# Chronicler: Lightweight Recording to Reproduce Field Failures

Jonathan Bell, Nikhil Sarda and Gail Kaiser
Department of Computer Science
Columbia University
New York, NY 10027
Email: {jbell,kaiser}@cs.columbia.edu, ns2847@columbia.edu

***Abstract*—When programs fail in the field, developers are often left with limited information to diagnose the failure. Automated error reporting tools can assist in bug report generation but without precise steps from the end user it is often difficult for developers to recreate the failure. Advanced remote debugging tools aim to capture sufficient information from field executions to recreate failures in the lab but often have too much overhead to practically deploy. We present CHRONICLER, an approach to remote debugging that captures non-deterministic inputs to applications in a lightweight manner, assuring faithful reproduction of client executions. We evaluated CHRONICLER by creating a Java implementation, CHRONICLERJ, and then by using a set of benchmarks mimicking real world applications and workloads, showing its runtime overhead to be under 10% in most cases (worst case 86%), while an existing tool showed overhead over 100% in the same cases (worst case 2,322%).**

***Index Terms*—Debugging aids, Software maintenance, Error handling and recovery, Maintainability**

## I. Introduction

While software may behave properly under testing prior to deployment, it can be difficult to fully anticipate all possible usage scenarios and configurations in the field, where software is required to operate on different operating systems and in conjunction with various external systems. Reproducing field failures in the lab can be difficult — especially in the case of software that behaves non-deterministically, relies on remote resources, or has complex reproduction steps. Even when end-users file bug reports, it can be difficult to coerce users to provide detailed enough steps to reproduce the failure [6] (indeed, they may not even know how to reproduce it). To bridge the information gap, remote debugging tools aim to automatically capture information from the failing code and transmit it to developers.

A typical approach to remote debugging captures the state of the system just before a bug is encountered [3], [28]. However, unless such a system knows in advance that a bug is about to be encountered, it is impossible to provide developers with the exact state of the system *before* the bug is encountered, unless that state is constantly logged in anticipation of a defect. This approach tends to produce high overheads (reaching 2,000%+ overhead) in the deployed application [3], which may make it unacceptable for many uses. Novel solutions that lower this overhead typically limit the depth of information recorded (e.g. to use only a stack trace, rather than a complete state history) [27] or the breadth of information recorded (e.g. to only record information on a particular subsystem that a developer identifies as potentially buggy) [28]. While these approaches reduce overhead significantly (indeed, in many cases [27] sees 0%), there remain cases wherein restricting the breadth of recording reduces the effectiveness of bug reproduction.

Specifically, limiting the depth of information gathered may fail to reproduce an error if the defect does not present itself immediately. Imagine a program that reports its stack trace (along with each parameter for those methods) upon encountering a bug and contains (among others) methods $A$ and $Z$. Method $A$ sets a heap variable $V$, and method $Z$ reads it. The program calls method $A$, which sets $V$ to an invalid value and later on calls method $Z$, which reads the invalid value in $V$ and crashes. In this situation, a stack trace would show the invocation of $Z$ but not the invocation of $A$, as it occurred in another branch of the execution tree. Although symbolic analysis may be able to discover that original invocation of $A$ (as in [27]), this is not always feasible.

Similarly, by limiting logging to a specific subcomponent of an application, it is only possible to reproduce the bug if it occurred within that subcomponent. This technique requires that developers know a priori which sections of code will be likely to crash and if they select too many the performance of the system degenerates to the case where everything is logged. If too large of a subsystem is selected then the performance benefit shrinks in the event that all subcomponents are executing on the same CPU.

When filtering logging information in either of the two ways described above fails, the typical solution is to resort to a heavier recording — capturing a trace, for instance — leading to runtime overhead increases. In this paper, we present CHRONICLER: a technique that supports remote debugging by capturing program execution in a manner that allows for accurate replay in the lab, with low overhead. In addition to simple stand-alone applications, CHRONICLER supports accurate and efficient record and replay of client-server applications, where developers may replay server interactions on clients without requiring the server to participate in the replay process.

CHRONICLER only logs sources of non-determinism at the library level — allowing for a light recording process while still supporting a complete replay for debugging purposes. When a failure occurs, CHRONICLER generates a test case that consists of the inputs (e.g. file or network I/O, user inputs, random numbers, etc.) that caused the system to fail. This general approach is diagramed in Figure 1 and can be applied to any language that runs in a VM (for instance, Java or Microsoft's .NET CLR), requiring no modifications to that VM. We demonstrate the feasibility of CHRONICLER by implementing it in Java, and found that the overhead for real world applications was minimal (<5% in the case of Eclipse performance tests, <10% for Tomcat).

The main contributions of this paper are:

- A presentation of our remote-debugging model for accurate bug reproduction: CHRONICLER
- CHRONICLERJ, an implementation of CHRONICLER for Java, available for download and use now on github [4]
- A thorough evaluation of its performance demonstrating its low overhead on real world applications

The record and replay technique used by CHRONICLER can be applied to several other research areas to improve performance concerns that have been holding back greater progress:

- Test suite generation — Existing tools impose high overhead but could be run offline on captured executions [26]
- Efficient checkpoint and restart for VM based languages — existing tools [1] are not suited to VM based languages
- Benchmark generation — Existing benchmarks are hand written, but can be automated with record-replay frameworks [39].

The rest of the paper is organized as follows. In Section II, we discuss related work in the field of record and replay systems, error reporting and test case generation. We elaborate on the CHRONICLER approach in Section III and present implementation details in Section IV. Our empirical evaluation of CHRONICLER and a comparison with another Java-based bug-reproduction tool is presented in Section V. In Section VI, we discuss some of the limitations of CHRONICLER. Finally, we conclude and outline some ideas for future work.

## II. RELATED WORK

There are several widely used systems for collecting runtime information to diagnose failures. Microsoft's Windows Error Reporting tool has been in use since 1999 and has collected billions of error reports since then [22]. This tool collects system information after the point of crash such as register contents, thread stacks, hardware specifications and with the user's permission, transfers it back to the vendor for analysis. Apple's iPhone OS error reporter [2] and Firefox's Breakpad [19] are similar, reporting system state after a crash. While these systems have minimal runtime overhead (they are dormant until after an error occurs), their reports do not contain steps to reproduce the crash, nor a test case.

More recently, tools have been developed that leverage similar information to guide symbolic execution or static analysis to reproduce field errors [11], [15], [27], [48]. ESD uses bug reports [48], while Crameri et al's approach uses a partial recording of branch traces to generate test cases [15]. BBR similarly captures branch traces, but can create partial replays to better support long-running applications [11]. BugRedux [27] can use four different kinds of execution data: points of failure, call stack at failure, call sequences and complete program traces. ESD and BugRedux (when operating in call stack or point of failure mode) capture no information until failure, and therefore add no runtime overhead. However, these techniques are imperfect and can not always reproduce the failure, hence BugRedux also supports more detailed logging methods, which can produce higher overhead.

ReCrashJ [3] is a Java-based tool that automatically generates test cases when software crashes by tracking method arguments for the entire call stack. The system is limited in performance, showing overhead as high as 100,000%, 60%, or 42% (depending on the logging method used, presented here in descending order of soundness).

Scarpe [28] is a bug reproduction tool that targets Java and requires developers to annotate their application to show component boundaries, capturing interactions between the classes of interest and external code. This approach can be quite efficient when the component selected for logging has limited external interaction, but in other cases the overhead is as high as 877%. In contrast to Scarpe which captures inter-component interaction within an application, CHRONICLER records interactions between the application of interest and its environment.

At their core, techniques such as CHRONICLER, ReCrashJ and Scarpe are essentially built on record and replay systems. Record and replay systems capture program executions and deterministically replay them. Some of the earliest such systems were machine-wide, aimed to debug operating systems [18], [33], [40], [46]. Unlike CHRONICLER, these systems capture everything running on the machine (rather than within a specific program) and are invasive, requiring custom hardware, a modified operating system, or a specialized virtual machine.

Liblog [20], ADDA [13], Mugshot [31], and R2 [23] are four application-level record and replay systems with the same underlying principle as CHRONICLER— logging non-determinism. Liblog is a tool for C applications to record and replay all interactions between the application and the operating system at the system call level. ADDA works at the libc level and is optimized to generate shorter replays that still produce the same failures. However, these approaches are insufficient to capture all sources of non-determinism in C programs, which can interact with the outside system through mechanisms such as shared memory or asynchronous intercepts, and therefore can not guarantee complete replay (concerns that do not arise in a VM). Mugshot [31] is a record and replay system targeting JavaScript applications with the same underlying principle as liblog and ADDA, designed to function in the limited execution model of browser-based applications (based on non-preemptive callbacks). This model does not apply to languages such as Java which have a rich execution model, full multithreading support and many more sources of non-determinism than Javascript. R2 requires developers to manually annotate their application to indicate how each function should be logged. CHRONICLER's approach

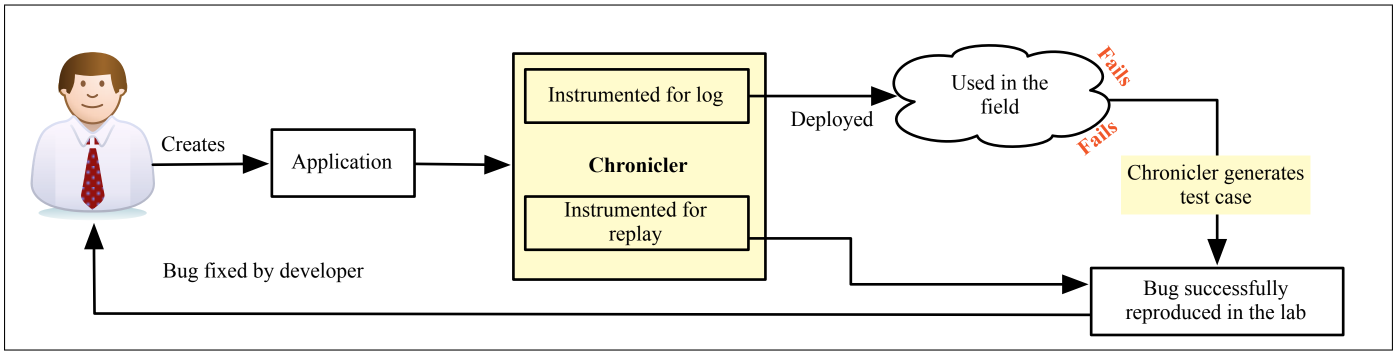

**Fig. 1:** High Level Overview of CHRONICLER

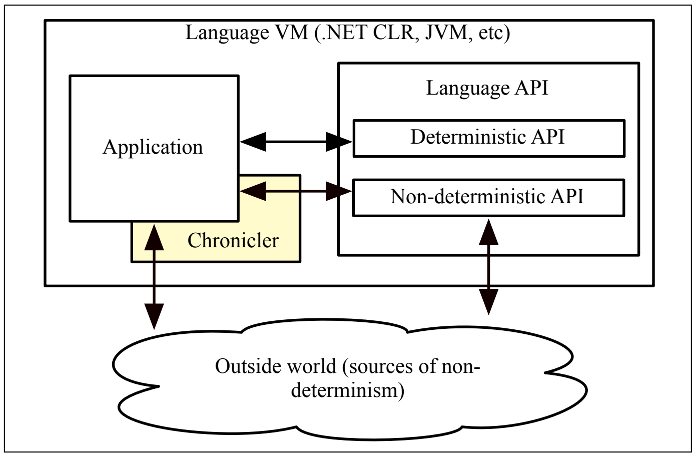
Language VM (.NET CLR, JVM, etc)
Application
Language API
Deterministic API
Non-deterministic API
Chronicler
Outside world (sources of non-determinism)

**Fig. 2:** Logging Non-determinism with CHRONICLER

is fundamentally very similar to these systems in that all three systems log non-determinism, although key to our approach is the level at which we intercept the non-determinism, which differs from these systems and allows us to be more performant.

Although several record-replay systems have been described in the literature, only a few target VM-based languages. DeJaVu [12] was one of the earliest JVM based record and replay systems, but required invasive changes to the JVM itself. jRapture [41] is a Java record and replay system designed to be used for profiling executions after they have been captured. jRapture uses an overall approach similar to CHRONICLER but requires modifications to the core JRE API libraries, which complicate its widespread distribution. Preliminary performance testing showed jRapture to have overheads ranging from 0.80-10,000% depending on the relative proportion of I/O in the application being logged [41].

While test case generation tools (e.g. [16], [35], [37], [51]) generate test cases to increase test suite code coverage offline, CHRONICLER generates test cases that specifically reproduce field failures. Although these tools are run statically, CHRONICLER could be used in conjunction with test case generation tools that are guided by an execution, such as [26].

## III. APPROACH

The CHRONICLER approach relies on a simple principle: if a bug occurs deterministically, then reproducing it in the lab can be made trivial — the developer need only run the program, and the bug will present itself. Unfortunately, software often fails to behave completely deterministically, with inputs provided by outside systems (via network, file or console I/O, shared memory access, etc), from random numbers, from system properties, such as the current time or machine configuration, or from thread interleaving. Hence CHRONICLER records sources of non-determinism in a program and replays them to reproduce the bug.

Figure 2 shows the overall approach to logging non-determinism with CHRONICLER. CHRONICLER is designed to function in any VM-style programming language, where interaction outside of the VM is restricted to a finite set of methods. CHRONICLER runs completely within the VM, and sits between the application and all sources of non-determinism, logging them as they enter the application code. Note that although thread interleavings present non-determinism internal to the VM, they are not logged. This limitation is addressed further in Section VI, but does not prevent CHRONICLER from replaying non-race bugs.

Unlike systems like liblog [20] that record non-determinism at the granularity of system calls, CHRONICLER records non-determinism at the granularity of the methods provided by the VM. This distinction means that there will be a wider selection of methods that need to be logged, as VM-based languages (such as Java or .NET) typically provide a common library or API of utility functions. For instance, in order to read data from a file, a programmer may have at their disposal methods to read by line, by word, or in a binary format into a byte array. The liblog approach would record the underlying call from all of these methods that actually reads data from the file. On the other hand, CHRONICLER will record the invocation of each of the language utility methods (such as to read a line), rather than the native routine itself.

Our key insight of logging at the API level removes the need to modify any language-provided libraries and results in increased performance for CHRONICLER. Returning to our example of reading data from a file, imagine an implementation of the “readLine” method provided by the language that reads $N$ bytes from a file into a buffer until it reaches a newline character. Rather than log the buffer every time that the underlying "read" method is called, CHRONICLER simply logs the line that is eventually returned. Of course, application code can also directly call native methods (without utilizing the language-provided API), and these calls are logged as well.

CHRONICLER automatically scans the language API to identify all potentially non-deterministic method calls and then instruments the application code to log the result of each such call, all at the byte code level. CHRONICLER logs a unique, reproducible identifier for each thread to denote which log entry should be replayed in which thread. The log is buffered in memory and flushed to disk as the log size increases, or when a failure is detected. CHRONICLER similarly instruments the application code to create a "replay" version, which replaces non-deterministic method calls with instructions to replay from the log file, to be used to reproduce failures.

We create a special case to handle event-driven systems, where the event dispatcher is part of the native code (e.g. Swing in Java). In these cases, non-deterministic input may drive the language API to fire events to listeners in client application code, but the application never directly reads that input. To reproduce these events, we log each invocation of these listener methods, so that we can fire them in the same ordering with the rest of our log. By logging all sources of non-determinism, we can then reproduce the same execution, and hence, the same failure. Moreover, this approach will reproduce failures even if they are dependent on external services that are unavailable for replay.

CHRONICLER instruments the application to generate a test case and log file to transmit to developers upon encountering an error. Test case generation can be triggered simply by an exception being thrown or through external bug detection techniques, supporting bugs that do not throw exceptions. With test case in hand, reproducing the bug is simple: developers must only run the "replay" instrumented application with the log file as input. Execution begins at the same entry point as the original failed execution and all inputs are reproduced from the log, reproducing the same program execution. With this technique we allow developers to observe the entire execution and use existing automated debugging tools that they may already be comfortable with.

Note that throughout the entire CHRONICLER approach, no source code is necessary, and all instrumentation can be performed directly on bytecode. In order to evaluate the performance of this approach we implemented CHRONICLER for Java and the JVM, although the approach is general enough to apply to other languages within the JVM (e.g. Scala [36]) or other VMs (e.g. .NET).

## IV. IMPLEMENTATION

To elaborate on the CHRONICLER approach, we describe CHRONICLERJ, our Java implementation of CHRONICLER. Figure 3 shows an overview of CHRONICLERJ. More technical details about it are available in the code itself, available on github [4]. Its implementation can be broken into the following four core components:

### *A. Detecting Non-deterministic Methods in the JVM*

Our approach requires instrumenting the call site of every method in client code that returns non-deterministic input (e.g. I/O from the user, files, or network, random numbers, etc). As we noted previously, within the JVM the only way that code can receive non-deterministic input is if it makes a call that executes native (non-Java) code. Facilities for generating random numbers, accessing system properties (such as the current time, IP address, hostname, etc), or interacting with files and sockets are all implemented in native code.

Therefore the first step to identifying non-deterministic methods in the Java API is to scan the entire API and mark all methods that are "native" as non-deterministic. However, not all native methods are non-deterministic. For instance, the typical approach to copy the contents of an array is to use a native call System.arraycopy. While an array-copy could be implemented in Java, the native approach is more efficient as it directly copies the contents of the entire array (stored contiguously in memory) rather than copying entry-by-entry. This native method (and many others) implement basic tasks deterministically and efficiently. We manually constructed a stop list of methods which are native but deterministic, ensuring that the "default" classification for a native method was non-deterministic, perhaps sacrificing performance for correctness, to avoid the risk of an incomplete log. This stop list is extensible, and for performance reasons can be tuned on a per-application basis.

The next step in identifying all non-deterministic methods in the Java API is to identify all API methods that call the previously identified non-deterministic methods. This process scans the API for all callers of non-deterministic methods and recursively marks those methods as well as their callers within the API as non-deterministic. CHRONICLERJ also carries non-deterministic flags up the inheritance hierarchy — for instance the interface method InputStream.read(byte[], int, int) will be marked as non-deterministic, since many of its implementers are — and marks any Java library methods that call these methods as non-deterministic as well.

At this point, we have identified all API methods that call a method which behaves non-deterministically, or returns a non-deterministic result. The final step is to identify methods which can behave non-deterministically because they share state with a non-deterministic method. CHRONICLERJ performs a very simple analysis to determine these methods, marking all fields set by a non-deterministic method as tainted, and then marking all methods that read those fields as non-deterministic. Similarly, all owners of methods called by a non-deterministic method are marked as non-deterministic. A more advanced control and data flow analysis could limit the number of methods falsely flagged as non-deterministic, but we found the performance of this technique to be adequate (a performance evaluation of CHRONICLERJ appears in Section V-A).

Finally, CHRONICLERJ checks all classes in the application of interest (as well as included libraries) to build a list of any methods that directly invoke native code. With this approach

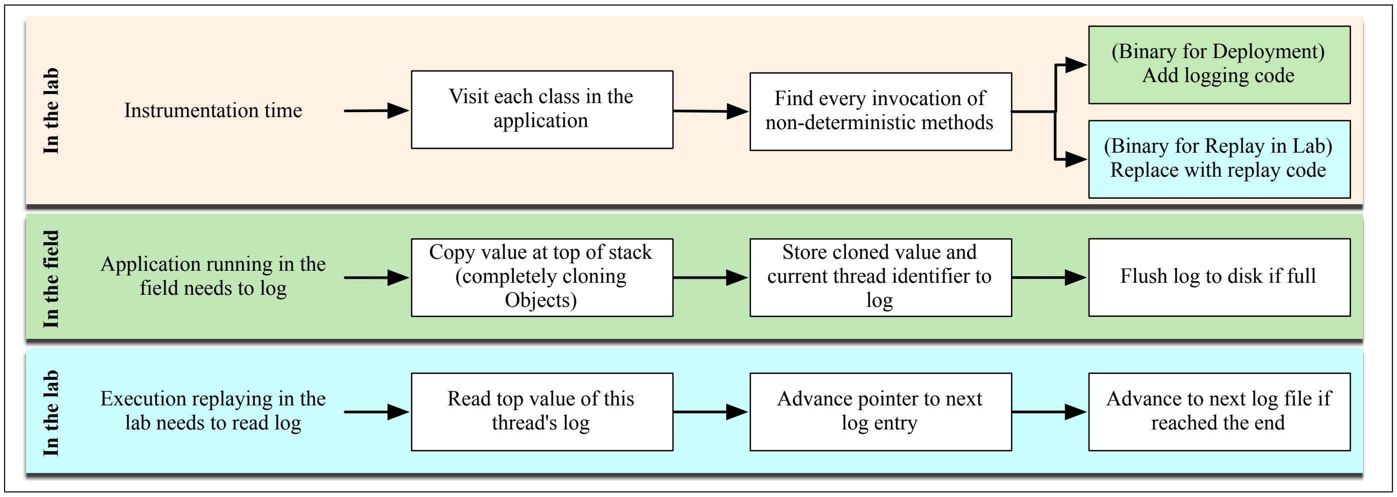
In the lab
Instrumentation time
Visit each class in the application
Find every invocation of non-deterministic methods
(Binary for Deployment) Add logging code
(Binary for Replay in Lab) Replace with replay code
In the field
Application running in the field needs to log
Copy value at top of stack (completely cloning Objects)
Store cloned value and current thread identifier to log
Flush log to disk if full
In the lab
Execution replaying in the lab needs to read log
Read top value of this thread's log
Advance pointer to next log entry
Advance to next log file if reached the end

**Fig. 3:** CHRONICLERJ Implementation Overview

| Number of Methods By Return Type | Immutable | Void | Mutable | Total |
|---|---|---|---|---|
| Native | 1,571 | 1,571 | 2,803 | 4,374 |
| Less stop list | 1,519 | 1,553 | 1,228 | 4,300 |
| Plus upcast | 1,608 | 1,590 | 1,235 | 4,433 |
| Plus all callers | 16,966 | 40,635 | 24,467 | 82,068 |

TABLE I
STATISTICS ON NONDETERMINISTIC METHODS

we guarantee detection of all methods that behave non-deterministically. In this way, CHRONICLERJ builds a list of approximately 100,000 methods (on JRE 1.7.0_05 running on Mac OS 10.8.0) that must be logged when called by the client code. This entire process is integrated into CHRONICLERJ, so that it can be re-run by developers for new releases of Java, but we do not expect this to be a common task.

Detailed statistics regarding the number of nondeterministic methods identified, broken down by return type and status are shown in Table I.

## B. Logging

CHRONICLERJ instruments all calls to the identified non-deterministic methods to record return values (and buffer(s), if applicable). All byte code instrumentation is performed using the ASM byte code framework [9]. The log is buffered in memory and written to disk at regular intervals (flushing the log to disk is described further in the following section). The log buffer can have a hard size limit, or optionally expand as necessary until it is flushed manually. CHRONICLERJ is thread-safe, and protects each log call with a barrier so that no two threads can log at the same time (there is only one log). This may result in increased thread contention, however, the critical region contains only the generated code to log the return value, and does not contain the actual computation that is being logged.

Logging code is embedded inline, just after the value that we need to log is pushed onto the stack. This technique captures all method invocations — both direct and implicit invocations (e.g. super methods). The instrumentation copies the object, grows the log if necessary, writes the object to the log, writes the thread id to the log and flushes it if necessary.

Rather than directly invoking methods that return a non-deterministic value, some applications may instead register callback methods that are themselves invoked non-deterministically (e.g. Swing Action Listeners). To support these sorts of callbacks, we identify all listener methods in the application code that may be invoked non-deterministically and record their invocations so that when we replaying we can fire the invocations at the appropriate, logical time. To efficiently support the manual invocation of these listeners (where we would not want to log their invocation explicitly), CHRONICLER rewrites them to only record their invocation when invoked through the listener interface.

When writing values to the log, immutable types (such as language primitives — Integer, Long, Short, Float, Double, Byte, Character and Boolean — and other classes such as String) are simply stored as pointer references. Since the values are immutable, we can be assured that during execution the log contents will not be inadvertently changed. Mutable types (such as arrays, or mutable classes) however must be fully copied, so as to ensure that the logged version represents the value at log time, and isn't modified by the process.

To efficiently copy arrays that contain immutable types, we created an inline fast cloner, observing that if the values in an array are immutable, the only way to change the array's contents is to assign new values to its indices. Our fast array cloner directly allocates a new array of the appropriate size, and uses the native, JVM-provided System.arrayCopy method to copy the array contents (which may be primitive values or object references). The remaining cases (mutable objects that must be cloned) are cloned using a runtime reflective cloning library [29] that copies all fields on an object, recursing through those fields to copy them as well. This can become time intensive — for an object $O$ that has $n$ fields, it is necessary to first allocate a new object $O'$, and then for each $n$ fields, copy each field, recursing to copy that field's objects, and so on. However, it is necessary to undertake this process to ensure a faithful reproduction.

We hold a special case for logging constructors since in Java a constructor has no return value. In most cases however, a reference to the newly constructed object is left on the stack after the invocation of the constructor — so that it can be stored in a field, or used in any way. However, a statement such as “new Object();” will not result in a reference to the object left on the stack — since only the constructor is called while the the object itself is unassigned. Therefore CHRONICLERJ tracks the state of the stack while instrumenting, and only generates log instructions after a non-deterministic constructor if the newly generated object is used.

We also hold a special case for logging events fired by Java itself. For each listener, at instantiation we create a unique ID based on the order in which it was created and the thread that created it. Then when the listener receives an event, we log the ID of the listener and the event that was fired.

Reflection is supported via a wrapper that checks any dynamically invoked methods at runtime to determine if they need to be logged (a technique similar to that used in [28]). CHRONICLERJ must carefully record non-deterministic calls from overridden Java *finalize* methods. The finalize method is called non-deterministically as the garbage collector destructs objects, so we assume that they may be called out of temporal order during replay. To reproduce events for these methods we associate each logged value with an ID for the object being destructed (based on a logical clock) and use this ID to reproduce them in the correct logical order.

At the same time that logging instrumentation is performed, a “replay” version of the application is created, which replaces non-deterministic calls with instructions to load the appropriate log value. This process leaves instructions that evaluate any argument expressions to these methods (which themselves may have side effects), to ensure a faithful reproduction. The replay application also similarly wraps reflective calls with a dynamic check to determine if they should be replayed or executed as usual.

### C. Flushing the log

By default, CHRONICLERJ flushes the log from memory to disk after 500,000 entries are stored in the log, using the number of entries as a heuristic for the total size (in memory) of the log. While it is possible to more accurately count the size of the log, doing so would add a performance overhead that we did not wish to incur. The flush interval is configurable, and can be disabled altogether, so that the developer can directly invoke the flushing mechanism. This can be particularly useful to ensure that the log flush occurs during a period that the system is not processing many events. The log is also automatically flushed when an uncaught exception occurs. Flushing occurs in a background thread and program execution can continue during the flushing process (it does not block logging).

CHRONICLERJ uses a shadow log during flushing, which allows new events to be logged to the primary log, while CHRONICLERJ flushes the shadow log. This allows for the critical region in the flushing process to be relatively small as only the creation of the shadow log and truncation of the primary log must be protected. The log is split into two parts: a log for Serializable types (such as primitives, primitive arrays, Strings and other Serializable classes), and a log for non-serializable classes. The log of Serializable types is flushed using Java’s built-in serialization mechanism, while the non-serializable log is exported to XML using the XStream library [45]. This technique takes advantage of the speed of Java’s serialization mechanism whenever possible.

### D. Test Case Generation

When an uncaught exception is encountered (or when the mechanism is manually invoked by including the CHRONICLERJ library and calling the static function *ChroniclerExportRunner.generateTestCase()*), CHRONICLERJ creates a test case that invokes the application with the same starting parameters and uses any necessary log files for input, executing the identical set of actions that caused the system to fail originally.

The generated test case contains all necessary log files and loads them sequentially as necessary, tracking the replay progress through each individual log. Each thread maintains its own position in the log, and CHRONICLERJ ensures that each thread receives the logged values for that thread, in the order that they were logged. Again, CHRONICLERJ takes special care to ensure that replay within finalizers occurs correctly, even if finalizers are called in different orders.

In our evaluation that follows we show that the logging overhead of CHRONICLERJ is reasonable for a real-world benchmark suite. Our figures indicate that CHRONICLERJ is lightweight compared to ReCrashJ, the only comparable system for Java that we were able to obtain to compare to directly (we had considered many of the tools discussed in Section II but were limited by availability of other Java-based tools).

## V. EMPIRICAL EVALUATION

We evaluated CHRONICLERJ in two dimensions: its performance in the field when capturing executions and its ability to reproduce failures, leading to the following evaluation metrics:

**EM1:** Performance overhead: Is the runtime overhead of CHRONICLERJ’s logging suitable to be deployed with production applications in the field?

**EM2:** Functionality: Does CHRONICLERJ reliably reproduce failures?

### A. EM1 - Performance Overhead

We used the DaCapo v9.12-bach suite of benchmarks [7], a set of Java benchmarks that focus on exercising applications in real-world conditions, to evaluate CHRONICLERJ’s performance for **EM1**. We also evaluated CHRONICLERJ’s performance overhead on the same set of benchmarks used by the Java bug reproduction system ReCrashJ [3]. Finally, we bound the best and worst case overhead of CHRONICLERJ by constructing synthetic benchmarks that specifically target CHRONICLERJ’s strengths and weaknesses. We executed all benchmarks on a 2.7 Ghz iMac with 16GB of RAM,

| Benchmark | Description |
|---|---|
| avrora | Simulates programs running on a grid of AVR microcontrollers |
| batik | Executes unit tests for Apache Batik, an SVG toolkit, producing several images |
| eclipse | Executes non-gui performance tests for Eclipse |
| fop | Parses and formats an XSL-FO file into a PDF |
| h2 | Runs an in-memory database benchmark, running transactions against a theoretical banking application |
| jython | Interprets and runs the pybench Python benchmark using jython |
| luindex | Indexes Shakespeare and the King James Bible with Apache Lucene |
| lusearch | Searches for keywords over a corpus including Shakespeare and the King James Bible with Apache Lucene |
| pmd | Performs a static analysis on Java source files |
| sunflow | Renders images with ray tracing |
| tomcat | Creates a tomcat server and runs a simple sample servlet |
| tradebeans | Executes the DayTrader [42] benchmark via Java Beans on an Apache Geronimo server with an in memory database |
| tradesoap | Executes the DayTrader [42] benchmark via SOAP on an Apache Geronimo server with an in memory database |
| xalan | Transforms several XML files into HTML |

TABLE II
DESCRIPTION OF BENCHMARKS, FROM THE DACAPO WEBSITE [8]

| Benchmark | CHRONICLERJ Overhead | ReCrashJ Overhead | |
|---|---|---|---|
| | | This study | From [3] |
| Eclipsec Channel | 17.22% | 20.40% | 34.00% |
| Eclipsec Content | 19.89% | 21.67% | 13.00% |
| Eclipsec String | 18.58% | 20.91% | 27.00% |
| Eclipsec JLex | 10.84% | 36.69% | 42.00% |
| SVNKit Checkout | 67.59% | 168.50% | 38.00% |
| SVNKit Update | 71.82% | 97.40% | 11.00% |

TABLE III
RECRASHJ BENCHMARK RESULTS FOR CHRONICLERJ AND RECRASHJ

Java 1.7_05 with the heap size configured to 12Gb and Mac OS 10.8.0 in a clean-room environment. We used the default configuration for both ReCrashJ and CHRONICLERJ.

*1) DaCapo Benchmarks:* The DaCapo suite consists of fourteen non-trivial workloads exercising a variety of open source, widely used applications. The benchmarks are diverse and include a widely used application server ("Tomcat"), a full text search engine ("Lucene") as well as "Jython", a Python interpreter written in Java. Several of the benchmarks are implementations of well known and accepted workloads. For instance, the "h2" benchmark utilizes the TPC-C workload [44], a common database benchmarking workload, to test an in-memory database. The "tradebeans" and "tradesoap" benchmarks run Apache's DayTrader [42] benchmark workload, an open source version of IBM's Trade 6 workload [25]. A complete list of the individual benchmarks executed along with a brief description appears in Table II.

We executed all 14 benchmarks in the DaCapo suite 100 times each in both the original and the CHRONICLERJ-instrumented environments. We attempted to compare CHRONICLERJ with other bug reproduction systems on the same benchmark, but were limited in tool availability — the only Java-based bug reproduction system that we were able to download was RecrashJ [3], version 0.3, a research tool published in 2008. RecrashJ is a bug reproduction tool that records the entire JVM state at every method invocation.

Figure 4 shows the average time per benchmark of the DaCapo suite. ReCrashJ was incompatible with the eclipse and sunflow benchmarks (which use Java features unavailable at the time that RecrashJ was created), so we do not have results for ReCrashJ for those benchmarks.

*2) ReCrashJ Comparison:* To create a fair basis for comparison with ReCrashJ, we also benchmarked CHRONICLERJ's performance on the same systems that the ReCrashJ authors benchmarked in their paper [3]:

- Using SVNKit 0.8.0 (an SVN client implemented entirely in Java) [43], checkout and update the project "amock" from GoogleCode [21]
- Using the Eclipse 2.1 Java compiler (a compiler implemented entirely in Java), compile the JDK 1.7 sample files "Content," "String," and "Channel" as well as version 1.2.4 of the JLex project [5]

In Table III we show the run-time overhead for CHRONICLERJ and ReCrashJ on our test platform as well as the original results previously obtained by [3]. We attribute the differences in overhead between our experiment and [3] to the architectural differences between the test systems. The extreme differences in the SVNKit benchmark results (between our evaluated 168% and the reported 38%) are likely related to varying external conditions (the benchmark executes against a third-party SVN server).

*3) Targeted benchmarks:* Although the DaCapo benchmarks simulate real world workloads, we wanted to explore the effects of injecting CHRONICLERJ with specialized workloads. Specifically, we wanted to observe how CHRONICLERJ would interact with a purely computational workload (which would entail little or no instrumentation) and an I/O heavy workload (which would be almost entirely instrumented).

We selected SciMark 2.0 [38] as our computational benchmark. Some of the programs included in this benchmark are an implementation of the Fast Fourier Transform, Monte Carlo integration and LU decomposition. SciMark uses non-deterministic functions only to build the test data: there is no non-determinism within the benchmark itself. We executed the SciMark benchmark 100 times and observed that the overhead imposed by CHRONICLERJ on purely computational workloads is insignificant ($< 1\%$).

In order to characterize CHRONICLERJ's worst case performance, we ran it with a program that did nothing but read files ranging in size from 2MB to 3GB. These files were generated from random binary data, and contain no linebreaks. The benchmark program uses the *readLine* method of *java.io.BufferedReader* to read the file into a string. We executed this process 100 times on our test machine and measured the average overhead. As shown in Figure 6, as we increased the file size, the overhead evened out at 86% as the file size grew, providing what we feel is a reasonable upper bound for the worst-case performance of CHRONICLERJ.

*4) Discussion:* As expected, CHRONICLERJ shows minimal overhead in benchmarks that contain low amounts of I/O, while overhead grows in I/O heavy situations. In all but one case (batik) CHRONICLERJ outperformed ReCrashJ, often by

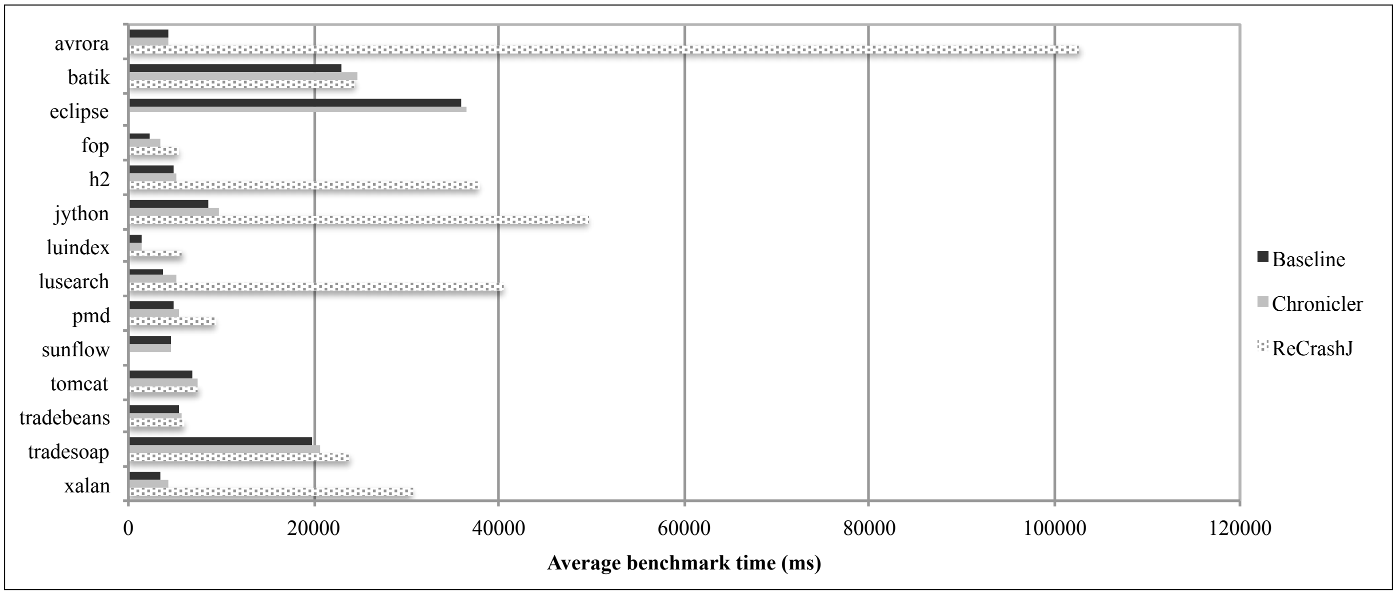

**Fig. 4:** DaCapo Benchmark Results for a baseline execution, CHRONICLERJ and ReCrashJ

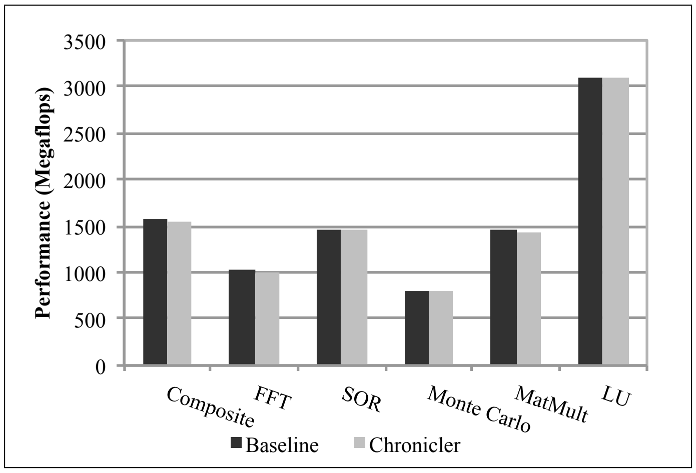

**Fig. 5:** Scimark Benchmark Performance for CHRONICLERJ

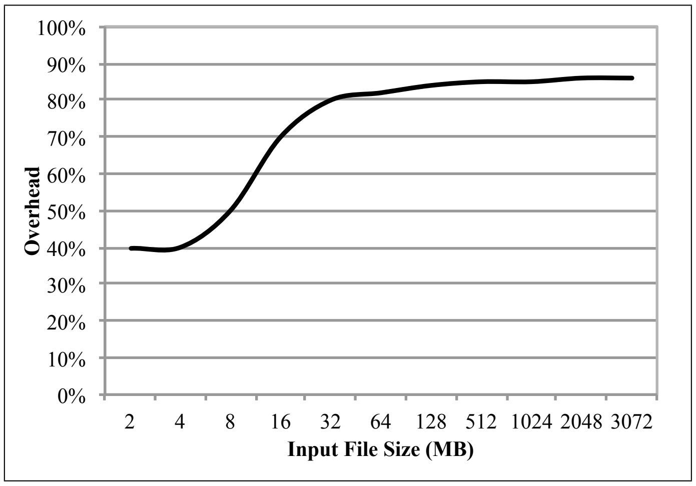

**Fig. 6:** I/O Benchmark Performance for CHRONICLERJ

large margins. In the case of batik, we believe that ReCrashJ performed slightly better due to a favorable environment (e.g. low stack depths), while the I/O component (reading and writing images) slowed CHRONICLERJ. CHRONICLERJ demonstrated a relative stability in performance (fluctuating from 1.56% to 39.96%) across the DaCapo suite as compared to ReCrashJ (fluctuating from 5.97% to 2,321.94%).

Interestingly, in the RecrashJ benchmarks, although CHRONICLERJ outperformed RecrashJ, the differences were not as severe as in the DaCapo benchmarks. We attribute our closer performance to the relatively shallow call stacks of the test suite used (which benefits RecrashJ) as well as the high-amount of file I/O (which hinders CHRONICLERJ).

We showed that even in the worst case, CHRONICLERJ maintains an 86% overhead, while its best case performance is under 1%. CHRONICLERJ's overhead exceeded 10% in only benchmarks that performed large amounts of input or output operations: fop, jython, lusearch, pmd, xalan, SVNKit and several Eclipsec instances. Based on our CHRONICLERJ evaluation, we conclude that the performance of the CHRONICLER approach is suitable for applications that are not dominated by file access, and even in those that contain large amounts of file access, remains consistently bounded.

### B. EM2 - Functionality

In order to evaluate the ability of CHRONICLER to successfully replay an execution we considered eleven real bugs in the following applications and libraries:

- *Jetty*: A widely used application server for Java. We considered a bug that caused an uncaught exception when the HTTP string parser was given a specific input [1]. The exception thrown was an EOFException which was caused because of a lack of infrastructure in the

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=363993

HttpTester to expect a body in an HTTP response. The full stacktrace that was thrown is

```
java.io.EOFException
        at org.eclipse.jetty.http.HttpParser.parseNext(HttpParser.java
            :309)
        at org.eclipse.jetty.http.HttpParser.parse(HttpParser.java:204)
        at org.eclipse.jetty.testing.HttpTester.parse(HttpTester.java
            :139)
```

The HttpParser had a flag that allowed it to expect a HEAD response but the HttpTester had no such flag causing the eof exception..

- *Apache Commons-Math*: A stateless library consisting of implementations of mathematical functions. We consider four bugs that resulted in uncaught exceptions and incorrect results [2]
  The first bug (801) we considered involved operations on quaternions. Normally, post construction, quaternions should be normalized but with certain input it wasn't.
  Specifically, using the constructor Vector3D-Vector3D-Vector3D-Vector3D- Rotation with a normalized angle lead to non-normalized quaternion. This case appeared to us with the following data :

```
u1 = (0.9999988431610581, −0.0015210774290851095, 0.0)
u2 = (0.0, 0.0, 1.0)
v1 = (0.9999999999999999, 0.0, 0.0)
v2 = (0.0, 0.0, −1.0)
```

  This lead to the following quaternion : (225783.35177064248, 0.0, 0.0, -3.3684446110762543E-9)
  which is obviously not normalized.
  The second bug (790) that we considered resulted in overflow on large data sets on the Mean Whitney-U Test. These data sets were large arrays of doubles, with sizes around 1500 or more. The reason this was occurring was that the underlying implementation of the test used an integer in a place where a double should have been used.
  The third bug (803) caused the functions ebeMultiply: RealVector - OpenMapRealVector and ebeDivide: RealVector - OpenMapRealVector to return wrong values when one of the entries in the parameters passed to these functions contains nans or infinity. The crux of the bug is an invalid assumption that for any double x, x * 0d = 0d is always true which is not the case with nan and infinity.
  The fourth bug (645) resulted in a run time exception when calling ebeMultiply. This was caused because the underlying implementation of the method was iterating on the copy of the RealVector passed to ebeMultiply which was simultaneously getting modified as well. Instead, the iterating should have been done on the *original* copy of the RealVector which is unchanging.
- *Apache Commons-Lang*: A stateless library that provide helper utilities for the java.lang API. We consider two bugs that resulted in exceptions[3].
  The first bug (72) resulted in an uncaught NPE while calling EqualsBuilder.append(Object[], Object[]). This NPE was caused by null-value elements in the first object array.
  The second bug (300) resulted in NumberFormatException when valid strings were passed to the NumberUtils.createNumber method.
- *Groovy*: A JVM based dynamic language. We consider four bugs that lead to program crashes[4].
  The first bug (5649) resulted in a StackOverflowError when accessors were annotated with @CompileStatic.
  For instance, the following code results in the above error

```
class HaveOption {

  private String helpOption;

  @CompileStatic
  public void setHelpOption(String helpOption) {
    this.helpOption = helpOption
  }
}
```

  The second bug (3914) resulted in a java.lang.LinkageError with the following code

```
import groovy.xml.DOMBuilder
def filePath = ``MestaXml.log'';
def doc = DOMBuilder.parse(new FileReader(filePath));
def docElm = doc.documentElement;
```

  The third bug (2503) results in an exception when a closure accesses a private method. The following code highlights the bug.

```
package test
public class Parent{
  private String parentMethodB(){
    return ``parentMethodB'';
    }
  protected String parentMethodC(){
    def closure={
      return parentMethodB()}
    return closure()
    }
}
```

```
package test
public class Child extends Parent{
  public static void main(def args){
    Child c = new Child();
    println(c.parentMethodC())
  }
}
```

  When this code is run we get the following MissingMethodException.

```
Exception in thread ``main'' groovy.lang.MissingMethodException: No
      signature of method: test.Child.parentMethodB() is applicable for
      argument types: () values: {}
```

  The fourth bug (2256) results in an exception when validating arguments passed to the mocked method using Groovy mocks.
  The following script reproduces the bug.

```
import groovy.mock.interceptor.StubFor

def class SomeClass {
    def methodOne(int age) {
        return age * 12 * 30 * 24
    }
}

def someStub = new StubFor(SomeClass)

someStub.demand.methodOne {
  number -> assert number > 0
  return 1
}

someStub.use {
  assert new SomeClass().methodOne(2) == 1
}
```

[2]https://issues.apache.org/jira/browse/MATH-{645,790,801,803}
[3]https://issues.apache.org/jira/browse/LANG-{72,300}
[4]http://jira.codehaus.org/browse/GROOVY-{2256,2503,3914,5649}

CHRONICLERJ was able to faithfully reproduce the executions, in each case. All programs terminated with the same uncaught exception that CHRONICLERJ had captured earlier.

## VI. THREATS TO VALIDITY AND LIMITATIONS

We performed our experiments towards evaluating **EM1** on the DaCapo benchmarks, which we believe are representative of a diverse set of real-world loads. However, it is both possible and likely that there exist applications with workloads that are not represented by the benchmarks. To provide insight into performance for other workloads, the targeted benchmarks can be used to gauge best case and worst case overheads based on the amount of logging necessary relative to the overall application. Although we are confident that our worst-case benchmark truly stresses CHRONICLERJ to its limit, it remains possible that there is some other use case that would stress it further. Unfortunately, we were only able to directly compare CHRONICLERJ to RecrashJ. However, other tools (notably [27]) are likely to show equal or better performance in this metric — but could not be executed on the same benchmark as they do not target Java.

For **EM2**, the key threat to validity is the sample size and selection. We were only able to evaluate eleven failures, given the time-consuming process of finding real bugs that exercise CHRONICLERJ's capabilities, downloading and compiling that older version of software, and reproducing the bug — although we have reported in this paper every failure that we encountered and attempted to reproduce. Although we did not formally evaluate the efficacy of CHRONICLERJ on GUI based applications, our anecdotal experiences show it to be successful in reproducing GUI-based executions. We believe that given the approach we have taken, which we are confident is capable of reproducing (non-race) bugs, our sample size for EM2 is not a great concern. We are currently pursuing feedback from developers regarding the usability of CHRONICLERJ (based on real world scenarios), available on github [4], and indeed have already received useful input.

During our study for EM2, we did not encounter any instances of the replay taking noticeably longer to execute than the original version. However, we did not directly study the speed of replay using CHRONICLERJ, and it would be interesting to study this to identify potential improvements.

There are several known limitations to our approach and implementation. First, CHRONICLER does not log thread interleavings, and therefore does not necessarily reproduce races. However, LEAP is a promising Java-based system for reproducing racy executions which displays a reasonable overhead, around 10% on Tomcat and Derby (but up to 600% in the worst case, depending on thread accesses), and we could potentially combine CHRONICLERJ with LEAP to record thread interleavings (at a higher overhead). Additionally, traditional race-detection techniques (e.g. [32]) could be used within the lab on replayed executions to detect possible races without adding additional overhead to the field execution.

The second key limitation to CHRONICLER is end-user privacy. While the thoroughness of CHRONICLER's input logging ensures that executions observed in the field are reproduced accurately in the lab, this approach may leak sensitive end user information to developers. This is a typical problem in remote-debugging systems, and several approaches have been developed specifically to protect the privacy of inputs recorded in the field that could be combined with CHRONICLER (none of these systems are record and replay systems themselves). One approach towards solving this problem is input minimization [47], [50], where input that is non-essential to replicate the program failure is removed. However, there is no guarantee that the minimized input does not contain any sensitive data. Camouflage [14] addresses this issue by mutating a failure inducing input so that although the original and mutated version share no sensitive data, the program execution paths that they create are identical. Castro, et al [10] used symbolic execution in conjunction with record replay techniques in order to anonymize sensitive data present in bug reports. Other systems (such as [27]) address end-user-privacy by collecting execution data (e.g. stack traces), rather than user data. We have not yet integrated any of these approaches with CHRONICLER, and leave this for future work, which we expect will be key to an adoption of CHRONICLER.

All record and replay systems, including CHRONICLER, generate logs that grow over time. CHRONICLER's log grows in proportion to user input, so for systems that operate on minimal input, the log will not grow significantly and can be efficiently flushed to disk between user interactions. In other cases, we may be able to combine novel approaches to reducing the log size [11], [30] along with compression techniques to decrease log sizes.

Detecting and logging all non-deterministic methods is key to the CHRONICLER approach — which can only be applied to languages that operate in a virtual machine. One potential failure point is if the set of non-deterministic methods varied between individual VMs. However, this is only possible in the unlikely event that there is a bug in the VM itself. Similarly, defects in the underlying operating system or computer hardware may not be reproduced.

Additionally, it is possible to circumvent CHRONICLER's logging by creating a native method that non-deterministically mutates its parameters (since CHRONICLER does not generally log parameters passed to non-deterministic methods, only return values and buffer parameters). However, this technique is discouraged and rarely used (in the case of Java), as it is inefficient to access object parameters in native code [17]. For our CHRONICLERJ implementation, we ensured that no Java library methods behave in this manner by surveying all of the native methods in the JRE, and found only 37 that accepted a mutable object as a parameter, none of which were actually modified. This process would need to be performed for implementations of CHRONICLER for other languages to make sure that this assumption holds.

## VII. CONCLUSION AND FUTURE WORK

Reproducing bugs encountered in the field is a difficult task faced by developers. In this paper we presented CHRONICLER, a record and replay technique that can faithfully reproduce bugs even in non-deterministic conditions. We presented the approach used by CHRONICLER for bug reproduction: logging non-deterministic inputs at a layer *above* the language API and evaluated our approach with CHRONICLERJ by simulating real world workloads, showing a worst-case overhead of 86%, with average-case performance significantly lower. We demonstrated that CHRONICLER can be used to reproduce bugs in deployed software by generating test cases from logs.

In the short term, we plan to make CHRONICLER a more robust approach by addressing its inability to deterministically replay thread interleavings and its lack of privacy control (for example, by investigating possible synergies with [14], [24]). Our future research direction also involves using CHRONICLER to introduce fault tolerance in deployed software (in combination with systems like [49]), and investigating the application of CHRONICLER to mutable replay [34].

Another interesting research topic would be to combine CHRONICLER with other fault reproduction tools so that upon failure an offline and more privacy-preserving approach is attempted to reproduce the bug (such as BugRedux [27]). If the low ever overhead/more private approach fails, then CHRONICLERJ could be automatically employed as an efficient means to generate a trace as input to these tools.

## VIII. ACKNOWLEDGEMENTS

The authors are members of the Programming Systems Laboratory, funded in part by NSF CCF-1161079, NSF CNS-0905246 and NIH U54 CA121852.

## REFERENCES

[1] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[2] Apple Computer. Crash Reporting for iPhone OS Applications. Technical report, Apple Computer, 2009.

[3] S. Artzi, S. Kim, and M. D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 542–565, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] J. Bell, N. Sarda, and G. Kaiser. Programming-Systems-Lab/chroniclerj. https://github.com/Programming-Systems-Lab/chroniclerj.

[5] E. Berk. JLex. http://www.cs.princeton.edu/~appel/modern/java/JLex/.

[6] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 308–318. ACM, 2008.

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190. ACM, Oct. 2006.

[8] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmark suite. http://www.dacapobench.org/benchmarks.html.

[9] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[10] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 319–328. ACM, 2008.

[11] A. Cheung, A. Solar-Lezama, and S. Madden. Partial replay of long-running applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 135–145. ACM, 2011.

[12] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *In Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, 1998.

[13] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *Proc. of the 29th International Conference on Software Engineering (ICSE)*, pages 261–270, 2007.

[14] J. Clause and A. Orso. Camouflage: automated anonymization of field data. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 21–30. ACM, 2011.

[15] O. Crameri, R. Bianchini, and W. Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 199–214. ACM, 2011.

[16] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, Sept. 2004.

[17] M. Dawson, G. Johnson, and A. Low. Best practices for using the Java Native Interface. http://www.ibm.com/developerworks/java/library/j-jni.

[18] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, pages 211–224. ACM, 2002.

[19] Firefox. Breakpad. http://kb.mozillazine.org/Breakpad, 2009.

[20] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 27–27, Berkeley, CA, USA, 2006. USENIX Association.

[21] D. Glasser. amock. http://code.google.com/p/amock/.

[22] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 103–116. ACM, 2009.

[23] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: an application-level kernel for record and replay. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 193–208, Berkeley, CA, USA, 2008. USENIX Association.

[24] J. Huang, P. Liu, and C. Zhang. Leap: lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 385–386. ACM, 2010.

[25] IBM. Ibm trade performance benchmark. https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=trade6.

[26] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. Ocat: object capture-based automated testing. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 159–170. ACM, 2010.

[27] W. Jin and A. Orso. Bugredux: reproducing field failures for in-house debugging. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 474–484, Piscataway, NJ, USA, 2012. IEEE Press.

[28] S. Joshi and A. Orso. Scarpe: A technique and tool for selective capture and replay of program executions. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 234 –243, oct. 2007.

[29] K. Kougios. Java cloning library. http://code.google.com/p/cloning/.

[30] K. H. Lee, Y. Zheng, N. Sumner, and X. Zhang. Toward generating reducible replay logs. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 246–257. ACM, 2011.

[31] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for javascript applications. In *NSDI'10*, pages 159–174, 2010.

[32] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 308–319. ACM, 2006.

[33] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Recording application-level execution for deterministic replay debugging. *IEEE Micro*, 26(1):100–109, Jan. 2006.

[34] V. Nicholas, S. Nair, and J. Nieh. Transparent mutable replay for multicore debugging and patch validation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, March 2013.

[35] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov. Ballerina: automatic generation and clustering of efficient random unit tests for multithreaded code. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 727–737, Piscataway, NJ, USA, 2012. IEEE Press.

[36] M. Odersky. The scala programming language. http://www.scala-lang.org/, 2001.

[37] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 815–816. ACM, 2007.

[38] R. Pozo and B. Miller. SciMark 2.0. http://math.nist.gov/scimark2/, 1999.

[39] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of JavaScript benchmarks. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 677–694. ACM, 2011.

[40] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[41] 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*, ISSTA '00, pages 158–167. ACM, 2000.

[42] The Apache Software Foundation. Apache geronimo v2.0 daytrader benchmark. https://cwiki.apache.org/GMOxDOC20/daytrader.html.

[43] TMate Software. SVNKit. http://svnkit.com, 2004.

[44] Transaction Processing Performance Council. Tpc-c v5. http://www.tpc.org/tpcc/default.asp.

[45] XStream. Xstream 1.4.3. http://xstream.codehaus.org/.

[46] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 122–135. ACM, 2003.

[47] M. Zalewski. tmin: Fuzzing Test Case Optimizer. http://code.google.com/p/tmin/, 2011.

[48] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 321–334. ACM, 2010.

[49] A. Zavou, G. Portokalidis, and A. D. Keromytis. Self-healing multitier architectures using cascading rescue points. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 379–388. ACM, 2012.

[50] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.

[51] S. Zhang. Palus: a hybrid automated test generation tool for java. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1182–1184. ACM, 2011.