Chronicler

Lightweight Recording to Reproduce Field Failures

Jonathan Bell, Nikhil Sarda, Gail Kaiser

Department of Computer Science
Columbia University in the City of New York
New York City, NY 10027

jbell@cs.columbia.edu, ns2847@columbia.edu,
gail@cs.columbia.edu

September x, 2012

Motivation

Jonathan Bell,
Nikhil Sarda,
Gail Kaiser

- Software errors are costly!
- Traditional testing strategies are useful but inadequate
  - Complex software has a large state space.
- Errors occurring in the field are hard to debug
  - They occur under a very specific set of circumstances.
  - They may involve multiple systems.
- Writing good bug reports takes skill
  - Bug reports for FOSS projects require steps to replicate and/or a test case. Cannot expect the average user to make this kind of an effort.
  - Alternative is to use automatic error reporting tools, but these do not provide any insight into the error (symptom vs underlying cause)

With Chronicler, we aim to make errors in the field easier to capture and debug without requiring any effort from the end user.

But before we proceed ...

A quick tour of modern software testing strategies

- Test case generation
- In-vivo testing
- Metamorphic testing
- Bug finding

Test case generation

Software engineers typically write unit tests as sanity checks for their code. These tools supplement these efforts

- Randoop: Feedback directed random testing
- Palus: Combined static and dynamic test generation
- Ballerina: Test generation to uncover races

RANDOOP

Jonathan Bell,
Nikhil Sarda,
Gail Kaiser

Feedback directed random testing

- Generates random input that conforms with the program input space
- Randomized creation of new test input depends on feedback from previous tests

How do we evaluate such a tool?

- Improve coverage
- Find new bugs!

RANDOOP contd.

Program in, test suite out

- Input
  - Class under test (CUT)
  - Time limit
  - Set of contracts
    - Method contracts (e.g. a call to hashcode() will not result in an exception)
    - Object invariants (e.g. o.equals(o) will return true)
- Output: Contract violating test cases

RANDOOP contd.

Randoop generates test cases that disclose bugs
But what is the secret ingredient?

- Generates random input that conforms with the program input space
- Randomized creation of new test input depends on feedback from previous tests

How do we evaluate such a tool?

- Improve coverage
- Find new bugs!

How is it done?

- Build test inputs incrementally
  - What is a test input? A sequence of valid method calls for a class under test (CUT)
  - New sequences will depend on older ones
- Execute them as they are created
- Use feedback to guide generation

The idea is to keep sequences that work while discarding those that don't. For valid sequences, augment them with randomly generated sequences.

Evaluation of RANDOOP

Subjects:

- JDK 2 library (53k loc, 272 classes)
  - Test cases input: 32, Error revealing cases: 29, distinct errors: 8
- Apache Commons (114k loc, 974 classes over 5 libraries)
  - Test cases input: 187, Error revealing cases: 29, distinct errors: 6
- .Net framework (582k loc, 3330 classes)
  - Test cases input: 192, Error revealing cases: 192, distinct errors: 192

Ballerina

Random testing to detect concurrency bugs
Central thesis: Most concurrency bugs occur in the presence of two threads

- Generates randomized sequences of methods similar to RANDOOP
- Adds additional threading code to trigger concurrency bugs

How do we evaluate such a tool?

- Find new bugs!

Coverage as a metric is irrelevant here as it relates to testing all possible paths that a program can take *sequentially*.

How is it done?

Building the parallel prefix

- Select two methods with the most number of parameters
- Select sequential sequences to plug in after the methods
- Create parameters which will be consumed by these sequences

The idea is to create multithreaded code that acts on some objects in a randomized manner in the hope that a concurrency bug will manifest.

Sounds good, but...

One major problem with this approach is a lot of false positives
Ballerina deals with this problem using handwav...err...statistics!

- Select some key characteristics of the error (exception thrown and method executed)
- Cluster similar errors
- Use randomized sampling to select error reports to analyze

Using Ballerina, three previously unknown bugs were found in Apache Log4J and Apache Pool.

In-vivo testing: Testing programs in the field

The idea is to test actively deployed programs in an unobtrusive manner

- One way is to instrument programs so that a method execution results in a forked JVM that executes the corresponding unit-test
- Distribute tests to different machines or cores in order to improve performance
- Hash previously seen application states so that we only test new states
- Current research focus is on applying invivo testing to security issues

Several open research questions:

- Can we offload tests other systems? (GPUs for instance)
- Classification of defects that can be detected by invivo tests
- Test sandboxing

Metamorphic testing: Testing in the absence of oracles

Jonathan Bell, Nikhil Sarda, Gail Kaiser

The idea is that even in the absence of testing oracles, we can always test for invariants
Many applications reflect metamorphic properties that define a relationship between pairs of inputs and outputs

- Corduroy: A tool that allows developers to specify metamorphic properties using JML
- Amsterdam: Checks metamorphic properties at runtime using execution traces

Current research focuses on automatically extracting metamorphic properties from programs.

Bug finding

Modern tools for bug finding pioneered by Dawson Engler's group at Stanford U.
Coverity was founded by Engler's students and makes use of several ideas developed there
The most important one is belief analysis

- Key idea: Programmer beliefs are reflected in source code
- Checkers extract beliefs using templates. A simple one is that ¡a¿ must accompany ¡b¿. Beliefs can also be extracted by observing code.
- Beliefs may be of two types: MUST and MAY
- MUST beliefs are propagated for internal consistency
- MAY beliefs are treated as MUST beliefs at first. Statistical analysis is used to separate errors from coincidences.

Other tools developed there include KLEE (symbolic execution engine) and eXplode (system specific model checking).

Back to Chronicler

Chronicler is an approach for in-vivo test case generation. Some related work

- RecrashJ
- Scarpe
- BugRedux

As we will see, each of these tools have weaknesses that Chronicler seeks to address.

Some existing tools are inadequate

RecrashJ

- Key ideas
  - Monitors a running JVM application and writes out a test case in the event of an uncaught exception.
  - The principal idea is to record the parameters of each method call. When generating the test case, use these recorded parameters in that case.
- Weaknesses
  - Extremely slow, 20x overhead in the worst case.
  - Things get especially bad with deep call stacks.
  - Does not work with some newer software.

Some existing tools are inadequate

Scarpe

- Key ideas
  - Key idea is to record only a partial execution by isolating a subsystem and capturing all information flowing in and out of it.
  - When attempting to replay a bug in that subsystem, replay those flows.
- Weaknesses
  - Again, very slow. 10x overhead for some applications.
  - Not publicly available.
  - Very weak evaluation.

Some existing tools are inadequate

BugRedux: Recreating failure conditions from crash data

- Key ideas
  - Log specific execution data and use symbolic execution to guide generation of tests
  - Valid execution data; points of failure, call sequences, execution trace.
- Weaknesses
  - Ability to reproduce a failure accurately depends on completeness of set of intermediate states logged
  - Because it uses symbolic execution, it is susceptible to path explosion

But performance is reasonable and promising: 94% of bugs observed were recreated (for Chronicler the figure is 100%)

The Chronicler approach

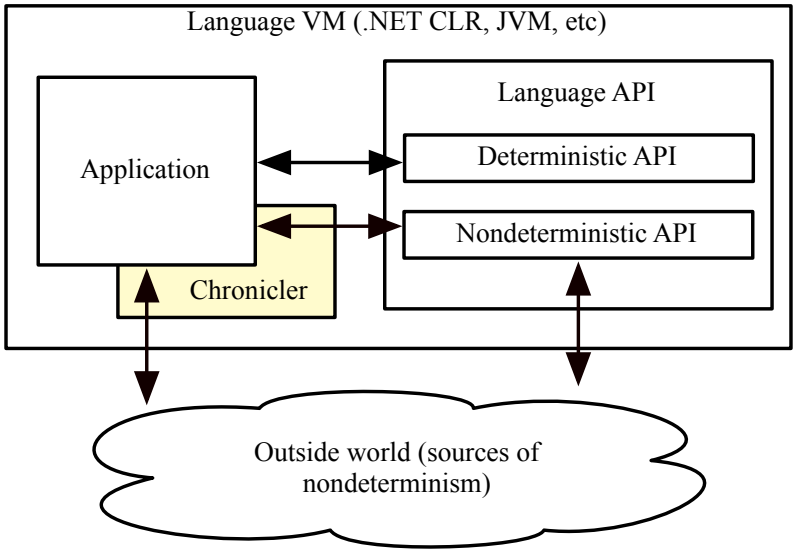
Key idea: Capture sources of non determinism by logging at a layer above the Java API and replay those sources

Definition (Non-determinism)

Dependence on factors other than initial state and input.

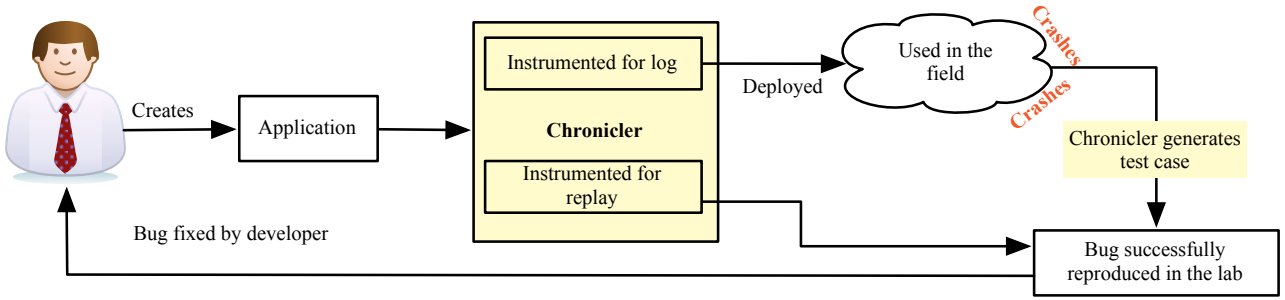
What do we need to look out for?

- All sources of user input (file.read(), buf.readLine())
- Methods that invoke native calls (System.currentTimeMillis())



Using Chronicler in the field

Figure: Workflow illustrating how Chronicler could be deployed in the field



Implementation Details

API scanning

- Mark all native methods as non deterministic
- Recursively mark callers of above methods as ND
- Non-determinism is propagated up the inheritance heirarchy

Creating the recorder and replayer

- Instrument bytecode to log results of ND method call
- In the replayer, replace invocations to method call with logged results
- Special case for event driven systems (Swing)

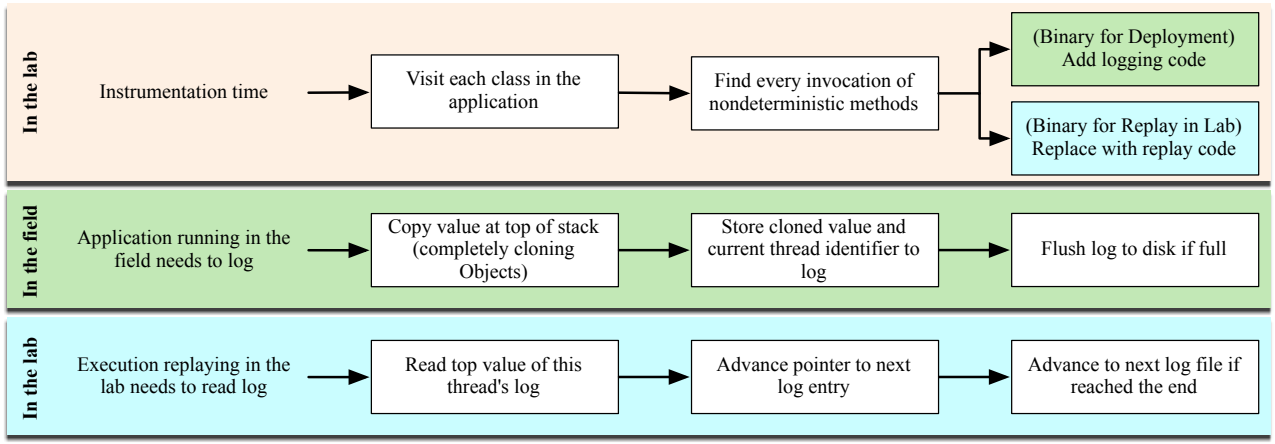
Implementation Details

- Logging code is embedded inline into the bytecode representation of the program
- We also record events dispatched nondeterministically
- Log is flushed after it is large enough
- Uncaught exceptions are handled using a global exception handler which writes out a test case

Logging is thread-safe and write protected using a barrier

Implementation Details

Figure: A diagram illustrating Chronicler's implementation strategy



Example

Note that we do not show object creation

Listing˜1: Bytecode to create a buffer and read a file into it

```
SIPUSH 5000
NEWARRAY T_CHAR
ASTORE 3
ALOAD 2: r
ALOAD 3: buf
ICONST_0
ALOAD 3: buf
ARRAYLENGTH
INVOKEVIRTUAL BufferedReader.read(char[], int,
     int) : int <- Non deterministic method!
POP
```

Example contd. : Recorder

Listing˜2: Bytecode modified for recorder

```
SIPUSH 5000
NEWARRAY T_CHAR
ASTORE 3
ALOAD 2: r
ALOAD 3: buf
ICONST_0
ALOAD 3: buf
ARRAYLENGTH
INVOKEVIRTUAL BufferedReader.read(char[], int,
    int)
// Special case here, BufferedReader returns
   an int and modifies char, in our case buf
// Create a copy of the topmost value on stack
    and store it in the log
// Create a copy of buf and store it in our
   log
POP
```

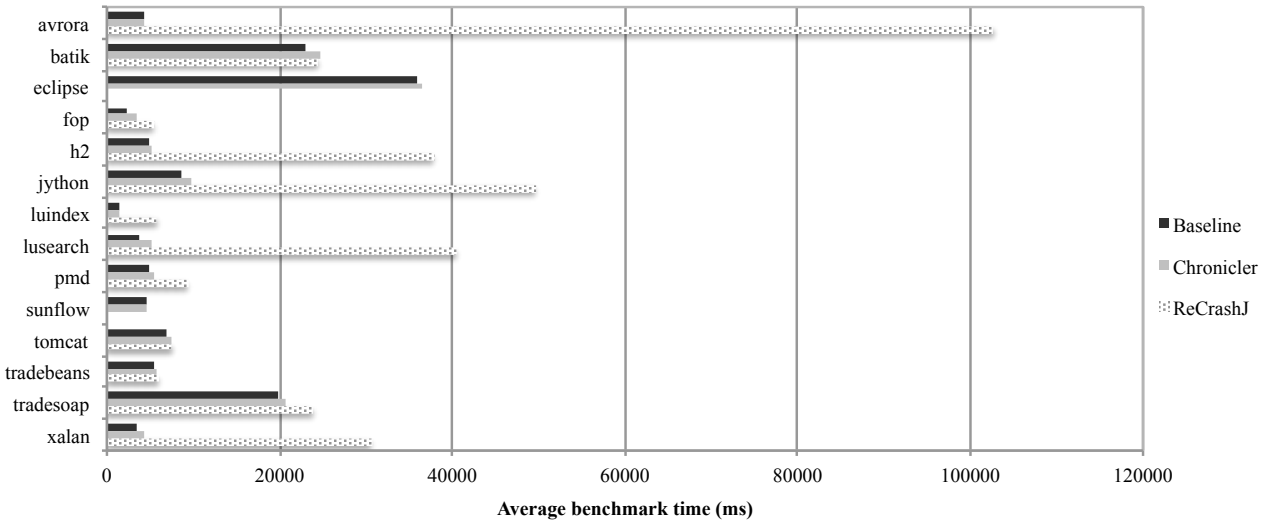
Example contd. : Replayer

Listing˜3: Bytecode for the replayer

```
SIPUSH 5000
NEWARRAY T_CHAR
ASTORE 3
ALOAD 2: r
ALOAD 3: buf
ICONST_0
ALOAD 3: buf
ARRAYLENGTH
// Ignore this call: INVOKEVIRTUAL
    BufferedReader.read(char[], int, int)
// Retrieve the return value of this call from
     the log and push it onto the stack
// Retrieve the copy of buf in the log and use
     System.ArrayCopy to copy it onto buf
POP
```

Performance: Dacapo

Dacapo is a suite of Java benchmarks presented in OOPSLA 2006 and is designed to stress test JVMs.
It consists of several workloads of varying nature, from a python interpreter written in Java to an IDE.

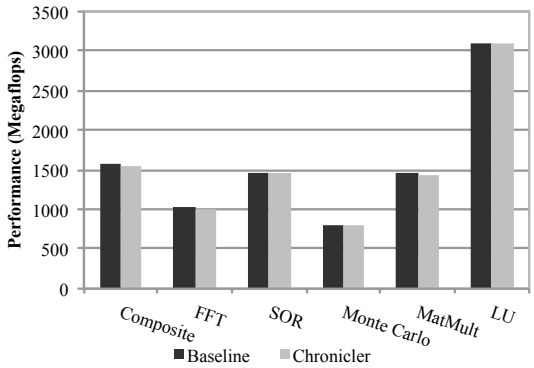


We hypothesize that since our performance on Dacapo is reasonable, Chronicler is well suited for running in the field.

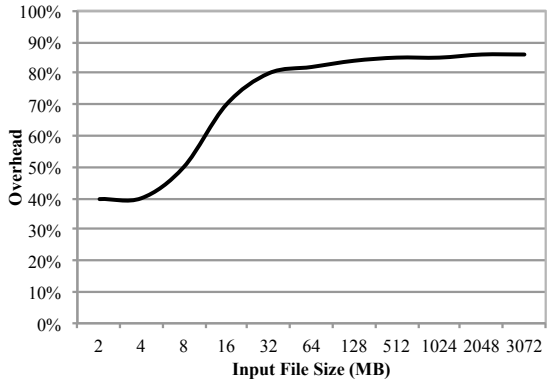
Performance: Targeted

Under what circumstances does Chronicler display its best performance? Worst?

Scimark performance



IO Performance



Weaknesses

- Thread interleavings are not recorded
- Privacy concerns have not been addressed
- Native methods that mutate their parameters

Related Work

Application level record-replay systems

- Liblog
- R2
- Mugshot

OS or VM level record-replay systems

- Zap
- DeJaVu

Conclusions

- Capture sources of non determinism by logging at a layer above the language API
- Replay those sources in order to reproduce bugs
- Solid performance numbers, worst case is upper bounded
- Chronicler can replay all non-race bugs

Future directions

- Checkpoint-restart using the record replay framework
- Thread migration

Future directions

- Checkpoint-restart using the record replay framework
- Thread migration
- Developing a warp drive...

Questions