtime patches generated by First-Aid can prevent future failures caused by the diagnosed bugs. Additionally, First-Aid provides detailed diagnostic information on both the root cause and the manifestation of the bugs. Furthermore, First-Aid incurs low overhead (0.4-11.6% with an average of 3.7%) during normal execution for the tested buggy applications, SPEC INT2000, and four allocation intensive programs.

To sum up, the runtime methods and tools can provide great help in enhancing reliability of system software in various scenarios. In addition, combining system support with dynamic analysis in the runtime tools can bring key advantages such as high efficiency, high accuracy, and high usability.
BIBLIOGRAPHY


[61] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, M. Massa, R. Short, J. Vert, J. Bar-


GNU, “Gdb: The gnu project debugger.”


“Parallel bzip2.” http://compression.ca/pbzip2/.


