AFID: An Automated Fault Identification Tool

Alex Edwards
University of California, Irvine
afedward@uci.edu

Sean Tucker
University of California, Irvine
shtucker@uci.edu

Sébastien Worms
Ecole Nationale Supérieure de
Techniques Avancées
sebastien.worms@ensta.fr

Rahul Vaidya
University of California, Los
Angeles
rvaidya@ucla.edu

Brian Demsky
University of California, Irvine
bdemsky@uci.edu

ABSTRACT

We present the Automatic Fault IDentification Tool (AFID). AFID automatically constructs repositories of real software faults by monitoring the software development process. AFID records both a fault revealing test case and a faulty version of the source code for any crashing faults that the developer discovers and a fault correcting source code change for any crashing faults that the developer corrects. The test cases are a significant contribution, because they enable new research that explores the dynamic behaviors of the software faults.

AFID uses a `ptrace`-based monitoring mechanism to monitor both the compilation and execution of the application. The `ptrace`-based technique makes it straightforward for AFID to support a wide range of programming languages and compilers. Our benchmark results indicate that the monitoring overhead will be acceptable for most developers. We performed a short case study to evaluate how effectively the AFID tool records software faults. In our case study, AFID recorded 12 software faults from the 8 participants.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Documentation, Measurement

Keywords

Fault Collection

1. INTRODUCTION

Our research community has traditionally relied upon anecdotal information and intuition about the relative importance of software faults to guide our research. Researchers even sometimes evaluate prototype fault location tools on a few hand-selected faults or synthetically-injected faults. Even when researchers use their tools to detect new faults in existing systems, they must manually verify that the software faults the tool discovers are both real and important. Moreover, the researcher must still provide a proof or other evidence that the tool does not miss important faults. The community has traditionally avoided using large sets of real software faults because few data sets of software faults exist. Furthermore, the data sets that are available typically lack test cases to reproduce the faults or contain manually-injected synthetic faults.

Programming language and software engineering researchers have recently begun to use empirical methods to explore large data sets of software faults. These recent studies have mined fault data from CVS archives that have become available in recent years due to the creation of large, open software systems by the open-source community.

Empirical software fault data sets have the potential to provide a powerful new tool for software engineering and programming language researchers. Traditionally, software engineering and programming language researchers first identify a class of faults to research based on anecdotal evidence that the particular class is important in practice. Fault data sets would provide quantitative data to help researchers identify which classes of software faults pose the greatest problems in practice. Currently, researchers often create a set of simple applications with simplified, seeded software faults that they use to develop their new tool or technique. With a fault data set, researchers could extract several real instances of the given fault class to discover the nuances that appear in practice. We expect that this exploration process will lead to the creation of new, sophisticated analyses that are optimized for the intricacies of real

software faults. Finally, researchers often evaluate their research either by hand selecting a few real software faults or by simply injecting seeded software faults. The fault data set would provide many real software faults that researchers could use to evaluate their tools in an automated fashion.

One problem with most existing data sets is that they lack test cases that reveal software faults. In an attempt to remedy this situation, we tried to manually create a data set of real software faults. Our approach was to ask graduate students to record the faults that they corrected while developing software for their research. For each fault, we asked the students to record: (1) the test case that revealed the fault, (2) a copy of the source code that contained the fault, and (3) the source code change that removed the fault. They found recording this information to be tedious, and instead they often focused on the development task at hand and forgot to record any information. The lesson from this experience is that the successful collection of software faults must be automated.

1.1 Basic Approach

In this paper we introduce a novel approach that monitors the software development process to automatically record software fault data. For each fault, our approach records: (1) a test case that reveals the fault, (2) a version of the source code that contains the fault, and (3) a change to the source code that corrects the fault.
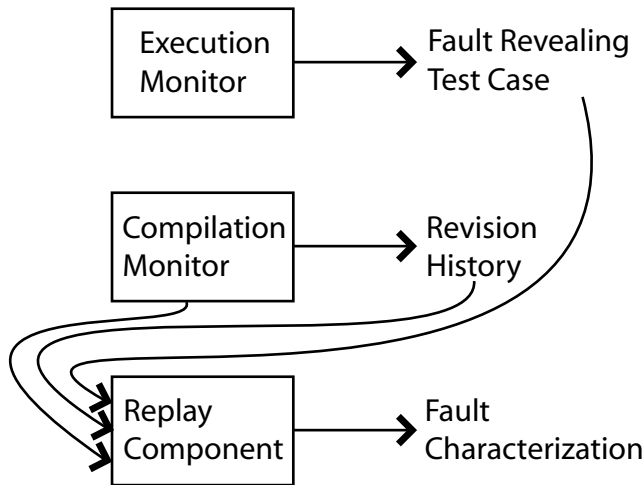

Figure 1: Overview of Fault Characterization

We have implemented this approach in the Automated Fault IDentification Tool (AFID). AFID automatically records software faults by monitoring the compilation and execution steps of the software development process. The underlying design principle for AFID is to record as much software fault data as possible while imposing minimal runtime overheads and requiring minimal assistance from the developer. The final goal of the AFID project is to collect fault data from a wide range of software developers working on real projects. Therefore, requiring the developer to ac-

tively participate in recording faults would potentially make finding developers to use AFID much more difficult. According to this principle, AFID has been designed to only detect faults that actually cause crashes. AFID does not recognize more subtle correctness faults because that would burden the developer with describing the desired behavior of an application. We expect that we can learn much interesting information from crashing faults alone.

Figure 1 presents an overview of our approach. Our approach contains the following key components:

- **Execution Monitor:** The execution monitor traces executions of the application under development. The execution monitor records the inputs to the application. If the application crashes, the execution monitor uses the recorded inputs to create a test case that reproduces the observed failed execution. At this point, AFID records (1) a test case that contains the application inputs that reveal the fault and (2) the source code version in which the fault was discovered.

- **Compilation Monitor:** The compilation monitor traces executions of the compiler to automatically discover which source files comprise the application under development. Whenever the application is recompiled, the compilation monitor records both a list of any new source files it discovers and a list of all source files that have changed since the last compilation. The compilation monitor then updates its internal subversion repository with any changes that have been made to the application. Finally, the compilation monitor invokes the replay component to check if the recent changes correct any known software faults.

- **Replay Component:** The replay component executes the newly compiled version of the application on all of the unresolved fault revealing test cases. If the application executes one of the fault revealing test cases without crashing, the replay component assumes that the most recent code change corrected the underlying fault. The replay component records the current version identifier as the fault correcting code change. The replay component then marks the test case as resolved. Researchers have developed many replay systems for debugging applications [1, 15, 6]. These other systems replay the exact execution, while AFID generates test cases from the application inputs with the goal of running different versions of the application on the same test case. The exact executions of these new versions can potentially differ from the version in which the test case was first recorded.

1.2 Contributions

This paper makes the following contributions:

- **Automated Fault Collection Strategy:** It presents heuristics that monitor the development process to automatically record fault revealing test cases

and automatically detect which code changes correct these software faults.

- **Process Monitoring Technique:** It presents a language and tool chain independent technique to monitor both the executions of the application under development and the evolution of its source code.

- **Automated Recording of Test Cases:** It presents a technique to automatically record test cases from failed executions. These test cases can potentially be incorporated into the application's regression test suite.

- **Monitoring Overhead Measurement:** It presents measurements of the runtime overhead of AFID's monitoring for both a computationally bound benchmark and an I/O bound benchmark.

- **Experience:** It presents our experience using the tool to collect software faults.

The remainder of the paper is structured as follows. Section 2 presents an example to illustrate how the approach works. Section 3 presents the automatic fault collection tool AFID. Section 4 discusses possible privacy concerns. Section 5 presents both overhead measurements and our initial experiences using AFID to collect software faults. Section 6 presents related work; we conclude in Section 7.

2. EXAMPLE

We next use an example to illustrate our approach. Let's suppose that the developer uses a text editor to write the program shown in Figure 2. This program takes a command parameter that specifies its input file. The program then opens this file and reads a series of commands from it. These commands instruct the program to either write a digit to an array element, print an array element, or sum the array elements. Note that line 20 is missing a `break` statement, which would cause the execution of the sum command to erroneously continue into the code for the read command.

2.1 Monitoring Compilation

After a developer finishes writing the program, he/she would typically compile the program using one of many Java compilers. AFID tracks the evolution of the program's code by monitoring the execution of the compiler. When the compiler compiles the example program, it would make a system call to the operating system to open `Example.java` for read access. AFID intercepts the `open` system calls made by the compiler to detect when the developer adds new source files to the application. AFID then examines the file's extension to determine that this file contains source code for the application. The primary benefit of this approach is that it enables AFID to support most compilers while not requiring the developer to manually identify the source files that comprise the application's source code.

```
1   public class Example {
2     public static void main(String[] arg)
3      throws IOException {
4       int array[]=new int[10];
5       FileReader fr=new FileReader(arg[0]);
6       while(true)
7         switch(fr.read()) {
8         /* Write to array element. */
9         case 'W':
10          int woff=fr.read()-'0';
11          int val=fr.read()-'0';
12          array[woff]=val;
13          break;
14        /* Sum array. */
15        case 'S':
16          int sum=0;
17          for(int i=0;i<10;i++)
18            sum+=array[i];
19          System.out.println(sum);
20        /* This line is missing a break. */
21        /* Print array element. */
22        case 'R':
23          int roff=fr.read()-'0';
24          System.out.println(array[roff]);
25          break;
26        case -1:
27          return;
28        }
29    }
30  }
```

Figure 2: Faulty Example Program

2.2 Monitoring Program Execution

In the normal development process, we expect that the developer would next execute the example program on an input file. Figure 3 presents an input file for the example program. The input file contains a sequence of three commands: `W23` instructs the program to write the value `3` to array element 2, `S` instructs the program to sum the array elements, and `R2` instructs the program to print the second array element. Note that this input file invokes the sum functionality and reveals the fault in the sum functionality of the example program.

```
W23SR2
```

Figure 3: Fault Revealing Input File `input.txt`

Typically, the developer would next execute the example program on this input file by typing `java Example input.txt`. AFID's execution monitor would then record the command line used to execute the program. The program's execution opens the file `input.txt` for read access using the `open` system call. AFID's process monitor intercepts this call and records that the execution reads from

the file `input.txt`. When the program processes the summation command, the fault causes the program to continue into the array element printing code. The program then uses the byte intended to specify the read command as an index. This causes the program to exit due to an array out of bounds exception. AFID inspects the execution's exit value to determine that the program crashed.

The goal is to create a test case that can reproduce the crash. AFID records the command line that was used to invoke the fault revealing execution, makes copies of all the input files that the program opened, stores a trace of any console user interactions, and stores the mapping from the pathnames of the files that the program opened to the copies made by AFID.

2.3 Detecting Fault Corrections

We expect that the developer will eventually correct any important software faults. Figure 4 gives the source code for the corrected example. The developer has corrected the fault in this program by changing line 20 to a `break` statement. When the developer compiles the corrected program, AFID would then detect that line 20 of the `Example.java` file has been changed.

```
1  public class Example {
2    public static void main(String[] arg)
3      throws IOException {
4      int array[]=new int[10];
5      FileReader fr=new FileReader(arg[0]);
6      while(true)
7        switch(fr.read()) {
8        /* Write to array element. */
9        case 'W':
10         int woff=fr.read()-'0';
11         int val=fr.read()-'0';
12         array[woff]=val;
13         break;
14       /* Sum array. */
15       case 'S':
16         int sum=0;
17         for(int i=0;i<10;i++)
18           sum+=array[i];
19         System.out.println(sum);
20         break;
21       /* Print array element. */
22       case 'R':
23         int roff=fr.read()-'0';
24         System.out.println(array[roff]);
25         break;
26       case -1:
27         return;
28       }
29   }
30 }
```

Figure 4: Corrected Example Program

AFID then invokes its replay component to replay the fault revealing test cases on the new version of the example program. The replay component executes the example program using the recorded command line. When the example program executes, it makes a system call to open the `input.txt` file. AFID intercepts this system call before the operating system processes it and changes the filename to the name of the copy in the test case. Because the developer corrected the underlying software fault, the program executes correctly on the test case. AFID inspects the program's return value to determine that the underlying fault was corrected.

At this point, AFID has identified that the most recent source code change corrects the underlying software fault. AFID has recorded the following information for the example fault: (1) the buggy version of the example program from Figure 2, (2) the test case that reveals a fault in the buggy version from Figure 3, and (3) a diff that gives the source code change that corrects the fault (for this example, replacing line 20 with `break;`). AFID records all of this information in its record for this fault. It then (optionally) uploads this fault information to a centralized fault repository.

3. AUTOMATED FAULT IDENTIFICATION

We have architected AFID as three basic components: (1) the execution monitor, which detects crashes and creates fault revealing test cases to reproduce these crashes, (2) the compilation monitor, which identifies new source files and tracks changes to the source code, and (3) the replay component, which detects when a source code change corrects a fault. Each component of AFID uses the same basic monitoring strategy — they intercept the system calls that the application or compiler uses to communicate with the underlying operating system. This approach enables AFID to easily support many different compilers, virtual machines, and programming languages with only small configuration changes.

The goal of AFID is to collect complete information for software faults. AFID collects the following information for each fault:

- **Fault Revealing Test Case:** For each reported fault, AFID records the test case that reveals this fault.

- **Version of the Application with the Fault:** For each reported fault, AFID records a copy of the source code of the application version that contains the fault. For space efficiency, this is stored as a version identifier to a version control system repository.

- **Fault Correction:** For each reported fault, AFID records the source code change that corrected the fault. For space efficiency, this is stored as a version identifier to the version control system update that stores the correction.

- **Revision History of the Application:** AFID records a fine-grained revision history of changes to the application's source code.

3.1 Recording Test Cases

AFID's execution monitor traces the executions of the application under development to generate fault revealing test cases. The execution monitor records the inputs to the application's execution by intercepting the system calls from the application to the underlying operating system.
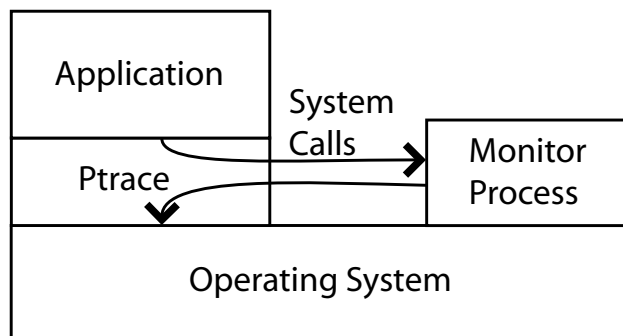

Figure 5: Ptrace Interface

The execution monitor uses the `ptrace` system call to monitor executions of the application under development [5]. Figure 5 presents an overview of the approach. The `ptrace` interface allows the execution monitor to intercept system calls made by the application under development before the operating system processes the call. We next describe our `ptrace`-based approach in more detail.

The execution monitor begins by forking a new child process, the child process calls `ptrace` with the `PTRACE_TRACEME` option to request tracing, and then the child process calls the `exec` system call to execute the application under development. When the child calls the `exec` system call, the previous invocation of `ptrace` with the `PTRACE_TRACEME` option causes the child process to stop before executing the new application image.

The monitoring process then calls the `ptrace` system call with the `PTRACE_SYSCALL` option and then calls `wait`. The next time the child process makes a system call, the operating system suspends the child process and wakes up the monitoring process. When the execution monitor is awoken, it uses `ptrace`'s `PTRACE_GETREGS` option to read the system call parameters to determine the type of the system call. If the child process is opening a file, the execution monitor inspects both the name of the file and the file access mode. The execution monitor uses `ptrace`'s `PTRACE_PEEKDATA` option to read the file's name out of the monitored process's memory space.

If the monitored application has requested to open the file for write access, the execution monitor must immediately make a copy of that file. If AFID delays copying the file to check if the monitored application crashes, the monitored application would likely have already changed the contents of the file. If the monitored application has requested to open the file for read access, the execution monitor uses a lazy copy strategy. It delays the overhead of copying the file until the monitored application actually crashes.

When the monitored application exits, the execution monitor inspects its return value to determine whether it crashed. If the monitored application has crashed, the execution monitor makes copies of all of the files that the monitored application read. It then stores the mapping between the pathnames that the monitored application used to access the files and the files' copies in a text file in the test case.

3.1.1 Recording User Interactions

We next describe how AFID records user interactions. AFID uses the same `ptrace`-based mechanism to record a trace of read events from standard input and write events to standard output. One potential issue with simply replaying the exact user interaction is that changes in the program (or even the time) may change the text that the program outputs. If we require that the output match exactly, the test case will have significant problems generalizing to future versions of the program. Instead, for each input event AFID computes the shortest suffix of the program output since the last input event that uniquely identifies when the input occurred. This fuzzy matching approach allows the recorded test case to generalize over small changes to the program's output.

3.1.2 Duplicate Test Cases

One potential issue is that the developer may rerun the same test case multiple times. To avoid storing multiple copies of the same test case, the monitor computes a hashcode for each test case. The monitor then compares this hashcode to a list of hashcodes for the other test cases. If AFID detects a hashcode match, it deletes the new test cases. AFID makes the assumption that the hash values do not collide. In the unlikely event that two different test cases have the same hash value, AFID only stores the first test case.

3.1.3 Filtering Inputs

The monitored application's execution typically reads many files that would not be considered inputs to the application. For example, the dynamic linker may load library files or a virtual machine may load class files, virtual machine components, virtual machine configuration files, and various system files. These extraneous input files would make the test cases very large. Moreover, recording input files from dynamic libraries or virtual machine internals could make the test case specific to the exact execution environment.

AFID employs a filtering mechanism to remove these extraneous files. The filter mechanism uses a configuration file that contains a list of regular expressions that match the filenames to exclude from the test cases. AFID can automatically generate this configuration file for Java applications by

monitoring the execution of a dummy Java application and then generating a list of files that are loaded by the JVM. AFID then adds some default expressions that exclude class files and other known extraneous files.

3.2 Monitoring Compilation

AFID stores a copy of the source code each time the developer compiles the application. To efficiently store multiple versions of the application's source code, AFID maintains an internal subversion repository. Subversion is an open-source version control system with support for atomic commits [2]. Each time the developer compiles the application, the compilation monitor component of AFID monitors the compiler to determine which files contain the application's source code. The compilation monitor uses the `ptrace`-based monitoring technique described in Section 3.1 to detect application source files.

When the compilation monitor discovers a new source file, it adds the file to its internal subversion repository. Then the compilation monitor commits all of the source code changes since the last compile to its internal subversion repository. Finally, the compilation monitor calls the replay component to replay all of the unresolved fault-revealing test cases on the new version of the application.

One challenge is that the subversion version control system that AFID uses to store its internal repository for the application creates hidden directories in the source code tree. If the developer also uses subversion, the directories for the developer's repository and AFID's internal repository would conflict. To maintain compatibility with subversion, the compilation monitor makes its own copy of the source code tree to use for its internal subversion repository. To avoid the overhead of copying large files, the compilation monitor makes hardlinks from the filename in its internal copy of the source code tree to the original in the developer's source code tree. The compilation monitor then calls subversion to build its internal repository using this copy of the source code tree.

3.3 Replaying Test Cases

The replay component checks whether the most recent source code changes correct any of the faults AFID has recorded. The basic strategy is to execute the new version of the application on each of the unresolved fault revealing test cases. If the application executes successfully, the replay component has determined that the most recent code change corrects the fault revealed by that test case. The replay component then stores the subversion version identifier of the source code version that corrects the fault in the test case and marks the test case as resolved.

3.3.1 Sandboxing Replay

A naive replay implementation would simply copy the files in the test case back to their original locations and then execute the application. However, this strategy has serious potential consequences — the replay component could po-

tentially overwrite important files when copying the test case files or the execution of the application could overwrite important files. AFID prevents the replay of applications from overwriting important data by using the same `ptrace`-based technique to partially sandbox the application. This sandbox is not intended to isolate a hostile application — it is intended to prevent the replay of normal applications from accidentally overwriting important files.

The replay component implements the sandbox by intercepting file open requests. If the application makes a file open request for one of the test case files, the replay component will redirect the request to the file in the test case. If the application makes a request for an excluded file, the replay component will pass the open request unmodified to the operating system. Note that if the application is modified or the fault is corrected, the application can open files that were neither present in the test case nor filtered by the filter expressions. It is straightforward to modify the replay component to make a copy of that file and redirect the request to the copy. This sandbox provides the application with the illusion that the test case files are in the same location as the files in the original execution — a secondary benefit of this approach is that it enables the test case to reproduce software faults that depend on the exact location of the input files.

We next discuss how we implement the sandbox using the `ptrace` system call. The replay component begins by making a copy of the test case. It then starts the monitored application's execution inside the partial sandbox. The basic idea is to use the technique described in Section 3.1 to intercept `open` system calls. When the replay component intercepts an `open` system call, it retrieves the requested filename. If the filename is contained in the test case, the replay component will modify the system call's parameters to open the copy in the test case. The replay component changes the `open` system call's filename by using `ptrace`'s `PTRACE_SETREGS` option to modify the register that stores the pointer to the filename to point to a new memory location. Then the replay component uses `ptrace`'s `PTRACE_POKEDATA` command to write the filename of the copy to this new memory location. The replay component then restarts the application to allow the operating system to service the system call.

Note that the replay tool must obtain memory in the other application's memory space to store the filenames of the copies. The replay system obtains this memory by intercepting the first system call that the application performs. The replay system rewrites this system call's parameters to change it into a `brk`[1] system call to obtain the initial bottom of the heap. The replay system restarts the application and then the operating system executes the injected `brk` call. The application is halted after the system call is performed and control is returned to the replay tool. The replay tool then modifies the program counter to cause the application to re-execute the same system call. The replay tool then

---

[1]The `brk` system call is used to read and set the bottom of the heap. This system call is the primitive that underlies library-based memory allocation functions such as `malloc`.

repeats the same system call injection strategy to inject a second `brk` system call that sets the new bottom of the heap. The replay system has now allocated its own space in the application's memory space. The replay system then resets the program counter another time to perform the initial system call. If the application later uses the `exec` system call to load a new binary, the replay system repeats the same procedure to obtain space in the newly loaded application's memory space.

If the application's execution is successful, the replay component has discovered that the most recent source code change corrects the fault. Note that the test case may not contain some files that were present on the local disk. In this case, it is straightforward for the replay component to add copies of these files to the test case.

3.3.2 Termination

It is possible that the developer may make a source code change that causes the application to loop on an unresolved test case. To address this issue, AFID records the elapsed time for each execution of the application. The replay component then uses this record of execution times to estimate an upper bound on the application's execution. When the application executes for longer than this bound, AFID assumes that the application is looping. This prevents the replay component from waiting indefinitely for a non-terminating computation. Note that in the worst case, when a timeout is used to incorrectly identify an execution as looping, the effect is only to prevent AFID from recognizing a fault correction.

3.4 AFID Server

AFID uses a web-based server application that aggregates the faults discovered by the AFID client. AFID supports two update modes: automated and manual. The automated mode automatically uploads a test case once the client has discovered the fault correcting code change. The manual mode allows the developer to manually control the uploading process. We developed the manual mode in anticipation that some developers will wish to maintain control over when uploads are performed. The client uploads the fault revealing test case, the version identifier for the source code version whose execution generated the fault revealing test case, the version identifier for the code change that corrects the fault revealing test case, and the latest version of AFID's internal subversion repository for the application.

3.5 Recording Regression Tests

The design of AFID is focused on recording fault data for research. However, we expect that practitioners may also find AFID beneficial for recording regression tests. In particular, AFID's fault data set includes test cases for each fault that the developer has discovered and corrected. We expect that this library of test cases may be a useful addition to the application's regression test suite. AFID's execution monitor provides the functionality to cleanly bundle the component files into test cases. AFID's replay component allows the test cases to be easily replayed on future versions of the application. Practitioners may find AFID particularly useful for test cases that contain files that are scattered throughout the directory structure or that involve the modification of common configuration files or any other files that are shared with other applications.

4. PRIVACY CONCERNS

Privacy may be a concern when using AFID for software fault user studies. Because AFID records all source code changes along with the application inputs, it may be possible to discover the actual identity of a study participant from the comments, coding style, project, and test cases. We expect that user studies will not use AFID to monitor the development of applications that contain sensitive source code or that may process sensitive inputs. Because a developer may accidentally input private information into the application under development, AFID supports a manual test case transfer mode that allows the developer to maintain complete control over including test cases in a data set.

5. EXPERIENCE

We next discuss our experience using the AFID implementation. The AFID implementation consist of approximately 3,100 lines of C code and shell scripts. The implementation is available for download at `http://demsky.eecs.uci.edu/afid/`. In this section, we report our measurements of AFID's monitoring overhead on two applications and then discuss our experiences using AFID to monitor software developers.

5.1 Monitoring Overhead

We measured AFID's monitoring overheads on a workstation with a 2.2 GHz Core 2 Duo, 1 GB of RAM, and Debian Linux running kernel version 2.6.23. We used version 1.5.0_13 of Sun's HotSpot JDK.

We used two different benchmarks: the Jasmin byte code assembler and the Inyo ray tracer. We used version 2.3 of the Jasmin bytecode assembler. It contains 11,450 lines of code and is available for download at `http://jasmin.sourceforge.net/`. We selected Jasmin because assembling bytecode involves a relatively large amount of I/O and therefore is likely to incur a significant monitoring overhead under AFID. The Inyo ray tracer contains 5,843 lines of code and is available for download at `http://inyo.sourceforge.net`. We selected Inyo to give results for a longer-running, computational-bound benchmark.

Table 1 presents the overhead measurements. Without monitoring, we measured the time to compile Jasmin as 1.07 seconds and the time to compile Inyo 0.77 seconds. With monitoring and updating AFID's internal SVN repository, we measured the time to compile Jasmin as 4.32 seconds and Inyo as 3.54 seconds. We then measured the time to compile with monitoring but without updating the internal SVN

	Jasmin	Inyo
Normal compile	1.07 s	0.77s
Monitored compile with svn	4.32 s	3.54 s
Monitored compile without svn	1.40 s	0.95 s
Normal execution	0.22 s	31.88 s
Monitored execution	0.47 s	32.64 s

Table 1: Monitoring Overhead

repository for Jasmin as 1.40 seconds and for Inyo as 0.95 seconds. This number is important because it measures how long the developer must wait for the application to be compiled. Indeed, it is conceptually straightforward for AFID to return control to the developer at this point and perform the SVN repository updates in the background.

Our workload for Jasmin consisted of all of the examples contained in the Jasmin distribution. Without monitoring, Jasmin took 0.22 seconds to execute on this workload. With monitoring, Jasmin took 0.47 seconds to execute on this workload. Our workload for Inyo consisted of the model file included with the Inyo distribution. Without monitoring, Inyo took 31.88 seconds to execute on this workload. With monitoring, Inyo took 32.64 seconds to execute on this workload. We expect that Jasmin's monitoring overhead of 113% represents a worst case and Inyo's monitoring overhead of 2% represents the best case. We expect that this range of overhead will be acceptable in most development environments.

5.2 Case Study

Our case study attempts to explore the most basic question one can ask about the AFID tool: Does it effectively record real software faults? To answer this question, we recruited a population of software developers and had each developer complete a programming problem while being monitored by AFID.

5.2.1 Developer Population

One goal of this case study is to verify that AFID's fault identification heuristics work with the wide range of debugging approaches used by developers. We attempted to represent this wide range in our study population by recruiting 8 students with diverse backgrounds: the study participants had widely varying educational backgrounds, industrial experience, years of programming experience, and countries of education. Their educational backgrounds ranged from current undergraduate students to doctorates. Several participants had industrial experience while other participants had only academic experience. The study participants were educated in the United States, China, and India.

5.2.2 Methodology

We installed the AFID tool in each developer's account and instructed the developer in the use of the AFID tool. We then asked each developer to complete a programming

problem in Java while using the AFID monitoring tool. We selected the programming problems from practice programming contest problems and basic data structure implementation problems.

5.2.3 Fault Breakdown

After a developer completed the problem, we asked the developer to go through the fault reports that AFID had collected, verify that the recorded corrections were correct, and if so, to describe the underlying programming error. We then examined their responses and attempted to classify the faults by their underlying programming errors. Table 2 presents a breakdown of the recorded faults by the type of the underlying programming error. One the two largest categories were errors in the logic for parsing the input and null pointer dereference errors. The parsing errors typically involved errors in reading the specification of the input format. The null pointer dereference errors were not simple omitted null pointer checks, but instead a wide range of logic errors that caused the programs to dereference null pointers.

Fault Type	Count
Parsing logic error	3
Null pointer dereference error	3
Initialization error	2
Missing condition check	1
Loop bound error	1
Shadowed field	1
Incorrect comparison	1

Table 2: Fault Breakdown

We observed that even though AFID can only detect failures that cause the application to throw an exception, in our case study, AFID recorded a rich set of software faults. The insight is that very high level conceptual errors can cause software application to exhibit low-level failures. Even in this small case study, AFID recorded high-level faults including errors caused by misunderstandings of the exact format of the input file.

5.2.4 Fault Detection Errors

We next discuss how often AFID recorded the correct fault-correcting source code change. For each recorded fault, we asked the participant to verify whether AFID had correctly identified this change as fault correcting. We report the results in Table 3. The table contains a row for each participant in the study. The first column gives designators for each participant, the second column reports the number of faults AFID recorded for that participant, and the third column reports how many of these faults contained the correct fault correcting source code change.

We note from the table that AFID has recorded fault data entries that contain the wrong fault correcting code change for two of the study participants. We then examined the incorrect fault correcting source code changes to better un-

Participant	Number of Recorded Faults	Number of Verified Corrections
A	2	2
B	1	1
C	4	2
D	8	5
E	1	1
F	1	1
G	0	0
H	0	0

Table 3: Fault Counts by Participant

derstand the problem. We found a surprise — these two study participants employed an experimental approach to correcting software faults. They made changes to the code to improve their understanding of why the application threw an exception. For example, in one case the participant commented out the line of code that was throwing the exception. AFID then detected that this source code change cause the program to no longer crash and record the experimental code change as the fault correcting code change.

AFID currently includes support for calling the compiler without monitoring. We plan to instruct future AFID users to use this functionality when they employ such debugging strategies. In response to this case study, we have extended AFID to verify suspected fault correcting source code changes with the developer before adding them to the repository.

5.2.5   Multiple Corrections

When we manually reviewed the fault correcting source code changes, we noticed one source code change that contained corrections for many different faults. In this case, what happened was when the developer discovered the first fault, he realized he had made the same mistake two more times in the same method and corrected all instances of this mistake. We observed only a single instance of a source code change that corrected multiple faults. We foresee that future versions of AFID will allow a developer to note when the developer believes that a source code change corrects multiple fault instances.

5.2.6   Developer Feedback

The user experience for AFID users is a concern for large user studies. After the user study, we asked the participants to provide feedback about their experience using the AFID monitoring tool. One participant commented that using the tool was unnoticeable as the user just used the regular javac and java commands. The participant thought the general experience was very good. One participant was "amazed...at how accurately AFID caught my critical bugs". Several participants noticed a slight delay when compiling programs. We plan to address this delay by performing both the repository updating and test case replaying in the background.

6.   RELATED WORK

Researchers have recently developed tools to mine CVS repositories to collect some of this information [13, 8, 16, 17, 9]. The CVS mining research identifies CVS commits that correct software faults through a heuristic analysis of the CVS checkin comments. Researchers have discovered many interesting properties including that code changes on Fridays are more likely to cause problems [11]. Other research discovers implicit interface rules by searching for code changes that occur together [7]. The primary way that our work differs from previous work on CVS mining is that our work provides fault revealing test cases in a format suitable for automated tools. The extra information provided by these test cases will enable empirical software research to explore software faults in new ways — for example, the test cases will enable researchers to use dynamic analyses to explore the faulty executions. Our work also more precisely characterizes software faults as compared to CVS as developers sometimes commit CVS updates that both correct a software fault and make other changes. CVS mining techniques cannot distinguish between the fault correcting changes and any other bundled changes and therefore can extract software fault corrections that are too large.

Researchers have also developed data sets of applications with seeded faults [4]. These data sets are limited in size because they are labor intensive to create — researchers must manually seed faults and create test cases that reveal these faults. While these data sets have proven to be a useful tool, potential differences between seeded faults and real software faults can threaten the validity of experiments. Moreover, because the software faults are seeded, the data set does not contain information that can be mined to learn about real-world software faults.

The iBUGS project is based on the observation that after developers correct a bug, they often add regression tests designed to ensure that future changes do not reintroduce similar bugs [3]. Their approach searches CVS commit messages for text that indicates that the change corrects a bug. They then build pre-fix and post-fix versions of the application and run the versions on the test suite to identify any test cases that reveal the given fault. They have successfully used this technique to build a repository of software bugs.

Researchers have developed many replay systems for debugging applications [1, 15, 6]. These other systems replay the exact execution, often with the goal to help developers deterministically replay software bugs in multi-threaded programs. AFID's goal is to execute new versions of the application on the same test case. The exact execution of the replay under AFID will often differ as the underlying code has likely been modified.

AFID relies on the `ptrace` interface to monitor both application compilation and execution. Researchers have used the `ptrace` interface to inject faults into applications [12] and to safely execute untrusted code [10]. Researchers have also used similar program monitoring techniques to implement user space file systems [14].

7. CONCLUSION

Data sets of real software faults have the potential to enable the creation of new tools for software engineering and programming language researchers. Our previous experience shows that manual efforts to collect such data are tedious. The AFID tool is a new approach for recording software fault data. A key benefit of AFID is that the data it collects includes fault revealing test cases in addition to a faulty version of the application and the fault correcting source code change. The key results in this paper include (1) a technique to automatically record software faults without requiring developer intervention, (2) the implementation of this technique in the AFID tool, (3) an evaluation of the overhead of these techniques, and (4) our experiences using the tool to record real software faults. Our measurements of AFID's monitoring overhead indicate that this approach is feasible for many development environments. Our case study results indicate AFID's approach to automatically recording software faults works effectively in practice.

8. REFERENCES

[1] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59, New York, NY, USA, 1998. ACM.

[2] B. Collins-Sussman. The Subversion project: Building a better CVS. *Linux Journal*, 2002(94):3, 2002.

[3] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, 2007.

[4] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering, An International Journal*, 10(4):405–435, October 2005.

[5] M. Haardt and M. Coleman. Ptrace(2). Linux programmer's manual, Section 2, November 1999.

[6] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987.

[7] B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2005.

[8] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceeding of the 28th International Conference on Software Engineering*, November 2006.

[9] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[10] R. Sekar, V. N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications, October 2003.

[11] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? on fridays. In *Proceedings of the International Workshop on Mining Software Repositories*, 2005.

[12] R. R. Some, W. S. Kim, G. Khanoyan, L. Callum, A. Agrawal, and J. J. Beahan. A software-implemented fault injection methodology for design and validation of system fault tolerance. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks*, 2001.

[13] J. Spacco, J. Strecker, D. Hovemeyer, and W. Pugh. Software repository mining with Marmoset: An automated programming project snapshot and testing system. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.

[14] R. P. Spillane, C. P. Wright, G. Sivathanu, and E. Zadok. Rapid file system development using ptrace. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, 2007.

[15] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/replay tool for observation-based testing. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 158–167, New York, NY, USA, 2000. ACM.

[16] C. Williams and J. K. Hollingsworth. Bug driven bug finders. In *In Proceedings of International Workshop on Mining Software Repositories*, May 2004.

[17] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining revision history. *IEEE Transactions on Software Engineering*, 30(9):574–586, September 2004.