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 have too much overhead to practically deploy. We present CHRONICLER, an approach to remote debugging that captures nondeterministic inputs to applications in a lightweight manner, guaranteeing 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 nondeterministically, 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 [17]. 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 [13], [41]. 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 [13], 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) [39] or the breadth of information recorded (e.g. to only record information on a particular subsystem that a developer identifies as potentially buggy) [41]. While these approaches can reduce overhead significantly, to a best case of 1% (with worst cases over 800%), they do not guarantee reproducibility.

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.

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. These systems work well if it is clear to the developers what section of code is most likely to crash, but if they select too large of a subsystem, the performance benefit shrinks.

In this paper, we present CHRONICLER: a technique that supports remote debugging by soundly capturing program execution in a manner that guarantees accurate replay in the lab, with very low overhead. In addition to simple stand-alone applications, CHRONICLER supports accurate and efficient record-replay of execution of client-server applications. CHRONICLER only logs sources of nondeterminism — allowing for a lighter 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 that brought the system to fail, which allows any bug that presents itself during execution to be replayed, regardless of the time between failure and detection. This is a general approach that 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 (1.76% in the case of Eclipse performance tests, 6.62% for Tomcat).

The main contributions of this paper are:

- A presentation of our remote-debugging tool that guarantees bug reproduction: CHRONICLER
- CHRONICLERJ , an implementation of CHRONICLER for Java, available for download and use now on github [14].
- A thorough evaluation of its performance demonstrating its low overhead on real world applications

Our approach relies on an efficient record and replay system that could be be used by the community to further explore areas such as:

- Test suite generation — Existing tools impose high overhead but could be run offline on captured executions [38]
- Process Migration — Existing tools require OS extensions [48] or additional developer effort [30]
- Efficient checkpoint and restart for VM based languages — existing tools [11] are not suited to VM based languages

The record and replay technique used by CHRONICLER can be applied to these lines of research to ameliorate performance burdens that have been holding back greater development.

The rest of the paper is organized as follows. In Section II, we discuss related work in the field of record and replay systems, ubiquitous error reporting and test case generation. We elaborate on the CHRONICLER approach in Section III and present implementation details for CHRONICLERJ n Section IV. Our empirical evaluation of CHRONICLER and a comparison with another 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 [33]. 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 [12] and Firefox's Breakpad [28] 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. The developer must still infer the cause of the crash, a problem that CHRONICLER aims to solve.

More recently, tools have been developed specifically to generate test cases to reproduce errors caught in the field.

BugRedux [39] uses symbolic execution to guide the synthesis of tests that can reproduce failures from four different kinds of execution data; points of failure, call sequences and complete program traces. The runtime overhead imposed by logging call sequences varied from 1% to nearly 50%. However, its ability to reproduce an observed failure is dependent on the completeness of the set of intermediate states which are used to guide the synthesis of the tests. If the states extracted from the execution traces are incomplete then the observed failure may not be reproduced. CHRONICLER has the same end-goal as BugRedux, to generate test cases that reproduce field errors, but uses an approach that can guarantee reproduction.

Bbr [22] is a system that uses symbolic analysis to re-create program runs that are isomorphic in execution paths for long running programs such as databases and web servers. While the states generated by bbr might reproduce the execution path that lead to the error, they are not identical to the original state. With CHRONICLERthe actual state of the system from the point when recording commences is preserved.

ReCrashJ [13] is a Java-based tool that automatically generates test cases when software crashes. ReCrashJ maintains a log of method arguments for the entire call stack, and in the event of a crash, uses the log to create a test. 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; the 42% approach does not maintain copies of the state). While a second recording mode exists that is lightweight in comparison, it requires that a crash be reproduced a second time in the field — which may be unacceptable if the crash leads to a loss of data. ReCrashJ is also limited in applicability to failures that present themselves in deep call-stacks — if the stack depth is too low, the information collected may not be enough to reproduce the same failure.

Scarpe [41] is a bug reproduction tool that requires developers to annotate their application to show component boundaries, and captures 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, systems such as CHRONICLER, ReCrashJ and Scarpe are essentially record and replay systems. Record and replay systems capture program execution and deterministically replay it. Some of the earliest such systems were machine-wide, aimed to debug operating systems [27], [46], [53], [59]. 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 [29] and Mugshot [44] are two application-level record and replay systems very similar to CHRONICLER. Liblog is a tool for C applications to record and replay all interactions between the application and the operating system by providing a libc wrapper. However, this approach is insufficient to capture all sources of nondeterminism 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

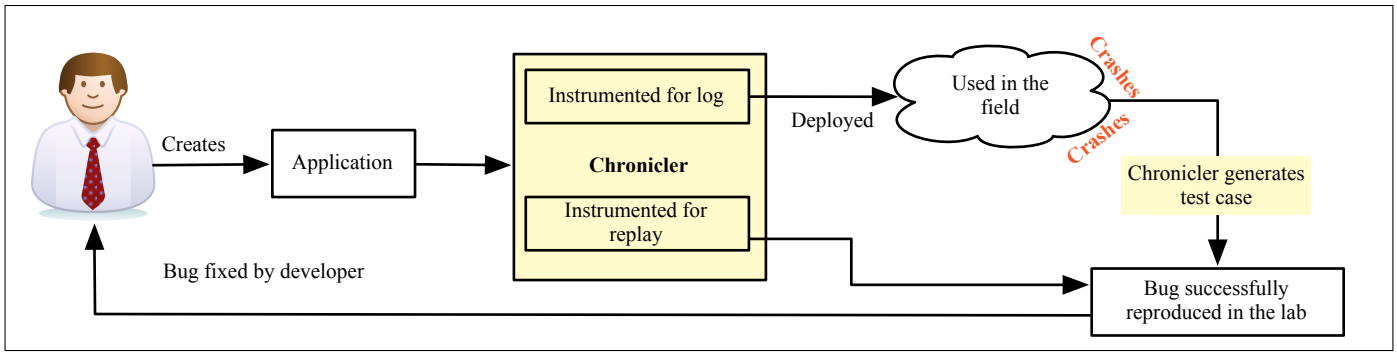

Fig. 1: High Level Overview of CHRONICLER

(concerns that are addressed by a VM). Additionally, liblog does not support logging multithreaded applications, which CHRONICLER supports by maintains a log for each thread. CHRONICLER's approach is fundamentally very similar to liblog in that both systems log nondeterminism, although key to our approach is the mechanism in which we intercept the nondeterministic calls. A direct application of liblog to a VM based language would necessitate modifying the VM or the core language API, whereas our technique involves instrumenting the byte code representation of the program. CHRONICLER takes advantage of the portability of VM based languages, while liblog only runs on systems that use the libc library.

Mugshot [44] is a record and replay system targeting JavaScript applications with the same underlying technique as liblog, using JavaScript reflection to intercept nondeterministic inputs. Mugshot imposes low overhead both in terms of storage and computation (7% for interactive games). CHRONICLER is very similar to both tools in that it too achieves low-overhead record and replay by capturing sources of nondeterminism. However, the principles involved in Mugshot's implementation are not generally applicable to VM based languages, as the interception technique (based on reflection) can be a computationally expensive operation, particularly in the JVM. Mugshot also is designed to function in the limited execution model of browser-based applications (based on non-preemptive callbacks), which is not the case for other languages such as Java, with a rich execution model, full multithreading support and many more sources of non determinism than Javascript.

R2 [34] is an application level record replay system that is similar to liblog but improves upon it in a few ways. While liblog intercepts calls to low level libraries such as libc, R2 requires developers to annotate the application interfaces they wish to replay. This allows developers the flexibility of choosing an interface whose interception will cause low overhead as well as to bypass liblog's limitation of not guaranteeing faithful replay. Conversely, this implies extra work for the developers to manually annotate the APIs they care about while CHRONICLER automatically determines the correct methods to annotate without developer input.

Although several record-replay systems have been described in the literature, only a few target the JVM. DeJaVu [26] was one of the earliest JVM based record and replay systems. However, it required invasive changes to the JVM which limited its potential for widespread adoption.

JaReC [31] and LEAP [36] are record and replay systems for Java applications that specifically target replaying thread interleavings. JaReC suffers from high overhead, ranging from 100-2,490%. LEAP records less information — only partial thread access information, which allows it to display much less overhead, around 10% on Tomcat and Derby (but up to 600% in the worst case, depending on thread accesses). Both only record thread interleavings (and no other inputs) however, and cannot faithfully replay a recording, unlike CHRONICLER.

jRapture [54] 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 [54].

While test case generation tools (e.g. [24], [47], [49], [62]) focus on generating test cases to increase test suite code coverage offline, CHRONICLER generates test cases that specifically reproduce field failures.

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 nondeterminism in a program and replays them to reproduce the bug.

Figure 2 shows the overall approach to logging nondeterminism 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

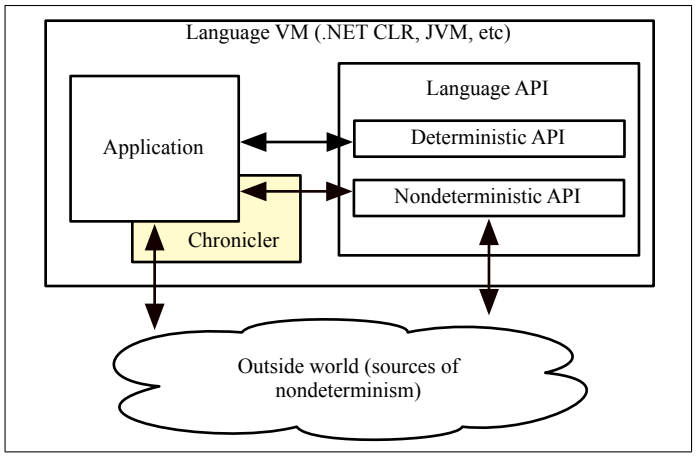

Fig. 2: Logging Nondeterminism with CHRONICLER

methods. CHRONICLER runs completely within the VM, and sits between the application and all sources of nondeterminism, logging them as they enter the application code. Note that although thread interleavings are a source of nondeterminism that may manifest internal to the VM, and 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 [29] and Jockey [52] that record nondeterminism at the granularity of system calls, CHRONICLER records nondeterminism 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.

This approach removes the need to modify any language-provided libraries, and can result in performance gains 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.

Once all nondeterministic method calls are identified, CHRONICLER instruments the application code to log the result of the call. 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.

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, nondeterministic 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.

This approach is *complete*: by logging all sources of nondeterminism, we can guarantee that we can reproduce the same execution, and hence, the same failure. Moreover, this approach will reproduce a failure even if it goes unnoticed for some long time, as long as the log files are retained for the entire period.

CHRONICLER instruments the application to generate a test case and log file to transmit to developers upon encountering an error. Once the developers receive the test case and log, reproduction is simple: CHRONICLER re-instruments the application code for replay rather than recording, and then executes the instrumented application with the log file. To instrument the application for replay, we identify the same methods that we previously logged and replace the method calls with instructions to load the logged input values. Then we begin execution at the same entry point as the original failed execution and play back the log. With this technique we allow developers to observe the entire execution and use existing automated debugging tools that they may already be comfortable with, such as [35], [40], [61].

Note that through the entire CHRONICLER approach, no source code is necessary, and all instrumentation can be performed directly on byte code.

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) or other VMs (e.g. .NET).

IV. IMPLEMENTATION

To further elaborate on the CHRONICLER approach, we provide CHRONICLERJ, our Java implementation of CHRONICLER. Figure 3 shows an overview of the CHRONICLERJ implementation.

We describe its implementation in the following four core components:

A. Detecting Nondeterministic Methods in the JVM

Our approach requires instrumenting the call site of every method in client code that receives nondeterministic input. As we noted previously, within the JVM the only way that code can receive nondeterministic 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 nondeterministic methods in the Java API is to scan the entire API, and mark all methods that are "native" as nondeterministic. However, not all native methods are nondeterministic. For instance, the typical

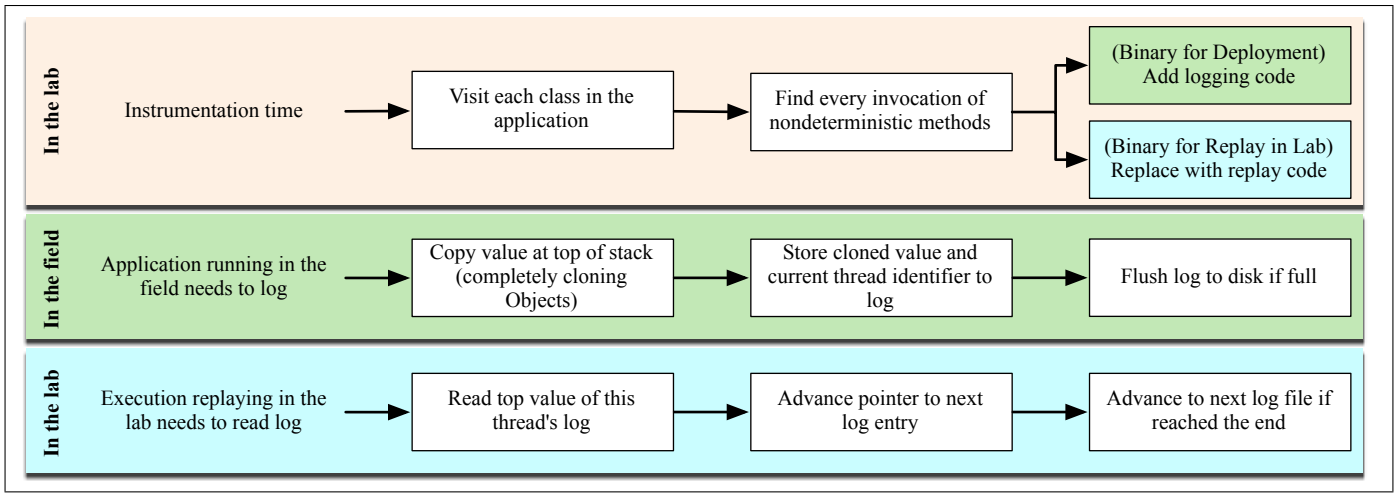

Fig. 3: CHRONICLERJ Implementation Overview

approach to copy the contents of an array is to use a native call System.arraycopy(Object source, int sourceOffset, Object dest, int destOffset, int length). While an array-copy could be implemented in pure Java, the native approach is far more efficient, as it results in directly copying the contents of the entire array (stored contiguously in memory) rather than an entry-by-entry approach. 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 nondeterministic, sacrificing performance for correctness, rather than risking an incomplete log.

The next step in identifying all nondeterministic methods in the Java API requires identifying all API methods that call the previously identified nondeterministic methods. This process scans the API for all callers of nondeterministic methods and recursively marks those methods as well as their callers as nondeterministic. CHRONICLERJ also carries nondeterministic flags up the inheritance hierarchy — this will, for instance result in the interface method InputStream.read(byte[], int, int) to be marked as nondeterministic, since many of its implementers are. Since Java exercises dynamic binding, it may be impossible to know statically exactly which implementation of a method will be invoked at run time, and therefore, we err on the side of caution.

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

This process of carrying flags through the hierarchy repeats until the list of nondeterministic methods is stable. 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 we guarantee detection of all methods that behave nondeterministically. 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.

B. Logging

CHRONICLERJ instruments all calls to the identified nondeterministic methods to record return values (and buffer(s), if applicable). All byte code instrumentation is performed using the ASM byte code framework [20]. This 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. CHRONICLERJ is thread-safe, and protects each log call with a barrier so that no two threads can log at the same time.

Logging code is embedded inline, just after the value that we need to log (a return value or parameter) is pushed onto the stack. The instrumentation copies the object, grows the log if necessary, writes the object to the log, writes the current thread to the log and flushes the log if necessary. For completeness, we also record events dispatched nondeterministically by the language API (e.g. Swing ActionListeners) so that the replaying application can fire them at the appropriate time.

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 [42] 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, and only generates log instructions after a nondeterministic 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.

At the same time that this instrumentation is performed, a "replay" version of the application is created, which replaces nondeterministic 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.

C. Flushing the log

By default, CHRONICLERJ flushes the log 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 if enough processors are available, program execution can continue during the flushing process.

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 [58]. 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. To perform this replay, CHRONICLERJ re-instruments the application to replace the code that originally generated the log, with code to replay the log. In this way, CHRONICLERJ removes nondeterministic method calls, replacing them with calls 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 generated test case contains all necessary log files (as the log may have been flushed more than once before the failure), 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.

In our evaluation that follows, we show that the logging overhead of CHRONICLERJ is reasonable for several applications. Our figures indicate that CHRONICLERJ is very lightweight compared to ReCrashJ, the only previous solution for Java that we were able to obtain to compare to directly.

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?

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 [51]
luinden	Indexes the Shakespeare and the King James Bible with Apache Lucene
lusearch	Searches for keywords over a corpuse including Shakespeare and the King James Bible with Apache Lucene
pmd	Performs a static analysis on Java source files
sun flow	Renders images with ray tracing
tomcat	Creates a tomcat server and runs a simple sample servlet
trade beans	Executes the DayTrader [55] benchmark via Java Beans on an Apache Geronimo server with an in memory database
tradesoap	Executes the DayTrader [55] benchmark via SOAP on an Apache Geronimo server with an in memory database
xalan	Transforms several XML files into HTML

TABLE I
DESCRIPTION OF BENCHMARKS, FROM THE DACAPO WEBSITE [19]

Benchmark	CHRONICLERJ Overhead	ReCrashJ Overhead
avrora	1.56%	2,321.94%
batik	6.59%	5.97%
eclipse	1.76%	N/A
fop	39.96%	130.11%
h2	6.46%	669.56%
jython	15.45%	481.11%
luindex	4.66%	312.90%
lusearch	36.60%	951.27%
pmd	10.04%	90.93%
sunflow	1.96%	N/A
tomcat	6.62%	9.06%
tradebeans	3.11%	8.15%
tradesoap	5.31%	21.13%
xalan	26.59%	780.25%

TABLE II
BENCHMARK PERFORMANCE FOR CHRONICLERJ AND RECRASHJ AGAINST DACAPO

A. EM1 - Performance Overhead

We used the DaCapo v9.12-bach suite of benchmarks [18], a set of Java benchmarks that focus on exercising applications in real-world conditions to evaluate CHRONICLERJ's performance for **EM1**. We also evaluate CHRONICLERJ's performance overhead on the same set of benchmarks used by the Java bug reproduction system, ReCrashJ [13]. 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, 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

Benchmark	CHRONICLERJ Overhead	ReCrashJ Overhead	
		This study	From [13]
SVNKit Checkout	68.42%	TBD	38.00%
Eclipsec Channel	9.95%	TBD	34.00%
Eclipsec Content	11.62%	TBD	13.00%
Ecipsec String	10.87%	TBD	27.00%
Eclipsec Jlex	7.18%	TBD	42.00%
SVNKit Update	TBD%	TBD	TBD%

TABLE III
BENCHMARK PERFORMANCE FOR CHRONICLERJ AND RECRASHJ AGAINST RECRASHJ'S BENCHMARK SUITE

are implementations of well known and accepted workloads. For instance, the "h2" benchmark utilizes the TPC-C workload [57], a common database benchmarking workload, to test an in-memory database. The "tradebeans" and "tradesoap" benchmarks run Apache's DayTrader [55] benchmark workload, an open source version of IBM's Trade 6 workload [37]. A complete list of the individual benchmarks executed along with a brief description appears in Table I.

We executed all 14 benchmarks in the DaCapo suite 100 times each on both the unmodified benchmark and the CHRONICLERJ-instrumented benchmark. 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 [13], version 0.3.

During our preliminary testing, we found that in some instances, the CHRONICLERJ-instrumented version of the benchmark ran faster than the uninstrumented version. This seemed counter-intuitive, so we investigated and found that the benchmarks that ran faster with CHRONICLERJ (xalan, tradebeans, tradesoap and sunflow) were all multithreaded. In its default configuration, DaCapo uses as many worker threads as there are cores on the machine (four, in our case). However, we believe that several of the benchmarks (most notably, xalan tradebeans tradesoap and xalan) were exhibiting significant amounts of thread contention. By adding CHRONICLERJ, which adds a lock around each log statement (no two threads can write to the log at once), we believe that we actually reduced the contention, hence increasing the performance. We experimented with running the non-instrumented version of DaCapo and found optimal performance on our test machine (for the non-instrumented DaCapo) when limiting the number of threads to two. Figure 4 represent the average time per benchmark of the DaCapo suite limiting the external concurrency to only two threads. Table II shows the results of our study in terms of overhead in percent increase of time per benchmark. Note that ReCrashJ failed to properly instrument the eclipse and sunflow benchmarks — for these cases the instrumentation caused a runtime crash in both cases, and we therefore do not have results for ReCrashJ for those benchmarks.

2) Recrash 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 [13]:

- Using SVNKit 0.8.0 (an SVN client implemented entirely

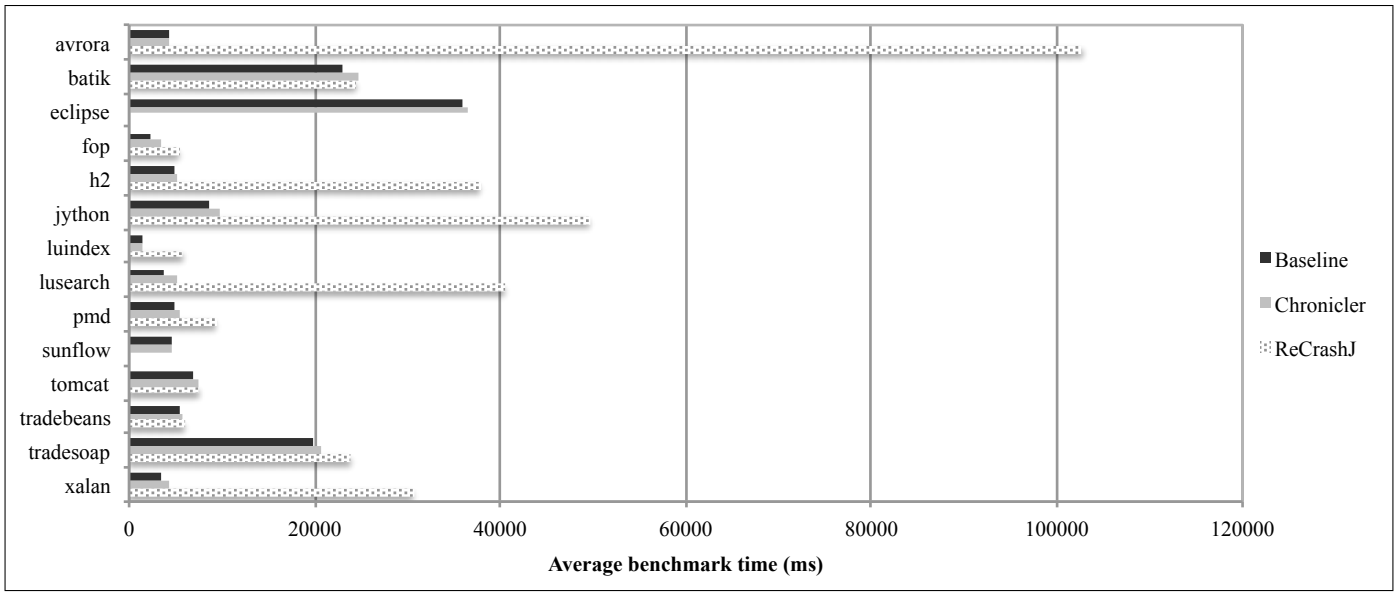

Fig. 4: DaCapo Benchmark Results for a baseline execution, CHRONICLERJ and ReCrashJ

	Original			ChroniclerJ				ReCrashJ			
	Average	Standard Dev.	Median	Average	Standard Dev.	Median	Overhead	Average	Standard Dev.	Median	Overhead
avrora	4241.6	103.2	4,231.0	4,308	201	4,285	1.56%	102,729	5,624	106,099	2,321.94%
batik	23129.4	3,297.6	22,420.5	24,653	4,111	24,025	6.59%	24,510	3,628	24,052	5.97%
eclipse	35867.3	1,067.2	35,788.0	36,498	945	36,432	1.76%	0	0	0	-100.00%
fop	2370.8	56.0	2,346.0	3,318	108	3,269	39.96%	5,456	131	5,450	130.11%
h2	4916	338.8	4,802.5	5,234	491	5,101	6.46%	37,832	295	37,802	669.56%
jython	8539.5	324.8	8,471.5	9,859	1,034	9,505	15.45%	49,624	1,269	49,678	481.11%
luindex	1414.2	162.1	1,383.5	1,480	148	1,465	4.66%	5,839	506	5,927	312.90%
lusearch	3852	626.8	3,878.5	5,262	640	5,462	36.60%	40,495	860	40,556	951.27%
pmd	4909.5	190.8	4,894.5	5,402	210	5,405	10.04%	9,374	459	9,493	90.93%
sunflow	4479.1	98.0	4,458.0	4,567	96	4,483	1.96%	0	0	0	-100.00%
tomcat	6969	1,032.7	6,815.0	7,431	971	7,133	6.62%	7,600	2,778	7,165	9.06%
tradebeans	5581.7	249.2	5,528.5	5,755	233	5,749	3.11%	6,036	170	6,024	8.15%
tradesoap	19753.1	414.8	19,872.0	20,802	3,853	19,830	5.31%	23,927	10,152	20,064	21.13%
xalan	3470.1	177.7	3,425.5	4,404	114	4,384	26.90%	30,622	6,984	26,315	782.46%

TABLE IV
BENCHMARKED PERFORMANCE OF CHRONICLERJ ON 2 CORES, ALL TIMES IN MILLISECONDS

in Java) [56], checkout and update the project "amock" on GoogleCode [32]
- 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 [16]

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 [13]. We attribute the differences in overhead between our experiment and [13] to the architectural differences between the test systems.

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 [50] 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 only calls nondeterministic functions to build the test data: there is no nondeterminism within the benchmark itself (and hence, it was not instrumented in any way). 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 from 2Mb all the way up 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

	Original			ChroniclerJ				ReCrashJ			
	Average	Standard Dev.	Median	Average	Standard Dev.	Median	Overhead	Average	Standard Dev.	Median	Overhead
avrora	4279.5	101.9	4,284.5	4,288	164	4,269	0.20%	97,543	6,748	96,445	2,179.29%
batik	22364.3	2,877.8	21,583.5	24,217	4,559	23,728	8.29%	28,758	4,979	28,129	28.59%
eclipse	35521.2	1,019.5	35,373.0	35,906	946	35,794	1.08%	0	0	0	-100.00%
fop	2357.5	42.3	2,346.5	3,266	181	3,210	38.52%	5,488	270	5,476	132.77%
h2	5260.3	410.0	5,176.5	6,299	161	6,288	19.75%	37,111	569	37,213	605.49%
jython	8433.7	302.9	8,322.5	8,924	355	8,978	5.82%	51,503	2,312	51,247	510.68%
luindex	1442.1	168.8	1,446.0	1,533	186	1,524	6.31%	6,134	625	6,068	325.38%
lusearch	1441.4	315.9	1,351.5	2,965	137	2,958	105.68%	37,200	1,806	37,010	2,480.91%
pmd	4981.3	178.8	4,945.5	5,094	187	5,070	2.27%	11,162	605	11,171	124.08%
sunflow	3251.7	108.7	3,237.0	3,218	123	3,199	-1.05%	0	0	0	-100.00%
tomcat	5333.2	1,000.6	5,189.0	5,747	804	5,618	7.76%	0	0	0	-100.00%
tradebeans	6049.9	213.3	6,119.0	5,978	185	6,016	-1.19%	0	0	0	-100.00%
tradesoap	17135.8	510.4	17,181.5	16,560	703	16,685	-3.36%	0	0	0	-100.00%
xalan	4259.4	188.3	4,306.0	4,048	175	4,058	-4.95%	40,660	982	40,671	854.59%

TABLE V
BENCHMARKED PERFORMANCE OF CHRONICLERJ ON 4 CORES, ALL TIMES IN MILLISECONDS

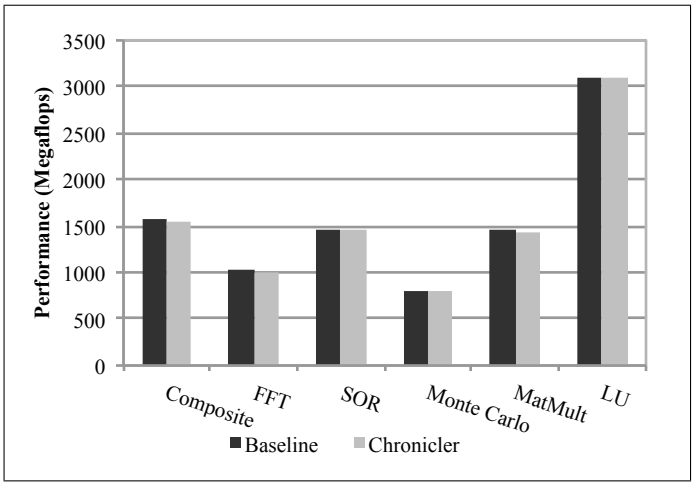

Fig. 5: Scimark Benchmark Performance for CHRONICLERJ

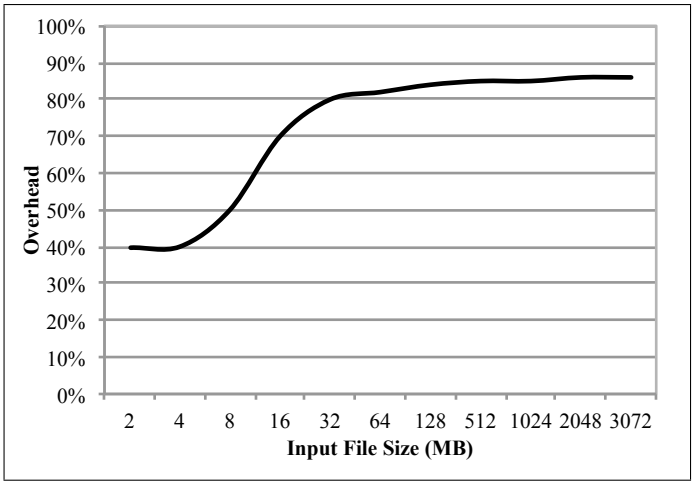

Fig. 6: I/O Benchmark Performance for CHRONICLERJ

increased the file size, the overhead evened out at 86% as the file size grew, providing an 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 most cases CHRONICLERJ outperformed ReCrashJ, with the exception of the batik benchmark and the SVNKit benchmark. We believe that this was caused by an amiable environment for ReCrashJ (e.g. low stack depths), while the I/O components of batik (reading and writing images) and SVNKit (reading files from the network and writing them to disk) slowed down 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%).

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 (bug #363993) The exception thrown was an EOFException which was caused because of a lack of infrastructure in the HttpTester to expect a body in an HTTP response. The full stacktrace that was thrown is

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

Benchmark	Composite	Fast Fourier Transform	SOR	Monte Carlo	Matrix Multiplication	LU Decomposition
Baseline	1564.10376	1025.23461	1465.01133	796.65661	1448.65638	3084.95988
Chronicler	1560.45056	1013.53144	1460.76157	795.14972	1444.79085	3088.01921
Overhead	0.23%	1.15%	0.29%	0.19%	0.27%	-0.10%

TABLE VI
BENCHMARK PERFORMANCE FOR CHRONICLERJ AGAINST SCIMARK

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.
  The first bug [1] 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 me 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 [2] 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 [3] 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 [4] 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.
  The first bug [5] 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 [6] 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.
  The first bug [7] 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 [8] 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 [9] 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 [10] 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 {
```

```
12    number -> assert number > 0
13    return 1
14 }
15
16 someStub.use {
17    assert new SomeClass().methodOne(2) == 1
18 }
```

CHRONICLERJ was able to faithfully reproduce the executions, in each case. All programs terminated with the same uncaught exception that CHRONICLERJ had captured earlier. A complete list and description of each of the bugs that we reproduced are omitted for brevity, but appear in the accompanying technical report [15].

VI. THREATS TO VALIDITY AND LIMITATIONS

We performed our experiments towards evaluating **EM1** on the DaCapo benchmarks, which we 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 the worst case, it is possible that there is some other use case that would stress it further.

For **EM2**, the key threat to validity is the sample size. 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. Nonetheless, we believe that given the approach we have taken, which we are confident is capable of reproducing any (non-race) bug, this 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 [14].

There are several limitations to our approach and implementation. Current tools that reproduce races in Java applications have worst-case overheads of over 500% [31], [36] — therefore we have omitted this functionality. However, traditional race-detection techniques (e.g. [45]) could be used within the lab on replayed executions to detect possible races, without adding any 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 way of solving this problem is through input minimization [60], [61], 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 [23] addresses this issue by mutating a failure inducing input in such a manner that although the original and mutated version share no sensitive data, the program execution paths that they actuate are identical. Castro, et al [21] had earlier used symbolic execution in conjunction with record replay techniques in order to anonymize sensitive data present in bug reports.

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. In other cases, we may be able to combine novel approaches to reducing the log size [22], [43] along with compression techniques to decrease log sizes.

Detecting and logging all nondeterministic methods is key to the CHRONICLER approach — which can only be applied to languages that operate in a virtual machine (such as Java or .NET). One potential failure point is if the set of nondeterministic 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 nondeterministically mutates its parameters (since CHRONICLER does not generally log parameters passed to nondeterministic methods, only return values and buffer parameters). 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. We manually verified that none of these methods were specified to modify their parameters. This process would need to be performed for implementations of CHRONICLER for other languages to make sure that this assumption holds. Additionally, a developer may still create their own native methods that nondeterministically mutates its parameters, and this information would not be captured by CHRONICLERJ. However, this technique is discouraged and rarely used (in the case of Java), as it is inefficient to access object parameters in native code [25].

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 nondeterministic conditions. We presented the sound approach used by CHRONICLER to guarantee bug reproduction: logging nondeterministic inputs at a layer *above* the language API. We evaluated our approach by creating CHRONICLERJ in Java and simulating real world workloads. While state-of-the-art bug reproduction systems can have high overhead (with worst-case scenarios over 2,000%) or fail to guarantee to reproduce defects, CHRONICLERJ has a worst-case overhead of only 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 sophisticated privacy control. Our future research direction involves developing tools that leverage CHRONICLER to introduce fault tolerance in deployed software. Another interesting research topic would be to combine the traces produced by CHRONICLER and techniques such as symbolic execution or model checking in order to automatically produce test inputs that can highlight hidden bugs. These approaches will serve to complement existing static analysis tools.

VIII. ACKNOWLEDGEMENTS

The authors are members of the Programming Systems Laboratory, funded in part by NSF CCF-1161079, NSF CNS-0905246 and NIH 2 U54 CA121852-06.

REFERENCES

[1] https://issues.apache.org/jira/browse/MATH-801.
[2] https://issues.apache.org/jira/browse/MATH-790.
[3] https://issues.apache.org/jira/browse/MATH-803.
[4] https://issues.apache.org/jira/browse/MATH-645.
[5] https://issues.apache.org/jira/browse/LANG-72.
[6] https://issues.apache.org/jira/browse/LANG-300.
[7] http://jira.codehaus.org/browse/GROOVY-5649.
[8] http://jira.codehaus.org/browse/GROOVY-3914.
[9] http://jira.codehaus.org/browse/GROOVY-2503.
[10] http://jira.codehaus.org/browse/GROOVY-2256.
[11] 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.
[12] Apple Computer. Crash Reporting for iPhone OS Applications. Technical report, Apple Computer, 2009.
[13] 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.
[14] J. Bell, N. Sarda, and G. Kaiser. Programming-Systems-Lab/chroniclerj. https://github.com/Programming-Systems-Lab/chroniclerj.
[15] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight recording of nondeterministic inputs to reproduce field failures. Technical Report CUCS-012-12, Department of Computer Science, Columbia University, New York, New York, August 2012.
[16] E. Berk. JLex. http://www.cs.princeton.edu/~appel/modern/java/JLex/.
[17] 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, New York, NY, USA, 2008. ACM.
[18] 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, New York, NY, USA, Oct. 2006. ACM Press.
[19] 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.
[20] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.
[21] 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, New York, NY, USA, 2008. ACM.
[22] 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, New York, NY, USA, 2011. ACM.
[23] 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, New York, NY, USA, 2011. ACM.
[24] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, Sept. 2004.
[25] M. Dawson, G. Johnson, and A. Low. Best practices for using the java native interface. http://www.ibm.com/developerworks/java/library/j-jni/#global.
[26] J. deok 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.
[27] 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, New York, NY, USA, 2002. ACM.
[28] Firefox. Breakpad. http://kb.mozillazine.org/Breakpad.
[29] 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.
[30] J. Gehweiler and M. Thies. Thread migration and checkpointing in java. Technical Report tr-ri-10-315, Heinz Nixdorf Institut, June 2010.
[31] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. Jarec: a portable record/replay environment for multi-threaded java applications. *Softw. Pract. Exper.*, 34(6):523–547, May 2004.
[32] D. Glasser. amock. http://code.google.com/p/amock/.
[33] 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, New York, NY, USA, 2009. ACM.
[34] 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.
[35] D. Hao, T. Xie, L. Zhang, X. Wang, J. Sun, and H. Mei. Test input reduction for result inspection to facilitate fault localization. *Automated Software Engg.*, 17(1):5–31, Mar. 2010.
[36] 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, New York, NY, USA, 2010. ACM.
[37] IBM. Ibm trade performance benchmark. https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=trade6.
[38] 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, New York, NY, USA, 2010. ACM.
[39] 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.
[40] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.
[41] 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.
[42] K. Kougios. Java cloning library. http://code.google.com/p/cloning/.

[43] 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, New York, NY, USA, 2011. ACM.

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

[45] 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, New York, NY, USA, 2006. ACM.

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

[47] 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.

[48] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36(SI):361–376, Dec. 2002.

[49] 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, New York, NY, USA, 2007. ACM.

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

[51] Python Software Foundation. Jython: Python for the java platform. http://www.jython.org/.

[52] Y. Saito. Jockey: A user-space library for record-replay debugging. In *In AADEBUG05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 69–76. ACM Press, 2005.

[53] 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.

[54] 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, New York, NY, USA, 2000. ACM.

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

[56] TMate Software. SVNKit. http://svnkit.com.

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

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

[59] 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, New York, NY, USA, 2003. ACM.

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

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

[62] 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, New York, NY, USA, 2011. ACM.